파이썬으로 배우는

알고리즘 트레이딩

내 손으로 만드는 자동 주식 거래 시스템

파이썬으로 배우는

알고리즘 트레이딩

내 손으로 만드는 자동 주식 거래 시스템

지은이 **조대표**

펴낸이 **박찬규** 엮은이 **이대엽** 디자인 **북누리** 표지디자인 Arowa & Arowana

펴낸곳 **위키북스** 전화 031-955-3658, 3659 팩스 031-955-3660

주소 **경기도 파주시 문발로 115, 311호(파주출판도시, 세종출판벤처타운)**

가격 **40,000** 페이지 **728** 책규격 **188 x 240mm**

초판 발행 **2017년 05월** 17일

2쇄 발행 **2017년 08월** 24일

3쇄 발행 **2018년 03월** 19일

ISBN **979-11-5839-063-1 (93000)**

등록번호 **제406-2006-000036호** 등록일자 **2006년 05월 19일**

홈페이지 **wikibook.co.kr** 전자우편 **wikibook@wikibook.co.kr**

이 책의 한국어판 저작권은 저작권자와 독점 계약으로 위키북스가 소유합니다.

신저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단 전재와 복제를 금합니다.

이 책의 내용에 대한 추가 지원과 문의는 위키북스 출판사 홈페이지 wikibook.co.kr이나

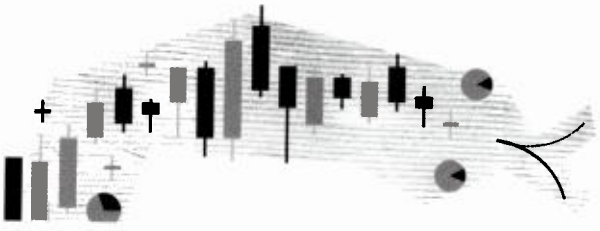
이메일 wikibook@wikibook.co.kr을 이용해 주세요.

이 도서의 국립중앙도서관 출판시도서목록 CIP는

서지정보유통지원시스템 홈페이지(http://seoji.nl.go.kr)와

국가자료공동목록시스템(http://www.nl.go.kr/kolisnet)에서 이용하실 수 있습니다.

CIP제어번호 **CIP2017010454**



파이썬으로 배우는 알고리즘 트레이딩

내 손으로 만드는 자동 주식 거래 시스템

조대표 지음



위키북스

저자 서문

저자는 두 권의 책을 공저했고 이번에 『파이썬으로 배우는 알고리즘 트레이딩』으로 세 번째 책을 집필했습니다. 서문을 쓰는 것은 기나긴 집필 과정을 끝냄을 의미하기 때문에 기분 좋은 일임이 분명합니다. 하지만 대학원 시절 몇 문장도 안 되는 논문의 서문을 수없이 수정했던 기억을 떠올리면 서문을 잘 쓰는 것은 쉽지 않고 조금 부담되는 일입니다. 이번에는 잘 좀 써보겠습니다.

저자는 주식을 잘 알지도 못하는 주알못입니다. 그런데도 주식과 관련된 책을 쓰기로 결심한 것은 국내에 알고리즘 트레이딩 분야에서 초보자들이 참고할 만한 입문서가 없기 때문입니다. 2010년경에는 대부분 C/C++ 같은 프로그래밍 언어로 알고리즘 트레이딩을 했기 때문에 프로그래머의 전유물이나 다름없었습니다. 물론 최근에는 C#이나 엑셀 기반으로 몇몇 도서가 출간됐지만 여전히 입문용으로는 어려워 보였습니다. 또한 근래에 출판되고 있는 주식 전문가들이 쓴 계량 투자 관련 책들도 이공계 전공을 한 저에게는 별나라 이야기처럼 들렸습니다. 분명 책을 읽을 때는 술술 읽히는 것 같은데 읽고 나면 이상하게 별로 남는 게 없었습니다.

이 책은 "프로그래밍 책이지만 기존의 전공자를 위한 프로그래밍 책이 아니라 비전공자와 일반인이 파이썬을 쉽고 재미있게 배울 수 있어야 한다"는 목표와 "이 책은 주식과 관련된 주식 책이지만 기존 주식 책과는 달리 실제 노하우를 어떻게 프로그램으로 구현할 수 있을지를 구체적으로 설명해야 한다"는 목표를 가지고 2014년부터 집필했습니다.

저자는 2001년도부터 파이썬이라는 프로그래밍 언어를 사용해왔지만 정작 밥벌이하는 일은 파이썬과 큰 관련은 없습니다. 파이썬은 제가 그냥 가장 좋아하는 프로그래밍 언어입니다. C/C++로 알고리즘 트레이딩을 하기에는 너무나 개발 시간이 오래 걸려서 마침 파이썬으로 개발을 시작했습니다.

공부를 하다 보니 파이썬이 쉬운 프로그래밍 언어이기 때문에 프로그래밍을 배운 적이 없는 일반인도 충분히 배울 수 있고 사용할 수 있다는 확신이 들었습니다. 그래서 위키독스(http://

저자 서문

wikidocs.net)라는 사이트에 본격적으로 글을 올리기 시작했습니다. 2018년 3월 기준으로 위키독스에 공개한 책은 한 달에 약 26만의 페이지뷰를 기록할 만큼 인기가 높습니다.

이 책은 제가 회사에 다니는 동안 평일 저녁과 주말에 틈틈이 공부하면서 집필했습니다. 저는 주식 투자야말로 직장인이 할 수 있는 최고의 재테크라고 생각하기 때문에 시간을 쪼개서 공부하고 있습니다. 현재 젊은이들이 살아가는 세대에는 100세 시대가 된다고 하는데 대학교 때 배운 지식을 바탕으로 일하다가 40~50대에 은퇴해야 하는 국내 현실을 고려하면 지금부터 차근차근 이 책을 통해 누구나 쉽게 알고리즘 트레이딩에 입문하기를 바랍니다. 단순히 이 책의 집필이 끝이 아니라 독자들과 소통하고 더 좋은 책을 만들기 위해 저자는 다음과 같은 사이트를 운영하고 있습니다. 많이 오셔서 참여 부탁드립니다.

- 네이버 카페(친목 및 정보 공유): http://cafe.naver.com/pystock
- Wiki 사이트(파이썬/알고리즘 트레이딩 관련 위키 사이트): http://sharebook.kr
- WIKIDOCS(이 책의 온라인 무료 버전): http://wikidocs.net/book/110

마지막으로 이 책의 출판을 제안하고 무려 1년 8개월 동안 도움을 주신 위키북스 박찬규 대표님과 리뷰를 도와주신 위키북스 직원분들과 후배 종훈에게 감사드립니다. 그리고 이 책을 쓰는 동안 저에게 건강상의 큰 시련이 왔지만 이를 이겨낼 수 있도록 응원해주시고 도와준 부모 형제들, 그리고 처가 식구들, 마지막으로 사랑하는 와이프 시후, 딸 수아, 아들 수민에게 이 책을 바칩니다. 모두의 바람처럼 앞으로 아프지 않고 건강하겠습니다.

조대표(조현호) 올림

목차

1부 파이썬 프로그래밍

1장 파이썬 시작하기 2

01. 파이썬과 알고리즘 트레이딩 2
- 프로그래밍과 프로그래밍 언어 3
- 주식투자 5
- 알고리즘 트레이딩 6
- 파이썬이란? 7

02. 파이썬 설치 8
- 아나콘다 설치 파일 다운로드 10
- 아나콘다 설치 10

03. 파이썬 인터프리터 실행 15
- 파이썬 IDLE 이용하기 16
- 파이썬 들여쓰기 18

연습 문제 20

2장 파이썬 변수와 문자열 21

01. 파이썬으로 하는 계산 21

02. 변수 24
- 파이썬 변수와 객체 26

03. 문자열 29
- 문자열 인덱싱 및 슬라이싱 30
- 문자열 자르기 32
- 문자열 합치기 33

04. 기본 데이터 타입 34

연습 문제 36

3장 기본 자료구조 38

01. 리스트 38

리스트 생성 41

리스트 인덱싱 41

리스트 슬라이싱 43

리스트에 데이터 삽입하기 46

리스트 데이터 삭제 47

02. 튜플 49

튜플 슬라이싱 50

03. 딕셔너리 50

딕셔너리에 데이터 삽입 및 삭제 52

딕셔너리로부터 키-값 구하기 53

연습 문제 56

4장 제어문 58

01. Boolean 58

02. 논리 연산자 61

03. if 문 62

if ~ else 문 64

if ~ elif ~ else 문 66

04. for 문 68

for와 range 70

for와 리스트 72

for와 튜플 73
for와 딕셔너리 73

05. while 문 75
while 문을 이용한 상한가 계산 76
while과 if 77
break와 continue 78

06. 중첩 루프 81

연습 문제 86

5장 함수와 모듈 89

01. 함수 90

02. 반환값이 있는 함수 94
함수 호출 과정 이해하기 95
두 개의 값 반환하기 99

03. 모듈 100
모듈 만들기 101
파이썬에서 시간 다루기 107
OS 모듈 109
모듈을 임포트하는 세 가지 방법 112

04. 파이썬 내장 함수 115

연습 문제 119

6장 클래스 121

01. 클래스란? 122

클래스 기초 124

클래스 정의 126

클래스에 메서드 추가하기 127

02. 클래스 생성자 132

03. self 이해하기 135

04. 클래스 네임스페이스 139

05. 클래스 변수와 인스턴스 변수 144

06. 클래스 상속 146

7장 파일 다루기 151

01. 파일 읽기 151

02. 파일 쓰기 156

연습 문제 158

8장 실전 프로젝트 159

01. PyCharm을 이용한 개발 환경 구축 159

PyCharm 설치 프로그램 다운로드 160

PyCharm 설치 161

PyCharm 초기 설정 164

프로젝트 생성 165
PyCharm 기타 설정 169

02. 주소록 프로젝트 170
Contact 클래스 만들기 171
사용자로부터 데이터 입력받기 173
메인 메뉴 구성하기 174
연락처 입력 동작시키기 175
연락처 출력하기 178
연락처 삭제하기 179

03. 주소록 프로젝트의 기능 향상 182
연락처 저장 함수 작성하기 182
연락처 불러들이기 184
명령 프롬프트에서 프로그램 실행하기 188

2부
증권사 API 기초

9장 파이썬과 COM 191

01. COM과 파이썬 192

02. 파이썬으로 엑셀 다루기 195

엑셀 파일 읽기 199

셀에 색깔 입히기 200

10장 대신증권 API 203

01. 개발 환경 구축 203

계좌 개설 및 CYBOS 5 설치 204

CYBOS Plus 로그인 206

02. 기초 API 익히기 208

연결 상태 확인 209

주식 코드 조회 211

03. 기본 API 익히기 215

종목 코드 가져오기 215

과거 데이터 구하기 219

PER, EPS 데이터 구하기 227

04. API를 이용한 알고리즘 개발 229

거래량 분석을 통한 대박 주 포착 230

업종별 PER 분석을 통한 유망 종목 찾기 237

05. 매수/매도 API 익히기 241

모의투자 가입하기 241

매수/매도하기 243

11장 이베스트투자증권 API 248

01. 개발 환경 구축 248
계좌 개설 및 xingAPI 패키지 설치 249
모의투자 가입하기 252

02. 기초 API 익히기 253
로그인 255
계좌 정보 조회 261

03. 기본 API 익히기 263
DevCenter 사용하기 263
단일 데이터 조회하기 266
반복 데이터 조회하기 271
차트 데이터 받아오기 276

12장 키움증권 API 281

01. 개발 환경 구축 282
계좌 개설 및 모듈 설치 282
모의투자 가입하기 287
KOA Studio 사용하기 289

02. PyQt 기초 292
PyQt 소개 292
Hello PyQt 294
위젯과 윈도우 297
이벤트 처리 301

03. 기초 API 익히기 303
Open API+ 로그인하기 303
로그인 이벤트 처리하기 307

기본 정보 요청하기 311
계좌 정보 가져오기 321
종목 코드 및 한글 종목명 가져오기 324

3부 데이터 분석 기초

13장 pandas를 이용한 데이터 분석 기초 328

01. pandas Series 329
파이썬 리스트, 튜플, 딕셔너리 329
Series 기초 331
Python Console 336

02. pandas DataFrame 338
DataFrame 생성 339
DataFrame 칼럼, 로우 선택 344

03. 주식 데이터 받기 345
DataReader 사용하기 348
차트 그리기 350

04. 이동평균선 구하기 353
이동평균선 354
pandas를 이용한 주가이동평균 계산 357
주가이동평균선 그리기 359

14장 pandas와 Zipline을 이용한 백테스팅 362

01. Zipline 시작하기 363
Zipline 설치 363
Zipline 기초 371

02. 이동평균선 전략 백테스트 382
Zipline을 이용한 이동평균선 전략 테스트 383
이동평균선 전략 보완하기 391

03. 유가증권시장/코스닥시장 백테스팅 395
거래 수수료 설정 395
초기 투자 금액 설정 397

15장 matplotlib를 이용한 데이터 시각화 399

01. pyplot 기초 400
간단한 그래프 그리기 400
Figure와 subplots 404
라벨 및 범례 표시하기 408
matplotlib 구성 412

02. 다양한 그래프 그리기 417
수정 종가와 거래량 한번에 그리기 417
캔들 스틱 차트 그리기 421
bar 차트 그리기 433
pie 차트 그리기 439

16장 PyQt를 이용한 GUI 프로그래밍 443

01. PyQt 기초 443
아나콘다에서 PyQt 설치 및 업데이트 444
PyQt 복습하기 445

02. Qt Designer 451
Qt Designer 소개 451

Qt Designer를 이용한 UI 구성 451
UI 파일을 파이썬 코드로 변환하기 460
UI 파일을 파이썬 코드에서 로드하기 462
이벤트 처리하기 464

03. 기본 위젯 466
QPushButton 4668
QLabel 468
QLineEdit와 QStatusBar 470
QRadioButton과 QGroupBox 472
QCheckBox 475
QSpinBox 479
QTableWidget 482

04. Layout 487
QVBoxLayout 489
QHBoxLayout 490
QGridLayout 492
레이아웃 중첩 493

05. 다이얼로그 498
QFileDialog 499
QInputDialog 501
메인 윈도우와 다이얼로그의 상호작용 503

06. PyQt와 matplotlib 연동 507
기본 레이아웃 구성 507
그래프 그리기 510

4부 실전 알고리즘 트레이딩

17장 주가 데이터 저장하기 515

01. SQLite 516
- sqlite3 모듈 기초 516
- 데이터베이스 브라우저 521
- 데이터베이스로부터 데이터 읽기 526

02. pandas와 SQLite 528
- DataFrame 객체를 SQLite DB에 저장하기 528
- SQLite DB에서 테이블 로드하기 530
- pandas를 이용한 주가 데이터 저장 532

03. 증권사 API를 이용한 주가 데이터 저장 534
- 종목 코드 리스트 얻어 오기 534
- 일봉 데이터 연속조회 540
- SQLite DB에 일봉 데이터 저장하기 549

18장 실전 프로그램 개발 554

01. 개발 1일 차 555
- 자동 버전 처리 스크립트 556
- 윈도우 작업 스케줄러 561
- PyTrader 구현 566
- 키움 OpenAPI+ 자동 로그인 574

02. 개발 2일 차 577
- UI 구성 578
- Kiwoom.py 파일 업데이트 585
- pytrader.py 파일 업데이트 587
- 매수 테스트 590

03. 개발 3일 차 592
UI 구성 593
Kiwoom.py 파일 업데이트 598
pytrader.py 파일 업데이트 606

04. 개발 4일 차 612
UI 구성 및 매수/매도 목록 파일 생성 612
선정 종목 정보 출력하기 614
자동 주문 구현하기 617

19장 실전 프로그램 개발(2) 622

01. 개발 5일 차 622
유가증권/코스닥 시장 종목코드 가져오기 624
일별 데이터 가져오기 626
급등주 포착 알고리즘 구현 628

02. 개발 6일 차 634
HTML 기초 635
웹 페이지 크롤링 639
배당률 기반 투자 알고리즘 648
배당률 기반 투자 알고리즘 구현(1) 653
배당률 기반 투자 알고리즘 구현(2) 662
배당률 기반 투자 알고리즘 구현(3) 666

부록 A. 소스코드 다운로드 및 실행 672

부록 B. PyCharm을 이용한 디버깅 676

01. 브레이크포인트 설정 676

02. 디버깅 모드로 프로그램 실행 678

03. 함수 디버깅하기 681

04. 추가 기능 살펴보기 684

부록 C. 유용한 PyCharm 단축키 687

연습 문제 풀이 691

1부

파이썬 프로그래밍

1부에서는 단순히 파이썬 문법을 설명하는 것이 아니라 주식과 관련된 예제를 통해 더욱 재미있게 파이썬을 배울 수 있습니다.

01

파이썬 시작하기

01 파이썬과 알고리즘 트레이딩

이 책을 보시는 분들 중 대부분은 이미 한 번쯤 주식투자를 해보신 분들일 것입니다. 그리고 열에 아홉은 투자금의 손실을 보신 분들이겠죠. 이 책을 쓰고 있는 저자 역시 재테크의 한 방편으로 몇 년 동안 주식투자를 해봤는데, 수익은커녕 적지 않은 금액의 손실을 봤습니다.

한마디로 저자는 주식 초보입니다. 혹시나 저자가 돈 버는 비법을 이 책에 적었기를 바라며 이 책을 구매하신 분들이 있다면 지금이라도 늦지 않았으니 빨리 이 책을 환불하시기 바랍니다. 상식적으로 생각해보십시오. 주식투자를 통해 쉽게 돈 버는 비법을 아는 저자가 무엇 때문에 3만 원 정도 하는 책에 그런 비법을 공개하겠습니까? 저뿐만 아니라 서점에 즐비한 수많은 주식 책들의 저자들이 돈 버는 비법을 알고 있었다면 왜 힘들게 머리를 쓰면서 책을 쓰고 있겠습니까? 그 시간에 투자해서 더 많은 돈을 버는 것이 수지 타산이 맞을 텐데요.

'그렇다면 도대체 이 책의 정체는 뭔가?' 하고 궁금하신 분들이 있을 겁니다. 앞서 말씀 드린 것처럼 제가 여러분께 돈 버는 비법을 알려드릴 수는 없습니다. 그러나 여러분이 컴퓨터를 통해 자동으로 주식을 매수/매도하거나 주식 시장을 분석해서 유용한 정보를 제공하는 컴퓨터 프로그램을 만드는 방법에 대해서는 알려드릴 수 있습니다.

이를 통해 여러분은 여러분만의 투자 기법을 더 쉽게 개발할 수 있으며, 이를 잘 응용한다면 돈을 많이 벌게 될 수도 있습니다. 이 책을 통해 배운 내용을 바탕으로 뭘 할 수 있을지 잘 이해되지 않으신다고요? 다음과 같이 더 구체적인 시나리오를 생각해봅시다.

1) 내가 자는 새벽 동안 컴퓨터가 미국 증시의 패턴을 분석해 아침에 카카오톡으로 국내 주식 시장에 대한 투자 방향을 알려준다.

2) 최근 유가증권시장/코스닥시장에서 거래량이 60일 평균치에 비해 1,000%(10배) 이상 급등한 종목을 나에게 알려준다.

3) 내가 지정해둔 종목의 주식을 알아서 적정 매수가격에 도달하면 사고, 설정한 매도가격에 오면 다시 팔아준다.

4) 유가증권시장/코스닥시장의 전 종목을 실시간으로 분석해 급등할 것 같은 종목들을 나에게 알려준다.

5) 주식 종목 추천 사이트의 무료 종목 추천을 매일매일 검색해 중복으로 추천된 종목을 나에게 미리 알려주고, 며칠 후에는 해당 종목들을 가상 투자했을 때의 수익률을 리포팅해준다.

이 책을 통해 알고리즘 트레이딩을 위한 프로그래밍 기초를 배우고 조금 더 공부하신다면 위의 시나리오뿐 아니라 여러분이 생각해둔 아이디어를 그대로 컴퓨터가 실행할 수 있게 만들 수 있습니다. 그렇다면 이렇게 하기 위해서는 무엇을 공부해야 할까요?

프로그래밍과 프로그래밍 언어

몇 년 동안 주식 투자를 해오신 분들은 나름대로 주식 투자를 위한 자신만의 노하우가 있을 겁니다. 예를 들어, 속칭 '상한가 따라잡기' 등과 같은 매수 기법도 일종의 노하우라고 할 수 있습니다. 컴퓨터와 관련된 학문에서는 노하우를 다른 말로 '알고리즘'이라고 합니다. 그리고 이러한 알고리즘을 컴퓨터가 알아들을 수 있도록 일련의 명령문으로 작성하는 것을 프로그래밍이라고 합니다.

아직 잘 이해가 안 가시죠? 누군가에게 여러분을 대신해 마트에서 물건을 사 오도록 지시하는 상황을 생각해 봅시다. 먼저 그 사람에게 가까운 마트에 어떻게 가는지(여러분만이 알고 있는 지름길이 있으시죠? 이게 바로 노하우, 즉 알고리즘입니다), 그리고 어떤 물건을 사서, 결제는 어떻게 하는지에 대해 알려줄 것입니다. 예를 들어, 다음과 같이 말이죠.

1. 일단 마을버스 777번을 타세요.
2. 3개의 정거장을 지난 후 하차하세요.
3. 정거장 우측으로 100m를 직진한 후 A 마트로 가세요.
4. 마트의 2층 식품 판매대에서 B를 사세요.
5. 1층 결제 코너에서 C 카드를 통해 결제하세요.
6. 길을 건너서 맞은편 버스 정류장에서 777 버스를 타세요.
7. 3개의 정거장을 지난 후 하차하세요.

컴퓨터에게 어떤 일을 시킬 때도 위와 같은 형태로 지시 내용을 나열해야 합니다. 이처럼 명령어가 나열된 것을 프로그램이라고 하며, 프로그램을 작성하는 행위를 프로그래밍이라고 합니다. 어떤가요? 프로그래밍이라는 것이 여러분의 삶과 동떨어진 이야기가 아니죠? 이미 여러분은 일상생활에서 프로그래밍을 해왔던 것입니다. 다만 이러한 지시 사항을 컴퓨터에게 이야기하는 것이 아니라 여러분의 가족이나 동료, 즉 사람에게 지시하거나 부탁해왔던 것이죠.

위와 같은 부탁이나 지시를 사람에게 할 때는 여러분과 지시를 받는 사람이 모두 알아들을 수 있는 한국어를 이용하면 되겠죠? 프로그램을 작성할 때도 명령을 내리는 여러분과 지시를 받는 컴퓨터가 모두 알아들을 수 있는 언어가 필요한데, 이를 프로그래밍 언어라고 합니다. 위의 예처럼 마트에 가서 물건 사는 법을 여러분의 친구에게 말했다면 그것이 바로 프로그램이고, 여러분이 한국어로 말했다면 한국어가 바로 프로그래밍 언어가 되는 것입니다.

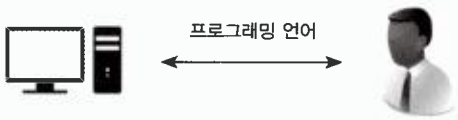


그림 1.1 프로그래밍 언어의 역할

그런데 사람이 아닌 컴퓨터에게 명령을 내리려면 어떻게 해야 할까요? 일단 앞에서 배운 것처럼 지시 사항을 순서대로 나열해야 합니다. 보통 이러한 작업을 명령어를 기술한다고 표현합니다. 이때 중요한 것은 명령어를 기술할 때 여러분과 컴퓨터가 서로 알아들을 수 있는 언어를 사용해야 한다는 점입니다.

우리가 사는 지구에 한국어, 영어, 중국어, 일본어 등 다양한 언어가 존재하는 것처럼 컴퓨터가 알아들을 수 있는 컴퓨터 프로그래밍 언어도 다양합니다. 이 책에서는 그중에서 파이썬(Python)이라는 언어를 배울 것입니다.

갑자기 프로그래밍 언어 이야기를 하니 벌써 걱정이 되고 머리가 아프죠? 걱정하지 마세요. 파이썬은 프로그래밍 언어 중 가장 배우기가 쉬워서 프로그래밍을 전공하지 않는 분들도 쉽게 배울 수 있답니다. 프로그래밍을 배우지 않은 분들이 시스템 트레이딩을 하고자 한다면 저는 다른 어떤 언어보다 파이썬을 추천해 드립니다. 파이썬은 엑셀을 기반으로 하는 VBA보다 배우기 쉽고, C/C++와 같은 언어와는 비교할 수 없을 만큼 쉽습니다.

주식투자

주식투자의 목적은 사람마다 다르겠지만 아마도 대부분의 사람은 그림 1.2와 같이 돈을 벌기 위해서일 것입니다. 쉽게 말하면 재테크 수단의 하나라는 것입니다.

부동산 투자를 통한 재테크는 집값 상승을 일으켜 사회적 약자에게 상실감을 주거나 경제를 심하게 왜곡할 수 있습니다. 이와 달리 주식투자는 여러분이 주식투자를 통해 이익을 보거나 손해를 본다고 해서 주변 사람에게 피해가 가는 것은 거의 없답니다. 물론 가족의 생활비를 주식투자로 모두 탕진해 버린다면 이야기가 달라집니다. 다시 말해, 주식투자는 사람들이 생각하는 것보다 상당히 건전하며, 주위 사람에게 피해를 주지 않는 건전한 재테크의 한 방법이라고 할 수 있습니다.

또한 사람들이 잘 보지도 않던 경제 뉴스를 챙겨 읽게 하고, 사회에서 일어나는 여러 가지 일들에 대해서도 좀 더 관심 있게 지켜보게 하는 좋은 역할도 합니다. 그러나 문제는 개인 투자자들이 주식시장에서 돈을 벌기가 생각만큼 쉽지 않다는 것이죠. 왜 개인투자자들은 주식시장에서 돈을 벌기가 어려울까요?



그림 1.2 주식과 부동산 투자의 목적

여기에는 여러 요인이 있겠지만 우선 개인 투자자들은 기관 투자자나 외국인보다 얻을 수 있는 정보가 한정적이라는 것이 첫 번째 이유입니다. 아시는 분도 있겠지만 일부 회사는 회사의 정보를 미리 기관 투자자나 외국인에게 불법적으로 제공하기도 합니다. 이따금 증권방송에서 일부 애널리스트들이 특정 주식 종목을 추천하지만 정작 자신들은 이미 매수를 해놓고 기다리다가 일반 투자자들이 매수를 시작

하면 바로 해당 주식을 매도해서 수익을 올리기도 합니다. 이러한 요인은 모두 개인 투자자의 역량과는 상관없이 환경적으로 발생하는 외적 요인이라 할 수 있습니다.

개인 투자자들이 주식투자에 실패하는 또 다른 이유는 개인 투자자들은 주식을 생업으로 삼고 투자하는 경우가 많지 않아 실제로 주식 투자에 대해 공부하기보다는 무작정 '특정 종목이 오를 것 같다', '이 종목은 충분히 저점이다'라고 판단하거나, 또는 주변의 누군가로부터 추천받은 종목을 아무런 사전 조사 없이 사는 경향이 강하다는 점입니다. 이러한 것들은 모두 개인과 관련된 내적 요인이라 할 수 있습니다.

이 같은 내적 요인과 외적 요인을 극복하고 '용돈 벌이', '재테크'를 목적으로 주식 투자를 하는 개인 투자자들은 어떻게 해야 거대 기관 투자자나 외국인 사이에서 주식투자에 성공할 수 있을까요?

이 질문의 답은 여러 가지가 있겠지만 저자는 이 책에서 다룰 '알고리즘 트레이딩'이 하나의 답이 될 수 있다고 생각합니다. 개미들이 시장에서 성공할 수 없는 이유를 다음과 같은 두 가지로 요약했을 때 알고리즘 트레이딩을 이용하면 이러한 문제를 어느 정도 해결할 수 있기 때문입니다.

- 정보의 부족
- 시간의 부족

알고리즘 트레이딩

그렇다면 '알고리즘 트레이딩'이란 무엇일까요? 일단 트레이딩은 한국어로 '거래'를 의미하고, 알고리즘은 주로 컴퓨터 분야에서 많이 사용하는 용어로서 '어떤 문제를 해결하기 위한 절차나 방법'을 의미합니다. 즉, 알고리즘 트레이딩이란 '컴퓨터를 이용해 어떤 일정한 알고리즘에 따라 자동으로 거래하는 매매 방식'을 의미합니다.

앞에서도 간단히 예를 들어 설명했지만 알고리즘 트레이딩의 구체적인 시나리오에 대해 한 번 더 생각해 봅시다. 먼저 컴퓨터가 주식 시장이 개장되기 전에 전날의 거래 패턴(거래량, 종가, 시가, 주식 가격의 변동 폭)을 분석하고 이를 통해 여러분에게 투자 유망 종목을 자동으로 추천해 줄 수 있습니다. 또는 더 나아가 여러분이 낮에 일하는 동안 컴퓨터가 자동으로 주식을 매수하고, 매도하면서 수익을 낼 수도 있습니다.

최근 정기적금의 은행이자가 연 2% 이하임을 고려했을 때 여러분이 잘 만든 시스템은 하루에 15% 이상의 수익을 낼 수도 있는 것입니다. 물론 온종일 사고팔고를 잘한다면 그 이상의 수익률도 이론적으로 불가능하지는 않습니다.

알고리즘 트레이딩의 개념을 알아봤으니 그럼 이번에는 어떻게 해야 컴퓨터로 이런 것들을 만들 수 있을지 알아봅시다.

효율적인 '프로그램 매매'를 위해서는 무엇보다도 여러분만의 알고리즘(투자 노하우)이 있어야 합니다. 예를 들어, 여러 주식 책을 보면 거래량 분석을 통해 다수의 세력이 장기간에 걸쳐 매집하는 종목을 찾는 알고리즘이 있습니다. 이 알고리즘은 몇 개월 동안 거래량을 분석하고 있다가 특정 종목의 일일 거래량이 최근의 평균 거래량보다 1,000% 이상 급등하는 종목을 찾습니다. 이처럼 단기간의 거래량이 평균 거래량보다 폭발적으로 늘어나는 종목은 단기간에 급등할 수 있다는 것이 주요 아이디어입니다.

문제는 국내 주식 시장(유가증권시장, 코스닥시장)의 주식 종목이 2,000개 정도 되기 때문에 매일 퇴근한 후 이런 종목들을 자세히 조사하고 투자 계획을 세우기가 어렵다는 것입니다. 회사에서 이런 일을 한다면 아마 여러분은 당장 해고될 것입니다.

이 문제의 간단한 해결책은 그런 일들은 컴퓨터가 대신 하게끔 시키면 된다는 것이죠. 사실 컴퓨터는 스스로 생각하거나 판단하는 것은 사람보다 못하지만(물론 아직은 그렇다는 얘깁니다) 다수의 데이터를 반복적으로 처리하는 것은 사람보다 월등히 잘합니다. 그리고 여러분의 인건비에 비하면 컴퓨터를 사용하는 데 드는 비용은 구매 비용과 전기세 정도밖에 안 됩니다.

알고리즘이 준비되면 다음 단계는 프로그래밍 언어를 이용해 여러분의 '알고리즘'을 컴퓨터가 알아들을 수 있게 프로그래밍하는 것입니다.

파이썬이란?

파이썬은 프로그래밍 언어의 일종이라고 앞에서 간단히 설명했습니다. 파이썬이 프로그래밍 언어라는 사실을 아는 걸로도 충분하겠지만 다른 누군가에게 파이썬에 대해 좀 더 자세히 설명해줄 수 있다면 좋겠죠? 이를 위해 조금 지루할 수 있겠지만 파이썬이라는 프로그래밍 언어의 역사와 특징을 좀 더 알아보겠습니다.

파이썬은 1991년 귀도 반 로섬(Guido van Rossum)이 만든 배우기 쉽고 강력한 프로그래밍 언어입니다. 곧 알게 되겠지만 파이썬은 C, C++, 자바(Java) 같은 프로그래밍 언어보다 훨씬 배우기 쉽고 문법도 간단합니다.

또한 파이썬은 C, C++, 자바보다 약 1/3 또는 1/5 정도의 코드만으로도 같은 기능을 수행할 수 있어 개발 생산성이 매우 높습니다. 쉽게 말해서 컴퓨터에게 대충 명령을 내려도 컴퓨터가 알아서 일을 잘 처리하기 때문에 C, C++, 자바보다 프로그래밍하는 시간이 줄어든다는 것입니다. 같은 알고리즘을 C,

C++ 프로그래머가 구현해서 트레이딩 시스템을 구축하는 것보다 파이썬을 이용했을 때 개발 시간을 크게 단축할 수 있다는 뜻입니다.

파이썬이라는 이름은 귀도 반 로섬이 좋아하는 코미디 프로그램인 "Monty Python's Flying Circus"에서 따온 것이라고 합니다. 하지만 파이썬이라는 이름의 뱀도 있어서 실제로 파이썬 프로그래밍 책을 보면 표지에 뱀이 그려진 것이 많습니다. 그리고 파이썬과 관련된 프로그램의 아이콘에도 뱀이 자주 등장하는 것을 볼 수 있습니다. 그렇다면 파이썬을 개발한 귀도 반 로섬은 요즘 무슨 일을 하고 있을까요?

- https://www.python.org/~guido/

위의 URL로 들어가 보면 파이썬 공식 홈페이지에 귀도를 위한 페이지가 있는 것을 볼 수 있습니다. 뭔가 창시자에 대한 예우처럼 보이지 않습니까? 귀도는 파이썬 언어의 발전에 크게 기여해오고 있는데, 2013년까지는 구글(Google)에서 일했고 최근에는 드롭박스(Dropbox)로 회사를 옮겼습니다. 중요한 점은 구글과 드롭박스에서 약 50%의 시간에는 회사를 위해 일하고, 나머지 50%는 파이썬을 개발하는 데 시간을 보내고 있다고 합니다. 그만큼 세계 최고 수준의 프로그래머에 대한 회사 차원의 배려가 아닌가 합니다. 물론 구글과 드롭박스의 많은 부분이 파이썬으로 개발되고 있어서 파이썬 자체의 발전을 위한 투자로도 생각해볼 수 있습니다.

02 파이썬 설치

본격적으로 파이썬이라는 프로그래밍 언어로 프로그래밍을 공부하기에 앞서 먼저 파이썬을 설치해 보겠습니다. 이 책에서는 마이크로소프트 윈도우 7 환경에서 파이썬을 설치하는 경우만 다루겠지만 파이썬은 윈도우 버전에 크게 상관없이 설치할 수 있습니다. 또한 리눅스나 맥 OS X과 같은 시스템에도 쉽게 설치할 수 있습니다.

파이썬을 설치하려면 먼저 설치 프로그램을 내려받아야 합니다. 파이썬 설치 파일은 크게 두 가지로 구분됩니다.

- 파이썬 공식 홈페이지에서 제공하는 설치 파일
- 배포판이라고 불리는 설치 파일

첫 번째는 파이썬 공식 홈페이지(http://www.python.org)에서 제공하는 설치 파일로서 CPython이라고 합니다. 해당 파일을 내려받아 설치하면 파이썬 인터프리터와 몇 가지 프로그램이 설치됩니다. 이 설치 파일은 파이썬으로 프로그래밍할 수 있도록 최소한의 프로그램을 컴퓨터에 설치합니다. 그래서 설치 파일의 용량도 23MB 정도로 적은 편입니다.

두 번째는 배포판이라고 불리는 설치 파일입니다. 뒤에서 설명하겠지만 파이썬에는 파이썬 패키지라고 하는 것들이 있는데, 여기서 패키지라는 것은 특정 분야에 최적화된 프로그램의 묶음이라고 보면 됩니다. 예를 들어, 주식 프로그램을 작성하는데 처음부터 모든 것을 다 작성하는 것이 아니라 주식 패키지를 설치하면 기본적인 것들이 이미 구현돼 있는 것이죠. 어떤가요? 말로만 들어도 편하겠지요? 좀 더 쉽게 설명하면 어떤 보고서를 작성하는데, 이미 누군가가 그 보고서의 초안을 작성한 후 제공하는 것과 비슷합니다.

그런데 문제는 파이썬에는 패키지가 너무나도 많아서 필요한 패키지를 그때그때 내려받아서 직접 설치해야 한다는 것입니다. 그리고 요즘은 상황이 좀 더 나아졌지만 파이썬 버전에 따라 일부 패키지는 초보자가 설치하기에는 좀 어려운 경우가 있습니다.

이러한 문제를 깔끔하게 해결하고자 배포판이라고 하는 설치 파일들이 등장했습니다. 쉽게 말해서 배포판은 '파이썬 인터프리터 + 파이썬 패키지'라고 생각하면 됩니다. 배포판은 파이썬 인터프리터와 자주 사용하는 백여 개의 패키지를 한 번에 컴퓨터에 설치해줍니다. 초보자 입장에서 보면 배포판은 매우 편리한 설치 파일의 한 종류입니다. 다만 설치 파일의 크기가 200MB~1GB 정도로 크다는 단점이 있습니다. 물론 둘 중 어떤 설치 파일로 설치하든 파이썬 프로그램을 작성하는 데는 문제가 없지만 이 책에서는 여러 유용한 패키지까지 한 번에 설치할 수 있는 '아나콘다'라는 배포판을 이용할 것입니다. 현재 널리 사용되는 파이썬 배포판을 표 1.1에 정리했습니다.

표 1.1 파이썬 배포판

배포판 이름	공식 홈페이지
Anaconda	https://www.anaconda.com/download/
Python(x, y)	http://python-xy.github.io/
Enthought Canopy	https://www.enthought.com/products/epd/
WinPython	http://winpython.github.io/

아나콘다 설치 파일 다운로드

아나콘다는 Continuum Analytics라는 곳에서 만든 파이썬 배포판으로, 수백 개의 파이썬 패키지를 포함하고 있습니다. 또한 상업용으로도 무료로 사용할 수 있으므로 회사 내에서도 사용할 수 있다는 장점이 있습니다.

먼저 아나콘다 배포판을 설치하기 위해 다음 URL로 이동합니다. 파이썬은 현재 2.7 버전과 3.6 버전이 있는데, 이 책에서는 파이썬 3.6을 사용합니다.

- https://www.anaconda.com/download/

파이썬 3.6 버전에도 다시 윈도우 64비트용 설치 파일과 32비트용 설치 파일이 있습니다. 보통은 사용 중인 PC에 64비트 윈도우가 설치돼 있으면 64비트용 설치 파일을 내려받고, 32비트 윈도우가 설치돼 있다면 32비트용 설치 파일을 내려받으면 됩니다.

그러나 이 책의 후반부에서 다룰 국내 증권사 API를 정상적으로 사용하려면 반드시 그림 1.3과 같이 32비트용 설치 파일(32-Bit Graphical Installer)을 클릭해서 내려받아야 합니다. 특별한 경우가 아니라면 파이썬 3.6 32비트용 설치 파일을 내려받기 바랍니다. 참고로 내려받은 파일의 이름에 x64가 포함되어 있다면 64비트용 설치 파일을 받으신겁니다.



그림 1.3 아나콘다 32비트 배포판의 설치 파일 선택

아나콘다 설치

아나콘다 32비트 배포판 설치 파일을 정상적으로 내려받았다면 해당 파일(Anaconda3-5.1.0-Windows-x86.exe)에 마우스 오른쪽 버튼을 클릭한 후 [관리자 권한으로 실행] 메뉴를 선택해서 설치를 시작합니다. 그림 1.4와 같이 첫 번째 설치 화면이 나오면 [Next] 버튼을 클릭해 다음 단계로 진행합니다.

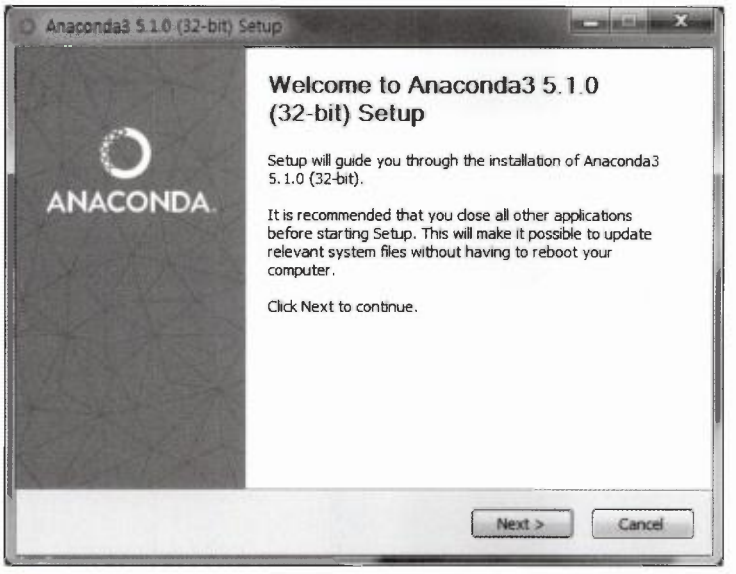


그림 1.4 아나콘다 설치(1/8)

그림 1.5는 라이선스 동의에 관한 내용입니다. [I Agree] 버튼을 클릭해 다음 단계로 이동합니다.

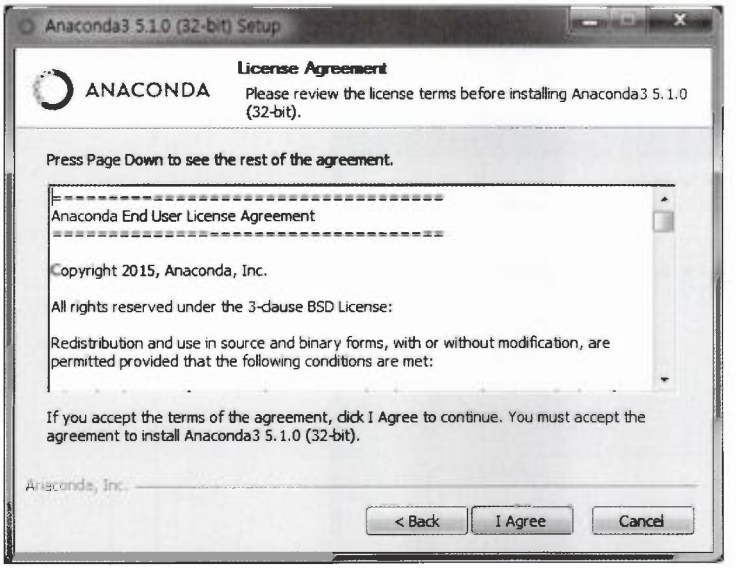


그림 1.5 아나콘다 설치(2/8)

설치 유형을 선택하는 단계에서는 [All Users]를 선택한 후 [Next] 버튼을 클릭합니다.

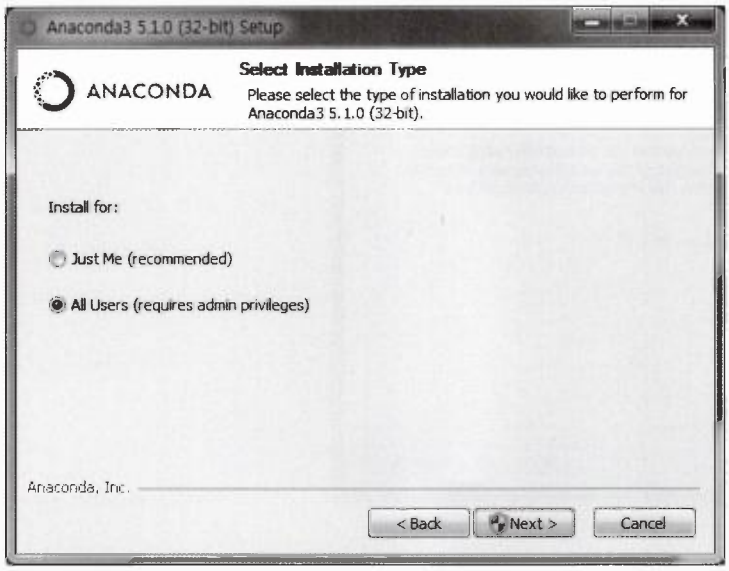


그림 1.6 아나콘다 설치(3/8)

아나콘다 배포판이 설치될 디렉터리를 선택하는 화면에서는 [Destination Folder]를 'C:\Anaconda3'으로 변경한 후 [Next] 버튼을 클릭합니다.

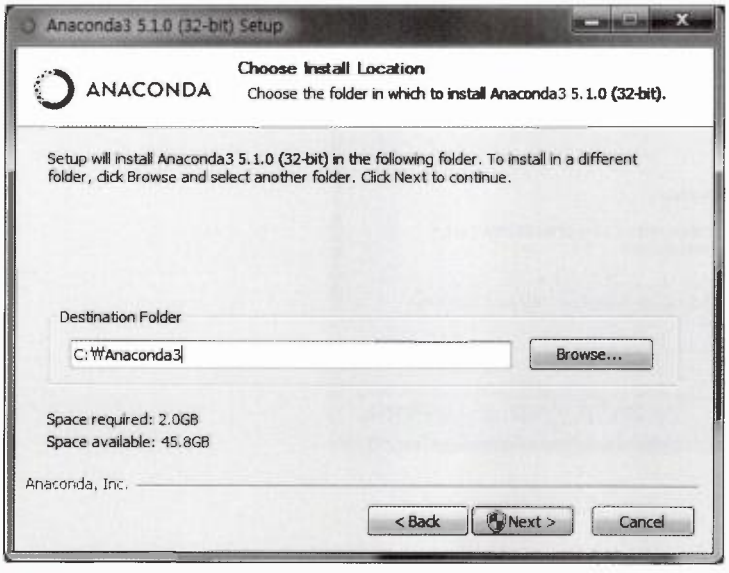


그림 1.7 아나콘다 설치(4/8)

추가 옵션 선택 화면에서 그림 1.8과 같이 두 가지 옵션을 모두 선택한 후 [Install] 버튼을 클릭해 다음 단계로 이동합니다.

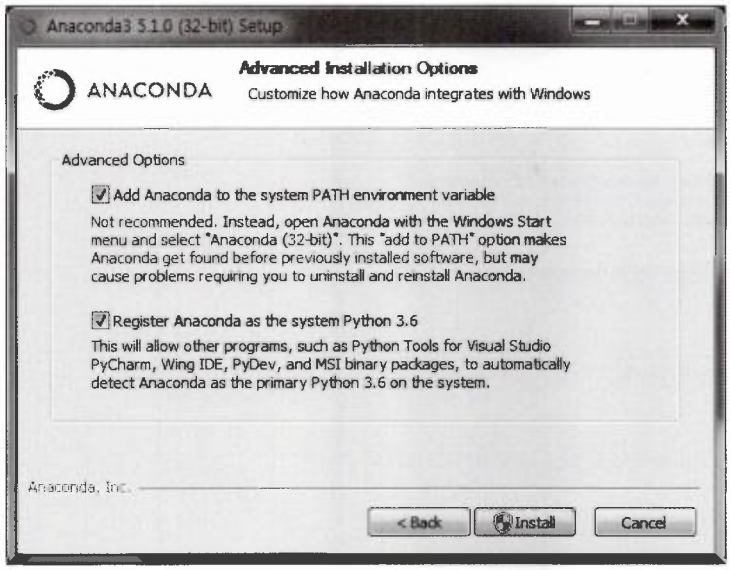


그림 1.8 아나콘다 설치(5/8)

설치가 완료되면 화면에 'Completed'라는 메시지가 출력됩니다. 이때 [Next] 버튼을 눌러 다음 단계로 이동합니다.

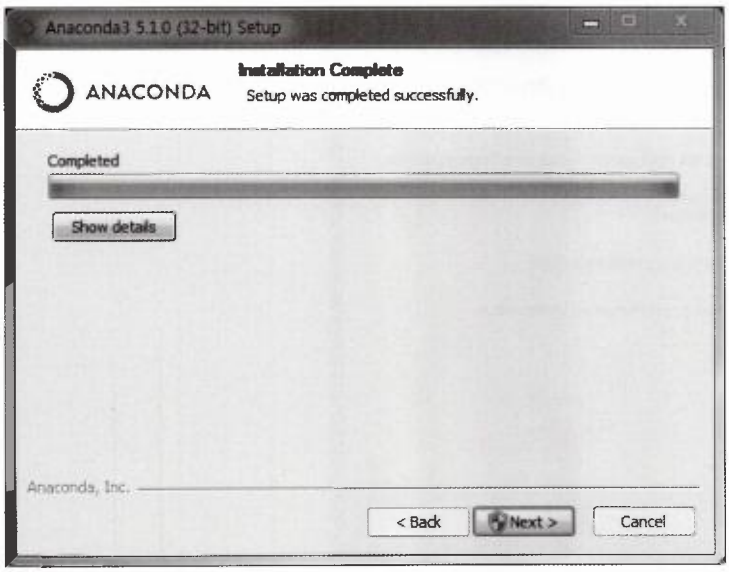


그림 1.9 아나콘다 설치(6/8)

그림 1.10에서 [Skip] 버튼을 클릭해 'Visual Studio Code'라는 프로그램은 설치하지 않고 다음 단계로 넘어갑니다.

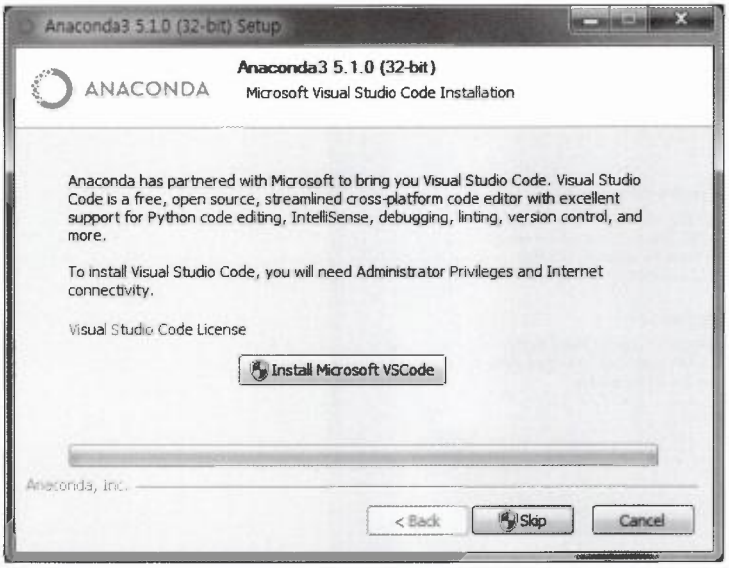


그림 1.10 아나콘다 설치(7/8)

이로써 아나콘다 배포판의 모든 설치 과정이 완료됐습니다. 그림 1.11과 같이 [Finish] 버튼을 클릭해 설치 과정을 종료합니다.

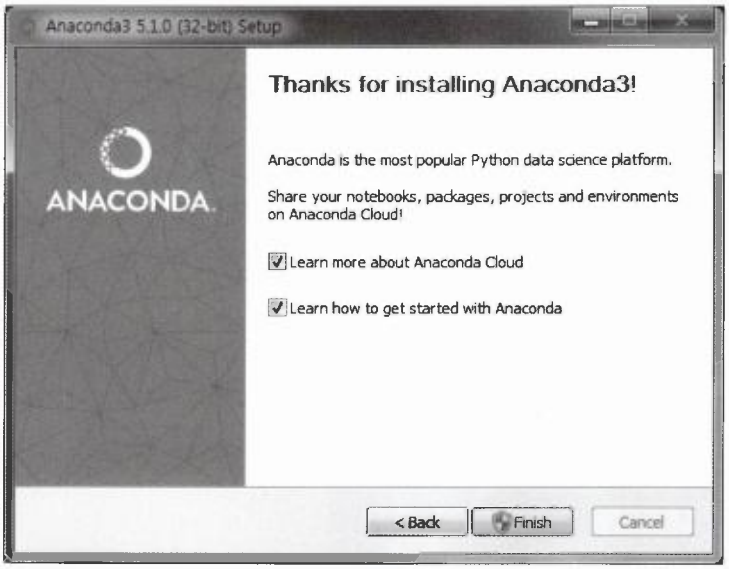


그림 1.11 아나콘다 설치(8/8)

03 파이썬 인터프리터 실행

1.2절을 통해 PC에 정상적으로 '아나콘다'라는 이름의 파이썬 배포판을 설치했습니다. 아마도 프로그래밍이란 것을 한 번도 해보지 않는 분들은 배우기도 전에 뭔가 복잡한 설치 과정이 있어 조금 당황스러우셨을 겁니다. 그렇지만 옛말에 '시작이 반'이라고 했습니다. 조금만 참고 따라오시면 재미있는 프로그래밍의 세계로 입문하실 수 있을 것입니다.

앞에서 아나콘다 배포판을 'C:\Anaconda3'에 설치했습니다. 먼저 윈도우의 탐색기 등을 사용해 해당 경로로 이동해 봅시다. 그림 1.12와 같이 해당 경로에 여러 파일과 디렉터리가 설치돼 있음을 확인할 수 있습니다. 이 경로에서 Lib 디렉터리로 이동한 후 다시 idlelib 디렉터리로 이동합니다. 최종 경로는 다음과 같습니다.

```
C:\Anaconda3\Lib\idlelib
```

위의 경로에도 여러 디렉터리와 파일이 있는데 그림 1.13과 같이 idle.bat 파일에 마우스 오른쪽 버튼을 클릭한 후 [보내기(N)] → [바탕 화면에 바로 가기 만들기]를 선택합니다. 파이썬으로 프로그래밍하려면 파이썬 IDLE(Integrated DeveLopment Environment)가 필요한데, 해당 파일을 더 쉽게 실행하기 위해 바탕 화면에 바로 가기 아이콘을 생성한 것입니다. 이 과정을 모두 잘 따라 했다면 그림 1.14와 같이 바탕 화면에 아이콘이 생성된 것을 확인할 수 있습니다.

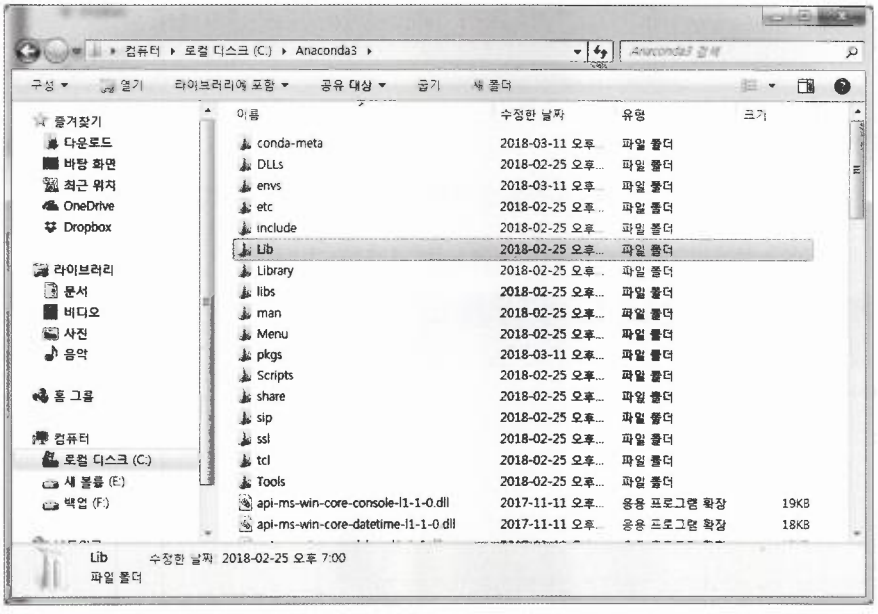


그림 1.12 Anaconda3 디렉터리의 구성

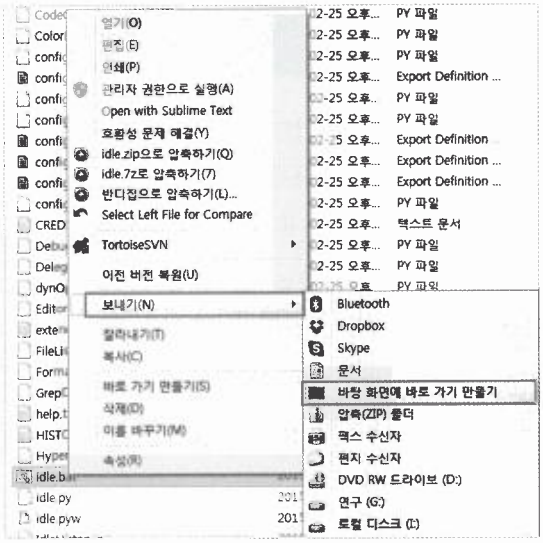


그림 1.13 idle.bat 파일에 대한 바로 가기 아이콘 만들기



그림 1.14 바탕 화면에 만들어진 idle.bat 바로 가기

파이썬 IDLE 이용하기

바탕 화면에 생성된 [idle.bat - 바로 가기] 아이콘을 클릭해 파이썬 IDLE를 실행합니다. 지금까지 설명한 과정을 문제 없이 따라 했다면 그림 1.15와 같이 파이썬 IDLE가 실행됩니다.

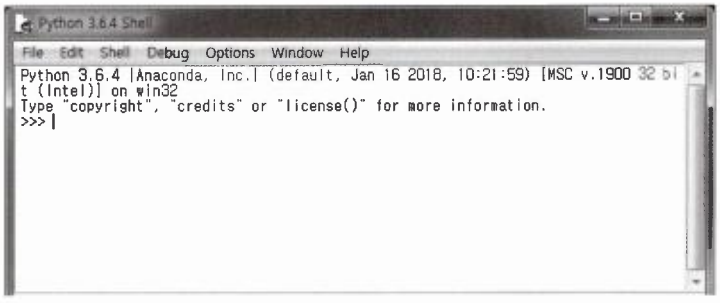


그림 1.15 Python IDLE 실행 화면

컴퓨터로 보고서를 쓸 때 마이크로소프트 워드나 한글과 컴퓨터의 한글이라는 프로그램을 실행한 후 글을 쓰듯이 파이썬 프로그램을 작성하기 위해 파이썬 IDLE를 사용하는 것입니다. 여기서 IDLE(Integrated DeveLopment Environment)라는 단어를 해석해보면 "통합 개발 환경"이라는 의미입니다. 아마도 '파이썬 프로그래밍을 통합적으로 지원하는 개발 환경'이라는 뜻에서 IDLE라는 이름을 붙인 것 같습니다.

한 번도 프로그래밍을 해보지 않았던 분들은 무슨 뜻인지 모르겠지만 앞에서 워드와 한글이 글쓰기를 위한 여러 가지 기능을 제공하는 것처럼 파이썬 IDLE도 여러분이 좀 더 쉽게 프로그래밍할 수 있게 몇 가지 기능을 제공합니다. 파이썬 문법을 배우는 1부에서는 주로 IDLE를 이용해 프로그래밍하는 법을 배우겠습니다.

옛말에 '백문이 불여일견'이라는 말이 있는데, 프로그래머들 사이에는 '백문이 불여일타'라는 말이 있답니다. 백 번 듣는 것보다 한번 보는 것이 나은 것처럼, 프로그래밍을 공부할 때는 백 번 듣는 것보다 한 번 타이핑(typing)해보는 것이 낫다는 의미입니다.

사람들이 영어 공부할 때 보면 선생님이나 강사님이 아무리 강의를 잘해도 자기가 직접 말하거나 쓰지 않는다면 실력이 빨리 늘지 않는 것과 같은 이치입니다. 프로그래밍 언어도 하나의 "언어"라는 점을 잊지 말고 이 책에서 나오는 각 예제를 똑같이 실습하다 보면 금방 실력이 향상되는 것을 확인할 수 있을 것입니다.

그림 1.15에서 '>>>'는 프롬프트(prompt)라고 합니다. 앞으로 이 프롬프트에 파이썬 문법에 맞는 명령을 내리면 컴퓨터가 여러분의 명령을 해석해 실제 동작을 수행하게 됩니다. 여기서 중요한 점은 여러분이 내리는 명령이 파이썬의 문법을 따라야 한다는 점입니다. IDLE 프롬프트에 다음과 같이 print 명령어를 입력한 후 키보드의 엔터(ENTER) 키를 눌러 봅시다.

```
>>> print("Hello World")
Hello World
```

파이썬 IDLE를 보면 그림 1.16과 같이 화면에 Hello World라는 글자가 출력(print)된 것을 볼 수 있습니다. 조금 전 파이썬 프롬프트(>>>) 상에서 print라는 명령어를 사용해 Hello World라는 문장을 화면에 출력하라는 명령을 내렸기 때문에 컴퓨터가 이를 알아듣고 해당 명령어를 수행한 것입니다.

파이썬을 공부하다 보면 앞으로 기본적으로 외워야 하는 것들이 여럿 나오는데, 대표적으로 print라는 함수가 있습니다. 함수는 중학교 수학책에 나오는데, 보통 어떤 입력을 주면 어떤 정해진 기능을 수행하는 것을 의미합니다.

파이썬에서도 비슷하게 print라는 함수는 괄호 안으로 입력되는 함수의 인자(argument)를 화면에 출력하는 기능을 제공합니다. 그래서 print("Hello World")의 수행 결과로 화면에 Hello World가 출력되는 것입니다. 참고로 print 함수로 문자를 출력할 때는 출력하고자 하는 문자를 큰따옴표로 양쪽을 둘러싸거나 작은따옴표로 양쪽을 둘러싸면 됩니다.

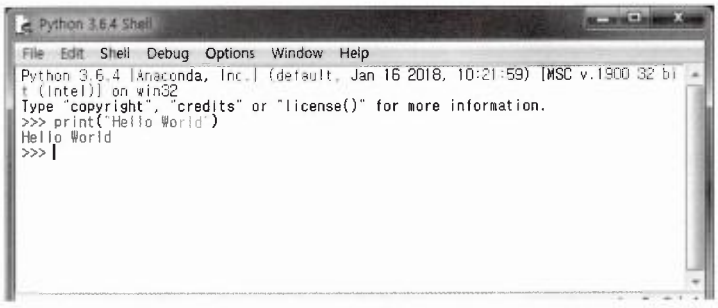


그림 1.16 print 명령어의 수행 결과

그렇다면 꼭 영어만 출력되는 것일까요? 물론, 정답은 "아니오"입니다. 프롬프트에서 다음과 같이 print 함수를 호출해도 함수의 인자 값인 '안녕하세요'가 정상적으로 화면에 출력되는 것을 확인할 수 있습니다.

```
>>> print("안녕하세요.")
안녕하세요.
```

막상 처음으로 프로그래밍해보니 어떤가요? 뭔가 부족함이 느껴지지 않습니까? 프로그래밍을 하게 되면 뭔가 주식 가격을 알아맞히고, 대규모 매수 세력의 존재를 확인하고 이를 분석해서 종목을 추천하는 그런 대단한 프로그램을 만들 것을 기대했는데, 고작 화면에 문장을 출력하고 있다니요.

그러나 너무 조급해 하지 마시기 바랍니다. 영어 단기 완성이나 1주일 완성이라고 내건 곳을 몇 년 다녀도 영어를 제대로 구사하는 분들을 찾아보기 힘든 것처럼 프로그래밍을 아무것도 모르던 분들이 하루 이틀 공부한다고 프로그래밍을 잘 하게 되는 것은 아닙니다. 물론 이 책을 보고 일주일 정도 열심히 공부한다면 잘할 수 있을 것입니다. 희망을 품고 꾸준히 공부하기 바랍니다.

파이썬 들여쓰기

파이썬은 다른 프로그래밍 언어와 달리 들여쓰기(indent)에 매우 민감한 언어입니다. 보통 C나 C++ 같은 다른 프로그래밍 언어는 '{'와 '}' 같은 기호를 사용해 코드가 수행되는 영역을 표시합니다. 그렇지만 '{'와 '}' 기호는 초보자가 보기에는 조금 어려워 보입니다. 그리고 C나 C++ 같은 프로그래밍 언어

는 사실 띄어쓰기 없이도 코딩할 수 있으므로 코드 자체는 수행되지만 사람들이 코드를 보고 이해하기가 어려울 수 있습니다.

"나는아침을먹고학교에갔는데오늘은학교개교기념일이었다."

우리의 모국어인 한국어도 위와 같이 띄어쓰기 없이 붙여서 쓰면 정말 읽기가 어렵죠? 프로그램도 마찬가지입니다. 프로그램을 작성할 때는 작성한 코드를 적당히 띄어쓰기해야 가독성이 좋아진답니다(코드를 읽기 쉽다는 뜻입니다).

파이썬은 '{'와 '}' 기호 없이 들여쓰기로만 영역을 나타내기 때문에 항상 조심해서 들여쓰기해야 합니다. 예를 들어, 그림 1.17과 같이 파이썬 IDLE에서 한 칸 공백을 주고(키보드의 스페이스 바를 누르면 됩니다) 명령어 코드를 작성하면 정상적으로 수행되지 않습니다.

아직은 잘 모르겠지만 '파이썬에서는 띄어쓰기가 정말 중요하다'라는 점을 꼭 기억하기 바랍니다. 사실 저도 파이썬을 처음 배울 때는 언제 띄어쓰기를 하고 언제는 하지 말아야 할지 판단하기가 어려웠는데, 이는 파이썬 문법에 좀 더 익숙해지면 저절로 해결되는 문제입니다. 앞으로 파이썬 IDLE에서 프로그래밍할 때 'Indent'라는 단어가 포함된 오류 메시지가 나타나면 띄어쓰기에 문제가 있는지 코드를 확인해보기 바랍니다.

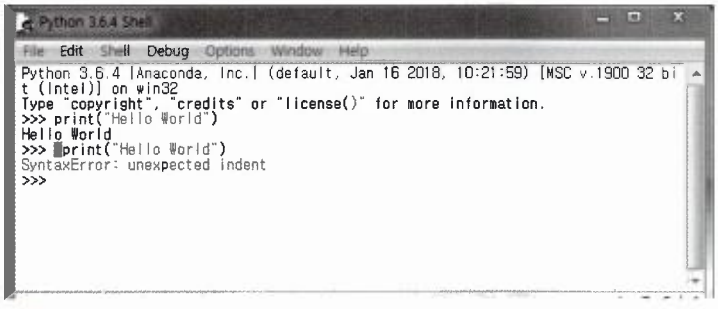


그림 1.17 파이썬 들여쓰기 오류의 예

연습 문제

파이썬이라는 언어를 배울 때 가장 좋은 방법은 직접 프로그래밍해보는 것입니다. 눈으로 아무리 책을 읽더라도 직접 문제를 해결해보고 키보드로 타이핑해보지 않으면 배우기가 쉽지 않습니다. 다음 장으로 진도를 나가기 전에 꼭 아래의 연습문제를 직접 해결해보기 바랍니다.

문제 1-1

파이썬을 윈도우 환경에 직접 설치합니다.

문제 1-2

파이썬 IDLE를 실행한 후 화면에 '주식은 대박이다.'라는 글자를 출력해 보세요.

문제 1-3

파이썬에서 콘솔에 어떤 값을 출력할 때 사용하는 키워드(keyword)는 무엇인가요?

문제 1-4

다음 다섯 가지 파이썬 표현 중 정상적으로 화면에 값이 출력되는 것을 고르시오.

1) `print("I love 'you'")`

2) `print("I like you')`

3) `print('Korea')`

4) `print{Hello}`

5) `print[Hello]`

02

파이썬 변수와 문자열

이번 장에서는 파이썬을 계산기처럼 사용해 보면서 파이썬에 좀 더 익숙해진 후, 모든 프로그래밍 언어에서 사용하는 기본 개념 중 하나인 변수(variable)에 대해 배우겠습니다. 그리고 파이썬 프로그래밍의 기본 데이터 타입인 정수, 실수, 문자열(string)을 알아보겠습니다.

01 파이썬으로 하는 계산

네이버(Naver) 주식 10주를 가지고 있다고 가정해 봅시다. 참고로 2015년 6월 12일 장 종료를 기준으로 네이버 주가는 한 주당 601,000원이었습니다. 10주를 모두 주당 601,000원에 매도했다면 매도 금액은 얼마가 될까요?

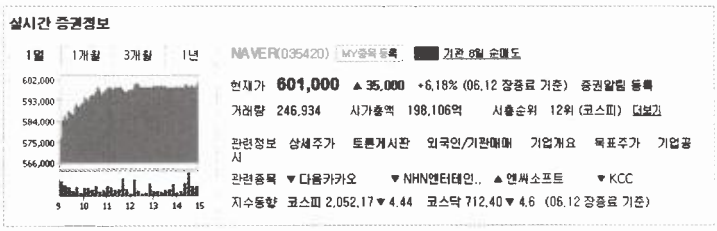


그림 2.1 네이버 주가(출처: 다음증권)

계산이 빠르신 분들은 아마 6,010,000원이라고 바로 답하신 분도 있겠지만 산수에 약하신 분들은 그림 2.2와 같이 윈도우에서 제공하는 계산기를 사용하실 것입니다.



그림 2.2 계산기를 이용한 계산

1장에서 배운 파이썬 IDLE을 이용하면 윈도우 계산기를 이용하는 것보다 더 쉽게 이러한 계산을 할 수 있습니다. 윈도우 계산기에서 벗어나 파이썬 IDLE에서 계산하는 순간, 이미 여러분은 프로그래밍을 하고 있는 것입니다. 먼저 파이썬 IDLE를 실행한 후 파이썬 프롬프트에 "601000*10"을 입력한 후 엔터 키를 눌러봅시다.

```
>>> 601000*10
6010000
>>>
```

윈도우 계산기를 이용했던 것과 마찬가지로 주당 601000을 입력하고(이때 쉼표를 넣지 않아야 합니다) 곱하기 기호를 누른 후 10을 입력했습니다. 그런 다음 엔터 키를 누르면 여러분이 예상했던 것처럼 6,010,000으로 정확히 계산됩니다. '프로그래밍을 공부하다가 갑자기 웬 계산이냐?'라고 생각하는 분도 있겠지만 주식에서는 더하기, 빼기, 곱하기, 나누기를 잘해야 돈을 벌 수 있습니다.

예를 들어, 매수 금액 대비 현재가가 3% 이하로 떨어지면 보유 주식을 모두 매도하는 프로그램을 만든다고 가정해 봅시다[1]. 이러한 프로그램을 직접 만들려면 먼저 여러분이 매수한 주식의 주가가 매수 가격 대비 3% 떨어졌을 때의 가격을 알아야 합니다. 가령 네이버 주식 10주를 601,000원에 매수했다고 가정하면 3% 하락 시의 주당 가격은 "601,000 * 0.97"이 됩니다.

IDLE 프롬프트에 계산할 때 숫자와 곱하기 기호(*) 사이에는 공백이 있어도 되고, 없어도 됩니다. 즉, 601000*0.97이나 601000 * 0.97이나 같은 결과가 나옵니다. 다만 프로그램도 규모가 커지다 보면 여러 사람이 나눠서 작성하게 되는데, 이 경우 여러분이 작성한 프로그램의 소스코드(source code)를 다른 누군가도 보고 이해해야 합니다. 또는 여러분이 지금 작성한 코드를 한 달 후에 다시 여러분이 수정해야 하는 경우도 자주 발생합니다. 따라서 소스코드를 작성할 때 항상 가독성을 생각해서 적당한 공백(blank)을 넣는 것이 좋습니다. 다음 코드와 같이 곱하기 기호 좌우에 공백을 넣는 것이 훨씬 가독성이 좋지 않나요?

1 이러한 기능을 '스탑로스'라고 하며, 웬만한 증권사의 HTS는 이러한 기능을 기본적으로 제공합니다.

```
>>> 601000 * 0.97
582970.0
```

파이썬 IDLE에 나타난 결괏값을 확인하면 매수 가격(601,000원)에서 3% 하락한 금액은 582,970원입니다. 즉, 여러분이 만든 프로그램은 주식시장이 열려 있는 동안 주기적으로 현재가를 얻어온 후 현재가가 582,970보다 낮으면 바로 매도 주문을 넣으면 됩니다. 이를 글로 써보면 다음과 같습니다.

```
만약 현재가가 582970보다 낮으면
    보유 주식을 전량 매도
그렇지 않으면
    보유
```

물론 1장에서 배웠듯이 파이썬 프로그래밍은 영어로 된 키워드(keyword)를 사용하기 때문에 위와 같이 한글로는 명령을 내릴 수 없습니다. 다만 앞으로 파이썬 프로그래밍을 배워서 작성할 코드의 구성은 앞의 예제와 정확히 일치할 것입니다.

지금까지는 곱셈 연산만 해봤지만 파이썬은 덧셈, 뺄셈, 곱셈, 나눗셈, 즉 사칙연산을 지원합니다. 즉, 다음과 같은 코드도 얼마든지 작성할 수 있습니다.

```
>>> 601000 - 587000
14000
>>> 180 * 10 + 10000
11800
>>>
```

위 코드에서 첫 번째 구문은 네이버 주식의 매수 가격(601,000원)과 매도 가격(587,000원)을 뺀 것으로, 주당 손실액이 14,000원임을 알 수 있습니다. 두 번째 구문은 네이버와 관계없이 저자가 단순히 파이썬 IDLE를 이용해 "180 * 10 + 10000"을 계산한 것입니다.

여기서 한 가지 주의할 점이 있는데 산수와 마찬가지로 프로그래밍의 사칙 연산에도 연산자 우선순위가 있다는 것입니다. 즉, 덧셈, 뺄셈, 곱셈, 나눗셈이 동시에 사용되는 경우 곱셈, 나눗셈이 먼저 계산되고, 그다음으로 덧셈, 뺄셈이 계산됩니다. 물론 괄호가 있는 경우에는 괄호가 가장 먼저 실행됩니다. 따라서 "180 * 10"이 먼저 실행되고 그 결괏값에 10000이 더해지므로 최종값은 11800이 됩니다.

사칙 연산의 순서를 기억하기 어렵다면 파이썬으로 코드를 작성할 때 다음과 같이 괄호를 이용해 연산자의 우선순위를 명시적으로 나타내는 것이 좋습니다.

```
>>> (180 * 10) + 10000
11800
>>>
```

02 변수

2.1절에서는 파이썬 IDLE을 이용해 간단한 사칙 연산을 해봤습니다. 이번 절에서는 변수(variable)라는 것을 배울 텐데, 보통 프로그래밍 분야에서 변수라는 용어는 어떤 값을 저장하는 공간을 의미합니다. 변수는 프로그래밍에서 가장 기초적인 내용이기 때문에 변수의 개념을 잘 이해하는 것은 매우 중요합니다. 그렇다면 왜 변수라는 것이 필요할까요?

2.1절에서 네이버의 시가가 601,000원이라고 가정했습니다. 그런데 자꾸 계산할 때마다 이 값을 기억하고 있자니 참으로 귀찮고 어렵기도 합니다. 우리는 일상생활에서 어떤 것을 기억해야 할 때 종종 수첩에 기록해두고 수첩을 보면서 해당 내용을 떠올립니다. 이처럼 프로그래밍할 때도 어떤 값 자체를 기억하는 것이 아니라 어떤 이름을 통해 해당 값을 가리키게 해두고 싶습니다. 이 같은 상황에서 사용하는 것이 바로 '변수'입니다.

먼저 파이썬 IDLE를 실행한 후 프롬프트에 다음과 같이 입력하고 엔터 키를 눌러 봅시다.

```
>>> naver = 601000
```

위 코드의 의미는 그림 2.3처럼 'naver라는 이름(변수)이 601000이라는 값을 가리키게 하라'는 것입니다. 산수에서는 등호 표시(=)가 '같음'을 의미하지만 파이썬에서는 바인딩(binding)을 의미합니다. 참고로 binding이라는 영어 단어는 '가리키는 것'을 의미합니다.



그림 2.3 파이썬 바인딩의 의미

컴퓨터에서 처리되는 데이터는 메모리의 어딘가에 실제로 존재하게 됩니다. 즉, 위의 예에서 601000이라는 값 역시 메모리의 어딘가에 위치하는데, 그 위치를 naver라는 변수가 가리키고 있는 것입니다.

이번에는 변수의 개념을 일상생활과 관련 지어 좀 더 쉽게 설명해 보겠습니다. 여러분이 친구 집으로 편지를 보내는 상황을 생각해 봅시다. 편지를 보낼 때 '받는 주소' 부분에 친구의 주소를 기록해 두면 집배원은 그 주소를 통해 친구의 집을 정확히 찾아갈 수 있습니다.

파이썬의 변수도 마찬가지입니다. 어떤 데이터가 있을 때 그 데이터가 메모리 상에 위치하는 주소를 변수라는 곳에 저장해두고, 나중에 변수에 저장된 메모리 상의 주소에 가서 실제 값을 읽을 수 있는 것입니다.

아직도 변수가 잘 이해되지 않는 분들도 있겠지만 일단 우리에게 중요한 것은 이제 당분간은 네이버의 시가인 601,000원을 잊고 살아도 된다는 사실입니다. 해당 값을 알고 싶으면 프롬프트 (>>>)에 naver라고 입력하면 그림 2.4와 같이 값이 리턴(return)되기 때문입니다.

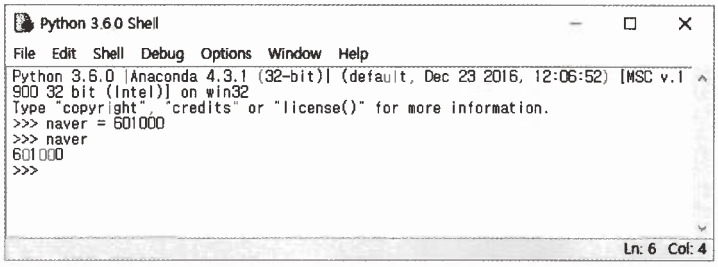


그림 2.4 변수를 통한 값 확인

아직도 변수의 편리함을 모르겠다는 분들은 파이썬 IDLE의 프롬프트에 다음과 같이 입력한 후 엔터 키를 눌러 보기 바랍니다. 이 코드는 네이버 주식 10주를 601,000원에 모두 매도했을 때의 매도 총액을 계산해줍니다. 어떤가요? 601,000원이라는 값을 직접 사용하는 것보다 변수를 사용하니 값 자체를 기억할 필요도 없고 매우 편리하죠?

```
>>> naver * 10
6010000
>>>
```

파이썬에서 변수를 사용할 때 한 가지 중요한 점은 변수의 이름을 잘 정해야 한다는 것입니다. 파이썬에서 변수란 어떤 값을 대신 기억하기 위해 값에 이름표를 붙여둔 것과 비슷한데, 값과 상관없는 이름표를 붙여두면 더 불편하기 때문입니다. 예를 들어, '콜라'에 '간장'이라는 이름표를 붙여 두고, '간장'에 '콜라'라는 이름을 붙여두면 나중에 둘을 구분하기가 매우 어려워지겠죠?

파이썬에서 변수의 이름을 정할 때는 몇 가지 규칙이 있습니다. 예를 들어, 변수의 첫 글자는 영어 또는 언더스코어(_)로 시작해야 하고, 숫자로는 시작할 수 없습니다. 즉, 그림 2.5와 같이 변수를 선언하면 오류가 발생하고 파이썬 코드가 실행되지 않습니다.

```
>>> inaver = 601000
SyntaxError: invalid syntax
>>>
```

그림 2.5 변수 선언 시 오류가 발생하는 예

참고로 앞에서 영어 또는 언더스코어로 시작해야 한다고 했지만 사실 한글로도 변수명을 만들 수는 있습니다. 다만 일반적으로 프로그래밍할 때는 한글 변수명보다는 영어 소문자와 언더스코어를 조합해서 변수명을 만드는 것이 좋습니다.

```
>>> 네이버 = 601000
>>> 네이버
601000
>>> naver_juga = 601000
>>> naver_juga
601000
>>>
```

파이썬 변수와 객체

보통 프로그래밍 언어를 배울 때 변수라는 개념을 맨 먼저 배우는데, 프로그래밍에 입문하는 많은 분들이 이 단계에서 쉽게 좌절하곤 합니다. 변수라는 개념 자체도 이해되지 않고 왜 사용하는지 모르기 때문에 프로그래밍에 대한 흥미를 잃어버리곤 합니다.

변수의 개념을 다시 한 번 강조하기 위해 간단한 예제를 들어 설명해 보겠습니다. 네이버의 월요일 주가가 10,000원에서 시작했는데, 월요일과 화요일에 연속해서 하한가(-30%)를 기록했을 때 화요일의 종가를 계산해 봅시다. 단 화요일의 '시작가'가 월요일의 종가와 같다고 가정합시다.

저자가 위 문제를 풀어본다면 아마 다음과 같이 월요일의 종가를 먼저 어딘가에 계산한 후 해당 가격을 이용해 다시 화요일에 하한가를 적용해 최종적으로 화요일의 종가를 계산할 것입니다.

```
월요일 종가 = 10,000 - (10,000 * 0.3) = 7,000
화요일 종가 = 7,000 - (7,000 * 0.3) = 4,900
```

일상생활에서 어떤 복잡한 값을 계산할 때 보통은 위와 같이 중간 과정의 결괏값을 어딘가에 기록해 두고, 조금씩 문제를 해결해 나갑니다. 마찬가지로 프로그램을 작성할 때도 의미가 있는 값을 저장해두거나 해당 값들을 가리키는 역할을 하는 것이 바로 변수입니다.

위에서 간단히 손으로 계산했던 것을 파이썬 코드로 옮기면 다음 코드와 같습니다. 해당 코드를 보면 저자가 변수의 이름을 정할 때 고심한 흔적이 보이나요? 이처럼 변수명이 조금 길더라도 나중에라도 해당 변수가 가리키는(바인딩하는) 값이 무엇인지 쉽게 알 수 있게 만드는 것이 매우 중요합니다.

```
>>> monday_closing_price = 10000 - (10000 * 0.3)
>>> tuesday_closing_price = monday_closing_price - (monday_closing_price * 0.3)
>>> tuesday_closing_price
4900.0
>>>
```

이제 변수의 필요성을 어느 정도 느끼셨을 테니 파이썬의 변수와 바인딩에 대해 좀 더 자세히 알아보겠습니다. 다음 코드를 파이썬 IDLE에서 실행해 보기 바랍니다.

```
>>> x = 100
>>>
```

파이썬 IDLE에서 위 코드를 실행했을 때 실제로는 어떤 일이 벌어질까요? 먼저 100이라는 값이 메모리의 어딘가에 할당되는데, 파이썬에서는 이를 객체(object)라고 부릅니다. 변수 x 역시 메모리의 어딘가에 할당되는 공간인데, 그 공간에는 100이라는 객체의 메모리 주솟값이 저장돼 있습니다. 이를 다르게 표현하자면 변수 x는 100이라는 값의 객체를 가리키고 있는 것입니다.

변수와 객체의 관계를 일상생활과 연관 지어 설명하자면 여러분이 살고 있는 집이 바로 객체입니다. 실제로 존재하는 것이죠. 이와 달리 주소 자체는 집은 아니지만 다른 누군가가 여러분 집의 주소를 통해 여러분이 실제로 사는 집에 갈 수 있습니다. 파이썬도 마찬가지입니다. 변수를 통해 실제 객체에 접근할 수 있는 것입니다.

위 코드에서 변수 x와 객체 100은 그림 2.6과 같은 관계를 맺습니다. 그림 2.6에서 객체는 네모 박스로 표시했고 변수는 동그라미로 표시했습니다. 변수는 "집 주소"로 생각하면 되고, 객체는 실제 여러분이 살고 있는 집으로 생각하면 좀 더 쉽게 이해할 수 있습니다.



그림 2.6 변수와 객체의 바인딩 예

그렇다면 파이썬에서 메모리에 할당된 객체의 주소(정확히 말하면 주소는 아니지만 의미상으로는 주소로 생각해도 됩니다)는 어떻게 확인할 수 있을까요? 다음과 같이 id라는 함수를 사용하면 됩니다.

```
>>> x = 100
>>> id(x)
```

```
1773787088
>>>
```

여기까지 이해하신 분들은 다음 코드에서 id(x)와 id(y) 값이 같을지 또는 다를지 생각해보기 바랍니다.

```
>>> x = 100
>>> y = 100
>>> id(x), id(y)
```

예제를 보면 먼저 메모리에 100이라는 값이 할당되고, 그 주소를 x가 가리킵니다. 그리고 다음 줄을 보면 또 100이 나오는데, 이때 다음과 같이 두 가지 방식이 있을 것입니다.

1. 메모리에 조금 전의 100과 다른 위치에 100을 할당하고, 그 주솟값을 y가 가리킨다(메모리에 두 개의 '100'이 존재).
2. 메모리가 아까우니 조금 전의 100의 주소를 y가 가리키게 한다(메모리에 100은 실제로 하나만 존재).

위 코드를 실행한 후 결괏값을 보면 id 값이 같습니다. 즉, 두 변수가 서로 같은 객체를 가리키고 있음을 확인할 수 있습니다. 이를 그림으로 나타내면 그림 2.7과 같습니다. 파이썬은 위 코드에 대해 2번과 같은 방식으로 동작했던 것입니다.

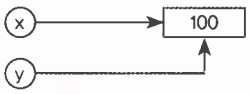


그림 2.7 변수와 객체의 바인딩 예(1)

단, 파이썬이 항상 2번 방식으로 동작하는 것은 아닙니다. 다음 코드를 실행해 보면 id 값이 서로 다르게 나옵니다.

```
>>> x = 10000
>>> y = 10000
>>> id(x), id(y)
(44212048, 48072352)
>>>
```

위 코드에서 변수와 객체의 관계를 그림으로 나타내면 그림 2.8과 같습니다. 변수 x와 변수 y는 서로 다른 객체를 가리키고 있습니다. 그림 2.8에서 파란색으로 나타낸 부분은 객체의 id 값을 의미합니다. 참고로 위 코드를 실행했을 때 반환되는 id 값은 실행할 때마다 달라질 수 있으므로 이 책에 나온 값과 여러분이 확인한 값이 서로 다를 수 있습니다.

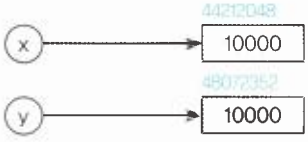


그림 2.8 변수와 객체의 바인딩 예(2)

파이썬이 이렇게 동작하는 이유는 프로그램을 작성할 때 정숫값(Integer) 중 자주 사용할 것 같은 범위의 정숫값은 메모리에 한 번만 올려두고 이를 여러 변수가 가리키게 함으로써 메모리를 효과적으로 사용하기 위해서입니다. 심심하신 분들은 파이썬 IDLE를 통해 위와 같은 방식으로 숫자에 대해 x, y라는 변수로 바인딩해보면 256까지는 id 값이 같지만 257부터는 서로 다른 객체가 생성되는 것을 확인할 수 있습니다.

03 문자열

2.2절에서 변수를 이용해 메모리에 할당된 어떤 정숫값을 가리키는 것(바인딩)을 배웠습니다. 이번 절에서는 변수로 문자열을 가리키는 것에 대해 알아보겠습니다.

C/C++ 같은 프로그래밍 언어에서는 'a', 'b', 'c'와 같은 알파벳 글자 하나를 '문자'라고 부르며, 'house'와 같이 두 개 이상의 문자로 구성된 것을 '문자열(String)'이라고 합니다. 다른 프로그래밍 언어와 달리 파이썬은 문자와 문자열을 구분하지 않고 작은따옴표(')나 큰따옴표("")로 묶인 문자의 모음을 문자열이라고 부릅니다.

예를 들어, 다음의 세 구문은 모두 변수가 파이썬 문자열을 바인딩하는 예입니다. 앞서 설명한 것처럼 파이썬은 문자와 문자열을 구분하지 않고 모두 문자열로 취급합니다. 이때 한 가지 주의해야 할 점은 문자열을 작은따옴표로 시작한 경우에는 반드시 작은따옴표로 끝나야 하고, 큰따옴표로 시작한 경우에는 반드시 큰따옴표로 끝나야 한다는 것입니다. 즉, 문자열의 시작과 끝에 서로 다른 종류의 따옴표를 사용할 수 없습니다.

```
>>> mystring  = 'hello world'
>>> mystring1 = 'a'
>>> mystring2 = "a"
>>> mystring3 = "abc mart"
```

위 코드처럼 변수가 문자열을 가리키고 있으면 변수명을 통해 해당 문자열을 참조할 수 있습니다. 일단 mystring이라는 변수가 문자열 객체를 잘 가리키고 있는지 확인하기 위해 파이썬 프롬프트에 mystring을 타이핑하고 엔터 키를 눌러봅시다.

```
>>> mystring
'hello world'
>>>
```

'hello world'라고 정상적으로 값이 나오는 것을 확인할 수 있습니다. 이번에는 화면에 값을 출력하는 함수인 print를 통해 mystring이 가리키는 값을 출력해 보겠습니다.

```
>>> print(mystring)
hello world
>>>
```

문자열 인덱싱 및 슬라이싱

파이썬은 다른 프로그래밍 언어에 비해 문자열을 매우 쉽게 다룰 수 있습니다. 먼저 'hello world'라는 문자열에 대해 총 몇 글자가 있는지 알아보겠습니다.

다음 코드를 실행하면 11이라는 값이 반환되는데, 이는 'hello world'라는 문자열이 총 11개의 글자로 구성돼 있다는 의미입니다(참고로 len 함수는 length의 줄임말입니다). 그런데 왜 10자가 아니라 11자일까요? 그것은 'hello'와 'world'라는 단어 사이에 있는 공백도 하나의 문자로 간주하기 때문입니다.

```
>>> mystring  = 'hello world'
>>> len(mystring)
11
>>>
```

'hello world'라는 문자열 중에서 'hello'라는 글자만 구하고 싶다면 어떻게 하면 될까요? 파이썬에서는 '슬라이싱(slice)'이라는 기능을 제공하는데, 다음과 같이 가져오고 싶은 문자열의 범위를 지정하면 됩니다. 참고로 slice라는 영어 단어는 '자르다', '일부', '한 조각'과 같은 의미가 있습니다.

```
>>> mystring[0:5]
'hello'
>>>
```

위 코드를 보면 [0:5]라는 표현을 사용했습니다. [ ]는 슬라이싱할 범위를 지정할 때 사용하는 기호이며, 0은 시작 위치를, 5는 끝 위치를 의미합니다. 시작과 끝을 구분하기 위해 그 사이에 콜론(:)을 사용합니다.

참고로 프로그래밍 언어에서는 우리의 일상생활과 달리 범위가 1부터 시작하는 것이 아니라 0부터 시작합니다. 그래서 위 예제에서 시작 값이 1이 아니라 0으로 돼 있는 것입니다. 그림 2.9는 'hello world'라는 문자열의 각 글자 사이에 인덱스를 붙여본 것입니다.



그림 2.9 문자열 인덱싱 예

파이썬 문자열의 슬라이싱을 조금 더 응용해 보겠습니다. 이번에는 mystring이라는 변수를 통해 'world'만 가져오려면 어떻게 해야 할까요? 정답은 다음과 같습니다.

```
>>> mystring[6:11]
'world'
>>>
```

위 코드의 문제점은 맨 마지막 글자의 위치를 세기가 참으로 귀찮다는 것입니다. 다행히도 파이썬의 슬라이싱에서는 시작값 또는 끝값을 생략하면 알아서 해당 문자열의 시작과 끝을 의미합니다. 즉, 다음과 같이 작성할 수 있습니다.

```
>>> mystring[6:]
'world'
>>>
```

hello를 구하는 코드도 시작값을 생략해서 다음과 같이 작성할 수 있습니다.

```
>>> mystring[:5]
'hello'
>>>
```

파이썬의 문자열 인덱싱에서 인덱스 값으로 양수만 사용할 수 있는 것은 아니고 그림 2.10에서처럼 음수를 사용할 수도 있습니다. 인덱스 값이 음수인 경우에는 문자열의 뒤쪽부터 역순으로 글자를 셉니다.



그림 2.10 음수 인덱싱 예

음수 인덱싱을 응용하면 다음 코드처럼 문자열에 접근할 수 있습니다. 그런데 아쉽게도 'world'에서 'd'가 빠져버렸네요.

```
>>> mystring[6:-1]
'worl'
>>>
```

문자열 자르기

저자는 몇 년 전에 네이버와 다음 주식을 가지고 있었습니다. 그래서 my_jusik이라는 변수가 "naver daum"이라는 문자열 객체를 가리키게 해 봤습니다(중간에 공백이 하나 있습니다).

```
>>> my_jusik = "naver daum"
>>>
```

파이썬 IDLE에 my_jusik이라고 입력한 후 엔터 키를 눌렀더니 'naver daum'이라고 문자열이 잘 반환됩니다. 여기서 한 가지 팁을 드리면 파이썬 IDLE에서 my_jusik을 모두 입력하지 않고 'my_' 정도만 입력한 상태에서 키보드의 탭(Tab) 키를 누르면 자동으로 변수명이 완성됩니다. 이러한 자동 완성 기능은 변수명이 긴 경우에 유용하게 활용할 수 있습니다.

```
>>> my_jusik
'naver daum'
>>>
```

my_jusik이라는 변수가 가리키는 문자열 객체에서 'naver'만 출력하려면 어떻게 해야 할까요? 즉, naver와 daum이 현재는 하나의 문자열인데 이를 분리하고 싶은 것입니다. 2.3.1절에서 배운 문자열 슬라이싱으로도 문자열을 분리할 수 있지만 파이썬에서는 문자열을 쉽게 분리할 수 있게 split이라는 메서드(method)[2]를 지원합니다. 참고로 split이라는 단어는 '나누다', '쪼개다'라는 뜻이 있습니다.

```
>>> my_jusik.split(' ')
['naver', 'daum']
>>>
```

파이썬에서 문자열을 분리하고 싶을 때는 위 코드와 같이 split 메서드를 사용하면 됩니다. 이때 메서드의 인자(argument)로 어떤 문자를 기준으로 문자열을 나눌지 알려줘야 합니다. 'naver'와 'daum'이라는 문자열 사이에는 공백(blank)이 있으므로 공백을 기준으로 문자열을 나누면 됩니다. 이를 위해 split(' ')과 같이 작은따옴표 사이에 공백을 하나 넣었습니다.

2 메서드는 함수와 유사한데 자세한 내용은 6장에서 다룹니다.

그러나 위 코드에서 반환값을 보면 여전히 우리가 원하는 값은 아닙니다. 반환값을 보면 'naver'와 'daum'이라는 두 개의 문자열이 '['와 ']'로 둘러싸여 있음을 확인할 수 있습니다. 파이썬에서는 이것을 리스트(list)라고 합니다(리스트에 대해서는 3장에서 다룹니다).

이번 장에서는 리스트 내의 값에 접근할 때 [0], [1], [2]와 같이 인덱스 값을 주면 된다는 것만 기억하면 됩니다. 리스트의 인덱싱 기능을 이용해 다음과 같이 코드를 수정해 보겠습니다. split(' ')의 반환값이 리스트이므로 [0]을 사용해 리스트의 첫 번째 값을 가져왔습니다.

```
>>> my_jusik.split(' ')[0]
'naver'
>>>
```

화면에 출력하는 함수는 print였죠? 이번에는 출력 함수까지 쭉 붙여서 써보겠습니다.

```
>>> print(my_jusik.split(' ')[0])
naver
>>>
```

위 코드에서는 한 번에 붙여서 썼더니 가독성이 조금 떨어진 것 같습니다. 다음과 같이 변수를 이용해 중간값을 바인딩해서 좀 더 이해하기 쉽게 코드를 작성해 봅시다.

```
>>> split_jusik = my_jusik.split(' ')
>>> split_jusik
['naver', 'daum']
>>> print(split_jusik[0])
naver
>>>
```

문자열 합치기

2014년 6월 카카오(Kakao)와 다음(Daum)이 합병했습니다. 그 당시 합병된 회사의 이름은 여러분도 잘 아시다시피 'Daum KaKao'였습니다. 이를 파이썬 문자열로는 어떻게 표현할 수 있을까요?

먼저 daum이라는 변수가 'Daum'이라는 문자열 객체를 가리키게 해봅시다. 그리고 kakao라는 변수는 'KAKAO'라는 문자열 객체를 가리키게 해봅시다.

```
>>> daum = "Daum"
>>> kakao = "KAKAO"
>>>
```

이제 두 변수를 더해봅시다. 정수를 더할 때 사용하는 '+' 기호를 그대로 사용해 다음과 같이 작성하면 됩니다.

```
>>> daum + kakao
'DaumKAKAO'
>>>
```

'DaumKAKAO'라는 이름 사이에 공백이 없으니 뭔가 허전합니다. 읽기도 어렵고 가짜 회사가 된 것 같습니다. 문자열을 더할 때 두 문자열 사이에 공백을 하나 넣어보겠습니다. 문자열을 합치는 것은 그냥 숫자를 더하듯이 여러 문자열을 그냥 + 기호로 더하면 됩니다. 공백을 나타내고 싶으면 작은따옴표 사이에 공백을 넣어서 표현하면 됩니다.

```
>>> daum + ' ' + kakao
'Daum KAKAO'
>>>
```

이름을 변수로 가리켜 두지 않으면 'Daum KAKAO'라는 문자열 객체를 사용할 때마다 두 문자열을 합치는 귀찮은 작업을 해야 합니다. 다음과 같이 합쳐진 문자열을 다른 변수가 가리키게 해줍니다.

```
>>> daum_kakao = daum + ' ' + kakao
>>> daum_kakao
'Daum KAKAO'
>>>
```

이제 앞으로는 'Daum KAKAO'라는 이름을 출력할 때 daum_kakao라는 변수를 사용하면 됩니다.

04 기본 데이터 타입

지금까지 정수, 실수, 문자열에 대해 배웠습니다. 이러한 정수, 실수, 문자열은 파이썬 프로그래밍에서 자주 사용되는데, 이를 기본 데이터 타입이라고 합니다.

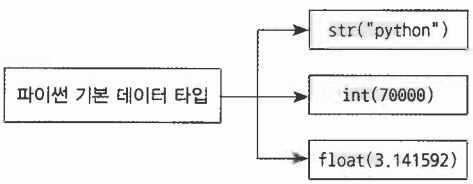


그림 2.11 파이썬 기본 데이터 타입

파이썬에서 어떤 값(객체)에 대한 데이터 타입을 확인하려면 type이라는 내장 함수를 사용하면 됩니다. 다음 코드를 파이썬 IDLE를 통해 실행해보기 바랍니다.

```
>>> type(70000)
<class 'int'>
>>>
```

위 코드의 실행 결과를 살펴보면 type 함수의 반환값이 <class 'int'>임을 알 수 있습니다. 여기서 class라는 단어는 잠시 잊고, int에 주목하기 바랍니다. int는 integer의 약자로서 정수를 의미합니다. 참고로 산수가 약하신 분들을 위해 설명해 드리면 1, 2, 3, … 등을 정수라고 하고, 1.234, 1.432 등을 실수라고 합니다. 즉, 우리가 알고 있는 정수 70000에 대해 type 함수를 이용해 파이썬에게 타입을 물어봤더니 파이썬이 'int'라고 답한 것입니다.

파이썬에는 정숫값 말고도 실숫값도 있으니 이 경우에도 type 함수가 잘 동작하는지 확인해 보겠습니다. 3.141592라는 값에 대한 type 함수의 반환값을 보니 'float'입니다. float의 사전적 의미는 '뜨는 것', '둥둥 떠다니는 것'을 의미합니다. 프로그래밍에서 float는 부동 소수점(실수)을 의미합니다.

```
>>> type(3.141592)
<class 'float'>
>>>
```

파이썬의 기본 데이터 타입으로 정수와 실수 외에도 문자열이 있죠? 문자열에 대해서도 타입을 확인해 보겠습니다.

```
>>> type('python')
<class 'str'>
>>> type('jusik')
<class 'str'>
>>>
```

앞에서 배운 것처럼 작은따옴표나 큰따옴표로 둘러싸인 값에 대해 'str'이라는 값이 반환되는 것을 볼 수 있습니다. 참고로 str은 영어 단어에서 문자열을 의미하는 'string'에서 앞 세 글자를 딴 것입니다.

연습 문제

문제 2-1

다음(Daum)의 주가가 89,000원이고 네이버(Naver)의 주가가 751,000원이라고 가정하고, 어떤 사람이 다음 주식 100주와 네이버 주식 20주를 가지고 있을 때 그 사람이 가지고 있는 주식의 총액을 계산하는 프로그램을 작성하세요.

문제 2-2

문제 2-1에서 구한 주식 총액에서 다음과 네이버의 주가가 각각 5%, 10% 하락한 경우에 손실액을 구하는 프로그램을 작성하세요.

문제 2-3

우리나라는 섭씨 온도를 사용하는 반면 미국과 유럽은 화씨 온도를 주로 사용합니다. 화씨 온도(F)를 섭씨 온도(C)로 변환할 때는 다음과 같은 공식을 사용합니다. 이 공식을 사용해 화씨 온도가 50일 때의 섭씨 온도를 계산해 보세요.

```
C = (F-32)/1.8
```

문제 2-4

화면에 "pizza"를 10번 출력하는 프로그램을 작성하세요.

문제 2-5

월요일에 네이버의 주가가 100만 원으로 시작해 3일 연속으로 하한가(-30%)를 기록했을 때 수요일의 종가를 계산해 보세요.

문제 2-6

다음 형식과 같이 이름, 생년월일, 주민등록번호를 출력하는 프로그램을 작성해 보세요.

이름: 파이썬 생년월일: 2014년 12월 12일 주민등록번호: 20141212-1623210

문제 2-7

s라는 변수에 'Daum KaKao'라는 문자열이 바인딩돼 있다고 했을 때 문자열의 슬라이싱 기능과 연결하기를 이용해 s의 값을 'KaKao Daum'으로 변경해 보세요.

문제 2-8

a라는 변수에 'hello world'라는 문자열이 바인딩돼 있다고 했을 때 a의 값을 'hi world'로 변경해 보세요.

문제 2-9

x라는 변수에 'abcdef'라는 문자열이 바인딩돼 있다고 했을 때 x의 값을 'bcdefa'로 변경해 보세요.

03

기본 자료구조

2장에서는 파이썬의 기본 데이터 타입인 정수형, 실수형, 문자열을 배웠습니다. 이번 장에서는 정수형, 실수형, 문자열 등의 데이터가 여러 개 있을 때 이를 효과적으로 관리하는 데 사용되는 자료구조에 대해 배우겠습니다. 파이썬에는 여러 가지 자료구조가 있는데, 그중 가장 많이 사용되는 리스트(list), 튜플(tuple), 사전(dict)에 대해 다룹니다. 이러한 세 가지 자료구조는 파이썬으로 프로그래밍할 때 자주 사용되므로 반드시 잘 익혀두시기 바랍니다.

01 리스트

유가증권(코스피)과 코스닥을 합치면 대략 2000 종목 정도가 있는데 그중에서 관심 있는 종목이 삼성전자, LG전자, 네이버라고 가정해 봅시다. HTS(Home Trading System)에 관심 종목이 나오는 것처럼 관심 종목을 파이썬으로 관리해야 한다면 어떻게 하면 될까요? 아마도 2장에서 배운 변수를 이용해 다음과 같이 관심 종목명을 문자열로 표현한 후 가리키게 할 수 있을 것입니다.

```
>>> interest1 = "삼성전자"
>>> interest2 = "LG전자"
>>> interest3 = "네이버"
>>>
```

2장에서 설명했듯이 파이썬의 변수는 C/C++ 같은 프로그래밍 언어와 달리 실제 데이터가 저장되는 공간 자체가 아니라 데이터가 메모리상에서 위치하는 주솟값이 저장되는 공간입니다. 파이썬에서는 그림 3.1과 같이 변수가 실제 데이터가 존재하는 위치를 가리키고 있는 것을 바인딩(binding)한다고 표현합니다. 바인딩, 바인딩, 바인딩 ... 딱 10번만 반복해봅시다. 지금은 '가리킨다'라는 표현과 '바인딩'이라는 표현을 의도적으로 섞어서 사용하고 있는데 이 책의 후반부로 갈수록 '바인딩'이라는 표현을 주로 사용할 것입니다.

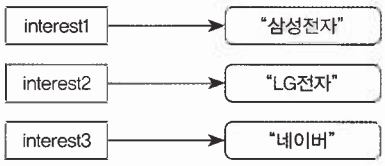


그림 3.1 파이썬 바인딩의 예

파이썬에서 변수가 어떤 값을 바인딩하고 있으면(가리키고 있으면), 프롬프트 상에서 변수명을 입력한 후 엔터 키를 누르면 해당 변수가 가리키는 값이 반환됩니다. 즉, 다음과 같이 interest1을 입력한 후 엔터 키를 누르면 interest1 변수가 가리키고 있던 문자열 데이터인 '삼성전자'가 반환됩니다.

```
>>> interest1
'삼성전자'
>>>
```

위와 같이 관심 종목의 수가 적은 경우에는 각자 서로 다른 변수명을 사용해서 관심 종목을 바인딩하고 있을 수 있지만 그래도 뭔가 불편해 보입니다. 지금과 같은 방식에서는 관심 종목의 수가 100개라면 이를 표현하기 위해 interest1, interest2, interest3, ···, interest100과 같이 100개의 변수명을 사용해야 할 것입니다.

이러한 불편을 해결하고자 파이썬에서는 리스트(list)라는 기본 자료구조를 제공합니다. 리스트를 이용하면 여러 데이터를 한 번에 저장할 수 있으므로 변수명을 여러 개 사용할 필요가 없습니다. 여러 개의 데이터를 리스트에 담으려면 '['와 ']' 사이에 데이터를 쉼표로 구분해서 넣기만 하면 됩니다. 예를 들어, 앞의 예제와 같이 삼성전자, LG전자, 네이버를 관심 종목이라고 할 때 이를 리스트로 만들려면 다음과 같이 작성하면 됩니다.

```
>>> interest = ["삼성전자", "LG전자", "네이버"]
```

파이썬의 리스트를 이용하면 여러 데이터를 한 번에 가리킬 수 있으므로 interest1, interest2, interest3과 같이 각각 변수명을 사용하는 것이 아니라 interest처럼 한 개의 변수만 있으면 됩니다. 파이썬에서 여러 데이터를 리스트를 통해 한 번에 저장하고 관리하는 것은 그림 3.2처럼 출석부에 학생들의 이름을 적어 두는 것과 비슷한 개념입니다.

출석부(코스피/코스닥)

번호	학년/반	성명	1	2	3	4	5	6	7
1		삼성전자							
2		LG전자							
3		네이버							
4									
5									

그림 3.2 파이썬 리스트와 출석부

학창 시절에 선생님께서 출석부를 보시고 1번이라고 호명하면 1번 학생이 대답했던 것처럼 파이썬에서도 리스트를 통해 바인딩하고 있는 데이터는 리스트의 인덱스 값을 통해 접근할 수 있습니다. 단, 출석부의 번호가 1번부터 시작했던 것과 달리 파이썬 리스트의 인덱스는 0번부터 시작하기 때문에 이를 고려해야 합니다. 즉, 리스트의 첫 번째 데이터에 접근하려면 1이 아니라 0을 사용하면 됩니다.

다음 코드는 [0], [1], [2]라는 인덱스 값을 통해 interest 리스트에 있는 첫 번째, 두 번째, 세 번째 관심 종목 데이터를 차례대로 얻어옵니다.

```
>>> interest = ["삼성전자", "LG전자", "네이버"]
>>> interest[0]
'삼성전자'
>>> interest[1]
'LG전자'
>>> interest[2]
'네이버'
>>>
```

리스트 생성

3.1절에서 파이썬 기본 자료구조 중 하나인 리스트에 대해 간단히 살펴봤습니다. 그렇다면 리스트는 언제 사용하는 것이 좋을까요? 바로 여러 개의 데이터를 '순서대로' 저장하고 이를 관리해야 할 때입니다.

주식 분석 프로그램을 만드는데 '대신증권'이라는 종목의 최근 5일 치의 종가인 9130, 9150, 9150, 9300, 9400원을 저장해야 하는 경우를 생각해 봅시다. 이 경우 다음과 같이 파이썬의 리스트를 이용하면 종가를 '순서대로' 저장할 수 있습니다.

```
>>> daishin = [9130, 9150, 9150, 9300, 9400]
>>>
```

파이썬의 리스트에 정수 데이터만 넣을 수 있는 것은 아닙니다. 다음과 같이 실수 데이터로도 리스트를 생성할 수 있습니다.

```
>>> daishin = [9130.0, 9150.0, 9150.0, 9300.0, 9400.0]
>>>
```

물론 앞서 살펴본 것처럼 문자열로도 리스트를 만들 수 있습니다. 이뿐만 아니라 서로 다른 자료형을 동시에 넣을 수도 있습니다. 네이버 주식을 5,000주 가지고 있어서 이를 리스트로 표현한다면 다음과 같이 작성할 수 있습니다.

```
>>> mystock = ['Naver', 5000]
>>>
```

파이썬의 리스트를 만들 때 항상 데이터가 있어야 하는 것은 아닙니다. 다음과 같이 리스트를 만들 때 '['와 ']' 기호 사이에 데이터를 넣지 않으면 아무것도 들어 있지 않은 빈 리스트가 만들어집니다.

```
>>> mystock = []
>>>
```

리스트 인덱싱

파이썬 리스트의 인덱스 값을 통해 리스트가 가리키는 값에 접근할 수 있습니다. 리스트의 인덱스 역시 문자열 인덱스처럼 0부터 시작합니다.

다음 코드는 'Naver'라는 종목의 주식을 5,000주 매수하기 전에 매수할 종목명과 매수할 주식 수를 리스트로 표현한 것입니다. 리스트는 데이터를 순서대로 저장하므로 리스트의 첫 번째 항목에는 종목명이 있고, 두 번째 항목에는 매수할 주식 수가 있습니다. 따라서 buy_list[0]이라는 표현을 통해 리스트의 첫 번째 항목으로 저장된 'Naver'라는 문자열 데이터를 가져올 수 있습니다.

```
>>> buy_list = ['Naver', 5000]
>>> buy_list[0]
'Naver'
>>> buy_list[1]
5000
>>>
```

마찬가지로 대신증권의 최근 5일 치 종가가 담긴 리스트를 만든 후 인덱싱을 통해 각 종가를 가져올 수 있습니다. 파이썬의 인덱싱은 0부터 시작하기 때문에 daishin이라는 리스트에 있는 첫 번째 데이터인 9130을 가져오려면 daishin[0]이라고 입력하면 됩니다.

```
>>> daishin = [9130, 9150, 9150, 9300, 9400]
>>> daishin[0]
9130
>>> daishin[1]
9150
>>> daishin[2]
9150
>>> daishin[3]
9300
>>> daishin[4]
9400
>>>
```

지금까지 살펴본 것처럼 파이썬 리스트의 인덱싱은 문자열 인덱싱과 동일합니다. 2장의 문자열 인덱싱에서 배운 것처럼 인덱싱 값으로 음수를 사용하면 리스트의 뒤쪽부터 데이터에 접근할 수 있습니다. 단, 음수를 사용할 때는 -1부터 사용할 수 있습니다. 왜냐하면 0은 이미 리스트의 맨 앞에 있는 데이터를 의미하기 때문입니다.

리스트를 인덱싱할 때 [-1]은 리스트의 데이터 중 맨 마지막에 있는 데이터를 의미합니다. 마찬가지로 [-2]는 리스트의 데이터 중 끝에서 두 번째 항목을 의미합니다. 다음 코드에서 대신증권 데이터를 예로 들어 설명하면 daishin[-1]의 값은 9400으로 리스트에 맨 마지막에 위치하는 데이터임을 알 수 있습니다.

```
>>> daishin = [9130, 9150, 9150, 9300, 9400]
>>> daishin[-1]
9400
>>> daishin[-2]
9300
>>> daishin[-3]
```

```
9150
>>> daishin[-4]
9150
>>> daishin[-5]
9130
>>>
```

위 코드에서 daishin 리스트에 대한 인덱싱을 그림으로 표현하면 그림 3.3과 같이 나타낼 수 있습니다. 인덱싱할 때 언제 양수 인덱싱을 사용하고 음수 인덱싱을 사용할지에 대한 규칙이나 제약은 없습니다. 프로그래밍할 때 필요에 따라 적절하게 사용하면 되는 것입니다.

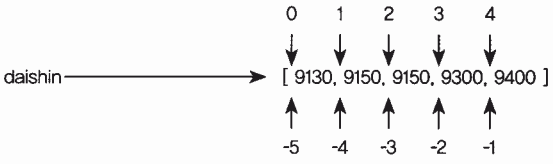


그림 3.3 파이썬 리스트 인덱싱

리스트 슬라이싱

파이썬 리스트에 있는 데이터에 하나씩 접근할 때는 인덱싱을 사용하면 됩니다. 그런데 리스트에 있는 여러 개의 데이터에 동시에 접근하려면 어떻게 해야 할까요? 이럴 때 사용하는 것이 바로 파이썬 슬라이싱입니다. 파이썬 리스트의 슬라이싱은 문자열 슬라이싱과 동일합니다.

잠깐 주식 시장 이야기를 해보면, 2015년 6월 28일 기준으로 코스피 시가총액 상위 10종목은 표 3.1과 같습니다. 코스피 시가총액 상위 10종목은 순서가 있는 데이터이기 때문에 이를 파이썬으로 표현하려면 자료구조 중 리스트를 사용하는 것이 좋습니다. 다만 표 3.1의 데이터는 순위와 종목명으로 구성돼 있지만 파이썬의 리스트에는 이미 각 인덱스 자체가 순서를 의미하므로 순위 값은 저장할 필요 없이 1순위부터 10순위까지의 종목명을 리스트로 저장하면 됩니다.

표 3.1 코스피 시가총액 10위 목록(2015년 6월 28일 기준)

순위	종목명
1	삼성전자
2	SK하이닉스
3	현대차
4	한국전력
5	아모레퍼시픽

순위	종목명
6	제일모직
7	삼성전자우
8	삼성생명
9	NAVER
10	현대모비스

표 3.1을 파이썬 리스트로 표현하면 다음과 같습니다.

```
>>> kospi_top10 = ['삼성전자', 'SK하이닉스', '현대차', '한국전력', '아모레퍼시픽', '제일모직', '
삼성전자우', '삼성생명', 'NAVER', '현대모비스']
```

앞에서 배운 파이썬 인덱싱을 사용해 코스피 기준 시가총액 5위를 출력하는 프로그램을 작성하면 다음과 같습니다. 여기서 한 가지 주의할 점은 파이썬의 인덱스는 0부터 시작하기 때문에 kospi_top10[4]가 리스트 내에서 5번째에 위치하는 데이터라는 것입니다.

```
>>> print("시가총액 5위: ", kospi_top10[4])
시가총액 5위:  아모레퍼시픽
>>>
```

현재 kospi_top10이라는 리스트에는 10개의 종목이 있는데, 만약 코스피 상위 5종목만으로 구성된 새로운 리스트를 만들고 싶다면 어떻게 할까요? 다음과 같이 리스트의 슬라이싱을 사용하면 새로운 리스트를 손쉽게 만들 수 있습니다.

```
>>> kospi_top5 = kospi_top10[0:5]
>>> kospi_top5
['삼성전자', 'SK하이닉스', '현대차', '한국전력', '아모레퍼시픽']
>>>
```

슬라이싱은 "[시작 인덱스: 끝 인덱스]"와 같은 형태로 표현하면 되는데, 위 코드에서는 0이 시작 인덱스이고 5가 끝 인덱스입니다. 따라서 kospi_top10[0:5]는 kospi_top10이라는 리스트에서 0번 위치에서부터 5번 위치까지를 가져오라는 의미가 됩니다. 그림 3.4와 같이 kospi_top10이라는 이름의 리스트에 앞에서부터 인덱스를 붙여보면 [0:5]라는 슬라이싱의 의미를 더 쉽게 파악할 수 있습니다.

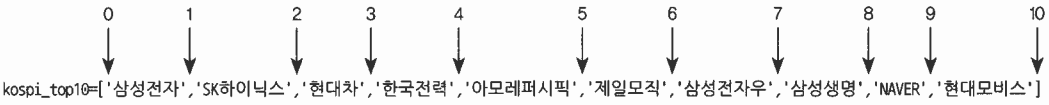


그림 3.4 리스트 인덱싱의 의미

지금부터는 배운 내용을 조금 응용해 보겠습니다. 이번에는 코스피 시가총액 기준으로 6위부터 10위에 대한 목록을 가져와 보겠습니다. 먼저 6위를 가져올 때 파이썬 인덱스가 0부터 시작하기 때문에 [5]라는 인덱싱을 사용해야 합니다. 그리고 "끝 인덱스" 값은 9가 아니라 10이어야 합니다. 그림 3.4의 인덱스를 참고하면 9를 사용하는 경우 'NAVER'까지만 가져오는 것을 확인할 수 있습니다.

```
>>> kospi_top10[5:10]
['제일모직', '삼성전자우', '삼성생명', 'NAVER', '현대모비스']
>>>
```

위 코드에서 한 가지 불편한 점은 슬라이싱할 때 리스트의 끝을 알려주기 위해서는 리스트 안에 있는 데이터의 개수를 미리 알고 있어야 한다는 점입니다. 위의 예제에서는 리스트 안에 총 10개의 종목이 있었기 때문에 예제에서는 10을 사용했습니다.

파이썬 리스트도 문자열과 마찬가지로 시작 인덱스나 끝 인덱스를 생략할 수 있습니다. 예를 들어, 다음 코드는 6위부터 10위에 대한 목록을 가져옵니다. 끝 인덱스에 해당하는 값에 아무 값도 넣지 않으면 자동으로 리스트의 끝을 알아서 인덱싱해줍니다.

```
>>> kospi_top10[5:]
['제일모직', '삼성전자우', '삼성생명', 'NAVER', '현대모비스']
>>>
```

이번에는 '시작 인덱스'에 해당하는 부분을 생략해 보겠습니다. 다음 코드를 참고하면 시작 인덱스에 해당하는 부분에 값을 생략하면 자동으로 리스트의 시작부터 값을 가져오는 것을 확인할 수 있습니다.

```
>>> kospi_top10[:5]
['삼성전자', 'SK하이닉스', '현대차', '한국전력', '아모레퍼시픽']
>>>
```

앞에서 파이썬 리스트의 인덱싱에서 음수를 사용하는 경우를 살펴봤습니다. 마찬가지로 슬라이싱에서도 음수를 사용할 수 있습니다. 예를 들어, 6위부터 9위까지의 종목을 가져와야 하는 경우에는 다음과 같이 하면 됩니다. 다음 코드를 보면 [5:9]라고 슬라이싱하는 것과 [5:-1]이라고 슬라이싱하는 것이 동일한 결과를 보여주는 것을 알 수 있습니다.

```
>>> kospi_top10[5:9]
['제일모직', '삼성전자우', '삼성생명', 'NAVER']
>>> kospi_top10[5:-1]
['제일모직', '삼성전자우', '삼성생명', 'NAVER']
>>>
```

[5:-1] 슬라이싱의 의미가 잘 이해되지 않는 분들은 그림 3.5를 참조하면 쉽게 이해할 수 있을 것입니다. 리스트 인덱싱에서 -1이라는 값은 리스트에 있는 데이터 중 맨 마지막 데이터의 앞쪽을 가리키는 인덱스 값입니다. 따라서 kospi_top10[5:-1]은 ['제일모직', '삼성전자우', '삼성생명', 'NAVER']의 리스트를 반환합니다.

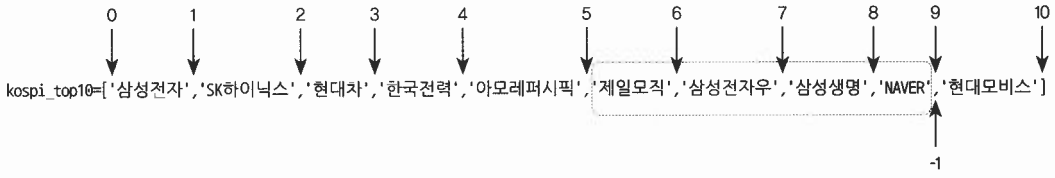


그림 3.5 리스트 인덱싱의 의미(2)

리스트에 데이터 삽입하기

기존의 리스트에 새로운 데이터를 삽입하려면 append 메서드를 사용하면 됩니다. 다음 코드와 같이 코스피 상위 10종목에 대한 리스트가 구성된 상태에서 해당 리스트에 코스피 11번 순위인 'SK텔레콤'을 추가해야 하는 경우를 생각해 봅시다. 이러한 상황에서 새로운 데이터를 리스트로 삽입할 때 사용하는 메서드가 바로 append입니다.

```
>>> kospi_top10 = ['삼성전자', 'SK하이닉스', '현대차', '한국전력', '아모레퍼시픽', '제일모직', '
삼성전자우', '삼성생명', 'NAVER', '현대모비스']
```

리스트에서 append 메서드를 사용하려면 다음 코드와 같이 먼저 리스트 이름을 적은 후 마침표를 붙이고 그다음에 메서드 이름을 적으면 됩니다. 참고로 append라는 영어 단어에 '~를 추가하다'라는 뜻이 있다는 점을 감안하면 append 메서드의 사용법을 기억하는 데 도움될 것입니다. 앞에서 append의 뜻이 '~를 추가하다'라고 설명했는데, 이때 '~를'에 해당하는 부분을 괄호 사이에 넣어야 합니다. 다음 코드에서는 "SK텔레콤"이라는 문자열 데이터를 리스트에 추가할 것이기 때문에 append("SK텔레콤")이라고 한 것입니다.

```
>>> kospi_top10.append("SK텔레콤")
>>>
```

append 메서드를 '호출'해 리스트에 데이터를 추가했으므로 리스트에 데이터가 잘 입력됐는지 확인해 보겠습니다. 앞서 여러 번 설명한 것처럼 리스트는 데이터를 순서대로 저장하고 있는 자료구조인데, append 메서드는 데이터를 리스트의 끝에 추가합니다. 따라서 다음과 같이 리스트의 이름을 파이썬 프롬프트에 입력해 보면 kospi_top10 리스트에 'SK텔레콤'이라는 문자열이 추가되어 총 11개의 문자열이 존재하는 것을 확인할 수 있습니다.

```
>>> kospi_top10
['삼성전자', 'SK하이닉스', '현대차', '한국전력', '아모레퍼시픽', '제일모직', '삼성전자우', '삼성
생명', 'NAVER', '현대모비스', 'SK텔레콤']
>>>
```

이제 kospi_top10이라는 변수는 시가총액 상위 11개 종목이 담긴 리스트를 바인딩하고 있으므로 kospi_top10이라는 변수명은 적절하지 않습니다. 이럴 때는 간단히 리스트의 변수명을 변경할 수 있습니다.

```
>>> kospi_top11 = kospi_top10
>>> kospi_top11
['삼성전자', 'SK하이닉스', '현대차', '한국전력', '아모레퍼시픽', '제일모직', '삼성전자우', '삼성
생명', 'NAVER', '현대모비스', 'SK텔레콤']
>>>
```

파이썬 리스트에 데이터를 추가하는 두 번째 방법은 insert라는 이름의 메서드를 사용하는 것입니다. 앞에서 배운 append 메서드를 사용하면 데이터를 리스트에 추가할 수 있는데, 왜 insert라는 메서드도 있을까요? 왜냐하면 append 메서드가 데이터를 항상 리스트의 끝에 삽입하는 것과 달리 insert 메서드는 사용자가 원하는 위치에 데이터를 삽입할 수 있기 때문입니다.

예를 들어, 앞의 예제에서처럼 코스피 상위 10개 종목이 있는데, 11위였던 'SK텔레콤'이 시가총액 기준으로 4위로 갑자기 상승한 경우를 생각해 봅시다. 이 경우 insert(3, 'SK 텔레콤')을 통해 4위에 'SK 텔레콤'을 삽입할 수 있습니다. 참고로 insert 메서드에서 첫 번째 인자인 3은 리스트에 데이터가 삽입될 위치를 의미하고, 두 번째 인자는 리스트에 삽입될 데이터를 의미합니다.

```
>>> kospi_top10 = ['삼성전자', 'SK하이닉스', '현대차', '한국전력', '아모레퍼시픽', '제일모직', '
삼성전자우', '삼성생명', 'NAVER', '현대모비스']
>>> kospi_top10.insert(3, 'SK 텔레콤')
>>> kospi_top10
['삼성전자', 'SK하이닉스', '현대차', 'SK 텔레콤', '한국전력', '아모레퍼시픽', '제일모직', '삼성
전자우', '삼성생명', 'NAVER', '현대모비스']
>>>
```

리스트 데이터 삭제

3.1.4절에서 insert 메서드를 이용해 'SK텔레콤'을 kospi_top10 리스트의 4번째에 삽입해 봤습니다. 그래서 kospi_top10이라는 이름의 리스트에는 총 11개의 종목이 들어 있게 됐습니다. 또한 'SK텔레콤'이 4순위로 입력됐기 때문에 '한국전력'부터는 자신의 순위가 하나씩 밀리게 됐고, 결국 '현대모비스'는 시가총액 상위 11위가 됐습니다.

리스트의 이름은 kospi_top10인데 현재 11개의 종목이 리스트에 들어 있으므로 이번에는 시가총액 11위인 '현대모비스'를 리스트에서 제거하겠습니다. 이를 위해 먼저 리스트에 총 몇 개의 데이터가 있는지 파이썬 코드를 통해 확인해 보겠습니다. 리스트에 입력된 데이터의 개수를 확인할 때는 len이라는 내장 함수(built-in function)를 사용하면 됩니다.

```
>>> len(kospi_top10)
11
>>>
```

참고로 파이썬에서 기본으로 제공하는 함수를 내장 함수라고 합니다. 일상생활에서 '빌트인'이라는 용어를 사용하곤 하는데, 그 '빌트인'이 여기서 사용하는 단어와 같은 의미입니다. len이라는 내장 함수는 length라는 영어 단어의 줄임말이며, 리스트뿐 아니라 문자열의 길이를 확인할 때도 사용할 수 있습니다.

len 함수의 반환값을 확인해 보니 예상한 대로 kospi_top10에는 현재 11개의 데이터가 있습니다. 이제 마지막에 있는 데이터를 확인하고 이를 지워보겠습니다. 맨 마지막 데이터를 확인하기 위해서는 [-1]이라는 파이썬 인덱싱을 사용하면 됩니다. 그리고 리스트의 마지막 데이터를 지우기 위해서는 del을 사용하면 됩니다. 이때 한 가지 주의할 점은 del 다음에 빈칸을 주고 그다음에 지울 데이터를 입력해야 한다는 점입니다.

다음과 같이 리스트의 마지막 데이터인 '현대모비스'를 지운 후 kospi_top10 데이터를 다시 확인해보면 정상적으로 상위 10종목만 남아 있는 것을 확인할 수 있습니다. 더불어 len 함수를 호출해보면 리스트에는 10개의 데이터만 들어있는 것도 확인할 수 있습니다.

```
>>> kospi_top10[-1]
'현대모비스'
>>> del kospi_top10[-1]
>>> kospi_top10
['삼성전자', 'SK하이닉스', '현대차', 'SK 텔레콤', '한국전력', '아모레퍼시픽', '제일모직', '삼성
전자우', '삼성생명', 'NAVER']
>>> len(kospi_top10)
10
>>>
```

02 튜플

3.1절에서는 파이썬의 리스트라는 자료구조를 배웠습니다. 이번 절에서 배울 튜플(tuple)도 여러 개의 데이터를 순서대로 담아두는 데 사용합니다. 단, 튜플과 리스트는 다음과 같은 두 가지 차이점이 있습니다.

1. 리스트는 '['와 ']'를 사용하는 반면 튜플은 '('와 ')'를 사용한다.
2. 리스트는 리스트 내의 원소를 변경할 수 있지만 튜플은 변경할 수 없다.

사실 파이썬의 튜플은 리스트에 있는 여러 기능이 빠져 있고, 반대로 리스트는 튜플이 지원하는 모든 기능을 포함하고 있으므로 튜플을 사용하지 않고 리스트라는 자료구조만 사용해도 프로그래밍하는 데 전혀 불편함이 없습니다. 다만 튜플은 리스트보다 속도가 빠르다는 장점이 있습니다. 따라서 한번 데이터를 저장해둔 후 추가하거나 삭제할 필요가 없는 경우라면 되도록 리스트보다는 튜플을 사용하는 것이 좋습니다. 먼저 튜플을 하나 만들어 봅시다.

```
>>> t = ('Samsung', 'LG', 'SK')
>>> t
('Samsung', 'LG', 'SK')
>>>
```

튜플은 리스트와 비슷하게 인덱싱을 통해 튜플의 원소에 접근할 수 있습니다. 먼저 len 내장 함수를 이용해 튜플의 원소의 개수를 확인해 보니 총 3개가 있는 것을 확인했습니다. 다음과 같이 [0], [1], [2] 인덱싱을 통해 튜플의 각 원소에 접근하는 것을 확인할 수 있습니다. 그러나 튜플은 리스트와 달리 원소를 수정할 수 없으므로 원소의 내용을 바꾸려고 하면 에러가 발생합니다.

```
>>> len(t)
3
>>> t[0]
'Samsung'
>>> t[1]
'LG'
>>> t[2]
'SK'
>>> t[0] = "Naver"
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    t[0] = "Naver"
```

```
TypeError: 'tuple' object does not support item assignment
>>>
```

튜플 슬라이싱

리스트에 저장된 데이터의 일부를 가져올 때 슬라이싱을 이용했습니다. 슬라이싱할 때는 [0:2]와 같은 형태로 데이터를 가져올 시작 인덱스와 끝 인덱스를 지정했습니다. 튜플도 리스트와 동일하게 슬라이싱이 가능합니다. 다음과 같이 t가 바인딩하는 튜플에서 'Samsung'과 'LG'만 가져오려면 다음과 같이 슬라이싱하면 됩니다.

```
>>> t
('Samsung', 'LG', 'SK')
>>> t[0:2]
('Samsung', 'LG')
>>>
```

여기서 한 가지 주의할 점은 튜플을 생성할 때는 '('와 ')' 기호를 사용하지만 데이터에 접근하는 인덱싱이나 슬라이싱에서는 데이터의 범위를 '['와 ']' 기호로 표현한다는 점입니다. 즉, 다음 코드는 파이썬 문법에 어긋나기 때문에 에러가 발생합니다.

```
>>> t(0:2)
SyntaxError: invalid syntax
>>>
```

03 딕셔너리

파이썬에서 리스트, 튜플과 함께 자주 사용되는 자료구조로 딕셔너리(dictionary)가 있습니다. dictionary는 여러분도 잘 알다시피 '사전'이라는 뜻을 가진 영어 단어입니다. 영어사전에는 영어 단어가 알파벳순으로 정렬돼 있어서 단어를 쉽게 찾을 수 있습니다. 파이썬의 딕셔너리도 영어사전과 유사하게 키(key)와 값(value)이라는 것을 쌍으로 저장함으로써 더 쉽게 저장된 값을 찾을 수 있는 구조입니다. 말로는 딕셔너리가 무엇인지 잘 이해되지 않으니 직접 코드를 보면서 딕셔너리에 대해 배워보겠습니다.

여러 종목의 현재가를 저장하는 프로그램을 만든다고 해봅시다. 주식 종목별로 현재가를 저장해야 하는데, 리스트나 튜플로는 이를 쉽게 표현하기가 불가능합니다. 지금까지 배운 리스트나 튜플에는 값을

하나씩 저장했지 지금처럼 '종목'-'종목에 대한 현재가'와 같이 서로 연관된 두 개의 데이터를 저장하지는 못했습니다. 이러한 구조는 파이썬의 딕셔너리를 사용하면 아주 쉽게 표현할 수 있습니다.

딕셔너리는 '{'와 '}' 기호를 사용합니다. 리스트와 튜플을 복습해보면 리스트는 '['와 ']' 기호를 사용했고, 튜플은 '('와 ')' 기호를 사용했습니다. 다음과 같이 아무것도 들어 있지 않은 빈 딕셔너리를 하나 만들어 봅시다.

```
>>> cur_price = {}
>>>
```

위 코드를 실행하면 메모리의 어딘가에 딕셔너리가 하나 만들어지고, 이를 cur_price라는 이름의 변수가 바인딩합니다. 정말로 딕셔너리가 잘 만들어졌는지 type 내장 함수를 이용해 확인해 봅시다. type 함수의 반환값을 보면 'dict'라는 문자열이 있는데, 이것은 dictionary의 줄임말입니다.

```
>>> type(cur_price)
<class 'dict'>
>>>
```

원소가 들어 있지 않은 딕셔너리가 잘 만들어졌으니 딕셔너리에 키-값 쌍을 하나 추가해 보겠습니다. 대신증권의 '현재가'가 30,000원이라고 하면 다음과 작성하면 됩니다. 여기서 'daeshin'이 키고 30000이 값이 됩니다.

```
>>> cur_price['daeshin'] = 30000
>>>
```

사전에 키와 값 쌍을 하나 추가했으니 해당 데이터가 잘 추가됐는지 딕셔너리를 확인해 봅시다. 파이썬 IDLE의 프롬프트에 cur_price라는 변수를 입력하고 엔터 키를 누르면 현재 딕셔너리에 저장된 값을 볼 수 있습니다.

```
>>> cur_price
{'daeshin': 30000}
>>>
```

이번에는 'Daum KAKAO' 한 주의 현재가가 80,000원이라는 것을 딕셔너리에 추가해 봅시다. 값을 추가한 후 cur_price에 값이 잘 입력됐는지도 확인해보기 바랍니다.

```
>>> cur_price['Daum KAKAO'] = 80000
>>> cur_price
{'Daum KAKAO': 80000, 'daeshin': 30000}
>>>
```

딕셔너리에 들어 있는 데이터의 개수를 확인할 때도 리스트나 튜플과 동일하게 len 내장 함수를 사용하면 됩니다. 현재 cur_price 딕셔너리에는 두 개의 키-값 쌍이 입력돼 있으므로 len 내장 함수의 반환값이 2라는 것을 확인할 수 있습니다.

```
>>> len(cur_price)
2
>>>
```

파이썬의 딕셔너리는 리스트와 튜플과 달리 인덱싱을 지원하지 않습니다. 딕셔너리는 리스트와 튜플과 달리 데이터를 순서대로 저장하는 것이 아니라 키와 값의 쌍이 서로 연결되도록만 저장하기 때문입니다. 따라서 다음과 같이 딕셔너리에 정숫값을 이용해 인덱싱하면 오류가 발생합니다.

```
>>> cur_price[0]
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    cur_price[0]
KeyError: 0
>>> cur_price[1]
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    cur_price[1]
KeyError: 1
>>>
```

딕셔너리는 리스트나 튜플과 달리 원소를 추가한 순서대로 데이터가 저장돼 있지 않으므로 딕셔너리에 추가한 데이터를 구하려면 키 값을 사용해야 합니다. 앞서 'daeshin'이 키이고 30000이 값이라고 했습니다. 따라서 cur_price 딕셔너리로부터 'daeshin'의 '현재가'를 구하려면 'daeshin'이라는 키 값을 사용하면 됩니다. 마찬가지로 'Daum KAKAO'라는 문자열이 키이므로 'Daum KAKAO'의 현재가도 'Daum KAKAO'라는 키를 통해 구할 수 있습니다.

```
>>> cur_price['daeshin']
30000
>>> cur_price['Daum KAKAO']
80000
>>>
```

딕셔너리에 데이터 삽입 및 삭제

딕셔너리에 데이터를 삽입하는 것은 앞에서 해봤습니다. 딕셔너리에 데이터를 삽입하는 방법을 정리해 보면 딕셔너리는 리스트의 append나 insert 같은 메서드를 지원하지 않으며, 명시적으로 키-값 쌍을

넣어야 했습니다. 예제를 통해 한 번 더 복습해 보겠습니다. cur_price라는 딕셔너리에 'naver'의 현재가가 80,000원이라는 데이터를 추가해 보겠습니다.

```
>>> cur_price['naver'] = 800000
>>> cur_price
{'Daum KAKAO': 80000, 'naver': 800000, 'daeshin': 30000}
>>>
```

참고로 파이썬에서 딕셔너리를 생성할 때 다음과 같이 초깃값을 넣어서 딕셔너리를 생성할 수도 있습니다.

```
>>> cur_price = {'Daum KAKAO': 80000, 'naver':800000, 'daeshin':30000}
>>> cur_price
{'naver': 800000, 'Daum KAKAO': 80000, 'daeshin': 30000}
>>>
```

딕셔너리에 저장된 데이터를 삭제하려면 리스트와 마찬가지로 del을 사용하면 됩니다. 참고로 튜플은 리스트와 딕셔너리와 달리 데이터를 삭제할 수 없었습니다. 주식 종목의 현재가를 저장하고 있는 cur_price에서 'daeshin'이라는 종목의 현재가를 지우고 싶다면 다음과 같이 키를 이용하면 됩니다. 삭제한 후에는 cur_price에서 키-값 쌍이 잘 지워졌는지 확인해보기 바랍니다.

```
>>> del cur_price['daeshin']
>>> cur_price
{'naver': 800000, 'Daum KAKAO': 80000}
>>>
```

딕셔너리로부터 키-값 구하기

딕셔너리에는 키-값 쌍이 저장된다고 했습니다. 그렇다면 딕셔너리에 있는 키 값만 구하려면 어떻게 해야 할까요? 이를 위해 파이썬의 딕셔너리는 keys라는 메서드를 제공합니다. 따라서 키 값만 구하고 싶을 때 "딕셔너리이름.keys()"라고 호출하면 됩니다. 그런데 언제 딕셔너리로부터 키 값만 구해야 할까요?

다음 코드의 cur_price 딕셔너리에는 세 종목에 대한 현재가가 저장돼 있습니다. cur_price 딕셔너리를 통해 어떤 주식 종목에 대한 현재가를 조회하고 싶다면 먼저 해당 종목명이 cur_price라는 딕셔너리에 있는지 확인해야 합니다. 이를 위해서는 cur_price라는 딕셔너리에서 키 값을 구해와야겠죠?

```
>>> cur_price = {'Daum KAKAO': 80000, 'naver':800000, 'daeshin':30000}
>>> cur_price.keys()
dict_keys(['naver', 'Daum KAKAO', 'daeshin'])
>>>
```

위 코드를 살펴보면 키 값을 구하기 위해 딕셔너리의 이름인 cur_price 다음에 마침표를 하나 붙이고 keys()라고 적는 것을 확인할 수 있습니다. 이러한 방식은 리스트에서 insert나 append 같은 메서드를 호출하는 방식과 동일합니다.

keys 메서드의 반환값을 보면 'Daum KAKAO'와 'naver'가 파이썬의 리스트로 표현된 것 같기도 한데 그 앞에 'dict_keys'라는 것이 있습니다. 사실 keys 메서드의 반환값은 리스트가 아니며, 리스트로 만들려면 list라는 키워드를 이용해 타입을 변환해야 합니다.

```
>>> list(cur_price.keys())
['naver', 'Daum KAKAO', 'daeshin']
>>>
```

그러나 이처럼 단순히 list를 통해 반환값을 받으면 사용하기 어려울 것입니다. 항상 변수를 통해 반환값을 바인딩하는 것이 좋습니다. 다음과 같이 stock_list라는 이름의 변수로 list의 반환값을 바인딩합니다. stock_list가 바인딩하고 있는 값을 보면 파이썬의 리스트임을 확인할 수 있습니다.

```
>>> stock_list = list(cur_price.keys())
>>> stock_list
['naver', 'Daum KAKAO', 'daeshin']
>>>
```

딕셔너리에서 키 목록을 구하는 것과 비슷하게 값 목록을 구하려면 values라는 메서드를 사용하면 됩니다. cur_price 딕셔너리에서 값 목록을 리스트로 만들어 바인딩해 봅시다.

```
>>> price_list = list(cur_price.values())
>>> price_list
[800000, 80000, 30000]
>>>
```

'Samsung'이라는 종목의 현재가를 cur_price라는 딕셔너리로부터 구해야 한다고 해봅시다. 이를 위해서는 앞서 이야기한 것처럼 먼저 'Samsung'이라는 키 값이 cur_price 딕셔너리에 있는지 확인해야 합니다. 딕셔너리의 키 값은 keys 메서드를 통해 얻을 수 있었죠? 다음과 같이 in이라는 키워드와 keys 메서드의 반환값을 이용하면 이를 쉽게 확인할 수 있습니다.

```
>>> 'Samsung' in cur_price.keys()
False
>>>
```

위 코드를 한국어로 해석해보면 "'Samsung'이 keys() 안에 있는가?" 정도가 됩니다. 참고로 in이라는 영어 단어는 "안에", "내에"라는 뜻을 가지고 있습니다. 일단 반환값을 보면 False라는 영어 단어를 볼

수 있는데 이 단어는 "거짓의", "틀린"이라는 의미가 있습니다. 즉, 'Samsung'이 cur_price 딕셔너리의 키에 들어있지 않다는 것을 알 수 있습니다.

그렇다면 'naver'는 어떨까요? 다음 코드를 보면 True가 반환된 것을 확인할 수 있습니다. True는 '사실', '정말', '진짜의'라는 뜻이 있습니다. 즉, 'naver'라는 종목명은 cur_price의 키 중 하나임을 알 수 있습니다.

```
>>> 'naver' in cur_price.keys()
True
>>>
```

'naver'는 cur_price 딕셔너리의 키 중 하나이기 때문에 키를 통해 현재가 데이터를 구할 수 있습니다. 그러나 'Samsung'은 키에 포함돼 있지 않기 때문에 'Samsung'이라는 키를 사용하면 오류가 발생합니다.

```
>>> cur_price['naver']
800000
>>> cur_price['Samsung']
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    cur_price['Samsung']
KeyError: 'Samsung'
>>>
```

연습 문제

문제 3-1

2015년 9월 초의 네이버 종가는 표 3.2와 같습니다. 09/07의 종가를 리스트의 첫 번째 항목으로 입력해서 naver_closing_price라는 이름의 리스트를 만들어보세요.

표 3.2 네이버 종가

날짜	요일	종가
09/11	금	488,500
09/10	목	500,500
09/09	수	501,000
09/08	화	461,500
09/07	월	474,500

문제 3-2

문제 3-1에서 만든 naver_closing_price를 이용해 해당 주에 종가를 기준으로 가장 높았던 가격을 출력하세요. (힌트: 리스트에서 최댓값을 찾는 함수는 max()이고, 화면에 출력하는 함수는 print()입니다.)

문제 3-3

문제 3-1에서 만든 naver_closing_price를 이용해 해당 주에 종가를 기준으로 가장 낮았던 가격을 출력하세요. (힌트: 리스트에서 최솟값을 찾는 함수는 min()이고, 화면에 출력하는 함수는 print()입니다.)

문제 3-4

문제 3-1에서 만든 naver_closing_price를 이용해 해당 주에서 가장 종가가 높았던 요일과 가장 종가가 낮았던 요일의 가격 차를 화면에 출력하세요.

문제 3-5

문제 3-1에서 만든 naver_closing_price를 이용해 수요일의 종가를 화면에 출력하세요.

문제 3-6

문제 3-1의 표 3.2를 이용해 날짜를 딕셔너리의 키 값으로, 종가를 딕셔너리의 값으로 사용해 naver_closing_price2라는 딕셔너리를 만드세요.

문제 3-7

문제 3-6에서 만든 naver_closing_price2 딕셔너리를 이용해 09/09일의 종가를 출력하세요.

04

제어문

저자는 매일 아침 회사에 출근합니다. 이때 스마트폰을 통해 날씨를 확인하고 비가 올 것 같으면 우산을 챙기고 그렇지 않으면 그냥 출근합니다. 여기서 매일 아침 회사로 출근하는 것은 반복적인 일에 해당하고, 날씨에 따라 우산을 챙기는 일은 조건을 보고 어떤 것을 선택하는 일에 해당합니다.

프로그램에도 이와 비슷한 것들이 있습니다. 어떤 조건을 보고 코드를 수행하는 것을 분기문이라고 하고, 어떤 조건하에서 반복적으로 코드를 수행하는 것을 반복문이라고 합니다. 일반적으로 프로그램은 명령어가 기술된 순서대로 순차 수행되는데 분기문과 반복문은 순차 수행의 흐름을 변화시키기 때문에 제어문이라고도 합니다. 파이썬에는 `if`, `for`, `while`이라는 제어문이 있으며, 이번 장에서는 이러한 제어문을 배우겠습니다.

01 Boolean

2장에서는 파이썬의 기본 자료형인 정수형, 실수형, 문자열을 배웠습니다. 이 세 가지 자료형이 파이썬에서 주로 사용되긴 하지만 Boolean(불리언)도 알게 모르게 자주 사용되는 기본 자료형입니다. 다만 Boolean은 다른 자료형과 달리 True 또는 False라는 값만 바인딩할 수 있습니다.

다음 코드에서 a와 b라는 변수는 각각 True와 False라는 값을 바인딩합니다. type 내장 함수를 이용해 타입을 확인해보면 <class 'bool'>이 반환되는 것을 볼 수 있습니다. 짐작하시겠지만 여기서 bool은 Boolean의 줄임말입니다. 참고로 True나 False는 파이썬의 예약어로 true나 false가 아니라 첫 글자로 대문자를 사용해야 합니다.

```
>>> a = True
>>> type(a)
<class 'bool'>
>>> b = False
>>> type(b)
<class 'bool'>
>>>
```

그렇다면 Boolean이라는 자료형은 언제 사용될까요? Boolean이라는 자료형은 파이썬의 비교 연산자에서 많이 사용됩니다. 여기서 비교 연산자라는 것은 어떤 두 값을 비교하는 데 사용되는 연산자를 의미합니다. 예를 들어, 표 4.1의 연산자가 바로 파이썬에서 사용되는 비교 연산자입니다.

표 4.1 파이썬 비교 연산자

연산자	연산자 의미
==	같다.
!=	다르다.
>	크다.
<	작다.
>=	크거나 같다.
<=	작거나 같다.

파이썬의 비교 연산자는 비교 연산자를 기준으로 왼쪽과 오른쪽에 비교할 값이 존재합니다. 다음 코드의 첫 번째 문장에서는 정수형 값인 3을 서로 비교합니다. 첫 번째 문장의 결과를 보면 True가 반환된 것을 확인할 수 있는데, 비교 연산자를 기준으로 왼쪽에 있는 3과 오른쪽에 있는 3이 같은 값이기 때문입니다. 두 번째 문장을 보면 반환값이 False인 것을 확인할 수 있습니다. 이는 두 값이 같지 않다(!=)라고 했는데, 실제로는 같은 값이기 때문입니다. 세 번째는 왼쪽의 3이 오른쪽의 3보다 작다(<)라고 했는데, 실제로는 작지 않기 때문에 이번에도 반환값이 False가 됩니다.

```
>>> 3 == 3
True
>>> 3 != 3
```

```
False
>>> 3 < 3
False
>>> 3 > 3
False
>>> 3 <= 3
True
>>> 3 >= 3
True
>>>
```

위 코드처럼 Boolean 자료형은 비교 연산자의 반환값으로 자주 사용됩니다. 표 4.1의 비교 연산자는 정수형뿐만 아니라 실수형과 문자열 자료형에도 그대로 적용할 수 있습니다. 다음 코드를 보면 mystock이라는 변수는 "Naver"라는 문자열을 바인딩하고 있습니다. 이 상태에서 mystock이 바인딩하는 문자열이 "Naver"와 같은지(==) 확인해 보니 반환값으로 True가 반환된 것을 확인할 수 있습니다.

```
>>> mystock = "Naver"
>>> mystock == "Naver"
True
>>>
```

사실 파이썬의 비교 연산자를 보면 너무나 당연해서 비교 연산자를 처음 배울 때는 큰 감흥이 없습니다. 그러나 비교 연산자는 앞으로 주식 프로그래밍을 할 때 자주 사용하게 될 연산자이니 꼭 기억하기 바랍니다. 예를 들어, 상한가였던 종목을 자동으로 매수하는 프로그램을 작성하는 경우를 생각해 봅시다. 유가증권시장과 코스닥시장에 있는 수많은 종목 중에서 어제 상한가였던 종목을 알아내려면 어떻게 해야 할까요? 일단 수많은 종목 중에서 찾는 것은 이번 장에서 배우게 될 반복문과 관련이 있으므로 그때 다시 다루기로 하고, 먼저 한 종목에 대해 두 개의 종가가 있을 때 이 종목이 상한가였는지 확인해 봅시다.

다음 코드에서 day1은 어떤 주식 종목의 전일 종가이고 day2는 당일 종가를 의미합니다. day2의 값인 13,000원은 day1의 값인 10,000원에서 30% 오른 가격이므로 상한가가 맞습니다(참고로 2015년 6월 15일부터는 가격제한폭이 30%로 증가했습니다). 이러한 상한가를 확인하는 알고리즘을 파이썬 코드로 작성해보면 day2에서 day1을 뺀 금액이 day1의 0.3(30%)에 해당하는 금액과 같은지(==) 확인하면 됩니다.

```
>>> day1 = 10000
>>> day2 = 13000
>>> (day2- day1) == (day1 * 0.3)
True
>>>
```

02 논리 연산자

앞서 비교 연산자를 이용해 상한가를 확인하는 코드를 작성해봤습니다. 그러나 사실 한국 거래소(www.krx.co.kr)의 가격제한폭 제도를 살펴보면 정확한 상한가를 산출하는 방식은 기준 가격(전일 종가)에 단순히 0.3을 곱하는 것이 아니며 다음과 같이 조금 복잡하게 계산됩니다.

1차 계산: 기준 가격에 0.3을 곱한다.

2차 계산: 기준 가격의 호가 가격단위에 미만을 절사한다.

3차 계산: 기준 가격에 2차 계산에 의한 수치를 가감하되, 해당 가격의 호가 가격단위 미만을 절사한다.

예를 들어, 기준 가격이 9,980원인 경우 다음과 같이 계산됩니다.

1차 계산: 9,980원 x 0.3 = 2,994원

2차 계산: 2,990원(기준 가격: 9,980원)의 호가 가격단위인 10원 미만 절사

3차 계산:

합산가격: 9,980원 + 2,990원 = 12,970원

호가 가격단위 적용: 12,970원의 호가 가격단위인 50원 미만 절사(2차 절사)

상한가: 12,950원

위의 계산 과정을 살펴보면 호가 가격에 따라 두 번의 절사 과정을 거칩니다. KRX 한국 거래소에서 제공하는 호가 가격단위를 살펴보면 표 4.2와 같습니다. 여기서 '호가 가격단위'란 가격대별로 호가할 수 있는 최소 단위를 말합니다. 참고로 '호가'는 거래소의 회원인 증권회사가 자기 명의로 시장에 매도 또는 매수 의사를 표시하는 것을 의미합니다. 잘 알고 계시다시피 증권사는 증권사의 고객으로부터 주문을 위탁받은 후 동 주문을 거래소에 호가하게 되는 것입니다.

표 4.2 호가 가격단위

구분	단위
1,000원 미만	1원
1,000원 이상 5,000원 미만	5원
5,000원 이상 10,000원 미만	10원
10,000원 이상 50,000원 미만	50원
50,000원 이상 100,000원 미만	100원
100,000원 이상 500,000원 미만	500원
500,000원 이상	1,000원

문제는 "표 4.2로부터 위에서 예시로 든 기준 가격(9,980원)에 대한 호가를 파이썬을 통해 어떻게 구할 것인가?"입니다. 이럴 때 사용할 수 있는 연산자가 바로 논리 연산자인데, 파이썬에는 and, or, not이라는 세 가지 종류의 논리 연산자가 있습니다. 논리 연산자 중 'and'에는 '그리고'라는 뜻이 있고, 'or'에는 '또는'이라는 뜻이 있습니다. 마지막으로 'not'은 '~아닌'이라는 뜻이 있습니다.

간단한 코드를 통해 논리 연산자에 대해 좀 더 살펴보겠습니다. 다음 코드는 cur_price가 바인딩하고 있는 기준 가격이 '5,000원 이상 10,000원 미만'의 호가 구간에 해당하는지 나타냅니다. cur_price라는 변수가 현재 바인딩하고 있는 값인 9,980원은 5,000원보다는 큰 가격이고 10,000원보다 낮은 가격이기 때문에 True라는 불리언 값이 반환됩니다. and 연산자는 나열된 여러 조건이 모두 True일 때만 연산의 결과로 True를 반환합니다.

```
>>> cur_price = 9980
>>> cur_price >= 5000 and cur_price < 10000
True
>>>
```

앞서 당일 종가가 전일 종가보다 정확히 30% 상승했을 때 상한가로 판단했습니다. 그러나 앞에서 설명한 것처럼 실제로는 호가의 범위에 따라 두 번의 절사 과정을 거치기 때문에 실제로는 30%보다 낮은 가격일 수도 있습니다. 물론 호가를 정확히 계산하고 절사도 정확히 계산해서 실제 상한가를 계산할 수도 있지만 다음 코드와 같이 or라는 논리 연산자를 사용해 대략 상한가와 비슷하게 오른 종목을 찾을 수도 있습니다. 다음 코드는 당일 종가(day2)에서 전일 종가(day1)를 뺀 금액이 전일 종가의 30%에 해당하는 금액이거나 29.2%에 해당하는 금액보다 큰 금액이면 상한가로 판단합니다. or 연산자는 나열된 여러 가지 조건 중 하나라도 True이면 연산의 결과로 True를 반환합니다.

```
>>> day1 = 10000
>>> day2 = 13000
>>> ((day2- day1) == (day1 * 0.3)) or ((day2-day1) > (day1 * 0.292))
True
>>>
```

03 if 문

이번 절에서는 어떤 조건에 따라 코드를 다르게 수행하는 조건문에 대해 배우겠습니다. 조건문이라고 하니 막상 뭔가 어려워 보이지만 아래와 같은 문장이 조건문의 예입니다. 주식 투자를 하다 보면 실제로도 아래와 같은 행동을 많이 합니다.

"위키북스의 현재가가 10,000원 이상이면 10주 매수"

"위키북스의 현재가가 9,000원 이하이면 10주 매도"

그렇다면 파이썬에서는 위와 같은 것들을 어떻게 표현할 수 있을까요? 먼저 위의 두 문장을 잘 살펴보면 공통으로 '~이면'이라는 단어가 있는 것을 볼 수 있습니다. 참고로 영어 단어 중 if라는 단어의 뜻이 '만약 ~라면'입니다. 파이썬에서도 '만약 ~라면'이라는 뜻을 가진 if라는 키워드로 조건문을 표현합니다. if 문을 이용해 앞에서 예로 든 매수 상황을 파이썬으로 표현하면 다음과 같습니다.

```
>>> wikibooks_cur_price = 11000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")

Buy 10
>>>
```

위 코드를 살펴보면 몇 가지 새로운 파이썬 문법을 볼 수 있습니다.

1. 조건문에는 if라는 키워드를 사용한다.
2. if 다음에는 '조건'이 존재하는데 이 조건이 참(True)이면 들여쓰기한 문장이 실행된다.
3. if 문의 끝에는 콜론(:)을 입력한다.
4. if 문의 조건이 참(True)일 때 실행되는 문장은 들여쓰기해야 한다.

위 코드에서 첫 번째 문장이 실행되면 wikibooks_cur_price라는 변수는 11000이라는 정숫값을 바인딩하게 됩니다. 두 번째 문장이 실행될 때 먼저 if 문의 조건에 해당하는 'wikibooks_cur_price >= 10000'에 대해 참인지 거짓인지가 판단됩니다. wikibooks_cur_price 변수가 현재 바인딩하는 값은 11000이고, 이 값은 10000보다 크거나 같으므로 조건식은 참이 됩니다. if 문의 조건식이 참이므로 4칸의 공백으로 들여쓰기된 'print("Buy 10")'이라는 문장이 실행됩니다. 그래서 화면에 'Buy 10'이 출력된 것입니다.

if 문을 이용해 위의 매도 상황을 파이썬 코드로 표현하면 다음과 같습니다. 조건문을 표현하기 위해 if라는 키워드를 사용했으며, if 뒤에 조건식을 적었습니다. wikibooks_cur_price 변수는 현재 8,000원이라는 값을 바인딩하고 있으므로 if 문의 조건식인 'wikibooks_cur_price < 9000'은 True가 됩니다. 따라서 들여쓰기된 코드가 실행되고 화면에는 'Sell 10'이 출력됩니다.

```
>>> wikibooks_cur_price = 8000
>>> if wikibooks_cur_price < 9000:
```

```
        print("Sell 10")

Sell 10
>>>
```

앞의 코드에서는 if 문의 조건식이 참일 때 들여쓰기된 한 줄의 코드만 실행됐습니다. 그런데 "위키북스의 현재가가 10,000원 이상이면 5주씩 3번을 연속해서 매수"하는 경우는 어떻게 프로그래밍할까요? 이럴 때는 다음 코드와 같이 조건이 참일 때 수행될 문장을 동일한 들여쓰기로 표현하면 됩니다. 참고로 C/C++ 또는 자바 같은 프로그래밍 언어에서는 동시에 수행될 코드 블록을 { } 기호로 감싸서 표현하는 반면 파이썬에서는 들여쓰기를 통해 코드 블록을 표현합니다.

```
>>> wikibooks_cur_price = 11000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 5")
        print("Buy 5")
        print("Buy 5")

Buy 5
Buy 5
Buy 5
>>>
```

파이썬이 아닌 다른 프로그래밍 언어를 배우신 분이라면 이 부분이 처음에는 조금 어색할 수 있습니다. 그러나 시간이 지남에 따라 파이썬과 같이 들여쓰기를 통해 코드 블록을 표현하는 방식이 더 가독성 있는 코드를 작성하는 데 도움된다는 사실을 알 수 있을 것입니다. 참고로 파이썬에서 들여쓰기는 키보드의 스페이스 바를 눌러서 같은 개수의 공백을 넣거나 탭을 사용해 들여쓰면 되는데, 보통 네 칸의 공백을 사용합니다.

if ~ else 문

이번에는 "위키북스의 현재가가 10,000원 이상이면 10주 매수, 그렇지 않으면 보유"와 같은 문장을 파이썬 코드로 작성해 봅시다. 우선 앞에서 배운 조건문과 다른 부분을 살펴보면 '그렇지 않으면'이라는 새로운 조건이 추가된 것을 확인할 수 있습니다. 파이썬에서는 '그렇지 않으면'이라는 조건을 else라는 키워드를 사용해서 표현합니다.

앞의 문장을 if와 else라는 키워드를 사용해 다음 코드와 같이 나타낼 수 있습니다. 이때 else 문은 if 문과 달리 뒤에 조건식이 위치하지 않습니다. 왜냐하면 else 문은 if 문의 조건식이 참이 아닐 때 수행되기 때문입니다. 다음 코드에서 wikibooks_cur_price라는 변수는 11000을 바인딩하고 있으므로

if 문의 조건식은 참이 됩니다. 따라서 if 문에 들여쓰기된 'print("Buy 10")'이 수행되고 화면에는 'Buy 10'이 출력됩니다.

```
>>> wikibooks_cur_price = 11000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")
else:
        print("Holding")

Buy 10
>>>
```

이번에는 코드를 조금 수정해서 wikibooks_cur_price가 바인딩하는 값을 9000으로 변경한 후 if ~ else 문을 실행해 보겠습니다. 다음 코드에서는 if 문 이후의 조건식이 False가 되므로 else 문에 들여쓰기된 코드인 'print("Holding")'이 실행되어 화면에 'Holding'이 출력됩니다.

```
>>> wikibooks_cur_price = 9000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")
else:
        print("Holding")

Holding
>>>
```

if ~ else 문에서 한 가지 주의해야 할 점은 바로 들여쓰기입니다. 그림 4.1의 코드를 보면 앞서 작성한 코드와 거의 비슷하지만 에러가 발생했습니다. 이는 else 문의 위치가 잘못됐기 때문입니다. 그림 4.1에서 if 키워드 앞에는 공백이 없지만 else 키워드 앞에는 4칸의 공백이 존재하는데, 이처럼 if와 else의 들여쓰기가 서로 다르면 파이썬에서는 문법 에러가 발생합니다.

```
>>> wikibooks_cur_price = 9000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")
    else:
        print("Holding")

SyntaxError: unindent does not match any outer indentation level
>>>
```

그림 4.1 파이썬 들여쓰기 오류

파이썬 IDLE를 처음 사용할 때는 오히려 다음과 같은 코드가 if와 else의 들여쓰기가 잘못된 것처럼 보입니다.

```
>>> if 3 > 1:
        print("3이 크다")
else:
        print("3이 작다")


3이 크다
>>>
```

그러나 파이썬 IDLE의 프롬프트(>>>)를 지워보면 실제로는 if와 else가 같은 위치에서 시작됨을 확인할 수 있습니다.

```
if 3 > 1:
        print("3이 크다")
else:
        print("3이 작다")
```

if ~ elif ~ else 문

if ~ else 문을 이용하면 if 문 뒤에 존재하는 조건식이 참이냐 거짓이냐에 따라 그림 4.2와 같이 두 개의 코드 블록 중 하나를 실행할 수 있었습니다.

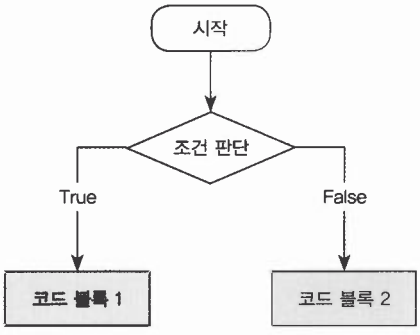


그림 4.2 if ~ else 문의 실행 구조

그런데 실행해야 할 경우의 수가 두 가지가 아니라 그보다 많은 경우에는 어떻게 표현해야 할까요? 예를 들어, 표 4.2의 호가 가격단위를 보면 호가가 결정되는 경우의 수가 두 가지만 있는 것이 아니라 가격 범위가 7개라는 것을 확인할 수 있습니다. 이럴 때 사용할 수 있는 추가 키워드가 바로 elif입니다. elif 키워드는 '그렇지 않고 ~이면'이라는 의미입니다. 표 4.2를 한국어로 표현해보면 다음과 같이 표현할 수 있을 것입니다.

현재가가 1,000원 미만이면 호가 가격은 1원,

그렇지 않고 현재가가 1,000원 이상이고 5,000원 미만이면 호가 가격단위는 5원,

그렇지 않고 현재가가 5,000원 이상이고 10,000원 미만이면 호가 가격단위는 10원,

그렇지 않고 현재가가 10,000원 이상이고 50,000원 미만이면 호가 가격단위는 50원,

그렇지 않고 현재가가 50,000원 이상이고 100,000원 미만이면 호가 가격단위는 100원,

그렇지 않고 현재가가 100,000원 이상이고 500,000원 미만이면 호가 가격단위는 500원,

그렇지 않고 현재가가 500,000원 이상이면 호가 가격단위는 1,000원

위의 문장에서 '~이면'을 if로, '그렇지 않고'를 elif라는 키워드로 표현하면 다음과 같은 파이썬 코드를 작성할 수 있습니다. 물론 조건식이 있는 부분에는 파이썬의 비교 연산자와 논리 연산자를 사용했습니다.

```
>>> price = 7000
>>> if price < 1000:
        bid = 1
elif price >= 1000 and price < 5000:
        bid = 5
elif price >= 5000 and price < 10000:
        bid = 10
elif price >= 10000 and price < 50000:
        bid = 50
elif price >= 50000 and price < 100000:
        bid = 100
elif price >= 100000 and price < 500000:
        bid = 500
elif price >= 500000:
        bid = 1000

>>> bid
10
>>>
```

위 코드에서 price라는 변수는 7000을 바인딩하고 있으므로 호가 가격단위(bid)는 10원이 됩니다. 파이썬 IDLE에서도 bid 변수가 바인딩하고 있는 값을 확인해보면 10이 반환되는 것을 확인할 수 있습니다.

04 for 문

이번 절에서는 파이썬의 제어문 중 하나인 for 반복문을 배우겠습니다. '반복'이라는 단어의 의미는 '같은 일을 되풀이함'입니다. 그렇다면 프로그래밍에서는 언제 반복문이 필요할까요?

화면에 1부터 10까지 숫자를 출력한다고 가정해 봅시다. 파이썬에 반복문이 없다면 다음 코드와 같이 한 번에 하나씩 출력해야 할 것입니다. 그러나 10까지가 아니라 100까지 출력해야 할 경우 이를 코드로 작성하려면 시간이 오래 걸리고 지루할 것입니다. 이럴 때 사용하는 것이 바로 반복문입니다.

```
>>> print(1)
1
>>> print(2)
2
>>> print(3)
3
>>> print(4)
4
>>> print(5)
5
>>> print(6)
6
>>> print(7)
7
>>> print(8)
8
>>> print(9)
9
>>> print(10)
10
>>>
```

for 문을 사용하면 1~10까지 출력하는 기능을 다음과 같이 구현할 수 있습니다.

```
>>> for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
        print(i)

0
1
2
3
```

```
4
5
6
7
8
9
10
>>>
```

위 코드에서는 화면에 문자열을 출력하는 함수인 print를 단 한 번만 사용했지만 화면에는 0부터 10까지의 값이 출력된 것을 확인할 수 있습니다. 이처럼 for 문을 사용하면 중복된 부분의 코드를 줄일 수 있어 프로그램을 더욱 빨리 작성할 수 있고 코드의 가독성도 향상됩니다.

위 코드를 살펴보면 for라는 새로운 파이썬 키워드를 볼 수 있습니다. 영어 사전에서 for라는 단어의 의미를 찾아보면 정말로 많은 뜻이 나옵니다. 대표적으로 '왜냐하면', '동안', '위한', '대한', '로써는', '치고는', '에게', '찬성하는', '에도 불구하고'와 같은 뜻이 있습니다. 물론 이러한 뜻을 다 외울 필요는 없고 파이썬에서 for는 '~대한' 정도의 의미를 가진 것으로 생각하면 됩니다.

in이라는 영어 단어는 '~안에'라는 뜻이 있으므로 위 파이썬 코드를 영어 문장을 해석하듯이 해석해보면 "파이썬 리스트 안에 있는 각 i에 대해 print(i)하시오." 정도가 됩니다. for 문을 작성할 때 주의할 점은 앞에서 배운 if 문과 마찬가지로 for 문의 끝에 콜론(:)이 있어야 하며, for 문에서 수행되는 문장은 들여쓰기돼 있어야 한다는 점입니다.

이번에는 for 문을 사용해 0부터 4까지의 숫자가 두 번씩 출력되는 프로그램을 작성해 봅시다. 힌트를 조금 드리자면 if 문에서도 조건이 참일 때 수행될 문장은 if 문의 시작 위치로부터 동일하게 들여쓰기 했습니다.

다음 코드를 보면 for 문에서 들여쓰기된 두 개의 print 문을 볼 수 있습니다. 코드가 실행되면 먼저 리스트 내의 첫 번째 요소인 0에 대해 print(0), print(0)이 수행됩니다. 그런 다음 리스트의 두 번째 요소인 1에 대해 print(1), print(1)이 수행됩니다. 이렇게 해서 리스트의 마지막 요소인 4에 대해서까지 print 문이 수행됩니다.

```
>>> for i in [0, 1, 2, 3, 4]:
        print(i)
        print(i)

0
0
```

```
1
1
2
2
3
3
4
4
>>>
```

위 코드에서 for라는 키워드 뒤에 위치하는 i는 변수로서 in 다음에 있는 리스트의 요소를 하나씩 바인딩합니다. 처음에는 0을 바인딩하고 그다음에는 1, 2, 3, 4 순으로 하나씩 바인딩합니다. i는 for나 in과 같은 키워드가 아니라 변수이기 때문에 변수 이름 규칙만 만족한다면 마음대로 지정할 수 있습니다.

for와 range

앞서 for라는 파이썬 키워드를 사용해 단 두 줄의 코드로 화면에 0부터 10까지를 출력해봤습니다. 그러나 한 가지 불편한 점은 출력하고자 하는 숫자를 리스트를 통해 명시적으로 저장하고 있어야 한다는 점입니다. 물론 앞의 예시처럼 반복하는 데이터가 많지 않은 경우에는 리스트를 사용해 명시적으로 데이터를 나열할 수 있습니다. 그런데 0부터 10까지가 아니라 0부터 100까지 출력해야 할 때는 어떻게 해야 할까요?

이럴 때 사용하는 것이 바로 range입니다. range는 '범위'라는 뜻을 가진 영어 단어인데, 파이썬에서 range를 이용하면 간단히 정수 범위를 표현할 수 있습니다. 예를 들어, range(1, 10)은 1부터 9까지의 숫자 범위를 나타냅니다.

```
>>> range(1,10)
range(1, 10)
>>>
```

range에서 중요한 점은 시작 숫자와 끝 숫자를 지정했을 때 끝 숫자는 범위에 포함되지 않는다는 점입니다. 이를 확인하기 위해 range(1, 10)을 list라는 키워드를 통해 리스트 객체로 변환해보면 리스트에 정말로 1부터 9까지만 저장돼 있음을 확인할 수 있습니다.

```
>>> list(range(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

range를 이용하면 큰 수의 범위에 대해서도 for 문을 쉽게 작성할 수 있습니다. 먼저 0부터 10까지의 숫자를 출력하는 코드를 for와 range를 사용해 다시 작성해 보겠습니다. 다음 코드를 보면 in 다음에

리스트를 사용하는 경우보다 코드가 매우 간단하다는 것을 확인할 수 있습니다. 0부터 10까지 출력하기 위해 range(0, 10)이 아니라 range(0, 11)이라는 표현이 사용된 것도 눈여겨보기 바랍니다.

```
>>> for i in range(0, 11):
        print(i)

0
1
2
3
4
5
6
7
8
9
10
>>>
```

독자 중에서는 다음 코드와 같이 작성한 분도 있을 것입니다. range(0, 11)을 list라는 키워드를 이용해 리스트 객체로 변환한 후 해당 리스트 객체에 대해 for 문을 수행했습니다. range(0, 11)을 사용했을 때와 list(range(0, 11))을 사용해 실행한 결과는 동일합니다. 다만 list(range(0, 11))과 같이 명시적으로 리스트 객체로 변환하기보다는 range(0, 11)을 그대로 사용하는 편이 메모리 사용 관점에서 더 효율적입니다.

```
>>> for i in list(range(0, 11)):
        print(i)

0
1
2
3
4
5
6
7
8
9
10
>>>
```

range 객체를 그대로 사용하면 for 문에서 필요할 때마다 값을 i라는 변수가 바인딩해서 사용하게 되지만 리스트 객체로 변환하면 모든 범위의 값을 한 번에 리스트로 만들어서 저장하기 때문에 더 큰 메모리가 필요해집니다. 메모리 절약뿐만 아니라 코드 가독성 측면에서도 range 객체를 그대로 사용하는 편이 더 간결해 보입니다.

for와 리스트

앞 절에서는 리스트에 숫자를 저장한 후 for 문에서 해당 리스트를 이용해 반복적으로 어떤 작업을 수행하는 코드를 작성했습니다. 앞에서 배운 것처럼 리스트에는 정수형만 입력할 수 있는 것이 아니라 문자열도 입력할 수 있습니다. 따라서 for 문과 리스트 자료구조를 적절히 사용하면 리스트 내에 있는 원소에 대해 반복적인 작업을 손쉽게 수행할 수 있습니다.

예를 들어, 관심 있게 보고 있는 종목들의 종목명을 리스트로 저장하고 있다가 해당 종목에 대해 각각 10주씩 매수하는 프로그램을 작성한다고 가정해 봅시다. 다음 코드를 보면 interest_stocks 리스트에는 'Naver', 'Samsung', 'SK Hynix'와 같은 종목명이 있습니다.

```
>>> interest_stocks = ["Naver", "Samsung", "SK Hynix"]
```

관심 종목 리스트에 있는 각 원소에 대해 10주씩 매수하기 위해 for 문을 사용했습니다. for 키워드 다음에 나오는 company는 변수 이름으로서 interest_stocks 리스트에 있는 원소를 하나씩 바인딩하는 역할을 합니다. for 문에서 리스트의 각 원소에 대해 수행될 문장은 print 문인데 이때 %s라는 표현이 나옵니다. print 함수에서는 문자열이 출력될 위치에 %s를 쓰고 실제로 출력될 문자열은 % 기호 뒤에 변수명으로 지정하면 됩니다.

```
>>> for company in interest_stocks:
        print("%s: Buy 10" % company)

Naver: Buy 10
Samsung: Buy 10
SK Hynix: Buy 10
>>>
```

print 함수의 사용법은 C 언어나 자바와 비슷하므로 이전에 프로그래밍해본 경험이 있는 분들은 쉽게 이해하실 것입니다. 그러나 프로그래밍 경험이 없는 분들은 처음에는 매우 어색하고 복잡하게 느껴지실 겁니다. 만일 다음과 같이 'print("company Buy 10")'이라고 작성하면 리스트 안에 있는 관심 종목명이 출력되는 것이 아니라 'company Buy 10'이라는 문자열만 화면에 출력됩니다.

```
>>> interest_stocks = ["Naver", "Samsung", "SK Hynix"]
>>> for company in interest_stocks:
      print("company: Buy 10")

company: Buy 10
company: Buy 10
company: Buy 10
>>>
```

for와 튜플

파이썬의 튜플은 리스트와 유사하게 데이터를 순서대로 저장할 수 있는 자료구조입니다. 다만 리스트와 달리 튜플은 한번 만들어지면 수정할 수 없습니다. for 문에서 리스트를 사용하는 것과 비슷하게 튜플도 사용할 수 있습니다. 다음은 앞서 작성한 코드에서 리스트 대신 튜플을 사용한 코드입니다.

```
>>> interest_stocks = ("Naver", "Samsung", "SK Hynix")
>>> for company in interest_stocks:
      print("%s: Buy 10" % company)

Naver: Buy 10
Samsung: Buy 10
SK Hynix: Buy 10
>>>
```

앞에서 이야기한 것처럼 튜플은 수정할 수 없기 때문에 리스트보다 빠릅니다. 따라서 관심 종목을 변경할 필요가 없는 경우에는 이를 튜플로 관리하고 파이썬의 for 문을 사용하는 것이 좋고, 관심 종목을 변경할 필요가 있는 경우에는 데이터를 리스트로 관리하는 것이 좋습니다.

for와 딕셔너리

3장에서 배운 자료구조 중 리스트와 튜플은 데이터를 순서대로 저장하는 자료구조입니다. 이와 달리 딕셔너리는 키-값 쌍을 저장하는 구조로서 데이터를 입력한 순서대로 저장하지 않습니다. 따라서 딕셔너리에 대해 for 문을 사용하려면 조금 특별한 방법이 필요합니다. 그렇다면 어떨 때 for 문과 딕셔너리를 함께 사용하면 좋을까요?

앞의 예제에서는 관심 종목에 대해 동일하게 주식을 10주씩 매수했습니다. 그런데 이번에는 종목마다 다른 수량만큼 주식을 매수하고자 합니다. 프로그램으로 주식을 매수하기 전에 종목별로 얼마만큼 매수할 것인지를 미리 표현해야 한다면 딕셔너리를 통해 다음과 같이 쉽게 데이터를 표현할 수 있습니다.

여전히 관심 종목은 'Naver', 'Samsung', 'SK Hynix'이지만 매수 수량은 종목마다 다른 것을 확인할 수 있습니다.

```
>>> interest_stocks = {"Naver":10, "Samsung":5, "SK Hynix":30}
>>>
```

interest_stocks라는 딕셔너리에 있는 키-값 쌍에 대해 매수 동작을 반복적으로 수행하려면 for 문을 사용하면 됩니다. 다만 딕셔너리의 한 원소에는 키와 값이 있기 때문에 for 문 다음에 있는 변수가 한 개가 아니라 두 개를 적어야 합니다. 그리고 딕셔너리 이름을 적은 다음 .items()를 붙여야 합니다.

```
>>> for company, stock_num in interest_stocks.items():
      print("%s: Buy %s" % (company, stock_num))

SK Hynix: Buy 30
Naver: Buy 10
Samsung: Buy 5
>>>
```

위 코드에서 화면 출력을 수행하는 코드를 보면 company와 stock_num이라는 두 변수가 바인딩하는 값을 출력하기 위해 두 개의 %s를 사용했고 % 기호 다음에 실제 변수명을 지정할 때는 (company, stock_num)과 같이 튜플로 표현한 것을 볼 수 있습니다.

3장에서는 딕셔너리의 키-값 쌍만을 구하기 위해 keys와 values라는 메서드를 사용했습니다. 딕셔너리의 keys 메서드를 이용하면 다음과 같이 코드를 작성할 수도 있습니다. 앞에서 설명한 것처럼 interest_stock.keys()의 반환값은 dict_keys(['SK Hynix', 'Naver', 'Samsung'])인데, 이것은 리스트는 아니지만 이를 명시적으로 리스트로 변환할 필요는 없습니다. 파이썬의 for 문은 dict_keys 타입에 대해서도 제대로 동작하기 때문이죠.

```
>>> interest_stocks = {"Naver":10, "Samsung":5, "SK Hynix":30}
>>> for company in interest_stocks.keys():
      print("%s: Buy %s" % (company, interest_stocks[company]))

SK Hynix: Buy 30
Naver: Buy 10
Samsung: Buy 5
>>>
```

05 while 문

4.4절에서는 반복적인 작업을 파이썬의 for 문을 통해 처리했습니다. 파이썬에는 for 문 말고도 동일하게 반복문을 작성할 수 있는 while 문이 있습니다. for 문으로도 충분히 반복문을 작성할 수 있는데 왜 while 문이라는 것도 있을까요? 일반적으로 파이썬의 for 문은 반복 횟수가 미리 정해져 있거나 리스트, 튜플, 사전과 같은 파이썬 자료구조와 함께 사용됩니다.

반면 while 문은 반복해야 할 횟수가 특별히 정해지지 않고 어떤 조건을 충족하는 동안만 실행될 때 주로 사용합니다. 당연한 말이지만 for 문을 사용하나 while 문을 사용하나 결과는 동일합니다. 다만 반복을 수행하는 구조를 보고 for 문을 사용하는 것이 프로그래밍하기 더 쉬워 보이면 for 문으로 작성하는 것이고, while 문이 더 쉬워 보이면 while 문으로 작성하면 됩니다.

먼저 for 문에서 했던 것처럼 0부터 10까지의 숫자를 while 문을 사용해 화면에 출력해 보겠습니다. 다음 코드를 보면 먼저 i라는 변수가 초깃값으로 0을 바인딩합니다. while 문에는 조건문(i <= 10)이 있는데, 현재 i가 바인딩하는 값이 10보다 작거나 같으므로 해당 조건을 만족합니다. 조건문이 참이므로 while 문에 속하는 문장을 순서대로 실행합니다. 즉, print(i)라는 문장이 먼저 실행되고 현재 i라는 변수가 바인딩하고 있는 값인 0이 화면에 출력됩니다. 그리고 'i = i+1'이라는 문장이 수행되어 i가 바인딩하는 값이 0에서 1로 바뀝니다.

```
>>> i = 0
>>> while i <= 10:
        print(i)
        i = i+1

0
1
2
3
4
5
6
7
8
9
10
>>>
```

while 문에 속하는 모든 문장이 수행된 경우 다시 while 문의 조건문으로 이동해서 조건식을 다시 판단합니다. 이번에는 i라는 변수가 1을 바인딩하고 있지만(앞에서 1만큼 증가했으므로) 여전히 10보다는 작거나 같은 값입니다. 즉, 여전히 조건식은 참이 됩니다. 따라서 while 문에 속하는 문장을 차례로 수행합니다. 이러한 방식으로 i라는 변수가 바인딩하는 값이 10일 때까지 while 문 내부의 문장을 순서대로 수행하고 i가 11이 될 때 더는 조건식을 만족하지 않아 while 문을 빠져나오게 됩니다.

위 코드에서 i라는 변수는 초깃값으로 0을 바인딩하고 있었습니다. 그러나 while 문에서 조건문을 만족할 때마다 값이 1씩 증가했습니다. 정말로 i 값이 증가했는지 확인해 봅시다. 다음 코드와 같이 파이썬 IDLE에서 i 값을 확인해보면 i 값이 11로 증가한 것을 확인할 수 있습니다. 앞서 설명한 것처럼 i 값이 11이 되면 더는 while 문의 조건식(i <= 10)을 만족하지 않기 때문에 while 문을 빠져나오게 된 것이죠.

```
>>> i
11
>>>
```

while 문을 이용한 상한가 계산

while 문에 좀 더 익숙해지기 위해 주식과 관련된 간단한 코드를 작성해 보겠습니다. '위키북스'의 주식 1주를 보유하고 있고, 주가가 10,000원이었는데 5일 연속으로 상한가를 기록했다고 가정해봅시다. 첫 번째 상한가 후의 가격은 다음과 같이 계산할 수 있을 것입니다.

```
>>> 10000 + 10000 * 0.3
13000.0
>>>
```

보다시피 10,000원에서 30% 오른 가격을 더해줌으로써 상한가를 계산한 것입니다. 계산 결과는 13,000원입니다. 이 금액에서 두 번째 상한가가 발생하면 다시 30%가 더 오르기 때문에 다음과 같이 계산할 수 있을 것입니다.

```
>>> 13000 + 13000 * 0.3
16900.0
>>>
```

두 번의 상한가를 기록하게 되면 10,000원에서 16,900원이 됩니다. 문제는 두 번의 상한가를 계산하는 것도 점점 복잡해지고 있다는 것입니다. 아직도 남아있는 세 번의 상한가 계산을 하자니 머리가 아파집니다.

위와 같은 상황에서 while 문을 이용하면 다섯 번의 상한가 후의 가격도 쉽게 계산할 수 있습니다. 각자 while 문을 이용해 어떻게 프로그래밍할 수 있을지 간단히 생각해본 후 다음 코드를 보시기 바랍니다.

```
>>> wikibooks = 10000
>>> day = 1
>>> while day < 6:
        wikibooks = wikibooks + wikibooks * 0.3
        day = day + 1

>>> wikibooks
37129.3
>>>
```

어떤가요? 여러분이 생각한 것과 비슷하게 코드가 작성됐나요? 먼저 wikibooks라는 변수에 첫 번째 상한가가 일어나기 전날의 종가인 10,000원을 바인딩해 뒀습니다. 그리고 5번의 상한가를 세기 위해 날짜를 저장하는 변수인 day에 초깃값으로 1을 바인딩했습니다.

현재 day는 1을 바인딩하고 있으므로 'day < 6'의 조건을 만족합니다. 조건문을 만족하기 때문에 while 문 안쪽에 들여쓰기된 코드가 순서대로 수행됩니다. 따라서 먼저 wikibooks가 바인딩하는 값에 0.3을 곱한 값이 계산되고 이 값이 현재 wikibooks가 바인딩하고 있는 값에 더해집니다. 이렇게 최종 계산된 값을 다시 wikibooks라는 변수가 바인딩하게 됩니다. 따라서 wikibooks는 13,000원을 바인딩하게 됩니다. 그런 후 day라는 변수의 값이 1만큼 증가해서 2가 됩니다.

이러한 동작은 day 값이 5일 때까지 반복되므로 총 5번의 상한가가 발생한 금액을 wikibooks라는 변수가 바인딩하게 됩니다. 위 코드를 보면 10,000원에서 5일 연속 상한가가 발생하면 약 37,000원 정도가 되는 것을 확인할 수 있습니다.

while 문의 동작 원리를 정리하면 다음과 같습니다.

1) while 문의 조건 확인

2) 조건을 만족하면 while 문 내부의 코드를 차례로 수행하고, 조건을 만족하지 않으면 while 문의 다음 문장을 수행

3) 2)의 과정에서 while 문 내부의 코드를 차례로 수행한 경우 다시 while 문의 조건 확인으로 이동

while과 if

이번에는 파이썬의 반복문 중 하나인 while 문과 분기문인 if 문을 함께 사용하는 것에 대해 알아보겠습니다. 이러한 코드는 반복적인 작업을 수행하다가 특정 조건일 때는 다르게 처리하고자 할 때 많이

사용합니다. 예를 들어, 앞에서는 while 문을 이용해 0부터 10까지의 숫자를 화면에 출력해봤습니다. while 문과 if 문을 함께 사용하면 0부터 10까지의 숫자 중 홀수만 출력하는 프로그램을 구현할 수 있습니다.

먼저 0부터 10까지의 숫자에 대해 반복해야 하므로 while 문을 사용하겠습니다. while 문에서 조건을 만족하는 경우 while 문 내에서 들여쓰기된 문장을 순서대로 수행하게 되는데, 이때 홀수인 경우에만 화면에 숫자를 출력하면 됩니다. 그런데 어떻게 숫자가 홀수인지 또는 짝수인지 확인할 수 있을까요?

프로그래밍할 때 어떤 숫자가 홀수인지 짝수인지 확인하는 일반적인 방법은 숫자를 2로 나눈 후 나머지를 보는 것입니다. 예를 들어, 1을 2로 나눈 나머지는 1이 되므로 홀수입니다. 2는 2로 나누면 나머지는 0이 되므로 짝수입니다. 3을 2로 나누면 나머지는 1이 되므로 홀수입니다.[1]

다음 코드는 while 문과 if 문을 사용해 0부터 10까지의 숫자 중에서 홀수만 화면에 출력하는 프로그램을 구현한 것입니다. 전반적으로 저자가 설명한 방식으로 구현돼 있으며, 한 가지 특이한 점은 num이라는 변수가 바인딩하고 있는 값을 1만큼 증가시킬 때 'num = num + 1' 대신 'num += 1'이라고 표현했다는 것입니다. 사실 두 표현은 의미가 같은데, 'num += 1'과 같은 표현도 많이 사용되기 때문에 눈으로 익혀두고 자주 사용해 보기 바랍니다.

```
>>> num = 0
>>> while num <= 10:
        if num % 2 == 1:
            print(num)
        num += 1

1
3
5
7
9
>>>
```

break와 continue

for나 while 문을 이용해 어떤 작업을 반복적으로 수행하다가 특정 조건일 때 반복문 자체를 빠져나와야 할 때가 있습니다. 예를 들어, 주식 프로그램을 만들었다면 해당 프로그램은 온종일 켜져 있다가 증시 거래 시간이 지나면 동작을 멈춰야 하는 것들이 이 경우에 해당합니다.

1 파이썬에서 나머지는 % 연산자를 통해 구할 수 있습니다.

먼저 프로그램을 계속해서 실행하게 하려면 앞에서 배운 while 문을 이용해 계속해서 반복하게 해야 합니다. while 문에서 계속 반복하게 하려면 while 문의 조건이 항상 참이 되도록 만들면 됩니다. 그러면 while 문 내의 문장을 실행하고 다시 조건을 판단할 때 항상 참이 되므로 while 문 내의 코드가 무한히 실행되는 것입니다.

이처럼 무한히 실행되는 반복문을 무한 루프(infinite loop)라고 하며, 간단하게는 다음 코드와 같이 작성할 수 있습니다. 이 코드를 실행하면 파이썬 IDLE에 계속해서 'Find stocks'가 출력됩니다. 'Find stocks'가 무한히 출력되도록 프로그래밍했기 때문에 무한히 출력되는 것입니다. 프로그램을 강제로 종료하려면 파이썬 IDLE에서 Ctrl + C 키를 눌러 중단해야 합니다.

```
>>> while 1:
        print("Find stocks")
```

파이썬의 반복문(while 문 또는 for 문)에서 빠져나오려면 break라는 키워드를 사용하면 됩니다. 위 코드에서 무한 루프는 원래 무한히 반복해야 하는데, 여기에 break라는 키워드를 넣으면 break를 만나는 순간 해당 위치에서 반복문을 빠져나오게 됩니다.

다음 코드를 실행해 보면 먼저 while 문의 조건식을 판단하게 되고 조건식에 1이라는 숫자만 있으므로 참으로 판단됩니다. 참고로 거짓이 되려면 0이라는 값이 있어야 하고 0이 아닌 다른 숫자가 있으면 모두 참으로 판단합니다.

while 문의 조건이 참이므로 while 문 내의 코드가 실행되겠죠? 따라서 먼저 print("Find stocks")라는 문장이 실행되고, 그 결과 화면에 'Find stocks'라는 문자열이 출력됩니다. 그다음 break 문이 실행되는데 앞서 설명한 것처럼 break라는 키워드는 루프를 빠져나가게 하므로 더는 코드가 실행되지 않고 프로그램은 종료됩니다.

```
>>> while 1:
        print("Find stocks")
        break

Find stocks
>>>
```

while 문과 break를 이용해 간단한 프로그램을 하나 더 작성해보겠습니다. 이번에도 화면에 0부터 10까지의 숫자를 출력하는데, 이번에는 while 문에서 무한 루프 조건('while 1:')을 사용해보기 바랍니다. 다음 코드를 보면 while 문 내에서 if 문을 이용해 현재 숫자가 10이면 break를 통해 while 문 바깥으로 나가도록 프로그래밍된 것을 볼 수 있습니다.

```
>>> num = 0
>>> while 1:
        print(num)
        if num == 10:
            break
        num += 1

0
1
2
3
4
5
6
7
8
9
10
>>>
```

while 문이나 for 문을 사용하는 반복문에서 break를 사용하면 반복문 전체를 빠져나오게 됩니다. 그런데 반복문 전체를 빠져나오는 것이 아니라 해당 조건만 건너뛰고 싶을 때는 어떻게 하면 될까요? 예를 들어, 화면에 1부터 10까지 출력하는데 5인 경우에만 출력하지 않고 건너뛰는 것처럼 말이죠. 이럴 때 사용할 수 있는 키워드가 바로 continue입니다.

다음 코드를 참조하면 while 문 내에서 먼저 num 값을 증가하게 했습니다. 그리고 증가한 num 값이 5와 같으면 continue가 실행되도록 프로그래밍돼 있습니다. 파이썬 프로그램이 실행되다가 continue를 만나면 그 아래의 코드를 수행하지 않고 while 문의 조건을 판단하는 곳으로 점프하게 됩니다. 따라서 다음 코드를 실행하면 num 값이 5일 때는 화면에 5가 출력되지 않은 채로 다시 while 문의 조건식으로 이동하게 되고 그다음 'num += 1'에 의해 num 값이 5에서 6으로 증가하게 됩니다.

```
>>> num = 0
>>> while num < 10:
        num += 1
        if num == 5:
            continue
        print(num)

1
2
```

```
3
4
6
7
8
9
10
>>>
```

물론 1부터 10까지의 숫자를 출력하는데 5만 출력하지 않아야 한다고 할 때 위 코드와 같이 continue 문을 사용해야만 하는 것은 아닙니다. 단순히 if 문을 통해 num 값이 5가 아닐 때만 출력하게 하는 편이 구현하기 더 쉽습니다. 다만 복잡한 프로그램을 작성하다 보면 반복문의 특정 조건에서만 코드를 실행하지 않고 다음으로 이동하고 싶을 때가 있는데 이럴 때 continue를 사용하면 된다는 정도만 기억하면 됩니다.

06 중첩 루프

이번 절에서는 중첩 루프(nested loop)에 대해 알아보겠습니다. 여기서 '루프'라는 용어는 반복을 의미하고 '중첩'이라는 것은 여러 개가 겹치는 것을 의미합니다. 즉, 반복문 여러 개가 겹쳐 있는 구조를 중첩 루프라고 합니다. 보통 두 개의 반복문이 겹쳐 있는 '이중 루프'와 세 개의 반복문이 겹쳐 있는 '삼중 루프'를 가장 많이 사용합니다.

다음은 반복문 두 개가 겹쳐 있는 '이중 루프'의 예입니다. 반복문은 for 키워드를 사용했고 for 문 내부에서 조건을 만족할 때 수행되는 문장에는 pass 키워드를 사용했습니다. 참고로 파이썬의 pass 키워드는 아무것도 수행하지 않음을 의미합니다.

```
>>> for i in [1, 2, 3, 4]:
        for j in [1, 2, 3, 4]:
            pass

>>>
```

파이썬 반복문에는 실행될 문장이 최소한 하나라도 있어야 문법 오류가 나지 않으므로 pass를 사용해 문법 오류가 발생하는 것을 방지한 것입니다. 물론 pass 대신 print("") 같은 구문을 넣어도 프로그램의 동작을 간단히 테스트해볼 수도 있습니다.

그렇다면 어떤 경우에 두 개의 반복문을 겹쳐서 사용하는 것일까요? 그림 4.3은 제가 살고 있는 아파트의 각 세대를 간단히 표시해 본 것입니다. 1층에는 101호, 102호, 103호, 104호가 있고 2층에는 201호, 202호, 203호, 204호가 있습니다. 지금까지 배운 반복문으로는 101호, 102호, 103호, 104호에 대해 반복적인 일을 수행할 수 있습니다. 그런데 그림 4.3은 1층에만 세대가 있는 것이 아니라 2층, 3층, 4층에도 각 세대가 있습니다. 우리가 자주 사용하는 엑셀도 데이터가 그림 4.3과 같은 형태로 저장되는데 이러한 구조를 2차원 데이터라고 표현합니다. 이러한 2차원 데이터를 다룰 때 필요한 것이 바로 이중 루프입니다.

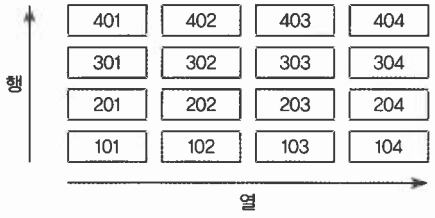


그림 4.3 2차원 데이터의 표현

그림 4.3의 각 세대에 대해 신문을 자동으로 배달하는 로봇을 만든다고 가정해 봅시다. 지금까지 배운 반복문을 사용한다면 다음과 같이 로봇을 프로그래밍해야 할 것입니다.

```
1층에 가서 1층의 각 세대에 신문을 배달
2층에 가서 2층의 각 세대에 신문을 배달
3층에 가서 3층의 각 세대에 신문을 배달
4층에 가서 4층의 각 세대에 신문을 배달
```

1층의 각 세대에 신문을 배달하는 것을 파이썬의 반복문을 사용하면 다음과 같이 표현할 수 있습니다. 이 코드를 복사해서 리스트의 값만 바꿔주면 2층, 3층, 4층에도 신문을 배달할 수 있습니다. 그러나 이 방식은 각 층에 있는 세대에 대해서는 반복문을 사용했지만 층별로는 여전히 반복문을 사용하지 못했기 때문에 코드가 중복되고 길어진다는 단점이 있습니다.

```
>>> for i in [101, 102, 103, 104]:
      print("Newspaper delivery: ", i)

Newspaper delivery:  101
Newspaper delivery:  102
Newspaper delivery:  103
Newspaper delivery:  104
>>>
```

이번에는 이중 루프를 이용해 코드를 작성해 보겠습니다. 이를 위해서는 먼저 그림 4.3과 같은 2차원 데이터를 파이썬의 2차원 리스트로 표현해야 합니다. 3장에서 배운 1차원 리스트는 [ ] 기호 사이에 데

이터가 쉼표로 구분되어 저장되는 형태였습니다. 2차원 리스트는 리스트 안에 정수형, 문자열, 실수형과 같은 데이터가 존재하는 것이 아니라 리스트 타입이 존재하는 구조를 의미합니다. 예를 들어, 그림 4.3은 다음과 같이 2차원 리스트로 표현할 수 있습니다.

```
>>> apart = [[101, 102, 103, 104],[201, 202, 203, 204],[301, 302, 303, 304], [401, 402, 403,
404]]
>>>
```

2차원 리스트도 1차원 리스트와 마찬가지로 인덱싱할 수 있습니다. apart[0]으로 apart 리스트의 첫 번째 원소에 접근할 수 있는데, 리스트의 첫 번째 원소 역시 리스트입니다. 다음 코드를 보면 2차원 리스트에 대한 인덱싱을 통해 그림 4.3의 각 층의 세대 정보가 저장된 리스트에 접근할 수 있습니다. apart[0]에 위치한 값이 정말로 리스트 타입인지 확인하기 위해 type 함수를 사용해보면 정말로 리스트 타입인 것을 확인할 수 있습니다.

```
>>> apart[0]
[101, 102, 103, 104]
>>> apart[1]
[201, 202, 203, 204]
>>> apart[2]
[301, 302, 302, 303]
>>> apart[3]
[401, 402, 403, 404]
>>> type(apart[0])
<class 'list'>
>>>
```

2차원 리스트의 경우 인덱싱을 한번 하면 리스트 안쪽의 리스트에 접근할 수 있습니다. 따라서 한 번 더 인덱싱하면 접근한 리스트 안의 데이터에 접근할 수 있게 됩니다. 예를 들어, 다음 코드를 보면 apart[0]으로 인덱싱했을 때 1층의 세대 정보가 담긴 리스트에 접근했고 연속해서 인덱싱함으로써 각 세대의 정보를 구할 수 있습니다.

```
>>> apart[0][0]
101
>>> apart[0][1]
102
>>>
```

2차원 구조의 데이터를 2차원 리스트로 표현했으니 이중 루프를 이용해 신문 배달을 해보겠습니다. 다음 코드를 보면 먼저 바깥쪽 for 문을 통해 각 층에 대한 리스트를 구합니다. 바깥쪽 for 문이 동작하

는 동안 floor 변수는 [101, 102, 103, 104], [201, 202, 203, 204], [301, 302, 303, 304], [401, 402, 403, 404]라는 리스트를 순서대로 바인딩하게 됩니다. 이를 그림으로 나타내면 그림 4.4와 같습니다. 그림 4.4는 바깥쪽 for 문이 첫 번째로 실행될 때 floor 변수가 2차원 리스트의 첫 번째 원소인 [101, 102, 103, 104] 리스트를 바인딩하는 것을 나타냅니다.

```
>>> for floor in apart:
      for room in floor:
          print("Newspaper delivery: ", room)

Newspaper delivery:  101
Newspaper delivery:  102
Newspaper delivery:  103
Newspaper delivery:  104
Newspaper delivery:  201
Newspaper delivery:  202
Newspaper delivery:  203
Newspaper delivery:  204
Newspaper delivery:  301
Newspaper delivery:  302
Newspaper delivery:  302
Newspaper delivery:  303
Newspaper delivery:  401
Newspaper delivery:  402
Newspaper delivery:  403
Newspaper delivery:  404
>>>
```

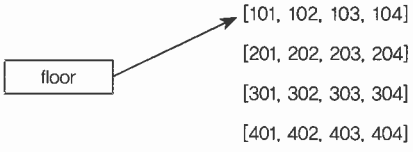


그림 4.4 2차원 리스트 바인딩

for 문은 반복문이 실행될 때 for 문에서 들여쓰기된 문장을 실행하는데, 이중 루프에서는 for 문 안쪽에서 실행할 문장이 역시 또 다른 for 문입니다. 따라서 'for room in floor'라는 안쪽 for 문이 실행됩니다.

바깥쪽 for 문이 처음 실행될 때 floor 변수는 [101, 102, 103, 104]를 바인딩하고 있으므로 안쪽 for 문이 동작하면 room 변수는 101, 102, 103, 104라는 값을 순서대로 바인딩하게 될 것입니다. 안쪽 for

문이 처음으로 실행될 때 room은 101이라는 값을 바인딩한 상태로 for 문 안쪽의 문장을 실행하게 됩니다. 따라서 화면에는 101호에 대한 신문 배달이 출력될 것입니다. 그다음 역시 안쪽 for 문에서 102호에 대한 신문 배달이 이뤄지고, 이러한 작업은 104호까지 계속됩니다.

104호까지 신문 배달이 끝나면 안쪽 for 문은 모두 수행이 완료됐기 때문에 바깥쪽 for 문 기준으로는 두 번째 반복이 시작됩니다. 따라서 floor 변수가 이번에는 [201, 202, 203, 204]를 바인딩한 상태에서 다시 안쪽 for 문을 수행하게 됩니다. 따라서 room이라는 변수는 이번에는 201, 202, 203, 204를 차례대로 바인딩하게 되며 이 값이 순서대로 출력됩니다.

연습 문제

문제 4-1

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요. 참고로 print('*', end='')와 같이 print 함수를 사용하면 줄바꿈 없이 화면에 출력할 수 있습니다.

```
*****
```

문제 4-2

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요(힌트: 중첩 루프 사용).

```
*****
*****
*****
*****
```

문제 4-3

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
*
**
***
****
*****
```

문제 4-4

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
*****
****
***
**
*
```

문제 4-5

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
    *
   **
  ***
 ****
*****
```

문제 4-6

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
*****
 ****
  ***
   **
    *
```

문제 4-7

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
    *
   ***
  *****
 *******
*********
```

문제 4-8

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
*********
 *******
  *****
   ***
    *
```

문제 4-9

중첩 루프를 이용해 신문 배달을 하는 프로그램을 작성하세요. 단, 아래에서 arrears 리스트는 신문 구독료가 미납된 세대에 대한 정보를 포함하고 있는데, 해당 세대에는 신문을 배달하지 않아야 합니다.

```
>>> apart = [[101, 102, 103, 104],[201, 202, 203, 204],[301, 302, 302, 303], [401, 402, 403,
404]]
>>> arrears = [101, 203, 301, 404]
>>>
```

05

함수와 모듈

어느 정도 규모가 있는 프로그램을 개발하다 보면 중복으로 사용되는 코드가 생깁니다. 이럴 때 보통 프로그래밍 경험이 많지 않은 초보자 분들은 이전에 작성된 코드를 단순히 복사한 후 붙여넣는 식으로 프로그램을 작성하곤 합니다. 그러나 이런 식으로 프로그램을 작성하게 되면 점점 코드를 유지 보수하기가 어려워집니다.

예를 들어, 어느 날 이전에 작성한 코드에 문제가 있다는 사실을 알게 되어 해당 문제를 수정한다고 하더라도 이미 해당 코드는 프로그램의 여러 곳에서 복사되어 사용되고 있을 수 있습니다. 이 경우 해당 코드를 하나도 빠짐없이 찾아서 제대로 수정해야 할 것입니다. 어떤가요? 생각만 해도 끔찍하겠죠?

이러한 문제를 막고자 중복 코드를 재사용하고 싶을 때는 단순히 코드를 복사해서 붙여넣는 식으로 프로그램을 작성하는 것이 아니라 함수(function)라는 형태로 코드를 작성하는 것이 좋습니다. 함수라는 것은 한번 잘 작성해두면 두고두고 이를 '호출'해서 사용할 수 있으므로 개발 생산성을 크게 높일 수 있습니다.

파이썬에서 화면에 값을 출력할 때 사용했던 print라는 함수는 여러분이 직접 구현하지 않았음에도 이것을 호출해서 사용했던 것을 생각해보면 함수가 얼마나 유용한지 알 수 있습니다. 리스트나 튜플에 들어있는 원소의 개수를 확인할 때 사용하는 내장 함수 len도 기억나실 겁니다. 이번 장에서는 파이썬에서 제공되는 기본 함수를 살펴보고 유용한 기능을 수행하는 함수도 만들어 보겠습니다.

아울러 파이썬의 모듈에 대해서도 알아보겠습니다. 파이썬의 모듈(module)은 함수보다 더 큰 단위의 코드 묶음을 의미합니다. 보통 함수가 수십 줄 내의 코드로 구성된다면 모듈은 파일 단위의 코드 묶음을 의미합니다. 좀 더 쉽게 설명하자면 마이크로소프트 워드로 글을 쓴다고 했을 때 한 문단이 '함수'에 해당한다면 워드 파일 하나가 바로 모듈에 해당합니다.

01 함수

지금까지 파이썬의 변수, 자료구조(리스트, 튜플, 딕셔너리), 분기문(if, else), 반복문(for, while) 등을 배웠습니다. 이것들은 어떤 프로그래밍 언어를 배우더라도 공통으로 존재하는 필수 구성 요소로서 프로그래밍 언어마다 약간의 차이만 있을 뿐입니다. 따라서 2장 ~ 4장까지의 내용을 반복적인 실습을 통해 익히는 것이 매우 중요합니다. 사실 프로그래밍이라는 것은 처음에는 어렵지만 반복해서 코드를 입력해서 실행하고, 결괏값을 확인하는 과정을 통해 실력이 향상된답니다.

서론이 좀 길었는데, 이번 장에서는 함수(function)를 배우겠습니다. 함수에 대해서는 아마 초등학교나 중학교 수학 시간에 이미 배웠을 것입니다. 수학 분야에서는 그림 5.1처럼 입력으로 어떤 값을 넣으면 내부적으로 어떤 처리를 수행하고 최종 결괏값을 출력하는 것을 함수라고 합니다. 여기서 중요한 점은 함수 내부가 어떻게 구현돼 있는지 몰라도 입력을 넣으면 뭔가 출력이 나온다는 점입니다. 그래서 보통 함수 내부는 블랙박스로 생각하고 단순히 입력값과 출력값의 관계만을 생각합니다.

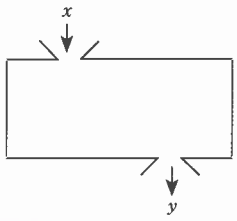


그림 5.1 함수의 입력과 출력

그렇다면 프로그래밍 분야에서 함수란 어떤 것일까요? 결론부터 말씀드리면 프로그래밍 분야의 함수도 수학 분야의 함수와 같습니다. 단지 프로그래머들은 입력값을 넣었을 때 원하는 출력값을 만들어주는 함수를 지금까지 배운 파이썬 문법을 이용해 구현합니다. 프로그래밍 분야에서도 함수 내부의 구현은 블랙박스와 같아서 함수를 사용하는 입장에서는 이를 알 필요가 없습니다. 단순히 함수에 입력값을 넣으면 우리가 원하는 출력값을 얻을 수 있다는 점을 아는 것이 중요합니다.

지금부터는 코드를 작성해보면서 함수를 익혀 보겠습니다. 파이썬 IDLE를 이용해 화면에 "대신증권"을 출력해 보기 바랍니다.

```
>>> print("대신증권")
대신증권
>>>
```

'대신증권'이라는 문자열을 화면에 출력하신 분들은 이번에는 '대신증권'을 화면에 세 번 출력해보기 바랍니다. 3번으로 횟수가 정해져 있으므로 반복문 중 for 문을 이용하면 쉬울 것 같습니다.

```
>>> for i in range(3):
        print("대신증권")

대신증권
대신증권
대신증권
>>>
```

이번에는 함수를 이용해 '대신증권'을 화면에 출력하는 프로그램을 작성하겠습니다. 단순히 '대신증권'을 3번 출력하는 것이 아니라 함수의 입력으로 출력할 '횟수'를 받은 후 그 '횟수'만큼 '대신증권'을 출력하는 함수를 만들어 보겠습니다.

파이썬에서 함수를 만들 때 중요한 점은 함수의 입력, 출력, 함수의 동작을 파악하는 것입니다. 아래와 같이 대략적인 함수 명세서를 구성해봅시다. 참고로 프로그램을 작성할 때 무조건 손이 가는 대로 프로그램을 작성하기보다는 잠시 컴퓨터에서 떨어져서 구현할 내용에 대해 생각하고 어떻게 구현할지를 설계한 후 프로그래밍하는 습관을 기르는 것이 좋습니다.

함수의 입력: 출력 횟수

함수의 출력: '대신증권'이라는 문자열(횟수만큼)

함수의 동작: 출력 횟수만큼 '대신증권' 문자열을 출력

파이썬에서는 함수를 만들 때 def라는 키워드를 사용합니다. 참고로 def는 definition이라는 영어 단어의 줄임말입니다. definition에 '정의'라는 뜻이 있는 것을 생각하면 파이썬에서 함수를 정의할 때 def 키워드를 사용하는 것은 적절한 작명센스 같습니다.

변수명을 적절히 지정해야 하는 것처럼 def 키워드 다음에는 함수의 기능과 연관된 적당한 함수명을 지정해야 합니다. 여기서는 다음과 같이 함수명을 print_ntimes라고 지정했습니다.

```
>>> def print_ntimes():
        print("대신증권")

>>>
```

함수명('print_ntimes') 다음에는 함수의 입력 값을 넣을 괄호가 나오는데, 현재 구현에서는 입력 값이 없으므로 괄호 안에 아무 값도 없습니다. 함수의 몸통 즉, 함수의 구현 부분을 살펴보면 화면에 '대신증권'을 출력하는 print 문이 있습니다. 파이썬에서 함수를 정의할 때도 분기문, 반복문처럼 키워드가 있는 줄의 끝에 콜론(:)이 있고, 실행될 코드 블록은 들여쓰기돼 있어야 합니다.

위에서 구현한 함수는 앞서 작성한 함수 명세에 대한 구현과는 상당한 거리가 있습니다. 일단 함수의 입력으로 출력 횟수를 받아야 하는데 해당 기능이 구현돼 있지 않습니다. 먼저 출력 횟수를 의미하는 n 변수를 함수의 입력으로 받도록 코드를 수정합니다. n과 같이 함수에 어떤 값을 넘겨줄 때 사용하는 변수를 함수 파라미터 또는 함수 인자라고 합니다.

```
>>> def print_ntimes(n):
        print("대신증권")

>>>
```

파이썬에서 함수라는 것을 일단 만들기는 했으니 만들어진 함수를 호출해 보겠습니다. 파이썬에서 기본적으로 제공하는 print 함수를 사용할 때 print("hello")처럼 함수명을 입력했던 것처럼 함수를 호출하려면 여러분이 만든 함수의 이름을 파이썬 프롬프트에 입력하면 됩니다. 물론, 여러분이 만든 함수는 인자 값이 하나 있으므로 print_ntimes()가 아니라 print_ntimes(3)처럼 괄호 안에 값을 하나 넣어야 합니다.

```
>>> print_ntimes(3)
대신증권
>>> print_ntimes(2)
대신증권
>>>
```

위 코드를 보면 뭔가 제대로 동작하지 않는 것을 바로 알 수 있습니다. 함수를 호출할 때 함수의 인자 값으로 출력 횟수를 입력했지만(각각 3과 2) 입력된 출력 횟수와는 상관없이 '대신증권'이라는 문자열은 항상 한 번만 출력됩니다. 이것은 그림 5.2처럼 함수의 입력 값인 n 값과 상관없이 함수 본체에서는 항상 한 번만 출력하도록 구현했기 때문입니다.

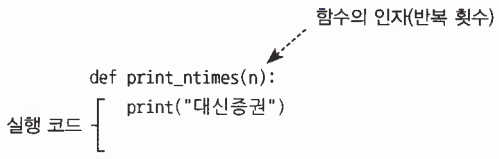


그림 5.2 파이썬 함수 호출 예

함수의 입력 값인 n만큼 문자열이 출력되게 하려면 함수 내에서 for 문을 사용하면 됩니다. for와 range()를 사용해 반복문을 만들고 반복문에서 수행할 문장에 print("대신증권")을 적어주면 됩니다. for 문을 3번 반복하게 하려면 range(3)을 사용하면 되는데 print_ntimes 함수에서는 함수 인자 n을 통해 반복 횟수가 입력되므로 함수 인자인 n만큼 반복하려면 range(n)을 사용하면 됩니다.

```
>>> def print_ntimes(n):
        for i in range(n):
            print("대신증권")

>>>
```

수정한 함수가 제대로 동작하는지 테스트해 보겠습니다. 잠깐 함수 내부의 구현은 잊고 함수의 입력 값으로 출력 횟수를 넣었을 때 출력 횟수만큼 '대신증권'이 출력되는지 확인해보기 바랍니다. 다음과 같이 함수를 호출해보면 처음으로 만든 함수치고는 상당히 잘 동작하는 것을 확인할 수 있습니다.

```
>>> print_ntimes(1)
대신증권
>>> print_ntimes(2)
대신증권
대신증권
>>> print_ntimes(3)
대신증권
대신증권
대신증권
>>>
```

이처럼 프로그래밍 분야에서 함수라는 것은 처음에 한 번만 잘 만들어 두면 두고두고 호출하는 식으로 원하는 기능을 실행할 수 있습니다. 사실 프로그래밍할 때 꼭 모든 함수를 직접 만들 필요는 없습니다. 여러분 주변이나 인터넷 세상의 누군가가 이미 특정 기능을 하는 함수를 잘 만들어 뒀다면 여러분은 단지 그 함수를 잘 호출하기만 하면 되는 것입니다. 물론 이미 구현된 함수의 인자로 입력 값을 잘 넣어야 올바른 출력 값이 나오므로 함수를 잘 사용하는 것은 여러분의 몫입니다.

02 반환값이 있는 함수

이번에는 어떤 주식 종목의 전날 종가를 입력받아 상한가(30%)를 계산하고 그 값을 반환(return)하는 함수를 작성해 보겠습니다. 이번 절에서 배울 내용 중 중요한 부분은 앞에서 배운 함수와 달리 함수에 반환값이 있다는 점입니다. 참고로 5.1절에서 작성한 함수는 함수를 호출했을 때 함수의 결과가 화면에 '출력'됐지 어떤 결괏값이 '반환'된 것은 아니었습니다.

먼저 상한가를 계산하는 함수의 이름을 cal_upper라고 하고 함수의 입력으로 전날 종가를 받도록 구현해 보겠습니다. 함수 내부는 아직 어떻게 구현하면 좋을지 감이 오지 않으므로 임시방편으로 앞에서 배운 pass를 사용해 다음과 같이 작성합니다.

```
>>> def cal_upper(price):
        pass

>>>
```

새로 함수를 만들었으니 함수를 호출해 보겠습니다. 10,000원에 대한 상한가를 계산하기 위해 cal_upper(10000)을 호출했지만 아직은 아무런 동작도 하지 않는 것을 확인할 수 있습니다.

```
>>> cal_upper(10000)
>>>
```

앞서 설명한 것처럼 상한가는 하루에 최대한 오를 수 있는 한도로 2015년 6월 15일부터는 전날 종가에서 30% 오른 금액입니다. 따라서 상한가를 계산하려면 먼저 전날 종가의 30%에 해당하는 금액을 계산해야 합니다. 다음 코드에서 increment 변수는 전날 종가에 해당하는 price에 0.3을 곱한 결괏값을 바인딩하고 있습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3

>>>
```

위 코드에서 increment 변수가 전날 종가의 30%에 해당하는 금액을 바인딩하고 있으므로 전날 종가(price)에 increment('전날 종가의 30%')를 더하면 대략적인 상한가를 계산할 수 있습니다. (앞서 설명한 것처럼 정확한 상한가 계산은 호가에 따른 절사 과정을 고려해야 합니다.) 지금까지 설명한 내용을 파이썬 코드로 나타내면 다음과 같습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment

>>>
```

위 코드를 보면 먼저 전날 종가의 30%에 해당하는 금액을 increment가 바인딩하고 있고, 이 값을 전날 종가(price)와 더해서 상한가를 계산합니다. 최종적으로 계산된 상한가는 upper_price가 바인딩하게 됩니다.

수정된 함수가 제대로 동작하는지 확인하기 위해 다시 함수를 호출해 보겠습니다. 함수의 인자로 10,000원을 입력하면 30% 오른 금액인 13,000원이 나와야 함수가 제대로 구현된 것입니다. 다음 코드를 보면 함수의 인자로 10000을 사용해 호출했지만 파이썬 IDLE에서는 아무런 변화 없이 명령어 프롬프트(>>>)가 다시 나타나고 있습니다. 함수의 구현 내용을 살펴보면 문제가 없어 보이는데 왜 값이 반환되지 않는 것일까요?

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment

>>> cal_upper(10000)
>>>
```

함수 호출 과정 이해하기

앞서 구현한 코드가 제대로 동작하지 않는 이유를 이해하려면 파이썬이 함수를 호출하는 과정에서 메모리의 상태 변화를 알아야 합니다. 파이썬에서 def 키워드를 통해 함수를 정의하는 것은 말 그대로 함수를 정의하는 것이며, 함수가 정의됐다고 해서 함수가 실행되는 것은 아닙니다. 함수가 실행되려면 함수를 호출해야 합니다. 그렇다면 함수가 호출될 때 메모리에서는 어떤 일이 벌어질까요?

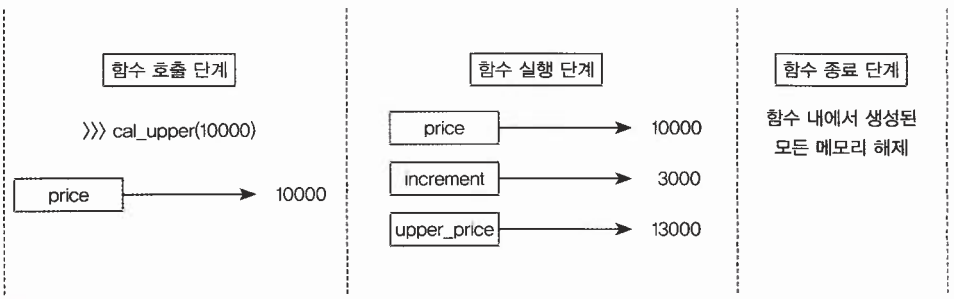


그림 5.3 함수 호출의 3단계

그림 5.3은 cal_upper 함수가 호출됐을 때의 함수 호출 3단계를 나타냅니다. 파이썬에서 함수를 호출하면 함수의 인자에 위치하는 변수(price)가 함수 호출 시 입력된 인자 값(10000)을 바인딩하게 됩니다. 함수 호출 시 입력된 인자 값(10000)은 컴퓨터 메모리의 어딘가에 할당됩니다. 여기까지가 함수 호출 단계의 메모리 상태입니다. 중요한 개념이니 노트에 그림을 그려가면서 이해하시기 바랍니다.

함수 호출의 두 번째 단계는 함수 실행 단계입니다. 이 단계에서는 함수의 수행할 문장이 실행됩니다. cal_upper 함수에서 수행할 문장은 다음과 같습니다.

```
increment = price * 0.3
upper_price = price + increment
```

먼저 첫 번째 문장이 실행되고 이에 따라 price * 0.3의 결괏값이 메모리의 어딘가에 할당됩니다. increment 변수는 이 할당된 값을 바인딩합니다. 여기서 중요한 것은 price * 0.3이라는 연산의 결괏값이 메모리의 어딘가에 실제로 위치한다는 점입니다. 그다음 위의 코드에서 두 번째 문장이 실행되고 이에 따라 price + increment의 결괏값이 메모리의 어딘가에 할당되고, 그 값을 upper_price 변수가 바인딩하게 됩니다. 이를 그림으로 표현하면 그림 5.3의 중간 부분에 해당합니다. 코드가 여기까지 수행됐을 때 price, increment, upper_price 변수가 바인딩하는 값이 실제로 메모리에 저장돼 있다는 개념을 이해하는 것이 중요합니다.

함수 호출의 세 번째 단계는 함수 종료 단계입니다. 함수가 호출되면 함수 호출 단계 및 함수 실행 단계에서 중간값들이 메모리에 할당된다는 사실을 배웠습니다. 이처럼 함수가 호출됨에 따라 메모리가 사용되기 때문에 함수를 계속해서 호출하면 언젠가는 시스템의 메모리가 고갈될 것입니다. 가끔 스마트폰이나 컴퓨터에서 메모리가 부족해서 시스템이 중지되거나 느려지는 경험을 한 적이 있을 것입니다. 이것은 모두 프로그램이 실행되면서 메모리를 사용하기 때문에 발생하는 문제입니다.

파이썬에서는 위와 같은 메모리 관리 문제를 해결하기 위해 할당된 메모리를 자동으로 관리하는 메커니즘을 사용합니다. 한 예로 함수 호출 과정에서 할당된 메모리는 함수 호출이 끝나 더는 사용되지 않을 때 자동으로 메모리에서 삭제됩니다. 따라서 그림 5.3의 함수 종료 단계처럼 앞의 두 단계에서 할당된 메모리는 함수 호출이 끝나면(함수를 호출한 쪽으로 실행 흐름이 되돌아가면) 더는 사용되지 않기 때문에 모두 삭제됩니다. 이처럼 함수가 종료될 때 함수 내부에서 계산된 값이 메모리에서 함께 삭제됐기 때문에 앞서 구현했던 cal_upper 함수가 제대로 동작하지 않았던 것입니다.

그림 5.4는 함수 호출의 3단계를 코드의 실행 흐름 측면에서 보여주는 또 다른 그림입니다. 파이썬에서 함수가 호출되면 함수가 정의된 곳으로 가서 메모리를 할당하면서 코드가 실행되고 함수 내부의 수행할 문장에 대한 실행이 완료되면 그림 5.4의 3번처럼 다시 함수가 호출된 곳으로 되돌아갑니다. 그

림 5.4의 3번 과정에서는 이미 함수의 호출이 종료됐기 때문에 함수 내에서 사용된 모든 메모리가 해제됐고 따라서 함수 내부의 변수를 참조할 수도 없습니다.

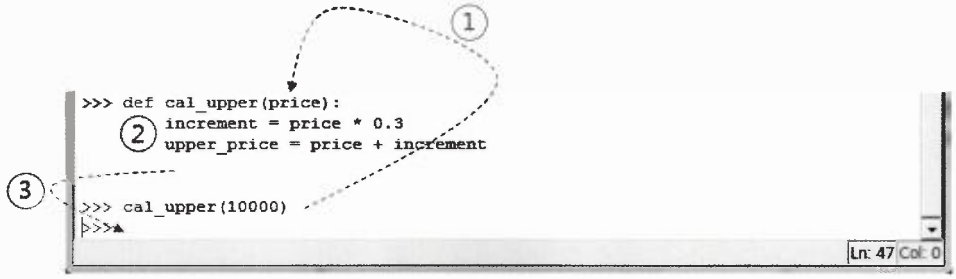


그림 5.4 함수 호출의 3단계(2)

그림 5.5는 함수를 호출한 후 함수 내에서 사용했던 변수를 참조해 본 것입니다. 그림 5.5를 보면 IDLE에서 함수 호출이 종료된 후 price, increment, upper_price와 같이 함수 내부에서 사용되던 변수를 참조하면 모두 "정의되지 않았다(not defined)"라는 오류 메시지가 출력되는 것을 확인할 수 있습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment


>>> cal_upper(10000)
>>> price
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    price
NameError: name 'price' is not defined
>>> increment
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    increment
NameError: name 'increment' is not defined
>>> upper_price
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    upper_price
NameError: name 'upper_price' is not defined
>>>
```

Ln: 62 Col: 4

그림 5.5 변수 참조 오류의 예

지금까지 설명한 파이썬 함수 호출의 원리를 잘 이해했다면 cal_upper 함수를 호출했을 때 왜 어떤 값도 사용할 수 없었는지 이해하셨을 겁니다.

그렇다면 함수 내부에서 계산된 최종 결괏값을 어떻게 함수의 외부로 전달할 수 있을까요? 파이썬은 함수 내부에서 계산된 값을 함수 외부로(함수를 호출한 곳으로) 보내기 위해 return이라는 키워드를

사용합니다. return이라는 영어 단어에 '돌려주다'라는 뜻이 있는 것을 생각하면 함수 내에서 계산된 값을 함수 외부로 보내기 위해 return이라는 키워드를 사용하는 것이 적절해 보입니다.

다음 코드는 수정된 cal_upper 함수 및 함수의 호출 결과입니다. 수정되기 전의 코드와 비교하면 함수의 끝부분에 함수 내에서 계산된 값인 upper_price를 반환하는 코드가 있음을 알 수 있습니다. 이처럼 함수에서 계산된 값은 항상 반환해야(돌려줘야) 함수를 호출한 부분에서 그 값을 사용할 수 있습니다. 수정된 코드에서는 함수가 호출된 다음 파이썬 프롬프트에 13000.0이 나타나는 것을 확인할 수 있습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment
        return upper_price

>>> cal_upper(10000)
13000.0
>>>
```

다음 코드는 수정된 cal_upper 함수를 이용해 전일 종가가 1,000원일 때의 상한가와 5,000원일 때의 상한가를 계산해 본 것입니다. cal_upper 함수는 함수를 호출한 쪽으로 계산된 상한가를 반환하기 때문에 그 값을 바인딩하기 위해 upper_price라는 변수를 사용했습니다.

```
>>> upper_price = cal_upper(1000)
>>> print(upper_price)
1300.0
>>> upper_price = cal_upper(5000)
>>> print(upper_price)
6500.0
>>>
```

함수 호출 과정에서 발생하는 메모리 해제가 잘 이해되지 않는 분들은 다음 이야기를 읽어보기 바랍니다.

"어떤 노인이 평생 모은 돈을 통장(함수의 결괏값에 해당)에 넣어뒀고 통장은 집 어딘가에 숨겨뒀습니다. 노인이 그 돈을 자식에 물려주고자 한다면 자신이 죽기 전에 통장 위치를 자식에게 알려줘야 합니다(값을 반환하는 것에 해당). 만약 노인이 자식에게 통장의 위치를 알려주지 않고 죽어버리면(값을 반환하지 않고 함수를 종료) 자식들은 그 통장에 있는 돈을 영영 찾을 수 없게 됩니다."

어떤가요? 좀 이해되셨나요?

함수의 호출 과정을 '방'을 이용해 다음과 같이 설명하기도 합니다.

"집에 거실과 안방이 있는데 파이썬 코드는 기본적으로 거실에서 실행됩니다. 이 상태에서 함수가 호출되면 안방 문을 열고 들어갑니다. 이때 함수 인자가 거실에서 안방으로 전달됩니다. 안방 문이 닫히고 안방 내에서 어떤 연산이 수행됩니다. 함수 내의 모든 코드가 실행되고 나면 다시 안방 문이 열리고 결괏값을 거실로 넘겨줍니다(함수 반환). 함수 호출이 완료되면 안방 문이 닫히기 때문에 거실에서는 더는 안방에 어떤 값이 있었는지 확인할 수 없습니다. 오직 함수가 호출될 때, 그리고 함수가 종료될 때만 인자 및 반환값의 전달을 통해서만 값이 전달될 수 있습니다. 함수가 종료된 후 안방 문이 닫히면 안방 내에서 생성된 값들은 모두 정리됩니다(메모리 해제)."

여러분이 매수한 종목이 항상 상한가만 되고 하한가가 되는 일이 없으면 좋겠습니다. 그러나 주식을 하다 보면 가끔 매수한 종목이 하한가로 갈 때가 있습니다. 전날 종가에서 하한가가 됐을 때 얼마가 되는지를 알아보기 위해 이번에는 하한가를 계산하는 cal_lower라는 함수를 작성해 보겠습니다.

```
>>> def cal_lower(price):
        decrement = price * 0.3
        lower_price = price - decrement
        return lower_price

>>> cal_lower(1000)
700.0
>>> cal_lower(5000)
3500.0
>>>
```

위 코드를 보면 먼저 함수의 이름이 cal_lower로 바뀌었고 함수 내부에서 사용하는 변수가 increment에서 decrement로 변경된 것을 알 수 있습니다. 그리고 하한가는 전날 종가에서 30%만큼 하락해야 하므로 price - decrement로 작성한 것을 확인할 수 있습니다. 작성된 cal_lower 함수를 통해 전날 종가가 1,000원일 때와 5,000원일 때의 하한가를 계산해보면 정상적으로 값이 계산되는 것을 확인할 수 있습니다.

두 개의 값 반환하기

파이썬을 배우기 전에 C/C++ 같은 프로그래밍 언어를 배운 분들은 함수의 반환값이 오직 하나라고 알고 계실 겁니다. 예를 들어, 상한가와 하한가를 동시에 구하는 함수를 작성하더라도 결괏값을 반환할 때는 두 값을 동시에 반환할 수 없습니다. 그럼 파이썬은 어떨까요? 파이썬도 기본적으로는 함수의 반

환값으로 하나의 객체만 반환할 수 있습니다. 다만 파이썬에서는 반환할 여러 개의 데이터를 앞에서 배운 튜플이라는 자료구조에 넣은 후 반환함으로써 동시에 여러 개의 값을 반환할 수 있습니다.

앞에서 상한가와 하한가를 구하는 함수를 각각 작성해봤습니다. 이번에는 하나의 함수에서 전일 종가를 입력받은 후 상한가와 하한가를 한 번에 계산하는 함수인 cal_upper_lower 함수를 작성해보겠습니다.

```
>>> def cal_upper_lower(price):
        offset = price * 0.3
        upper = price + offset
        lower = price - offset
        return (upper, lower)
>>>
```

위 코드를 보면 상한가와 하한가를 구하는 부분은 기존의 함수와 같습니다. 다만 함수에서 결괏값을 반환할 때 '('와 ')' 기호를 사용해 상한가와 하한가를 하나의 튜플로 구성한 후 반환하는 것을 볼 수 있습니다. 어떤가요? 생각보다 간단하죠?

함수를 만들었으니 함수가 정상적으로 동작하는지 테스트해 보겠습니다. 함수를 호출하는 부분에 함수에서 반환되는 두 개의 값을 받기 위해 (upper, lower)라는 튜플을 사용한 것을 볼 수 있습니다. 10,000원에 대한 상한가와 하한가를 계산해보니 상한가는 13,000원으로, 하한가는 7,000원으로 잘 계산된 것을 확인할 수 있습니다.

```
>>> (upper, lower) = cal_upper_lower(10000)
>>> upper
13000.0
>>> lower
7000.0
>>>
```

03 모듈

이번 절에서는 파이썬 모듈(module)에 대해 배우겠습니다. module이라는 영어 단어를 사전에서 찾아보면 여러 뜻이 있는데, 그중 컴퓨터 분야에서는 '프로그램이나 하드웨어 기능의 단위'라는 의미로 사용되는 것을 확인할 수 있습니다. 파이썬의 모듈도 '프로그램의 기능 단위'를 의미하는데 다만 파이썬은 파일 단위로 작성된 파이썬 코드를 모듈이라고 부릅니다.

예를 들어, 그림 5.6의 stock.py라는 파일에는 두 개의 함수(cal_upper_lower, cal_macd)와 하나의 리스트(mylist)가 있는 것을 볼 수 있습니다. 여기서 함수 및 리스트를 포함하는 파일의 이름이 stock이므로 파이썬에서는 이 파일을 stock 모듈이라고 부릅니다. 참고로 마이크로소프트의 워드 파일에 .docx와 같은 확장자가 붙는 것처럼 보통 파이썬 코드는 .py라는 확장자를 사용합니다.

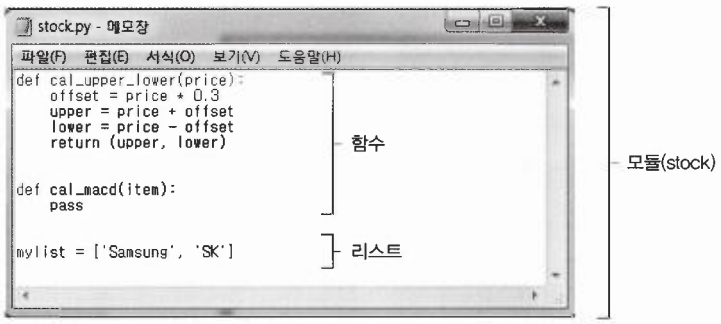


그림 5.6 파이썬 모듈의 구성

그렇다면 파이썬에서는 왜 모듈을 사용할까요? 앞서 함수를 사용하는 이유가 코드의 재사용을 위해서라고 설명해 드렸습니다. 모듈은 함수보다 상위 개념으로 함수와 마찬가지로 코드의 재사용을 위해 사용합니다.

예를 들어, 알고리즘 트레이딩과 관련된 프로그램을 작성할 경우 다른 사람이 작성한 함수를 재사용하고 싶다면 어떻게 해야 할까요? 먼저 함수의 코드를 복사한 후 여러분의 프로그램에 붙여넣은 다음 함수를 호출해야 합니다. 다른 사람이 작성한 함수를 사용하기 위해 함수 코드 전체를 복사한 후 붙여넣는 작업은 여간 번거로운 작업이 아닙니다. 또한 시간이 흐른 후에 해당 함수가 업데이트되는 경우를 생각하면 함수가 업데이트될 때마다 코드를 새로 복사한 후 붙여넣기해야 합니다.

파이썬의 모듈을 사용하는 경우에는 함수가 정의돼 있는 파일 자체를 복사한 후 모듈을 임포트(import)하기만 하면 해당 파일(모듈)에 구현된 모든 함수 및 자료구조를 사용할 수 있습니다. 즉, 모듈을 사용하면 함수를 사용하기 위해 함수 코드를 여러분의 코드에 복사한 후 붙여넣는 작업을 하지 않아도 됩니다.

모듈 만들기

파이썬에는 다양한 기능을 수행하는 모듈이 기본적으로 제공됩니다(파이썬을 설치할 때 함께 설치됩니다). 파이썬으로 프로그래밍하다 보면 많은 부분을 파이썬의 모듈을 통해 구현합니다. 파이썬 모듈이란

자주 사용되는 기능에 대해 이미 코드로 구현해둔 것이기 때문에 파이썬 모듈을 이용하면 직접 특정 기능을 구현하는 것보다 쉽고 빠르게 프로그램을 개발할 수 있습니다.

파이썬의 기본 모듈은 프로그래밍을 잘하는 개발자들이 이미 개발해 둔 것인데, 이처럼 이미 개발된 모듈을 사용할 수도 있지만 직접 모듈을 만들 수도 있습니다. 이번 절에서는 모듈을 직접 만들어보면서 모듈의 개념을 더욱 정확히 이해해 보겠습니다.

앞에서 상한가와 하한가를 구하는 함수를 파이썬 IDLE에서 구현해 봤습니다. 문제는 파이썬 IDLE를 닫으면 더는 해당 함수를 사용할 수 없게 된다는 점입니다. 여러분이 구현한 상한가와 하한가를 계산하는 함수를 파일로 작성해 이를 모듈로 만들어 봅시다. 이렇게 모듈로 만들어 두면 두고두고 사용할 수 있습니다.

그림 5.7과 같이 파이썬 IDLE에서 [File] 메뉴를 선택한 후 [New File]을 선택합니다.

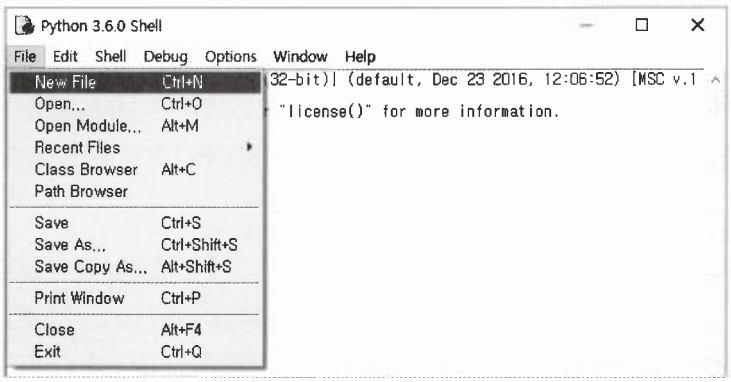


그림 5.7 파이썬 IDLE에서 새 파일 열기

새 창에 다음과 같이 코드를 작성합니다. 앞에서 구현한 상한가와 하한가를 구하는 함수를 그대로 구현하고 추가로 author라는 변수가 pystock이라는 문자열을 바인딩하게 했습니다. 한 가지 주의할 점은 다음 코드는 말 그대로 파이썬 코드이므로 >>>와 같은 파이썬 프롬프트가 없다는 점입니다.

```
def cal_upper(price):
    increment = price * 0.3
    upper_price = price + increment
    return upper_price

def cal_lower(price):
    decrement = price * 0.3
    lower_price = price - decrement
```

```
    return lower_price

author = "pystock"
```

코드를 작성하고 나면 그림 5.8과 같이 [File] 메뉴에서 [Save] 메뉴를 클릭해 파일로 저장합니다.

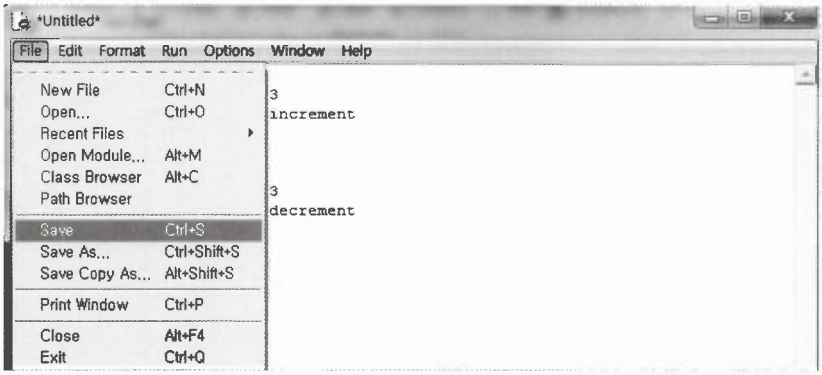


그림 5.8 파이썬 코드 저장

파이썬 코드를 파일로 저장할 때 그림 5.9와 같이 저장될 파일의 경로를 C:/Anaconda3 디렉터리로 선택한 후 파일명을 stock으로 변경하고 [저장] 버튼을 클릭합니다. 따라서 방금 작성한 코드의 저장 경로는 C:/Anaconda3/stock.py가 됩니다.

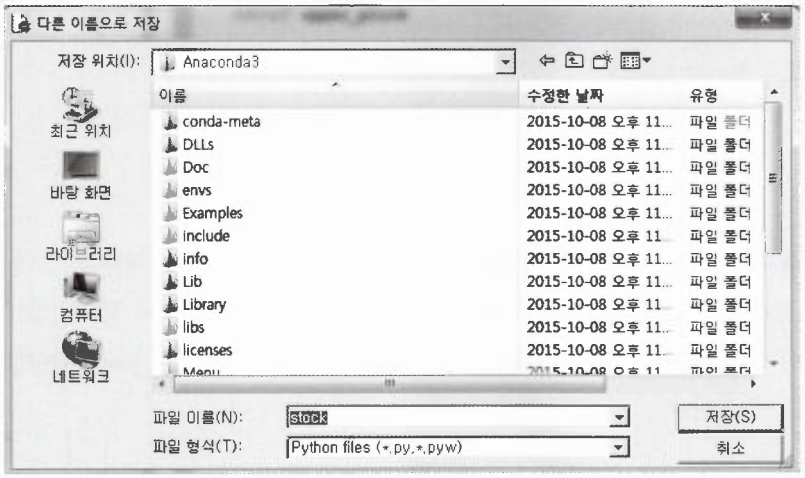


그림 5.9 파이썬 코드 저장(2)

지금까지 작업한 내용을 요약해보면 함수를 구현하고 이를 stock.py라는 이름으로 하드디스크에 저장했습니다. 이제 앞에서 작성한 stock이라는 모듈을 사용해 보겠습니다. 앞서 설명한 것처럼 모듈을 사용하려면 모듈을 임포트해야 합니다.

그림 5.10과 같이 파이썬 IDLE에서 stock 모듈을 임포트해 봅시다. 여기서 오류가 발생하지 않으면 stock 모듈을 잘 만들었고 해당 모듈이 정상적으로 임포트된 것입니다.

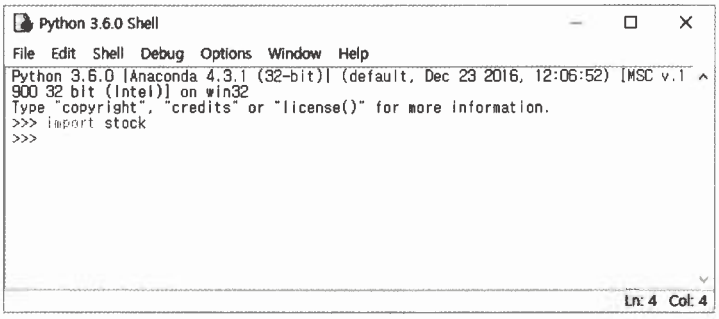


그림 5.10 stock 모듈 임포트

앞에서 구현한 stock 모듈에는 author라는 변수와 cal_upper와 cal_lower라는 함수가 있었습니다. 파이썬에서는 모듈을 임포트하면 모듈 내에 구현된 함수나 변수를 사용할 수 있습니다. 먼저 다음과 같이 stock 모듈 내에 있던 author 변수를 화면에 출력해봅시다. 이때 author라는 변수명을 바로 적으면 안 되고 stock.author라고 적어야 합니다. 모듈 내의 함수나 변수명에 접근할 때는 항상 '모듈명.함수명(변수명)'과 같은 형태로 적어야 합니다.

```
>>> print(stock.author)
pystock
>>>
```

이번에는 stock 모듈 내에 구현된 함수를 호출해 보겠습니다. 함수 역시 먼저 모듈명을 적은 후 '.'를 붙이고 함수명을 적어야 합니다. 이렇게 하는 이유는 해당 함수가 stock이라는 모듈 내에 정의돼 있다는 것을 명시적으로 알려주기 위해서입니다. stock.py 파일에 구현된 cal_upper와 cal_lower 함수를 사용할 수 있음을 확인할 수 있습니다.

```
>>> stock.cal_upper(10000)
13000.0
>>> stock.cal_lower(10000)
7000.0
>>>
```

앞으로 모듈을 개발하고 이를 배포하고자 한다면 모듈에 구현된 함수가 정상적으로 동작하는지 충분히 테스트한 후 모듈을 배포해야 합니다. 이를 위해서는 모듈 파일 내에서 함수를 직접 호출하는 테스트 코드를 구현하는 것이 좋습니다.

다음과 같이 stock.py 파일에 cal_upper와 cal_lower 함수를 호출하는 코드를 추가합니다. 그리고 print(__name__)도 추가합니다.

```
def cal_upper(price):
    increment = price * 0.3
    upper_price = price + increment
    return upper_price

def cal_lower(price):
    decrement = price * 0.3
    lower_price = price - decrement
    return lower_price

author = "pystock"

print(cal_upper(10000))
print(cal_lower(10000))
print(__name__)
```

수정된 stock.py 파일을 실행하려면 그림 5.11처럼 [Run] → [Run Module] 메뉴를 차례로 선택합니다.

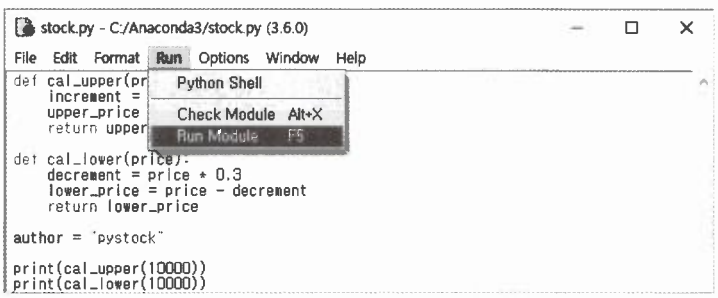


그림 5.11 모듈 코드 실행

파이썬 IDLE에 다음과 같이 stock.py 파일의 실행 결과가 출력됩니다. 10,000원에 대한 상한가와 하한가가 모두 정상적으로 출력되는 것을 확인할 수 있습니다. 그리고 print(__name__)의 결과로 __main__이라는 문자열이 화면에 출력되는 것을 확인할 수 있습니다.

```
>>>
========== RESTART: C:\Anaconda3\stock.py ==========
13000.0
7000.0
__main__
>>>
```

업데이트된 stock 모듈을 다시 임포트해 보겠습니다. stock 모듈을 임포트하면 앞서 추가한 print 구문의 결괏값이 출력됨을 확인할 수 있습니다. 모듈을 테스트하는 입장에서는 모듈 내에 테스트 코드를 넣는 방식이 유용하지만 모듈을 사용하는 입장에서는 임포트할 때 모듈을 테스트하는 코드가 함께 실행되는 것은 불필요한 기능입니다.

```
>>> import stock
13000.0
7000.0
stock
>>>
```

여기서 한 가지 눈여겨볼 점은 모듈을 직접 실행했을 때는 print(__name__)의 결과로 __main__이라는 문자열이 출력되지만 해당 모듈을 임포트했을 때는 모듈의 이름인 stock이라는 문자열이 출력된다는 점입니다.

사실 파이썬에서 __name__이라는 변수는 파이썬 자체에서 사용하는 변수로서 특정 파이썬 파일이 직접 실행된 것인지 또는 다른 파이썬 파일에서 임포트된 것인지를 확인하는 용도로 사용됩니다. 특정 파이썬 파일이 독립적으로 실행됐다면 __name__이라는 변수는 __main__이라는 문자열을 바인딩하며 다른 파일에 임포트된 경우에는 자신의 파일명을 바인딩합니다.

파이썬의 __name__이라는 변수를 이용하면 앞서 모듈을 테스트하는 코드가 모듈을 임포트했을 때 자동으로 호출되는 문제를 해결할 수 있습니다. stock.py 파일에 추가했던 테스트 코드를 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    print(cal_upper(10000))
    print(cal_lower(10000))
    print(__name__)
```

추가된 코드의 의미를 살펴보면 __name__이라는 파이썬 변수가 바인딩하고 있는 값이 '__main__'이라는 문자열이라면 들여쓰기된 세 개의 print 문을 실행합니다. 즉, stock.py 파일이 직접 실행된 경우라면 세 개의 print 문이 실행됩니다. 만약 다른 파일에 stock.py 파일(모듈)이 임포트된 경우라면 __

name__은 모듈의 이름인 stock을 바인딩하므로 if 문의 조건식을 만족하지 않고 따라서 세 개의 print 문도 실행되지 않습니다.

stock.py 파일을 수정했다면 stock 모듈을 다시 임포트해보기 바랍니다. stock.py 파일에 여전히 모듈을 테스트하는 코드가 존재하지만 해당 코드가 실행되지 않음을 확인할 수 있습니다.

```
Python 3.6.0 |Anaconda 4.3.1 (32-bit)| (default, Dec 23 2016, 12:06:52) [MSC v.1900 32 bit (In-
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import stock
>>>
```

그림 5.11과 같이 stock.py 파일을 직접 실행하면 다음과 같이 여전히 테스트 코드가 출력되는 것을 확인할 수 있습니다.

```
>>>
========== RESTART: C:\Anaconda3\stock.py ==========
13000.0
7000.0
__main__
>>>
```

파이썬에서 시간 다루기

알고리즘 트레이딩을 하려면 파이썬에서 시간을 잘 다룰 수 있어야 합니다. 유식한 말로 주식 시장의 데이터를 시계열 데이터라고 하는데 여기서 시계열이란 '일정 시간 간격으로 배치된 데이터의 수열'을 말합니다. 그렇다면 파이썬에서 시간과 날짜를 다루려면 어떻게 해야 할까요?

바로 앞에서 파이썬의 모듈을 공부했다는 점을 생각해보면 파이썬에서 시간을 다루는 가장 쉬운 방법은 모듈을 사용하는 것임을 눈치채셨을 수도 있습니다. 파이썬에는 시간과 날짜를 다루기 위한 time과 datetime이라는 기본 모듈이 있습니다.

먼저 time 모듈을 사용해 보겠습니다. 다음과 같이 time 모듈을 사용하려면 먼저 time 모듈을 임포트해야 합니다. 이미 알고 계시겠지만 여기서 모듈의 이름이 time인 이유는 파일명이 time.py이기 때문입니다. time 모듈에서 현재 시각을 구하는 함수는 time입니다. time은 현재 시각을 반환하는 함수인데 1970년 1월 1일 0시 0분 0초를 기준으로 초 단위로 지난 시간을 알려줍니다.

```
>>> import time
>>> time.time()
```

```
1444532446.467043
>>> time.ctime()
'Sun Oct 11 12:00:50 2015'
>>> type(_)
<class 'str'>
>>>
```

위 코드를 보면 time.time()의 결과가 우리가 알기 어려운 실수 형태로 반환됐음을 확인할 수 있습니다. 사람들이 좀 더 쉽게 읽을 수 있는 시간을 구하려면 ctime 함수를 사용하면 됩니다. ctime 함수의 결과를 확인하면 '2015년 10월 11일 12시 00분 5초'임을 확인할 수 있습니다. 요일은 일요일입니다.

ctime 함수의 반환값인 'Sun Oct 11 12:00:50 2015'는 문자열처럼 보이긴 하지만 정확한 타입을 확인하기 위해 type 함수를 사용했습니다. 여기서 type 함수의 인자로 '_'를 사용했는데 파이썬 IDLE에서 '_'는 가장 최근의 반환값을 바인딩하고 있는 변수입니다. 위 코드에서 type(_) 코드가 나오기 전에 가장 최근의 값은 time.ctime 함수의 반환값이므로 type(_)은 time.ctime 함수의 반환값에 대한 타입을 확인하는 것이 됩니다. 예상대로 <class 'str'>, 즉 문자열 타입임을 알 수 있습니다.

time.ctime의 반환값에서 연도(year)만 구하려면 어떻게 해야 할까요? 앞서 time.ctime의 반환값이 문자열이었으므로 문자열에서 제공하는 메서드를 사용하면 되겠죠?

다음 코드는 time.ctime 함수의 반환값을 cur_time이라는 변수가 바인딩하게 한 후 문자열 객체가 제공하는 split이라는 메서드를 사용해 공백을 기준으로 문자열을 분리했습니다. 분리된 문자열 중 맨 마지막에 존재하는 원소를 화면에 출력하면 연도가 화면에 출력됩니다.

```
>>> cur_time = time.ctime()
>>> print(cur_time.split(' ')[-1])
2015
>>>
```

이번에는 time 모듈의 sleep 함수를 이용해 프로그램을 잠깐 잠재워 보겠습니다. time 모듈의 sleep 함수는 인자로 전달하는 값에 해당하는 초(sec) 동안 코드의 실행을 멈추는 역할을 합니다. 예를 들어, 다음 코드는 0부터 9까지의 숫자를 출력하는데, 1초 간격으로 숫자를 출력하는 코드입니다. 코드를 실행하면 파이썬 IDLE에서 숫자가 1초 간격으로 출력되는 것을 확인할 수 있습니다.

```
>>> for i in range(10):
        print(i)
        time.sleep(1)
```

앞에서 설명한 것처럼 파이썬에서 모듈은 파이썬 파일을 의미합니다. 그러나 모듈의 실행 속도가 중요한 일부 모듈은 파이썬으로 작성되지 않고 C 언어로 작성됩니다. 이 경우에는 해당 모듈의 코드를 직접 볼 수는 없습니다.

다음 코드는 time 모듈과 random 모듈을 임포트한 후 해당 모듈의 위치를 확인하는 코드입니다. time 모듈은 내장 모듈이기 때문에 해당 모듈의 위치가 따로 출력되지는 않지만 random 모듈은 해당 모듈이 위치하는 경로가 출력되는 것을 볼 수 있습니다.

```
>>> import time
>>> time
<module 'time' (built-in)>
>>> import random
>>> random
<module 'random' from 'C:\\Anaconda3\\lib\\random.py'>
>>>
```

파이썬 모듈에는 여러 기능을 수행하는 함수와 변수가 있을 수 있는데 모듈 안에 어떤 함수나 변수가 있는지 어떻게 확인할 수 있을까요? 가장 간단한 방법은 다음과 같이 dir 함수를 사용하는 것입니다. 임포트된 모듈에 대해 모듈명을 사용해 dir 내장 함수를 호출하면 해당 모듈의 구성 요소를 확인할 수 있습니다.

```
>>> import time
>>> dir(time)
['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'altzone',
'asctime', 'clock', 'ctime', 'daylight', 'get_clock_info', 'gmtime', 'localtime', 'mktime',
'monotonic', 'perf_counter', 'process_time', 'sleep', 'strftime', 'strptime', 'struct_time',
'time', 'timezone', 'tzname']
>>>
```

OS 모듈

파이썬에는 기본적으로 제공되는 다양한 모듈이 있습니다. 이러한 모듈은 모두 유용하게 사용되지만 자주 사용되는 모듈 가운데 os라는 모듈이 있습니다. os 모듈은 Operating System의 약자로서 운영체제에서 제공되는 여러 기능을 파이썬에서 수행할 수 있게 해줍니다.

예를 들어, 파이썬을 이용해 파일을 복사하거나 디렉터리를 생성하고 특정 디렉터리 내의 파일 목록을 구하고자 할 때 os 모듈을 사용하면 됩니다. os 모듈은 다양한 기능을 제공하는데 이 중 자주 사용되는 몇 가지만 살펴보겠습니다.

먼저 현재 경로를 구하려면 os 모듈의 getcwd 함수를 사용하면 됩니다. 다음 코드를 보면 현재 경로로 C:\\Anaconda3\\Lib\\idlelib이 반환되는 것을 확인할 수 있습니다. 이는 현재 파이썬 IDLE가 해당 경로에서 실행됐기 때문입니다.

```
>>> import os
>>> os.getcwd()
'C:\\Anaconda3\\Lib\\idlelib'
>>>
```

앞으로 프로그래밍하다 보면 getcwd 함수가 필요가 경우가 종종 있을 것입니다. 일단 getcwd 함수로 현재 경로를 얻을 수 있다는 것만 기억하기 바랍니다.

이번에는 특정 경로에 존재하는 파일과 디렉터리 목록을 구하는 함수인 listdir 함수를 사용해보겠습니다. 다음 코드를 보면 현재 경로인 C:\\Anaconda3\\Lib\\idlelib에 존재하는 파일과 디렉터리 목록이 리스트로 구성된 후 반환되는 것을 확인할 수 있습니다.

```
>>> os.listdir()
['aboutDialog.py', 'AutoComplete.py', 'AutoCompleteWindow.py', 'AutoExpand.py', 'Bindings.py',
'CallTips.py', 'CallTipWindow.py', 'ChangeLog', 'ClassBrowser.py', 'CodeContext.py', 'ColorDel-
egator.py', 'config-extensions.def', 'config-highlight.def', 'config-keys.def', 'config-main.
def', 'configDialog.py', 'configHandler.py', 'configHelpSourceEdit.py', 'configSectionNameDia-
log.py', 'CREDITS.txt', 'Debugger.py', 'Delegator.py', 'dynOptionMenuWidget.py', 'EditorWindow.
py', 'extend.txt', 'FileList.py', 'FormatParagraph.py', 'GrepDialog.py', 'help.txt', 'HISTORY.
txt', 'HyperParser.py', 'Icons', 'idle.bat', 'idle.py', 'idle.pyw', 'IdleHistory.py', 'idlever.
py', 'idle_test', 'IOBinding.py', 'keybindingDialog.py', 'macosxSupport.py', 'MultiCall.py',
'MultiStatusBar.py', 'NEWS.txt', 'ObjectBrowser.py', 'OutputWindow.py', 'ParenMatch.py', 'Path-
Browser.py', 'Percolator.py', 'PyParse.py', 'PyShell.py', 'README.txt', 'RemoteDebugger.py',
'RemoteObjectBrowser.py', 'ReplaceDialog.py', 'rpc.py', 'RstripExtension.py', 'run.py', 'Script-
Binding.py', 'ScrolledList.py', 'SearchDialog.py', 'SearchDialogBase.py', 'SearchEngine.py',
'StackViewer.py', 'tabbedpages.py', 'textView.py', 'TODO.txt', 'ToolTip.py', 'TreeWidget.py',
'UndoDelegator.py', 'WidgetRedirector.py', 'WindowList.py', 'ZoomHeight.py', '__init__.py', '__
main__.py', '__pycache__']
>>>
```

listdir 함수의 인자로 경로를 전달하는 경우 해당 경로에 존재하는 파일과 디렉터리 목록을 구할 수 있습니다. 다음 코드는 listdir 함수의 인자로 'c:/Anaconda3'을 전달한 경우입니다. 이처럼 listdir 함수의 인자로 특정 경로를 지정하는 경우 해당 경로에 있는 파일과 디렉터리 목록이 반환됩니다.

```
>>> os.listdir('c:/Anaconda3')
['conda-meta', 'DLLs', 'Doc', 'envs', 'Examples', 'include', 'info', 'Lib', 'Library', 'libs',
'licenses', 'LICENSE_PYTHON.txt', 'Menu', 'node-webkit', 'pkgs', 'python.exe', 'python34.dll',
```

```
'pythonw.exe', 'qt.conf', 'Scripts', 'share', 'stock.py', 'tcl', 'Tools', 'Uninstall-Anaconda.
exe', 'xlwings32.dll', 'xlwings64.dll', '__pycache__']
>>>
```

윈도우의 파일 탐색기를 이용해 'c:/Anaconda3' 경로로 이동한 후 파일과 디렉터리 목록을 확인하고 위 코드에서 listdir 함수의 반환값과 한 번 비교해 보기 바랍니다.

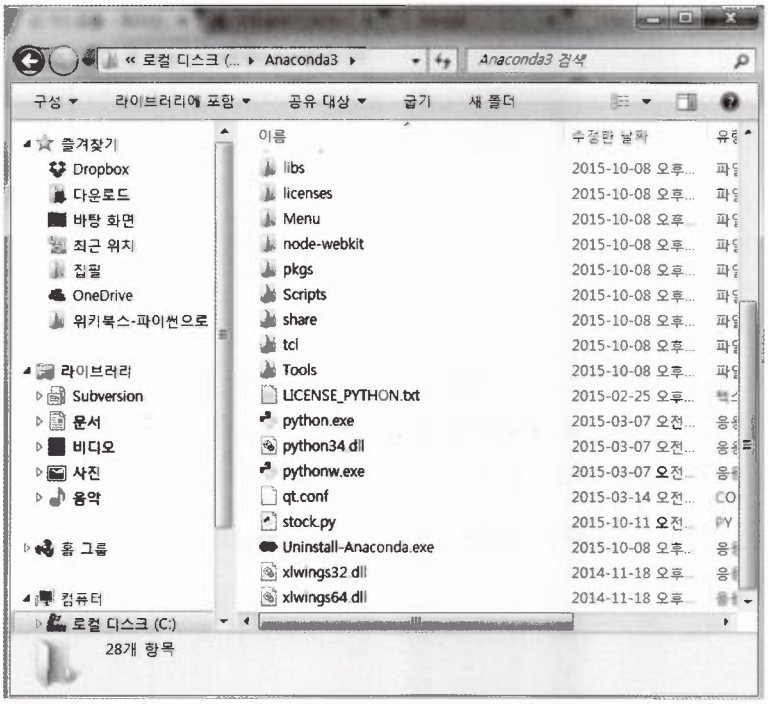


그림 5.12 파일 및 디렉터리 확인

해당 경로에 총 몇 개의 파일 또는 디렉터리가 존재하는지 확인해 보겠습니다. listdir 함수의 반환값은 파이썬 리스트인데, 리스트, 튜플, 문자열 등에서 원소의 개수를 세기 위해서는 len 함수를 사용하면 된다고 앞에서 이야기했습니다. 다음은 listdir 함수의 반환값에 len 함수를 사용한 예입니다. 해당 경로에는 28개의 파일 또는 디렉터리가 존재하는 것을 확인할 수 있습니다. 참고로 type 함수를 이용해 listdir 함수의 반환값 타입을 확인해보면 리스트임을 확인할 수 있습니다.

```
>>> files = os.listdir('c:/Anaconda3')
>>> len(files)
28
>>> type(files)
<class 'list'>
>>>
```

이번에는 'c:/Anaconda3'이라는 경로에 있는 파일 중 확장자가 'exe'로 끝나는 파일만 출력하는 코드를 작성해 보겠습니다. 이를 위해서는 listdir 함수와 if 문, 문자열의 endswith 메서드를 조합하면 됩니다. 다음을 보면 'c:/Anaconda3' 경로에 있는 파일 목록 중 문자열이 'exe'로 끝나는 경우를 출력하는 것을 확인할 수 있습니다.

```
>>> for x in os.listdir('c:/Anaconda3'):
      if x.endswith('exe'):
          print(x)

python.exe
pythonw.exe
Uninstall-Anaconda.exe
>>>
```

모듈을 임포트하는 세 가지 방법

지금까지 모듈을 임포트할 때 'import 모듈명'이라는 규칙을 사용했습니다. 그런데 다른 사람이 작성한 파이썬 코드를 보다 보면 다양한 형태로 모듈을 임포트하는 코드를 볼 수 있습니다. 이번 절에서는 모듈을 임포트하는 방식 간의 차이점을 살펴보겠습니다.

먼저 파이썬 IDLE를 새롭게 실행한 후 dir 함수를 파이썬 IDLE에서 호출해 보겠습니다.

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>>
```

이번에는 os 모듈을 임포트한 후 다시 dir 함수를 호출해 이전 dir 함수의 결괏값과 비교해 봅시다. 차이점이 보이나요? os 모듈을 임포트한 후에는 os라는 이름이 dir 함수의 결괏값 리스트에 추가된 것을 확인할 수 있습니다. 이처럼 os라는 이름이 추가됐기 때문에 os라는 이름을 사용하는 os.listdir()과 같은 표현이 가능했던 것입니다.

```
>>> import os
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'os']
>>>
```

이번에는 모듈을 임포트하는 두 번째 방법을 살펴보겠습니다. 열려 있던 파이썬 IDLE를 닫고 새로 파이썬 IDLE를 실행한 후 다음과 같이 실행해보기 바랍니다. 'from os import listdir' 구문의 의미는 "os 모듈로부터(from) listdir을 임포트하라"입니다. 두 번째 방식으로 모듈을 임포트한 경우

dir()의 결괏값 리스트에는 os가 아니라 os 모듈에 구현된 listdir 함수만 존재하는 것을 확인할 수 있습니다.

```
>>> from os import listdir
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'listdir']
>>>
```

따라서 다음과 같이 os라는 모듈명을 사용하거나 os.listdir()과 같이 os 모듈 내의 listdir 함수를 호출할 경우 오류가 발생합니다. listdir()과 같이 해당 함수를 직접 이용하는 방법만 가능합니다.

```
>>> os
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    os
NameError: name 'os' is not defined
>>> os.listdir()
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    os.listdir()
NameError: name 'os' is not defined
>>> listdir()
```

모듈 내의 함수를 호출할 때 모듈을 임포트하는 첫 번째 방식에서는 '모듈명.함수명' 형태로 사용했는데, 두 번째 방식은 함수 이름만으로도 바로 함수 호출이 가능하므로 프로그래밍할 코드의 수가 적어진다는 장점이 있습니다. 다만 두 번째 방식은 기존에 선언된 변수나 함수와 이름이 충돌할 가능성이 있습니다.

예를 들어, "김철수"라는 이름이 "종로"에도 있고 "종각"에도 있을 때 "종로.김철수", "종각.김철수"와 같은 식으로 표현하면 서로 구분할 수 있습니다. 그런데 지역을 빼고 "김철수"라고 하면 부를 때는 좋지만 누가 누군지 모르는 경우가 발생합니다. listdir이라는 이름을 여러분이 만든 함수나 변수에서 이미 사용 중이라면 'from os import listdir'을 하는 순간 해당 이름의 변수나 함수가 없어지고 os 모듈의 listdir로 대체되는데, 이러한 것들이 나중에는 프로그램의 잠재적인 버그가 될 수 있습니다.

모듈을 임포트하는 세 번째 방식은 모듈 내의 모든 것을 임포트하는 것입니다. 이전과 마찬가지로 IDLE를 새로 실행한 후 다음 코드를 실행해 봅시다. 'from os import *'은 "os 모듈 내의 모든 것을 임포트하라"라는 의미입니다. 따라서 dir 함수의 결괏값을 보면 os 모듈이 지원하는 함수나 변수가 모두 리스트에 포함된 것을 확인할 수 있습니다.

```
>>> from os import *
>>> dir()
['F_OK', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINHERIT', 'O_RANDOM', 'O_RDONLY',
'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEMPORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_
DETACH', 'P_NOWAIT', 'P_NOWAITO', 'P_OVERLAY', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_
SET', 'TMP_MAX', 'W_OK', 'X_OK', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', '_exit', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'close', 'closerange', 'cpu_
count', 'curdir', 'defpath', 'device_encoding', 'devnull', 'dup', 'dup2', 'environ', 'error',
'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp', 'execvpe', 'extsep',
'fdopen', 'fsdecode', 'fsencode', 'fstat', 'fsync', 'get_exec_path', 'get_handle_inheritable',
'get_inheritable', 'get_terminal_size', 'getcwd', 'getcwdb', 'getenv', 'getlogin', 'getpid',
'getppid', 'isatty', 'kill', 'linesep', 'link', 'listdir', 'lseek', 'lstat', 'makedirs', 'mk-
dir', 'name', 'open', 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 'read-
link', 'remove', 'removedirs', 'rename', 'renames', 'replace', 'rmdir', 'sep', 'set_handle_
inheritable', 'set_inheritable', 'spawnl', 'spawnle', 'spawnv', 'spawnve', 'startfile', 'stat',
'stat_float_times', 'stat_result', 'statvfs_result', 'strerror', 'supports_bytes_environ', 'sym-
link', 'system', 'terminal_size', 'times', 'times_result', 'umask', 'uname_result', 'unlink',
'urandom', 'utime', 'waitpid', 'walk', 'write']
>>>
```

세 번째 방식으로 os 모듈을 임포트했을 때 listdir()을 사용하려면 두 번째 방식과 동일하게 listdir()을 바로 사용하면 됩니다. 참고로 세 번째 방식도 두 번째 방식과 마찬가지로 기존에 listdir이라는 이름의 함수가 이미 있는 경우 이름 충돌과 관련된 문제가 발생할 수 있습니다.

그렇다면 모듈을 임포트하는 가장 좋은 방법 또는 일반적으로 권장되는 스타일은 무엇인지 궁금하실 겁니다. 이에 대한 대답은 바로 첫 번째 방식입니다. 모듈을 임포트할 때는 'import 모듈명'을 사용하고, 모듈 내의 함수 또는 변수를 사용할 때는 '모듈명.함수명(변수)'의 형태를 사용하는 것입니다.

그러나 이따금 모듈명이 너무 긴 경우가 있습니다. 또는 모듈명이 너무 짧아서 모듈의 이름만으로는 정확히 구분하기가 어려워 모듈명을 다르게 사용하고 싶을 때도 있을 것입니다. 이 경우 다음과 같이 모듈을 임포트할 때 이름을 변경할 수도 있습니다.

'import os as winos'의 의미는 "os 모듈을 winos로 임포트하라"입니다. dir 함수의 결괏값을 확인해보면 os 대신 winos 항목이 결괏값 리스트에 있는 것을 확인할 수 있습니다. 따라서 os 모듈 내에 있던 함수를 사용할 때도 os.getcwd()가 아니라 winos.getcwd()와 같이 사용해야 합니다.

```
>>> import os as winos
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'winos']
>>> winos.getcwd()
'C:\\Anaconda3\\Lib\\idlelib'
>>>
```

04 파이썬 내장 함수

파이썬에서는 자주 사용되는 함수를 내장 함수(Built-in Functions)라는 이름으로 기본적으로 제공합니다. 예를 들어, 절댓값을 구하거나 원소의 개수를 세는 것과 같은 함수가 내장 함수에 해당합니다. 표 5.1은 파이썬이 제공하는 내장 함수의 목록입니다. 이번 절에서는 이 가운데 자주 사용되는 내장 함수에 대해서만 간단히 알아보겠습니다.

표 5.1 파이썬 내장 함수

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

abs(x) 내장 함수는 정수형 또는 실수형 값을 입력받은 후 해당 값의 절댓값을 반환하는 함수입니다. 다음과 같이 내장 함수의 인자로 음수 값을 입력했을 때 입력 값의 절댓값이 반환되는 것을 확인할 수 있습니다. 다만 정수형 또는 실수형이 아닌 문자열 값이 입력되면 오류가 발생합니다.

```
>>> abs(-3)
3
>>> abs(-3.0)
3.0
>>>
>>> abs("hello")
```

```
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    abs("hello")
TypeError: bad operand type for abs(): 'str'
>>>
```

chr(i) 내장 함수는 유니코드 값을 입력받은 후 그 값에 해당하는 문자열을 반환합니다. 예를 들어, 아스키(ASCII) 코드에서 알파벳 대문자 A는 65, 소문자 a는 97에 해당하는데, 해당하는 정숫값을 chr 함수의 인자로 전달하면 문자열이 반환되는 것을 확인할 수 있습니다.

```
>>> chr(97)
'a'
>>> chr(65)
'A'
>>>
```

enumerate 내장 함수는 입력으로 시퀀스 자료형(리스트, 튜플, 문자열) 등을 입력받은 후 enumerate 객체를 반환합니다. enumerate 내장 함수는 다음과 같이 for 문에서 시퀀스 자료형 내의 값과 인덱스를 동시에 구하고 싶을 때 자주 사용됩니다.

```
>>> for i, stock in enumerate(['Naver', 'KAKAO', 'SK']):
        print(i, stock)

0 Naver
1 KAKAO
2 SK
>>>
```

다음 코드는 enumerate 내장 함수를 사용하지 않고 위의 코드를 다시 구현한 것입니다. 이 경우 시퀀스 자료형인 리스트는 따로 인덱스를 갖고 있지 않으므로 i라는 변수를 통해 따로 인덱스 값을 계산해야 하는 번거로움이 있습니다.

```
>>> i = 0
>>> for stock in ['Naver', 'KAKAO', 'SK']:
        print(i, stock)
        i += 1

0 Naver
1 KAKAO
2 SK
>>>
```

id(object) 내장 함수는 객체를 입력받아 해당 객체의 고윳값을 반환합니다. 고윳값은 파이썬 인터프리터의 구현 방식에 따라 다르지만 보통 객체가 할당된 메모리의 주솟값을 의미합니다. 다음과 같이 a라는 변수와 b라는 변수가 모두 1이라는 값을 바인딩하고 있을 때 a와 b가 서로 같은 객체를 바인딩하고 있는지 확인하려면 id 내장 함수의 반환값이 같은지 확인하면 됩니다. id 내장 함수의 반환값이 같은 경우 두 변수는 서로 같은 객체를 바인딩하고 있음을 의미합니다.

```
>>> a = 1
>>> b = 1
>>> id(a)
1918947744
>>> id(b)
1918947744
>>>
```

len(s) 내장 함수는 리스트, 튜플, 문자열, 딕셔너리 등을 입력받아 해당 객체의 원소 개수를 반환합니다.

```
>>> len(['SK', 'naver'])
2
>>> len('SK hynix')
8
>>> len({1:'SK', 2:'Naver'})
2
>>>
```

list 내장 함수는 문자열이나 튜플을 입력받은 후 리스트 객체로 만들고 해당 리스트를 반환합니다. 다음과 같이 튜플을 리스트로 변환할 때 자주 사용됩니다.

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
>>> list((1,2,3))
[1, 2, 3]
>>>
```

max 내장 함수는 입력값 중 최댓값을 반환합니다. 반대로 min 내장 함수는 입력값 중 최솟값을 반환합니다.

```
>>> max(1,2,3)
3
>>> max([1,2,3])
3
```

```
>>> min(1,2,3)
1
>>> min([1,2,3])
1
>>>
```

sorted 내장 함수는 입력값을 정렬한 후 정렬된 결괏값을 '리스트'로 반환합니다.

```
>>> sorted((4,3,1,0))
[0, 1, 3, 4]
>>> sorted([5, 4, 3, 2, 1])
[1, 2, 3, 4, 5]
>>> sorted(['c', 'b', 'a'])
['a', 'b', 'c']
>>>
```

int(x) 내장 함수는 문자열을 인자로 입력받아 해당 문자열을 정수형으로 변환한 후 반환합니다. 반대로 str(x) 내장 함수는 객체를 입력받아 문자열로 변환합니다. int(x)와 str(x) 내장 함수는 다음과 같이 문자열을 정수형으로, 정수형을 문자열로 변환할 때 자주 사용됩니다.

```
>>> int('3')
3
>>> str(3)
'3'
>>>
```

연습 문제

문제 5-1

두 개의 정숫값을 받아 두 값의 평균을 구하는 함수를 작성하세요.

```
def myaverage(a, b):
    # 함수 구현
```

문제 5-2

함수의 인자로 리스트를 받은 후 리스트 내에 있는 모든 정숫값에 대한 최댓값과 최솟값을 반환하는 함수를 작성하세요.

```
def get_max_min(data_list):
    # 함수 구현
```

문제 5-3

절대 경로를 입력받은 후 해당 경로에 있는 *.txt 파일의 목록을 파이썬 리스트로 반환하는 함수를 작성하세요.

```
def get_txt_list(path):
    # 함수 구현
```

문제 5-4

체질량 지수(BMI; Body Mass Index)는 인간의 비만도를 나타내는 지수로서 체중과 키의 관계로 아래의 수식을 통해 계산합니다. 여기서 중요한 점은 체중의 단위는 킬로그램(kg)이고 신장의 단위는 미터(m)라는 점입니다.

$$BMI = \frac{\text{체중}(kg)}{\text{신장}(m)^2}$$

일반적으로 BMI 값에 따라 다음과 같이 체형을 분류하고 있습니다.

```
BMI < 18.5, 마른체형
18.5 <= BMI < 25.0, 표준
25.0 <= BMI < 30.0, 비만
BMI >= 30.0, 고도 비만
```

함수의 인자로 체중(kg)과 신장(cm)을 받은 후 BMI 값에 따라 '마른체형', '표준', '비만', '고도 비만' 중 하나를 출력하는 함수를 작성하세요.

문제 5-5

사용자로부터 키(cm)와 몸무게(kg)를 입력받은 후 BMI 값과 BMI 값에 따른 체형 정보를 화면에 출력하는 프로그램을 작성해 보세요. 파이썬에서 사용자 입력을 받을 때는 input 함수를 사용하며, 작성된 프로그램은 계속해서 사용자로부터 키와 몸무게를 입력받은 후 BMI 및 체형 정보를 출력해야 합니다(무한 루프 구조).

문제 5-6

삼각형의 밑변과 높이를 입력받은 후 삼각형의 면적을 계산하는 함수를 작성하세요.

```
def get_triangle_area(width, height):
    # 함수 구현
```

문제 5-7

함수의 인자로 시작과 끝을 나타내는 숫자를 받아 시작부터 끝까지의 모든 정수값의 합을 반환하는 함수를 작성하세요(시작값과 끝값을 포함).

```
def add_start_to_end(start, end):
# 함수 구현
```

문제 5-8

함수의 인자로 문자열을 포함하는 리스트가 입력될 때 각 문자열의 첫 세 글자로만 구성된 리스트를 반환하는 함수를 작성하세요. 예를 들어, 함수에 ['Seoul', 'Daegu', 'Kwangju', 'Jeju']가 입력될 때 함수의 반환값은 ['Seo', 'Dae', 'Kwa', 'Jej']입니다.

06

클래스

이번 장에서는 클래스(class)를 배우겠습니다. 사실 기본적인 프로그램을 작성하는 데는 지금까지 배운 변수, 기본 자료구조, 제어문, 함수 정도면 충분합니다. 즉, 이번 장에서 배울 클래스를 몰라도 기본적인 프로그램 정도는 작성할 수 있습니다. 참고로 C 언어처럼 개발된 지 오래된 프로그래밍 언어는 언어 자체에서 클래스를 지원하지 않는 경우도 많습니다.

클래스를 지원하지 않는 C 언어와 달리 1990년대 이후에 개발된 C++, 자바(Java), C# 같은 프로그래밍 언어는 기본적으로 클래스를 지원합니다. 최근에는 클래스를 이용한 객체지향 프로그래밍이 컴퓨터 프로그래밍 분야에서 기본적인 개발 패러다임으로 자리 잡았습니다. 객체지향 프로그래밍을 이용하면 우리가 일상생활에서 생각하는 것과 비슷한 논리로 프로그램을 작성할 수 있기 때문에 프로그램 개발과 유지보수가 간편해집니다. 이러한 장점 때문에 객체지향 프로그래밍은 대규모 프로그램을 개발할 때 자주 사용됩니다.

01 클래스란?

지금까지 클래스라는 개념을 몰라도 프로그래밍하는 데 큰 어려움이 없었습니다. 그러나 이 책의 뒷부분에서 살펴볼 증권사(대신증권, 이베스트증권, 키움증권) API를 사용하려면 반드시 클래스라는 개념을 잘 알아야 합니다. 왜냐하면 국내 증권사에서 제공하는 API가 대부분 클래스를 이용한 객체지향 프로그래밍 방식으로 개발됐기 때문입니다.

물론, 이처럼 직접적인 이유가 아니더라도 클래스를 이용해 프로그래밍하면 데이터와 데이터를 조작하는 함수를 하나의 묶음으로 관리할 수 있으므로 복잡한 프로그램도 더욱 쉽게 작성할 수 있습니다. 간단한 예제를 통해 파이썬의 클래스에 대해 본격적으로 배워보겠습니다.

여러분이 명함을 제작하는 스타트업을 창업했다고 생각해 봅시다. 이 스타트업은 고객의 명함을 제작하기에 앞서 명함에 들어갈 고객 정보(이름, 이메일 주소, 근무지 주소)를 프로그램을 통해 입력받습니다. 파이썬 변수를 이용해 다음과 같이 값을 저장할 수 있습니다.

```
>>> name = "kimyuna"
>>> email = "yunakim@naver.com"
>>> addr = "Seoul"
>>>
```

다음으로 입력받은 데이터를 사용해 실제로 명함을 출력하는 함수를 하나 만들어보겠습니다. 다음과 같이 이름, 이메일 주소, 근무지 주소를 함수 인자로 입력받아 포맷에 맞춰 입력 값을 출력합니다.

```
>>> def print_business_card(name, email, addr):
        print("-------------------------")
        print("Name: %s" % name)
        print("E-mail: %s" % email)
        print("Office Address: %s" % addr)
        print("-------------------------")

>>>
```

작성한 함수가 잘 동작하는지 테스트하기 위해 다음과 같이 함수에 인자값을 넣어 함수를 호출해보기 바랍니다. 참고로 함수의 인자로 사용된 변수는 입력된 값을 바인딩하고 있습니다. 결괏값을 보니 여러분이 작성한 명함 출력 함수가 제대로 동작하는 것을 알 수 있습니다.

```
>>> print_business_card(name, email, addr)
--------------------------
Name: kimyuna
E-mail: yunakim@naver.com
Office Address: Seoul
--------------------------
>>>
```

여러분이 창업한 스타트업은 어느덧 입소문을 타서 드디어 회원 수가 한 명에서 두 명으로 늘어났습니다. 회원이 한 명뿐이었던 시절에는 고객 정보를 바인딩할 변수로 name, email, addr이라는 이름을 이용했습니다. 이제 회원 수가 한 명 더 늘어서 두 번째 회원에 대한 입력 정보는 다음과 같이 새로운 변수를 사용해 저장했습니다. 참고로 같은 이름의 변수를 사용하면 첫 번째 회원에 대한 정보가 손실됩니다.

```
>>> name2 = "DusanBack"
>>> email2 = "dusan.back@naver.com"
>>> addr2 = "Kyunggi"
>>>
```

두 번째 회원의 고객 정보도 잘 입력받았으므로 명함을 출력해보겠습니다. 앞서 명함을 출력하기 위한 print_business_card라는 이름의 함수를 작성해 뒀기 때문에 두 번째 회원의 명함을 출력하려면 기존에 작성한 명함 출력 함수에 적절한 입력값만 전달하고 함수를 호출하면 됩니다.

```
>>> print_business_card(name2, email2, addr2)
--------------------------
Name: DusanBack
E-mail: dusan.back@naver.com
Office Address: Kyunggi
--------------------------
>>>
```

아직은 회원이 두 명이라 걱정이 없는데 앞으로 회원 수가 늘어나면 개인 정보를 어떤 방식으로 저장하는 것이 좋을까요? 그리고 고객 정보에 전화번호 및 팩스 번호를 추가로 저장하고 이를 출력하는 기능을 추가해야 한다면 어떻게 프로그램을 변경하는 것이 좋을지 생각해 봅시다.

지금까지 작성한 프로그램을 생각해보면 보통 어떤 데이터가 존재하고 데이터를 조작하는 함수를 통해 데이터를 목적에 맞게 변경한 후 이를 다시 출력하는 형태였습니다. 앞서 살펴본 명함 출력 프로그램도 사용자로부터 개인정보를 입력받은 후 해당 데이터를 조작하고 이를 출력하는 함수로 구성돼 있었습니다.

명함 출력에 사용된 이름, 이메일, 주소 등을 효과적으로 관리하기 위해 보통 그림 6.1과 같이 각 데이터에 대해 별도의 리스트 자료구조를 사용합니다. 그림 6.1과 같은 자료구조에서 첫 번째 회원에 대한 개인 정보를 얻기 위해서는 `name_list[0]`, `email_list[0]`, `address_list[0]`과 같이 각 리스트별로 인덱싱을 수행해야 합니다. 참고로 그림 6.1과 같이 데이터와 데이터를 처리하는 함수가 분리돼 있고 함수를 순차적으로 호출함으로써 데이터를 조작하는 프로그래밍 방식을 절차지향 프로그래밍이라고 합니다.

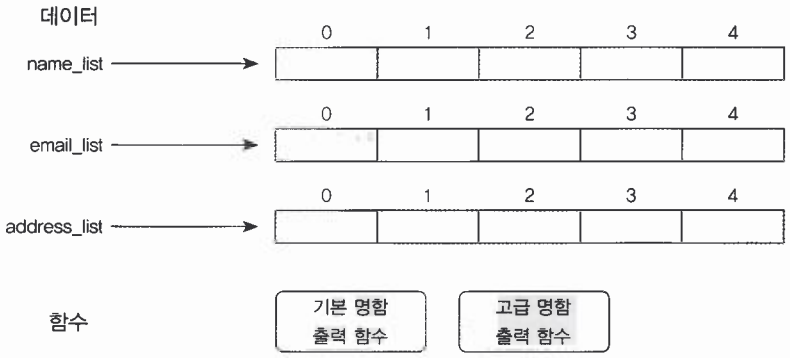


그림 6.1 데이터와 함수

절차지향 프로그래밍 방식과 달리 객체지향 프로그래밍은 객체를 정의하는 것에서 시작합니다. 앞서 예시로 사용한 명함 스타트업을 통해 객체지향 프로그래밍에서 말하는 객체라는 개념을 배워봅시다.

먼저 명함을 구성하는 데이터를 생각해보면 명함에는 이름, 이메일, 주소라는 값이 있습니다. 다음으로 명함과 관련된 함수를 생각해보면 기본 명함을 출력하는 함수와 고급 명함을 출력하는 함수가 있을 것입니다. 객체지향 프로그래밍이란 그림 6.2와 같이 명함을 구성하는 데이터와 명함과 관련된 함수를 묶어서 명함이라는 새로운 객체(타입)를 만들고 이를 사용해 프로그래밍하는 방식을 의미합니다.

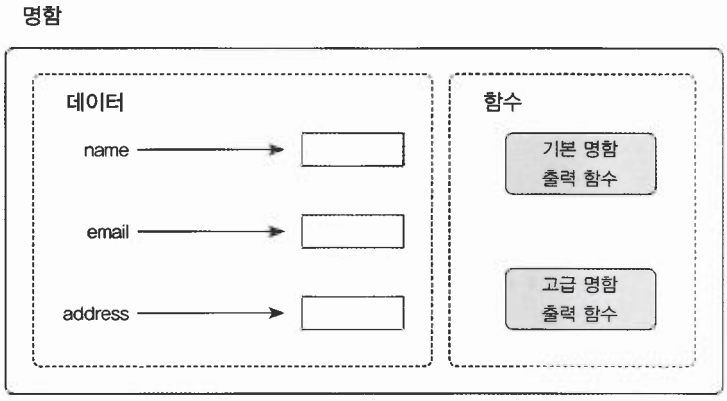


그림 6.2 명함 객체 정의

파이썬을 통해 명함이라는 객체를 만들어 두면 명함 객체가 정수, 실수, 문자열과 마찬가지로 하나의 타입으로 인식되기 때문에 그림 6.3과 같이 여러 명의 명함 정보도 쉽게 관리할 수 있습니다(이전에는 그림 6.1처럼 이름, 이메일, 주소가 각각 따로 관리되는 형태였습니다). 마치 다섯 개의 정숫값을 변수로 바인딩하는 것처럼 다섯 개의 명함 값을 변수로 바인딩하거나 리스트에 명함이라는 객체를 저장하고 있으면 되는 것입니다.

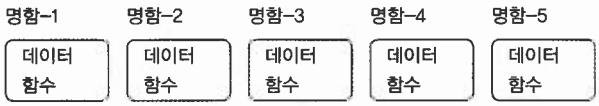


그림 6.3 클래스 객체를 이용해 여러 데이터를 표현

클래스를 이용하면 새로운 타입을 만들 수 있다고 했는데, 사실 앞서 배운 정수, 실수, 문자열, 리스트, 튜플과 같은 기본 자료형과 기본 자료구조도 모두 클래스를 통해 만들어진 타입입니다. 다만 여러분이 만든 것이 아니라 이미 만들어져 있었다는 차이점만 있습니다.

다음 코드를 실행해 여러 파이썬 객체에 대한 반환 타입을 보면 각 타입 앞에 class라는 키워드가 있음을 확인할 수 있습니다.

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type('3')
<class 'str'>
>>> type([])
```

```
<class 'list'>
>>> type(())
<class 'tuple'>
>>>
```

다른 프로그래밍 언어를 배운 분들에게는 조금 생소하겠지만 파이썬에서는 함수도 객체입니다. 다음과 같이 foo라는 함수를 정의한 후 해당 함수의 type을 확인해보니 역시 class라는 키워드가 붙어 있음을 확인할 수 있습니다.

```
>>> def foo():
        pass

>>> type(foo)
<class 'function'>
```

클래스 정의

파이썬에서 함수를 사용하기 전에 먼저 함수를 정의한 것처럼 클래스 역시 사용하기 전에 먼저 정의해야 합니다. 파이썬 IDLE를 실행한 후 다음과 같이 코드를 입력해 봅시다.

```
>>> class BusinessCard:
        pass

>>>
```

파이썬에서 함수를 정의할 때 def라는 키워드를 썼던 것처럼 파이썬에서 클래스를 정의하려면 class라는 키워드를 사용합니다. 클래스를 사용하는 목적이 변수와 함수를 묶어서 하나의 새로운 객체(타입)로 만드는 것이기 때문에 당연히 클래스에 변수나 함수를 포함시켜 클래스를 정의할 수 있습니다. 다만 위 코드에서는 가장 간단한 형태의 클래스 정의를 보여주기 위해 변수나 함수를 넣지 않고 pass라는 키워드만 사용했습니다. pass 키워드를 사용하면 클래스 내부에 아무것도 넣지 않은 상태로 클래스를 정의할 수 있습니다.

파이썬에서 클래스를 정의한다는 것은 새로운 데이터 타입을 정의한 것이기 때문에 이를 실제로 사용하려면 인스턴스라는 것을 생성해야 합니다. 클래스와 인스턴스의 관계를 붕어빵을 만드는 작업에 비유해 보면 클래스를 정의하는 것은 붕어빵을 만들 때 사용하는 기본 틀을 제작하는 것에 해당합니다. 잉어빵, 황금 붕어빵과 같은 차별화된 붕어빵을 만들기 위해 틀을 잘 정의하는 것이지요. 그러나 아무리 틀을 예쁘고 정교하게 만들었다고 해서 붕어빵이 나오는 것은 아닙니다. 실제로 먹을 수 있는 붕어

빵을 만들려면 붕어빵 틀에 반죽을 넣고 붕어빵을 구워야 합니다. 이처럼 붕어빵 틀에 반죽을 넣어서 만들어진 붕어빵이 인스턴스에 해당합니다.

파이썬에서 정의된 클래스를 이용해 인스턴스를 생성하려면 다음과 같이 클래스 이름 뒤에 ()를 넣으면 됩니다. 어떻게 보면 인스턴스를 생성하는 과정은 함수를 호출하는 것과 비슷해 보입니다. 다음 코드에서 첫 번째 문장이 실행되면 그림 6.4와 같이 'BusinessCard'라는 클래스의 인스턴스가 메모리의 0x0302ABB0 위치에 생성되고 card1이라는 변수가 이를 바인딩하게 됩니다.

```
>>> card1 = BusinessCard()
>>> card1
<__main__.BusinessCard object at 0x0302ABB0>
>>>
```



그림 6.4 클래스의 인스턴스 생성

생성된 인스턴스의 타입을 살펴보면 조금 전에 정의했던 BusinessCard 클래스임을 확인할 수 있습니다. 앞서 설명해드린 것처럼 파이썬의 class 키워드를 사용하면 원하는 객체(타입)를 만들 수 있습니다.

```
>>> type(card1)
<class '__main__.BusinessCard'>
>>>
```

클래스에 메서드 추가하기

앞서 정의한 BusinessCard 클래스는 클래스 내부에 변수나 함수가 없었습니다. 그래서 인스턴스를 만들었음에도 해당 인스턴스로 할 수 있는 일이 별로 없었습니다. 이번에는 BusinessCard 클래스에 사용자로부터 데이터를 입력받고 이를 저장하는 기능을 수행하는 함수를 추가해보겠습니다. 참고로 클래스 내부에 정의돼 있는 함수를 특별히 메서드(method)라고 합니다.

다음 코드는 BusinessCard 클래스에 set_info라는 메서드를 추가한 것입니다. 메서드를 정의할 때도 함수를 정의할 때와 마찬가지로 def 키워드를 사용합니다. set_info 메서드는 네 개의 인자를 받는데, 그중 name, email, addr은 사용자로부터 입력받은 데이터를 메서드로 전달할 때 사용하는 인자입니다.

그렇다면 메서드의 첫 번째 인자인 self는 무엇일까요?

```
>>> class BusinessCard:
      def set_info(self, name, email, addr):
          self.name = name
          self.email = email
          self.addr = addr

>>>
```

파이썬 클래스에서 self의 의미를 정확히 이해하는 것이 중요하지만 아직 제대로 설명하기는 조금 이른 감이 있습니다. 일단 클래스 내부에 정의된 함수인 메서드의 첫 번째 인자는 반드시 self여야 한다고 외우길 바랍니다.

위 코드에서 메서드 내부를 살펴보면 메서드 인자로 전달된 name, email, addr 값을 self.name, self.email, self.addr이라는 변수에 대입하는 것을 볼 수 있습니다. 앞서 여러 번 설명한 것처럼 파이썬에서 대입은 바인딩을 의미합니다. 따라서 set_info 메서드의 동작은 그림 6.5와 같이 메서드 인자인 name, email, addr이라는 변수가 가리키고 있던 값을 self.name, self.email, self.addr이 바인딩하는 것입니다.

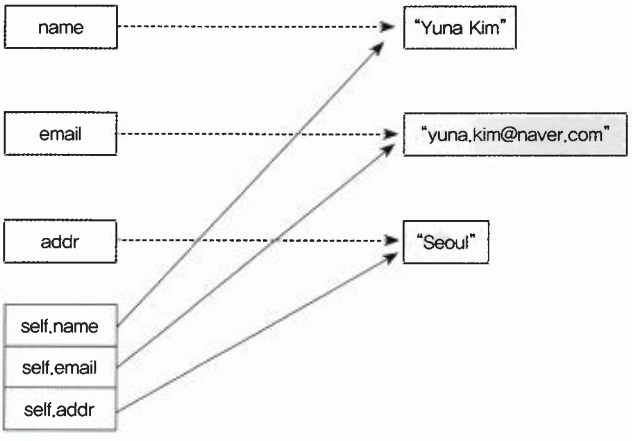


그림 6.5 변수의 바인딩

BusinessCard 클래스를 새롭게 정의했으므로 새롭게 정의된 클래스로 인스턴스를 생성해 봅시다. 붕어빵에 비유해 보면 붕어빵을 굽는 틀을 새롭게 바꿨으므로 새로 붕어빵을 구워보는 것입니다.

```
>>> member1 = BusinessCard()
>>> member1
```

```
<__main__.BusinessCard object at 0x030248F0>
>>>
```

새롭게 정의된 BusinessCard 클래스는 set_info라는 메서드를 포함하고 있습니다. 따라서 member1 인스턴스는 set_info 메서드를 호출할 수 있습니다.

그림 6.6을 보면 member1이라는 클래스 인스턴스를 통해 set_info라는 메서드를 호출할 수 있음을 확인할 수 있습니다. 단, 메서드에 인자를 전달하기 위해 파이썬 IDLE에서 괄호를 입력하면 메서드의 인자가 네 개가 아니라 세 개로 표시됩니다. 앞서 set_info 메서드를 정의할 때는 self, name, email, addr의 네 개의 인자가 사용됐는데 메서드를 호출할 때는 왜 세 개만 사용될까요?

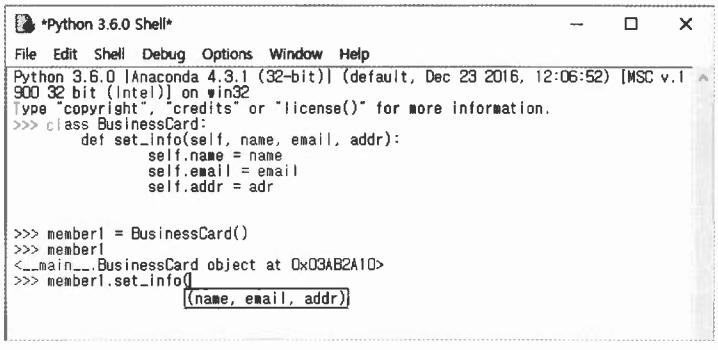


그림 6.6 클래스 인스턴스를 통한 메서드 호출

일단 다음 코드처럼 파이썬 IDLE가 알려주는 대로 세 개의 인자만 set_info 메서드의 입력으로 전달합니다. 항상 그렇듯이 파이썬 IDLE에서 에러가 발생하지 않았다면 정상적으로 코드가 실행된 것을 의미합니다.

```
>>> member1.set_info("Yuna Kim", "yunakim@naver.com", "Seoul")
>>>
```

set_info 메서드는 메서드 인자로 전달된 값을 self.name, self.email, sellf.addr로 바인딩했습니다. 그런데 현재 사용 가능한 변수는 클래스 인스턴스인 member1뿐입니다. 어떻게 하면 member1을 통해 self.name, self.email, self.addr에 접근할 수 있을까요?

self.name, self.email, self.addr과 같이 'self.변수명'과 같은 형태를 띠는 변수를 인스턴스 변수라고 합니다. 인스턴스 변수는 클래스 인스턴스 내부의 변수를 의미합니다. 위 코드에서 member1이라는 인스턴스를 생성한 후 set_info 메서드를 호출하면 메서드의 인자로 전달된 값을 인스턴스 내부 변수

인 self.name, self.email, self.addr이 바인딩하는 것입니다. 클래스를 정의하는 순간에는 여러분이 생성할 인스턴스의 이름이 member1인지 모르기 때문에 self라는 단어를 대신 사용하는 것입니다.

정리해보면 set_info 메서드 내에서 self.name이라는 표현은 나중에 생성될 클래스 인스턴스 내의 name 변수를 의미합니다. 따라서 다음과 같이 인스턴스 이름을 적고 '.'를 붙인 후 인스턴스 변수의 이름을 지정하는 식으로 특정 인스턴스 변수에 접근할 수 있습니다.

```
>>> member1.name
'Yuna Kim'
>>> member1.email
'yunakim@naver.com'
>>> member1.addr
'Seoul'
>>>
```

클래스는 한번 정의해두면 해당 클래스를 이용해 여러 클래스 인스턴스를 만들 수 있습니다. 아직도 개념이 잘 이해되지 않는다면 항상 붕어빵 틀을 생각하세요. 붕어빵 틀을 한 번 만들어두면 붕어빵을 계속해서 구울 수 있는 것과 비슷합니다.

이번에는 두 번째 회원에 대한 정보를 저장하는 인스턴스를 만들어보겠습니다. 다음 코드를 보면 먼저 member2라는 인스턴스를 생성한 후 set_info 메서드를 호출해 인스턴스에 값을 저장하는 것을 확인할 수 있습니다.

```
>>> member2 = BusinessCard()
>>> member2.set_info("Sarang Lee", "sarang.lee@naver.com", "Kyunggi")
>>>
```

새로 생성한 인스턴스 역시 member2라는 이름을 통해 해당 인스턴스 내의 변수에 접근할 수 있습니다.

```
>>> member2.name
'Sarang Lee'
>>> member2.email
'sarang.lee@naver.com'
>>> member2.addr
'Kyunggi'
>>>
```

그림 6.7은 클래스 인스턴스인 member1과 member2를 그림으로 표현해 본 것입니다. member1과 member2는 서로 동일한 이름의 인스턴스 변수인 name, email, addr을 갖고 있지만 각각 다른 데이터를 바인딩하고 있습니다.

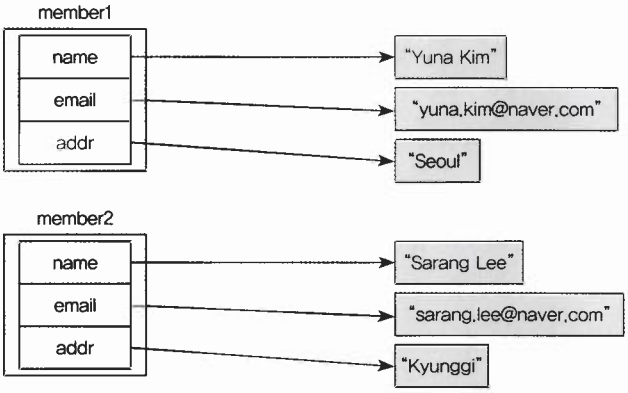


그림 6.7 클래스 인스턴스의 상태

이번에는 BusinessCard 클래스에 명함을 출력하는 메서드를 추가해보겠습니다. 다음과 같이 print_info 메서드를 추가해 BusinessCard 클래스를 새롭게 정의합니다. 새로 추가된 print_info 메서드를 살펴보면 self라는 인자 값만을 사용합니다. 앞서 간단히 설명한 것처럼 파이썬 클래스의 메서드는 항상 self라는 기본 인자를 하나 갖고 있어야 하므로 self를 제외하면 print_info 메서드는 사실 인자가 없는 함수나 마찬가지입니다.

```
>>> class BusinessCard:
        def set_info(self, name, email, addr):
            self.name = name
            self.email = email
            self.addr = addr
        def print_info(self):
            print("--------------------")
            print("Name: ", self.name)
            print("E-mail: ", self.email)
            print("Address: ", self.addr)
            print("--------------------")

>>>
```

print_info 메서드의 내부를 살펴보면 print 함수를 통해 self.name, self.email, self.addr 값을 출력하는 것을 확인할 수 있습니다. 앞서 set_info 메서드에서 이름을 저장할 때 self.name에 저장했으므로 print_info 메서드에서도 self.name을 통해 바로 데이터에 접근하면 됩니다.

BusinessCard 클래스를 재정의했으므로 다음과 같이 새로 인스턴스를 생성해 봅시다.

```
>>> member1 = BusinessCard()
>>> member1.set_info("YunaKim", "yuna.kim@naver.com", "Seoul")
>>>
```

member1이라는 인스턴스에 값이 제대로 저장됐는지 확인하기 위해 member1.name과 같이 인스턴스 변수에 바로 접근할 수도 있습니다. 하지만 BusinessCard 클래스에 print_info 메서드를 정의해뒀기 때문에 다음과 같이 해당 메서드를 호출해 값을 화면에 출력해 볼 수 있습니다.

```
>>> member1.print_info()
--------------------
Name:  YunaKim
E-mail:  yuna.kim@naver.com
Address:  Seoul
--------------------
>>>
```

02 클래스 생성자

6.1절에서 파이썬의 클래스에 대해 배웠습니다. 지금까지 배운 내용을 정리해보면 다음과 같습니다.

- 파이썬의 클래스를 이용하면 프로그래머가 원하는 새로운 타입을 만들 수 있다.
- 생성된 타입은 데이터와 데이터를 처리하는 메서드(함수)로 구성돼 있다.

그럼 지난 시간에 만든 클래스를 다시 한 번 살펴볼까요?

```
>>> class BusinessCard:
        def set_info(self, name, email, addr):
            self.name = name
            self.email = email
            self.addr = addr
        def print_info(self):
            print("--------------------")
            print("Name: ", self.name)
            print("E-mail: ", self.email)
            print("Address: ", self.addr)
            print("--------------------")

>>>
```

파이썬의 class 키워드를 통해 BusinessCard라는 새로운 타입을 만들었으니 인스턴스를 생성해야겠죠? 다시 한번 말씀드리면 클래스 인스턴스를 생성하려면 '클래스 이름()'과 같이 적으면 됩니다. 그리고 생성된 인스턴스에 점(.)을 찍어서 인스턴스 변수나 메서드에 접근할 수 있었습니다.

```
>>> member1 = BusinessCard()
>>> member1.set_info("kim", "kim@gmail.com", "USA")
>>>
```

위 코드를 살펴보면 먼저 인스턴스를 생성하고 생성된 인스턴스에 데이터를 입력하는 순으로 코드가 구성돼 있습니다. 붕어빵에 비유해 보면 붕어빵 틀(클래스)을 이용해 팥소를 넣지 않은 상태로 붕어빵을 구운 후(인스턴스 생성) 나중에 다시 붕어빵 안으로 팥소를 넣는 것과 비슷합니다. 어떻게 하면 클래스 인스턴스 생성과 초깃값 입력을 한 번에 처리할 수 있을까요?

파이썬 클래스에는 인스턴스 생성과 동시에 자동으로 호출되는 메서드인 생성자가 존재합니다. 참고로 생성자는 C++나 자바 같은 객체지향 프로그래밍 언어에도 있는 개념입니다. 파이썬에서는 __init__(self)와 같은 이름의 메서드를 생성자라고 하며, 파이썬 클래스에서 __로 시작하는 함수는 모두 특별한 메서드를 의미합니다.

다음은 생성자인 __init__(self) 메서드를 가진 MyClass 클래스를 정의한 것입니다. 앞서 설명한 것처럼 생성자의 첫 번째 인자도 항상 self여야 합니다. 생성자 내부에는 print 문을 통해 간단한 메시지를 출력했습니다.

```
>>> class MyClass:
        def __init__(self):
            print("객체가 생성됐습니다.")

>>>
```

MyClass라는 클래스를 정의했으니 이제 이 클래스의 인스턴스를 생성해보겠습니다. 다음 코드를 보면 인스턴스를 생성하자마자 화면에 메시지가 출력되는 것을 확인할 수 있습니다. 이는 인스턴스가 생성되는 시점에 자동으로 생성자인 __init__(self) 메서드가 호출됐기 때문입니다. 참고로 init이라는 영어 단어는 '초기화하다'라는 뜻이 있는 initialize의 약어입니다.

```
>>> inst1 = MyClass()
객체가 생성됐습니다.
>>>
```

클래스 생성자를 이해했다면 BusinessCard 클래스를 수정해 인스턴스의 생성과 동시에 명함에 필요한 정보를 입력받도록 클래스를 새롭게 정의해 봅시다. 눈치가 빠르신 분들은 벌써 정답을 맞혔지요? 기존에 구현했던 BusinessCard 클래스의 set_info 메서드가 데이터를 입력받는 역할을 수행했으므로 set_info 메서드를 __init__ 메서드로 이름만 변경하면 됩니다.

```
>>> class BusinessCard:
        def __init__(self, name, email, addr):
            self.name = name
            self.email = email
            self.addr = addr

        def print_info(self):
            print("--------------------")
            print("Name: ", self.name)
            print("E-mail: ", self.email)
            print("Address: ", self.addr)
            print("--------------------")

>>>
```

새로 정의된 BusinessCard 클래스의 생성자는 인자가 4개임을 확인할 수 있습니다. 물론 첫 번째 인자인 self는 생성되는 인스턴스를 의미하고 자동으로 값이 전달되므로 인스턴스를 생성할 때 명시적으로 인자를 전달해야 하는 것은 3개입니다. 따라서 인스턴스를 생성할 때 3개의 인자를 전달하지 않으면 오류가 발생합니다. 생성자 호출 단계에서 오류가 발생하면 인스턴스도 정상적으로 생성되지 않게 됩니다.

```
>>> member1 = BusinessCard()
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    member1 = BusinessCard()
TypeError: __init__() missing 3 required positional arguments: 'name', 'email', and 'addr'
>>>
```

새로 정의된 BusinessCard 클래스는 생성자에서 3개의 인자(name, email, addr)를 받기 때문에 다음과 같이 인스턴스를 생성할 때 3개의 인자를 전달해야 정상적으로 인스턴스가 생성됩니다. member1이라는 인스턴스가 생성된 후에는 인스턴스 메서드를 호출해 인스턴스 변수의 값을 화면에 출력할 수 있습니다. 어떤가요? 클래스의 생성자를 사용하니 인스턴스의 생성과 초깃값 저장을 한 번에 할 수 있어 편리하지요?

```
>>> member1 = BusinessCard("Kangsan Lee", "kangsan.lee", "USA")
>>> member1.print_info()
--------------------
Name:  Kangsan Lee
E-mail:  kangsan.lee
Address:  USA
--------------------
>>>
```

03 self 이해하기

앞에서 클래스 내에 정의된 함수를 메서드라고 부른다고 했습니다. 그리고 메서드의 첫 번째 인자는 항상 self여야 한다고 했습니다. 하지만 메서드의 첫 번째 인자가 항상 self여야 한다는 것은 사실 틀린 말입니다. 이번 절에서는 파이썬 클래스에서 self의 정체를 확실히 이해해 봅시다.

먼저 다음과 같이 두 개의 메서드가 정의된 Foo 클래스를 만들어 봅시다. 여기서 눈여겨봐야 할 점은 func1() 메서드의 첫 번째 인자가 self가 아님에도 클래스를 정의할 때 에러가 발생하지 않는다는 점입니다.

```
>>> class Foo:
        def func1():
            print("function 1")

        def func2(self):
            print("function 2")

>>>
```

일단 클래스를 정의했으니 해당 클래스에 대한 인스턴스를 생성해보겠습니다. 그리고 생성된 인스턴스를 통해 인스턴스 메서드를 호출해보겠습니다. Foo 클래스의 func2 메서드는 메서드의 인자가 self뿐이므로 실제 메서드를 호출할 때는 인자를 전달할 필요가 없습니다.

```
>>> f = Foo()
>>> f.func2()
function 2
>>>
```

위 코드에서 메서드를 호출한 결과를 보면 화면에 정상적으로 'function 2'가 출력된 것을 볼 수 있습니다. 참고로 func2 메서드의 첫 번째 인자는 self지만 호출할 때는 아무것도 전달하지 않는 이유는 첫 번째 인자인 self에 대한 값은 파이썬이 자동으로 넘겨주기 때문입니다.

그렇다면 func1 메서드처럼 메서드를 정의할 때부터 아무 인자도 없는 경우에는 어떻게 될까요? 다음과 같이 인스턴스를 통해 func1()을 호출해보면 오류가 발생합니다. 오류 메시지를 살펴보면 "func1()은 인자가 없지만 하나를 받았다"라는 것을 볼 수 있습니다. 이는 앞서 설명한 것처럼 파이썬 메서드의 첫 번째 인자로 항상 인스턴스가 전달되기 때문에 발생하는 문제입니다.

```
>>> f.func1()
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    f.func1()
TypeError: func1() takes 0 positional arguments but 1 was given
>>>
```

이번에는 self의 정체를 좀 더 확실히 밝혀보기 위해 파이썬 내장 함수인 id를 이용해 인스턴스가 메모리에 할당된 주솟값을 확인해보겠습니다. 다음 코드처럼 Foo 클래스를 새로 정의합니다. func2 메서드가 호출될 때 메서드의 인자로 전달되는 self의 id 값을 화면에 출력하는 기능이 추가됐습니다.

```
>>> class Foo:
      def func1():
          print("function 1")

      def func2(self):
          print(id(self))
          print("function 2")

>>>
```

Foo 클래스를 새롭게 정의했으므로 인스턴스를 다시 만든 후 id() 내장함수를 이용해 인스턴스가 할당된 메모리 주소를 확인해 봅시다.

```
>>> f = Foo()
>>> id(f)
43219856
>>>
```

생성된 인스턴스가 메모리의 43219856번지에 있음을 확인할 수 있습니다. 참고로 이 값은 해당 코드의 실행 환경에 영향을 받게 되므로 여러분이 직접 실행했을 때 이 값과 다른 값이 나올 수 있습니다.

위 코드에서 f와 생성된 인스턴스의 관계를 그림으로 나타내면 그림 6.8과 같습니다. Foo 클래스에 대한 인스턴스는 메모리의 43219856번지부터 할당돼 있고 변수 f는 인스턴스의 주솟값을 담고 있습니다. 일단 인스턴스가 할당된 메모리 주솟값을 기억해두기 바랍니다. 곧 놀라운 광경을 목격할 수 있을 것입니다.

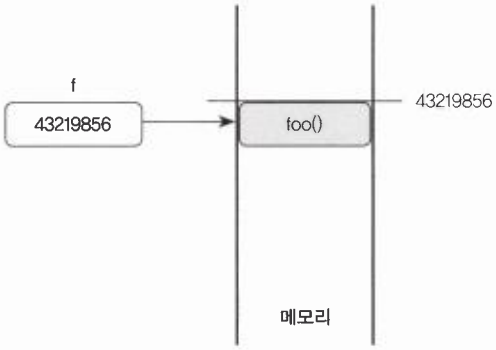


그림 6.8 인스턴스의 메모리 주소 확인

이번에는 인스턴스 f를 이용해 func2 메서드를 호출해보기 바랍니다. 다음 코드를 살펴보면 func2 메서드를 호출할 때 아무런 값도 전달하지 않았습니다.

```
>>> f.func2()
43219856
function 2
>>>
```

실행 결과를 살펴보면 43219856이라는 값이 출력되는 것을 확인할 수 있습니다. Foo 클래스를 정의할 때 id(self)를 출력하게 했는데 id(self)의 값이 바로 43219856인 것입니다. 이 값은 그림 6.8에서 볼 수 있듯이 f라는 변수가 바인딩하고 있는 인스턴스의 주솟값과 동일합니다. 즉, 클래스 내에 정의된 self는 클래스 인스턴스임을 알 수 있습니다.

아직 이 부분이 잘 이해되지 않는다면 객체를 하나 더 만들어보겠습니다. 새로 생성한 객체는 f2가 가리키고 있는데, id를 통해 확인해보니 주소가 47789232입니다. f2를 통해 func2 메서드를 호출해보면 47789232가 출력됐습니다. 이 값은 바로 f2가 가리키고 있는 객체를 의미합니다.

```
>>> f2 = Foo()
>>> id(f2)
47789232
>>> f2.func2()
47789232
```

```
function 2
>>>
```

파이썬의 클래스는 그 자체가 하나의 네임스페이스이기 때문에 인스턴스 생성과 상관없이 클래스 내의 메서드를 직접 호출할 수 있습니다(네임스페이스는 다음 절에서 설명합니다).

```
>>> Foo.func1()
function 1
>>>
```

위 코드는 func1 메서드를 호출했지만 앞서 인스턴스를 통해 메서드를 호출했던 것과는 달리 오류가 발생하지 않는 것을 확인할 수 있습니다. 왜냐하면 인스턴스.메서드() 형태로 호출한 것과 달리 이번에는 클래스명.메서드() 형태로 호출했기 때문입니다. 그렇다면 func2()에 대해서도 클래스 이름을 통해 호출해 봅시다. 그림 6.9와 같이 클래스 이름을 통해 func2() 메서드를 호출하려고 하면 self 위치에 인자를 전달해야 한다고 파이썬 인터프리터가 알려줍니다.



그림 6.9 클래스 이름을 이용한 메서드 호출

self 위치에 인자를 전달하지 않고 메서드를 호출하면 그림 6.10과 같이 오류가 발생합니다. 오류 메시지를 확인하면 func2()를 호출할 때 인자를 하나 빠트렸음을 알 수 있습니다. 즉, 인자를 하나 전달해야 하는데 전달하지 않아서 오류가 발생한 것입니다.

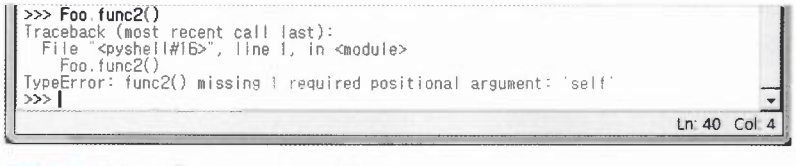


그림 6.10 클래스 이름을 이용한 메서드 호출 에러

그럼 도대체 어떤 값을 전달하면 되는 걸까요? 앞에서 메서드의 self로 전달되는 것은 인스턴스 자체라고 설명했습니다. 따라서 클래스에 대한 인스턴스를 생성한 후 해당 인스턴스를 전달하면 됩니다. 다음과 같이 새로운 객체를 만들고 이를 f3이 가리키게 합니다. id(f3) 구문 호출을 통해 f3이 가리키는 객체의 주소가 47789136임을 확인할 수 있습니다.

```
>>> f3 = Foo()
>>> id(f3)
47789136
>>>
```

다시 한 번 클래스 이름인 Foo를 이용해 func2 메서드를 호출해보겠습니다. 앞서 func2 메서드는 인자를 하나 필요로 하며, 해당 인자는 인스턴스여야 한다고 말씀드렸습니다. 현재 f3은 새로 생성한 인스턴스를 바인딩하고 있으므로 func2 메서드의 인자로 f3을 전달하면 됩니다.

```
>>> Foo.func2(f3)
47789136
func2
>>>
```

그렇다면 인스턴스를 통해 func2를 호출하는 것과 클래스 이름을 통해 func2를 호출하는 것은 어떤 차이가 있을까요? 결론부터 말씀드리면 둘 사이에는 아무런 차이가 없습니다. 다만 '인스턴스.메서드()'냐 '클래스.메서드(인스턴스)'냐라는 차이가 있을 뿐입니다. 보통은 '인스턴스.메서드()'와 같은 방식을 주로 사용합니다.

```
>>> f3.func2()
47789136
func2
>>>
```

04 클래스 네임스페이스

클래스와 인스턴스의 차이를 정확히 이해하는 것은 매우 중요합니다. 이를 위해서는 먼저 네임스페이스라는 개념을 알아야 합니다. 네임스페이스라는 것은 변수가 객체를 바인딩할 때 그 둘 사이의 관계를 저장하고 있는 공간을 의미합니다. 예를 들어, 'a = 2'라고 했을 때 a라는 변수가 2라는 객체가 저장된 주소를 가지고 있는데 그러한 연결 관계가 저장된 공간이 바로 네임스페이스입니다.

파이썬의 클래스는 새로운 타입(객체)을 정의하기 위해 사용되며, 모듈과 마찬가지로 하나의 네임스페이스를 가집니다. 먼저 Stock 클래스를 정의해 봅시다.

```
>>> class Stock:
        market = "kospi"

>>>
```

파이썬 IDLE에서 dir 내장 함수를 호출해보면 리스트로 된 반환값을 확인할 수 있습니다. 여기서 두 개의 언더바로 시작하는 것은 파이썬에서 이미 사용 중인 특별한 것들입니다. 이를 제외하고 보면 조금 전에 정의했던 Stock 클래스의 이름이 포함된 것을 확인할 수 있습니다.

```
>>> dir()
['Stock', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>>
```

dir 내장함수의 결괏값에 Stock 클래스가 들어있기 때문에 앞으로는 프롬프트에 Stock을 입력해도 오류가 발생하지 않습니다. 그러나 Stock1이라는 이름은 존재하지 않기 때문에 입력하면 오류가 발생합니다.

```
>>> Stock
<class '__main__.Stock'>
>>> Stock1
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    Stock1
NameError: name 'Stock1' is not defined
>>>
```

파이썬에서는 클래스가 정의되면 그림 6.11과 같이 하나의 독립적인 네임스페이스가 생성됩니다. 그리고 클래스 내에 정의된 변수나 메서드는 해당 네임스페이스 안에 파이썬 딕셔너리 타입으로 저장됩니다. Stock 클래스는 그림 6.11과 같이 Stock이라는 네임스페이스 안에 'market':'kospi'라는 값을 가진 딕셔너리를 포함합니다.

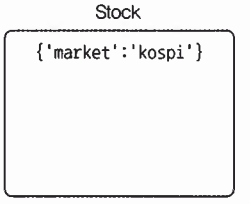


그림 6.11 파이썬 클래스 네임스페이스

Stock 클래스의 네임스페이스를 파이썬 코드로 확인하려면 클래스의 __dict__ 속성을 확인하면 됩니다. 딕셔너리 타입에 'market':'kospi'라는 키와 값 쌍이 존재하는 것을 확인할 수 있습니다.

```
>>> Stock.__dict__
mappingproxy({'market': 'kospi', '__module__': '__main__', '__dict__': <attribute '__dict__' of
'Stock' objects>, '__doc__': None, '__weakref__': <attribute '__weakref__' of 'Stock' objects>})
>>>
```

클래스가 독립적인 네임스페이스를 가지고 클래스 내의 변수나 메서드를 네임스페이스에 저장하고 있으므로 다음과 같이 클래스 내의 변수에 접근할 수 있는 것입니다.

```
>>> Stock.market
'kospi'
>>>
```

이번에는 인스턴스를 생성해보겠습니다. 다음과 같이 서로 다른 두 개의 인스턴스를 생성해보기 바랍니다. 생성된 인스턴스에 대한 id 값을 확인해보면 두 인스턴스가 서로 다른 메모리에 위치하는 것을 확인할 수 있습니다.

```
>>> s1 = Stock()
>>> s2 = Stock()
>>> id(s1)
50572464
>>> id(s2)
50348240
>>>
```

파이썬은 인스턴스를 생성하면 인스턴스별로 별도의 네임스페이스를 유지합니다. 즉, 위의 코드를 그림으로 표현하면 그림 6.12와 같습니다.

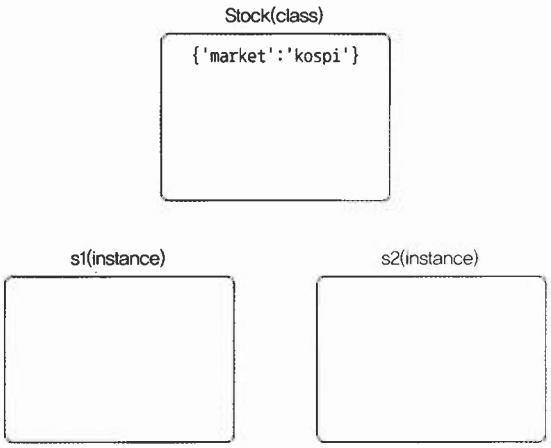


그림 6.12 클래스와 인스턴스의 네임스페이스

먼저 생성된 s1, s2 인스턴스가 네임스페이스에 있는지 코드를 통해 확인해 봅시다. dir 내장함수의 반환값을 확인하면 s1, s2가 Stock과 마찬가지로 존재하는 것을 확인할 수 있습니다.

```
>>> dir()
['Stock', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 's1', 's2']
>>>
```

생성된 s1, s2 인스턴스 각각에 대한 네임스페이스도 확인해봅시다. 클래스 또는 인스턴스에 대한 네임스페이스를 확인하려면 __dict__ 속성을 확인하면 됩니다.

```
>>> s1.__dict__
```

```
{}
>>> s2.__dict__
{}
>>>
```

위 코드를 보면 s1과 s2 인스턴스의 네임스페이스는 현재 비어 있음을 확인할 수 있습니다. s1 인스턴스에 market이라는 변수를 추가해 봅시다. 그런 다음 다시 __dict__ 속성을 확인해보면 'market':'kosdaq'이라는 키:값 쌍이 추가된 것을 볼 수 있습니다.

```
>>> s1.market = 'kosdaq'
>>> s1.__dict__
{'market': 'kosdaq'}
>>>
```

그러나 여전히 s2 인스턴스의 네임스페이스는 비어 있는 상태입니다. 현재 Stock 클래스와 s1, s2 인스턴스의 네임스페이스를 그림으로 나타내면 그림 6.13과 같습니다.

```
>>> s2.__dict__
{}
>>>
```

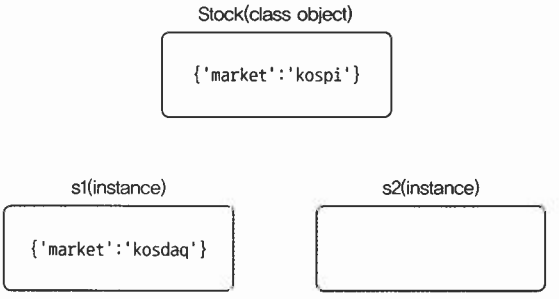


그림 6.13 클래스와 인스턴스의 네임스페이스 상태

현재 Stock 클래스로부터 s1, s2라는 두 개의 인스턴스를 생성했습니다. s1 인스턴스는 market이라는 변수를 가지고 있지만 s2의 네임스페이스에는 변수나 메서드가 존재하지 않습니다. 만약 s1.market, s2.market과 같이 인스턴스를 통해 market이라는 값에 접근하면 어떻게 될까요?

```
>>> s1.market
'kosdaq'
>>> s2.market
'kospi'
>>>
```

위 코드를 참조하면 s1.market의 값은 'kosdaq'입니다. 이것은 그림 6.13과 같이 s1 인스턴스의 네임스페이스에 'market':'kosdaq'이라는 키:값 쌍이 존재하기 때문에 가능합니다. 그런데 s2 인스턴스의 반환값을 살펴보면 조금 이상합니다. 왜냐하면 현재 s2의 네임스페이스(딕셔너리)에는 아무런 값도 존재하지 않기 때문입니다. s2의 네임스페이스에는 변수나 메서드가 존재하지 않지만 s2.market의 값으로 'kospi'라는 문자열이 반환되는 이유는 그림 6.14와 같이 동작하기 때문입니다.

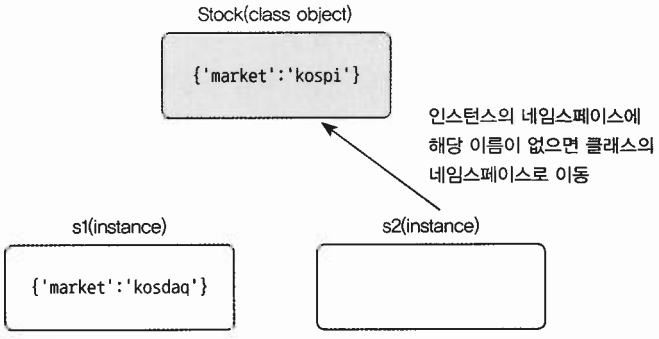


그림 6.14 클래스 및 인스턴스의 네임스페이스 참조 방식

s2 인스턴스를 통해 변수에 접근하면 파이썬은 먼저 s2 인스턴스의 네임스페이스에서 해당 변수가 존재하는지 찾습니다. s2의 네임스페이스에 해당 변수가 존재하지 않으면 s2 인스턴스의 클래스의 네임스페이스로 가서 다시 변수를 찾게 됩니다. 즉, s2.market이라는 문장이 실행되면 Stock 클래스의 네임스페이스에 있는 'market':'kospi' 키:값 쌍에서 'kospi'라는 문자열을 출력하게 됩니다.

이번에는 인스턴스의 네임스페이스에도 없고 클래스의 네임스페이스에도 없는 변수에 접근해 봅시다. 이 경우 volume이라는 값이 s2 인스턴스의 네임스페이스에 없으므로 Stock 클래스에서 찾게 되는데, Stock 클래스의 네임스페이스에도 volume이라는 값이 없으므로 오류가 발생합니다.

```
>>> s2.volume
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    s2.volume
AttributeError: 'Stock' object has no attribute 'volume'
>>>
```

05 클래스 변수와 인스턴스 변수

6.4절에서는 클래스의 네임스페이스와 인스턴스의 네임스페이스, 그리고 그 둘 사이의 관계에 대해 배웠습니다. 이번 절에서는 초보자들이 많이 어려워하는 개념 중 하나인 클래스 변수(class variable)와 인스턴스 변수(instance variable)에 대해 살펴보겠습니다.

다음은 은행 계좌를 클래스로 표현한 것입니다. Account 클래스에는 생성자와 소멸자가 있습니다. 생성자(__init__)가 클래스의 인스턴스가 생성될 때 자동으로 호출되는 함수라면 소멸자(__del__)는 클래스의 인스턴스가 소멸될 때 자동으로 호출되는 함수입니다.

```
>>> class Account:
      num_accounts = 0

      def __init__(self, name):
          self.name = name
          Account.num_accounts += 1

      def __del__(self):
          Account.num_accounts -= 1

>>>
```

Account 클래스에는 num_accounts와 self.name이라는 두 종류의 변수가 있습니다. num_accounts처럼 클래스 내부에 선언된 변수를 클래스 변수라고 하며, self.name과 같이 self가 붙어 있는 변수를 인스턴스 변수라고 합니다. 클래스 변수는 Account 클래스의 네임스페이스에 위치하며, self.name과 같은 인스턴스 변수는 인스턴스의 네임스페이스에 위치하게 됩니다.

그렇다면 언제 클래스 변수를 사용해야 하고 언제 인스턴스 변수를 사용해야 할까요? 이에 대한 답은 간단한 코드를 작성해보면서 천천히 설명해 드리겠습니다. 여러분이 은행에 가서 계좌를 개설하면 새로운 계좌가 하나 개설됩니다. 이러한 상황을 파이썬으로 표현하면 다음과 같이 Account 클래스의 인스턴스를 생성하는 것에 해당합니다.

```
>>> kim = Account("kim")
>>> lee = Account("lee")
>>>
```

생성된 kim과 lee 인스턴스에 계좌 소유자 정보가 제대로 저장돼 있는지 확인해 봅시다. 각 계좌에 대한 소유자 정보는 인스턴스 변수인 name이 바인딩하고 있습니다.

```
>>> kim.name
'kim'
>>> lee.name
'lee'
>>>
```

그렇다면 지금까지 은행에서 개설된 계좌는 총 몇 개일까요? 네, 정답은 'kim'과 'lee'에게 하나씩 개설됐기 때문에 두 개겠죠? kim 인스턴스나 lee 인스턴스를 통해 num_accounts라는 이름에 접근하면 총 계좌 개설 개수가 2개로 나오는 것을 알 수 있습니다.

```
>>> kim.num_accounts
2
>>> lee.num_accounts
2
>>>
```

물론 지금까지 공부를 잘 해오신 분들은 kim.num_accounts에서 먼저 인스턴스의 네임스페이스에서 num_accounts를 찾았지만 해당 이름이 없어서 클래스의 네임스페이스로 이동한 후 다시 해당 이름을 찾았고 그 값이 반환된 것임을 아실 것입니다.

이처럼 여러 인스턴스 간에 서로 공유해야 하는 값은 클래스 변수를 통해 바인딩해야 합니다. 왜냐하면 파이썬은 인스턴스의 네임스페이스에 없는 이름은 클래스의 네임스페이스에서 찾아보기 때문에 이러한 특성을 이용하면 클래스 변수가 모든 인스턴스에 공유될 수 있기 때문입니다. 참고로 클래스 변수에 접근할 때 아래와 같이 클래스 이름을 사용할 수도 있습니다.

```
>>> Account.num_accounts
2
>>>
```

지금까지 작성한 코드에서 클래스 변수와 인스턴스 변수를 그림으로 나타내면 그림 6.15와 같습니다. 앞으로 클래스 변수와 인스턴스 변수가 헷갈릴 때마다 이 그림을 기억하기 바랍니다.

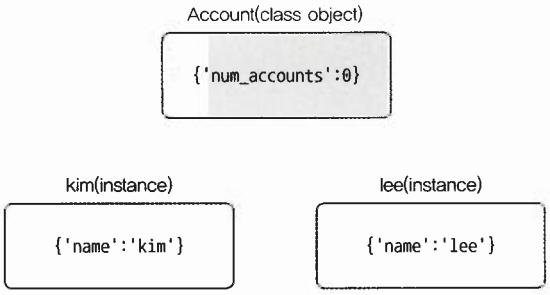


그림 6.15 클래스 변수와 인스턴스 변수

06 클래스 상속

상속이란 사람이 사망함에 따라 사망자의 재산 및 신분상의 지위에 대한 포괄적인 승계를 의미합니다. 드라마에서 보면 부모님으로부터 많은 재산을 상속받은 사람들을 종종 볼 수 있지요? 상속하는 사람 입장은 잘 모르겠지만 상속자들은 분명 상속받지 않은 경우보다 대부분 좋은 경우가 많을 것입니다. 특히 많은 재산을 상속받는 경우라면 더욱 좋겠지요?

프로그래밍 책에서 갑자기 상속 이야기를 한 이유는 객체지향 프로그래밍을 지원하는 프로그래밍 언어는 클래스에서 상속 기능을 지원하기 때문입니다. 자식이 부모님으로부터 재산 등을 상속받는 것처럼 다른 클래스에 이미 구현된 메서드나 속성을 상속한 클래스에서는 그러한 메서드나 속성을 그대로 사용할 수 있게 됩니다.

클래스의 상속을 또 다른 관점에서 생각해보면 클래스를 상속한다는 것은 부모 클래스의 능력을 그대로 전달받는 것을 의미합니다. 인간으로 치면 부모로부터 유전형질을 물려받아 부모의 능력을 그대로 물려받는 것과 비슷합니다.

일단 노래를 잘 부르는 부모 클래스가 있다고 생각해 봅시다. 이를 파이썬으로 표현하면 다음과 같이 노래를 부르는 메서드가 포함된 클래스를 정의할 수 있습니다.

```
>>> class Parent:
        def can_sing(self):
            print("Sing a song")

>>>
```

Parent 클래스를 정의했으니 클래스의 인스턴스를 생성해 보겠습니다. 그리고 노래를 정말 할 수 있는지 메서드를 호출해 확인해 보겠습니다.

```
>>> father = Parent()
>>> father.can_sing()
Sing a song
>>>
```

이번에는 노래를 잘 부르는 Parent 클래스로부터 상속받은 운이 좋은 자식 클래스를 정의해 봅시다. 클래스의 이름은 LuckyChild라고 하겠습니다. 클래스를 정의할 때 다른 클래스로부터 상속받고자 한다면 새로 정의할 클래스 이름 다음에 괄호를 사용해 상속받고자 하는 클래스의 이름을 지정하면 됩니다. LuckyChild 클래스는 상속받기 전까지는 아무런 능력이 없어서 내부에 메서드를 구현하지 않고 pass만 적어 주었습니다.

```
>>> class LuckyChild(Parent):
        pass

>>>
```

Parent 클래스로부터 상속받은 LuckyChild 클래스에 대한 인스턴스를 생성한 후 노래를 시켜보겠습니다. 중요한 점은 현재 LuckyChild 클래스에는 어떤 메서드도 존재하지 않는다는 것입니다. 다음 코드를 보면 역시나 부모 클래스로부터 상속받아서 그런지 자신은 메서드를 포함하고 있지 않지만 바로 노래를 부를 수 있군요.

```
>>> child1 = LuckyChild()
>>> child1.can_sing()
Sing a song
>>>
```

이번에는 부모로부터 어떤 능력이나 재산도 상속받지 못한 운이 좋지 않은 자식 클래스(UnLuckyChild)를 만들어 보겠습니다. 인간 세상으로 치면 평범한 서민 클래스라고 볼 수 있겠습니다.

```
>>> class UnLuckyChild:
        pass

>>>
```

운이 좋지 않은 UnLuckyChild 클래스에 대한 인스턴스를 생성한 후 노래를 시켜봅시다. 역시나 상속도 받지 못했고 자신도 아무런 메서드가 없으므로 노래를 부를 수 없습니다. 이처럼 능력이나 재산을 상속받지 못했다면 자신이 직접 노래를 잘할 수 있도록 연습해야겠지요? 프로그래밍 측면에서 보면 직접 메서드를 구현해야 한다는 뜻입니다.

```
>>> child2 = UnLuckyChild()
>>> child2.can_sing()
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    child2.can_sing()
AttributeError: 'UnLuckyChild' object has no attribute 'can_sing'
>>>
```

인간 세계에서는 상속을 받으면 좋겠구나, 라고 생각되겠지만 프로그래밍할 때 왜 상속을 해야 하는지 여전히 이해되지 않는 분도 계실 겁니다. 객체지향 프로그래밍에서는 어떤 클래스를 상속하면 부모 클래스의 모든 것을 내 것처럼 사용할 수 있고, 자신의 클래스에 메서드를 더 구현한다면 '플러스알파'가 되는 것입니다. 즉, 부모 클래스의 능력을 밑바탕으로 깔고 거기서부터 한 번 더 업그레이드되는 것입니다.

그럼 이번에는 노래도 잘 부르고 춤도 잘 추는 운이 좋은 자식 클래스2(LuckyChild2)를 정의해 보겠습니다.

```
>>> class LuckyChild2(Parent):
      def can_dance(self):
          print("Shuffle Dance")

>>>
```

LuckyChild2 클래스에 대한 인스턴스를 생성한 후 노래와 춤을 시켜봅시다. LuckyChild2 클래스는 부모로부터 노래 부르는 능력을 상속받았고 자기 자신이 춤추는 능력을 갖추고 있어 노래도 부르고 춤도 출 수 있게 됐습니다.

```
>>> child2 = LuckyChild2()
>>> child2.can_sing()
Sing a song
>>> child2.can_dance()
Shuffle Dance
>>>
```

물론 굳이 상속이라는 기능을 사용하지 않고도 부모 클래스에 구현된 메서드를 그대로 복사해서 새로 정의할 클래스에 코드를 붙여넣는 식으로 사용할 수도 있습니다. 단, 이렇게 하면 같은 기능을 하는 코드가 중복되기 때문에 코드를 관리하기가 어렵고 복사 및 붙여넣기를 해야 하므로 불편합니다. 이에 반해 클래스의 상속이라는 기능을 이용하면 최소한의 코드로도 부모 클래스에 구현된 메서드를 손쉽게 바로 이용할 수 있습니다.

연습 문제

문제 6-1

다음 조건을 만족하는 Point라는 클래스를 작성하세요.

- Point 클래스는 생성자를 통해 (x, y) 좌표를 입력받는다.
- setx(x), sety(y) 메서드를 통해 x 좌표와 y 좌표를 따로 입력받을 수도 있다.
- get() 메서드를 호출하면 튜플로 구성된 (x, y) 좌표를 반환한다.
- move(dx, dy) 메서드는 현재 좌표를 dx, dy만큼 이동시킨다.
- 모든 메서드는 인스턴스 메서드다.

문제 6-2

문제 6-1에서 생성한 Point 클래스에 대한 인스턴스를 생성한 후 4개의 메서드를 사용하는 코드를 작성하세요.

문제 6-3

아래의 Stock 클래스에 대해 2개의 인스턴스를 생성했을 때 클래스와 a와 b 인스턴스의 네임스페이스를 그려보세요.

```
>>> class Stock:
        market = "kospi"

>>> a = Stock()
>>> b = Stock()
```

문제 6-4

문제 6-3의 코드에서 추가로 아래와 같은 코드를 수행했을 때 '???'로 표시된 부분의 결괏값을 적어보세요.

```
>>> a.market
???
>>> b.market
???
>>> Stock.market
???
>>> a.market = "kosdaq"
>>> b.market = "nasdaq"
>>> a.market
???
>>> b.market
???
>>> Stock.market
???
>>>
```

문제 6-5

문제 6-3, 문제 6-4의 코드가 모두 수행된 후의 Stock 클래스, a 인스턴스와 b 인스턴스의 네임스페이스를 그려보세요.

07

파일 다루기

이번 장에서는 파이썬을 이용해 파일로부터 데이터를 읽거나 반대로 프로그램의 결괏값을 파일로 저장하는 방법을 알아보겠습니다. 참고로 알고리즘 트레이딩 분야에서는 통계 데이터를 틈틈이 파일로 저장해 여러분만의 데이터베이스를 구축하는 것이 매우 중요합니다. 이번 장을 통해 파이썬을 통한 파일 입/출력에 익숙해지기 바랍니다.

01 파일 읽기

컴퓨터에서 파일이란 무엇인가요? 아직 프로그래밍에 대해서는 잘 모르겠지만 한글 파일(.hwp) 또는 마이크로소프트의 워드 파일(.doc)이란 말은 종종 들어 봤을 겁니다. 자주 사용하는 문서 파일 외에도 동영상 파일(.avi)이나 음악 파일(.mp3)도 파일이란 용어를 사용하고 있습니다.

위키피디아[1]를 참고하면 파일은 컴퓨터 등의 기기에서 의미 있는 정보를 담는 논리적인 단위라고 합니다. 앞서 설명한 것처럼 컴퓨터에서 파일은 종류가 매우 다양합니다. 그중 가장 쉽게 만들 수 있는 파일 형태는 아마 텍스트 파일일 것입니다. 참고로 텍스트 파일은 윈도우의 메모장 등을 이용해 간단히 텍스트를 입력함으로써 쉽게 만들 수 있습니다.

파이썬을 이용한 파일 읽기에 앞서 윈도우 메모장을 이용해 주식 매수 리스트를 텍스트 파일로 만들어 보겠습니다. 윈도우 10에서 메모장 앱을 실행하려면 그림 7.1과 같이 검색 기능을 활용하면 됩니다.



그림 7.1 윈도우 10을 이용한 메모장 앱 실행

메모장 프로그램을 실행했다면 그림 7.2와 같이 NAVER, 현대차, 기아차를 입력해보기 바랍니다. 이때 중요한 점은 구분자로 종목을 서로 구분해야 한다는 점입니다. 참고로 그림 7.2와 같이 종목명을 입력한 후 엔터 키를 눌러 각 종목을 구분했습니다.

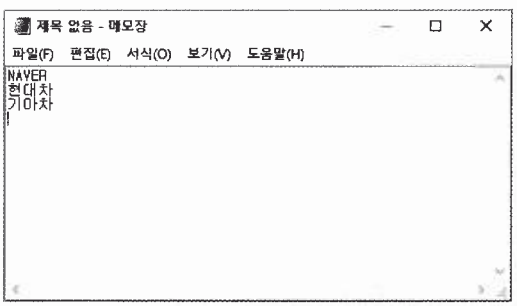


그림 7.2 매수 리스트

그림 7.2와 같이 메모장에 매수 리스트를 작성한 후 메모장의 메뉴에서 [파일] → [다른 이름으로 저장]을 차례로 선택합니다. 그림 7.3과 같은 [다른 이름으로 저장] 대화상자에서 먼저 저장 경로로 바탕 화면을 선택한 후 파일명을 buy_list로 입력하고 [저장] 버튼을 클릭합니다.

1 https://ko.wikipedia.org/wiki/컴퓨터_파일

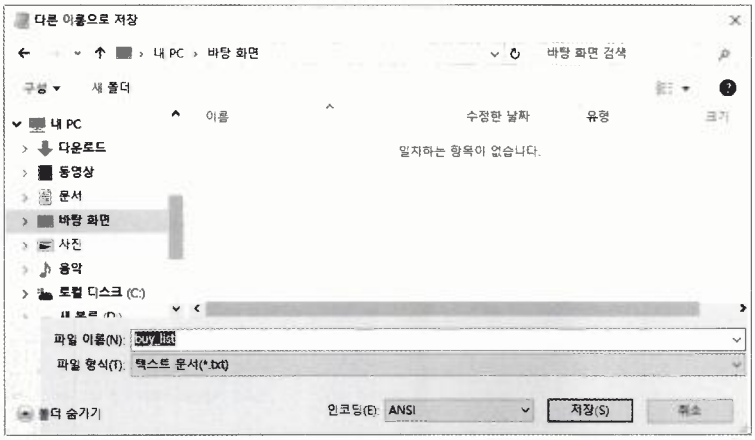


그림 7.3 텍스트 파일을 다른 이름으로 저장하기

윈도우 바탕 화면을 보면 그림 7.4와 같이 buy_list라는 이름으로 텍스트 파일이 하나 생성된 것을 확인할 수 있습니다.



그림 7.4 바탕 화면에 생성된 텍스트 파일

택배 기사가 여러분의 집으로 택배를 배달하는 경우를 상상해 봅시다. 택배를 제대로 배달하려면 여러분의 집 주소를 정확히 알아야 합니다. 마찬가지로 파이썬으로 컴퓨터 내의 파일을 읽기 위해서는 파일의 경로를 정확히 알아야 합니다.

앞서 저장한 buy_list.txt 파일의 경로를 확인하기 위해 해당 파일에 마우스 오른쪽 버튼을 클릭한 후 그림 7.5와 같이 [속성] 메뉴를 선택합니다.



그림 7.5 파일 속성 메뉴

그림 7.6과 같이 텍스트 파일에 대한 속성 창에서 위치 항목에 현재 파일 위치가 나타납니다. 그림 7.6을 참조하면 현재 파일은 C:\Users\Jason\Desktop이라는 디렉터리에 있으며, 파일의 이름은 buy_list이고 확장자는 .txt입니다.

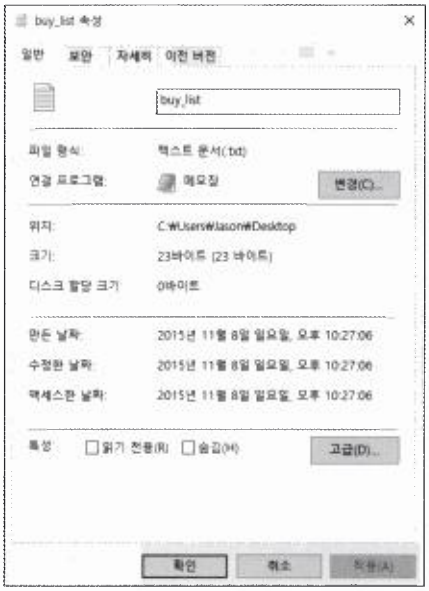


그림 7.6 파일의 위치, 이름 및 확장자 확인

컴퓨터에 저장된 워드 파일을 읽을 때를 생각해봅시다. 보통 워드 프로그램을 실행한 후 열기 메뉴를 이용해 워드 파일을 엽니다. 파일이 열린 후에는 해당 문서를 읽을 수 있습니다. 문서를 다 읽은 경우에는 역시 닫기 메뉴를 이용해 해당 문서를 닫습니다. 파이썬에서 파일을 읽고 쓰는 것도 이와 같습니다. 파일을 읽거나 쓰려면 파일을 먼저 열어야 합니다.

앞서 파일의 위치를 확인했으니 이번에는 파이썬 IDLE를 실행해 워드에서 문서를 여는 것처럼 파이썬으로 텍스트 파일을 열어보겠습니다. 파이썬에서 파일을 열려면 open이라는 내장 함수를 사용하면 됩니다.

다음은 바탕 화면에 저장한 buy_list.txt 파일을 파이썬으로 여는 코드입니다. open 함수는 여러 개의 인자를 받을 수 있는데, 첫 번째 인자로는 파일의 경로를 입력받습니다. 파일 경로를 입력할 때 주의할 점은 디렉터리의 구분자로 '\' 하나를 사용하는 것이 아니라 '\\' 또는 '/'를 사용해야 한다는 것입니다.

```
>>> f = open('c:\\Users\\Jason\\Desktop\\buy_list.txt', 'rt')
```

open 함수의 두 번째 인자에는 파일을 여는 모드를 전달합니다. 예를 들어, 파일을 읽기 모드로 열려면 'r'을 지정하면 되고 텍스트 파일인 경우에는 't'를 지정하면 됩니다. open 내장 함수는 파일에 대한 정보를 담고 있는 객체를 반환하는데 이를 바인딩하기 위해 f라는 이름의 변수를 사용했습니다.

파이썬에서 open 함수의 반환값을 변수로 바인딩한 후에는 해당 변수를 통해 파일을 조작할 수 있습니다. 예를 들어, 열린 파일에서 데이터를 읽기 위해서는 readlines 메서드를 사용하면 됩니다.

```
>>> lines = f.readlines()
>>> lines
['NAVER\n', '현대차\n', '기아차\n']
>>>
```

readlines 메서드는 파일을 읽어서 각 라인을 리스트에 넣은 후 리스트를 리턴합니다. 위 코드를 참조하면 lines라는 변수가 바인딩하는 리스트에는 그림 7.2에서 입력한 문자열이 저장돼 있음을 확인할 수 있습니다.

앞서 배운 for 문을 활용해 파일에서 읽어온 매수 종목을 화면에 출력해봅시다. 다음과 같이 for 문을 사용하면 lines라는 변수가 가리키고 있는 리스트 값을 읽어서 하나씩 화면에 출력할 수 있습니다. 참고로 앞에서 배운 것처럼 for 문에서 사용한 line이라는 변수는 lines 리스트에 있는 원소를 하나씩 바인딩하는 역할을 하는 것으로 이름이 꼭 line일 필요는 없습니다.

```
>>> for line in lines:
        print(line)

NAVER

현대차

기아차

>>>
```

실행 결과를 살펴보면 종목명이 출력된 후 빈 줄이 하나씩 존재하는 것을 볼 수 있습니다. print라는 출력 함수가 문자열을 출력할 때 자동으로 줄바꿈을 하는데, 기존에 lines라는 리스트에 있는 문자열에도 줄바꿈을 의미하는 값인 '\n'이 들어 있기 때문입니다. 이럴 때는 print 함수에 추가 인자를 넣어서 자동으로 줄바꿈되지 않게 하면 됩니다.

```
>>> for line in lines:
        print(line, end="")

NAVER
현대차
기아차
>>>
```

또는 print 함수는 그대로 두고 lines 리스트에 있는 원소에서 '\n' 기호를 제거해서 중복으로 줄바꿈되지 않도록 코드를 작성할 수도 있습니다.

```
>>> for line in lines:
        nline = line.split('\n')[0]
        print(nline)

NAVER
현대차
기아차
>>>
```

어떤가요? 파이썬으로 파일을 읽는 것이 생각보다 어렵지 않지요?

02 파일 쓰기

7.1절에서는 하드디스크에 이미 존재하는 텍스트 파일을 파이썬에서 열고 텍스트 파일 내의 데이터를 읽어봤습니다. 이번에는 파이썬을 이용해 하드디스크에 존재하지 않는 새로운 텍스트 파일을 생성하고 해당 파일에 데이터를 쓰는 작업을 해보겠습니다.

파이썬에서 파일 쓰기는 파일 읽기와 매우 비슷합니다. 워드에서 새로운 글을 작성하는 경우를 생각해 보면 다음과 같습니다.

1. 빈 문서 파일을 만든다.
2. 문서에 내용을 기록한다.
3. 문서 파일을 닫는다.

파이썬에서 파일을 쓰는 것도 기본적으로 위의 3단계를 따릅니다. 단계별로 코드를 작성하면서 파이썬으로 파일 쓰기를 익혀보겠습니다.

먼저 새로운 파일을 엽니다. 이때 중요한 점은 텍스트 파일이 생성될 위치(경로)와 파일명을 지정해야 한다는 것입니다. 앞서 확인했던 윈도우의 바탕 화면 경로에 sell_list.txt라는 빈 문서를 만들어 보겠습니다.

```
>>> f = open('c:\\Users\\Jason\\Desktop\\sel_list.txt', 'wt')
```

위 코드를 보면 읽기 모드로 파일을 열었던 것과 유사합니다. 차이점은 파일명이 sel_list.txt로 바뀌었고, open 함수의 두 번째 인자로 'wt'를 사용했다는 것입니다.

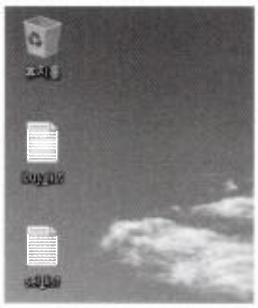


그림 7.7 생성된 텍스트 파일 확인

파이썬에서 파일을 쓰기 모드로 열면 그림 7.7과 같이 하드디스크에 실제 파일이 생성되는 것을 확인할 수 있습니다. 물론 파일 안에는 아직 아무런 내용도 없는 상태입니다.

열려있는 파일에 실제로 데이터를 쓰려면 write 메서드를 사용해야 합니다. 다음과 같이 write 메서드를 이용하면 파일에 데이터를 쓸 수 있습니다.

```
>>> f.write('삼성전자\n')
5
>>> f.write('SK하이닉스\n')
7
>>> f.close()
>>>
```

위 코드를 보면 파일에 '삼성전자'와 'SK하이닉스'라는 문자열을 쓸 때 종목을 구분하기 위해 줄바꿈 기호('\n')를 사용했습니다. 파일에 데이터를 쓴 후에는 close 함수를 사용해 파일을 닫으면 됩니다.

파이썬을 이용한 파일 쓰기가 정상적으로 수행됐는지 확인하기 위해 바탕 화면에 새로 생성된 sel_list.txt 파일을 열어보기 바랍니다. 그림 7.8처럼 sel_list.txt 파일에 '삼성전자'와 'SK하이닉스'라는 종목명이 들어있다면 예제가 정상적으로 실행된 것입니다.

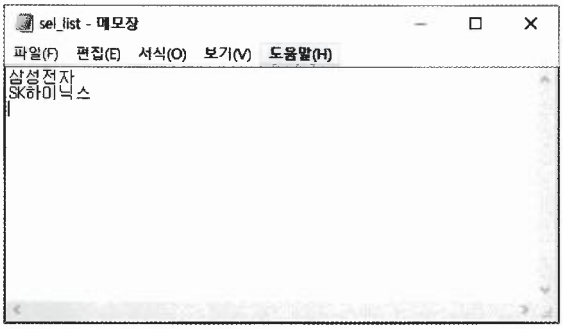


그림 7.8 sel_list 파일의 내용 확인

연습 문제

문제 7-1

1부터 10까지의 숫자를 각 라인 단위로 파일에 출력하는 프로그램을 작성하세요.

- 생성되는 파일의 이름은 `number.txt`다.

문제 7-2

사용자에게 경로를 입력받은 후 해당 경로에 있는 디렉터리와 파일 목록을 `flist.txt`라는 파일로 출력하는 함수를 작성하세요.

08

실전 프로젝트

01 PyCharm을 이용한 개발 환경 구축

이번 장에서는 지금까지 배운 파이썬 기초 문법을 활용해 간단한 프로젝트를 진행해보겠습니다. 이를 통해 지금까지 배운 파이썬 문법이 실제 프로그램 개발에 어떻게 사용되는지 감을 잡을 수 있을 것입니다. 그리고 더 나아가 프로그램 개발에 대한 자신감도 얻을 수 있을 것입니다.

프로그래밍을 처음 배우면 언어의 문법은 어느 정도 알 것 같은데, 막상 프로그램을 작성하려고 하면 어디서부터 어떻게 시작해야 할지 감이 오지 않는 경우가 많습니다. 이러한 단계에서 더 발전하려면 꼭 코드를 직접 입력해 보면서 이번 장에서 소개하는 프로젝트를 완성해 보기 바랍니다.

지금까지 Python IDLE를 이용해 파이썬 코드를 작성했습니다. Python IDLE는 사용하기 편하고 아나콘다 같은 파이썬 배포판이나 파이썬 공식 설치 프로그램만을 설치해도 바로 사용할 수 있어서 초보자에게 적합한 개발 환경입니다. 하지만 조금 규모가 있는 프로그램을 작성할 때는 Python IDLE만으로는 부족한 점이 있습니다.

전문 개발자는 보통 통합 개발 환경이라고 하는 IDE(Integrated Development Environment)를 이용해 프로그램을 개발합니다. 음악을 업으로 하는 사람들이 좋은 악기를 사용해서 연주하는 것을 생각해보면 더 좋은 개발 환경을 사용하는 상황이 쉽게 이해될 겁니다.

IDE는 변수나 함수 이름을 자동으로 완성하는 기능을 제공하고 있어 개발자가 더욱 빠르게 코드를 작성할 수 있게 합니다. 또한 코드의 실행과 결과 확인을 IDE를 통해 한 번에 처리할 수 있어 편리합니다. 그뿐만 아니라 작성한 프로그램이 정상적으로 동작하지 않을 때는 디버거(debugger)를 이용해 코드를 쉽게 분석할 수 있는 기능도 제공합니다. 최근에는 SVN이나 Git과 같은 버전 관리 시스템도 지원하고 있어 프로그램 개발의 전반적인 과정을 IDE를 통해 모두 처리할 수 있습니다. 이제 여러분도 IDE를 사용해 파이썬 프로그램을 개발해봅시다.

파이썬용 IDE에는 비주얼 스튜디오 기반의 PTVS(Python Tools for Visual Studio), 이클립스 기반의 PyDev, 그리고 최근 인기가 높은 JetBrains의 PyCharm이 있습니다. 이 책에서는 이 가운데 최근 인기가 높고 무료로 사용할 수 있는 PyCharm을 사용하겠습니다.

- PTVS: https://github.com/Microsoft/PTVS
- PyDev: http://www.pydev.org/
- PyCharm: https://www.jetbrains.com/pycharm/download/

PyCharm 설치 프로그램 다운로드

컴퓨터에 PyCharm을 설치하려면 먼저 PyCharm 다운로드 페이지[1]로 이동한 후 그림 8.1과 같이 Community Edition을 다운로드합니다. 참고로 다운로드 위치나 웹 페이지의 구성은 조금씩 바뀌므로 그림 8.1과 다르면 인터넷의 다른 글을 참고해서 Community Edition을 다운로드하면 됩니다.

1 https://www.jetbrains.com/pycharm/download/

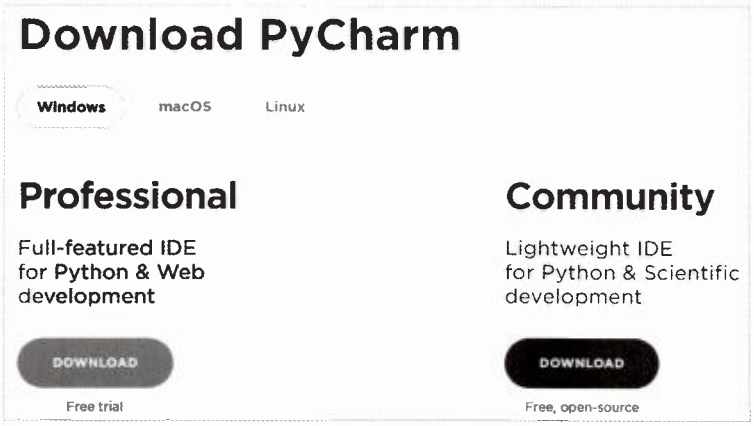


그림 8.1 PyCharm 커뮤니티 에디션 다운로드 페이지

PyCharm은 유료 버전인 Professional Edition과 무료 버전인 Community Edition이 있습니다. 물론 Professional Edition이 더 많은 기능을 지원하지만 Community Edition도 기본적인 프로그램을 개발하기에는 충분한 기능을 제공합니다.

PyCharm 설치

다운로드한 Community Edition 설치 파일을 실행하면 그림 8.2와 같은 설치 화면이 나타납니다. [Next] 버튼을 클릭해 다음 화면으로 이동합니다.



그림 8.2 PyCharm 설치 과정 1

그림 8.3은 설치 위치를 변경할 수 있는 화면입니다. 기본 설정을 그대로 사용하는 것으로 하고 [Next] 버튼을 클릭해 다음 설치 화면으로 이동합니다.

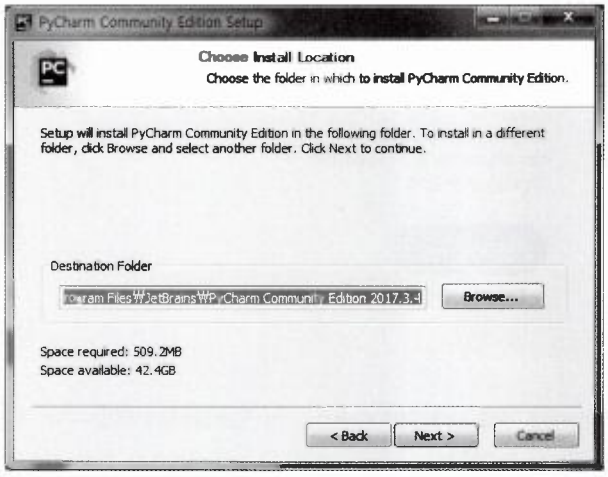


그림 8.3 PyCharm 설치 과정 2

자신의 윈도우가 64비트 버전이면 그림 8.4와 같이 64-bit launcher 항목에 체크한 후 [Next] 버튼을 클릭합니다. 잘 모르는 분들은 모두 체크하시면 됩니다.

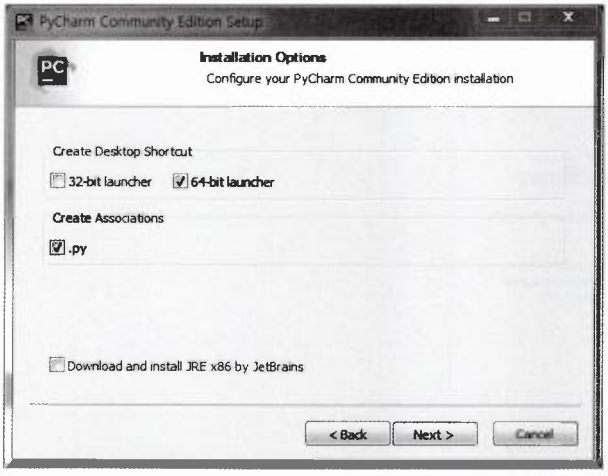


그림 8.4 PyCharm 설치 과정 3

그림 8.5는 윈도우 시작 메뉴 폴더에 등록되는 이름을 변경할 수 있는 화면입니다. 기본값을 그대로 두고 [Install] 버튼을 클릭해 설치를 진행합니다.

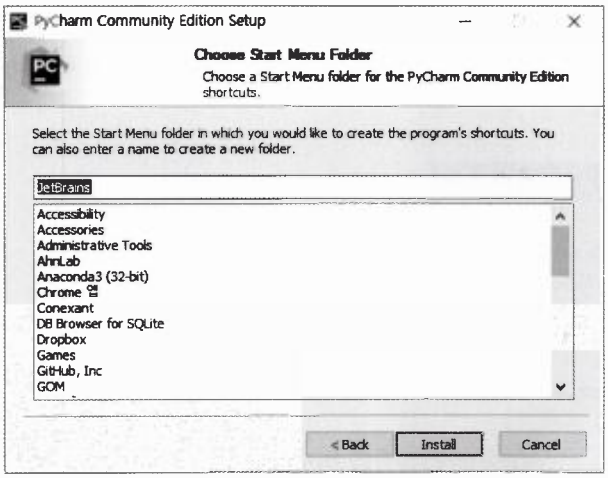


그림 8.5 PyCharm 설치 과정 4

그림 8.6은 PyCharm의 설치가 끝났음을 알리는 화면입니다. [Finish] 버튼을 클릭해 설치를 마칩니다.

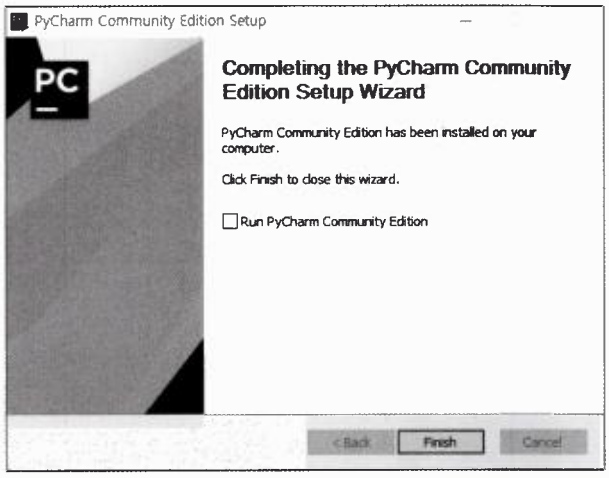


그림 8.6 PyCharm 설치 과정 5

PyCharm 초기 설정

PyCharm 설치가 완료되면 윈도우 바탕 화면에 PyCharm 아이콘이 만들어집니다. 해당 아이콘을 클릭해 PyCharm을 실행해 봅시다. PyCharm을 설치한 후 처음 실행하면 그림 8.7과 같은 화면이 나타납니다. 설정값을 그대로 둔 채 [OK] 버튼을 클릭합니다. Data Sharing Options 창에서는 [Cancel] 버튼을 클릭합니다.



그림 8.7 PyCharm 초기 설정 과정 1

그림 8.8은 PyCharm 키보드 단축키와 IDE 테마 등을 선택하는 화면입니다. 이 부분은 각자 취향에 맞춰 설정하는 부분인데 저자는 그림 8.8과 같이 IntelliJ 테마를 선택했고 [Skip Remaining and Set Defaults] 버튼을 눌러 나머지는 기본 값으로 설정했습니다.

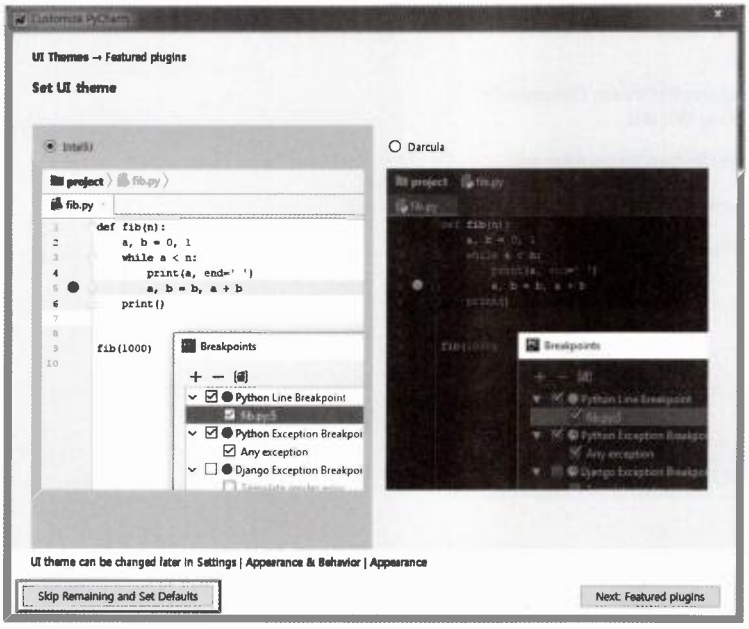


그림 8.8 PyCharm 초기 설정 과정 2

프로젝트 생성

PyCharm은 다른 IDE와 마찬가지로 프로젝트 단위로 소스 파일을 관리합니다. 지금까지는 Python IDLE를 통해 프로그래밍했기 때문에 따로 작성된 코드를 파일로 저장하지는 않았습니다. 그러나 앞으로 큰 규모의 프로그램을 개발하기 위해 작성된 코드를 컴퓨터에 저장하겠습니다.

먼저 PyCharm을 실행한 후 그림 8.9와 같이 [Create New Project]를 클릭해 프로젝트 생성 화면으로 이동합니다.

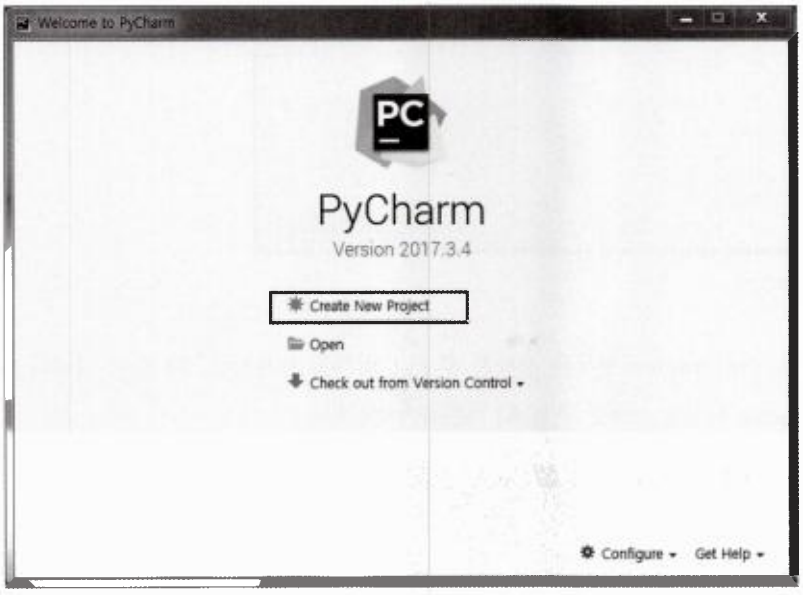


그림 8.9 PyCharm 프로젝트 생성 1

그림 8.10은 프로젝트 저장 위치와 파이썬 인터프리터를 설정하는 화면입니다. 그림 8.10과 같이 [Location]의 마지막 부분을 'HelloWorld'라는 이름으로 변경한 후 Project Interpreter 항목을 클릭합니다. Existing interpreter 항목을 선택한 후 톱니바퀴 모양의 아이콘을 클릭해 [Add Local] 메뉴를 선택합니다.

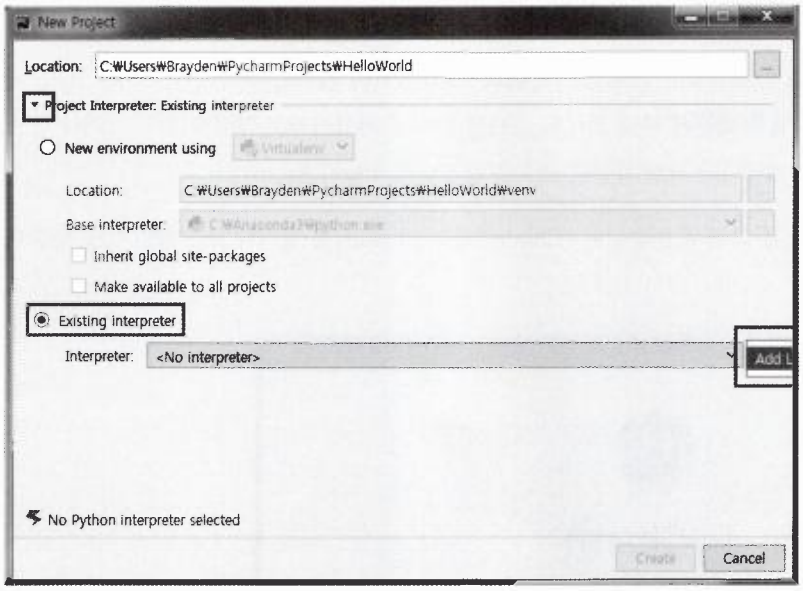


그림 8.10 PyCharm 프로젝트 생성 2

그림 8.11과 같이 System Interpreter 항목을 선택한 후 [...] 버튼을 누릅니다. 새 창이 나타나면 C:\Anaconda3\python.exe 파일을 선택한 후 [OK] 버튼을 누릅니다.

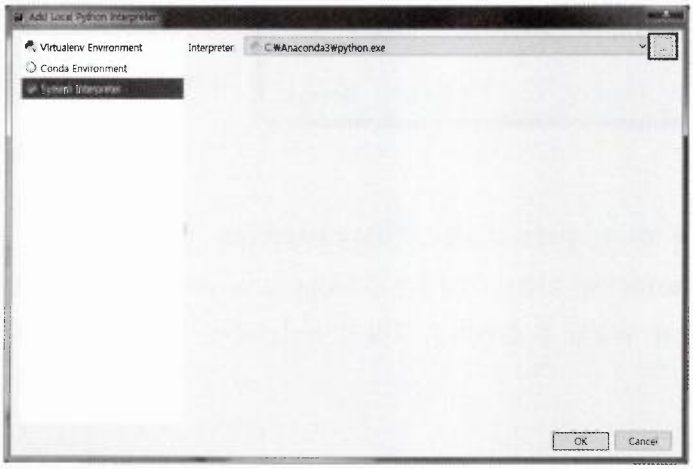


그림 8.11 PyCharm 프로젝트 생성 3

그림 8.12와 같이 설정했다면 [Create] 버튼을 눌러 프로젝트를 생성합니다.

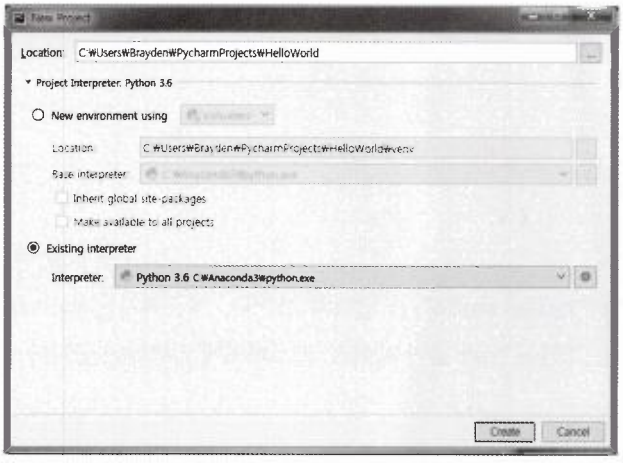


그림 8.12 PyCharm 프로젝트 생성 4

그림 8.13은 실행된 PyCharm의 화면 구성입니다. 프로젝트 부분에 HelloWorld가 보입니다. 이는 그림 8.10의 Location 부분에서 HelloWorld를 사용했기 때문입니다.

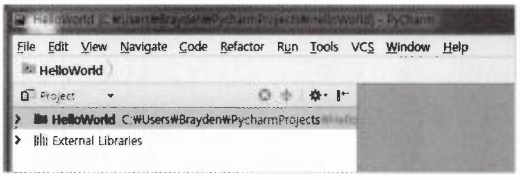


그림 8.13 PyCharm 프로젝트 생성 5

생성된 HelloWorld라는 프로젝트는 현재 비어있는 상태입니다. 프로젝트에 파이썬 파일을 추가하려면 그림 8.14와 같이 HelloWorld라는 프로젝트명에 마우스 오른쪽 버튼을 클릭한 후 [New]와 [Python File]을 순서대로 클릭합니다.

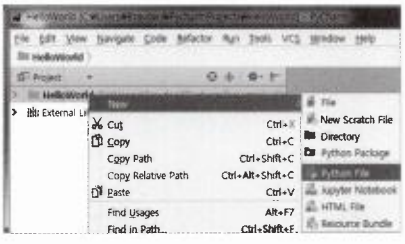


그림 8.14 PyCharm 파일 추가 1

그림 8.15와 같이 새 파일을 생성하는 창에서 [Name]에 'hello'를 입력합니다. [OK] 버튼을 클릭하면 hello.py라는 파일이 프로젝트에 추가됩니다.

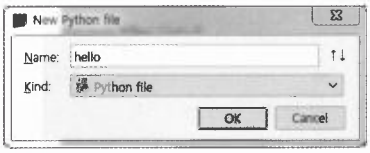


그림 8.15 PyCharm 파일 추가 2

hello.py 파일에 그림 8.16과 같이 print("hello world")를 입력해 봅시다. 지금까지는 Python IDLE를 통해 명령어를 입력했지만 앞으로는 그림 8.16과 같이 파이썬 코드 파일에 입력하면 됩니다.

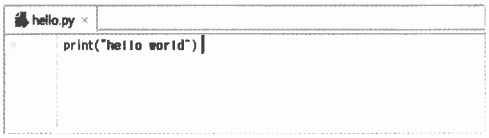


그림 8.16 파이썬 코드 작성

Python IDLE는 인터프리터라서 명령어를 입력할 때마다 해당 명령어를 바로 실행했습니다. 이와 달리 PyCharm은 코드를 작성한 후 실행 명령을 내리는 시점에 프로그램을 실행합니다. PyCharm에서 소스코드를 실행하려면 그림 8.17과 같이 소스코드에 마우스 오른쪽 버튼을 눌러 [Run] 메뉴를 클릭하거나 단축키인 Ctrl + Shift + F10을 누르면 됩니다. 참고로 PyCharm을 설치한 후 처음 실행하면 실행을 준비하는 데 시간이 소요됩니다. [Run] 메뉴가 활성화되지 않았다면 조금 기다렸다가 시도해보시기 바랍니다.

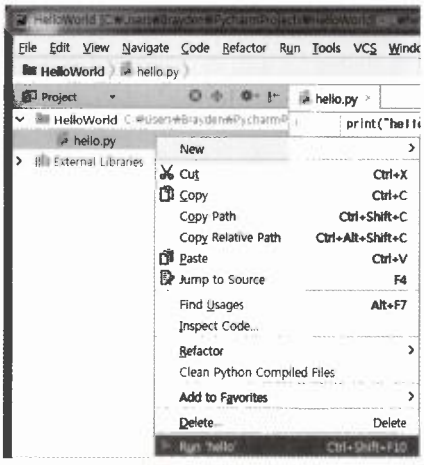


그림 8.17 파이썬 코드 실행 1

단축키 또는 [Run] 메뉴를 선택해 파이썬 코드를 실행하면 그림 8.18과 같이 PyCharm 하단부의 실행 창에 'hello world'라는 문자열이 출력됩니다.

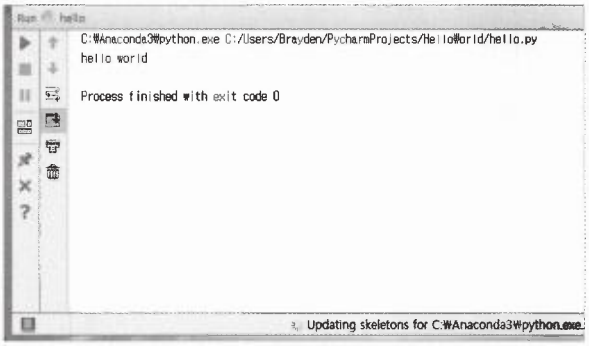


그림 8.18 파이썬 코드 실행 2

PyCharm 기타 설정

이번 절에서는 PyCharm에서 프로그램을 작성할 때 유용하게 사용할 수 있는 몇 가지 설정을 소개하겠습니다.

첫 번째로 소개할 내용은 파이썬 에디터 창에서 소스코드의 라인 넘버를 보이게 하는 설정입니다. 가끔 다른 누군가에게 코드의 특정 부분을 설명할 때 '몇 번째 줄에 있는 코드를 보라'처럼 표현하곤 합니다. PyCharm의 코드 에디터에서 라인 넘버를 보이게 하려면 [File] → [Settings] → [Editor] → [General] → [Appearance] 메뉴를 차례로 선택한 후 그림 8.19와 같이 [Show line numbers] 항목을 체크하면 됩니다.

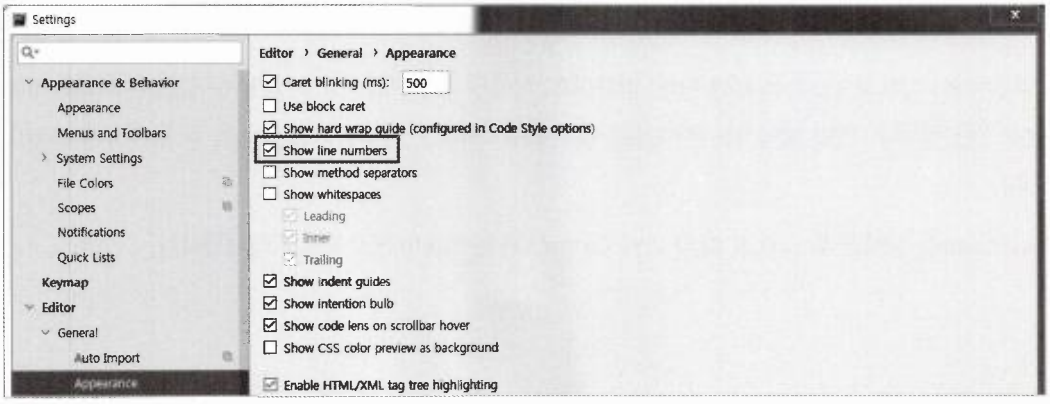


그림 8.19 라인 넘버 설정

두 번째로 소개할 내용은 PyCharm에서 파이썬 인터프리터를 사용하는 기능입니다. PyCharm으로 프로그램을 작성하다 보면 간단한 코드를 파이썬 인터프리터로 확인해보고 싶을 때가 있습니다. [Tools] → [Python Console] 메뉴를 차례로 선택하면 그림 8.20과 같이 PyCharm의 하단부에 Python IDLE와 같은 파이썬 콘솔(Python Console)이 나타납니다.

이처럼 Python IDLE과 비슷한 기능을 수행하는 파이썬 콘솔을 동시에 사용함으로써 더욱 편리하게 개발할 수 있습니다.

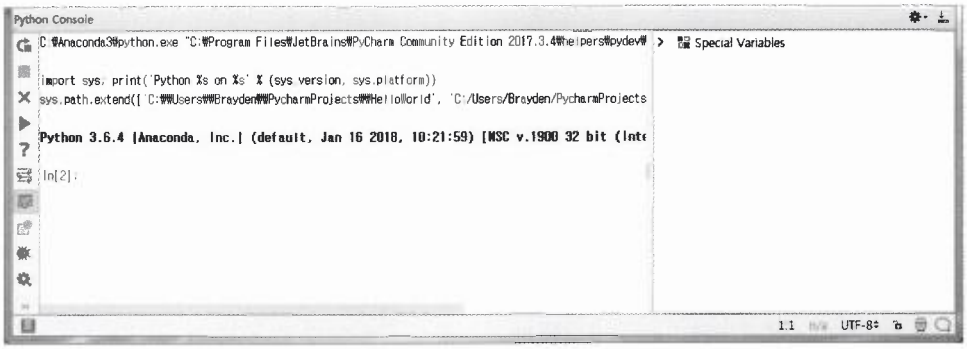


그림 8.20 파이썬 콘솔이 실행된 모습

마지막은 단축키 설정입니다. [File] → [Settings] → [Keymap]에서 Default를 Visual Studio로 변경한 후 [OK] 버튼을 클릭합니다. 앞으로는 파이썬 코드를 실행할 때 Ctrl + Shift + F10 대신 Ctrl + F9를 누르면 됩니다.

02 주소록 프로젝트

아이폰이나 안드로이드폰을 보면 이름, 전화번호, 이메일 등을 저장하고 관리할 수 있는 주소록이 기본으로 제공됩니다. 이번 절에서는 파이썬을 이용해 주소록을 관리하는 프로그램을 직접 개발해 보겠습니다.

PyCharm을 실행한 후 그림 8.21과 같이 'Contact'라는 이름의 프로젝트를 생성합니다.

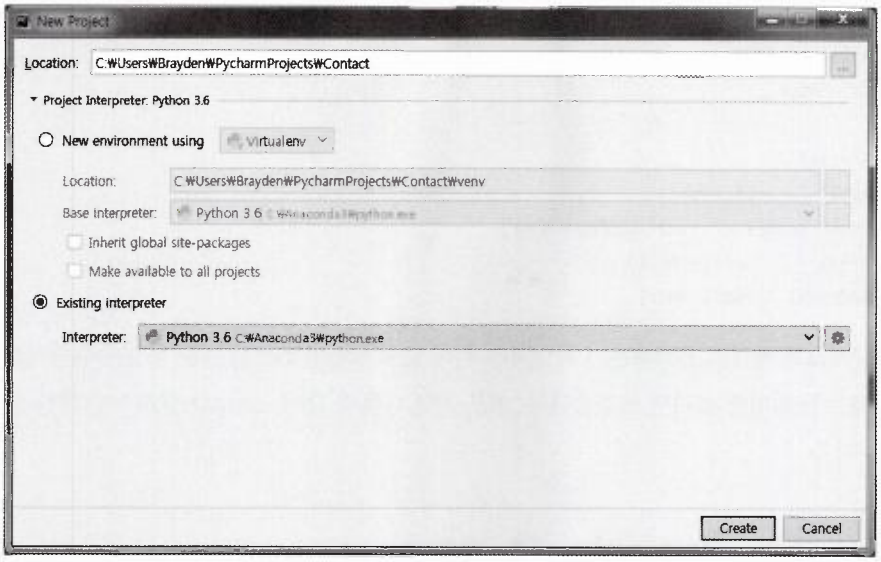


그림 8.21 Contact 프로젝트 생성

Contact 프로젝트에 contact.py 파일을 추가한 후 다음과 같이 기본 코드를 작성합니다. 그리고 Ctrl + F9 키를 눌러 화면에 'Contact'라는 문자열이 정상적으로 출력되는지 확인합니다.

```
def run():
    print("Contact")

if __name__ == "__main__":
    run()
```

Contact 클래스 만들기

아이폰이나 안드로이드폰의 연락처 프로그램을 살펴보면 연락처에는 이름, 전화번호, 이메일, 주소 등을 입력할 수 있습니다. 이를 파이썬에서 제공하는 기본 자료형으로 표현하기보다 하나의 클래스로 정의하는 것이 좋습니다.

6장에서 배운 클래스를 이용해 다음과 같이 'Contact'라는 이름의 클래스를 정의하겠습니다. 클래스 인스턴스를 생성할 때 이름, 전화번호, 이메일, 주소를 입력받을 수 있게 생성자를 선언하고 인스턴스 변수에 저장된 정보를 화면에 출력하기 위해 print_info라는 메서드도 정의했습니다.

```
class Contact:
    def __init__(self, name, phone_number, e_mail, addr):
        self.name = name
```

```
        self.phone_number = phone_number
        self.e_mail = e_mail
        self.addr = addr

    def print_info(self):
        print("Name: ", self.name)
        print("Phone Number: ", self.phone_number)
        print("E-mail: ", self.e_mail)
        print("Address: ", self.addr)
```

정의한 Contact 클래스가 제대로 동작하는지 확인하기 위해 앞서 작성한 run 함수를 수정해 연락처를 입력하고 입력된 정보를 화면에 출력해 보겠습니다. 이를 위해 다음과 같이 Contact 클래스에 대한 인스턴스를 생성합니다.

```
def run():
    kim = Contact('김일구', '010-8812-1193', 'ilgu.kim@python.com', 'Seoul')
    kim.print_info()
```

지금까지 작성한 전체 코드는 다음과 같습니다.

```
class Contact:
    def __init__(self, name, phone_number, e_mail, addr):
        self.name = name
        self.phone_number = phone_number
        self.e_mail = e_mail
        self.addr = addr

    def print_info(self):
        print("Name: ", self.name)
        print("Phone Number: ", self.phone_number)
        print("E-mail: ", self.e_mail)
        print("Address: ", self.addr)

def run():
    kim = Contact('김일구', '010-8812-1193', 'ilgu.kim@python.com', 'Seoul')
    kim.print_info()

if __name__ == "__main__":
    run()
```

PyCharm에서 Ctrl + F9 또는 Ctrl + F5를 눌러 지금까지 작성한 프로그램이 정상적으로 동작하는지 확인해보기 바랍니다. 그림 8.22와 같이 run 함수에서 입력한 값이 정상적으로 출력되면 됩니다.



그림 8.22 연락처 정보 출력

사용자로부터 데이터 입력받기

이번 절에서는 사용자로부터 데이터를 입력받아 보겠습니다. 파이썬에서 사용자로부터 데이터를 입력받을 때는 input 함수를 사용합니다. 사용자로부터 데이터를 입력받는 함수인 set_contact 함수를 새로 정의하고 run 함수에서 set_contact 함수를 호출하도록 코드를 변경해 봅시다.

```
class Contact:
    def __init__(self, name, phone_number, e_mail, addr):
        self.name = name
        self.phone_number = phone_number
        self.e_mail = e_mail
        self.addr = addr

    def print_info(self):
        print("Name: ", self.name)
        print("Phone Number: ", self.phone_number)
        print("E-mail: ", self.e_mail)
        print("Address: ", self.addr)

def set_contact():
    name = input("Name: ")
    phone_number = input("Phone Number: ")
    e_mail = input("E-mail: ")
    addr = input("Address: ")
    print(name, phone_number, e_mail, addr)

def run():
    set_contact()

if __name__ == "__main__":
    run()
```

수정한 코드를 실행하면 PyCharm의 아래쪽에 위치하는 실행 창을 통해 데이터를 입력할 수 있습니다. 데이터 입력이 완료되면 set_contact 함수의 마지막 라인에 있는 print 코드로 인해 지금까지 입력된 데이터가 화면에 출력됩니다. 그림 8.23과 같이 입력한 데이터와 출력된 데이터를 비교해 정상적으로 데이터가 변수에 저장되는지 확인해 봅시다.



그림 8.23 데이터 입력

메인 메뉴 구성하기

여러분이 작성할 연락처 프로그램은 기본적으로 연락처 입력, 연락처 출력, 연락처 삭제, 종료 기능을 제공합니다. 작성된 연락처 프로그램은 계속 실행 상태를 유지하다가 사용자로부터 종료 명령을 받으면 프로그램이 종료해야 합니다. 이번 절에서는 이러한 기본 실행 구조를 파이썬 코드로 작성해 보겠습니다.

작성한 프로그램이 한 번 실행됐을 때 종료하지 않은 상태로 계속 실행되게 하려면 무한 루프를 사용해야 합니다. 파이썬에서는 'while 1'이라는 구문을 사용하면 프로그램이 계속 실행되도록 만들 수 있습니다. 하지만 사용자로부터 종료 명령을 받았을 때는 프로그램이 종료해야겠지요? 이를 위해서는 while 문 내부에서 if 문을 사용해 조건에 따라 while 문이 종료하도록 만들면 됩니다.

여기서 while 문은 결국 반복문이기 때문에 이를 종료하려면 반복문을 빠져나오면 됩니다. 이러한 용도로 사용하는 구문이 바로 break 문입니다. 지금까지 설명한 내용을 코드로 작성하면 다음과 같습니다. 참고로 지금부터는 지면상 새로 추가된 부분이나 수정된 코드만 표시하겠습니다.

```
def print_menu():
    print("1. 연락처 입력")
    print("2. 연락처 출력")
    print("3. 연락처 삭제")
    print("4. 종료")
    menu = input("메뉴선택: ")
    return int(menu)
```

```
def run():
    while 1:
        menu = print_menu()
        if menu == 4:
            break
```

지금까지 작성한 프로그램을 실행해 봅니다. 그림 8.24는 현재까지 작성된 프로그램을 실행한 화면입니다. run 함수는 print_menu 함수를 호출하기 때문에 화면에 메인 메뉴가 출력됩니다. print_menu 함수는 input 함수를 통해 사용자로부터 메뉴를 입력받은 후 int라는 내장 함수를 통해 정수로 변환한 후 반환합니다.

run 함수에서는 사용자가 입력한 메뉴 값이 4와 같은지 확인하고 메뉴 값이 4일 때는 break를 사용해 무한 루프를 빠져나옵니다. 그림 8.24와 같이 여러분이 1번 메뉴를 선택했다면 while 문의 반복 구조로 인해 다시 한 번 화면에 메뉴가 출력됩니다. 이때 4를 입력하면 정상적으로 프로그램이 종료됨을 확인할 수 있습니다.

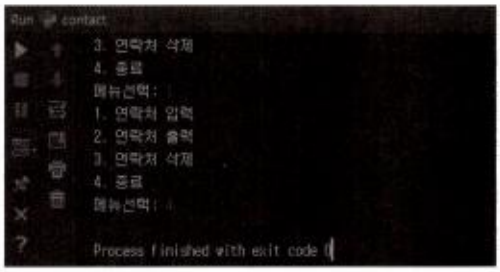


그림 8.24 프로그램의 메인 루프 테스트

연락처 입력 동작시키기

이번에는 1번 메뉴인 '연락처 입력' 기능이 정상적으로 동작하도록 프로그램을 변경해 보겠습니다. 먼저 사용자가 입력한 연락처 정보를 이용해 Contact 클래스 인스턴스를 생성하고 생성된 인스턴스를 반환하도록 set_contact 함수를 수정하겠습니다. set_contact 함수의 마지막 두 줄에서 Contact 클래스의 인스턴스를 생성하고 이를 반환하는 것을 확인할 수 있습니다.

```
def set_contact():
    name = input("Name: ")
    phone_number = input("Phone Number: ")
    e_mail = input("E-mail: ")
    addr = input("Address: ")
    contact = Contact(name, phone_number, e_mail, addr)
    return contact
```

현재 set_contact 함수는 단순히 함수가 정의된 상태입니다. 이 함수가 정상적으로 동작하려면 어디선가 해당 함수를 호출해야 합니다. set_contact 함수는 사용자로부터 연락처를 입력받을 때 사용하는 함수이므로 사용자가 '1. 연락처 입력' 메뉴를 선택했을 때 run 함수에서 set_contact 함수를 호출하면 됩니다.

다음과 같이 기존에 작성된 run 함수를 수정해 사용자가 1번 메뉴를 선택했을 때 set_contact 함수를 호출하도록 변경해 봅시다. 그리고 set_contact 함수의 반환값인 Contact 인스턴스를 저장하기 위해 contact_list라는 이름의 리스트 자료구조를 생성하고 해당 리스트에 생성된 인스턴스를 추가해 봅시다.

```
def run():
    contact_list = []
    while 1:
        menu = print_menu()
        if menu == 1:
            contact = set_contact()
            contact_list.append(contact)
        elif menu == 4:
            break
```

예제 8.1은 지금까지 작성한 코드입니다. 중간중간 코드를 수정하다가 흐름을 놓친 분들은 다음의 전체 코드를 살펴보시기 바랍니다.

예제 8.1 주소록 관리 프로그램의 소스코드(소스코드: book/ch08/01.py)

```
class Contact:
    def __init__(self, name, phone_number, e_mail, addr):
        self.name = name
        self.phone_number = phone_number
        self.e_mail = e_mail
        self.addr = addr

    def print_info(self):
        print("Name: ", self.name)
        print("Phone Number: ", self.phone_number)
        print("E-mail: ", self.e_mail)
        print("Address: ", self.addr)

def set_contact():
    name = input("Name: ")
```

```
    phone_number = input("Phone Number: ")
    e_mail = input("E-mail: ")
    addr = input("Address: ")
    contact = Contact(name, phone_number, e_mail, addr)
    return contact

def print_menu():
    print("1. 연락처 입력")
    print("2. 연락처 출력")
    print("3. 연락처 삭제")
    print("4. 종료")
    menu = input("메뉴선택: ")
    return int(menu)

def run():
    contact_list = []
    while 1:
        menu = print_menu()
        if menu == 1:
            contact = set_contact()
            contact_list.append(contact)
        elif menu == 4:
            break

if __name__ == "__main__":
    run()
```

예제 8.1의 코드를 실행한 후 그림 8.25와 같이 1번 메뉴를 선택해 연락처를 입력하고 4번 메뉴를 선택해 프로그램을 종료해 봅시다. 아직은 연락처 출력 기능이 없으므로 정상적으로 입력됐는지는 확인할 수 없지만 현 시점에서는 일단 에러가 발생하지 않는 것이 중요합니다.

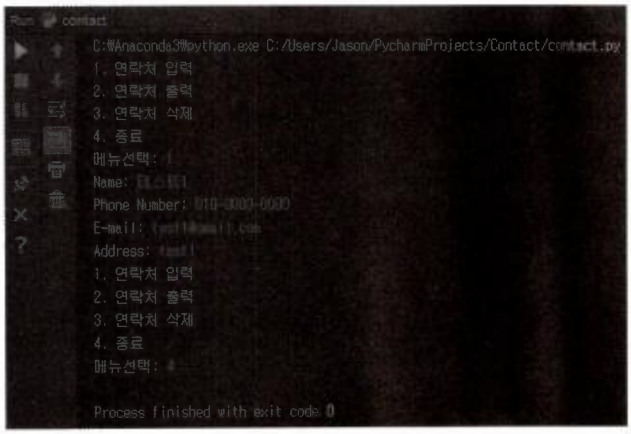


그림 8.25 주소록 프로그램의 중간 실행 결과

앞서 프로그램이 실행될 때 contact_list는 그림 8.26과 같이 입력된 '테스트1'에 대한 데이터를 가진 인스턴스를 바인딩하게 됩니다. 물론 4번 메뉴를 통해 프로그램을 종료하면 메모리에 할당된 모든 데이터가 자동으로 삭제됩니다.

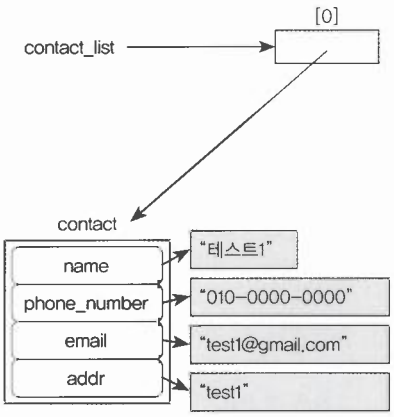


그림 8.26 contact_list의 상태

연락처 출력하기

이번에는 연락처를 출력하는 기능을 추가해 보겠습니다. 앞서 작성한 코드에서 연락처는 Contact라는 클래스의 인스턴스 형태로 저장되며, 인스턴스는 contact_list라는 이름의 리스트를 통해 순서대로 저장됩니다. 즉, 입력된 연락처를 모두 출력하려면 contact_list에 있는 인스턴스에 저장된 정보를 출력하면 되는 것입니다.

이미 여러분은 Contact 클래스를 정의할 때 인스턴스 변수의 데이터를 화면에 출력하는 메서드인 print_info 함수를 정의해 놓았습니다. 따라서 연락처를 출력하는 함수는 다음과 같이 간단히 구현할 수 있습니다.

```
def print_contact(contact_list):
    for contact in contact_list:
        contact.print_info()
```

print_contact 함수는 Contact 인스턴스를 저장하고 있는 리스트를 인자를 입력받은 후 for 문을 이용해 리스트에 저장된 인스턴스를 순회합니다. 이때 각 인스턴스에서 print_info 메서드를 호출합니다.

연락처 출력 기능은 메인 메뉴에서 '2. 연락처 출력'을 선택했을 때 수행돼야 합니다. 따라서 다음과 같이 run 함수도 수정합니다.

```
def run():
    contact_list = []
    while 1:
        menu = print_menu()
```

```
if menu == 1:
    contact = set_contact()
    contact_list.append(contact)
elif menu == 2:
    print_contact(contact_list)
elif menu == 4:
    break
```

지금까지 작성한 프로그램을 실행해 봅시다. 먼저 1번 메뉴를 선택한 후 임의로 연락처 정보를 입력하고 2번 메뉴를 선택하면 그림 8.27과 같이 입력된 정보가 정상적으로 출력되는 것을 확인할 수 있습니다.

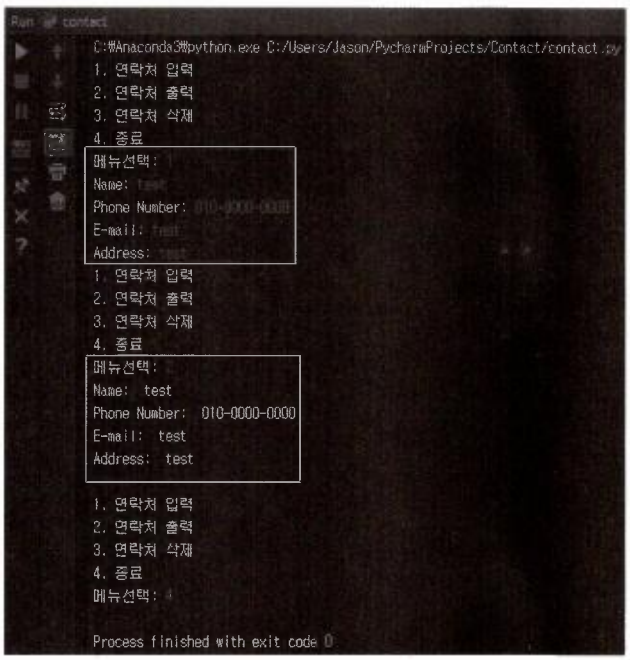


그림 8.27 연락처 출력 기능 확인

연락처 삭제하기

이번에는 입력된 연락처를 삭제하는 기능을 추가해 보겠습니다. 연락처를 삭제하려면 삭제하고자 하는 연락처의 이름을 입력받은 후 연락처 리스트에서 해당 이름을 찾아서 지우면 됩니다.

연락처 리스트에서 연락처를 삭제하기 위해 다음과 같이 delete_contact 함수를 작성합니다. 이 함수는 연락처 리스트와 삭제할 이름을 인자로 입력받습니다. delete_contact 함수는 연락처 리스트에 저

장된 인스턴스 중 삭제하고자 하는 연락처 이름과 같은 연락처가 있는지 확인한 후 같은 연락처가 있을 때 해당 연락처를 삭제합니다.

```
def delete_contact(contact_list, name):
    for i, contact in enumerate(contact_list):
        if contact.name == name:
            del contact_list[i]
```

연락처를 삭제하는 함수를 작성했다면 이번에는 run 함수를 수정하겠습니다. 연락처 삭제는 '3. 연락처 삭제' 메뉴를 선택했을 때 동작하는데, 이때 삭제할 연락처의 이름(Name)을 한 번 더 입력받습니다.

삭제할 연락처의 이름을 입력받았다면 연락처 리스트와 삭제할 이름을 delete_contact 함수의 인자로 전달하면 해당 함수를 통해 연락처가 삭제됩니다.

```
def run():
    contact_list = []
    while 1:
        menu = print_menu()
        if menu == 1:
            contact = set_contact()
            contact_list.append(contact)
        elif menu == 2:
            print_contact(contact_list)
        elif menu == 3:
            name = input("Name: ")
            delete_contact(contact_list, name)
        elif menu == 4:
            break
```

지금까지 작성한 주소록 프로젝트의 전체 소스코드는 예제 8.2와 같습니다. 여러분이 입력한 코드와 틀린 부분이 없는 확인하고 틀린 부분이 없다면 작성한 프로그램을 실행해 보기 바랍니다.

프로그램이 정상적으로 동작하는지 확인하기 위해 여러 명의 연락처를 입력해 보기 바랍니다. 그리고 그중 일부 연락처를 삭제한 후 연락처도 출력해 보기 바랍니다. 어떤가요? 지금까지 배운 파이썬의 기본 문법을 활용하고 프로그램을 단계별로 나눠서 작성하니 조금 복잡해 보이는 프로그램도 어렵지 않게 작성할 수 있지요?

예제 8.2 주소록 프로젝트의 전체 소스코드(소스코드: book/ch08/02.py)

```
class Contact:
    def __init__(self, name, phone_number, e_mail, addr):
        self.name = name
        self.phone_number = phone_number
        self.e_mail = e_mail
        self.addr = addr

    def print_info(self):
        print("Name: ", self.name)
        print("Phone Number: ", self.phone_number)
        print("E-mail: ", self.e_mail)
        print("Address: ", self.addr)

def set_contact():
    name = input("Name: ")
    phone_number = input("Phone Number: ")
    e_mail = input("E-mail: ")
    addr = input("Address: ")
    contact = Contact(name, phone_number, e_mail, addr)
    return contact

def print_contact(contact_list):
    for contact in contact_list:
        contact.print_info()

def delete_contact(contact_list, name):
    for i, contact in enumerate(contact_list):
        if contact.name == name:
            del contact_list[i]

def print_menu():
    print("1. 연락처 입력")
    print("2. 연락처 출력")
    print("3. 연락처 삭제")
    print("4. 종료")
    menu = input("메뉴선택: ")
    return int(menu)

def run():
    contact_list = []
```

```
    while 1:
        menu = print_menu()
        if menu == 1:
            contact = set_contact()
            contact_list.append(contact)
        elif menu == 2:
            print_contact(contact_list)
        elif menu == 3:
            name = input("Name: ")
            delete_contact(contact_list, name)
        elif menu == 4:
            break

if __name__ == "__main__":
    run()
```

03 주소록 프로젝트의 기능 향상

이번 절에서는 8.2절에서 작성한 연락처 관리 프로그램에 저장 기능을 추가해 보겠습니다. 지금까지 작성한 주소록 프로그램은 프로그램을 종료하면 입력된 모든 연락처가 사라집니다. 이는 입력받은 연락처가 Contact 클래스의 인스턴스로서 메모리에 할당됐다가 프로그램이 종료되면 메모리에 있는 모든 변수가 삭제되기 때문입니다.

이러한 문제를 해결하려면 어떻게 해야 할까요? 가장 간단한 방법은 프로그램이 종료되는 시점을 기준으로 유효한 연락처의 정보를 모두 파일로 저장하는 것입니다. 파일 입출력은 7장에서 살펴봤기 때문에 바로 해당 기능을 추가해 보겠습니다.

8.2절에서 구현한 메인 메뉴의 기능과 달리 파일 출력 기능은 메인 메뉴를 통해 제공되는 기능이 아닙니다. 단지 프로그램이 종료될 때 종료 시점에 유효한 데이터를 파일로 출력하면 됩니다. 반대로 프로그램이 실행될 때는 기존에 저장된 연락처를 불러와야(Load) 합니다.

연락처 저장 함수 작성하기

맨 먼저 작성해볼 함수는 프로그램이 종료될 때 유효한 연락처 정보를 파일로 저장하는 store_contact라는 함수입니다. 이 함수는 contact_list라는 리스트를 입력받은 후 해당 리스트에 있는 Contact 인스턴스를 순회하면서 데이터를 파일로 저장합니다.

```
def store_contact(contact_list):
    f = open("contact_db.txt", "wt")
    for contact in contact_list:
        f.write(contact.name + '\n')
        f.write(contact.phone_number + '\n')
        f.write(contact.e_mail + '\n')
        f.write(contact.addr + '\n')
    f.close()
```

store_contact 함수는 함수가 호출될 때 먼저 'contact_db.txt'라는 이름의 텍스트 파일을 'wt' 모드로 엽니다. 그런 다음 contact_list를 순회하면서 각 인스턴스의 정보를 write 함수를 통해 출력합니다. 이때 출력되는 각 정보가 파일에서 각 라인 단위로 저장되도록 '\n'을 사용했습니다. contact_list의 모든 데이터를 파일로 출력했다면 f.close()를 통해 파일을 닫습니다.

파일로 저장하는 기능은 프로그램이 종료될 때 호출하면 되므로 기존에 작성된 run 함수를 수정합니다.

```
def run():
    contact_list = []
    while 1:
        menu = print_menu()
        if menu == 1:
            contact = set_contact()
            contact_list.append(contact)
        elif menu == 2:
            print_contact(contact_list)
        elif menu == 3:
            name = input("Name: ")
            delete_contact(contact_list, name)
        elif menu == 4:
            store_contact(contact_list)
            break
```

수정된 프로그램을 실행한 후 그림 8.28과 같이 연락처를 하나 입력하고 프로그램을 종료해 봅시다.

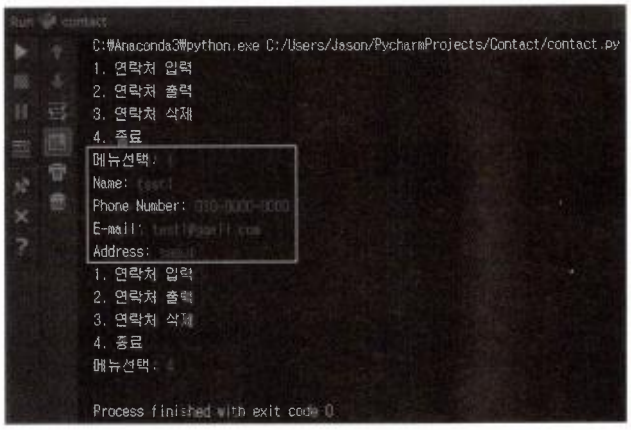


그림 8.28 연락처 입력 후 종료

그림 8.29와 같이 PyCharm의 프로젝트 화면을 보면 contact_db.txt라는 이름의 파일이 생성된 것을 확인할 수 있습니다.

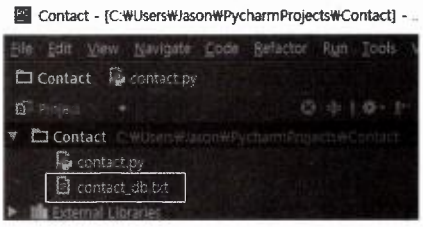


그림 8.29 생성된 파일 확인

해당 파일의 이름을 더블클릭하면 그림 8.30과 같이 PyCharm의 에디터 화면에서 생성된 텍스트 파일의 내용을 확인할 수 있습니다. 참고로 contact_db.txt 파일은 프로젝트 소스코드가 존재하는 디렉터리에 생성되는데, PyCharm에서는 이를 프로젝트 뷰에서 바로 보여주기 때문에 프로그램을 작성할 때보다 편리하게 파일 내용을 확인할 수 있습니다.



그림 8.30 생성된 파일 내용 확인

연락처 불러들이기

앞서 프로그램이 종료될 때 유효한 연락처를 파일로 저장하는 기능을 구현했습니다. 이번에는 이와 반대로 이전에 파일로 저장돼 있던 연락처 정보를 프로그램이 시작될 때 불러들이는 기능을 구현해 보겠습니다.

파일로부터 연락처를 로드하는 함수의 이름은 load_contact입니다. 이 함수 역시 contact_list를 인자로 받으며, 함수 내부에서 'contact_db.txt' 파일을 열어서 해당 파일을 라인 단위로 저장된 이름, 전화번호, 이메일, 주소로 읽어 들여 Contact 클래스의 인스턴스를 생성합니다. 그런 다음 생성한 인스턴스를 contact_list에 추가합니다.

```
def load_contact(contact_list):
    f = open("contact_db.txt", "rt")
    lines = f.readlines()
    num = len(lines) / 4
    num = int(num)

    for i in range(num):
        name = lines[4*i].rstrip('\n')
        phone = lines[4*i+1].rstrip('\n')
        email = lines[4*i+2].rstrip('\n')
        addr = lines[4*i+3].rstrip('\n')
        contact = Contact(name, phone, email, addr)
        contact_list.append(contact)
    f.close()
```

load_contact 함수는 먼저 readlines 함수를 이용해 파일에 있는 모든 데이터를 읽습니다. 연락처 하나당 4줄의 데이터가 존재하므로 파일에서 읽어 들인 전체 라인 수를 4로 나누어 몇 개의 데이터가 존재하는지 확인합니다. 나눗셈 연산을 수행하면 num 값이 실수가 되는데, 이 값을 int 내장 함수를 사용해 정수형으로 형변환합니다.

for 문에서는 num의 개수만큼 루프를 돌면서 lines 리스트에 저장된 데이터를 읽어 들여 Contact 클래스의 인스턴스를 생성하고 생성한 인스턴스를 contact_list에 추가합니다.

앞서 설명한 것처럼 파일로 저장된 연락처를 불러오는 것은 연락처 관리 프로그램이 실행될 때 이뤄져야 합니다. 따라서 다음과 같이 run 함수가 시작하는 부분에서 load_contact 함수를 호출하면 됩니다.

```
def run():
    contact_list = []
    load_contact(contact_list)
    while 1:
        menu = print_menu()
        if menu == 1:
            contact = set_contact()
            contact_list.append(contact)
        elif menu == 2:
```

```
            print_contact(contact_list)
        elif menu == 3:
            name = input("Name: ")
            delete_contact(contact_list, name)
        elif menu == 4:
            store_contact(contact_list)
            break
```

예제 8.3은 지금까지 작성한 주소록 프로젝트의 최종 코드입니다. 해당 코드는 파일 입출력 기능을 추가로 수행함으로써 프로그램이 종료되더라도 프로그램이 실행된 시점의 데이터를 하드디스크에 저장할 수 있습니다. 또한 프로그램이 시작될 때 하드디스크에 저장된 파일을 읽어 들여 프로그램이 종료되던 시점의 데이터를 그대로 복원하는 기능을 제공합니다.

예제 8.3 주소록 프로젝트의 최종 코드(소스코드: book/ch08/03.py)

```
class Contact:
    def __init__(self, name, phone_number, e_mail, addr):
        self.name = name
        self.phone_number = phone_number
        self.e_mail = e_mail
        self.addr = addr

    def print_info(self):
        print("Name: ", self.name)
        print("Phone Number: ", self.phone_number)
        print("E-mail: ", self.e_mail)
        print("Address: ", self.addr)

def set_contact():
    name = input("Name: ")
    phone_number = input("Phone Number: ")
    e_mail = input("E-mail: ")
    addr = input("Address: ")
    contact = Contact(name, phone_number, e_mail, addr)
    return contact

def print_contact(contact_list):
    for contact in contact_list:
        contact.print_info()
```

```
def delete_contact(contact_list, name):
    for i, contact in enumerate(contact_list):
        if contact.name == name:
            del contact_list[i]

def load_contact(contact_list):
    f = open("contact_db.txt", "rt")
    lines = f.readlines()
    num = len(lines) / 4
    num = int(num)

    for i in range(num):
        name = lines[4*i].rstrip('\n')
        phone = lines[4*i+1].rstrip('\n')
        email = lines[4*i+2].rstrip('\n')
        addr = lines[4*i+3].rstrip('\n')
        contact = Contact(name, phone, email, addr)
        contact_list.append(contact)
    f.close()

def store_contact(contact_list):
    f = open("contact_db.txt", "wt")
    for contact in contact_list:
        f.write(contact.name + '\n')
        f.write(contact.phone_number + '\n')
        f.write(contact.e_mail + '\n')
        f.write(contact.addr + '\n')
    f.close()

def print_menu():
    print("1. 연락처 입력")
    print("2. 연락처 출력")
    print("3. 연락처 삭제")
    print("4. 종료")
    menu = input("메뉴선택: ")
    return int(menu)

def run():
    contact_list = []
    load_contact(contact_list)
    while 1:
        menu = print_menu()
```

```
        if menu == 1:
            contact = set_contact()
            contact_list.append(contact)
        elif menu == 2:
            print_contact(contact_list)
        elif menu == 3:
            name = input("Name: ")
            delete_contact(contact_list, name)
        elif menu == 4:
            store_contact(contact_list)
            break

if __name__ == "__main__":
    run()
```

명령 프롬프트에서 프로그램 실행하기

지금까지 PyCharm을 통해 프로그램을 개발하고 동시에 실행해 보면서 프로그램을 작성했습니다. PyCharm을 이용해 프로그램을 작성하면 하드디스크에 파이썬 소스코드 형태로 프로그램이 저장됩니다. 이번 절에서는 윈도우의 명령 프롬프트를 이용해 직접 파이썬 코드를 실행해 보겠습니다.

윈도우에서 명령 프롬프트를 실행하려면 윈도우 시작 버튼을 누른 다음 [실행] → [cmd]를 입력하면 됩니다. 명령 프롬프트가 나타나면 지금까지 작성한 Contact 프로젝트 경로로 이동합니다. 저자의 PC를 기준으로 설명해 드리면 현재 프로젝트 파일은 'C:\Users\Jason\PyCharmProjects\Contact'에 있습니다. 따라서 다음과 같이 cd(change directory) 명령을 이용해 해당 디렉터리 경로로 이동합니다. 파이썬 프로젝트가 들어있는 곳으로 이동하고 나면 dir 명령어를 입력해 현재 디렉터리에 contact.py라는 파일이 있는지 확인합니다.

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Jason>cd PycharmProjects

C:\Users\Jason\PycharmProjects>cd Contact

C:\Users\Jason\PycharmProjects\Contact>dir
 C 드라이브의 볼륨에는 이름이 없습니다.
 볼륨 일련 번호: 3E7A-5817
```

```
 C:\Users\Jason\PycharmProjects\Contact 디렉터리

2016-10-05  오전 08:18    <DIR>          .
2016-10-05  오전 08:18    <DIR>          ..
2016-10-05  오전 08:15             2,319 contact.py
               1개 파일               2,319 바이트
               2개 디렉터리  93,350,182,912 바이트 남음

C:\Users\Jason\PycharmProjects\Contact>
```

cd 명령어를 이용해 정상적으로 프로젝트 경로로 이동했다면 'python contact.py'를 입력해 파이썬 프로그램을 실행할 수 있습니다.

```
C:\Users\Jason\PycharmProjects\Contact>python contact.py
1. 연락처 입력
2. 연락처 출력
3. 연락처 삭제
4. 종료
메뉴선택:
```

프로그램을 개발하는 과정에서는 PyCharm과 같은 IDE 상에서 파이썬 프로그램을 직접 실행하지만 프로그램 개발이 완료된 후에는 방금 설명한 바와 같이 직접 파이썬 소스코드를 사용해 프로그램을 실행하게 됩니다.

2부

증권사 API 기초

2부에서는 파이썬을 이용한 증권사 API 사용법을 다룹니다. 대신증권 CYBOS Plus, 이베스트투자증권 xing API, 키움증권 OpenAPI+의 기초 및 기본 API 사용법을 배울 수 있습니다.

09

파이썬과 COM

1부 '파이썬 프로그래밍'을 통해 파이썬의 기본 문법을 배웠습니다. 특히 8장에서는 파이썬을 이용해 간단한 프로그램도 직접 만들어봤습니다. 그러나 지금까지 배운 내용은 궁극적으로 만들어야 할 알고리즘 트레이딩과는 다소 거리가 있었습니다.

2부 '증권사 API 기초'에서는 알고리즘 트레이딩 시스템의 기본이 되는 증권사 API(Application Programming Interface)를 배우겠습니다. 여기서 API란 파이썬의 함수나 메서드와 같은 것입니다. 증권사가 제공하는 API를 이용하면 시세 정보를 얻거나 프로그램을 통한 매매와 같은 기본 동작을 수행할 수 있습니다.

그러나 여기에도 한 가지 문제가 있습니다. 국내 증권사에서 제공하는 API를 이용하면 프로그램을 쉽게 작성할 수 있는 것은 맞지만 국내 증권사의 API는 파이썬으로 구현돼 있지 않습니다. 파이썬으로 작성된 함수 또는 클래스라면 앞서 배운 내용을 바탕으로 어떻게 이용해볼 수는 있을 것 같은데, 다른 프로그래밍 언어로 작성된 함수를 파이썬에서 어떻게 호출할 수 있을까요?

이 질문의 답이 될 수 있는 기술이 바로 이번 장에서 다룰 마이크로소프트의 COM(Component Object Model)입니다. 이번 장을 통해 COM에 대한 간단히 알아보고 파이썬에서 COM을 사용하는 방법을 배워봅시다.

01 COM과 파이썬

COM이란 마이크로소프트에서 개발한 기술입니다. Component Object Model이라는 용어가 의미하듯이 여러 컴포넌트 객체를 이용해 프로그램을 개발하는 모델을 의미합니다. 컨테이너를 이용해 집을 지으면 쉽고 빨리 집을 지을 수 있는 것처럼 프로그램을 개발할 때도 특정 기능을 위해 개발된 Component Object를 이용하면 프로그램을 더 쉽고 빠르게 개발할 수 있습니다.

참고로 컴포넌트 객체란 파이썬으로 치면 클래스로부터 만들어지는 객체와 같은 것이라 이해하면 됩니다. 여기까지만 본다면 COM이라는 기술의 개념은 기존의 객체지향 프로그래밍과 크게 다르지 않습니다. 그러나 COM은 여기서 더 나아가 프로그래밍 언어와 상관없이 개발된 객체를 사용할 수 있게 해줍니다. 예를 들어, COM을 이용하면 C++ 같은 프로그래밍 언어로 개발된 객체도 파이썬에서 사용할 수 있게 됩니다.

벌써 클래스, 객체, 인스턴스라는 용어가 기억이 나지 않는 분들을 위해 간단한 예를 통해 설명해 드리겠습니다. 대신증권은 C/C++로 구현된 CpStockCode 클래스를 제공하는데 이 클래스와 비슷한 기능을 제공하는 클래스를 파이썬으로 구현해 보겠습니다. 참고로 대신증권에서 실제로 제공하는 API는 10장에서 좀 더 자세히 다룰 예정입니다.

```
class CpStockCode:
    def __init__(self):
        self.stocks = {'유한양행':'A000100'}

    def GetCount(self):
        return len(self.stocks)

    def NameToCode(self, name):
        return self.stocks[name]
```

CpStockCode 클래스는 생성자(__init__)와 GetCount, NameToCode라는 메서드를 갖고 있습니다. 여기서 GetCount 메서드는 주식 종목수를 리턴하는 메서드이고, NameToCode 메서드는 종목명을 입력하면 해당 종목에 대한 종목 코드를 리턴하는 메서드입니다.

참고로 그림 9.1과 같이 KRX 웹페이지(www.krx.co.kr)에서 제공하는 주식 종목 검색기에서 종목명을 입력하면 종목코드를 확인할 수 있습니다. 이와 비슷하게 증권사의 API에도 종목명을 입력하면 종목 코드를 반환하는 메서드가 있습니다. 여러분은 이러한 API를 프로그램 내에서 호출함으로써 종목 코드를 프로그램 내에서도 쉽게 얻을 수 있습니다.



그림 9.1 KRX 종목 검색기

6장에서 클래스를 정의하는 것은 붕어빵을 만드는 과정에서 붕어빵 틀을 만드는 것과 비슷하다고 설명했습니다. 붕어빵 틀에 반죽을 부은 후 붕어빵을 굽는 것처럼 여러분은 클래스(붕어빵 틀)를 이용해 인스턴스(붕어빵)를 만들 수 있었습니다.

```
instCpStockCode = CpStockCode()
```

위 코드는 CpStockCode 클래스를 이용해 instCpStockCode 객체를 생성한 예입니다. 즉, instCpStockCode 객체는 CpStockCode 클래스의 인스턴스입니다.

참고로 객체와 인스턴스라는 용어는 같은 의미이기 때문에 혼용하더라도 큰 무리는 없습니다. 다만, instCpStockCode와 같이 클래스를 통해 생성된 인스턴스 자체를 지칭할 때는 '무슨 무슨 객체'라는 표현을 주로 사용합니다. 이와 달리 'instCpStockCode 객체는 CpStockCode 클래스의 인스턴스'라는 표현에서처럼 객체가 어떤 클래스로부터 생성된 것인지를 의도할 때는 인스턴스라는 용어를 사용합니다.

객체와 인스턴스에 대한 차이는 이 정도로 설명하기로 하고 instCpStockCode 객체를 사용해 GetCount와 NameToCode 메서드를 호출해 보겠습니다. GetCount 메서드는 self 외에는 인자가 없으므로 호출할 때 인자를 전달하지 않아도 됩니다. NametoCode에는 종목 코드를 확인하고자 하는 종목의 종목명을 입력하면 됩니다. 참고로 NameToCode 메서드는 현재 '유한양행'이라는 종목에 대해서만 정상적으로 종목 코드를 리턴하도록 구현돼 있습니다.

```
print(instCpStockCode.GetCount())
print(instCpStockCode.NameToCode('유한양행'))
```

앞서 파이썬으로 구현된 CpStockCode 클래스에 대한 인스턴스를 생성할 때는 단순히 'instCpStockCode = CpStockCode()'와 같이 클래스 이름에 ()를 붙였습니다. 그러나 파이썬이 아닌 다른 프로그래밍 언어로 구현된 클래스에 대한 인스턴스는 파이썬에서 어떻게 생성할 수 있을까요?

파이썬에서 다른 프로그래밍 언어로 작성된 COM 객체를 생성하려면 win32com.client라는 모듈의 Dispatch 메서드를 사용하면 됩니다. 예를 들어, 마이크로소프트의 인터넷 익스플로러에 대한 객체를 생성하려면 다음과 같이 구현하면 됩니다.

```
import win32com.client

explore = win32com.client.Dispatch("InternetExplorer.Application")
explore.Visible = True
```

먼저 win32com.client라는 모듈을 사용하기 위해 모듈을 임포트(import)합니다. 그리고 win32com.client라는 모듈 내의 Dispatch 함수를 호출하는데, 이때 함수의 인자로 "InternetExplorer.Application"이라는 문자열을 사용했습니다. Dispatch 함수가 호출되면 객체가 생성되는데 explore라는 변수가 그림 9.2와 같이 생성된 COM 객체를 바인딩합니다.



그림 9.2 생성된 COM 객체에 대한 바인딩

COM 객체가 생성됐으니 COM 객체의 메서드를 호출하거나 속성을 변경할 수 있습니다. 'explore.Visible = True'는 Visible이라는 속성의 값을 True로 변경한 것입니다. 이 속성값을 True로 변경하면 그림 9.3과 같이 화면에 익스플로러가 나타납니다.



그림 9.3 COM 객체를 이용한 익스플로러 실행

어떤가요? 마이크로소프트의 COM이라는 기술을 사용해 인터넷 익스플로러를 파이썬 코드에서도 실행할 수 있다니 신기하지 않습니까? 컴퓨터에 마이크로소프트 워드가 설치된 분들은 다음 코드도 실행해보기 바랍니다.

```
import win32com.client

word = win32com.client.Dispatch("Word.Application")
word.Visible = True
```

이번에는 마이크로소프트 워드 프로그램이 자동으로 실행되는 것을 확인할 수 있습니다. 앞서 실행한 코드와 마찬가지로 win32com.client 모듈의 Dispatch 함수를 이용해 "Word.Application"에 대한 COM 객체를 생성한 후 객체의 Visible 속성값을 True로 변경했습니다.

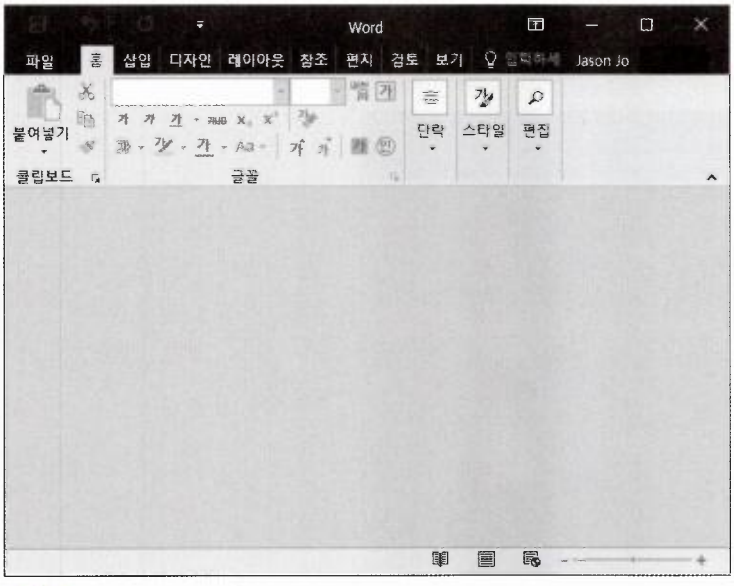


그림 9.4 COM 객체를 이용한 워드 실행

02 파이썬으로 엑셀 다루기

9.1절에서는 파이썬에서 COM 객체를 생성하는 방법과 생성된 객체를 통해 메서드를 호출하거나 속성값을 변경하는 법을 배웠습니다. 이번 절에서는 알고리즘 트레이딩을 위한 도구로 많이 사용되는 마이크로소프트 엑셀을 파이썬에서 다뤄보겠습니다.

앞서 워드나 인터넷 익스플로러에 대한 COM 객체를 생성한 것과 마찬가지로 엑셀에 대한 COM 객체를 생성해보기 바랍니다. 물론 다음 코드가 정상적으로 실행되려면 PC에 마이크로소프트 엑셀이 미리 설치돼 있어야 합니다.

```
import win32com.client
excel = win32com.client.Dispatch("Excel.Application")
```

엑셀도 워드나 익스플로러와 마찬가지로 Visible이라는 이름의 속성이 있습니다. 다음과 같이 Visible 속성의 값을 True로 변경하면 화면에 엑셀이 나타납니다. 사실 Dispatch 함수를 호출해서 Excel.Application에 대한 인스턴스를 생성했을 때부터 엑셀은 이미 실행돼 있었습니다. 다만 화면에 보이지는 않는 상태였을 뿐입니다.

```
import win32com.client
excel = win32com.client.Dispatch("Excel.Application")
excel.Visible = True
```

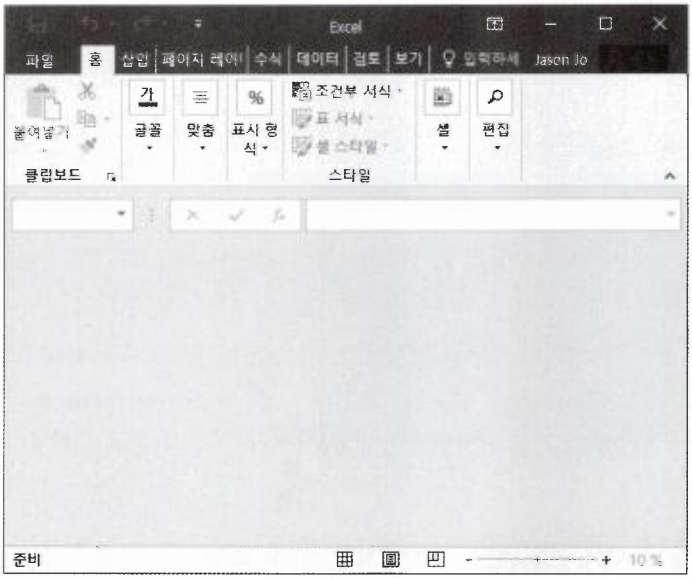


그림 9.5 COM으로 엑셀을 실행한 모습

엑셀이 화면에 나타났으니 이제 엑셀에 글자를 출력해 봅시다. 아마도 엑셀을 사용할 때 항상 Worksheet에서부터 시작했을 것입니다. 엑셀에는 여러 개의 Worksheet를 포함하는 상위 개념으로 Workbook이 있습니다. 즉, 엑셀 프로그램을 실행하면 Workbook이 자동으로 생성되고 'Sheet1'이라는 이름의 Worksheet가 자동으로 생성됩니다.

COM을 사용해 파이썬 코드로 엑셀을 직접 컨트롤할 때도 Workbook과 Worksheet의 관계를 고려해야 합니다. 먼저 Workbook을 추가한 후 해당 Workbook에서 'Sheet1'이라는 Worksheet를 선택합니다.

```
import win32com.client

excel = win32com.client.Dispatch("Excel.Application")
excel.Visible = True
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
```

위 코드를 실행하면 그림 9.6과 같이 우리가 평상시에 사용하는 엑셀의 '새 통합 문서' 상태가 됩니다.

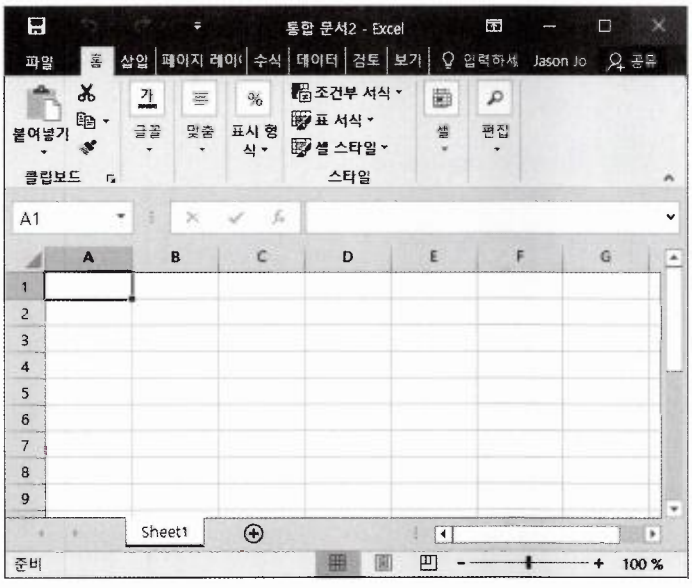


그림 9.6 파이썬을 이용한 엑셀 Workbook 생성

엑셀의 Worksheet에는 셀(Cell)이 있는데, 각 셀은 행과 열의 인덱스 값을 통해 구분됩니다. Worksheet의 특정 셀에 값을 입력하려면 Cells 속성을 사용하면 됩니다. Cells라는 키워드 다음에 행과 열에 대한 인덱스 값을 지정해 특정 셀에 접근할 수 있습니다.

예를 들어, 다음 코드에서 'Cells(1, 1)'이라는 표현은 첫 번째 행, 첫 번째 열에 해당하는 셀을 의미합니다. 즉, 그림 9.6의 Sheet1에서 A1 셀에 해당합니다. 셀이 선택되면 셀의 Value 속성에 'hello world'를 입력합니다.

```
import win32com.client

excel = win32com.client.Dispatch("Excel.Application")
excel.Visible = True
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
ws.Cells(1, 1).Value = "hello world"
```

위 코드를 실행하면 그림 9.7과 같이 'Sheet 1'의 A1 Cell에 'hello world'라는 문자열이 입력된 것을 확인할 수 있습니다.

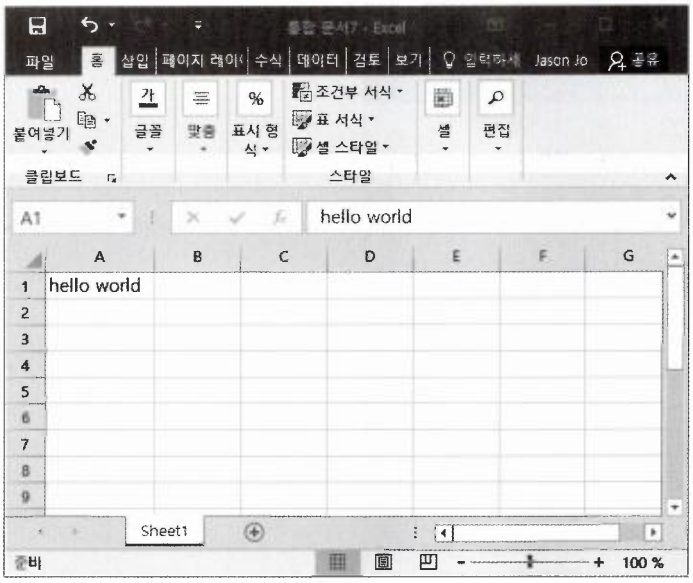


그림 9.7 A1 셀에 값 입력하기

엑셀에 정상적으로 데이터가 입력됐으니 해당 파일을 하드디스크에 저장하고 프로그램을 종료해 봅시다. 엑셀 프로그램 자체에도 [다른 이름으로 저장]이라는 저장 메뉴가 있듯이 엑셀 COM 객체에도 SaveAs라는 메서드가 있습니다. 다음과 같이 SaveAs 메서드에 파일을 저장할 경로를 인자로 전달하면 해당 경로에 엑셀 파일이 저장됩니다. SaveAs 메서드를 호출해 파일로 저장했다면 Quit 메서드를 호출해 엑셀을 종료합니다.

```
import win32com.client

excel = win32com.client.Dispatch("Excel.Application")
excel.Visible = True
```

```
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
ws.Cells(1, 1).Value = "hello world"
wb.SaveAs('c:\\Users\\Jason\\Desktop\\test.xlsx')
excel.Quit()
```

위 코드를 실행하면 그림 9.8과 같이 윈도우의 바탕 화면에 엑셀 파일이 저장됩니다. 참고로 SaveAs 함수의 인자로 입력되는 저장 경로는 환경에 맞춰 적절히 변경해야 합니다.

이처럼 파이썬에서 COM 기술을 이용하면 엑셀 프로그램을 직접 실행할 필요 없이 엑셀 실행, 데이터 입력, 저장 및 종료를 모두 프로그램 내에서 처리할 수 있습니다. 평상시에도 엑셀로 작업을 많이 한다면 파이썬을 이용해 자동으로 엑셀 문서를 만드는 프로그램을 만들어 보기 바랍니다.

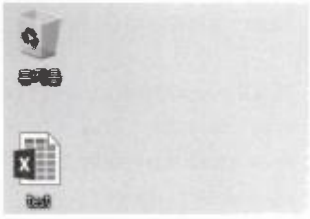


그림 9.8 바탕 화면에 저장된 엑셀 파일

엑셀 파일 읽기

이번에는 하드디스크에 이미 존재하는 엑셀 파일로부터 데이터를 읽어 오는 파이썬 프로그램을 작성해보겠습니다. 프로그램에서 사용할 샘플 엑셀 파일을 만들기 위해 그림 9.9와 같이 엑셀을 실행한 후 A1 셀에 python을 입력합니다.

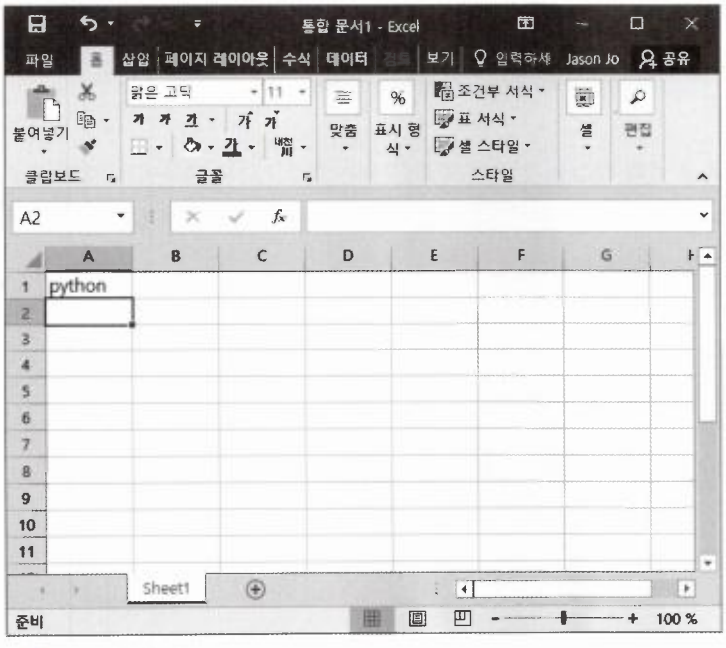


그림 9.9 입력 엑셀 파일 생성

엑셀 메뉴에서 [파일] → [다른 이름으로 저장]을 선택해 윈도우의 바탕 화면에 input.xlsx라는 이름으로 파일을 저장합니다.

파이썬에서 하드디스크에 있는 엑셀 파일을 읽는 것도 앞서 엑셀 파일을 쓰는 것과 비슷합니다. 다만 Workbook과 Worksheet가 이미 존재하기 때문에 기존에 있는 것 중 선택해서 사용해야 합니다. 먼저 다음 코드를 실행해 보기 바랍니다. Open 함수에 전달하는 인자는 윈도우 바탕 화면의 경로 및 저장한 파일의 이름에 맞춰 적절히 수정합니다.

```
import win32com.client

excel = win32com.client.Dispatch("Excel.Application")
excel.Visible = True
wb = excel.Workbooks.Open('C:\\Users\\Jason\\Desktop\\input.xlsx')
ws = wb.ActiveSheet
print(ws.Cells(1,1).Value)
excel.Quit()
```

위 코드에서 Open 함수가 호출되면 바로 Workbook 객체를 얻는 것을 확인할 수 있습니다. Workbook 객체를 통해 다시 Worksheet 객체를 얻을 수 있습니다. 이때 Worksheet 이름('Sheet1')을 사용해 여러 Worksheet 중 하나를 선택할 수도 있고, 위 코드와 현재 Active 상태인 Worksheet를 선택할 수도 있습니다.

Worksheet를 얻었다면 Worksheet를 통해 해당 Worksheet에 있는 셀에 접근해서 저장된 데이터 값을 읽어올 수 있습니다.

셀에 색깔 입히기

이번에는 엑셀 Worksheet에서 특정 셀의 색상을 변경해 보겠습니다. 엑셀 COM 객체에는 엑셀 프로그램에서 우리가 사용했던 기능이 거의 그대로 메서드나 속성 형태로 제공됩니다. 먼저 앞에서 사용한 input.xlsx 파일을 COM 객체를 통해 읽어 봅시다.

```
import win32com.client

excel = win32com.client.Dispatch("Excel.Application")
excel.Visible = True
wb = excel.Workbooks.Open('C:\\Users\\Jason\\Desktop\\input.xlsx')
ws = wb.ActiveSheet
```

B1 셀과 C1 셀에 'is'와 'good'이라는 문자열을 출력해 봅시다. 이를 위해 다음과 같이 Cells와 Range를 이용했습니다. 그리고 C1 셀의 색상을 변경하기 위해 Interior 객체의 ColorIndex 속성값을 변경했습니다.

```
ws.Cells(1,2).Value = "is"
ws.Range("C1").Value = "good"
ws.Range("C1").Interior.ColorIndex = 10
```

지금까지 작성한 코드를 실행하면 그림 9.10과 같이 B1과 C1 셀에 데이터가 입력되고 C1 셀의 색상이 변경된 것을 확인할 수 있습니다.

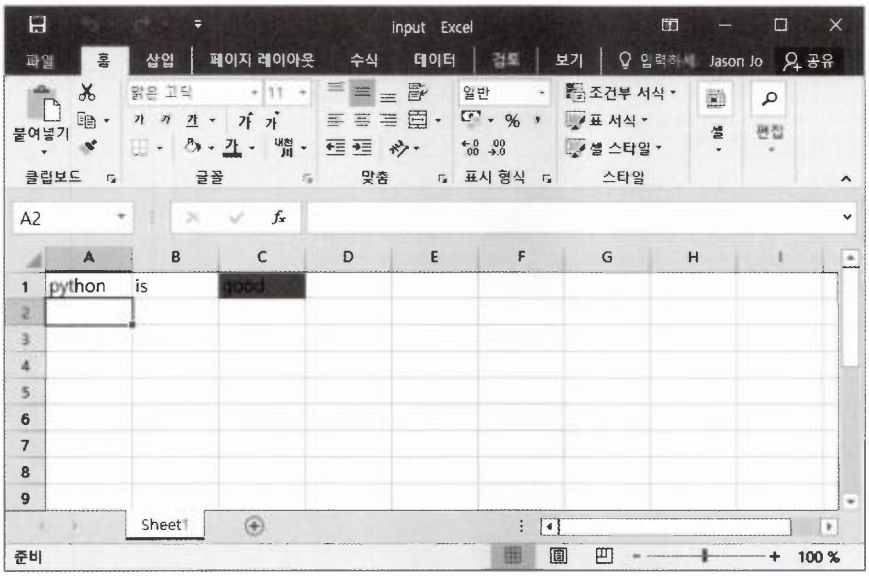


그림 9.10 엑셀 셀의 색상 변경

참고로 ColorIndex 속성은 0~56 범위의 값을 가질 수 있으며, 숫자별로 색상이 정해져 있습니다(자세한 색상 값은 마이크로소프트에서 제공하는 MSDN[1]을 참고합니다). 저 또한 이러한 정보는 그림 9.11과 같이 그때그때 구글 검색을 통해 찾습니다.

1 https://msdn.microsoft.com/en-us/library/cc296089(v=office.12).aspx

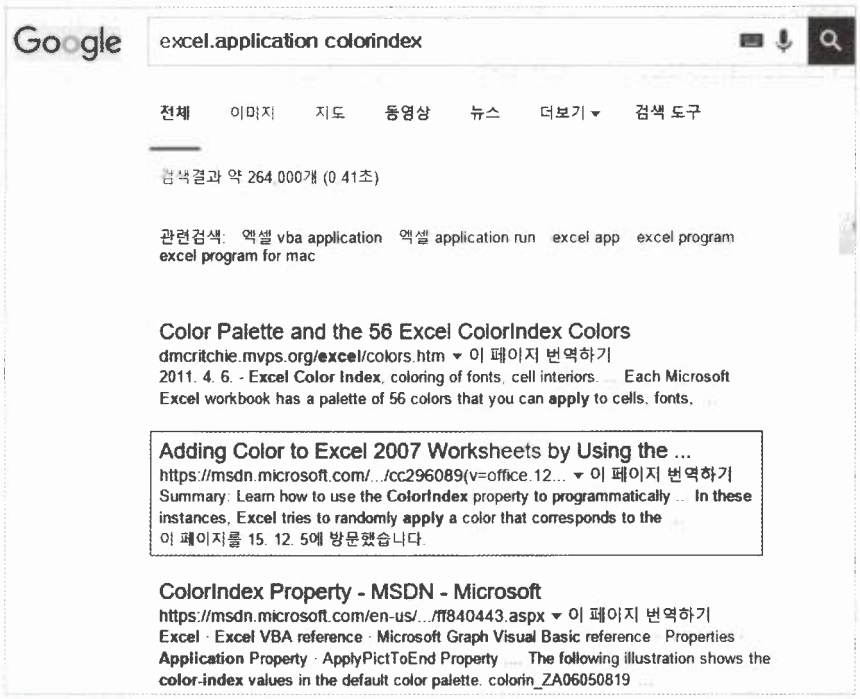


그림 9.11 검색을 통한 COM 객체의 속성 값 확인

앞서 Worksheet의 셀을 선택할 때 Cells와 Range라는 속성을 각각 이용했는데, 둘 다 셀을 선택하는 데 사용됩니다. 다만 Range 속성은 이름 자체가 의미하는 것처럼 여러 셀을 선택할 때 주로 사용합니다. 예를 들어, A2:C2 범위의 셀을 노란색으로 변경하려면 다음과 같이 작성하면 됩니다.

```
ws.Range("A2:C2").Interior.ColorIndex = 27
```

지금까지 COM을 이용해 마이크로소프트 인터넷 익스플로러, 워드, 엑셀 등을 조작해 보았습니다. 이를 통해 COM을 이용하면 다른 프로그래밍 언어로 작성된 객체도 쉽게 사용할 수 있음을 확인했습니다. 국내 증권사 API도 COM을 통해 지금까지 설명한 것과 유사한 방식으로 사용할 수 있습니다.

10

대신증권 API

현재 국내에는 수십 개의 증권사가 있지만 그중 일부 증권사만이 API(Application Programming Interface) 서비스를 제공하고 있습니다. 이 책에서는 국내 증권 API 서비스 중에서 가장 많은 사용자를 보유한 대신증권의 CybosPlus를 비롯해 이베스트투자증권의 xingAPI, 키움증권의 Open API+를 다루겠습니다. 아직 증권사 계좌가 없는 분들은 가까운 은행을 방문해 대신증권, 이베스트투자증권, 키움증권의 계좌를 한 번에 개설한 후 자신에게 맞는 API를 선택해서 사용하기 바랍니다.

이번 장에서는 먼저 대신증권의 API 서비스인 CybosPlus에 대해 다룹니다. 대신증권의 CybosPlus는 API 서비스를 제공해온 지 오래됐으며 초보자도 쉽게 사용할 수 있다는 장점이 있습니다.

01 개발 환경 구축

여러분은 이제 막 파이썬이라는 프로그래밍 언어의 문법만 배운 수준입니다. 영어로 치면 영문법 책 한 권 정도 간단히 읽어본 수준인 것입니다. 영문법 정도만 간단히 배운 사람에게 영어로 글을 써보라고 한다면 글쓰기를 잘할 수 있을까요?

지금 파이썬의 기초 문법 정도만 아는 상태인데 당장 주식 종목의 가격을 얻어오고 자동으로 매수/매도하는 프로그램을 만들려고 합니다. 아직 초보인 여러분들이 어떻게 그런 프로그램을 만들 수 있을까요?

정답은 바로 API에 있습니다. 앞으로 차근차근 배우겠지만, 주식 가격을 얻어오고, 매수/매도 주문을 넣는 데 사용하는 기본 함수를 증권사에서 제공합니다. 우리는 단지 파이썬 소스코드에서 관련 함수를 호출하고 리턴값을 확인하기만 하면 됩니다. 따라서 기본적인 파이썬 문법과 함수 호출 방법 정도만 알아도 웬만한 프로그램을 만들 수 있는 것입니다. 9장에서 COM을 이용해 워드와 엑셀을 제어했던 것을 생각하면 이런 상황이 쉽게 이해될 것입니다.

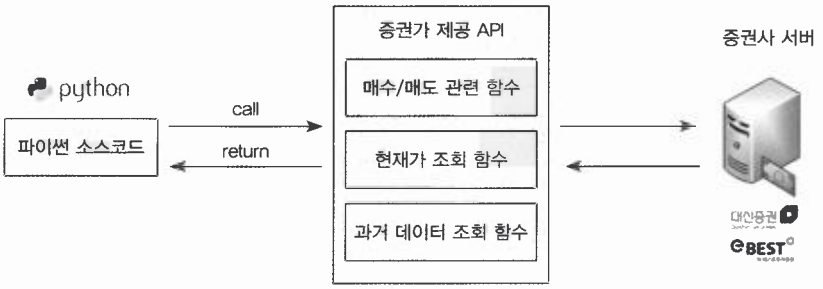


그림 10.1 증권사 API의 역할

그림 10.1은 여러분이 작성할 파이썬 소스코드, 증권사에서 제공하는 API, 실제 주문 및 기타 요청을 처리하는 증권사 서버의 관계를 나타냅니다. 파이썬 소스코드에서 COM 객체를 이용해 증권사에서 제공하는 API를 호출하면 해당 API는 증권사 서버에 해당 작업을 요청하게 됩니다.

계좌 개설 및 CYBOS 5 설치

대신증권에서 제공하는 API 서비스인 CybosPlus를 이용하려면 앞에서 설명한 것처럼 대신증권 계좌가 필요합니다. 참고로 증권사 계좌는 증권사 지점을 방문하지 않더라도 가까운 연계은행 지점을 방문해 개설할 수 있습니다. 또한 최근에는 영업점이나 은행을 방문하지 않고도 모바일을 통해 간편하게 계좌를 개설할 수 있는 비대면 계좌 개설을 지원합니다.

그림 10.2는 대신증권 홈페이지에 안내된 제휴은행 목록입니다. 연계은행을 통해 계좌를 개설하실 분은 이 중 한 은행을 방문한 후 대신증권 계좌를 개설해 달라고 요청하면 됩니다. 참고로 계좌를 개설할 때는 반드시 본인을 증명할 수 있는 신분증 등을 지참해야 합니다.



그림 10.2 대신증권 제휴은행

은행이나 영업점을 방문하기 어렵다면 비대면 계좌 개설이 편리합니다. 대신증권 웹 페이지(www.daishin.com)에서 [비대면계좌개설] 메뉴를 클릭하면 비대면 계좌 개설을 위한 상세 절차가 설명돼 있습니다.

대신증권 계좌를 개설했다면 PC에 CYBOS 5라는 프로그램을 설치해야 합니다. 먼저 대신증권 웹 페이지로 이동한 후 그림 10.3과 같이 웹 페이지 하단의 [다운로드센터] 메뉴를 선택합니다.

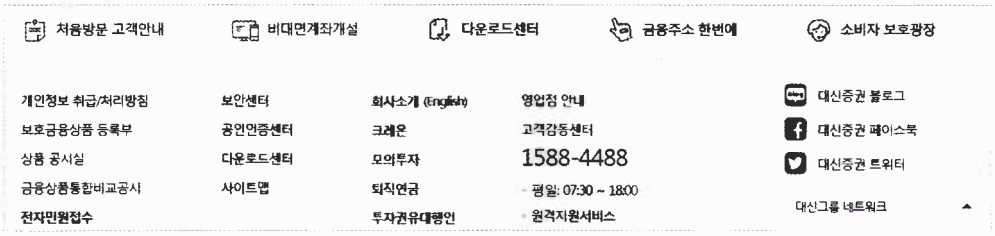


그림 10.3 대신증권 다운로드 센터

CYBOS 5는 대신증권 HTS 프로그램이고, CYBOS Plus는 대신증권의 API 서비스입니다. CYBOS Plus를 사용하려면 CYBOS 5가 기본적으로 설치돼 있어야 합니다. 그림 10.4와 같이 다운로드 버튼을 클릭해 CYBOS 5 설치 파일을 다운로드합니다.

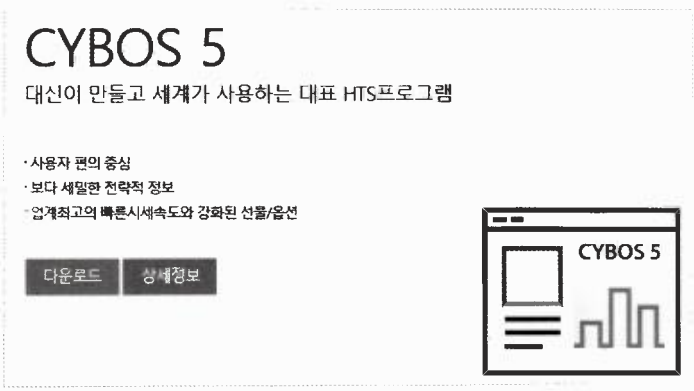


그림 10.4 CYBOS 5 다운로드

다운로드된 설치 파일을 더블클릭해 설치를 시작합니다. 그림 10.5와 같은 설치 마법사 화면이 나타나면 [다음] 버튼을 눌러 설치를 시작합니다.

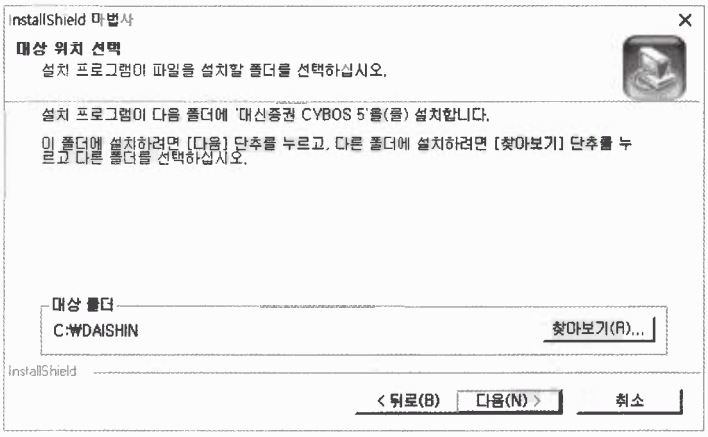


그림 10.5 대신증권 CYBOS 5 설치 1

설치가 완료되면 그림 10.6과 같은 화면이 나타납니다. [완료] 버튼을 눌러 설치 과정을 종료합니다.

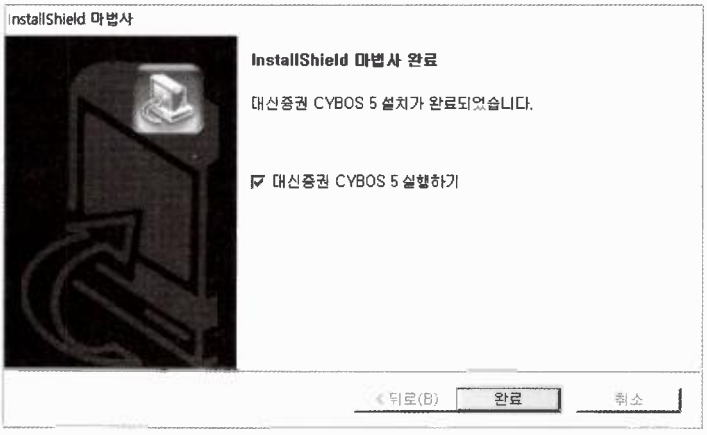


그림 10.6 대신증권 CYBOS 5 설치 2

CYBOS Plus 로그인

CYBOS Plus에 로그인하려면 통신 ID와 공인인증서를 등록해야 합니다. 대신증권 홈페이지의 [고객센터] → [신규고객가이드] → [처음 방문고객안내] 페이지를 참조해서 그림 10.7과 같이 정회원 ID를 등록합니다.

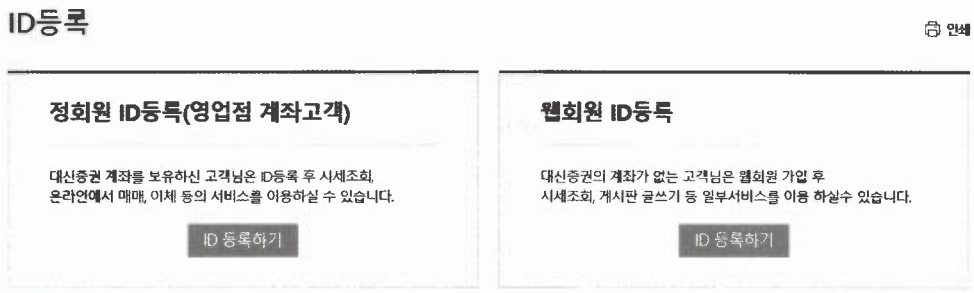


그림 10.7 ID 등록 메뉴

통신 ID 등록을 마치면 공인인증서를 등록해야 합니다. 참고로 범용 공인인증서가 아닌 은행에서 무료로 발급한 은행용 공인인증서는 증권 거래에 사용할 수 없습니다. 따라서 이 경우에는 증권사 홈페이지의 안내를 참조해서 증권 거래용 공인인증서를 새로 발급받아야 합니다. 다행히도 증권 거래용 공인인증서도 무료로 발급받을 수 있습니다.

통신 ID 등록과 공인인증서 발급을 모두 마치면 대신증권 CYBOS 5를 실행합니다. 그림 10.8과 같은 CYBOS 5 실행 화면에서 [CYBOS plus] 탭을 먼저 선택한 후 통신 ID, 통신 비밀번호, 공인인증 비밀번호를 입력한 후 [로그인] 버튼을 눌러 로그인합니다.

이 부분에서 많은 분들이 [CYBOS 5] 탭에서 로그인하는 실수를 범하곤 하는데, 대신증권의 API를 사용하려면 반드시 [CYBOS plus] 탭을 클릭한 후 로그인해야 합니다.



그림 10.8 CYBOS plus 로그인 화면

02 기초 API 익히기

대신증권은 CYBOS Plus에 대한 도움말 파일을 제공합니다. 파이썬에서 CYBOS Plus에서 제공하는 API를 사용할 때 이러한 도움말 파일을 참조하면 매우 편리합니다.

도움말 파일을 내려받기 위해 먼저 대신 증권 홈페이지에서 [다운로드센터] 페이지로 이동합니다. 그림 10.9와 같이 CYBOS Plus 항목의 [다운로드] 버튼을 클릭해 다운로드합니다.

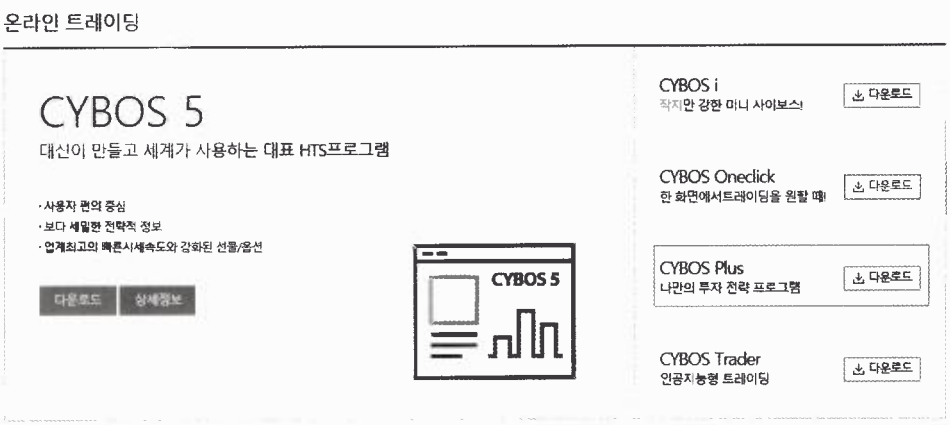


그림 10.9 CYBOS Plus 프로그램 샘플과 도움말 파일

도움말 파일은 *.zip 형태로 압축돼 있습니다. 도움말 파일의 압축을 해제한 후 CybosPlusHelp.chm 파일을 클릭하면 그림 10.10과 같이 도움말을 볼 수 있습니다.

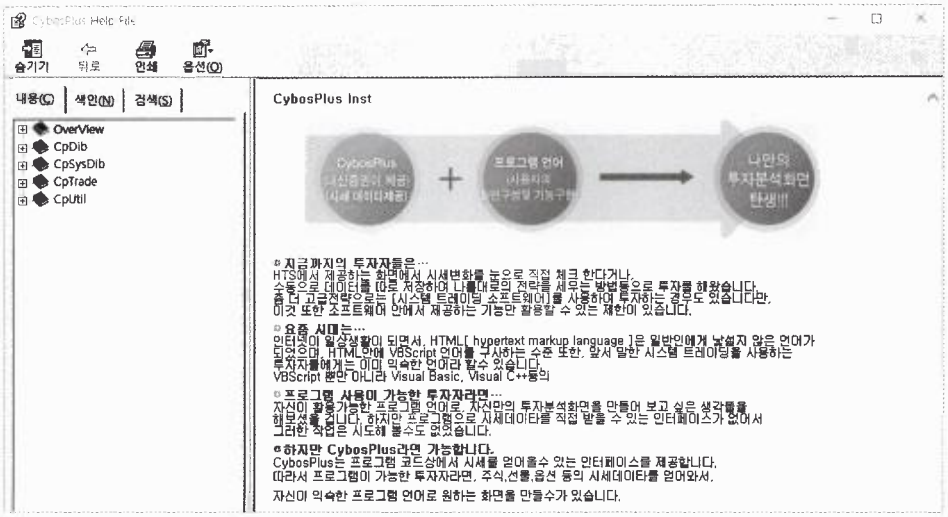


그림 10.10 CYBOS Plus 도움말 파일

연결 상태 확인

이번 절에서는 CYBOS Plus의 CpCybos 클래스를 이용해 CYBOS Plus의 연결 상태를 확인해보겠습니다. 먼저 그림 10.8을 참조해 CYBOS Plus에 로그인을 수행합니다. 파이썬 코딩은 IDLE나 PyCharm을 사용하면 됩니다. 다만 IDLE나 PyCharm을 실행할 때 반드시 관리자 권한으로 프로그램을 실행해야 합니다.

그림 10.11은 파이썬 IDLE를 관리자 권한으로 실행하는 모습입니다. 바탕 화면에 생성된 Python IDLE 아이콘에 마우스 오른쪽 버튼을 클릭하면 메뉴가 나타납니다. 여기서 [관리자 권한으로 실행]을 선택하면 됩니다.

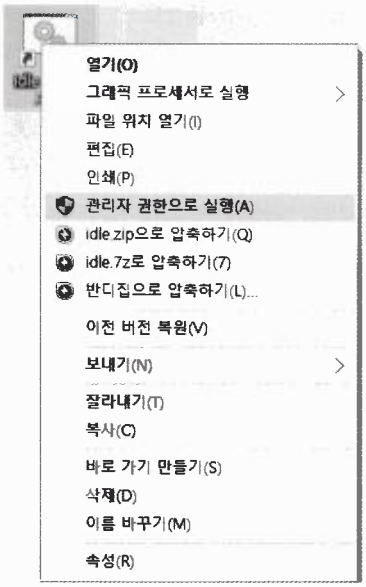


그림 10.11 IDLE을 관리자 권한으로 실행

PyCharm을 실행할 때도 PyCharm 실행 아이콘에 마우스 오른쪽 버튼 또는 Shift 키를 누른 상태에서 마우스 오른쪽 버튼을 클릭하면 그림 10.12와 같이 관리자 권한으로 실행할 수 있는 메뉴가 나타납니다.

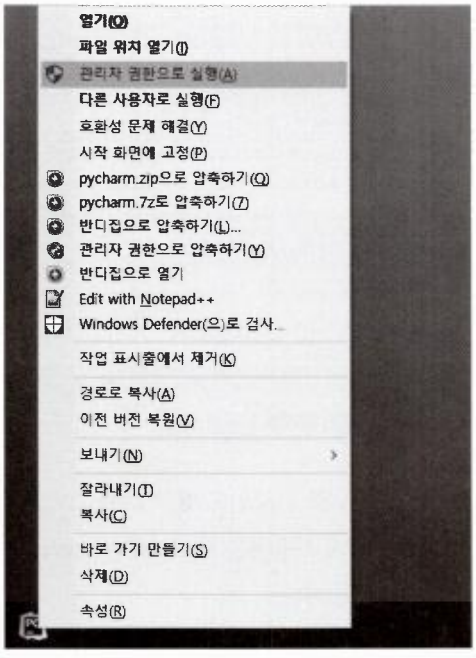


그림 10.12 PyCharm을 관리자 권한으로 실행

Python IDLE 또는 PyCharm을 관리자 권한으로 실행했다면 다음 코드를 Python IDLE 또는 PyCharm에서 실행해 보기 바랍니다.

예제 10.1 CYBOS Plus 연결 상태 확인(소스코드: book/ch10/01.py)

```
import win32com.client
instCpCybos = win32com.client.Dispatch("CpUtil.CpCybos")
print(instCpCybos.IsConnect)
```

1이라는 값이 출력되면 현재 CYBOS Plus가 정상적으로 연결된 것을 의미합니다. 이 값이 0으로 출력되면 CYBOS Plus와 연결되지 않은 상태입니다. 이 경우 10.1.2절에서 설명한 CYBOS Plus 로그인 과정을 제대로 수행했는지 확인해보기 바랍니다.

또는 작성한 코드에 오타가 있어도 그림 10.13과 같이 오류가 발생합니다. 그림 10.13의 코드를 살펴보면 Dispatch를 수행할 클래스 이름을 'CpUtil.CpCybos'가 아니라 'CpUtil.CyCybos'로 잘못 적어서 오류가 났음을 확인할 수 있습니다.

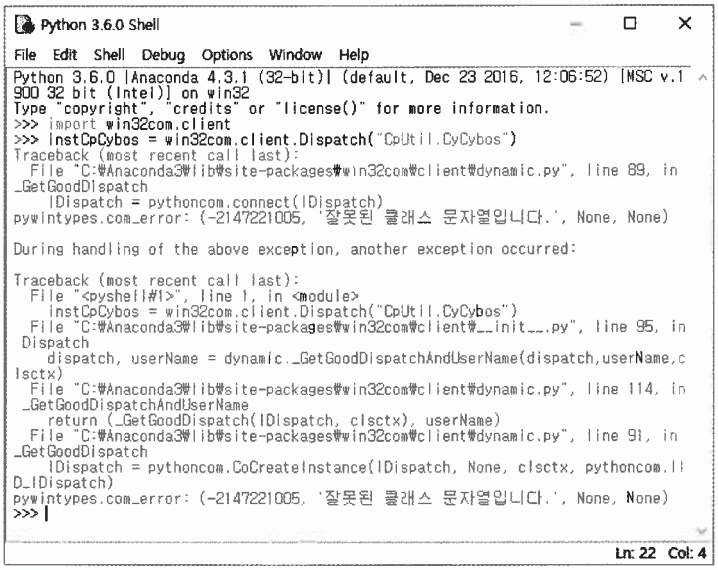


그림 10.13 잘못된 클래스 문자열 오류

CYBOS Plus로 로그인도 했고 그림 10.13과 같은 에러 메시지도 출력되지 않는데 출력값이 0이라면 파이썬 코딩에 사용하는 에디터(Python IDLE 또는 PyCharm)가 관리자 권한으로 실행된 것인지 확인해보기 바랍니다.

출력값이 1로 정상적으로 출력됐다면 예제 10.1의 코드를 살펴보기 바랍니다. 첫 번째 줄은 win32com.client 모듈을 임포트하는 코드입니다. CYBOS Plus는 파이썬으로 구현된 것이 아니므로 COM을 통해 API를 사용해야 합니다. 따라서 가장 먼저 win32com.client 모듈을 임포트했습니다.

win32com.cilent 모듈을 임포트한 후에는 사용하려는 클래스에 대한 인스턴스를 생성합니다. CYBOS Plus는 기능별로 다양한 클래스를 제공합니다. 앞서 내려받은 도움말 파일을 참조하면 CpUtil 항목에 CpCybos라는 클래스가 있음을 확인할 수 있습니다.

그림 10.14와 같이 도움말 파일에서 CpCybos 클래스를 보면 모듈 위치에 CpUtil.dll이라고 표시된 것을 확인할 수 있습니다. 참고로 대신증권에서 사용하는 모듈은 파이썬 모듈과는 관련이 없습니다. 다만 모듈의 위치를 고려해 클래스 인스턴스를 생성할 때 CpUtil.CpCybos라고 입력하면 됩니다.

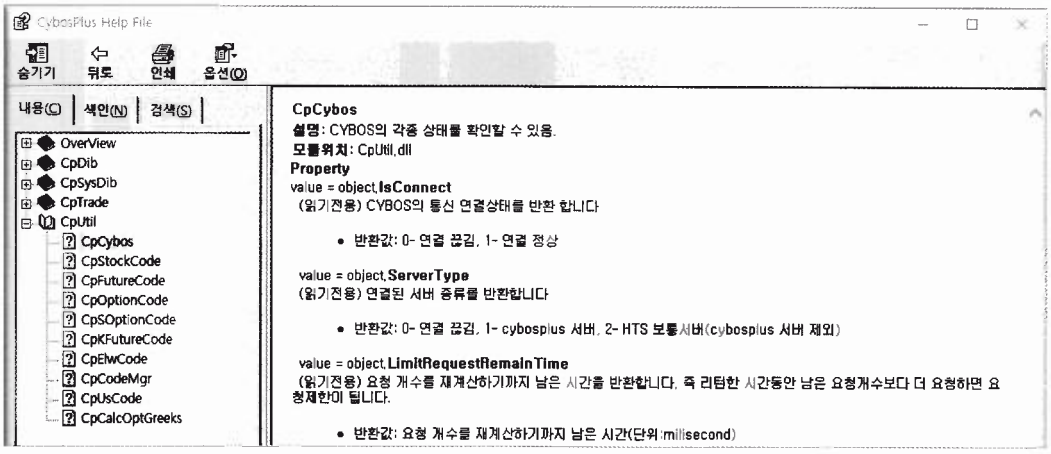


그림 10.14 CYBOS plus 도움말 파일(CpUtil.CpCybos 클래스)

예제 10.1의 두 번째 줄은 "CpUtil.CpCybos" 클래스에 대한 인스턴스를 생성하는 코드입니다. CpUtil.CpCybos 클래스의 인스턴스를 생성한 후 이를 instCpCybos라는 변수로 바인딩한 것입니다. 객체가 생성됐으니 해당 객체를 이용해 클래스 내의 속성 또는 메서드를 사용할 수 있겠지요?

도움말 파일에서 CpCybos 클래스에 대한 속성(Property)이나 메서드(Method) 부분을 확인해보면 IsConnect라는 이름의 속성을 확인할 수 있습니다. 예제 10.1의 세 번째 줄이 바로 instCpCybos라는 객체를 통해 IsConnect 속성값을 확인한 코드입니다. 이 값이 1이면 정상적으로 연결된 것을 의미하고, 0이면 연결이 끊긴 것을 의미합니다.

어떤가요? 증권사에서 제공하는 API라는 것도 파이썬에서 어렵지 않게 사용할 수 있지요?

주식 코드 조회

유가증권시장과 코스닥(KOSDAQ) 시장에는 많은 수의 종목이 상장돼 있습니다. 이뿐만 아니라 2013년 후반부터는 코넥스(KONEX)라고 하는 중소기업 전용 시장도 설립됐습니다. 그렇다면 국내 증권시장에 상장된 주식 종목은 총 몇 개일까요?

CYBOS Plus API를 이용해 이 질문의 답을 찾아보겠습니다. CYBOS Plus 도움말 파일을 참조하면 CpUtil 모듈에 CpStockCode라는 클래스가 있는데, CpStockCode 클래스는 표 10.1과 같은 주요 메서드를 갖고 있습니다. 이 가운데 GetCount라는 메서드의 기능을 살펴보니 종목 코드의 개수를 반환하는 것을 확인할 수 있습니다.

표 10.1 CpStockCode 클래스의 주요 메서드

메서드/속성명	기능	메서드/속성
CodeToName(code)	code에 해당하는 종목명을 반환	메서드
NameToCode(name)	name에 해당하는 종목명을 반환	메서드
CodeToFullCode(code)	code에 해당하는 FullCode를 반환	메서드
FullCodeToName(fullcode)	fullcode에 해당하는 종목명을 반환	메서드
FullCodeToCode(fullcode)	fullcode에 해당하는 Code를 반환	메서드
CodeToIndex(code)	code에 해당하는 Index를 반환	메서드
GetCount()	종목 코드 수를 반환	메서드
GetData(type, index)	해당 인덱스의 종목 데이터를 반환	메서드

CpStockCode 클래스의 GetCount 메서드를 사용하려면 다른 COM 객체와 마찬가지로 win32com.client 모듈을 이용해 CpStockCode 클래스에 대한 인스턴스를 생성한 후 메서드를 호출하면 됩니다. 다음은 GetCount 메서드를 호출하는 파이썬 코드입니다.

```
import win32com.client
instCpStockCode = win32com.client.Dispatch("CpUtil.CpStockCode")
print(instCpStockCode.GetCount())
```

위 코드를 실행해 결괏값을 확인해보니 '2800'이라는 값이 반환됐습니다. 참고로 이 값은 증권 시장에 새롭게 종목이 상장되거나 폐지됨에 따라 달라집니다. 또한 GetCount를 통해 리턴된 값은 시장에 상장된 종목뿐 아니라 주식 시장에는 상장되지 않는 종목도 일부 포함하고 있어 실제 상장 종목 수보다 크게 출력됩니다.

이번에는 2,800개의 종목에 대해 좀 더 구체적으로 확인해보겠습니다. 종목의 수가 2,800개라는 것은 0번부터 2,799번까지의 인덱스에 해당하는 종목이 있다는 의미입니다.

각 인덱스에 해당하는 종목에 대한 데이터를 얻기 위해서는 표 10.1에서 설명한 GetData 메서드를 사용하면 됩니다. GetData 메서드는 두 개의 인자를 받는데, 첫 번째 인자로 0, 1, 2 중 하나의 값을 받습니다. 첫 번째 인자의 값이 0이면 종목 코드를, 1이면 종목명을, 2면 FullCode를 리턴합니다.

예를 들어, 0번 인덱스에 위치하는 종목의 종목명을 확인하려면 다음과 같이 코드를 작성하면 됩니다.

```
import win32com.client
instCpStockCode = win32com.client.Dispatch("CpUtil.CpStockCode")
print(instCpStockCode.GetData(1, 0))
```

이 코드를 실행해보면 0번 인덱스에 해당하는 종목은 '동화약품'임을 확인할 수 있습니다. 만약 다음과 같은 결과가 출력되지 않는다면 CYBOS Plus가 실행돼 있는지, PyCharm과 같은 프로그램이 관리자 모드로 실행된 것인지 확인해 보기 바랍니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/ex03.py
동화약품

Process finished with exit code 0
```

'동화약품'에 대한 종목코드를 확인하려면 GetData 메서드의 첫 번째 인자 값을 1에서 0으로 변경하면 됩니다. 이때 주의할 점은 인자 값이 문자열이 아니고 정숫값이라는 점입니다.

```
import win32com.client
instCpStockCode = win32com.client.Dispatch("CpUtil.CpStockCode")
print(instCpStockCode.GetData(0, 0))

C:\Anaconda3\python.exe E:/GitHub/book/ch10/ex04.py
A000020

Process finished with exit code 0
```

역시나 API를 이용하니 몇 줄 안 되는 코드로 원하는 데이터를 척척 얻을 수 있어 흐뭇합니다. 이번에는 4장에서 배운 반복문을 이용해 0~9 인덱스에 해당하는 종목명을 화면에 출력하는 프로그램을 작성해보겠습니다.

```
import win32com.client
instCpStockCode = win32com.client.Dispatch("CpUtil.CpStockCode")
for i in range(0, 10):
    print(instCpStockCode.GetData(1,i))
```

이 코드를 실행하면 다음과 같이 0~9 인덱스에 해당하는 10개 종목의 종목명이 출력됩니다. 반복문을 사용해 간단히 10개의 종목명을 출력할 수 있었습니다. 어떤가요? 생각보다 쉽죠?

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/ex05.py
동화약품
우리은행
```

```
KR모터스
경방
메리츠화재
삼양홀딩스
삼양홀딩스우
하이트진로
하이트진로2우B
유한양행

Process finished with exit code 0
```

이번에는 프로그래밍 난이도를 좀 더 올려 NAVER라는 종목의 종목 코드를 출력하는 프로그램을 작성해 봅시다. 네이버의 종목명은 영문으로 'NAVER'입니다. 종목의 총 개수가 2,800이었으므로 'NAVER'는 0~2,799의 인덱스 중 어딘가에 있을 것입니다. 그러나 문제는 네이버의 인덱스가 몇 번인지 모른다는 점입니다.

가장 기본적인 해법은 모든 인덱스에 대해 하나씩 값을 확인해보는 것입니다. 일상생활에서도 학교나 회사에서 분실문을 습득했을 때 이를 찾아주는 가장 쉬운 방법은 모든 구성원에게 물어보는 것입니다. 프로그래밍도 마찬가지입니다.

예제 10.2는 반복문을 통해 'NAVER' 종목에 대한 종목 코드, 종목명, 인덱스를 출력하는 코드입니다. 예제 10.2를 살펴보면 GetCount를 통해 얻은 종목 코드의 수만큼 반복문을 수행하면서 반복문 내부에서 종목명이 'NAVER'인지 확인합니다. 만약 특정 인덱스에 해당하는 종목명이 'NAVER'라면 해당 인덱스를 이용해 종목 코드와 종목명을 출력합니다.

예제 10.2 NAVER의 종목 코드, 종목명, 인덱스 출력(소스코드: book/ch10/06.py)

```
import win32com.client
instCpStockCode = win32com.client.Dispatch("CpUtil.CpStockCode")
stockNum = instCpStockCode.GetCount()

for i in range(stockNum):
    if instCpStockCode.GetData(1, i) == 'NAVER':
        print(instCpStockCode.GetData(0,i))
        print(instCpStockCode.GetData(1,i))
        print(i)
```

예제 10.2의 실행 결과를 확인해보면 네이버의 종목 코드는 'A035420'이고 인덱스는 899입니다. 참고로 네이버의 인덱스는 종목이 추가로 상장되거나 폐지됨에 따라 변동될 수 있습니다. 따라서 이 책을 읽는 시점에 네이버의 인덱스 값은 다음 결과와 다를 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/06(ex02).py
A035420
NAVER
899

Process finished with exit code 0
```

사실 CpStockCode 클래스에는 종목명을 이용해 종목 코드를 구하는 NameToCode 메서드가 있습니다. 그리고 종목 코드를 이용해 인덱스를 구하는 CodeToIndex 메서드도 있습니다. 즉, 예제 10.2의 코드와 같이 반복문을 사용하지 않고도 종목명을 정확히 알고 있다면 다음과 같이 구현할 수도 있습니다.

```
import win32com.client
instCpStockCode = win32com.client.Dispatch("CpUtil.CpStockCode")

naverCode = instCpStockCode.NameToCode('NAVER')
naverIndex = instCpStockCode.CodeToIndex(naverCode)
print(naverCode)
print(naverIndex)
```

다음과 같이 'A035420'과 899가 정상적으로 출력된 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/07.py
A035420
899

Process finished with exit code 0
```

03 기본 API 익히기

종목 코드 가져오기

유가증권, 코스닥, 코넥스 시장에 상장된 종목은 모두 고윳값인 종목 코드를 갖고 있습니다. 예를 들어, 그림 10.15는 한국 거래소 웹페이지를 통해 유가증권 시장에 상장된 종목을 조회한 화면입니다. 각 종목에 6자리 숫자로 된 종목 코드가 있음을 확인할 수 있습니다.



▸ 시장구분 ◉ 전체 ◉ 유가증권 ◉ 코스닥 ◉ 코넥스

▸ 종목검색 ▸ 정렬방법 기업명 ▾ ◉ 상세검색

전체상장기업 768개사 전체상장종목 890종목 검색결과 768개사

(2015/12/16 오후 9:07:38)

종목코드	기업명	업종코드	업종	상장주식수 (주)	자본금	액면가	통화
095570	AJ네트웍스	126903	산업용 기계 및 장비 임대업	9,364,459	46,822,295,000	5,000	원
068400	AJ렌터카	126901	운송장비 임대업	22,146,300	11,073,150,000	500	원
006840	AK홀딩스	137105	회사본부, 지주회사 및 경영컨설팅	13,247,561	66,237,805,000	5,000	원
027410	BGF리테일	074701	종합 소매업	24,773,964	24,773,964,000	1,000	
138930	BNK금융지주	116409	기타 금융업	255,935,246	1,279,676,230,	5,000	원
001460	BYC	031401	봉제의복 제조업	624,615	4,200,000,000	5,000	원
001040	CJ	137105	회사본부, 지주회사 및 경영컨설팅	29,133,748	157,698,560,000	5,000	원
079160	CJ CGV	105901	영화, 비디오물, 방송프로그램 제작	21,161,313	10,580,656,500	500	원
000120	CJ대한통운	084903	도로 화물 운송업	22,812,344	114,061,720,000	5,000	원
011150	CJ씨푸드	031007	기타 식품 제조업	35,930,773	18,065,386,500	500	원
097950	CJ제일제당	031007	기타 식품 제조업	13,158,198	72,428,155,000	5,000	원
037560	CJ헬로비전	106002	텔레비전 방송업	77,446,865	193,617,162,500	2,500	원

그림 10.15 유가증권 상장 종목 예시

증권사가 제공하는 API를 보면 일반적으로 종목 코드를 통해 특정 종목에 대한 시세를 조회하거나 매수/매도 주문을 넣을 수 있습니다. 따라서 API를 사용해 각 종목에 대한 종목 코드를 구하는 것은 가장 기본적인 작업이라 할 수 있습니다.

이번 절에서는 유가증권시장에 상장된 종목의 종목 코드와 종목명을 파이썬의 딕셔너리 자료구조에 저장하는 프로그램을 작성해보겠습니다.

그림 10.16과 같이 CybosPlus 도움말 파일에서 검색 탭을 이용해 CpCodeMgr 클래스를 찾습니다. CpCodeMgr 클래스는 각종 코드 정보 및 코드 목록을 구하는 데 사용합니다.

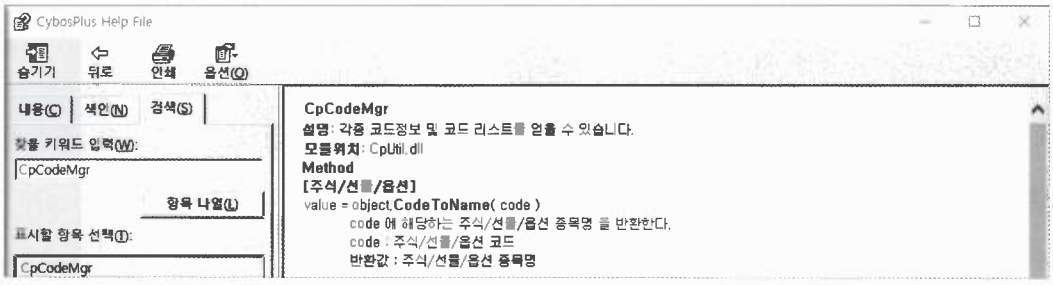


그림 10.16 CpCodeMgr 클래스

CpCodeMgr 클래스에서는 여러 메서드를 제공하는데 그중 GetStockListByMarket 메서드는 시장 구분에 따라 주식 종목을 리스트 형태로 제공합니다. 해당 메서드를 호출하기 위해 먼저 win32com.client 모듈을 임포트하고 CpCodeMgr 클래스의 객체를 생성합니다.

```
import win32com.client
instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
```

instCpCodeMgr 변수가 생성된 객체를 바인딩하고 있으므로 앞으로 instCpCodeMgr를 통해 GetStockListByMarket 메서드를 호출하면 됩니다. 이때 GetStockListByMarket 메서드의 인자로 1을 전달하면 유가증권시장의 종목을 파이썬 튜플 형태로 반환받을 수 있습니다.

```
codeList = instCpCodeMgr.GetStockListByMarket(1)
print(codeList)
```

이번에는 반복문을 통해 각 종목의 종목명을 구하고, 종목 코드를 키(key)로, 종목 이름을 값(value)으로 사전에 추가해봅시다. 다음 코드를 보면 종목 코드로부터 종목명을 구할 때는 CpCodeMgr 클래스의 CodeToName 메서드를 사용했습니다. 참고로 CpStockCode 클래스에도 CodeToName 메서드가 있었습니다.

```
import win32com.client

instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
codeList = instCpCodeMgr.GetStockListByMarket(1)

kospi = {}
for code in codeList:
    name = instCpCodeMgr.CodeToName(code)
    kospi[code] = name
```

이번에는 위 코드를 통해 생성한 kospi 딕셔너리를 CSV(Comma Separated Values) 파일 포맷으로 저장해 보겠습니다. 파이썬 딕셔너리는 프로그래밍할 때 유용한 자료구조지만 사람이 직접 데이터를 확인할 때는 CSV 파일로 저장한 후 엑셀로 보는 것이 편리합니다.

```
import win32com.client

instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
codeList = instCpCodeMgr.GetStockListByMarket(1)

kospi = {}
for code in codeList:
```

```
    name = instCpCodeMgr.CodeToName(code)
    kospi[code] = name

f = open('c:\\Users\\Jason\\Desktop\\kospi.csv', 'w')
for key, value in kospi.items():
    f.write("%s,%s\n" % (key, value))
f.close()
```

위 코드에는 kospi 딕셔너리를 순회하면서 key, value 쌍을 파일로 저장하는 코드가 추가됐습니다. 키/값 쌍을 파일로 저장할 때 중간에 쉼표(,)를 넣고 파일의 확장자를 .csv로 지정하면 그림 10.17처럼 마이크로소프트 엑셀에서 바로 읽을 수 있습니다.

참고로 kopi.csv 파일의 경로는 PC 환경에 따라 적절히 수정해야 합니다. 또한 윈도우는 경로에서 디렉터리(폴더)를 구분할 때 \\를 사용해야 한다는 점에 주의해야 합니다.

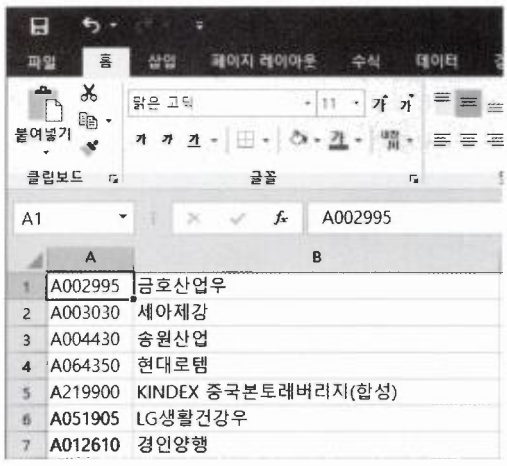


그림 10.17 데이터를 CSV 파일로 저장

GetStockListByMarket 메서드를 통해 구한 주식 코드 리스트에는 ETF(Exchange Traded Fund)와 ETN(Exchange Traded Note)도 포함돼 있습니다. 만약 ETF나 ETN 종목은 제외하고 순수하게 유가증권시장에 상장된 종목에 대한 코드를 구하려면 어떻게 해야 할까요?

CpCodeMgr 클래스를 살펴보면 GetStockSectionKind 메서드가 있는데, 이 메서드를 이용하면 어떤 종목이 주권인지 ETF인지 또는 ETN인지 확인할 수 있습니다.

다음은 유가증권시장의 종목에 대해 인덱스, 종목 코드, 부 구분 코드, 종목명을 출력하는 코드입니다. CpCodeMgr 클래스의 GetStockSectionKind 메서드에 인자로 종목 코드를 전달하면 부 구분 코드를 반환받을 수 있습니다. CybosPlus 도움말 파일의 CpCodeMgr 클래스를 참조하면 1번이 주권, 10번이 ETF, 17번이 ETN임을 알 수 있습니다.

```
import win32com.client

instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
codeList = instCpCodeMgr.GetStockListByMarket(1)

for i, code in enumerate(codeList):
    secondCode = instCpCodeMgr.GetStockSectionKind(code)
    name = instCpCodeMgr.CodeToName(code)
    print(i, code, secondCode, name)
```

위 코드의 실행 결과를 살펴보면 일부 종목은 ETF이고 A900050과 A900140은 외국 주권이며 Q500001~Q500003은 ETN임을 확인할 수 있습니다.

```
1073 A229200 10 KODEX 코스닥 150
1074 A229720 10 KODEX KTOP30
1075 A230480 10 KOSEF 미국달러선물 인버스2X(합성)
1076 A232080 10 TIGER 코스닥150
1077 A232590 10 KINDEX 골드선물 인버스2X(합성 H)
1078 A900050 13 중국원양자원
1079 A900140 13 코라오홀딩스
1080 Q500001 17 신한 K200 USD 선물 바이셀 ETN
1081 Q500002 17 신한 USD K200 선물 바이셀 ETN
1082 Q500003 17 신한 인버스 WTI원유 선물 ETN(H)
```

과거 데이터 구하기

알고리즘 트레이딩 시스템을 만들 때 중요한 일 중 하나가 바로 과거 데이터를 확보하는 것입니다. 예를 들어, 거래량 분석을 통해 유망한 투자 종목을 뽑는 프로그램을 개발한다면 먼저 각 종목에 대한 일정 기간의 거래량 데이터가 있어야 할 것입니다.

이러한 과거 데이터를 구하는 가장 효과적인 방법 중 하나가 바로 증권사에서 제공하는 API를 사용하는 것입니다. 이번 절에서는 API를 이용해 주식 종목의 시가, 고가, 저가, 종가, 거래량 등의 데이터를 가져오는 방법을 배우겠습니다.

대신증권의 CYBOS Plus 매뉴얼을 참조하면 과거 데이터를 조회하는 데 StockChart 클래스를 사용할 수 있음을 알 수 있습니다. 참고로 StockChart 클래스는 그림 10.18과 같이 CpSysDib 모듈에 위치합니다.

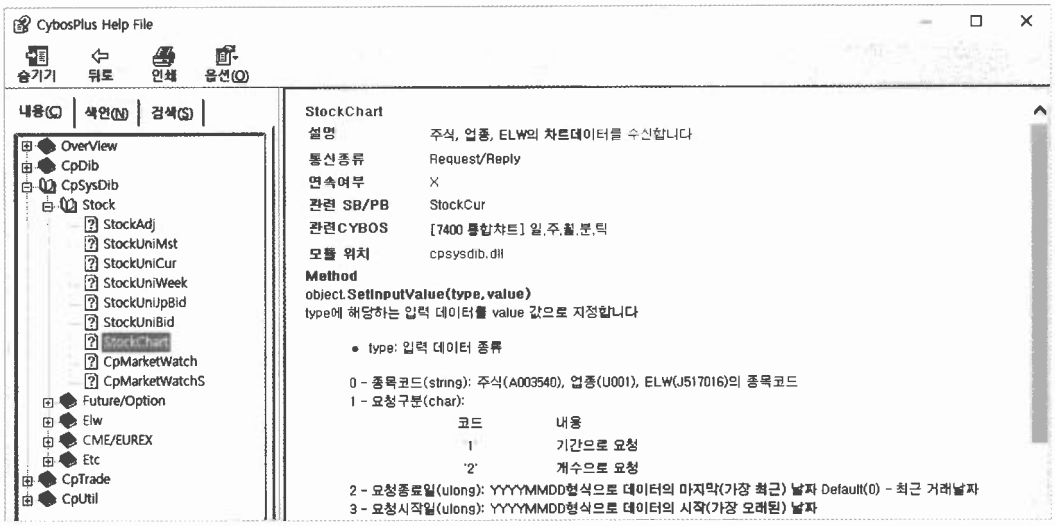


그림 10.18 StockChart 클래스

여러 종목의 시가, 고가, 저가, 종가, 거래량을 가져오는 프로그램을 처음부터 바로 구현하는 것은 어려울 수 있습니다. 먼저 API 사용법을 알아보기 위해 대신증권 종목에 대한 최근 10일간 종가 데이터를 구하는 코드를 작성해보겠습니다.

CYBOS Plus의 StockChart 클래스를 사용하려면 먼저 StockChart 클래스의 인스턴스를 생성해야 합니다.

```
import win32com.client
instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")
```

StockChart 클래스는 앞에서 배운 클래스에 비해 사용하기가 조금 복잡합니다. 그림 10.18에 있는 StockChart 클래스의 설명을 참조하면 통신 종류가 Request/Reply임을 알 수 있습니다. 여기서 Request/Reply 방식이란 사용자가 어떤 정보를 얻고자 할 때 얻으려는 정보가 무엇인지를 정확히 알려준 후 요청(Request)하면 CYBOS Plus가 이에 대한 답을 돌려주는(Reply) 방식을 의미합니다.

계속해서 코드를 완성해 보겠습니다. 앞서 instStockChart 객체를 생성했으므로 해당 객체를 통해 메서드를 호출하면 됩니다. 맨 먼저 할 일은 어떤 데이터를 원하는지 정확히 알려주는 일입니다. 이때 사용하는 메서드가 바로 SetInputValue입니다.

```
instStockChart.SetInputValue(0, "A003540")
instStockChart.SetInputValue(1, ord('2'))
instStockChart.SetInputValue(4, 10)
instStockChart.SetInputValue(5, 5)
```

```
instStockChart.SetInputValue(6, ord('D'))
instStockChart.SetInputValue(9, ord('1'))
```

코드를 보면 총 6번에 걸쳐 SetInputValue 메서드를 호출했습니다. SetInputValue 메서드의 첫 번째 인자는 입력 데이터 타입을 의미하고 두 번째 인자는 입력 데이터 값을 의미합니다.

예를 들어, SetInputValue(0, "A003540")에서 0은 종목 코드를 의미하고, 'A003540'은 조회하려는 종목의 코드값입니다. 앞서 대신증권 종목의 최근 10일 종가를 확인하기로 했는데 대신증권의 종목 코드가 바로 'A003540'입니다.

조회할 종목 코드에 대한 정보를 입력했으면 다음으로 조회할 기간을 입력합니다. StockChart 클래스는 종목에 대한 과거 데이터를 기간 또는 개수로 조회할 수 있습니다. 예제에서는 최근 10일간의 종가만 얻기 위해 개수로 지정했습니다. 기간으로 요청할 때는 '1'을 입력하고, 개수로 요청할 때는 '2'를 입력합니다. 다만 파이썬의 문자열을 그대로 입력하는 것이 아니라 ord라는 함수를 통해 문자열 값을 아스키 코드로 변환해야 합니다.

세 번째로 입력한 데이터는 요청 개수입니다. SetInputValue(4, 10)에서 4는 요청 개수라는 타입을 의미하고 10이 실제로 요청할 데이터의 개수를 의미합니다. 10은 최근 거래일로부터 10일 치에 해당하는 데이터를 의미합니다.

네 번째로 입력한 데이터는 요청할 데이터의 종류입니다. CYBOS Plus 도움말에 있는 StockChart 부분을 참조하면 종가에 해당하는 값이 5임을 알 수 있습니다.

다섯 번째로 입력한 데이터는 차트의 종류로서 일 단위의 데이터를 가져오기 위해 ord('D')를 입력했습니다. 여섯 번째로 입력한 데이터는 수정 주가의 반영 여부에 대한 것으로 수정 주가를 의미하는 ord('1')을 입력했습니다.

이처럼 조회하려는 종목과 조회 조건에 대한 정보를 SetInputValue 메서드를 통해 입력하고 나면 데이터 처리를 요청하면 됩니다.

```
instStockChart.BlockRequest()
```

Request/Reply 방식에서는 사용자가 BlockRequest 메서드 호출을 통해 서버에 데이터를 요청(request)하면 서버는 해당 데이터를 준비한 후 사용자에게 응답(reply)합니다. 서버로부터 데이터를 얻어올 때는 GetHeaderValue와 GetDataValue라는 메서드를 사용합니다.

먼저 GetHeaderValue 메서드를 통해 수신한 데이터의 개수를 확인합니다. 앞서 최근 10일 치에 대한 데이터를 요청했으므로 아마도 numData의 값은 10일 것입니다.

```
numData = instStockChart.GetHeaderValue(3)
```

numData만큼의 데이터를 가져오기 위해 간단히 for 문을 작성하고 해당 for 문 내에서 GetDataValue 메서드를 호출해 10일 치 데이터를 화면에 출력하면 됩니다.

```
for i in range(numData):
    print(instStockChart.GetDataValue(0, i))
```

GetDataValue 메서드의 첫 번째 인자는 요청한 데이터의 인덱스를 의미합니다. 앞서 일 단위로 종가 하나만 요청했으므로 이 값으로 0을 입력하면 됩니다. 만약 일 단위로 시가, 고가, 저가, 종가, 거래량을 얻어오는 경우라면 시가를 얻을 때는 0, 고가를 얻을 때는 1을 사용하면 됩니다. GetDataValue 메서드의 두 번째 인자는 for 문의 인덱스 값을 그대로 사용하면 됩니다.

지금까지 작성한 코드는 다음과 같습니다.

```
import win32com.client
instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")

instStockChart.SetInputValue(0, "A003540")
instStockChart.SetInputValue(1, ord('2'))
instStockChart.SetInputValue(4, 10)
instStockChart.SetInputValue(5, 5)
instStockChart.SetInputValue(6, ord('D'))
instStockChart.SetInputValue(9, ord('1'))

instStockChart.BlockRequest()

numData = instStockChart.GetHeaderValue(3)
for i in range(numData):
    print(instStockChart.GetDataValue(0, i))
```

PyCharm에서 위 코드를 실행하면 다음과 같이 최근 10일간의 종가가 출력됩니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/11.py
10500
10550
10650
10700
10600
10850
10850
10850
```

```
10850
10700
```

프로그램 실행 결과를 그림 10.19와 비교해보면 API를 통해 정확히 종가를 얻어왔음을 확인할 수 있습니다. 참고로 그림 10.19는 저자가 Daum 증권 페이지를 통해 대신증권의 최근 10일간의 종가를 캡처한 것입니다.

일자별 주가 일봉차트 자세히▸

일자	시가	고가	저가	종가	전일비	등락률	거래량
16.11.01	10,500	10,650	10,450	10,500	▼50	-0.47%	81,180
16.10.31	10,600	10,650	10,450	10,550	▼100	-0.94%	141,131
16.10.28	10,750	10,800	10,550	10,650	▼50	-0.47%	62,901
16.10.27	10,650	10,750	10,550	10,700	▲100	+0.94%	77,954
16.10.26	10,950	10,950	10,550	10,600	▼250	-2.30%	173,509
16.10.25	10,900	10,950	10,700	10,850	- 0	0.00%	160,825
16.10.24	10,850	10,950	10,800	10,850	- 0	0.00%	80,856
16.10.21	10,800	10,900	10,700	10,850	- 0	0.00%	209,439
16.10.20	10,750	10,900	10,650	10,850	▲150	+1.40%	137,679
16.10.19	10,650	10,750	10,600	10,700	▲50	+0.47%	123,217

그림 10.19 대신증권의 최근 10일 치 종가(16/11/01 기준)

그림 10.20은 CYBOS Plus의 Request/Reply 타입의 객체를 사용하는 절차를 그림으로 나타낸 것입니다. 맨 먼저 `win32com.client` 모듈의 `Dispatch` 함수를 사용해 객체를 생성합니다. 그런 다음 생성된 객체의 `SetInputValue` 메서드를 호출해 요청할 데이터에 대한 정보를 입력합니다. 데이터 입력이 끝나면 서버에 데이터 처리를 요청하는 `BlockRequest` 메서드를 호출합니다. 서버로부터 데이터 처리에 대한 응답이 오면 `GetDataValue` 메서드를 이용해 데이터를 가져옵니다.

CYBOS Plus에는 수많은 클래스가 있는데, 그중 많은 클래스가 Request/Reply 방식으로 동작합니다. 따라서 그림 10.20의 절차를 잘 기억해 둔다면 Request/Reply 방식의 다른 클래스도 손쉽게 사용할 수 있을 것입니다.

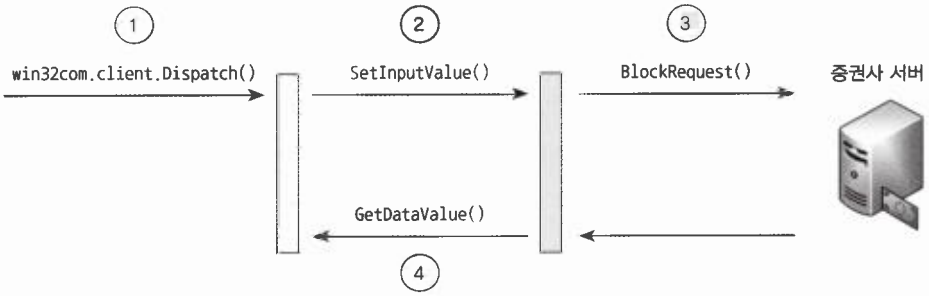


그림 10.20 Request/Reply 객체의 사용

지금까지 StockChart 클래스의 기본적인 사용법을 익혔습니다. 이번에는 조금 더 프로그램을 확장해 보겠습니다. 먼저 대신증권 종목에 대해 일자별로 시가, 고가, 저가, 종가, 거래량을 가져오는 프로그램을 구현해 보겠습니다.

표 10.2 StockChart 클래스 요약

타입(type)	입력 데이터의 종류	값(value)
0	종목 코드	요청할 종목의 종목 코드
1	요청 구분	'1': 기간으로 요청, '2': 개수로 요청
2	요청종료일	YYYYMMDD 형식
3	요청시작일	YYYYMMDD 형식
4	요청개수	요청할 데이터의 개수
5	필드	0: 날짜, 1: 시간, 2: 시가, 3: 고가, 4: 저가, 5: 종가, 6: 전일대비, 8: 거래량, 9: 거래대금, 10: 누적체결매도수량
6	차트구분	'D': 일, 'W': 주, 'M': 월, 'm': 분, 'T': 틱
9	수정주가	'0': 무수정주가, '1': 수정주가

표 10.2에 StockChart 클래스에 대한 정보를 요약했습니다. 표 10.2 또는 CYBOS Plus의 도움말 파일을 참조하면 시가, 고가, 저가, 종가, 거래량을 가져오기 위해서는 5번 type의 필드 값을 조정하면 된다는 것을 확인할 수 있습니다.

예를 들어, 일자별로 시가, 고가, 저가, 종가 거래량을 구하기 위해서는 2, 3, 4, 5, 8이라는 값을 SetInputValue의 두 번째 인자로 전달하면 됩니다. 이때 해당 값을 파이썬의 리스트나 튜플 형태로 전달하면 됩니다. 각 일자 데이터도 함께 출력하려면 (0, 2, 3, 4, 5, 8)과 같이 0을 추가하면 됩니다.

```
instStockChart.SetInputValue(5, (0, 2, 3, 4, 5, 8))
```

각 일자에 대해 6개의 데이터(일자, 시가, 고가, 저가, 종가, 거래량)를 요청했기 때문에 먼저 총 며칠 날에 해당하는 데이터가 반환됐는지와 일자별로 몇 개의 데이터가 반환됐는지 체크해야 합니다. 물론 SetInputValue를 통해 10일 치 데이터와 일별로 6개의 데이터를 요청했지만 반환된 데이터가 요청한 개수와 맞는지 확인하는 작업이 필요합니다.

```
numData = instStockChart.GetHeaderValue(3)
numField = instStockChart.GetHeaderValue(1)
```

일자별로 6개의 데이터가 반환되기 때문에 중첩 for 문을 사용해 데이터를 출력하도록 코드를 수정하면 됩니다.

```
for i in range(numData):
    for j in range(numField):
        print(instStockChart.GetDataValue(j, i), end=" ")
    print("")
```

수정된 전체 코드는 예제 10.3과 같습니다.

예제 10.3 일별로 일자, 시가, 고가, 저가, 종가, 거래량을 출력하는 코드(소스코드: book/ch10/12.py)

```
import win32com.client

# Create object
instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")

# SetInputValue
instStockChart.SetInputValue(0, "A003540")
instStockChart.SetInputValue(1, ord('2'))
instStockChart.SetInputValue(4, 10)
instStockChart.SetInputValue(5, (0, 2, 3, 4, 5, 8))
instStockChart.SetInputValue(6, ord('D'))
instStockChart.SetInputValue(9, ord('1'))

# BlockRequest
instStockChart.BlockRequest()

# GetHeaderValue
numData = instStockChart.GetHeaderValue(3)
numField = instStockChart.GetHeaderValue(1)

# GetDataValue
```

```
for i in range(numData):
    for j in range(numField):
        print(instStockChart.GetDataValue(j, i), end=" ")
    print("")
```

예제 10.3을 실행하면 다음과 같이 각 일자와 각 일자별 시가, 고가, 저가, 종가, 거래량이 출력됩니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/12.py
20161102 10500 10500 10500 10500 0
20161101 10500 10650 10450 10500 81180
20161031 10600 10650 10450 10550 141131
20161028 10750 10800 10550 10650 62901
20161027 10650 10750 10550 10700 77954
20161026 10950 10950 10550 10600 173509
20161025 10900 10950 10700 10850 160825
20161024 10850 10950 10800 10850 80856
20161021 10800 10900 10700 10850 209439
20161020 10750 10900 10650 10850 137679
```

이번에는 개수가 아니라 기간으로 데이터를 요청하도록 수정해보겠습니다. 기간 데이터를 요청하려면 SetInputValue 메서드를 이용해 타입 1, 타입 2, 타입 3의 데이터를 설정하면 됩니다. 먼저 타입 1에 '기간으로 요청'을 의미하는 '1'을 인자로 전달합니다. 이때 ASCII 코드값을 전달하기 위해서는 ord 함수를 사용해야 합니다.

기간으로 데이터를 요청했기 때문에 요청할 기간의 시작일과 종료일에 대한 정보를 타입 2와 타입 3에 설정합니다. 예를 들어, 2016년 10월 20일 ~ 2016년 10월 31일 동안의 데이터를 요청하려면 다음과 같이 SetInputValue 메서드를 호출하는 부분을 수정하면 됩니다.

```
instStockChart.SetInputValue(1, ord('1'))
instStockChart.SetInputValue(2, 20161031)
instStockChart.SetInputValue(3, 20161020)
```

변경된 코드를 실행하면 이번에는 요청한 기간에 대한 데이터만 화면에 출력됩니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/13.py
20161031 10600 10650 10450 10550 141131
20161028 10750 10800 10550 10650 62901
20161027 10650 10750 10550 10700 77954
```

```
20161026 10950 10950 10550 10600 173509
20161025 10900 10950 10700 10850 160825
20161024 10850 10950 10800 10850 80856
20161021 10800 10900 10700 10850 209439
20161020 10750 10900 10650 10850 137679
```

PER, EPS 데이터 구하기

PER(Price Earning Ratio, 주가 이익 비율)은 주가를 주당순이익(EPS)으로 나눈 값으로서 주가의 수익성 지표로 자주 활용됩니다. 예를 들어, A사의 주가가 5만 원이고 1주당 순이익이 1만 원이면 PER은 5(배)가 됩니다. 높은 PER은 기업이 영업활동으로 벌어들인 이익에 비해 주가가 고평가되고 있음을 의미하고, 반대로 낮은 PER은 주가가 상대적으로 저평가되고 있음을 의미합니다.

물론 단순히 PER 수치만을 가지고 주가가 고평가 또는 저평가됐다고 판단하기는 어렵습니다. 예를 들어, 기업의 성장성에 높은 점수를 주는 경우에는 상대적으로 PER이 높은 수치를 보입니다. 또한 업종에 따라서도 PER을 평가하는 기준이 다를 수 있습니다. 예를 들어, 의료 분야의 종목들은 성장성이 높으므로 제조 업종의 PER보다 높게 형성돼 있습니다.

이번 절에서는 종목별로 PER과 EPS 데이터를 구하는 프로그램을 작성해보겠습니다. 먼저 CYBOS Plus 도움말 파일의 검색 탭을 이용해 'per'이라는 키워드로 검색하면 그림 10.21과 같이 MarketEye 클래스에서 PER을 지원하는 것을 확인할 수 있습니다.

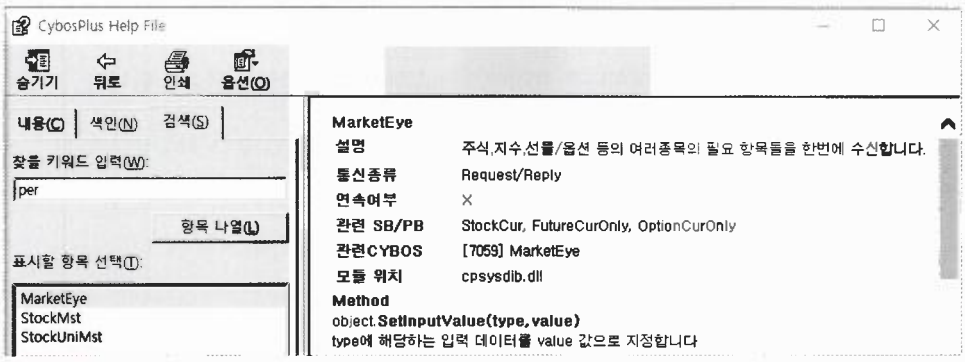


그림 10.21 PER 지원 클래스 검색

MarketEye 클래스의 통신 종류를 살펴보면 역시 Request/Reply 방식임을 확인할 수 있습니다. 따라서 그림 10.20과 같은 방식으로 MarketEye 클래스를 사용하면 됩니다. 먼저 MarketEye 클래스에 대한 인스턴스를 생성합니다.

```
import win32com.client

# 객체 생성
instMarketEye = win32com.client.Dispatch("CpSysDib.MarketEye")
```

SetInputValue 메서드를 이용해 타입 0에 요청하고자 하는 필드 값을 설정합니다. 예를 들어, 현재가, PER, EPS, 최근분기년월 데이터를 요청하려면 (4, 67, 70, 111)을 입력하면 됩니다. 더 자세한 내용은 CYBOS Plus 도움말 파일을 참조하기 바랍니다.

타입 1에는 조회할 종목에 대한 종목 코드를 지정하면 됩니다. 따라서 대신증권 종목에 대해 현재가, PER, EPS, 최근분기년월을 구하려면 다음과 같이 코드를 작성하면 됩니다.

```
# SetInputValue
instMarketEye.SetInputValue(0, (4, 67, 70, 111))
instMarketEye.SetInputValue(1, 'A003540')
```

요청할 데이터에 대한 설정이 완료됐으므로 BlockRequest 메서드를 호출합니다.

```
# BlockRequest
instMarketEye.BlockRequest()
```

마지막으로 GetDataValue 메서드를 이용해 서버로부터 반환된 데이터를 출력하면 됩니다. 이때 GetHeaderValue 메서드를 통해 반환되는 필드의 개수나 종목 개수를 확인할 수도 있지만 이미 알고 있다면 바로 GetDataValue 메서드를 사용해도 됩니다.

GetDataValue 메서드에서 첫 번째 인자는 필드에 대한 인덱스이고 두 번째 인자는 종목의 인덱스입니다. 지금은 대신증권 종목 하나에 대한 데이터를 요청했으므로 두 번째 인자의 값은 0이 됩니다.

앞서 필드 항목으로 현재가, PER, EPS, 최근분기연월을 요청했으므로 첫 번째 인자로 각각 0, 1, 2, 3을 지정하면 됩니다.

```
# GetData
print("현재가: ", instMarketEye.GetDataValue(0, 0))
print("PER: ", instMarketEye.GetDataValue(1, 0))
print("EPS: ", instMarketEye.GetDataValue(2, 0))
print("최근분기년월: ", instMarketEye.GetDataValue(3, 0))
```

예제 10.4는 지금까지 단계별로 설명해 드린 코드를 모은 것입니다.

예제 10.4 현재가, PER, EPS, 최근분기연월을 출력하는 코드(MarketEye)(소스코드: book/ch10/14.py)

```
import win32com.client

# Create Object
instMarketEye = win32com.client.Dispatch("CpSysDib.MarketEye")

# SetInputValue
instMarketEye.SetInputValue(0, (4, 67, 70, 111))
instMarketEye.SetInputValue(1, 'A003540')

# BlockRequest
instMarketEye.BlockRequest()

# GetData
print("현재가: ", instMarketEye.GetDataValue(0, 0))
print("PER: ", instMarketEye.GetDataValue(1, 0))
print("EPS: ", instMarketEye.GetDataValue(2, 0))
print("최근분기년월: ", instMarketEye.GetDataValue(3, 0))
```

예제 10.4를 실행하면 현재가, PER, EPS, 최근분기연월이 화면에 출력됩니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/14.py
현재가:  12350
PER:  8.98
EPS:  1375
최근분기년월:  201612
```

04 API를 이용한 알고리즘 개발

지금까지 대신증권 CYBOS Plus에서 제공하는 몇 가지 클래스의 기본 사용법을 배웠습니다. 이번 절에서는 이를 이용해 간단한 알고리즘을 개발하는 과정을 보여 드리겠습니다.

거래량 분석을 통한 대박 주 포착

시중에 출판된 차트 분석 책을 보면 단골손님으로 소개되는 주제가 바로 거래량 분석을 통한 대박 주 포착 알고리즘입니다. 그러한 책에서 주장하는 주요 요점은 '거래량은 조작하기 어려운 데이터이기 때문에 실제 세력들의 매집 패턴을 거래량을 통해 알 수 있다'는 것입니다. 알고리즘은 책마다 조금씩 차이가 있지만 대략 아래의 조건을 만족하는 종목을 찾으면 됩니다.

1. 대량 거래(거래량이 1,000% 이상 급증) 종목
2. 대량 거래 시점에서 PBR이 4보다 작아야 함

대박 주의 첫 번째 조건은 특정일의 거래량이 이전의 평균 거래량보다 10배 이상 증가한 경우입니다. 예를 들어, 최근 60일간의 평균 거래량이 30,000주 정도인 종목이 있는데 금일 거래량이 300,000주라면 이 조건을 만족하는 것입니다.

CYBOS Plus에서 거래량을 구하려면 앞에서 설명한 StockChart 클래스를 이용하면 됩니다. 다음은 대신증권 종목의 최근 60일 치 거래량을 volumes라는 리스트에 저장하는 코드입니다. 참고로 최근 거래일의 거래량이 리스트 앞쪽에 위치합니다. 다음 코드를 실행한 후 출력 값을 다음 증권이나 네이버 증권에서 제공하는 대신증권 종목 거래량과 비교해 보기 바랍니다.

```
import win32com.client

# Create object
instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")

# SetInputValue
instStockChart.SetInputValue(0, "A003540")
instStockChart.SetInputValue(1, ord('2'))
instStockChart.SetInputValue(4, 60)
instStockChart.SetInputValue(5, 8)
instStockChart.SetInputValue(6, ord('D'))
instStockChart.SetInputValue(9, ord('1'))

# BlockRequest
instStockChart.BlockRequest()

# GetData
volumes = []
```

```
numData = instStockChart.GetHeaderValue(3)
for i in range(numData):
    volume = instStockChart.GetDataValue(0, i)
    volumes.append(volume)
print(volumes)
```

이번에는 앞에서 구한 데이터를 이용해 거래량이 1,000% 급증했는지 확인하는 코드를 추가해보겠습니다. 앞의 코드에서 volumes라는 리스트에는 60일 치의 거래량이 저장됩니다. volumes[0]의 데이터가 최근 거래일의 거래량이므로 이를 제외한 나머지 59개의 거래량의 평균을 계산하겠습니다.

먼저 60일 치 거래량을 모두 더한 후 최근 거래량을 빼서 59일 치 거래량의 합을 계산합니다. 그리고 이 값을 59로 나눠서 평균 거래량을 계산합니다.

```
averageVolume = (sum(volumes) - volumes[0]) / (len(volumes) -1)
```

평균 거래량을 계산했으므로 이 값을 최근 거래일의 거래량인 volumes[0] 값과 비교하면 됩니다. 최근 거래일의 거래량이 평균 거래량의 10배 이상이면 화면에 '대박 주'를 출력하고, 그렇지 않다면 '일반 주'를 출력한 후 최근 거래일의 거래량과 평균 거래량의 비율을 출력했습니다.

```
if(volumes[0] > averageVolume * 10):
    print("대박 주")
else:
    print("일반 주", volumes[0] / averageVolume)
```

지금까지 구현된 대박 주 찾기 알고리즘의 전체 코드는 예제 10.5와 같습니다.

예제 10.5 한 종목에 대한 대박 주 찾기 코드(소스코드: book/ch10/16.py)

```
import win32com.client

# Create object
instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")

# SetInputValue
instStockChart.SetInputValue(0, "A003540")
instStockChart.SetInputValue(1, ord('2'))
instStockChart.SetInputValue(4, 60)
instStockChart.SetInputValue(5, 8)
instStockChart.SetInputValue(6, ord('D'))
instStockChart.SetInputValue(9, ord('1'))
```

```
# BlockRequest
instStockChart.BlockRequest()

# GetData
volumes = []
numData = instStockChart.GetHeaderValue(3)
for i in range(numData):
    volume = instStockChart.GetDataValue(0, i)
    volumes.append(volume)

# Calculate average volume
averageVolume = (sum(volumes) - volumes[0]) / (len(volumes) -1)

if(volumes[0] > averageVolume * 10):
    print("대박주")
else:
    print("일반주", volumes[0] / averageVolume)
```

예제 10.5의 실행 결과를 살펴보면 대신증권은 대박 주가 아님을 확인할 수 있습니다. 참고로 여러분이 이 코드를 실행하는 시점에 따라 본문과 다른 결과가 나올 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/16.py
일반주 1.890187807085377
```

이번에는 예제 10.5를 활용해 유가증권시장의 전 종목 중 거래량이 1,000% 이상 급증한 종목을 찾는 프로그램을 만들어 보겠습니다. 먼저 예제 10.5를 기반으로 한 종목에 대해 거래량이 급증했는지 점검하는 함수를 구현합니다. 다음 코드에서 CheckVolumn 함수를 보면 함수의 내부 코드가 예제 10.5와 거의 유사하다는 점을 알 수 있습니다.

```
import win32com.client

def CheckVolumn(instStockChart, code):
    # SetInputValue
    instStockChart.SetInputValue(0, code)
    instStockChart.SetInputValue(1, ord('2'))
    instStockChart.SetInputValue(4, 60)
    instStockChart.SetInputValue(5, 8)
    instStockChart.SetInputValue(6, ord('D'))
    instStockChart.SetInputValue(9, ord('1'))
```

```
    # BlockRequest
    instStockChart.BlockRequest()

    # GetData
    volumes = []
    numData = instStockChart.GetHeaderValue(3)
    for i in range(numData):
        volume = instStockChart.GetDataValue(0, i)
        volumes.append(volume)

    # Calculate average volume
    averageVolume = (sum(volumes) - volumes[0]) / (len(volumes) -1)

    if(volumes[0] > averageVolume * 10):
        return 1
    else:
        return 0
```

함수를 작성할 때는 함수의 인자를 어떻게 구성할지가 중요합니다. CheckVolumn 함수는 함수 내부에서 StockChart 인스턴스를 생성하지 않고 함수를 호출하는 곳에서 인스턴스를 생성하는 구조입니다. 만약 함수 내에서 인스턴스를 생성하도록 구현한다면 함수가 호출될 때마다 인스턴스가 생성됐다가 함수가 종료되면 인스턴스도 소멸해야 하므로 프로그램이 느려집니다.

유가증권시장의 전 종목에 대해 거래량 급증 여부를 체크하려면 반복문을 사용해 각 종목에 대해 CheckVolumn 함수를 호출하면 됩니다. 이를 위해 먼저 유가증권시장의 전 종목에 대한 코드 목록을 만들어야 합니다. 참고로 유가증권시장의 전 종목에 대한 코드 목록을 구하는 방법은 10.3.1절에서 다뤘습니다.

```
instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
codeList = instCpCodeMgr.GetStockListByMarket(1)
```

유가증권시장의 전 종목에 대한 종목 코드 리스트를 구했다면 반복문에서 각 종목 코드별로 CheckVolumn 함수를 호출하면 됩니다. 이때 거래량이 1,000% 이상 급증한 종목은 따로 코드 값을 결과 리스트에 추가합니다. CheckVolumn 함수의 첫 번째 인자로 StockChart 클래스에 대한 인스턴스가 사용되므로 미리 해당 인스턴스를 생성해야 합니다.

```
instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")
buyList = []
for code in codeList:
```

```
    if CheckVolumn(instStockChart, code) == 1:
        buyList.append(code)
```

유가증권시장의 전 종목에 대해 거래량 급증 여부를 확인하는 프로그램의 전체 코드는 다음과 같습니다.

```
import win32com.client

def CheckVolumn(instStockChart, code):
    # SetInputValue
    instStockChart.SetInputValue(0, code)
    instStockChart.SetInputValue(1, ord('2'))
    instStockChart.SetInputValue(4, 60)
    instStockChart.SetInputValue(5, 8)
    instStockChart.SetInputValue(6, ord('D'))
    instStockChart.SetInputValue(9, ord('1'))

    # BlockRequest
    instStockChart.BlockRequest()

    # GetData
    volumes = []
    numData = instStockChart.GetHeaderValue(3)
    for i in range(numData):
        volume = instStockChart.GetDataValue(0, i)
        volumes.append(volume)

    # Calculate average volume
    averageVolume = (sum(volumes) - volumes[0]) / (len(volumes) -1)

    if(volumes[0] > averageVolume * 10):
        return 1
    else:
        return 0

if __name__ == "__main__":
    instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")
    instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
    codeList = instCpCodeMgr.GetStockListByMarket(1)
    buyList = []
    for code in codeList:
        if CheckVolumn(instStockChart, code) == 1:
```

```
            buyList.append(code)
            print(code)
```

코드를 실행하면 다음과 같이 유가증권시장의 종목 중 거래량이 1,000% 이상 급증한 종목의 코드를 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/17.py
A000040
A003350
A008040
A011760
A020120
A090370
A114460
A114820
A196220
A227540
A227840
```

참고로 위 코드를 실행했을 때 그림 10.22와 같은 경고 메시지가 나타날 수 있습니다. 이는 유가증권시장의 전 종목에 대한 60일 치 거래량을 빠른 시간 동안 요청했기 때문입니다. 프로그래밍을 통해 데이터를 요청할 경우 사용자가 HTS로 데이터를 요청하는 것보다 짧은 시간 동안 많은 데이터를 요청할 수 있습니다. 사용자 입장에서는 API를 이용하는 것이 효율적이지만 증권사 서버는 이를 처리하는 데 부담이 됩니다. 이러한 문제 때문에 보통 증권사에서는 회원의 계좌 등급에 따라 데이터를 처리하는 데 제한을 두곤 합니다.

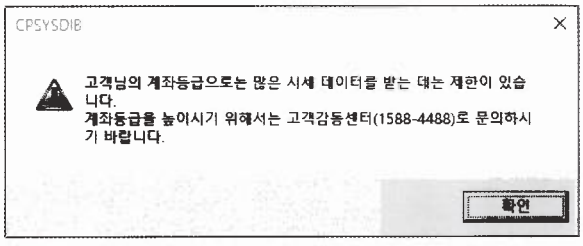


그림 10.22 CYBOS Plus 경고 메시지

이 같은 문제가 발생하는 경우 가장 쉬운 해결책은 데이터를 요청할 때 각 요청 시점마다 강제로 약간의 지연 시간을 두는 것입니다. 파이썬의 time 모듈에는 sleep이라는 함수가 있는데 이 함수를 이용하면 일정 시간 동안 멈췄다가 다시 코드를 실행하게 할 수 있습니다.

다음은 sleep(1)을 사용해 한 종목에 대한 데이터를 조회하고 1초 동안 멈춘 후에 다음 종목에 대한 데이터를 조회하도록 코드를 수정한 것입니다. 다만 이 경우 프로그램의 수행 시간이 길어진다는 단점이 있습니다.

```
import time

# 중간 코드 생략

for code in codeList:
    if CheckVolumn(instStockChart, code) == 1:
        buyList.append(code)
        print(code)
    time.sleep(1)
```

예제 10.6은 유가증권시장의 종목에 대해 거래량 급증 여부를 확인하는 최종 코드입니다.

예제 10.6 유가증권시장 전 종목에 대한 거래량 급증 확인 코드(소스코드: book/ch10/18.py)

```
import win32com.client
import time

def CheckVolumn(instStockChart, code):
    # SetInputValue
    instStockChart.SetInputValue(0, code)
    instStockChart.SetInputValue(1, ord('2'))
    instStockChart.SetInputValue(4, 60)
    instStockChart.SetInputValue(5, 8)
    instStockChart.SetInputValue(6, ord('D'))
    instStockChart.SetInputValue(9, ord('1'))

    # BlockRequest
    instStockChart.BlockRequest()

    # GetData
    volumes = []
    numData = instStockChart.GetHeaderValue(3)
    for i in range(numData):
        volume = instStockChart.GetDataValue(0, i)
        volumes.append(volume)

    # Calculate average volume
```

```
    averageVolume = (sum(volumes) - volumes[0]) / (len(volumes) -1)

    if(volumes[0] > averageVolume * 10):
        return 1
    else:
        return 0

if __name__ == "__main__":
    instStockChart = win32com.client.Dispatch("CpSysDib.StockChart")
    instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
    codeList = instCpCodeMgr.GetStockListByMarket(1)

    buyList = []
    for code in codeList:
        if CheckVolumn(instStockChart, code) == 1:
            buyList.append(code)
            print(code)
        time.sleep(1)
```

사실 지금까지 작성한 프로그램으로 알아낸 거래량 급증 종목은 HTS를 통해서도 쉽게 알 수 있습니다. 그러나 HTS에서는 제공하지 않는 여러분만의 조건을 추가하기 위해서는 직접 프로그래밍할 필요가 있습니다. 바로 이럴 때 프로그래밍을 통한 알고리즘 트레이딩이 빛을 발합니다.

업종별 PER 분석을 통한 유망 종목 찾기

PER 지표는 주식 초보자에게도 익숙한 대표적인 투자 지표 중 하나입니다. 단순히 PER 값이 10 이하이면 '싸다'라고 판단하거나 20 이상이면 '비싸다'라고 판단하면 안 된다는 것도 이미 알고 있을 것입니다.

이러한 PER 지표가 가진 문제를 보완하는 여러 가지 알고리즘이 있습니다. 가장 간단한 알고리즘 중 하나는 종목이 속하는 업종의 평균 PER을 구한 후 평균값보다 낮다면 상대적으로 저평가돼 있다고 판단하고, 평균보다 높다면 고평가돼 있다고 판단할 수 있을 것입니다. 물론 이 알고리즘에도 오류가 있습니다.

예를 들어, 업종별로도 우량 기업과 비우량 기업이 섞여 있는데 단순히 업종에 속하는 모든 기업의 평균값을 이용하는 것은 PER 값을 왜곡할 수 있습니다. 따라서 투자하려는 종목과 비슷한 규모의 회사 위주로 평가 그룹을 정하고 해당 그룹에 대한 평균 PER 값을 분석하는 것이 더 좋은 알고리즘이라 할 수 있습니다.

더 좋은 알고리즘의 개발은 여러분에게 맡기기로 하고 이번 절에서는 업종별 PER 분석을 통한 유망 종목을 찾는 알고리즘을 구현하는 과정을 설명하겠습니다. 먼저 업종별로 평균 PER을 계산하려면 업종별로 종목 코드 리스트를 만들어야 합니다.

CYBOS Plus의 도움말 파일을 참조하면 CpCodeMgr 클래스의 GetIndustryList 메서드를 사용하면 업종별 코드 리스트를 구할 수 있습니다. 먼저 CpCodeMgr 클래스의 인스턴스를 생성해 봅시다.

```
import win32com.client

instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
```

업종별 코드 리스트를 구하기 위해 GetIndustryList 메서드를 호출합니다. GetIndustryList 메서드는 업종별 코드 리스트를 파이썬 튜플 형태로 반환합니다.

```
industryCodeList = instCpCodeMgr.GetIndustryList()
```

각 업종별 코드에 해당하는 업종명을 출력해보면 어떤 업종이 있는지 더 정확히 파악할 수 있습니다. 업종별 코드에 해당하는 업종명을 구하려면 GetIndustryName 메서드를 사용하면 됩니다. 이때 industryCodeList에는 많은 수의 업종별 코드가 존재하므로 반복문을 사용해 업종별 코드에 대한 업종명을 출력하면 됩니다.

```
for industryCode in industryCodeList:
    print(industryCode, instCpCodeMgr.GetIndustryName(industryCode))
```

지금까지 작성한 코드는 다음과 같습니다.

```
import win32com.client

instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
industryCodeList = instCpCodeMgr.GetIndustryList()

for industryCode in industryCodeList:
    print(industryCode, instCpCodeMgr.GetIndustryName(industryCode))
```

이 코드를 실행해 유가증권시장의 업종별 코드와 이에 대응하는 업종명을 확인해보기 바랍니다. 출력 결과를 살펴보면 음식료품, 화학, 의약품과 같은 업종뿐 아니라 다양한 지수 업종도 함께 출력되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/19.py
001 종합주가지수
002 대형(시가총액)
003 중형(시가총액)
```

```
004 소형(시가총액)
005 음식료품
006 섬유,의복
007 종이,목재
008 화학
009 의약품
010 비금속광물
...
```

유가증권시장에서 '음식료품' 업종에 속하는 종목의 종목 코드 리스트를 구한 후 해당 리스트에 속하는 종목의 종목명을 출력해 봅시다. CYBOS Plus 도움말을 참조하면 GetGroupCodeList 메서드가 업종 코드에 해당하는 종목 코드 리스트를 반환하는 것을 알 수 있습니다.

예를 들어, 다음은 유가증권시장의 '음식료품' 업종에 속하는 종목 코드 리스트를 구한 후 각 종목의 종목 코드와 종목명을 출력하는 코드입니다.

```
import win32com.client

instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
tarketCodeList = instCpCodeMgr.GetGroupCodeList(5)

for code in tarketCodeList:
    print(code, instCpCodeMgr.CodeToName(code))
```

이 코드를 실행하면 다음과 같이 '음식료품' 업종에 속하는 종목 코드와 종목명이 출력됩니다(결과 중 일부만 출력). 참고로 종목명을 구하기 위해 CodeToName 메서드를 사용했습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch10/20.py
A011150 CJ씨푸드
A011155 CJ씨푸드1우
A097950 CJ제일제당
A097955 CJ제일제당 우
A023150 MH에탄올
A002140 고려산업
A003920 남양유업
A003925 남양유업우
A004370 농심
A001680 대상
```

이번에는 10.3.3절에서 다룬 내용을 바탕으로 '음식료품' 업종의 평균 PER을 계산해보겠습니다. 지금까지 작성한 코드를 살펴보면 CpCodeMgr 클래스의 인스턴스를 사용해 유가증권시장에서 '음식료품' 업종에 해당하는 종목의 종목 코드 리스트를 구했습니다.

각 종목의 PER 값은 CpSysDib 모듈의 MarketEye 클래스를 사용해 얻을 수 있었습니다. 벌써 기억이 잘 나지 않는 분들은 10.3.3절의 내용을 참고합니다. MarketEye 클래스를 사용하기 위해 먼저 인스턴스를 생성합니다.

```
instMarketEye = win32com.client.Dispatch("CpSysDib.MarketEye")
```

MarketEye 클래스는 Request/Reply 방식을 사용하므로 먼저 SetInputValue 메서드를 사용해 요청할 정보를 설정해야 합니다. 각 종목의 PER 값만 구하면 되므로 다음과 같이 코드를 작성하면 됩니다.

```
# Get PER
instMarketEye.SetInputValue(0, 67)
instMarketEye.SetInputValue(1, tarketCodeList)
```

10.3.3절에서는 한 종목의 PER 값만 구했지만 여러 종목의 PER을 한 번에 구하고 싶을 때는 위와 같이 종목 코드 리스트를 사용하면 됩니다. SetInputValue 메서드를 사용해 요청할 데이터를 설정했으므로 BlockRequest 메서드를 호출합니다.

```
# BlockRequest
instMarketEye.BlockRequest()
```

BlockRequest 메서드가 정상적으로 호출되면 GetDataValue 메서드를 통해 서버가 반환한 데이터를 가져올 수 있습니다. 먼저 반환된 종목의 수를 확인하기 위해 GetHeaderValue 메서드를 호출합니다.

```
# GetHeaderValue
numStock = instMarketEye.GetHeaderValue(2)
```

numStock이라는 변수에 PER 데이터를 구해온 종목의 수가 저장돼 있으므로 종목 수만큼 반복문을 실행하면 각 종목의 PER 값을 sumPer이라는 변수에 누적합니다. 그리고 누적된 sumPer 값을 종목 수로 나누면 유가증권시장의 '음식료품' 업종에 대한 평균 PER 값을 계산할 수 있습니다.

```
# GetData
sumPer = 0
for i in range(numStock):
    sumPer += instMarketEye.GetDataValue(0, i)

print("Average PER: ", sumPer / numStock)
```

지금까지 작성한 코드는 예제 10.7과 같습니다. 중간에 흐름을 놓치신 분은 전체 코드를 참조해서 코드의 흐름을 이해해 보시기 바랍니다.

예제 10.7 유가증권시장의 음식료품 업종에 대한 평균 PER 계산(소스코드: book/ch10/21.py)

```
import win32com.client

instCpCodeMgr = win32com.client.Dispatch("CpUtil.CpCodeMgr")
instMarketEye = win32com.client.Dispatch("CpSysDib.MarketEye")

tarketCodeList = instCpCodeMgr.GetGroupCodeList(5)

# Get PER
instMarketEye.SetInputValue(0, 67)
instMarketEye.SetInputValue(1, tarketCodeList)

# BlockRequest
instMarketEye.BlockRequest()

# GetHeaderValue
numStock = instMarketEye.GetHeaderValue(2)

# GetData
sumPer = 0
for i in range(numStock):
    sumPer += instMarketEye.GetDataValue(0, i)

print("Average PER: ", sumPer / numStock)
```

05 매수/매도 API 익히기

이번 절에서는 CYBOS Plus의 API를 이용해 주식 매수/매도 주문을 넣는 방법을 알아보겠습니다. 주식 매수/매도에 대한 API를 이용하는 것은 지금까지 다룬 API와 달리 금전적 손실을 줄 수 있으므로 반드시 모의 투자를 통해 충분히 관련 API를 숙지한 후 사용하길 권장합니다.

모의투자 가입하기

대신증권은 이용자들이 실전투자에 앞서 가상으로 투자 및 매매를 해볼 수 있도록 모의투자 시스템을 제공합니다. 알고리즘 트레이딩을 하는 여러분도 모의투자를 통해 안전하게 알고리즘을 검증해 볼 수 있으며, 매수/매도 API도 자유롭게 테스트해볼 수 있습니다.

대신증권의 모의투자를 사용하려면 대신증권 홈페이지를 통해 모의투자에 추가로 가입해야 합니다. 먼저 그림 10.23과 같이 대신증권 홈페이지에서 모의투자 메뉴를 선택합니다.



그림 10.23 대신증권의 모의투자 메뉴

모의투자를 사용하려면 먼저 모의투자 전용 회원가입을 해야 합니다. 그림 10.24와 같이 [모의투자 회원가입] 메뉴를 클릭해 모의투자를 위한 회원가입을 진행합니다. 모의투자를 위한 회원 가입 시 기존에 실전 투자를 위해 생성한 통신 ID와 통신비밀번호와 혼동되지 않도록 ID와 비밀번호를 만들어야 한다는 점에 주의합니다. 만일 모의투자인지 알고 매수/매도 주문을 넣었는데, 실제 계좌에서 매수/매도가 일어난 것이라면 손해가 엄청나겠지요?



그림 10.24 모의투자 시작하기

모의투자용 통신 ID와 통신비밀번호를 만들었다면 해당 ID와 통신비밀번호를 이용해 모의투자 사이트에 로그인합니다. 모의투자 사이트에 정상적으로 로그인했다면 [모의투자 대회참가] 메뉴를 선택해 대회 참가를 신청합니다. 대회참가에서는 그림 10.25와 같이 상시 모의 투자 약관에 동의한 후 참가 신청을 하면 됩니다.

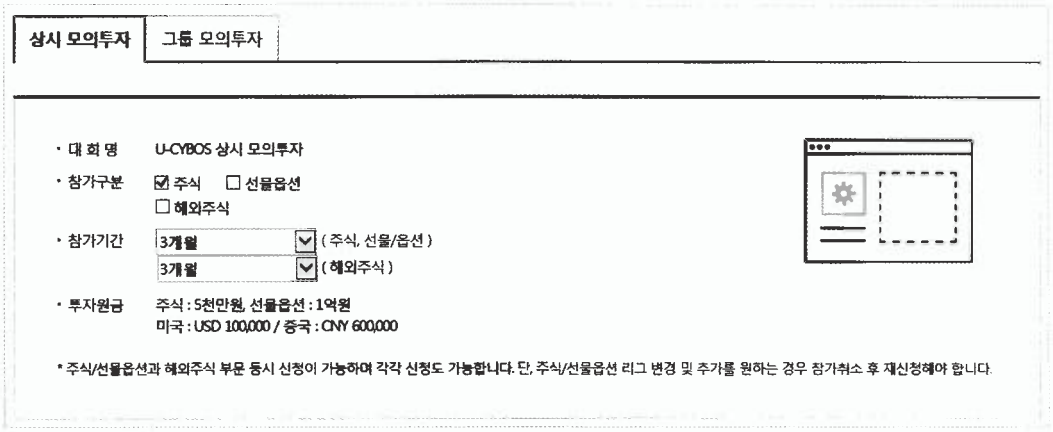


그림 10.25 모의투자 참가 신청 메뉴

매수/매도하기

모의투자용 계정을 만들고 모의투자 참가 신청이 완료되면 그림 10.26과 같이 CYBOS Plus를 실행한 후 먼저 상단의 [모의투자] 메뉴를 클릭합니다. 이처럼 모의투자 메뉴를 클릭해야 모의투자 계정으로 로그인할 수 있습니다.



그림 10.26 모의투자 메뉴 선택

CYBOS Plus 탭을 클릭한 후 모의투자용 계정 정보를 입력해 로그인하면 그림 10.27과 같이 모의투자 대회를 선택하는 메뉴가 나타납니다. 앞서 신청한 모의투자 대회가 선택된 것을 확인한 후 [모의투자 접속]을 클릭해 모의투자 시스템으로 접속합니다.

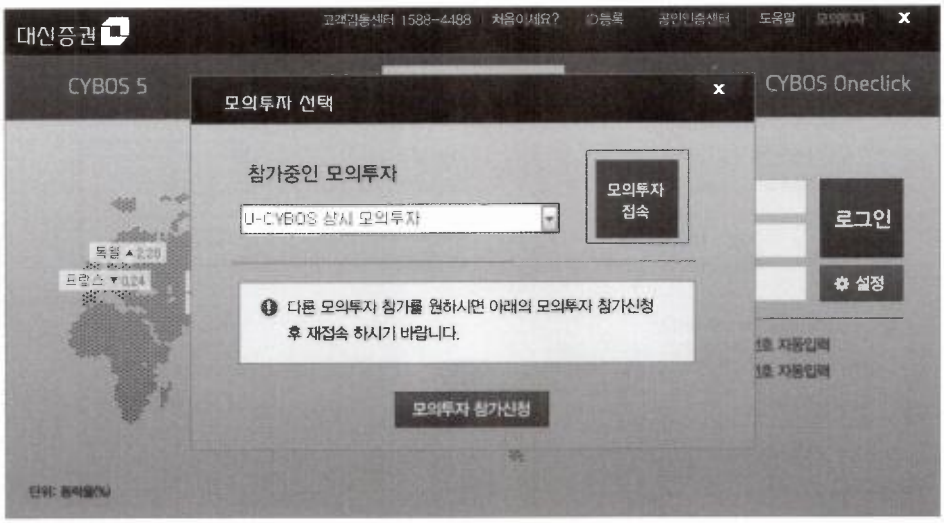


그림 10.27 모의투자 선택

정상적으로 모의투자 시스템에 접속되면 그림 10.28과 같이 윈도우 오른쪽 하단에 CYBOS Plus 실행 아이콘이 나타납니다.



그림 10.28 CYBOS Plus 실행 아이콘

CYBOS Plus 도움말 파일을 참조하면 매수/매도 주문을 위한 클래스는 CpTrade 모듈의 CpTd0311입니다. 다만 주문과 관련된 객체를 사용하기에 앞서 먼저 기본적으로 필요한 초기화 과정을 수행해야 하는데, 이를 위해서는 CpTrade 모듈의 CpTdUtill 클래스를 사용해야 합니다.

그림 10.29는 CpTdUtil과 CpTd0311 클래스의 메서드를 호출하는 순서를 나타냅니다. 앞서 설명한 것처럼 CpTdUtil 클래스의 TradeInit 메서드를 호출해 먼저 초기화 과정을 수행해야 합니다.

초기화가 정상적으로 이뤄지면 CpTd0311 클래스를 사용하면 됩니다. 해당 클래스는 Request/Reply 타입이므로 먼저 매수/매도할 정보를 SetInputValue 메서드를 통해 설정한 후 BlockRequest 메서드를 호출해 매수/매도를 서버에 요청하면 됩니다.

CpTd0311은 Request/Reply 기반의 통신 방식을 사용하지만 GetDataValue 메서드는 사용할 수 없습니다. 따라서 매수/매도 주문에 대한 체결 내역은 GetDataValue 메서드를 통해 구하는 것이 아니라 CpDib 모듈의 CpConclusion 클래스를 통해 따로 구해와야 합니다.

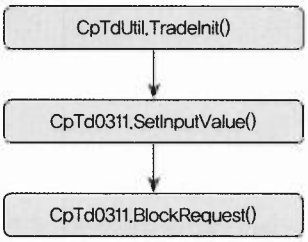


그림 10.29 주문 관련 API 호출 순서

본격적으로 API를 통해 가상계좌에서 매수를 수행해보겠습니다. 이를 위해 먼저 PyCharm과 같은 에디터를 관리자 모드로 실행해야 합니다. 참고로 매수/매도 API를 이용하려면 주식 시장의 운영 시간에 코드를 실행해야 합니다. 대신증권의 모의투자는 가상이긴 하지만 실제 시장의 운영 시간과 같은 시간에만 운영되기 때문입니다.

CpTrade 모듈의 CpTdUtil과 CpTd0311 클래스를 사용하기 위해 해당 클래스의 인스턴스를 먼저 생성합니다.

```
import win32com.client

instCpTdUtil = win32com.client.Dispatch("CpTrade.CpTdUtil")
instCpTd0311 = win32com.client.Dispatch("CpTrade.CpTd0311")
```

instCpTdUtil 객체가 정상적으로 생성되면 해당 객체의 TradeInit 메서드를 호출해 주문을 위한 초기화를 수행합니다.

```
instCpTdUtil.TradeInit()
```

TradeInit 메서드가 호출되면 화면에 그림 10.31과 같이 주문확인 설정 대화상자가 나타납니다. 이때 정상적으로 계좌 비밀번호를 입력해야 앞으로 CpTd0311과 같은 주문 객체를 사용할 수 있습니다. 대신증권 모의투자는 계좌비밀번호로 1234를 사용하므로 계좌비밀번호에 1234를 입력한 후 [확인] 버튼을 누릅니다. 이때 [주문내역 확인]의 체크박스는 체크하지 않습니다.

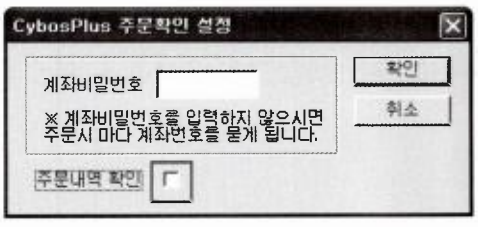


그림 10.30 주문확인 설정 대화상자

TradeInit 메서드를 통해 정상적으로 초기화가 수행되면 CpTd0311 클래스를 이용해 실제 주문을 수행하면 됩니다. 앞서 간단히 설명한 것처럼 CpTd0311 클래스는 Request/Reply 타입이므로 먼저 주문에 관련된 정보를 SetInputValue 메서드로 설정해야 합니다.

표 10.3은 CpTd0311 클래스에서 설정하는 주요 타입 및 입력 데이터의 종류를 나타낸 것입니다. 예를 들어, 매수 주문을 넣고 싶다면 SetInputValue(0, 2)와 같은 방식으로 SetInputValue 메서드를 호출하면 됩니다.

표 10.3 CpTd0311 클래스의 SetInputValue 메서드 타입 및 데이터

타입	입력 데이터의 종류	값
0	주문 종류 코드	1: 매도, 2: 매수
1	계좌 번호	주문을 수행할 계좌 번호
3	종목 코드	주문할 종목의 종목 코드
4	주문 수량	주문 수량
5	주문 단가	주문 단가

모의투자 계좌를 통해 대신증권 종목 10주를 13,000원에 매수하는 주문을 넣어보겠습니다. 이를 위해 먼저 SetInputValue 메서드를 호출해 해당 주문 데이터를 설정해야 합니다. 주문에 필요한 계좌 번호는 CpTdUtil 클래스의 AccountNumber 속성을 통해 구할 수 있습니다. 계좌 번호가 여러 개인 경우 해당 속성은 리스트 형태로 구성되는데, 현재 모의 투자는 한 개의 계좌만 사용 중이므로 AccountNumber[0]을 통해 계좌를 구할 수 있습니다.

```
accountNumber = instCpTdUtil.AccountNumber[0]
instCpTd0311.SetInputValue(0, 2)
instCpTd0311.SetInputValue(1, accountNumber)
```

주문 종류 코드와 계좌 번호를 설정하고 나면 주문할 종목의 종목 코드를 설정합니다. 종목 코드는 타입 3에 설정하면 되며, 대신증권의 종목 코드는 'A003540'입니다.

```
instCpTd0311.SetInputValue(3, 'A003540')
```

타입 4와 타입 5에 주문 수량과 주문 단가를 입력하면 되는데, 이때 중요한 것은 4장의 표 4.2에서 설명한 호가 규칙에 맞게 주문해야 한다는 점입니다.

```
instCpTd0311.SetInputValue(4, 10)
instCpTd0311.SetInputValue(5, 13000)
```

모든 설정이 완료되면 BlockRequest 메서드를 호출해 주문을 처리하도록 증권사 서버에 요청합니다.

```
instCpTd0311.BlockRequest()
```

지금까지 작성한 코드는 다음과 같습니다. 전체적인 코드의 흐름을 다시 한 번 떠올려 보기 바랍니다. 매도 주문을 넣는 것도 다음 코드와 크게 다르지 않기 때문에 여러분이 충분히 구현할 수 있을 것입니다.

```
import win32com.client

instCpTdUtil = win32com.client.Dispatch("CpTrade.CpTdUtil")
instCpTd0311 = win32com.client.Dispatch("CpTrade.CpTd0311")

instCpTdUtil.TradeInit()

accountNumber = instCpTdUtil.AccountNumber[0]
instCpTd0311.SetInputValue(0, 2)
instCpTd0311.SetInputValue(1, accountNumber)
instCpTd0311.SetInputValue(3, 'A003540')
instCpTd0311.SetInputValue(4, 10)
instCpTd0311.SetInputValue(5, 13000)
instCpTd0311.BlockRequest()
```

위 코드를 통해 주식 매수 주문을 실행해도 전송한 주문이 잘 체결됐는지 아직은 확인할 수 없습니다. 앞에서 설명한 것처럼 주식 주문의 체결 내역은 CpConclusion 클래스를 통해 확인할 수 있는데, 해당 클래스는 지금까지 배운 Request/Reply 통신 방식의 클래스와 달리 사용하기가 어렵습니다.

따라서 당분간은 CYBOS(CYBOS Plus 아님)를 실행해 실시간으로 주문 내역이 잘 처리됐는지 확인하겠습니다. CYBOS Plus의 클래스 중 Request/Reply 방식이 아닌 Subscribe/Unsubscribe 방식의 클래스는 이 책의 후반부까지 모두 공부한 후 직접 도전해 보시기 바랍니다.

11

이베스트투자증권 API

이번 장에서는 이베스트투자증권의 API 서비스인 xingAPI를 다룹니다. 이베스트투자증권은 이트레이드증권이 2014년 사명을 변경한 증권사입니다. 이베스트투자증권의 xingAPI는 타 증권사의 API에 비해 거래 수수료가 저렴하다는 장점 덕분에 국내의 많은 알고리즘 트레이더들이 사용하고 있습니다.

01 개발 환경 구축

이베스트투자증권의 xingAPI도 앞서 살펴본 대신증권의 CybosPlus와 비슷한 절차를 통해 개발 환경을 구축해야 합니다. 먼저 가까운 은행에 방문하거나 비대면 계좌개설을 통해 이베스트투자증권사 계좌를 개설합니다. 계좌 개설이 완료되면 이베스트투자증권 홈페이지를 통해 API 사용 등록을 합니다. API 사용 등록을 마친 후에는 PC에 xingAPI 패키지를 설치합니다.

계좌 개설 및 xingAPI 패키지 설치

그림 11.1은 이베스트투자증권 홈페이지(http://www.ebestsec.co.kr/)에 나온 계좌 개설 제휴은행 목록입니다. 이 가운데 한곳에 방문한 후 이베스트투자증권 계좌 개설을 요청하면 됩니다. 참고로 계좌를 개설할 때는 반드시 본인을 증명할 수 있는 신분증을 지참해야 합니다.



○ 계좌개설 제휴은행

* 은행 별 로고를 클릭하시면 해당은행의 계좌개설 관련 사항을 확인하실 수 있습니다.

구분	주식 (*펀드,해외주식)	선물옵션	해외선물(*)
KB	●	●	●
우리은행	●	●	●
KEB 하나은행	●	●	●
신한은행	●	●	●
동부저축은행	●	●	●
IBK	●	●	
대구은행	●	●	●
BNK 부산은행	●	●	
MG	●	●	
citibank	●	●	
Standard Chartered	●	●	
광주은행	●	●	
농협	●		
신협	●		

그림 11.1 이베스트투자증권 계좌 개설 제휴은행

이베스트투자증권 계좌 개설이 완료되면 PC에 xingAPI 패키지라는 프로그램을 설치해야 합니다. 이를 위해 먼저 이베스트투자증권 홈페이지(http://www.ebestsec.co.kr/)로 이동한 후 그림 11.2와 같이 웹 페이지 하단에 있는 API 메뉴를 클릭합니다.

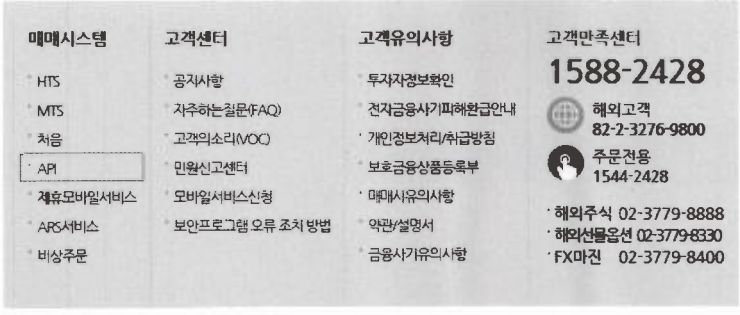


그림 11.2 대신증권 다운로드 센터

xingAPI 패키지는 이름에서 알 수 있듯이 사용자가 xingAPI를 좀 더 쉽게 사용할 수 있도록 여러 가지를 한 묶음으로 만들어 둔 것입니다. xingAPI 패키지에는 xingAPI 라이브러리, API 개발을 위한 보조 도구인 DevCenter, 야간에도 주간시장의 로직을 테스트해볼 수 있는 API 전용 거래소인 xingACE, 여러 가지 지표 데이터를 제공하는 ChartData 및 코드 샘플이 포함돼 있습니다.

xingAPI 패키지를 설치하기 위해 그림 11.3과 같이 [최신버전 설치] 버튼을 클릭해 설치 파일을 다운로드합니다.



그림 11.3 xingAPI 패키지의 최신 버전 다운로드

다운로드한 설치 파일을 더블클릭해서 설치를 시작합니다. 그림 11.4와 같이 설치 마법사 화면이 나타나면 [다음] 버튼을 눌러 설치를 시작합니다.

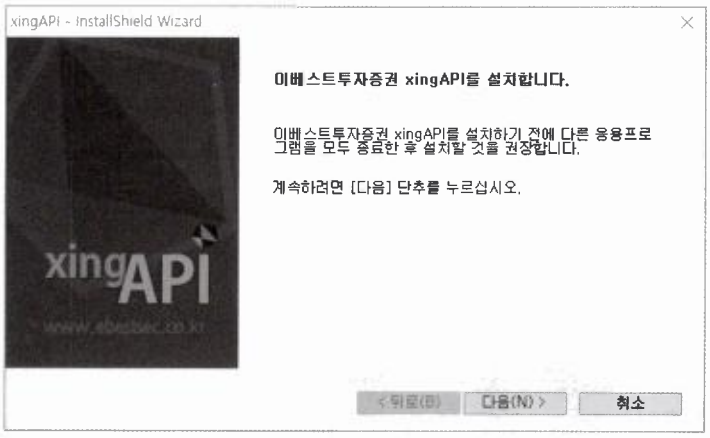


그림 11.4 xingAPI 패키지 설치 1

다음으로 xingAPI 패키지가 설치될 폴더를 설정하는 화면에서는 기본 설치 경로를 수정하지 말고 [다음] 버튼을 클릭합니다.

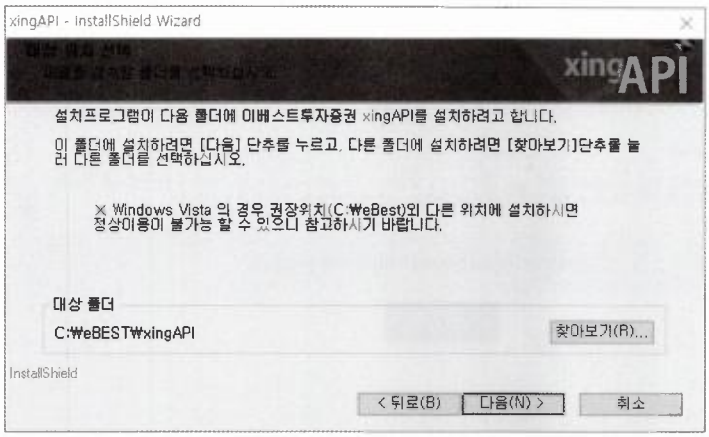


그림 11.5 xingAPI 패키지 설치 2

설치가 완료되면 그림 11.6과 같은 화면이 나타납니다. [완료] 버튼을 눌러 설치 과정을 종료합니다.

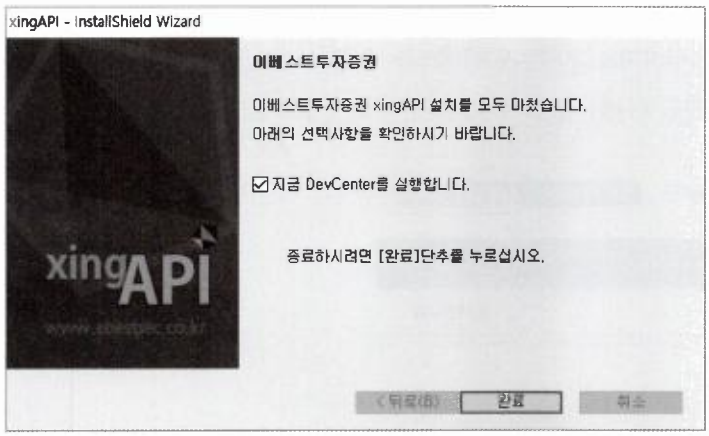


그림 11.6 xingAPI 패키지 설치 3

xingAPI 패키지의 설치가 완료되면 이베스트투자증권 홈페이지에서 ID를 등록합니다. ID 등록을 마친 후 해당 ID로 로그인합니다. xingAPI를 사용하려면 이베스트투자증권 홈페이지를 통해 API 사용 등록을 해야 합니다. 이를 위해 이베스트투자증권 홈페이지에서 [매매시스템] → [API] 메뉴로 이동한 후 그림 11.7과 같이 [사용등록/해지] 메뉴를 클릭합니다. xingAPI 이용약관을 읽어보고 약관에 동의한 후 [사용등록] 버튼을 클릭합니다.

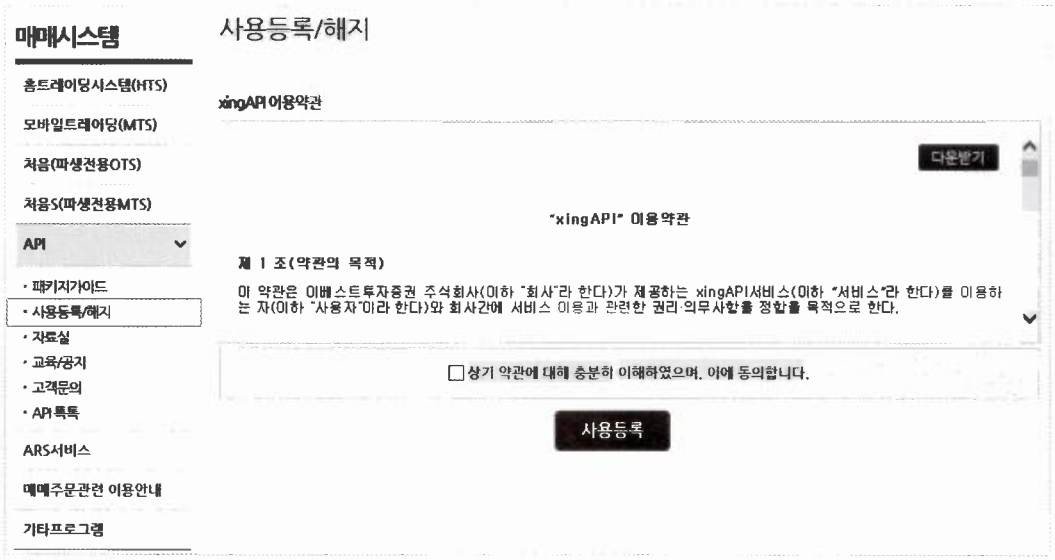


그림 11.7 xingAPI 사용 등록

모의투자 가입하기

이베스트투자증권도 대신증권과 마찬가지로 모의투자 서비스를 제공합니다. 모의투자 서비스를 이용하려면 그림 11.8과 같이 [모의투자신청] 메뉴를 클릭해 모의투자 참가 신청을 해야 합니다.

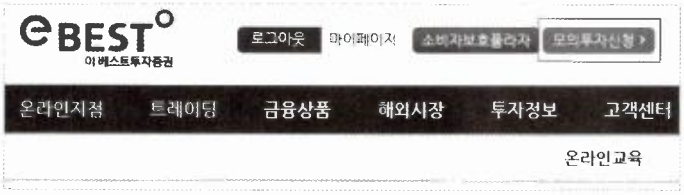


그림 11.8 모의투자 신청

모의투자 서비스는 실제 거래와의 구분을 위해 그림 11.9와 같이 모의투자 전용 비밀번호를 설정할 수 있습니다. 본인이 희망하는 투자원금과 기간을 설정한 후 참가 신청을 하면 됩니다.

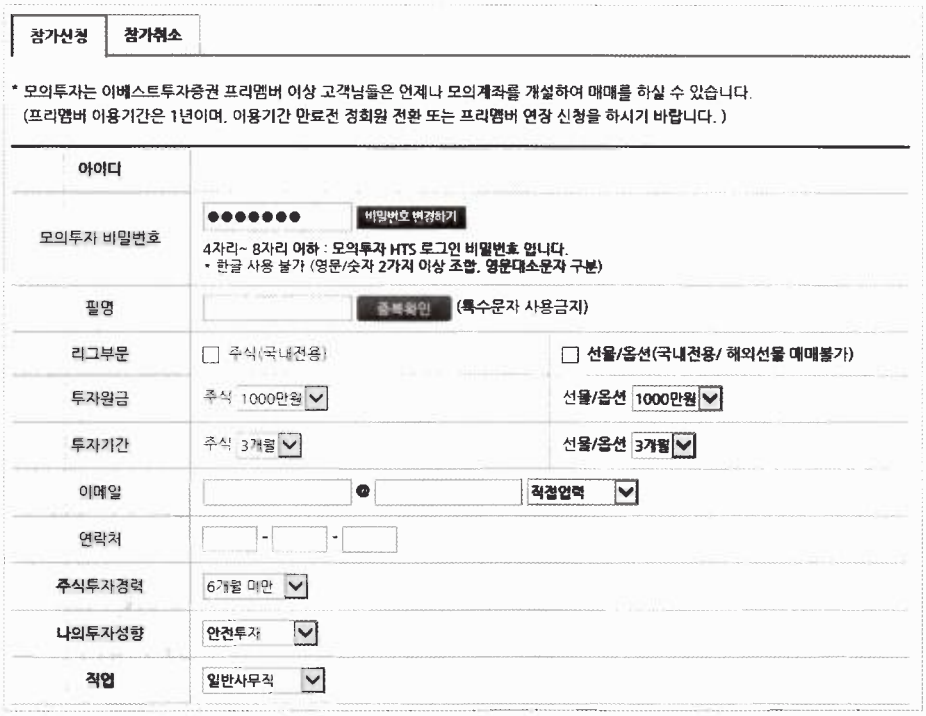


그림 11.9 모의투자 참가 신청 화면

02 기초 API 익히기

이베스트투자증권은 xingAPI 패키지를 더욱 쉽게 사용할 수 있도록 패키지 가이드를 제공합니다. 증권사에서 제공하는 API를 사용할 때는 이처럼 증권사에서 제공하는 도움말이나 가이드 파일을 참조하면 더욱 쉽게 프로그램을 개발할 수 있습니다.

이베스트투자증권의 패키지 가이드를 참조하려면 이베스트투자증권 홈페이지에서 [고객센터] → [매매시스템] → [API] → [패키지가이드]로 이동합니다. 이베스트투자증권의 패키지 가이드는 DLL 버전 개발가이드, COM 버전 개발가이드, 데브센터 가이드, 가상거래소, XingACE 가이드로 구성돼 있습니다. 이 가운데 그림 11.10과 같이 [COM 버전 개발가이드]를 클릭합니다.

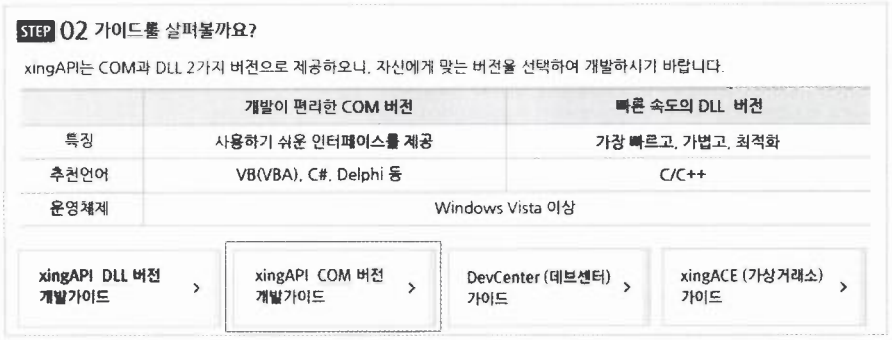


그림 11.10 패키지 가이드

개발 가이드는 그림 11.11과 같이 웹 브라우저를 통해 확인할 수 있으며 해당 주소[1]를 즐겨찾기에 추가해 두면 언제든지 편리하게 개발 가이드를 확인할 수 있습니다.

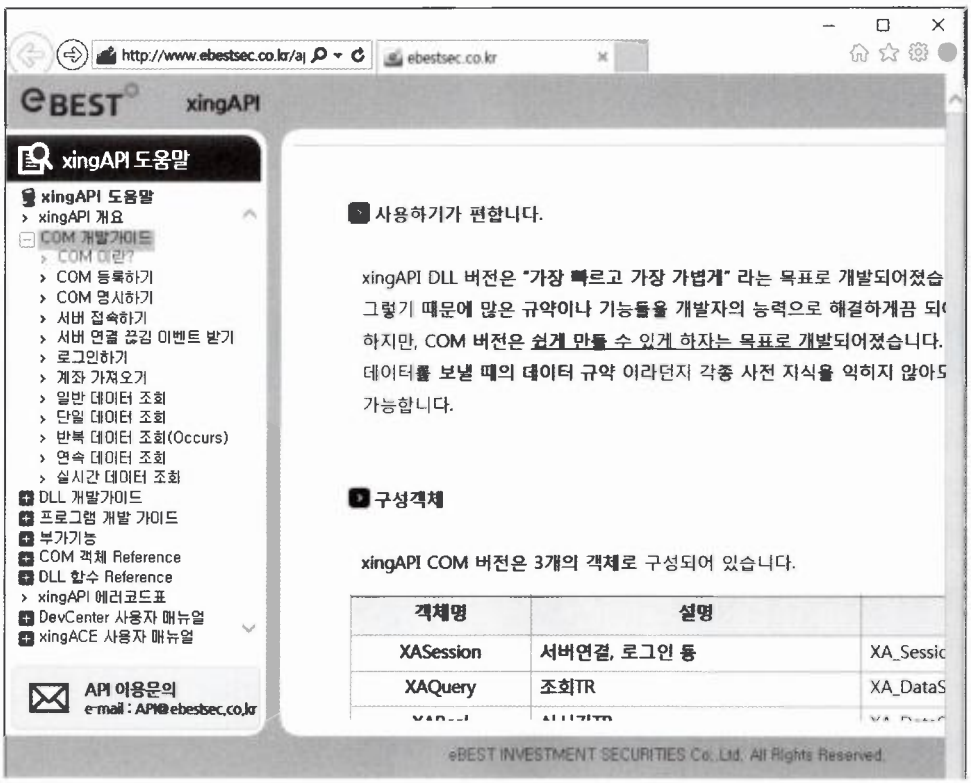


그림 11.11 COM 개발 가이드

1 http://www.ebestsec.co.kr/apiguide/guide.jsp?cno=200

로그인

이번 절에서는 xingAPI를 이용해 이베스트투자증권에 로그인해보겠습니다. 10장에서 배운 대신증권의 CYBOS Plus는 전용 프로그램을 통해 로그인했던 것과 달리 xingAPI는 API를 이용해 로그인할 수 있습니다.

API를 이용해 로그인하는 것은 알고리즘 트레이딩 프로그램 자동화 관점에서는 분명히 편리합니다. 다만 초보자에게는 API를 이용한 로그인이라는 과정 자체가 큰 진입장벽이 되기도 합니다.

xingAPI COM 버전은 표 11.1과 같이 3개의 클래스를 제공합니다. 대신증권의 CYBOS Plus가 많은 클래스를 제공했던 것과 달리 xingAPI는 클래스 수가 많지 않습니다. 서버 연결 및 로그인을 수행하려면 XASession이라는 이름의 COM 클래스에 대한 인스턴스를 생성하고 해당 인스턴스가 제공하는 메서드를 호출하면 됩니다.

표 11.1 xingAPI COM 버전 클래스

클래스명	설명	파일명
XASession	서버 연결, 로그인	XA_Session.dll
XAQuery	조회 TR	XA_DataSet.dll
XAReal	실시간 TR	XA_DataSet.dll

xingAPI의 XASession 클래스는 10장에서 배운 대신증권의 기초 API 및 Request/Reply 타입의 기본 API와 달리 이벤트(Event) 방식으로 동작합니다. 참고로 대신증권의 Request/Reply 타입 API는 단순히 서버에 데이터를 요청한 후 데이터를 가져오는 메서드를 호출해서 데이터를 바로 얻어올 수 있었습니다.

그러나 XASession 클래스는 서버에 데이터나 어떤 작업의 수행을 요청한 후 결괏값을 얻을 때 여러분이 직접 메서드를 호출해서 데이터를 얻는 구조가 아닙니다. 여러분이 요청한 데이터나 작업에 대해 서버가 작업을 수행한 후 완료됐을 때 다시 알려주는 방식입니다.

두 가지 방식의 차이점이 잘 이해되지 않는 분들을 위해 커피숍에서 커피를 주문하는 방법을 예로 들어 설명해 보겠습니다.

첫 번째 방식(Request/Reply)은 커피숍 계산대에서 커피를 주문하고 커피가 나올 때까지 그 자리에서 기다렸다가 커피를 받는 방식입니다. 두 번째 방식(Event)은 커피를 주문한 후 커피를 바로 받지 않고 진동벨만 받아 자리로 돌아온 후 커피가 준비됐을 때 진동벨을 통해 알림을 받는 방식입니다.

첫 번째 방식은 커피가 준비되는 동안 여러분이 계속 계산대에서 기다려야 하므로 불편할 것입니다. 또한 커피숍 입장에서도 주문을 받은 후 커피를 줄 때까지 새로운 주문을 받지 못하므로 주문을 효과적으로 처리하기가 어렵습니다.

이와 달리 두 번째 방식은 주문과 커피 준비를 따로 분리해서 처리할 수 있습니다. 고객 입장에서도 커피가 준비되는 동안 자리에 돌아가서 다른 일을 하다가 커피가 준비되면 즉시 알림을 받을 수 있으므로 편리합니다.

앞서 XASession 클래스가 이벤트 방식으로 동작한다고 설명했습니다. 여기서 이벤트라는 것은 커피숍에서 커피가 준비됐을 때 진동벨이 울리는 동작과 같습니다. 커피숍은 커피가 준비되면 고객이 알 수 있도록 진동벨이 울리는 이벤트를 발생시키는 것입니다. xingAPI도 메서드 호출을 통해 어떤 작업을 요청했을 때 해당 작업이 완료되면 이벤트를 발생시켜 알려주는 것입니다.

커피숍에서는 이벤트를 받기 위해 진동벨이라는 장비를 사용했습니다. xingAPI를 사용하는 프로그램에서는 어떻게 하면 서버가 알려주는 이벤트를 받을 수 있을까요? 이벤트 방식의 API는 분명 편리하지만 이벤트를 처리하는 코드를 추가로 작성해야 하므로 초보자에게는 어려울 수 있습니다.

커피숍에서 진동벨을 사용했던 것처럼 먼저 서버로부터 전송되는 이벤트를 처리하기 위한 전용 클래스인 XASessionEventHandler를 작성해보겠습니다. 참고로 XASession을 사용할 때 서버로부터 발생하는 이벤트를 처리하는 클래스이므로 클래스 이름을 XASessionEventHandler라고 한 것입니다.

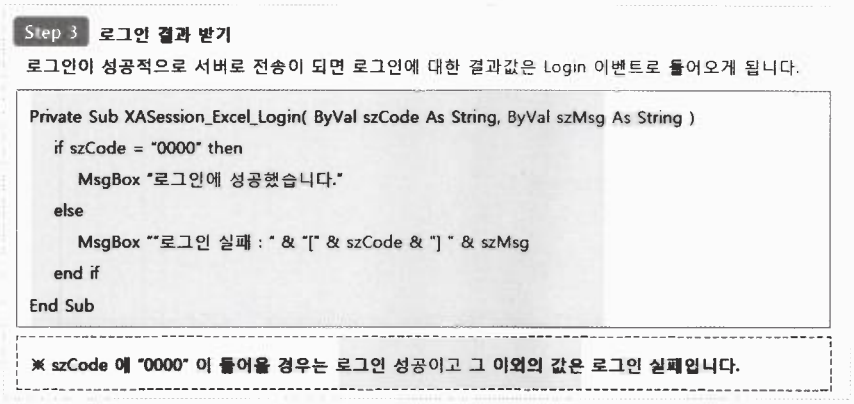
```
class XASessionEventHandler:
    login_state = 0

    def OnLogin(self, code, msg):
        if code == "0000":
            print("로그인 성공")
            XASessionEventHandler.login_state = 1
        else:
            print("로그인 실패")
```

XASessionEventHandler 클래스는 로그인에 대한 결괏값을 받을 수 있는 OnLogin 메서드를 포함합니다. 여기서 OnLogin 메서드는 여러분이 구현한 메서드지만 여러분이 호출하는 메서드는 아닙니다. 이베스트투자증권의 서버가 로그인 결과를 여러분에게 통지할 때 이 메서드를 호출하는 것입니다. 보통 프로그래밍에서는 이 같은 형태의 함수나 메서드를 콜백(callback) 함수 또는 콜백 메서드라고 합니다.

OnLogin 메서드는 이베스트투자증권 서버가 직접 호출하는 메서드이므로 메서드의 이름을 변경해서는 안 됩니다. 이름을 변경하면 서버가 여러분에게 이벤트 발생 여부를 알려줄 방법이 없습니다.

OnLogin 메서드의 인자를 살펴보면 self 외에도 추가로 code와 msg라는 변수가 있음을 확인할 수 있습니다. 이는 그림 11.12와 같이 개발 가이드의 '로그인 결과 받기' 부분을 참조해서 구현한 것입니다.


Step 3 로그인 결과 받기

로그인이 성공적으로 서버로 전송이 되면 로그인에 대한 결과값은 Login 이벤트로 들어오게 됩니다.

```
Private Sub XASession_Excel_Login( ByVal szCode As String, ByVal szMsg As String )
    if szCode = "0000" then
        MsgBox "로그인에 성공했습니다."
    else
        MsgBox ""로그인 실패 : " & "[" & szCode & "] " & szMsg
    end if
End Sub
```

※ szCode 에 "0000" 이 들어올 경우는 로그인 성공이고 그 이외의 값은 로그인 실패입니다.

그림 11.12 로그인 결과 받기(xingAPI 개발 가이드)

Login 이벤트가 발생할 때 szCode와 szMsg라는 값이 서버로부터 전달되기 때문에 파이썬 코드에서도 해당 값을 전달받기 위해 code와 msg라는 두 개의 인자를 추가했습니다. 특히 code라는 변수의 값으로 '0000'이라는 문자열 값이 들어오면 로그인이 성공한 것이고 그 밖의 값이 들어오면 로그인이 실패한 것이기 때문에 OnLogin 메서드를 이에 맞춰 구현했습니다.

XASessionEventHandler 클래스에는 login_state라는 클래스 변수가 있습니다. 클래스에는 인스턴스 변수와 클래스 변수가 있다는 것을 다들 기억하시죠? login_state 클래스 변수는 서버에 로그인된 상태인지를 저장하는 데 사용됩니다. 초깃값으로 0을 가지고 있다가 로그인이 성공하면 해당 값을 1로 변경합니다.

이벤트를 처리할 클래스를 구현했으니 XASession 클래스에 대한 인스턴스를 생성해보겠습니다. 이벤트 방식으로 동작하는 COM 클래스에 대한 인스턴스를 생성하기 위해서는 win32com.client 모듈의 Dispatch 함수가 아니라 DispatchWithEvents 함수를 사용하면 됩니다.

DispatchWithEvents 함수의 첫 번째 인자로는 인스턴스를 생성하려는 클래스의 이름을 지정하면 됩니다. 표 11.1을 참조하면 XASession 클래스는 XA_Session.dll이라는 파일에 구현돼 있으므로 'XA_Session.XASession'을 입력했습니다.

두 번째 인자로는 생성할 클래스에 대한 이벤트를 처리할 클래스의 이름을 지정하면 됩니다. XASessionEventHandler라는 이름으로 이벤트 처리용 클래스를 구현했으므로 XASessionEventHandler라는 이름을 입력하면 됩니다.

```
import win32com.client

class XASessionEventHandler:
    login_state = 0

    def OnLogin(self, code, msg):
        if code == "0000":
            print("로그인 성공")
            XASessionEventHandler.login_state = 1
        else:
            print("로그인 실패")

instXASession = win32com.client.DispatchWithEvents("XA_Session.XASession", XASessionEventHan-
dler)
```

instXASession이라는 이름의 객체가 생성됐으므로 해당 객체를 사용해 서버에 연결하거나 로그인을 위한 메서드를 호출하면 됩니다. 개발 가이드를 참조하면 서버 연결에는 ConnectServer라는 메서드를 사용하고, 로그인에는 Login이라는 메서드를 사용한다는 사실을 알 수 있습니다.

접속할 서버의 기본 주소는 'hts.ebestsec.co.kr'인데 모의 투자인 경우에는 'demo.ebestsec.co.kr'을 사용하면 됩니다. Login 메서드에는 사용자 아이디, 사용자 비밀번호, 공인인증서 비밀번호를 순서대로 입력해야 합니다. 계정 정보를 효과적으로 관리하기 위해 id, passwd, certificate_passwd라는 변수를 사용해 봅시다. 해당 변수에 여러분의 로그인 정보를 입력해두면 Login 메서드에서는 이러한 변수를 사용해 로그인을 진행합니다.

```
id = "id"
passwd = "passwd"
cert_passwd = "cert"

instXASession.ConnectServer("hts.ebestsec.co.kr", 20001)
instXASession.Login(id, passwd, cert_passwd, 0, 0)
```

PyCharm과 같은 IDE를 통해 지금까지 설명한 코드를 실행해 보면 다음과 같이 로그인 관련 메시지가 출력되지 않음을 확인할 수 있습니다. XASessionEventHandler 클래스에 OnLogin 메서드를 구현해 로그인이 성공하거나 실패할 때 메시지를 출력하게 했지만 어떤 메시지도 출력되지 않은 것입니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch11/02.py

Process finished with exit code 0
```

앞서 작성한 코드에서 어떤 부분이 잘못된 것일까요? 정답은 이벤트 방식으로 작성된 클래스에서 이벤트를 받기 위해서는 이벤트가 발생할 때까지 대기해야 한다는 것입니다.

커피숍의 예에서 커피를 주문한 후 진동벨을 가지고 자리에서 일정 시간 동안 기다리고 있어야 진동벨이 울리는 것을 확인할 수 있는 것과 같습니다. 그러나 조금 전에 작성한 코드에는 이벤트가 발생할 때까지 대기하는 코드가 없었습니다.

그림 11.13은 지금까지 작성한 파이썬 코드의 실행 흐름입니다. 파이썬 코드는 소스코드의 첫 번째 줄부터 소스코드가 끝날 때까지 순서대로 실행됩니다. 물론 함수, 메서드, 분기문, 조건문에 따라 코드의 실행 순서가 조금씩 바뀌기는 하지만 큰 흐름은 위에서 아래로 실행됩니다. 작성된 코드에서 ConnectServer나 Login 메서드를 호출하는 부분은 소스코드의 끝부분에 위치하기 때문에 파이썬 인터프리터는 해당 메서드를 호출하고 더는 실행할 코드가 없으므로 프로그램을 종료합니다. 따라서 증권사 서버가 콜백하고자 하는 시점에 여러분의 프로그램은 실행 상태에 있지 않았기 때문에 제대로 동작하지 않았던 것입니다.

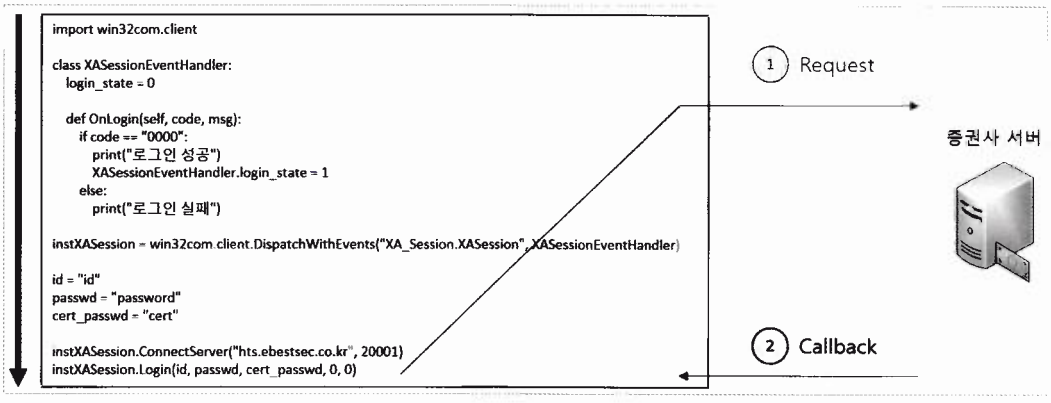


그림 11.13 파이썬 코드의 실행 흐름

여러분이 작성한 프로그램이 증권사 서버가 콜백 함수를 호출할 때까지 종료되지 않고 대기하게 하는 방법은 여러 가지가 있지만 가장 쉬운 방법은 pythoncom 모듈의 PumpWatingMessages라는 함수를 무한 루프 내에서 호출하는 것입니다.

파이썬에서 무한 루프는 while 문을 통해 구현할 수 있었지요? 그러나 무한 루프를 만들게 되면 무한 루프 부분에서 여러분이 작성한 프로그램이 계속해서 동작하기 때문에 단순히 'while 1:'과 같은 형

태로 무한 루프를 만들면 안 됩니다. 로그인이 정상적으로 이뤄졌다면 무한 루프를 벗어나도록 코드를 구현해야 로그인 후에 추가 작업을 위한 코드를 실행할 수 있습니다.

예제 11.1은 지금까지 설명한 코드에 pythoncom 모듈의 PumpWaitingMessages 함수를 이용해 무한 루프를 구현한 것입니다. while 문에서는 XASessionEvents 클래스의 클래스 변수인 login_state 값을 확인함으로써 해당 값이 1로 변경되는 시점에 무한 루프를 빠져나오도록 구현돼 있습니다.

예제 11.1 xingAPI 로그인(소스코드: book/ch11/03.py)

```
import win32com.client
import pythoncom

class XASessionEventHandler:
    login_state = 0

    def OnLogin(self, code, msg):
        if code == "0000":
            print("로그인 성공")
            XASessionEventHandler.login_state = 1
        else:
            print("로그인 실패")

instXASession = win32com.client.DispatchWithEvents("XA_Session.XASession", XASessionEventHandler)

id = "아이디"
passwd = "비밀번호"
cert_passwd = "공인인증서"

instXASession.ConnectServer("hts.ebestsec.co.kr", 20001)
instXASession.Login(id, passwd, cert_passwd, 0, 0)

while XASessionEventHandler.login_state == 0:
    pythoncom.PumpWaitingMessages()
```

예제 11.1의 코드를 실행하면 다음과 같이 '로그인 성공'이라는 메시지가 정상적으로 출력되는 것을 확인할 수 있습니다. 참고로 예제 11.1의 코드에서 id, passwd, cert_passwd라는 변수에는 여러분의 로그인 아이디, 비밀번호, 공인인증서 비밀번호를 입력해야 합니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch11/03.py
로그인 성공

Process finished with exit code 0
```

계좌 정보 조회

이번 절에서는 XASession 클래스를 이용해 계좌 정보를 조회해 보겠습니다. 계좌 정보는 이베스트투자증권 홈페이지의 [마이페이지]를 통해 확인할 수 있습니다. 모의투자인 경우에는 그림 11.14와 같이 [모의투자] 메뉴의 [나의계좌정보]에서 모의 투자 전용 계좌를 확인할 수 있습니다.

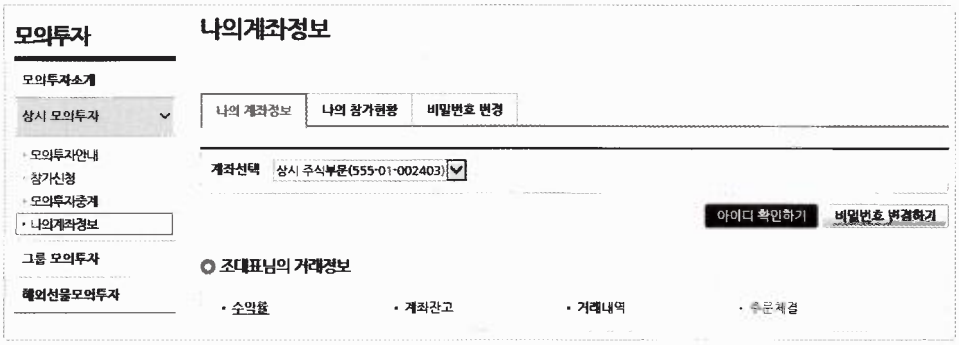


그림 11.14 계좌 정보 확인

xingAPI로 계좌 정보를 조회하려면 앞서 살펴본 XASession 클래스를 사용하면 됩니다. 하나 이상의 계좌가 있을 수 있으므로 계좌 정보를 가져오려면 먼저 계좌의 총 개수를 구해야 합니다. 그리고 계좌의 개수만큼 반복적으로 계좌를 가져오면 됩니다.

xingAPI의 개발 가이드를 참조하면 그림 11.15와 같은 코드를 볼 수 있습니다. GetAccountListCount 메서드는 계좌의 개수를 가져오는 메서드이고 GetAccountList는 계좌 번호를 가져오는 메서드입니다.

Full Code 계좌 가져오기 Full Code 입니다.

```
nAccountCount = XASession_Excel.GetAccountListCount()

For i=0 To nAccountCount - 1
    szAccount = XASession_Excel.GetAccountList( i )
Next
```

그림 11.15 계좌 가져오기(xingAPI 개발 가이드)

그림 11.15는 VBA(Visual Basic for Applications) 코드인데 이를 파이썬으로 다시 구현하면 다음과 같습니다. GetAccountListCount와 GetAccountList 메서드는 XASession 클래스의 메서드이므로 XASession 클래스의 인스턴스인 instXASession을 사용하면 됩니다.

```
num_account = instXASession.GetAccountListCount()
for i in range(num_account):
    account = instXASession.GetAccountList(i)
    print(account)
```

계좌를 가져오려면 당연히 로그인을 먼저 수행해야 합니다. 로그인 후 계좌를 가져오는 전체 코드는 예제 11.2와 같습니다.

예제 11.2 xingAPI를 이용한 로그인 및 계좌 조회(소스코드: book/ch11/04.py)

```
import win32com.client
import pythoncom

class XASessionEventHandler:
    login_state = 0

    def OnLogin(self, code, msg):
        if code == "0000":
            print("로그인 성공")
            XASessionEventHandler.login_state = 1
        else:
            print("로그인 실패")

instXASession = win32com.client.DispatchWithEvents("XA_Session.XASession", XASessionEventHandler)

id = "아이디"
passwd = "비밀번호"
cert_passwd = "공인인증서"

instXASession.ConnectServer("hts.ebestsec.co.kr", 20001)
instXASession.Login(id, passwd, cert_passwd, 0, 0)

while XASessionEventHandler.login_state == 0:
    pythoncom.PumpWaitingMessages()

num_account = instXASession.GetAccountListCount()
```

```
for i in range(num_account):
    account = instXASession.GetAccountList(i)
    print(account)
```

예제 11.2에서 id, passwd, cert_passwd에 아이디, 비밀번호, 공인인증서 비밀번호를 입력한 후 예제를 실행하면 다음과 같이 계좌 정보가 출력되는 것을 확인할 수 있습니다. 출력된 계좌 번호가 여러분의 계좌 번호와 동일한지 확인해보기 바랍니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch11/04.py
로그인 성공
00201721815

Process finished with exit code 0
```

03 기본 API 익히기

이번 절에서는 xingAPI의 XAQuery와 XAReal 클래스를 이용해 서버로부터 데이터를 받아오는 기본적인 API를 배워보겠습니다.

DevCenter 사용하기

이베스트투자증권에서는 DevCenter라는 xingAPI 테스트 및 디버깅 환경을 제공합니다. 이번 절에서는 파이썬에서 xingAPI를 사용하기에 앞서 DevCenter를 통해 API를 테스트해보겠습니다. 그림 11.16과 같이 DevCenter를 실행하고 ID, 비밀번호, 공인인증 비밀번호 등을 입력한 후 접속합니다.

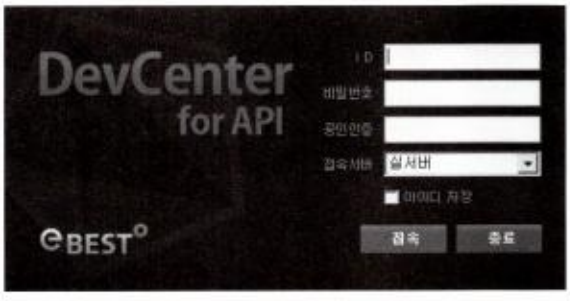


그림 11.16 DevCenter 로그인

DevCenter에 정상적으로 로그인했다면 그림 11.17과 같이 Res 파일 다운로드 버튼을 눌러 Res 파일을 컴퓨터로 다운로드합니다. 참고로 Res 파일을 다운로드하는 것은 DevCenter를 설치한 후 한 번만 하면 됩니다.

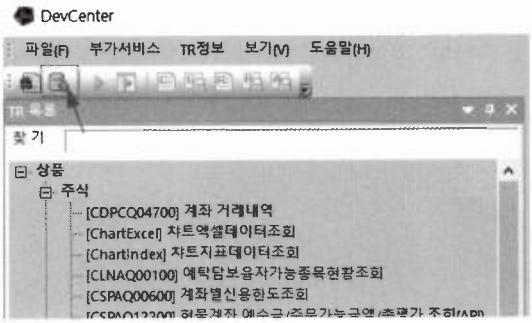


그림 11.17 Res 파일 다운로드

Res 파일은 C:\eBest\xingAPI\Res 경로와 같이 eBest의 xingAPI가 설치된 디렉터리로 다운로드됩니다. 참고로 Res 파일은 xingAPI가 제공하는 메서드의 구조를 표현하는 파일입니다. Res 파일이 궁금하신 분들은 메모장과 같은 프로그램으로 Res 파일을 열어보기 바랍니다.

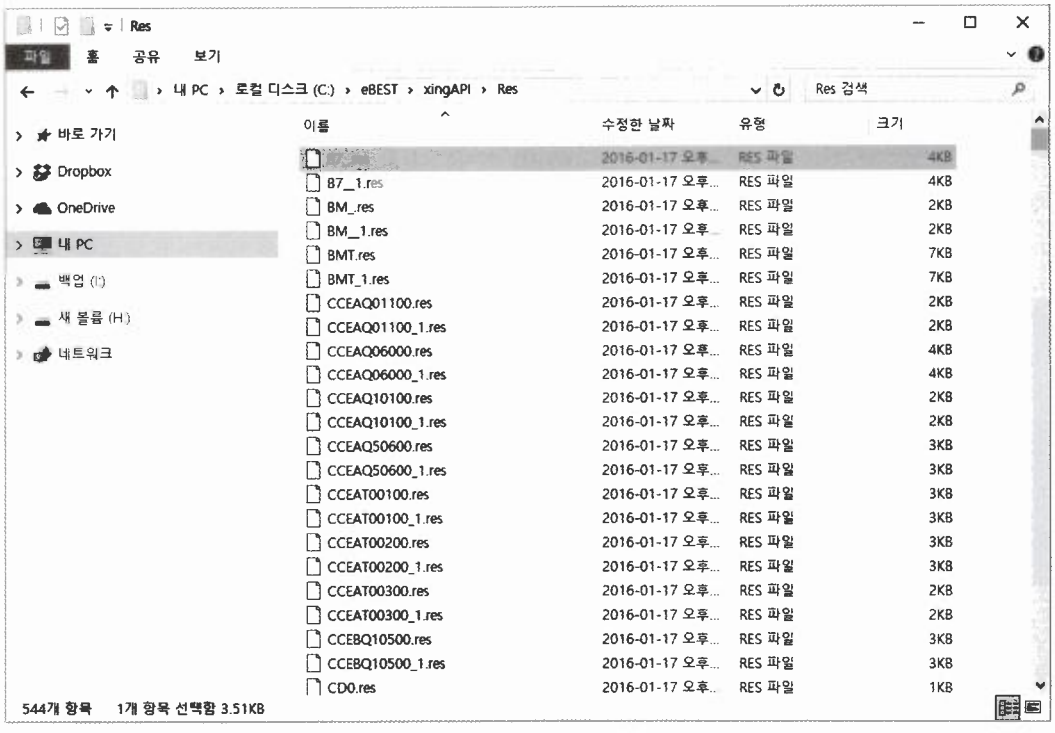


그림 11.18 xingAPI Res 파일

xingAPI 개발 가이드를 보면 xingAPI에서는 TR(Transaction)이라는 용어를 자주 사용합니다. xingAPI에서 TR이라는 용어는 '서버로부터 데이터를 주고받는 행위'를 의미합니다. 예를 들어, 특정 종목의 종목 코드를 입력해서 서버로부터 현재가를 받아오는 것도 TR이고, 계좌 번호를 입력해서 계좌의 잔고를 받아오는 것도 TR입니다.

DevCenter를 이용해 주식 종목의 현재가를 조회하는 't1102'를 테스트해보기 위해 그림 11.19와 같이 DevCenter에서 t1102를 선택한 후 [TR 확인 창] 버튼을 클릭합니다.

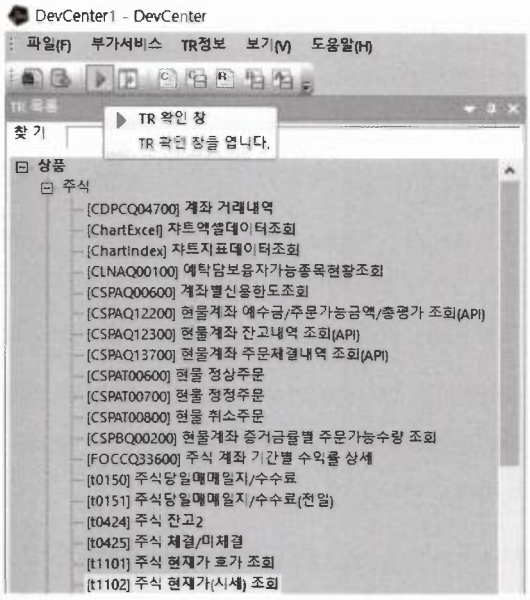


그림 11.19 TR 확인 창 열기

t1102는 주식의 현재가를 조회하는 데 사용되며 t1102와 같은 것을 TR 코드라고 합니다. xingAPI에는 기능에 따라 수많은 TR 코드가 존재합니다.

그림 11.20과 같이 t1102 TR 코드에 대한 TR 확인 창이 나타나면 shcode에 078020을 입력한 후 [조회] 버튼을 클릭합니다. [조회] 버튼을 클릭하면 Out Block이라고 표시된 부분에 이베스트투자증권의 종목 한글명, 등락율, 현재가, 상한가, 하한가 등 여러 데이터가 출력되는 것을 확인할 수 있습니다.

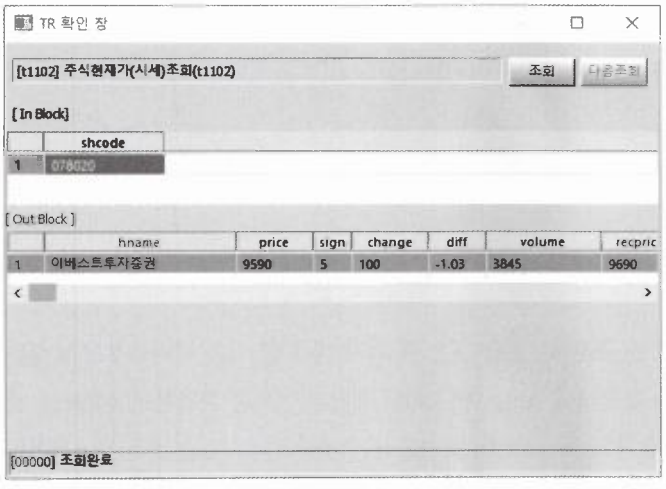


그림 11.20 t1102 TR 코드 확인 창

단일 데이터 조회하기

앞 절에서 DevCenter를 이용해 t1102이라는 TR 코드의 동작 및 입/출력 값을 살펴봤습니다. 이번 절에서는 xingAPI를 이용해 해당 기능을 수행하는 코드를 작성해보겠습니다. 참고로 t1102 TR 코드와 같이 값(종목 코드)을 한번 전송해서 결괏값을 한번 얻어오는 방식을 단일 데이터 조회라고 합니다.

그림 11.21은 XAQuery 클래스를 이용해 단일 데이터를 조회하는 순서를 나타낸 것입니다. XAQuery 클래스를 사용할 때도 발생하는 이벤트를 처리하기 위한 클래스를 먼저 구현해야 합니다(단계 1). 그리고 XAQuery 클래스에 대한 인스턴스를 생성합니다(단계 2).

xingAPI는 대신증권의 CYBOS Plus와 달리 클래스의 수가 많지 않습니다. 대신 요청하는 데이터나 작업을 TR 코드를 통해 구분하기 때문에 앞서 내려받은 Res 파일을 XAQuery 클래스의 인스턴스에 등록해야 합니다(단계 3).

XAQuery 클래스의 메서드인 SetFieldData 메서드를 통해 요청하려는 데이터나 작업을 위한 입력값을 적절히 설정한 후(단계 4) Request 메서드를 통해 서버에 요청합니다(단계 5).

서버에 요청한 작업은 서버가 해당 작업을 처리한 후 이벤트를 통해 알려줄 때까지는 완료된 것이 아닙니다. 따라서 서버로부터 작업 완료에 대한 이벤트가 올 때까지 대기해야 합니다(단계 6).

서버로부터 요청한 작업에 대한 완료 이벤트를 받았다면 XAQuery 클래스의 ReceiveData 메서드를 통해 데이터를 받아오면 됩니다(단계 7).

XAQuery 클래스를 처음 사용해보기 때문에 아직 순서가 잘 이해되지 않을 수 있습니다. 처음부터 이러한 순서를 외우려 하기보다는 각 순서의 흐름을 이해하면 되겠습니다.

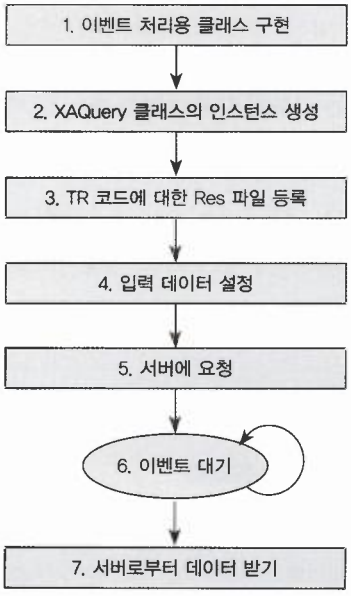


그림 11.21 단일 데이터를 조회하는 순서

그림 11.21에서 설명한 순서대로 코드를 구현해보겠습니다. 맨 먼저 해야 할 일은 서버로부터 발생하는 이벤트를 처리할 클래스를 구현하는 것입니다. xingAPI COM 개발 가이드를 참조하면 XAQuery 클래스에 대한 이벤트는 그림 11.22와 같이 ReceiveData를 사용하는 것을 알 수 있습니다. 즉, 서버는

XAQuery 인스턴스가 요청한 작업을 처리한 후 ReceiveData 메서드를 콜백함으로써 해당 프로그램에 작업 완료 여부를 통지합니다.

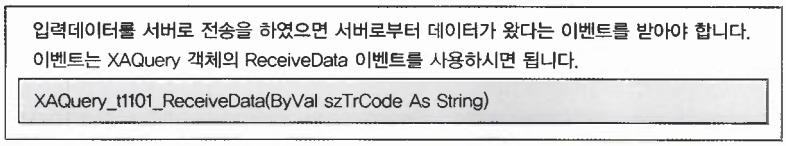


그림 11.22 XAQuery 객체의 ReceiveData 이벤트(xingAPI 개발 가이드)

t1102 TR 코드에 대한 이벤트를 처리할 클래스이므로 클래스 이름을 XAQueryEventHandlerT1102로 지정했습니다. 그리고 query_state라는 이름의 클래스 변수를 사용해 이벤트 처리 여부를 저장합니다. 예를 들어, 서버에 요청한 작업이 완료되면 서버는 OnReceiveData 메서드를 호출하게 되며, 이에 따라 query_state의 값이 1로 변경됩니다.

클래스 외부에서는 클래스 변수인 query_state를 통해 요청한 작업의 완료 여부를 확인할 수 있습니다.

```
class XAQueryEventHandlerT1102:
    query_state = 0

    def OnReceiveData(self, code):
        XAQueryEventHandlerT1102.query_state = 1
```

이벤트 처리를 위한 클래스를 정의했다면 XAQuery 클래스의 인스턴스를 생성합니다. XASession 클래스의 인스턴스 생성과 마찬가지로 DispatchWithEvents 함수를 사용합니다.

```
instXAQueryT1102 = win32com.client.DispatchWithEvents("XA_DataSet.XAQuery", XAQueryEventHandlerT1102)
```

생성된 instXAQueryT1102 객체는 단순히 XAQuery 클래스에 대한 인스턴스일 뿐입니다. 따라서 해당 인스턴스를 통해 메서드를 호출했을 때 서버는 어떤 TR 요청인지 알 수 없습니다.

xingAPI에서는 각 요청 작업을 구분하기 위해 TR 코드를 사용하는데 해당 TR 코드에 대한 Res 파일을 앞서 생성한 클래스의 인스턴스에 등록함으로써 서버가 어떤 TR 요청을 받은 것인지를 구분할 수 있게 합니다.

Res 파일은 앞서 일괄적으로 다운로드했었는데 xingAPI 설치 경로를 변경하지 않았다면 다음과 같이 입력하면 됩니다. 여기서 파이썬에서 경로를 입력할 때 디렉터리의 구분은 '\\' 또는 '/'를 사용해야 한다는 점에 주의합니다.

```
instXAQueryT1102.ResFileName = "C:\\eBEST\\xingAPI\\Res\\t1102.res"
```

TR 코드에 대한 Res 파일 등록을 완료했다면 요청할 작업에 필요한 입력 데이터를 설정합니다. 그림 11.23과 같이 DevCenter에서 t1102 TR 코드를 살펴보면 t1102InBlock이라는 블록명에 shcode라는 필드가 존재하는 것을 확인할 수 있습니다.



그림 11.23 t1102 TR 코드의 입/출력

파이썬 코드로 입력 데이터를 설정하려면 XAQuery 인스턴스를 통해 SetFieldData라는 메서드를 호출한 후 적절한 인자 값을 지정하면 됩니다. SetFieldData의 1번째 파라미터는 블록명이고, 2번째 파라미터는 필드명입니다. 3번째 파라미터에는 단일 데이터를 조회할 때는 0을 지정하면 되고, 4번째 파라미터는 필드에 해당하는 입력값입니다.

```
instXAQueryT1102.SetFieldData("t1102InBlock", "shcode", 0, "078020")
```

요청할 작업에 필요한 데이터 입력이 완료되면 Request 메서드를 호출해서 입력 데이터를 서버로 전송합니다.

```
instXAQueryT1102.Request(0)
```

서버에 TR 요청을 했다면 해당 작업이 완료됐다는 이벤트를 받을 때까지 프로그램이 종료되지 않고 대기해야 합니다.

```
while XAQueryEventHandlerT1102.query_state == 0:
    pythoncom.PumpWaitingMessages()
```

이베스트투자증권의 서버는 TR 처리가 완료되면 OnReceiveData 메서드를 콜백합니다. 이로 인해 XAQueryEventHandlerT1102.query_state 값이 1로 변경되어 이벤트 대기 루프에서 빠져나오게 됩니다.

작업 완료 이벤트를 받은 후에는 GetFieldData 메서드를 사용해 t1102OutBlock 블록명과 원하는 필드명을 입력해 데이터를 가져올 수 있습니다. 예를 들어, "hname" 필드명을 지정하면 한글종목명을 가져올 수 있고, "price" 필드명을 지정하면 종목의 현재가를 가져올 수 있습니다.

```
name = instXAQueryT1102.GetFieldData("t1102OutBlock", "hname", 0)
price = instXAQueryT1102.GetFieldData("t1102OutBlock", "price", 0)
print(name)
print(price)
```

예제 11.3은 지금까지 설명한 내용을 하나의 파일로 구현한 코드입니다. 물론 해당 코드를 실행하기 위해서는 먼저 서버 접속 및 로그인이 필요하므로 이를 위한 코드도 함께 포함돼 있습니다.

예제 11.3 단일 데이터 조회(t1102 TR 코드)(소스코드: book/ch11/06.py)

```
import win32com.client
import pythoncom

class XASessionEventHandler:
    login_state = 0

    def OnLogin(self, code, msg):
        if code == "0000":
            print("로그인 성공")
            XASessionEventHandler.login_state = 1
        else:
            print("로그인 실패")

class XAQueryEventHandlerT1102:
    query_state = 0

    def OnReceiveData(self, code):
        XAQueryEventHandlerT1102.query_state = 1

# ------------------------------------------------------------------------
# login
# ------------------------------------------------------------------------
```

```
id = "id"
passwd = "passwd"
cert_passwd = "cert"

instXASession = win32com.client.DispatchWithEvents("XA_Session.XASession", XASessionEventHan-
dler)
instXASession.ConnectServer("hts.ebestsec.co.kr", 20001)
instXASession.Login(id, passwd, cert_passwd, 0, 0)

while XASessionEventHandler.login_state == 0:
    pythoncom.PumpWaitingMessages()


# ------------------------------------------------------------------------------
# t1102
# ------------------------------------------------------------------------------
instXAQueryT1102 = win32com.client.DispatchWithEvents("XA_DataSet.XAQuery", XAQueryEventHan-
dlerT1102)
instXAQueryT1102.ResFileName = "C:\\eBEST\\xingAPI\\Res\\t1102.res"
instXAQueryT1102.SetFieldData("t1102InBlock", "shcode", 0, "078020")
instXAQueryT1102.Request(0)

while XAQueryEventHandlerT1102.query_state == 0:
    pythoncom.PumpWaitingMessages()

name = instXAQueryT1102.GetFieldData("t1102OutBlock", "hname", 0)
price = instXAQueryT1102.GetFieldData("t1102OutBlock", "price", 0)
print(name)
print(price)
```

예제 11.3의 코드를 PyCharm을 통해 실행해 보면 다음과 같이 '078020'에 해당하는 종목의 한글 종목명과 해당 종목의 현재가가 화면에 출력되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch11/06.py
로그인 성공
이베스트투자증권
9990

Process finished with exit code 0
```

반복 데이터 조회하기

반복 데이터는 데이터가 반복해서 나오는 구조를 말합니다. 예를 들어, 그림 11.24는 코스닥 시장의 각 종목에 대한 종목 단축 코드, 종목 확장 코드, ETF 구분 등과 같은 데이터를 보여줍니다. xingAPI에서는 이러한 형태의 데이터를 반복 데이터라고 하며, Occurs라는 용어로 부릅니다. 참고로 Occurs라는 단어는 프로그래밍 언어 중 COBOL에서 나온 단어라고 합니다.

	hname	shcode	expcode	etfgubun	uplmtprice	dnlmtprice	jnilclose	memedan	recprice	gubun
1	삼천당제약	000250	KR7000250001	0	11750	6330	9040	00001	9040	2
2	중앙에너비스	000440	KR7000440008	0	35800	19300	27550	00001	27550	2
3	신라섬유	001000	KR7001000009	0	15650	8450	12050	00001	12050	2
4	안국약품	001540	KR7001540004	0	26000	14000	20000	00001	20000	2
5	우림SP	001810	KR7001810001	0	4515	2435	3475	00001	3475	2
6	이화공영	001840	KR7001840008	0	3460	1870	2665	00001	2665	2
7	피에스텍	002230	KR7002230001	0	7460	4020	5740	00001	5740	2
8	삼일기업공사	002290	KR7002290005	0	3015	1625	2320	00001	2320	2
9	한탑	002680	KR7002680007	0	2290	1240	1765	00001	1765	2
10	선광	003100	KR7003100005	0	22900	12400	17650	00001	17650	2
11	대주산업	003310	KR7003310000	0	1335	725	1030	00001	1030	2
12	포스코캠텍	003670	KR7003670007	0	15000	8100	11550	00001	11550	2
13	에이스침대	003800	KR7003800000	0	180400	97200	138800	00001	138800	2
14	한국가구	004590	KR7004590006	0	47800	25800	36800	00001	36800	2
15	창해에탄올	004650	KR7004650008	0	32350	17450	24900	00001	24900	2
16	대륙제관	004780	KR7004780003	0	7930	4270	6100	00001	6100	2
17	동국산업	005160	KR7005160007	0	4320	2330	3325	00001	3325	2
18	동진쌔미켐	005290	KR7005290002	0	5930	3200	4565	00001	4565	2
19	푸드웰	005670	KR7005670005	0	6510	3510	5010	00001	5010	2

그림 11.24 반복 데이터 구조

xingAPI의 TR 코드 중 어떤 것이 단일 데이터이고 어떤 것이 반복 데이터인가를 구분하기 위해서는 그림 11.25와 같이 DevCenter에서 TR 속성 창의 Occurs 여부를 확인하면 됩니다. 여기에 동그라미 표시가 있다면 반복 데이터를 의미합니다.



그림 11.25 TR 코드의 Occurs 여부 확인

xingAPI를 이용한 반복 데이터 조회는 기본적으로 단일 데이터 조회와 큰 차이가 없습니다. 이번 절에서는 주식 종목 조회를 위한 TR 코드인 t8430을 사용해 반복 데이터를 조회하는 기본적인 방법을 살펴보겠습니다.

반복 데이터를 조회할 때도 기본적으로 그림 11.21의 순서대로 프로그램을 작성하면 됩니다. 먼저 이벤트 처리를 위한 클래스를 정의합니다.

```
class XAQueryEventHandlerT8430:
    query_state = 0

    def OnReceiveData(self, code):
        XAQueryEventHandlerT8430.query_state = 1
```

이벤트 처리를 위한 클래스를 정의했다면 XAQuery 클래스에 대한 인스턴스를 생성하고 해당 인스턴스에 t8430 TR 코드에 해당하는 Res 파일을 등록합니다.

```
instXAQueryT8430 = win32com.client.DispatchWithEvents("XA_DataSet.XAQuery", XAQueryEventHan-
dlerT8430)
instXAQueryT8430.ResFileName = "C:\\eBEST\\xingAPI\\Res\\t8430.res"
```

SetFieldData 메서드를 사용해 입력 데이터를 설정할 때는 그림 11.26과 같이 DevCenter에서 t8430에 대한 입력 블록명 및 필드명을 참조합니다.

t8430

이름	타입	크기	설명
t8430		78	주식종목조회
t8430InBlock	input	1	기본입력
gubun	string	1	구분(0:전체1:코스피2:코스닥)
t8430OutBlock - OCCURS	output	77	출력1
hname	string	20	종목명
shcode	string	6	단축코드
expcode	string	12	확장코드
etfgubun	string	1	ETF구분(1:ETF)
uplmtprice	long	8	상한가
dnlmtprice	long	8	하한가
jnilclose	long	8	전일가
memedan	string	5	주문수량단위
recprice	long	8	기준가
gubun	string	1	구분(1:코스피2:코스닥)

그림 11.26 t8430 TR 코드의 입/출력 블럭명 및 필드명

그림 11.26을 참조하면 입력 블록명은 t8430InBlock이고 입력에 사용되는 필드명은 gubun만 존재하는 것을 알 수 있습니다. gubun은 한글 '구분'의 발음을 그대로 변수명으로 사용한 것으로서 1을 입력하면 코스피(유가증권시장), 2를 입력하면 코스닥(코스닥시장), 0을 입력하면 전체 시장을 의미합니다. 유가증권시장의 종목명을 구하려면 다음과 같이 입력 데이터를 설정하면 됩니다.

```
instXAQueryT8430.SetFieldData("t8430InBlock", "gubun", 0, 1)
```

입력 데이터를 설정했다면 서버에 TR을 요청한 후 이벤트가 발생할 때까지 대기합니다.

```
instXAQueryT8430.Request(0)

while XAQueryEventsT8430.query_state == 0:
    pythoncom.PumpWaitingMessages()
```

TR에 대한 완료 이벤트가 발생했다면 데이터를 가져오면 됩니다. 이때 반복 데이터를 조회하려면 단일 데이터 조회와 달리 데이터의 총 개수를 먼저 구해야 합니다.

뭔가 복잡해 보이지만 그림 11.27과 같이 엑셀에서 각 행 단위로 접근해서 데이터를 구하는 것과 비슷합니다. 그림 11.27의 데이터를 기준으로 설명해보면 데이터의 총 개수는 5가 될 것입니다. 따라서 0~4까지의 인덱스를 통해 각 행에 해당하는 데이터를 구할 수 있습니다.

	A	B	C	D	E
1	0	동화약품	000020	KR7000020008	0
2	1	우리은행	000030	KR7000030007	0
3	2	KR모터스	000040	KR7000040006	0
4	3	경방	000050	KR7000050005	0
5	4	메리츠화재	000060	KR7000060004	0

그림 11.27 반복 데이터 구조의 엑셀 데이터

반복 데이터의 총 개수를 구하려면 GetBlockCount 메서드를 사용하고, 각 인덱스에 해당하는 데이터를 가져올 때는 앞서 살펴본 GetFieldData 메서드를 사용합니다. 다만 단일 데이터와 달리 3번째 파라미터에 반복 데이터의 인덱스를 지정하면 됩니다.

```
count = instXAQueryT8430.GetBlockCount("t8430OutBlock")

for i in range(5):
    hname = instXAQueryT8430.GetFieldData("t8430OutBlock", "hname", i)
    shcode = instXAQueryT8430.GetFieldData("t8430OutBlock", "shcode", i)
    expcode = instXAQueryT8430.GetFieldData("t8430OutBlock", "expcode", i)
    etfgubun = instXAQueryT8430.GetFieldData("t8430OutBlock", "etfgubun", i)
    print(i, hname, shcode, expcode, etfgubun)
```

위 코드는 0~4번 인덱스에 해당하는 유가증권시장 종목에 대한 종목명(hname), 단축코드(shcode), 확장코드(expcode), ETF 구분(etfgubun)을 출력합니다.

예제 11.4는 t8430 TR 코드를 이용해 반복 데이터를 조회하는 전체 코드입니다. 반복 데이터를 조회하는 코드는 단일 데이터를 조회하는 코드와 비슷합니다. 단지 GetBlockCount 메서드를 사용해 데이터의 총 개수를 가져오고 GetFieldData 메서드에서 데이터 인덱스를 입력해서 데이터를 가져온다는 점만 다릅니다.

예제 11.4 반복 데이터 조회(t8430 TR 코드)(소스코드: book/ch11/08.py)

```
import win32com.client
import pythoncom

class XASessionEventHandler:
    login_state = 0

    def OnLogin(self, code, msg):
        if code == "0000":
```

```
            print("로그인 성공")
            XASessionEventHandler.login_state = 1
        else:
            print("로그인 실패")

class XAQueryEventHandlerT8430:
    query_state = 0

    def OnReceiveData(self, code):
        XAQueryEventHandlerT8430.query_state = 1


# ------------------------------------------------------------------------------
# login
# ------------------------------------------------------------------------------
id = "id"
passwd = "passwd"
cert_passwd = "cert"

instXASession = win32com.client.DispatchWithEvents("XA_Session.XASession", XASessionEventHan-
dler)
instXASession.ConnectServer("hts.ebestsec.co.kr", 20001)
instXASession.Login(id, passwd, cert_passwd, 0, 0)

while XASessionEventHandler.login_state == 0:
    pythoncom.PumpWaitingMessages()


# ------------------------------------------------------------------------------
# T8430
# ------------------------------------------------------------------------------
instXAQueryT8430 = win32com.client.DispatchWithEvents("XA_DataSet.XAQuery", XAQueryEventHan-
dlerT8430)
instXAQueryT8430.ResFileName = "C:\\eBEST\\xingAPI\\Res\\t8430.res"

instXAQueryT8430.SetFieldData("t8430InBlock", "gubun", 0, 1)
instXAQueryT8430.Request(0)

while XAQueryEventHandlerT8430.query_state == 0:
    pythoncom.PumpWaitingMessages()
```

```
count = instXAQueryT8430.GetBlockCount("t8430OutBlock")
for i in range(5):
    hname = instXAQueryT8430.GetFieldData("t8430OutBlock", "hname", i)
    shcode = instXAQueryT8430.GetFieldData("t8430OutBlock", "shcode", i)
    expcode = instXAQueryT8430.GetFieldData("t8430OutBlock", "expcode", i)
    etfgubun = instXAQueryT8430.GetFieldData("t8430OutBlock", "etfgubun", i)
    print(i, hname, shcode, expcode, etfgubun)
```

예제 11.4의 코드를 PyCharm에서 실행하면 다음과 같이 0~4번에 해당하는 유가증권시장의 종목 데이터가 출력되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch11/08.py
로그인 성공
0 동화약품 000020 KR7000020008 0
1 우리은행 000030 KR7000030007 0
2 KR모터스 000040 KR7000040006 0
3 경방 000050 KR7000050005 0
4 메리츠화재 000060 KR7000060004 0

Process finished with exit code 0
```

차트 데이터 받아오기

시스템 트레이딩에서는 많은 사람이 일봉 차트 데이터를 사용합니다. 왜냐하면 틱이나 분 단위로는 과거 데이터를 무료로 내려받기가 쉽지 않으며, 데이터 용량이 커서 개인적으로 해당 데이터를 관리하기도 쉽지 않기 때문입니다.

이번 절에서는 xingAPI를 이용해 일봉 차트를 구성하는 주요 데이터인 일자, 시가, 고가, 저가, 종가 데이터를 가져와 보겠습니다.

먼저 그림 11.28과 같이 주식 차트와 관련된 TR 코드가 몇 개 존재하는 것을 DevCenter에서 확인할 수 있습니다. 일 단위의 데이터를 가져오려면 t8413 TR 코드를 사용하면 됩니다.

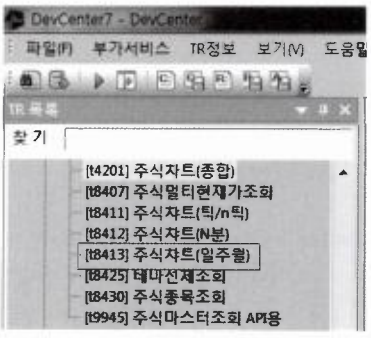


그림 11.28 xingAPI 주식 차트 관련 TR 코드

t8413 TR 코드를 사용하는 방법도 앞서 설명해 드린 것과 같습니다. 다만 그림 11.29를 참조하면 t8413InBlock 블록에 다수의 필드명이 존재하는 것을 확인할 수 있습니다. 따라서 SetFieldData 메서드를 이용해 입력 데이터를 설정할 때 각 필드 단위로 데이터를 설정해야 합니다.

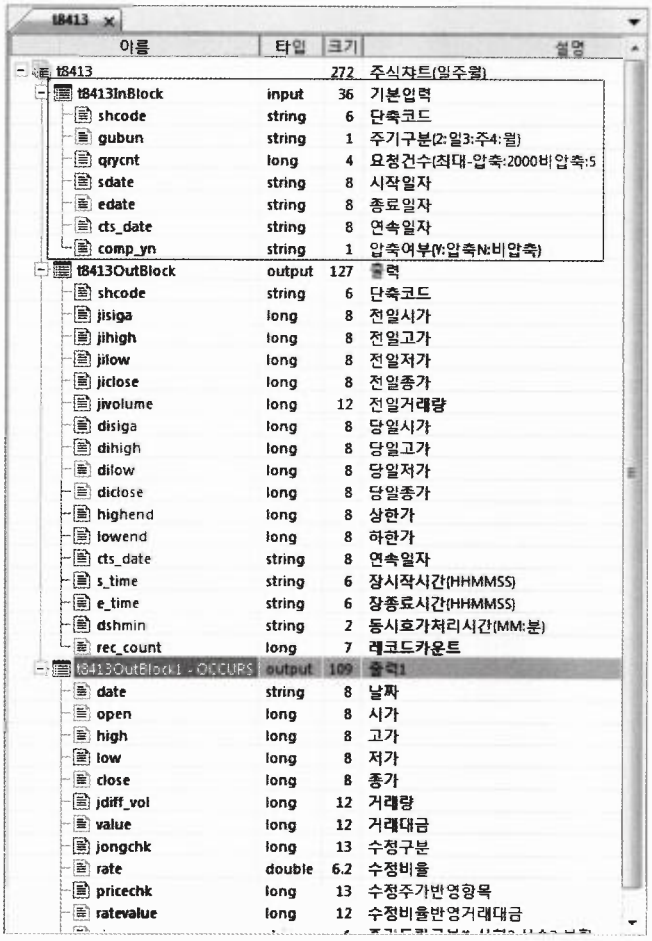


그림 11.29 t8413 TR 코드

예를 들어, 이베스트투자증권 종목에 대해 2016-01-11~2016-01-22 기간의 일봉 데이터를 받아오려면 다음과 같이 코드를 작성하면 됩니다.

```
instXAQueryT8413.SetFieldData("t8413InBlock", "shcode", 0, "078020")
instXAQueryT8413.SetFieldData("t8413InBlock", "gubun", 0, "2")
instXAQueryT8413.SetFieldData("t8413InBlock", "sdate", 0, "20160111")
instXAQueryT8413.SetFieldData("t8413InBlock", "edate", 0, "20160122")
instXAQueryT8413.SetFieldData("t8413InBlock", "comp_yn", 0, "N")
```

그림 11.29를 참조하면 t8413 TR 코드에는 t8413OutBlock과 t8413OutBlock1이라는 두 가지 종류의 출력 데이터 블록명이 존재하는 것을 확인할 수 있습니다. 이 가운데 t8413OutBlock1은 Occurs 속성을 가진 반복 데이터입니다. 따라서 먼저 데이터의 총 개수를 구한 후 각 인덱스에 해당하는 데이터를 가져오면 됩니다.

```
count = instXAQueryT8413.GetBlockCount("t8413OutBlock1")
```

예를 들어, 2016-01-11~2016-01-22 기간에는 총 10일의 주식 거래일이 존재하므로 데이터의 총 개수는 10개가 됩니다. 따라서 0~9까지의 각 인덱스는 특정 거래일을 의미하며, 해당 거래일의 날짜, 시가, 고가, 저가, 종가를 다음과 같은 코드로 구할 수 있습니다.

```
for i in range(count):
    date = instXAQueryT8413.GetFieldData("t8413OutBlock1", "date", i)
    open = instXAQueryT8413.GetFieldData("t8413OutBlock1", "open", i)
    high = instXAQueryT8413.GetFieldData("t8413OutBlock1", "high", i)
    low = instXAQueryT8413.GetFieldData("t8413OutBlock1", "low", i)
    close = instXAQueryT8413.GetFieldData("t8413OutBlock1", "close", i)
    print(date, open, high, low, close)
```

예제 11.5는 t8413 TR 코드를 이용해 일봉 데이터를 구하는 전체 코드입니다.

예제 11.5 t8413 TR 코드를 이용한 일봉 데이터 얻기(소스코드: book/ch11/09.py)

```
import win32com.client
import pythoncom

class XASessionEventHandler:
    login_state = 0

    def OnLogin(self, code, msg):
        if code == "0000":
            print("로그인 성공")
            XASessionEventHandler.login_state = 1
        else:
            print("로그인 실패")

class XAQueryEventHandlerT8413:
    query_state = 0

    def OnReceiveData(self, code):
        XAQueryEventHandlerT8413.query_state = 1
```

```
# ------------------------------------------------------------------------------
# login
# ------------------------------------------------------------------------------
id = "id"
passwd = "passwd"
cert_passwd = "cert"

instXASession = win32com.client.DispatchWithEvents("XA_Session.XASession", XASessionEventHan-
dler)
instXASession.ConnectServer("hts.ebestsec.co.kr", 20001)
instXASession.Login(id, passwd, cert_passwd, 0, 0)

while XASessionEventHandler.login_state == 0:
    pythoncom.PumpWaitingMessages()


# ------------------------------------------------------------------------------
# T8413
# ------------------------------------------------------------------------------
instXAQueryT8413 = win32com.client.DispatchWithEvents("XA_DataSet.XAQuery", XAQueryEventHan-
dlerT8413)
instXAQueryT8413.ResFileName = "C:\\eBEST\\xingAPI\\Res\\t8413.res"

instXAQueryT8413.SetFieldData("t8413InBlock", "shcode", 0, "078020")
instXAQueryT8413.SetFieldData("t8413InBlock", "gubun", 0, "2")
instXAQueryT8413.SetFieldData("t8413InBlock", "sdate", 0, "20160111")
instXAQueryT8413.SetFieldData("t8413InBlock", "edate", 0, "20160122")
instXAQueryT8413.SetFieldData("t8413InBlock", "comp_yn", 0, "N")

instXAQueryT8413.Request(0)

while XAQueryEventHandlerT8413.query_state == 0:
    pythoncom.PumpWaitingMessages()

count = instXAQueryT8413.GetBlockCount("t8413OutBlock1")
for i in range(count):
    date = instXAQueryT8413.GetFieldData("t8413OutBlock1", "date", i)
    open = instXAQueryT8413.GetFieldData("t8413OutBlock1", "open", i)
    high = instXAQueryT8413.GetFieldData("t8413OutBlock1", "high", i)
    low = instXAQueryT8413.GetFieldData("t8413OutBlock1", "low", i)
    close = instXAQueryT8413.GetFieldData("t8413OutBlock1", "close", i)
    print(date, open, high, low, close)
```

예제 11.5에서 사용자 계정 정보를 설정하는 코드 부분을 수정한 후 실행하면 다음과 같이 요청한 거래 기간의 일봉 데이터를 가져오는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch11/09.py
로그인 성공
20160111 9810 9860 9700 9800
20160112 9670 9990 9670 9750
20160113 9790 9840 9790 9800
20160114 9560 9810 9560 9690
20160115 9820 9820 9590 9590
20160118 9590 9590 9450 9450
20160119 9270 9320 9240 9240
20160120 9570 9570 9020 9080
20160121 9100 9260 9050 9050
20160122 9020 9200 8880 9060

Process finished with exit code 0
```

다음 증권 홈페이지를 통해 같은 기간의 이베스트투자증권의 일봉 데이터 값을 살펴보면 xingAPI를 이용해 가져온 값과 같음을 확인할 수 있습니다.

일자별 주가 | 일봉차트 | 자세히▸

일자	시가	고가	저가	종가	전일비	등락률	거래량
16.01.22	9,020	9,200	8,880	9,060	▲10	+0.11%	25,205
16.01.21	9,100	9,260	9,050	9,050	▼30	-0.33%	3,054
16.01.20	9,570	9,570	9,020	9,080	▼160	-1.73%	9,891
16.01.19	9,270	9,320	9,240	9,240	▼210	-2.22%	5,267
16.01.18	9,590	9,590	9,450	9,450	▼140	-1.46%	3,217
16.01.15	9,820	9,820	9,590	9,590	▼100	-1.03%	3,845
16.01.14	9,560	9,810	9,560	9,690	▼110	-1.12%	6,516
16.01.13	9,790	9,840	9,790	9,800	▲50	+0.51%	790
16.01.12	9,670	9,990	9,670	9,750	▼50	-0.51%	1,522
16.01.11	9,810	9,860	9,700	9,800	-0	0.00%	2,458

그림 11.30 이베스트투자증권 일봉 데이터(다음 증권, 2016/01/11~2016/01/22)

12

키움증권 API

이번 장에서는 파이썬으로 키움증권의 Open API+를 사용하는 방법을 알아보겠습니다. 주식 투자를 해보신 분들은 키움증권에 대해 한 번쯤은 들어봤을 것입니다. 현재 국내에는 약 60여 개 이상의 증권사가 있지만 그중 키움증권은 낮은 수수료 덕분에 대한민국 주식시장 점유율 12년 연속 1위(거래 대금 기준)를 지키고 있는 증권사입니다.

이 책의 집필을 시작하던 시점에는 대신증권이나 이베스트투자증권과 달리 키움증권에서는 API 기반의 서비스를 제공하지 않았습니다. 사실 자신의 주거래 증권사가 있는 상태에서 오직 알고리즘 트레이딩을 배우기 위해서 증권사를 옮기고 은행을 찾아다니면서 새롭게 계좌를 개설하는 것은 여간 번거로운 일이 아닙니다.

그런데 키움증권이 2014년 11월 17일부터 드디어 '키움 Open API'라는 이름으로 API 서비스를 시작했고, 최근에는 좀 더 향상된 API 서비스인 '키움 Open API+'를 새롭게 오픈했습니다. 국내 주식시장 점유율 1위의 증권사가 API 서비스를 시작했다는 점만으로도 국내의 알고리즘 트레이딩 시장에 주는 효과는 분명히 다를 것입니다.

01 개발 환경 구축

키움증권의 Open API+도 타 증권사의 API 서비스와 마찬가지로 사용하기 전에 개발 환경을 구축하는 과정이 필요합니다. 먼저 가까운 은행을 방문해서 키움증권 계좌를 개설하고 키움증권 홈페이지를 통해 API 사용 신청을 한 후 모듈 및 번개라는 이름의 HTS를 설치하면 됩니다.

계좌 개설 및 모듈 설치

키움증권의 Open API+를 이용하려면 앞서 설명한 것처럼 키움증권 계좌가 필요합니다. 참고로 키움증권도 타 증권사와 마찬가지로 전국의 15개 은행과 업무 제휴를 맺어 은행을 통해 계좌를 개설할 수 있습니다. 최근에는 키움증권도 비대면 계좌 개설을 지원하기 때문에 영업점이나 은행에 방문하지 않고도 계좌 개설이 가능합니다.

그림 12.1은 키움증권 홈페이지(http://www.kiwoom.com)에 나온 계좌 개설 제휴은행 목록입니다. 계좌 개설이 필요하다면 이 가운데 한곳을 방문한 후 키움증권의 계좌를 개설해 달라고 요청하면 됩니다. 참고로 계좌를 개설할 때는 반드시 신원을 증명할 수 있는 신분증을 지참해야 합니다.

▌계좌개설 가능은행

▸ 은행별 로고를 클릭하시면 필요서류 등 자세한 사항을 확인하실 수 있습니다.

	주식계좌	선물옵션	펀드계좌	해외주식	FX마진	해외선물
우정사업본부	O	O	O	-	-	-
KB 국민	O	O	O	O	O	O
신한은행	O	O	-	-	O	O
우리은행	O	O	O	O	O	O
농협	O	-	O	-	-	-
IBK	O	O	O	O	-	-
하나은행	O	O	O	O	O	O
citibank	O	O	O	-	-	-
대구은행	O	O	-	-	-	-
KEB	O	O	-	-	-	-
광주은행	O	O	-	-	-	-
경남은행	O	O	O	-	-	-
부산은행	O	O	-	-	-	-
새마을금고	O	O	O	-	-	-
Standard Chartered	O	O	O	-	-	-
신협	O	O	O	O	-	O

그림 12.1 키움증권 계좌 개설 제휴은행

키움증권의 계좌 개설이 완료되면 PC에 모듈 및 번개라는 이름의 HTS를 설치해야 합니다. 이를 위해 먼저 키움증권 홈페이지(https://www.kiwoom.com/)로 이동한 후 그림 12.2와 같이 웹 페이지 하단의 [Open API] 메뉴를 클릭합니다.

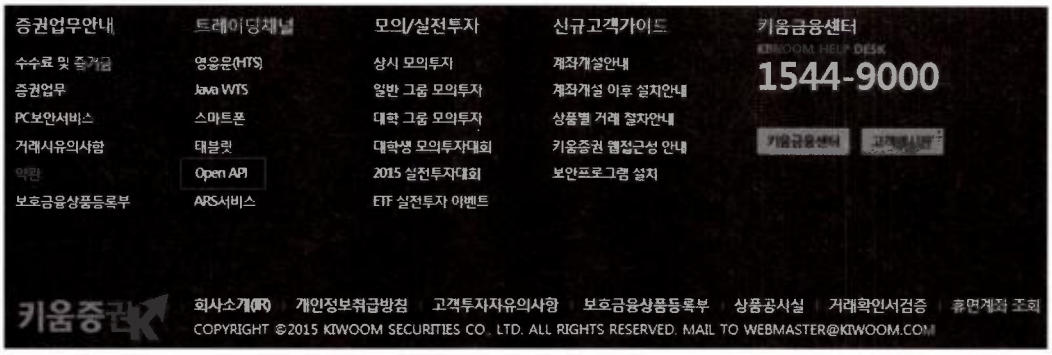


그림 12.2 키움증권 Open API

키움증권의 Open API+를 사용하려면 먼저 API 사용 신청을 해야 합니다. 그림 12.3과 같이 [사용 신청하러 가기] 메뉴를 클릭해 서비스 사용 등록을 진행합니다.

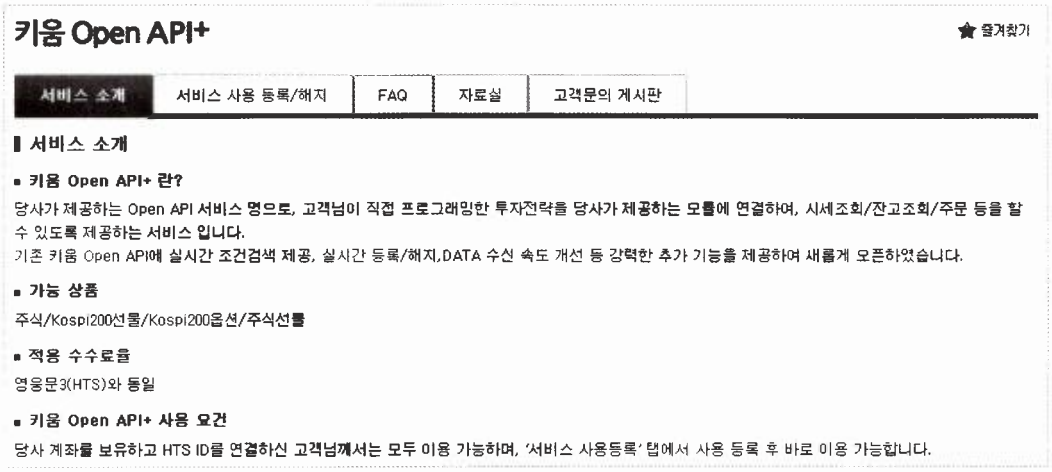




그림 12.3 API 사용 신청하기

그림 12.2에서 [영웅문(HTS)]을 클릭한 후 홈트레이딩시스템(HTS) 화면에서 번개3을 다운로드합니다. 참고로 키움증권에는 영웅문이라는 이름의 대표 HTS도 있는데, 번개는 영웅문에서 핵심 기능만 모아 둔 가벼운 버전의 HTS입니다.

내려받은 키움 번개 설치 파일을 클릭해 프로그램을 설치합니다. 그림 12.4에서 [다음] 버튼을 눌러 설치를 시작합니다.



그림 12.4 키움번개 설치 1

기본 설치 경로를 변경하지 말고 [다음] 버튼을 눌러 설치를 계속합니다.

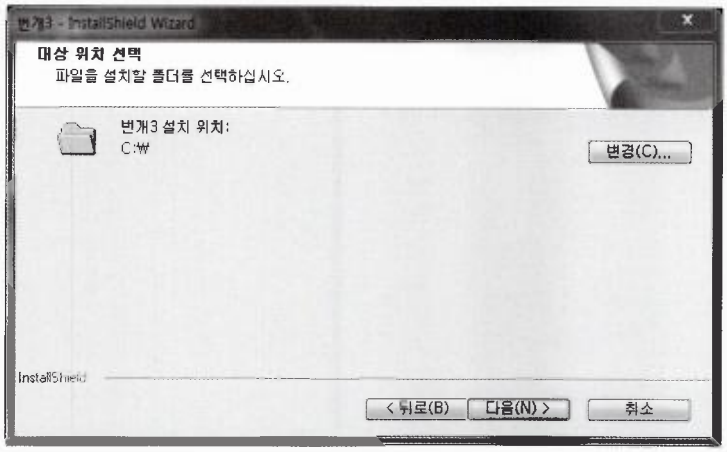


그림 12.5 키움번개 설치 2

그림 12.6에서 [설치] 버튼을 눌러 설치를 시작합니다.

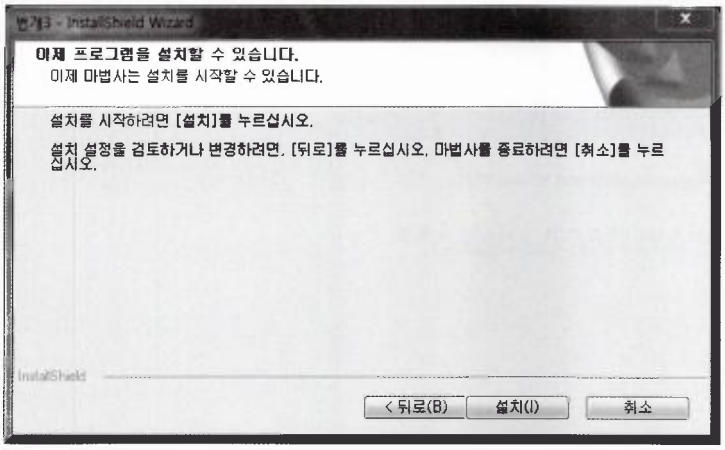


그림 12.6 키움번개 설치 3

설치가 완료되면 그림 12.7과 같이 [완료] 버튼을 눌러 설치를 종료합니다.



그림 12.7 키움번개 설치 4

키움번개를 설치했다면 그림 12.3을 참고해서 [키움 Open API+ 모듈 다운로드] 버튼을 클릭해 모듈 설치 파일을 다운로드합니다. 내려받은 파일을 더블클릭하면 그림 12.8과 같이 설치 마법사가 나타납니다. [다음] 버튼을 눌러 설치를 시작합니다.



그림 12.8 키움 Open API+ 모듈 설치 1

대상 위치 선택 화면에서 [다음] 버튼을 누른 후 그림 12.9 화면에서 [설치] 버튼을 누르면 설치가 시작됩니다.

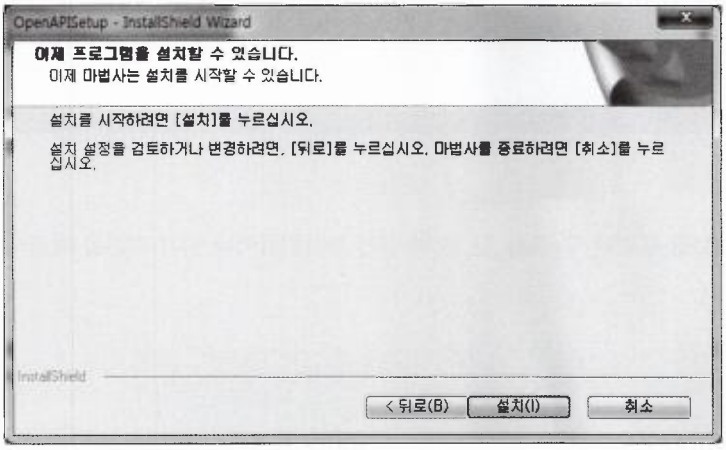


그림 12.9 키움 Open API+ 모듈 설치 2

모든 설치가 완료되면 [완료] 버튼을 눌러 설치 과정을 종료합니다.



그림 12.10 키움 Open API+ 모듈 설치 3

모의투자 가입하기

키움증권도 이베스트투자증권이나 대신증권과 마찬가지로 모의투자 서비스를 제공합니다. 모의투자 서비스를 이용하기 위해 그림 12.11과 같이 모의/실전투자 메뉴에서 [상시 모의투자]를 클릭합니다.

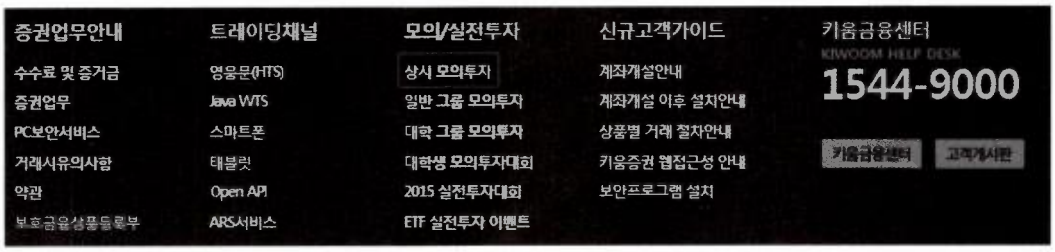


그림 12.11 모의투자신청

상시 모의투자에서 [참가신청] 메뉴를 클릭한 후 그림 12.12와 같이 투자원금이나 참가 기간을 적절히 입력한 후 참가 신청을 완료합니다.

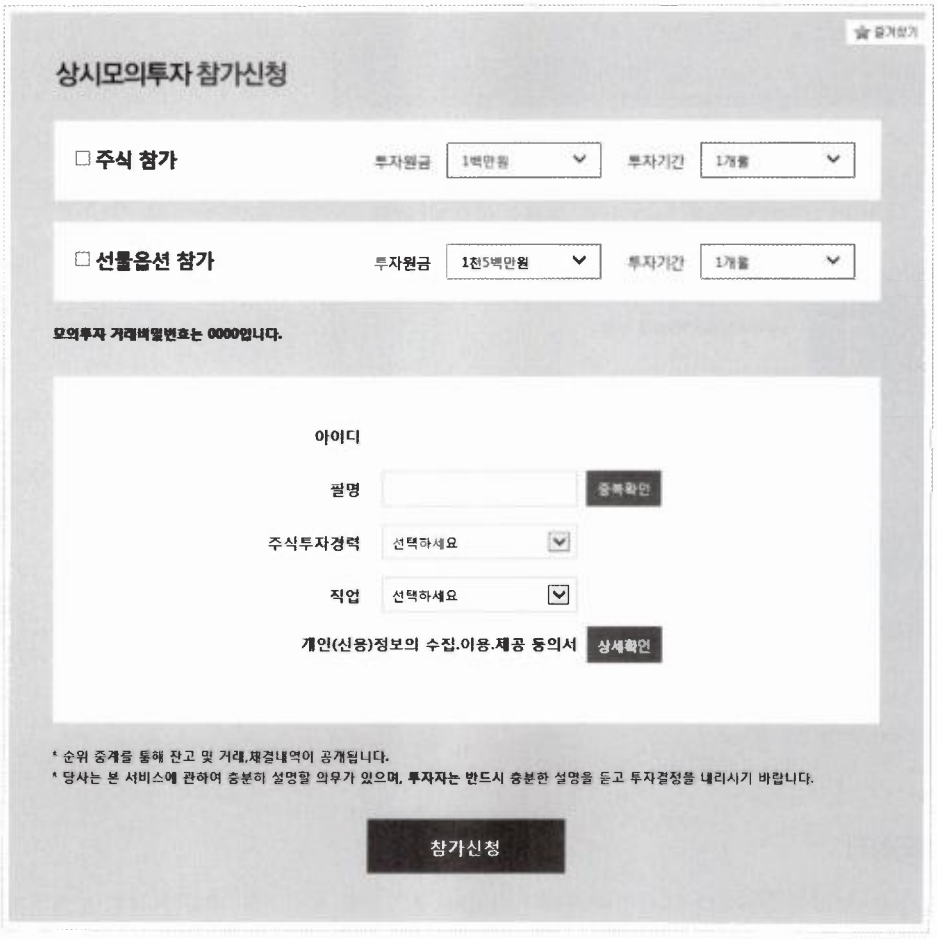


그림 12.12 모의투자 참가 신청

KOA Studio 사용하기

키움증권에서는 Open API+를 이용한 프로그램의 개발 편의성 향상을 위해 개발 가이드 및 KOA Studio를 제공합니다. KOA Studio는 이베스트투자증권의 DevCenter와 비슷한 역할을 하는 프로그램입니다.

그림 12.3에서 [KOA Studio 다운로드] 버튼을 클릭해 KOA Studio 압축 파일을 내려받습니다. 내려받은 파일의 압축을 풀면 그림 12.13과 같이 KOAStudioSA라는 실행 파일과 KOALader.dll이라는 파일을 볼 수 있습니다.

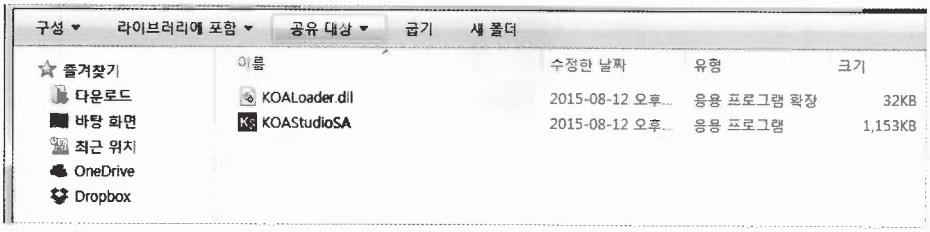


그림 12.13 KOAStudio 압축 해제

KOAStudioSA 파일을 클릭해 KOA Studio를 실행합니다. [파일] → [Open API 접속] 메뉴를 차례로 클릭해 그림 12.14와 같은 로그인 창이 표시되면 ID와 비밀번호를 입력한 후 로그인을 수행합니다.



그림 12.14 KOA Studio 로그인

KOA Studio에서 처음 로그인하면 그림 12.15와 같이 버전처리 메시지 창이 나타납니다. [확인] 버튼을 누르기 전에 먼저 KOA Studio를 종료한 후 메시지 창의 [확인] 버튼을 클릭해야 정상적으로 버전처리가 수행됩니다.

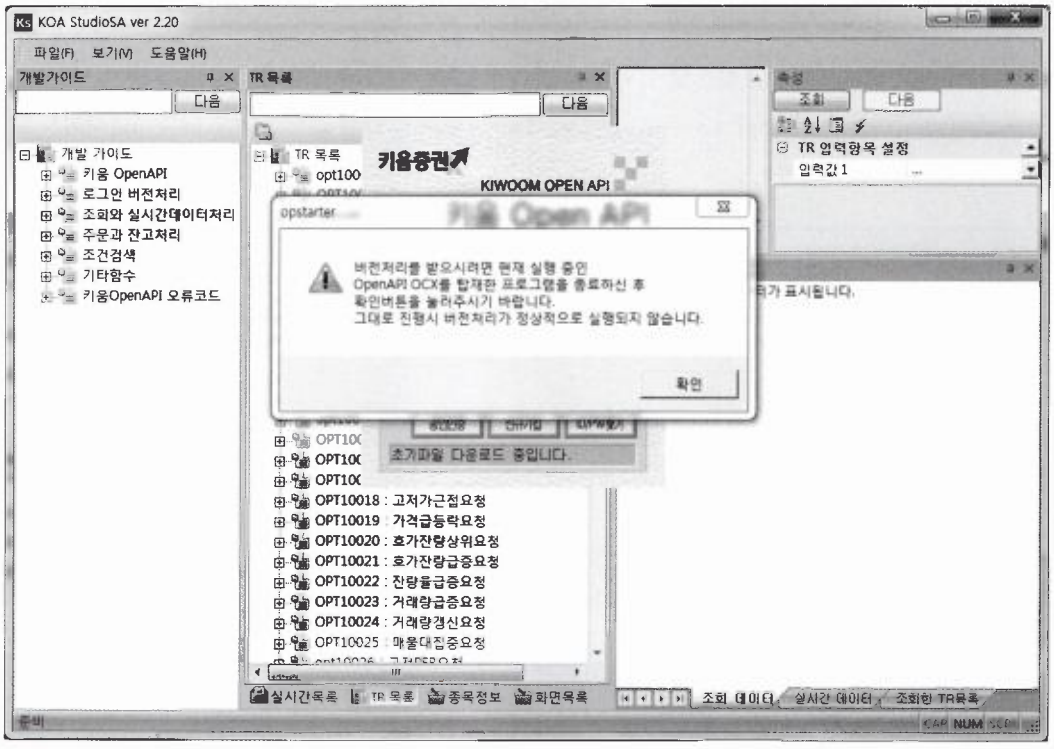


그림 12.15 버전처리 메시지 창

그림 12.16과 같이 정상적으로 버전 처리가 진행됩니다.

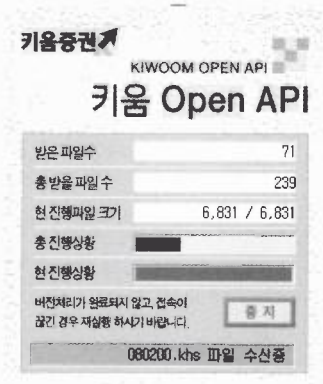


그림 12.16 버전 처리 과정

버전 처리가 완료되면 다시 KOA Studio를 실행한 후 [파일] → [Open API 접속] 메뉴를 차례로 클릭해 로그인합니다. 로그인이 정상적으로 이뤄지면 그림 12.17과 같이 KOA Studio 하단의 TR 목록을 선택한 후 TR 목록 창에서 opt10001 TR을 선택합니다. 종목코드에 '039490'을 입력한 후 [조회] 버튼을 클릭하면 조회 데이터 부분에 조회 결과가 출력되는 것을 확인할 수 있습니다.

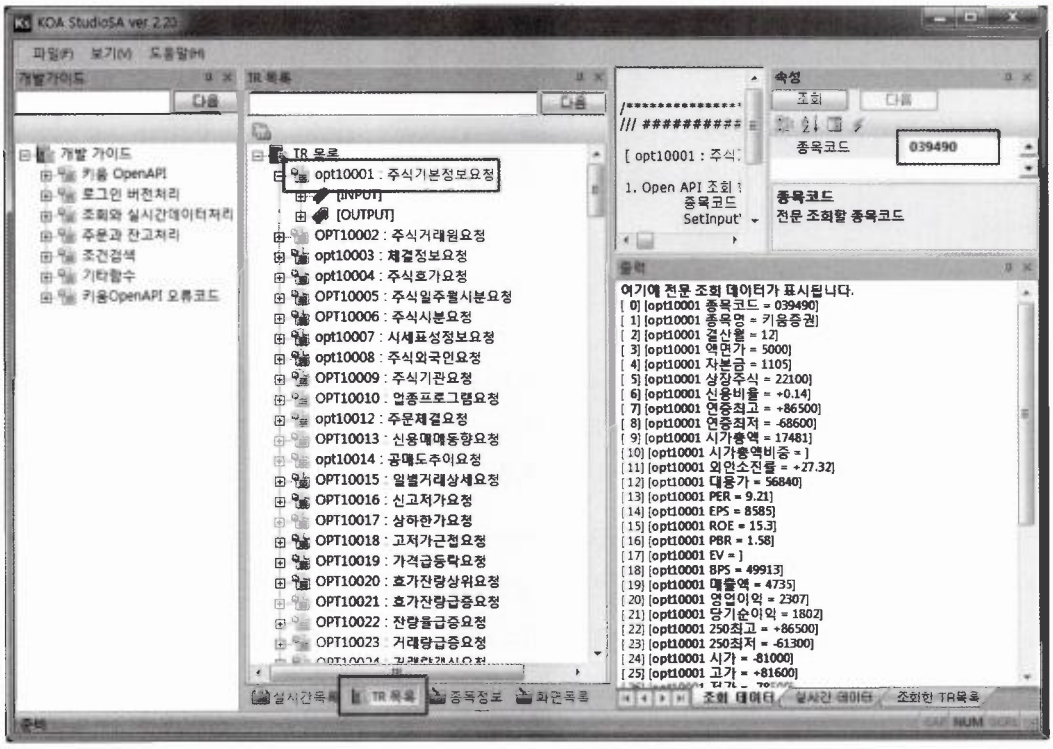


그림 12.17 KOA Studio를 이용한 데이터 조회

앞으로 KOA Studio는 이베스트투자증권의 DevCenter와 마찬가지로 API의 입/출력 값을 확인하거나 API의 동작을 확인하는 용도로 사용하면 되겠습니다.

02 PyQt 기초

xingAPI나 CYBOS Plus가 COM(Component Object Model) 방식을 사용하는 것과 달리 키움 Open API+는 OCX(Object Linking and Embedding Custom Control) 방식을 사용합니다. 기본적으로 COM과 OCX는 모두 마이크로소프트에서 만든 기술로서 매우 유사합니다. 그러나 OCX 방식은 COM에 비해 파이썬에서 사용하기 쉽지 않다는 문제가 있습니다.

이 책에서는 이러한 문제를 해결하기 위해 PyQt 패키지의 `QAxContainer` 모듈을 통해 OCX를 사용합니다. 이를 위해 이 책의 후반부에서 배울 PyQt의 내용 중 기초적인 부분을 이번 절에서 간단히 다루겠습니다.

PyQt 소개

PyQt 소개에 앞서 먼저 GUI(Graphical User Interface)라는 용어에 대해 알아보겠습니다. 지금까지 파이썬을 통해 만든 모든 프로그램은 TUI(Text-based User Interface) 기반으로 동작했습니다. 여기서 TUI란 검정색 콘솔 창에서 텍스트 기반으로 프로그램이 실행되는 것을 의미합니다.

이와 달리 GUI란 버튼이나 윈도우 창과 같은 그래픽 요소를 통해 사용자와 컴퓨터 간에 인터페이스를 구현한 방식입니다. 그림 12.18과 같이 많이 사용 중인 HTS가 바로 대표적인 GUI 방식의 프로그램이라 할 수 있습니다. 참고로 PyQt를 이용하면 그림 12.18과 같은 GUI 프로그램을 개발할 수 있습니다.



그림 12.18 키움 번개를 실행한 모습

파이썬으로 GUI 프로그래밍을 하려면 wxPython[1], PyQt[2], TkInter 등과 같은 패키지를 사용합니다. 그중 TkInter는 파이썬의 공식적인 GUI 패키지로서 별도의 프로그램을 설치하지 않고도 바로 사용할 수 있습니다. 예를 들어, 파이썬 IDLE를 실행한 후 그림 12.19와 같이 두 줄의 코드만 작성하면 화면에 윈도우가 나타나는 것을 확인할 수 있습니다.

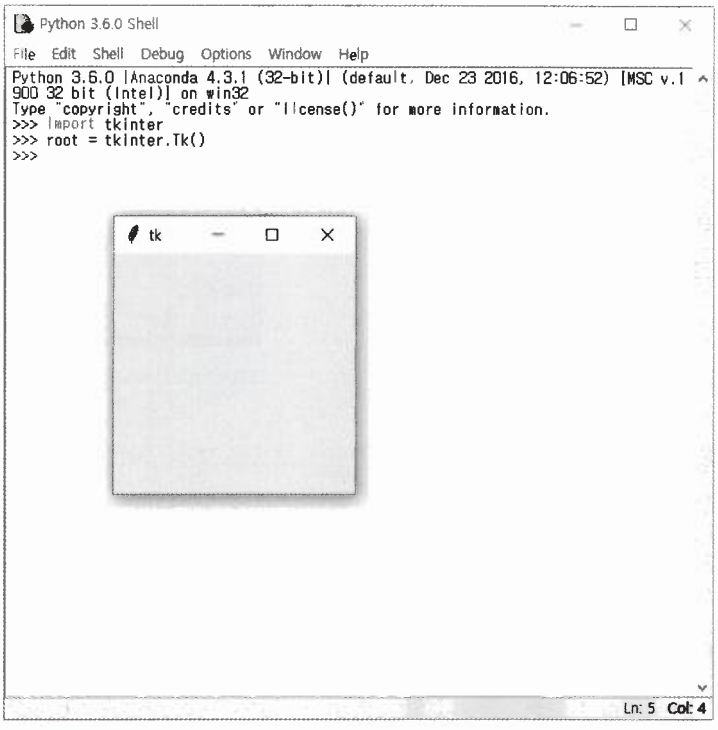


그림 12.19 TkInter를 이용한 윈도우 출력

이처럼 TkInter는 추가로 패키지를 설치할 필요가 없고 간단하며 배우기 쉽지만, 인터페이스가 다소 구식이고 복잡한 프로그램을 개발하기에는 몇 가지 부족한 면이 있습니다. 이 때문에 TkInter보다는 PyQt나 wxPython을 많이 사용합니다. 그중 이 책에서는 PyQt를 이용해 GUI 프로그램을 개발할 예정입니다.

PyQt는 이름에서 알 수 있듯이 Qt라는 GUI 프레임워크의 파이썬 바인딩입니다. Qt는 GUI 프로그램 개발에 널리 쓰이는 크로스 플랫폼 프레임워크로서 주로 C++라는 프로그래밍 언어를 사용해 프로그래밍합니다.

1 http://www.wxpython.org/

2 https://riverbankcomputing.com/

여기서 크로스 플랫폼 프레임워크란 윈도우나 리눅스와 같은 운영체제에 상관없이 같은 코드로 각 운영체제에서 동작하는 프로그램 개발을 지원하는 것을 의미합니다. 만일 크로스 플랫폼을 지원하지 않는 GUI 프레임워크를 사용한다면 윈도우, 리눅스, Mac OS X을 위한 코드를 별도로 작성해야 합니다.

현재 공식적으로 지원하는 Qt 메이저 버전은 Qt4와 Qt5입니다. 그래서 Qt의 파이썬 바인딩인 PyQt도 PyQt4와 PyQt5로 두 가지 버전이 제공됩니다. 이 책에서는 아나콘다 배포판에서 공식적으로 지원하는 PyQt5를 사용합니다.

Hello PyQt

이번 절에서는 PyQt를 이용해 그림 12.20과 같이 'Hello PyQt'라고 적힌 윈도우 창을 하나 출력하는 프로그램을 만들어보겠습니다.



그림 12.20 Hello PyQt

먼저 PyCharm을 실행한 후 예제 12.1의 코드를 입력해 보기 바랍니다. 코드를 모두 입력했다면 Ctrl + F9와 같은 단축키를 이용해 작성된 코드를 실행해 봅니다. 작성한 코드에 오타가 없다면 그림 12.20과 같은 창이 화면에 나타날 것입니다.

예제 12.1 Hello PyQt 코드(소스코드: book/ch12/01.py)

```
import sys
from PyQt5.QtWidgets import *

app = QApplication(sys.argv)
label = QLabel("Hello PyQt")
label.show()
app.exec_()
```

예제 12.1을 살펴보면 공백을 포함해서 단 7줄의 코드로 윈도우를 만들고 그 안에 'Hello PyQt'라는 문자열을 출력했습니다. 특히 처음 두 줄은 모듈을 임포트하는 구문으로, 실제로 윈도우를 생성하는 코드는 딱 4줄입니다. 어떤가요? GUI 프로그래밍이 생각보다 어렵지 않지요?

일단 프로그램의 정상적인 실행 결과를 확인했다면 예제 12.1을 자세히 분석해 보겠습니다. 예제 12.1에서 첫 두 줄은 모듈을 임포트하는 역할을 합니다. 이때 첫 번째 줄과 두 번째 줄은 모듈을 임포트하는 방식이 조금 다릅니다.

5장에서 다뤘듯이 'import 모듈명'과 같은 방식으로 모듈을 임포트했다면 모듈(파이썬 파일) 내에 정의된 변수, 함수, 클래스에 접근할 때 '모듈명.변수'와 같이 모듈 이름을 명시해야 합니다.

이와 달리 'from PyQt5.QtWidgets import *' 같은 방식은 PyQt5라는 디렉터리의 QtWidgets 파일에 있는 모든 것(*)을 임포트하라는 의미이므로 해당 모듈(QtCore)에 정의된 변수, 함수, 클래스를 모듈 이름을 통해 접근할 필요 없이 바로 사용할 수 있습니다.

참고로 그림 12.21을 보면 PyQt5.QtWidgets는 QtWidgets.pyd라는 이름으로 'C:\Anaconda3\Lib\site-packages\PyQt5' 디렉터리에 위치합니다. 파이썬 모듈은 파이썬 코드(*.py)뿐 아니라 *.pyd 파일의 형태도 사용될 수 있는데 여기서 *.pyd 파일은 윈도우의 *.dll과 같습니다. 이처럼 일부 모듈은 성능을 위해 파이썬으로 모듈을 구현하는 것이 아니라 C라는 프로그래밍 언어로 구현하고 파이썬에서 C 모듈을 임포트해서 사용하기도 합니다.

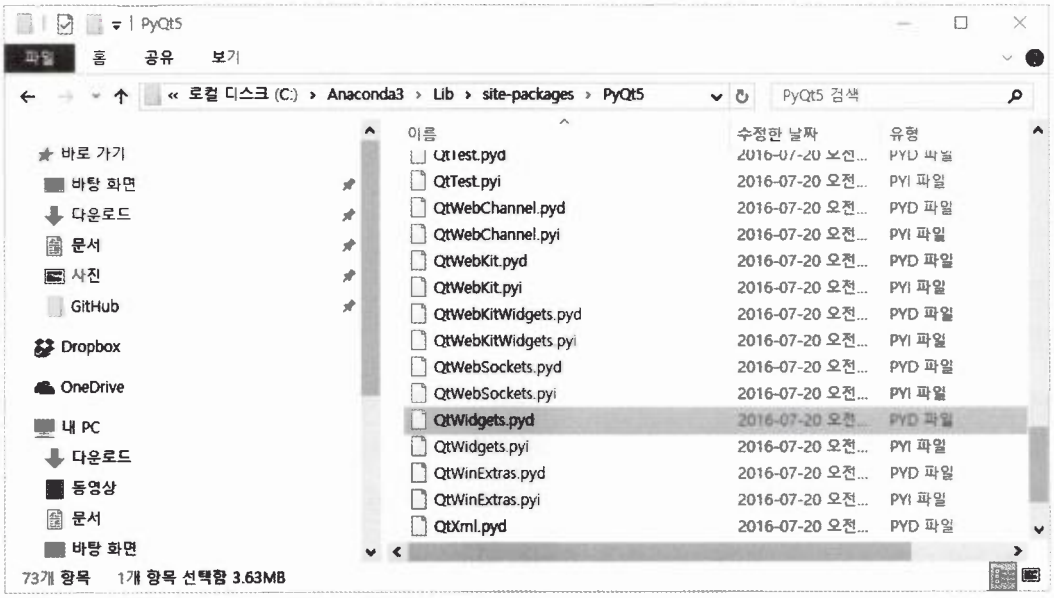


그림 12.21 QtGui.pyd 파일의 위치

예제 12.1에서 윈도우 생성과 직접 관련된 네 줄의 코드에 대한 실행 흐름을 그림으로 나타내면 그림 12.22와 같습니다. 먼저 QApplication 클래스에 대한 인스턴스를 생성하고 이를 app이라는 변수로 바인딩합니다. app을 통해 exec_ 메서드를 호출하면 프로그램은 이벤트 루프(event loop)에 진입합니다.

이벤트 루프란 무한 반복하면서 이벤트를 처리하는 상태를 의미합니다. 앞서 여러 번 설명했듯이 파이썬 코드는 위에서 밑으로 순서대로 실행된 후 프로그램이 종료됩니다. 따라서 예제 12.1의 코드가 실행되면 그림 12.20과 같이 윈도우가 화면에 계속 출력됩니다.

앞에서 작성한 대부분의 TUI 기반의 프로그램은 프로그램이 실행된 후 종료됐지만 그림 12.20과 같은 GUI 프로그램은 사용자가 윈도우를 닫기 전까지는 계속 실행된 상태로 남아 있습니다. 이처럼 프로그램이 종료되지 않고 실행 상태로 계속 남아 있을 수 있는 이유가 바로 이벤트 루프 때문입니다.

이벤트 루프는 기본적으로 무한 루프 구조이며, 무한 루프 상태에서 사용자가 발생한 이벤트를 그때그때 처리해줍니다. 예를 들어, 사용자가 윈도우를 클릭하면 '클릭 이벤트'가 발생하는데 이때 이벤트 루프에서 '클릭 이벤트'를 처리해주는 것입니다.

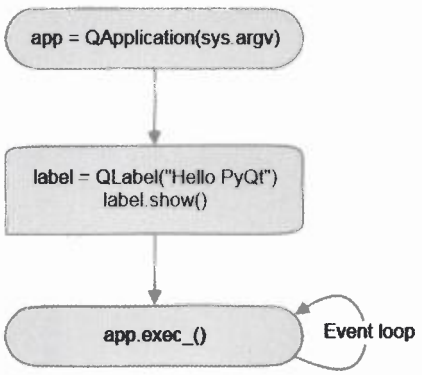


그림 12.22 PyQt 코드의 실행 흐름

그림 12.22에서 중간에 위치하는 네모 박스는 윈도우를 만들고 이를 화면에 보이게 하는 파이썬 코드로 구성돼 있습니다. QApplication 클래스의 인스턴스가 GUI 프로그램을 전반적으로 제어하는 역할을 한다면 네모 박스 부분은 UI 구성을 담당합니다.

예제 12.1에서는 네모 박스 위치에 해당하는 코드에서 QLabel 클래스의 인스턴스를 만들고 해당 인스턴스의 show 메서드를 호출함으로써 화면에 QLabel 위젯이 출력되는 것입니다. 만약 네모 박스 위치에 해당하는 코드를 수정하면 다른 형태의 윈도우가 출력되게 할 수 있습니다.

예를 들어, 다음 코드를 실행하면 그림 12.23과 같이 윈도우에 버튼이 하나 위치하는 프로그램이 나타납니다. 예제 12.1과 비교해보면 단순히 이벤트 루프가 실행되기 전에 윈도우 창을 구성하는 코드가 있다는 점을 확인할 수 있습니다.

```
import sys
from PyQt5.QtWidgets import *

app = QApplication(sys.argv)
label = QPushButton("Quit")
label.show()
app.exec_()
```



그림 12.23 QPushButton 위젯 출력

참고로 예제 12.1에서 sys.argv 값이 궁금하다면 sys.argv 값을 출력하는 코드를 추가한 후 코드를 다시 실행해보기 바랍니다.

```
import sys
from PyQt5.QtWidgets import *

app = QApplication(sys.argv)
print(sys.argv)
label = QLabel("Hello PyQt")
label.show()
app.exec_()
```

print 코드는 콘솔 출력이 되므로 생성된 윈도우에 출력되는 것이 아니라 PyCharm의 콘솔 윈도우에 출력됩니다. PyCharm 콘솔 윈도우에 출력된 sys.argv 값을 확인해보면 sys.argv 값은 현재 소스코드에 대한 절대 경로임을 확인할 수 있습니다. QApplication 클래스의 인스턴스를 생성할 때 생성자로 이 값을 전달해야 합니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch12/03.py
['E:/GitHub/book/ch12/03.py']

Process finished with exit code 0
```

위젯과 윈도우

PyQt에서 위젯(widget)은 사용자 인터페이스를 구성하는 가장 기본적인 부품 역할을 합니다. 예를 들어, 그림 12.24와 같은 사용자 인터페이스를 만들기 위해 QGroupBox, QLabel, QTextEdit, QDateEdit, QTimeEdit, QLineEdit와 같은 다양한 위젯이 사용됐습니다.

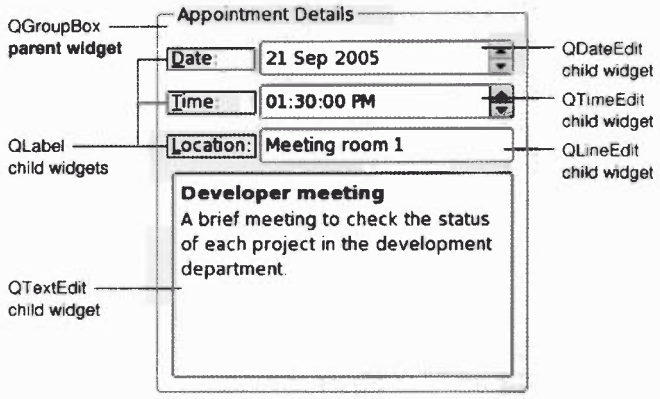


그림 12.24 PyQt 위젯[3]

그림 12.24를 보면 한 위젯은 다른 위젯에 포함될 수 있습니다. PyQt에서는 다른 위젯에 포함되지 않은 최상위 위젯을 특별히 윈도우(window)라고 합니다. 참고로 PyQt에서는 윈도우를 생성하기 위한 클래스로 QMainWindow나 QDialog 클래스가 일반적으로 사용됩니다.

이번 절에서는 QMainWindow 클래스를 사용해 그림 12.25와 같은 윈도우를 생성해보겠습니다.



그림 12.25 QMainWindow를 이용한 윈도우 생성

예제 12.2 QMainWindow를 이용한 윈도우 생성(소스코드: book/ch12/04.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("PyStock")
        self.setGeometry(300, 300, 300, 400)
```

3 출처: http://doc.qt.io/qt-4.8/qwidget.html

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

예제 12.2는 그림 12.25의 윈도우를 생성하는 전체 코드입니다. 예제 12.2를 살펴보면 크게 두 부분으로 구성된 것을 확인할 수 있습니다.

먼저 첫 번째 부분은 MyWindow라는 클래스를 정의하는 부분입니다. 두 번째 부분은 그림 12.22에서 설명한 것처럼 QApplication 클래스를 사용해 이벤트 루프를 만들고 화면에 출력될 객체를 생성하는 부분입니다.

MyWindow라는 클래스를 정의하는 부분을 살펴보면 MyWindow는 QMainWindow를 상속받는 것을 확인할 수 있습니다. 6장에서 배웠듯이 클래스를 상속하면 자식 클래스는 부모 클래스에 구현된 모든 속성이나 메서드를 그대로 사용할 수 있게 됩니다. 따라서 MyWindow 클래스는 PyQt가 제공하는 QMainWindow 클래스를 단지 상속함으로써 QMainWindow와 동등한 능력을 갖추게 되는 것입니다.

MyWindow 클래스의 생성자 부분을 살펴보면 먼저 super().__init__()을 호출하는데 이는 부모 클래스에 정의된 __init__()을 호출하는 것을 의미합니다. 이 예제에서는 MyWindow 클래스가 QMainWindow 클래스를 상속받았으므로 QMainWindow 클래스의 생성자인 __init__()을 호출하게 됩니다.

그렇다면 어떤 경우에 부모 클래스의 생성자를 호출하는 것이 필요할까요? 간단한 예제를 통해 확인해 보겠습니다.

Parent라는 부모 클래스를 상속하는 Child1, Child2 클래스를 정의해 봅시다. Child1 클래스의 생성자에서는 super 내장 함수를 이용해 부모 클래스의 생성자를 명시적으로 호출합니다. 이와 달리 Child2 클래스의 생성자에서는 아무것도 하지 않습니다.

```
class Parent:
    house = "yong-san"
    def __init__(self):
        self.money = 10000

class Child1(Parent):
    def __init__(self):
        super().__init__()
        pass
```

```
class Child2(Parent):
    def __init__(self):
        pass

child1 = Child1()
child2 = Child2()

print('Child1', dir(child1))
print('Child2', dir(child2))
```

정의된 Child1과 Child2 클래스의 인스턴스를 생성한 후 dir 내장 함수를 이용해 객체의 속성값을 확인해보면 child1과 child2 객체 모두 Parent 클래스의 클래스 변수인 house라는 속성값을 가진 것을 확인할 수 있습니다. 하지만 인스턴스 변수인 money는 Parent 클래스의 생성자를 명시적으로 호출했던 Child1 클래스의 인스턴스인 child1에만 존재하는 것을 확인할 수 있습니다.

```
Child1 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclas-
shook__', '__weakref__', 'house', 'money']

Child2 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclas-
shook__', '__weakref__', 'house']
```

간단히 super 내장 함수를 사용하는 것에 대해 살펴봤으니 계속해서 예제 12.2에서 윈도우 창을 생성하는 코드를 분석해 보겠습니다. 클래스에 정의된 메서드에서 self는 해당 클래스의 인스턴스를 의미합니다. setWindowTitle은 윈도우의 타이틀 창의 텍스트를 바꾸는 메서드고, setGeometry는 창의 위치 및 크기를 조절하는 메서드입니다.

```
self.setWindowTitle("PyStock")
self.setGeometry(300, 300, 300, 400)
```

윈도우 생성을 담당하는 MyWindow 클래스를 정의했다면 이번에는 이벤트 루프를 담당하는 코드를 살펴봅시다. 이벤트 루프는 QApplication 클래스의 인스턴스인 app을 생성하고 exec_ 메서드를 호출해서 생성할 수 있습니다. 이때 인스턴스의 생성과 exec_ 메서드 호출 사이에 생성될 윈도우를 만들면 됩니다.

MyWindow라는 클래스를 새롭게 정의했으므로 해당 클래스에 대한 인스턴스인 myWindow를 생성하고 show 메서드를 호출해 새롭게 정의한 윈도우가 화면에 출력될 수 있도록 설정합니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

이벤트 처리

GUI 프로그램은 기본적으로 사용자의 이벤트에 따라 동작합니다. 예를 들어, 그림 12.26의 윈도우에는 'Click me'라는 버튼이 하나 있습니다. 사용자가 해당 버튼을 마우스로 클릭하면 화면에 'clicked'라는 메시지 박스가 출력됩니다.

앞에서 작성한 PyQt 프로그램에는 UI와 관련된 코드만 있었고 사용자의 이벤트에 따라 이벤트를 처리하는 코드는 없었습니다. 이번 절에서는 PyQt를 이용해 사용자의 이벤트를 처리하는 방법을 알아보겠습니다.

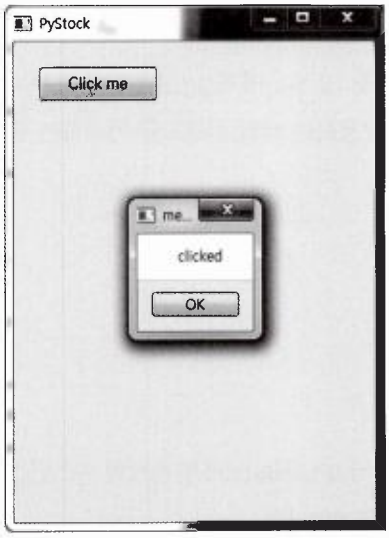


그림 12.26 PyQt 이벤트 처리

예제 12.3 PyQt 이벤트 처리(소스코드: book/ch12/06.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
```

```
class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("PyStock")
        self.setGeometry(300, 300, 300, 400)

        btn1 = QPushButton("Click me", self)
        btn1.move(20, 20)
        btn1.clicked.connect(self.btn1_clicked)

    def btn1_clicked(self):
        QMessageBox.about(self, "message", "clicked")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

예제 12.3은 그림 12.26에 나온 프로그램의 전체 코드입니다. 예제 12.3과 예제 12.2의 코드를 비교해보면 다음과 같이 윈도우에 버튼을 생성하는 부분과 버튼에서 발생하는 클릭 이벤트를 처리하는 함수가 추가된 것을 확인할 수 있습니다.

```
    btn1 = QPushButton("Click me", self)
    btn1.move(20, 20)
    btn1.clicked.connect(self.btn1_clicked)

def btn1_clicked(self):
    QMessageBox.about(self, "message", "clicked")
```

먼저 윈도우에 버튼을 추가하기 위해 MyWindow 클래스의 생성자에서 QPushButton 클래스의 인스턴스를 생성합니다. 그리고 버튼의 출력 위치를 move 메서드를 사용해 조절합니다.

```
btn1 = QPushButton("Click me", self)
btn1.move(20, 20)
```

사용자가 버튼을 클릭하면 'clicked'라는 이벤트가 발생합니다. 먼저 이벤트를 처리할 메서드를 구현합니다.

```
def btn1_clicked(self):
    QMessageBox.about(self, "message", "clicked")
```

이벤트를 처리하는 메서드를 구현했다면 이벤트와 이벤트를 처리할 메서드를 connect라는 메서드로 연결합니다. 해당 코드는 MyWindow 클래스의 생성자에 구현합니다.

```
def __init__(self):
    # 코드 생략
    btn1.clicked.connect(self.btn1_clicked)
```

위와 같이 이벤트와 이벤트 처리 함수를 연결해두면 버튼이 클릭되어 'clicked'라는 이벤트가 발생했을 때 QApplication의 이벤트 루프는 'clicked' 이벤트에 연결된 btn1_clicked 메서드를 호출합니다. 호출된 btn1_clicked 메서드는 QMessageBox 클래스를 사용해 화면에 간단히 메시지 박스를 표시합니다.

03 기초 API 익히기

지금까지 키움증권의 Open API+를 파이썬에서 사용하기 위해 먼저 PyQt의 기본적인 내용을 공부했습니다. 이번 절에서는 키움증권의 Open API+가 제공하는 기초 API를 파이썬에서 사용하는 방법을 알아보겠습니다.

Open API+ 로그인하기

키움증권의 Open API+를 사용하려면 맨 먼저 증권사 서버에 로그인해야 합니다. 앞서 여러 번 설명한 것처럼 파이썬에서 Open API+를 사용하려면 PyQt를 사용하는 것이 가장 편리한 방법입니다. 따라서 대신증권이나 이베스트투자증권의 API를 사용했던 것과 달리 GUI 기반으로 프로그램을 작성할 것입니다.

이번 절에서 만들어볼 프로그램은 그림 12.27과 같습니다. 그림 12.27을 살펴보면 윈도우에 두 개의 버튼이 있는데, 이 가운데 'Login' 버튼을 누르면 그림 12.28과 같이 키움증권이 제공하는 로그인 창이 화면에 표시됩니다.

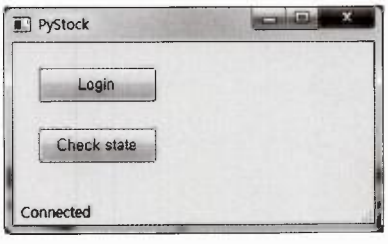


그림 12.27 Open API+ 로그인



그림 12.28 키움증권 로그인 창

그림 12.28의 로그인 창에 고객 정보를 입력한 후 [로그인] 버튼을 누르면 로그인됩니다. 이때 모의투자 서버에 접속하고자 한다면 [모의투자접속]이라는 체크박스를 체크한 후 [로그인] 버튼을 클릭하면 됩니다.

정상적으로 로그인됐는지 확인하려면 그림 12.27에서 [Check State]라는 버튼을 클릭하면 됩니다. 증권사 서버에 정상적으로 연결되면 그림 12.27과 같이 윈도우 하단에 'Connected'라는 문자열이 출력됩니다.

예제 12.4 OpenAPI+를 이용한 키움증권 로그인(소스코드: book/ch12/07.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QAxContainer import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("PyStock")
        self.setGeometry(300, 300, 300, 150)

        self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")

        btn1 = QPushButton("Login", self)
        btn1.move(20, 20)
        btn1.clicked.connect(self.btn1_clicked)
```

```
        btn2 = QPushButton("Check state", self)
        btn2.move(20, 70)
        btn2.clicked.connect(self.btn2_clicked)

    def btn1_clicked(self):
        ret = self.kiwoom.dynamicCall("CommConnect()")

    def btn2_clicked(self):
        if self.kiwoom.dynamicCall("GetConnectState()") == 0:
            self.statusBar().showMessage("Not connected")
        else:
            self.statusBar().showMessage("Connected")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

예제 12.4는 그림 12.27에 나온 프로그램의 전체 코드로서 기본 구성은 예제 12.3과 유사합니다. 파이썬에서 키움증권의 클래스를 사용하려면 PyQt의 QAxWidget 클래스를 사용해 인스턴스를 생성해야 합니다. 키움증권에서 제공하는 클래스는 각각 고유의 CLSID 또는 ProgID를 가지는데 해당 값을 QAxWidget 클래스의 생성자로 전달하면 인스턴스가 생성됩니다.

키움증권의 개발 가이드를 참조하면 CLSID는 {A1574A0D-6BFA-4BD7-9020-DED88711818D}임을 알 수 있고, 이를 윈도우 레지스트리 편집기를 통해 검색하면 ProgID는 그림 12.30과 같이 'KHOPENAPI.KHOpenAPICtrl.1'임을 알 수 있습니다.

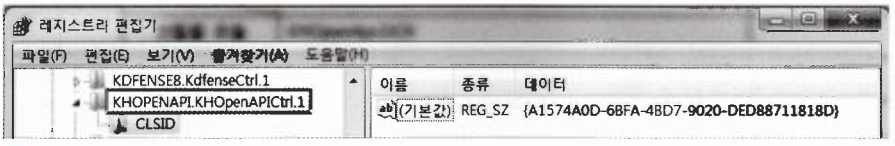


그림 12.29 키움증권 OpenAPI+ ProgID

참고로 대신증권과 이베스트투자증권의 클래스들에도 고유의 ProgID 값과 CLSID 값이 존재하며, 이 값들은 윈도우 운영체제의 레지스트리에 등록돼 있었습니다. CLSID는 16진수로 구성돼 있어 조금 복잡하기 때문에 파이썬에서 증권사 API에서 제공하는 클래스의 인스턴스를 생성할 때는 보통 문자열로 된 ProgID를 사용하는 것이 편리합니다.

키움증권에서 제공하는 클래스를 사용하기 위해 ProgID를 QAxWidget 클래스의 생성자로 전달해서 인스턴스를 생성합니다.

```
class MyWindow(QMainWindow):
    def __init__(self):
        # 코드 생략
        self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")
```

self.kiwoom은 QAxWidget 클래스의 인스턴스인데 QAxWidget 클래스는 그림 12.30과 같이 QWidget과 QAxBase라는 클래스를 상속받습니다. 따라서 여러분은 self.kiwoom을 통해 QWidget, QAxBase에 정의된 메서드를 사용할 수 있습니다.

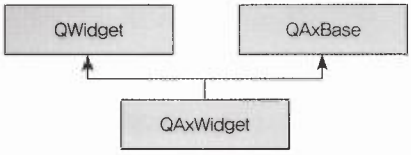


그림 12.30 QAxWidget Class 상속 구조

키움증권 개발 가이드를 참조하면 Open API+는 로그인 윈도우를 실행하는 CommConnect 메서드를 제공합니다. 즉, 인스턴스를 통해 CommConnect 메서드를 호출하기만 하면 로그인을 위한 윈도우가 자동으로 실행됩니다.

그림 12.31은 개발 가이드에서 CommConnect 메서드를 설명한 부분입니다. CommConnect 메서드는 인자를 받지 않으며, 반환 값을 통해 로그인 성공 여부를 확인할 수 있습니다. 앞서 설명한 것처럼 키움증권의 Open API+는 OCX 방식으로 개발됐기 때문에 파이썬에서 메서드를 호출하는 방식도 COM 방식과는 다릅니다.

1) CommConnect

원형	LONG CommConnect()
설명	로그인 윈도우를 실행한다.
입력값	없음
반환값	0 - 성공, 음수값은 실패
비고	로그인이 성공하거나 실패하는 경우 OnEventConnect 이벤트가 발생하고 이벤트의 인자값으로 로그인 성공 여부를 알 수 있다.

그림 12.31 CommConnect 메서드(개발 가이드)

COM 방식에서 인스턴스를 통해 메서드를 호출했던 것과 달리 OCX 방식에서는 QAxBase 클래스의 dynamicCall 메서드를 사용해 원하는 메서드를 호출할 수 있습니다. 그림 12.30과 같이 QAxWidget 클래스는 QAxBase 클래스를 상속받았으므로 QAxWidget 클래스의 인스턴스는 dynamicCall 메서드를 호출할 수 있습니다.

갑자기 PyQt의 클래스들이 많이 나와서 어려울 수 있지만 정리해보면 Open API+가 제공하는 메서드를 사용하려면 self.kiwoom 객체를 통해 dynamicCall 메서드를 호출해야 합니다. 이때 dynamicCall 메서드의 인자로 호출하려는 메서드를 전달합니다.

예를 들어, 로그인 윈도우를 실행하는 CommConnect 메서드를 파이썬에서 사용하려면 다음과 같이 코드를 작성하면 됩니다.

```
self.kiwoom.dynamicCall("CommConnect()")
```

그림 12.28의 로그인 윈도우에 계정 정보를 입력한 후 [로그인] 버튼을 클릭하면 키움증권 서버에 로그인됩니다. Open API+는 그림 12.32와 같이 현재 접속 상태를 반환하는 GetConnectState 메서드를 제공합니다. GetConnectState 메서드를 호출한 후 메서드의 리턴 값을 확인해 서버 접속 상태를 확인할 수 있습니다.

15) **GetConnectState**

원형	LONG GetConnectState()
설명	현재접속상태를 반환한다.
입력값	없음
반환값	접속상태
비고	0:미연결, 1:연결완료

그림 12.32 GetConnectState 메서드(개발 가이드)

파이썬 코드를 통해 GetConnectState 메서드를 호출하려면 역시 self.kiwoon 객체를 통해 dynamicCall 메서드를 호출하고 dynamicCall 메서드의 인자로 'GetConnectState()'를 전달하면 됩니다.

GetConnectState 메서드는 미연결 상태일 때 0을 리턴하고 연결 상태일 때 1을 리턴하므로 현재 상태를 출력하는 코드는 다음과 같이 구현하면 됩니다. 참고로 상태 표시줄에 메시지를 출력하는 코드도 함께 살펴보기 바랍니다.

```
if self.kiwoom.dynamicCall("GetConnectState()") == 0:
    self.statusBar().showMessage("Not connected")
else:
    self.statusBar().showMessage("Connected")
```

로그인 이벤트 처리하기

앞서 개발한 프로그램에서 Open API+의 CommConnect 메서드를 통해 키움증권 서버에 로그인을 시도하면 키움증권 서버는 그림 12.33과 같이 OnEventConnect라는 이벤트가 발생합니다. 따라서 프로

그램이 OnEventConnect 이벤트를 처리할 수 있다면 앞에서 사용한 GetConnectState 메서드를 사용하지 않고도 로그인 성공 여부도 확인할 수 있습니다.

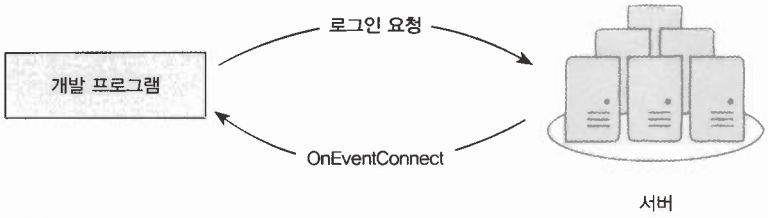


그림 12.33 OnEventConnect 이벤트 발생

그림 12.34는 OnEventConnect 이벤트를 처리해 로그인 성공 여부를 출력하는 프로그램입니다. 이 프로그램에서는 서버에서 발생하는 OnEventConnect 이벤트를 처리하는 메서드를 구현해 OnEventConnect 이벤트가 발생할 때 로그인 성공 여부를 판단합니다.

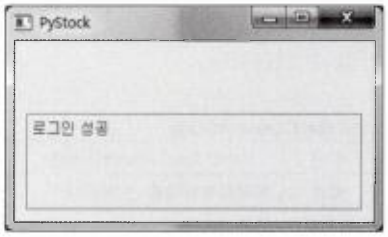


그림 12.34 로그인 여부 자동 확인

예제 12.5는 프로그램의 전체 코드입니다. 기본 구성은 지금까지 살펴본 프로그램과 유사합니다. 다만 PyQt의 QTextEdit 클래스의 인스턴스를 생성했고 이를 통해 메시지를 출력했습니다.

Open API+는 통신 연결 상태가 바뀔 때 OnEventConnect라는 이벤트가 발생합니다. 발생된 OnEventConnect 이벤트를 처리하기 위해 MyWindow 클래스에 event_connect라는 이름의 메서드를 구현하고 이벤트와 이벤트 처리 메서드만 연결하면 이벤트(OnEventConnect) 발생 시 자동으로 이벤트 처리 메서드(self.event_connect)가 호출됩니다.

예제 12.5 이벤트 처리를 통한 로그인 여부 확인(소스코드: book/ch12/08.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QAxContainer import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.setWindowTitle("PyStock")
        self.setGeometry(300, 300, 300, 150)

        self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")
        self.kiwoom.dynamicCall("CommConnect()")

        self.text_edit = QTextEdit(self)
        self.text_edit.setGeometry(10, 60, 280, 80)
        self.text_edit.setEnabled(False)

        self.kiwoom.OnEventConnect.connect(self.event_connect)

    def event_connect(self, err_code):
        if err_code == 0:
            self.text_edit.append("로그인 성공")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

예제 12.5에서 QTextEdit 객체를 생성하는 부분부터 자세히 살펴보겠습니다. QTextEdit 객체는 최상위 윈도우 안으로 생성돼야 하므로 QTextEdit 객체를 생성할 때 인자로 self 매개변수를 전달합니다.

```
self.text_edit = QTextEdit(self)
```

여기서 한 가지 기억해야 할 부분은 그림 12.35와 같이 생성된 객체를 self.text_edit라는 변수가 바인딩한다는 점입니다. PyQt로 GUI 프로그램을 개발하다 보면 위젯에 대한 객체를 많이 생성하는데 어떤 경우에는 self를 붙이지 않고 어떤 경우에는 self를 붙인 변수로 객체를 바인딩합니다.

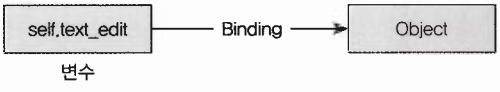


그림 12.35 변수와 객체의 바인딩

그림 12.35에서처럼 self를 사용하는 이유는 클래스의 다른 메서드에서도 해당 변수를 사용해 객체에 접근하기 위해서입니다. 만일 특정 객체가 다른 메서드에서는 사용될 필요가 없다면 self를 붙이지 않아도 됩니다. self.text_edit는 생성자에서 객체를 바인딩할 때 사용될뿐더러 event_connect 메서드에서도 사용되기 때문에 text_edit가 아니라 self.text_edit를 사용하는 것입니다.

QTextEdit 클래스의 크기 및 출력 위치를 조절하기 위해 setGeometry 메서드를 사용하고 읽기/쓰기 모드를 변경하기 위해 setEnabled 메서드를 사용합니다.

```
self.text_edit.setGeometry(10, 60, 280, 80)
self.text_edit.setEnabled(False)
```

Open API+ 개발 가이드를 참조하면 OnEventConnect 이벤트의 원형이 그림 12.36과 같이 정의돼 있습니다. OnEventConnect 함수의 인자로 nErrCode라는 정숫값이 입력되는데, 해당 값이 0이면 로그인 성공을 의미하고 해당 값이 음수이면 로그인 실패를 의미합니다.

5) OnEventConnect

원형	void OnEventConnect(LONG nErrCode);
설명	서버 접속 관련 이벤트
입력값	LONG nErrCode : 에러 코드
반환값	없음
비고	nErrCode가 0이면 로그인 성공, 음수면 실패 음수인 경우는 에러 코드 참조

그림 12.36 OnEventConnect 이벤트(개발 가이드)

PyQt에서 Open API+의 이벤트를 처리하려면 이벤트 함수의 원형을 참조해서 해당 이벤트를 처리할 메서드를 구현해야 합니다. 이벤트를 처리할 메서드의 이름은 자유롭게 사용해도 됩니다.

메서드의 첫 번째 인자는 인스턴스를 의미하는 self이므로 두 번째 인자로 nErrCode에 해당하는 값을 받으면 됩니다. 메서드의 이름과 마찬가지로 매개변수 이름도 꼭 nErrCode로 할 필요는 없습니다. 여기서는 err_code라고 했습니다.

```
def event_connect(self, err_code):
    if err_code == 0:
        self.text_edit.append("로그인 성공")
```

event_connect 메서드는 메서드의 인자로 전달되는 err_code 값을 확인합니다. 해당 값이 0이면 앞서 생성자에서 생성한 self.text_edit 객체에 '로그인 성공'이라는 문자열을 출력합니다. QTextEdit 객체에 문자열을 추가하기 위해서는 append라는 메서드를 사용합니다. 참고로 이 append는 파이썬 리스트의 append 메서드와 이름만 같을 뿐 전혀 관련이 없습니다.

이벤트를 처리할 메서드를 구현했다면 이벤트와 이벤트 처리 메서드를 연결하면 됩니다. 이벤트와 이벤트 처리 메서드를 연결할 때는 connect 메서드를 사용하면 됩니다.

```
def __init__(self):
    super().__init__()
```

```
    # 생략
    self.kiwoom.OnEventConnect.connect(self.event_connect)
```

예제 12.5에서는 CommConnect 메서드를 MyWindow 클래스의 생성자에서 호출했습니다. 따라서 윈도우가 생성됨과 동시에 로그인 창도 함께 화면에 표시됩니다.

```
self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")
self.kiwoom.dynamicCall("CommConnect()")
```

기본 정보 요청하기

키움증권도 이베스트투자증권과 마찬가지로 TR(Transaction)이라는 용어를 사용합니다. 키움증권 개발 가이드에는 TR에 대한 정의가 따로 돼 있지는 않지만 '서버로부터 데이터를 주고받는 행위'를 TR이라고 생각해도 큰 무리가 없습니다.

중요한 점은 키움증권의 Open API+도 이베스트투자증권과 비슷하게 다양한 TR을 제공한다는 점입니다. 그림 12.37은 KOA Studio를 통해 확인한 Open API+의 TR 목록입니다. 각 TR의 기능을 살펴보고 원하는 기능을 제공하는 TR을 사용해 데이터를 서버에 요청한 후 전송된 값을 받으면 됩니다.

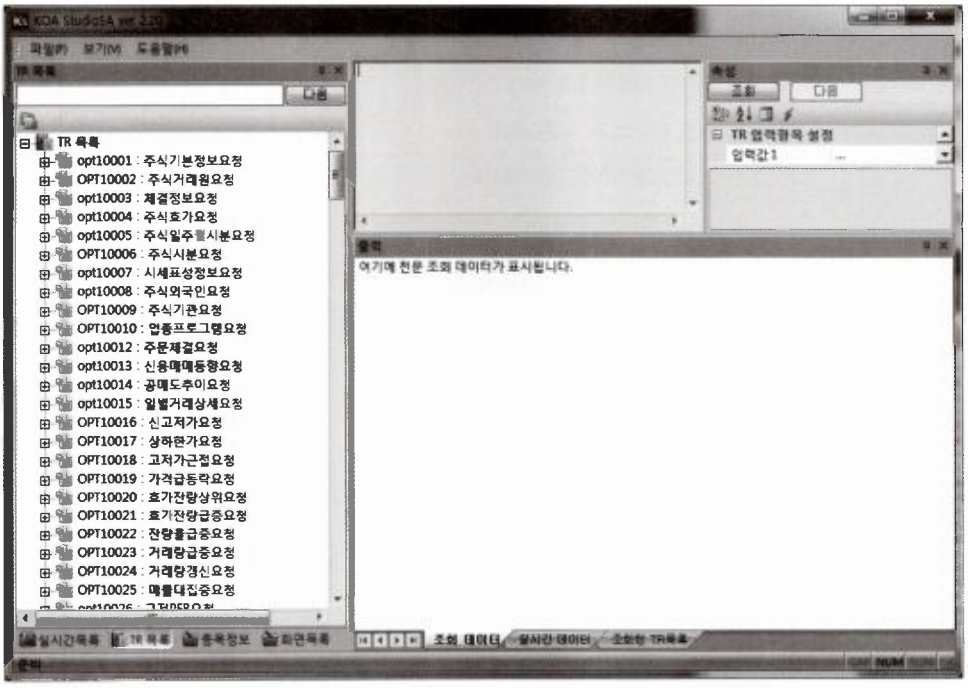


그림 12.37 Open API+ TR 목록

이번 절에서는 'opt10001' TR을 사용해 주식 종목의 기본적인 정보를 가져오는 프로그램을 만들어보겠습니다. 이번 절에서 만들 프로그램을 실행한 모습은 그림 12.38과 같습니다. 사용자로부터 종목 코드를 입력받은 후 사용자가 [조회] 버튼을 클릭하면 해당 종목에 대한 '종목명'과 '거래량'을 화면에 출력하는 프로그램입니다.

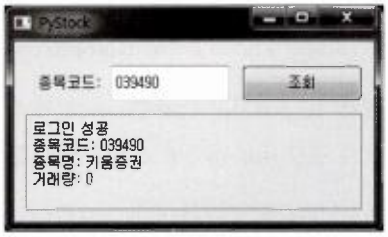


그림 12.38 기본 정보를 요청하는 프로그램

먼저 PyQt로 UI를 구성하는 코드부터 구현해 보겠습니다. 그다음 Open API+의 메서드를 호출해 키움증권 서버로부터 데이터를 받고 해당 데이터를 UI에 출력하는 코드를 구현하겠습니다.

그림 12.38의 프로그램에서 사용된 PyQt 위젯을 정리해보면 표 12.1과 같습니다. QLabel은 그림 '종목코드:'라는 문자열을 윈도우 창에 출력하는 데 사용됩니다. QLineEdit는 사용자로부터 종목 코드(039490)를 입력받을 때 사용됩니다. QPushButton은 '조회' 버튼에 사용되며, QTextEdit는 프로그램의 실행 결과를 출력하는 용도로 사용됩니다.

표 12.1 예제에 사용된 PyQt 위젯

클래스	용도
QLabel	간단한 텍스트 출력('종목코드:' 출력)
QLineEdit	간단한 사용자 입력 처리
QPushButton	버튼 생성('조회' 버튼 생성)
QTextEdit	메시지 출력(실행 결과 메시지)

예제 12.6 그림 12.38 프로그램의 UI 구성 코드(소스코드: book/ch12/09.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import  *
from PyQt5.QAxContainer import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("PyStock")
        self.setGeometry(300, 300, 300, 150)

        label = QLabel('종목코드: ', self)
        label.move(20, 20)
```

```
        self.code_edit = QLineEdit(self)
        self.code_edit.move(80, 20)
        self.code_edit.setText("039490")

        btn1 = QPushButton("조회", self)
        btn1.move(190, 20)

        self.text_edit = QTextEdit(self)
        self.text_edit.setGeometry(10, 60, 280, 80)
        self.text_edit.setEnabled(False)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

예제 12.6은 그림 12.38에 나온 프로그램의 UI를 구성하는 코드입니다. 아직은 각 위젯에 대한 이벤트 처리 함수를 구현하지 않았기 때문에 예제 12.6을 실행해도 버튼과 같은 위젯은 제대로 동작하지 않습니다.

QLabel은 텍스트나 이미지를 출력하는 데 사용합니다. 만약 텍스트를 출력한다면 생성자의 첫 번째 인자로 출력될 문자열을 전달하면 됩니다. 두 번째 인자에는 부모 위젯을 지정합니다. 그림 12.38에서 QLabel은 QMainWindow 안에 위치하므로 부모 위젯으로 self를 지정합니다.

```
label = QLabel('종목코드: ', self)
```

위젯의 출력 위치를 조절하기 위해 move 메서드를 사용했는데, 크기와 위치를 동시에 조절하려면 setGeometry 메서드를 사용하면 됩니다.

```
label.move(20, 20)
```

앞에서 설명한 것처럼 QLabel은 생성자에서 텍스트를 출력하는 용도로만 사용되며 다른 메서드에서는 사용되지 않습니다. 따라서 self.label이라는 변수로 바인딩하지 않고 label이라는 이름으로 바인딩했습니다.

사용자로부터 종목 코드를 입력받기 위해 QLineEdit 위젯을 사용했습니다. 역시 위젯의 출력 위치를 조절하기 위해 move 메서드를 사용합니다. 사용자로부터 종목 코드를 입력받기 전에 기본값으로 키움증권의 종목 코드를 QLineEdit에 출력하기 위해 setText 메서드를 사용했습니다.

```
self.code_edit = QLineEdit(self)
self.code_edit.move(80, 20)
self.code_edit.setText("039490")
```

QLineEdit 위젯은 생성자뿐 아니라 다른 메서드에서도 사용될 예정이므로 self.code_edit와 같이 self라는 키워드를 붙여서 객체를 바인딩했습니다.

그림 12.28에서 버튼은 QPushButton 클래스를 사용합니다. QPushButton의 생성자를 호출할 때 버튼에 출력될 텍스트를 첫 번째 인자로 전달하고, 버튼의 부모 위젯을 두 번째 인자로 전달합니다. 버튼 위젯이 표시될 위치는 move 메서드로 조절할 수 있습니다.

```
btn1 = QPushButton("조회", self)
btn1.move(190, 20)
```

로그인 처리 결과나 Open API+ 결과를 출력하기 위해 QTextEdit 위젯을 사용했습니다. 위치와 크기를 한 번에 조절하기 위해 setGeometry 메서드를 사용했고, 사용자가 QTextEdit 위젯을 통해 입력할 수 없고 오직 읽기 모드로만 사용하도록 setEnabled 메서드를 사용했습니다.

```
self.text_edit = QTextEdit(self)
self.text_edit.setGeometry(10, 60, 280, 80)
self.text_edit.setEnabled(False)
```

이번에는 사용자가 QLineEdit 위젯에 종목 코드를 입력한 후 [조회] 버튼을 클릭하면 입력된 종목 코드가 QTextEdit 위젯에 출력되도록 코드를 수정해 봅시다. 또한 12.3.2절에서 배운 로그인 이벤트 처리도 추가해 봅시다. 다음은 예제 12.6에 '조회' 버튼에 대한 이벤트 처리 및 로그인 이벤트 처리를 위한 코드를 추가한 코드입니다.

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import  *
from PyQt5.QAxContainer import *

class MyWindow(QMainWindow):
    def _init_(self):
        super().__init__()

        # Kiwoom Login
        self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")
        self.kiwoom.dynamicCall("CommConnect()")
        self.kiwoom.OnEventConnect.connect(self.event_connect)
```

```
        self.setWindowTitle("PyStock")
        self.setGeometry(300, 300, 300, 150)

        label = QLabel('종목코드: ', self)
        label.move(20, 20)

        self.code_edit = QLineEdit(self)
        self.code_edit.move(80, 20)
        self.code_edit.setText("039490")

        btn1 = QPushButton("조회", self)
        btn1.move(190, 20)
        btn1.clicked.connect(self.btn1_clicked)

        self.text_edit = QTextEdit(self)
        self.text_edit.setGeometry(10, 60, 280, 80)
        self.text_edit.setEnabled(False)

    def event_connect(self, err_code):
        if err_code == 0:
            self.text_edit.append("로그인 성공")

    def btn1_clicked(self):
        code = self.code_edit.text()
        self.text_edit.append("종목코드: " + code)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

버튼의 이벤트를 처리하기 위해 btn1 객체에서 'clicked' 이벤트가 발생하면 btn1_clicked 메서드가 호출되도록 설정합니다.

```
btn1.clicked.connect(self.btn1_clicked)
```

btn1_clicked 메서드에서는 먼저 self.code_edit를 통해 QLineEdit 객체에 접근한 후 text 메서드를 통해 QLineEdit에 사용자가 입력한 값을 가져옵니다. 사용자로부터 입력받은 값을 QTextEdit 위젯에 출력하기 위해 self.text_edit라는 변수를 통해 append 메서드를 호출합니다. 이처럼 한 위젯에서 이벤트가 발생했을 때 다른 위젯으로부터 값을 읽고 이를 처리한 후 또 다른 위젯에 값을 출력하는 것은 GUI 프로그래밍에서 매우 일반적입니다.

```
def btn1_clicked(self):
    code = self.code_edit.text()
    self.text_edit.append("종목코드: " + code)
```

이번에는 사용자에게서 입력받은 종목 코드와 Open API+가 제공하는 TR 메서드를 통해 기본 정보를 가져오는 코드를 작성해 보겠습니다. 해당 기능이 추가된 전체 코드는 예제 12.7과 같습니다

예제 12.7 opt10001 TR을 통해 기본 정보를 요청하는 코드(소스코드: book/ch12/11.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import  *
from PyQt5.QAxContainer import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # Kiwoom Login
        self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")
        self.kiwoom.dynamicCall("CommConnect()")

        # OpenAPI+ Event
        self.kiwoom.OnEventConnect.connect(self.event_connect)
        self.kiwoom.OnReceiveTrData.connect(self.receive_trdata)

        self.setWindowTitle("PyStock")
        self.setGeometry(300, 300, 300, 150)

        label = QLabel('종목코드: ', self)
        label.move(20, 20)

        self.code_edit = QLineEdit(self)
        self.code_edit.move(80, 20)
        self.code_edit.setText("039490")

        btn1 = QPushButton("조회", self)
        btn1.move(190, 20)
        btn1.clicked.connect(self.btn1_clicked)

        self.text_edit = QTextEdit(self)
        self.text_edit.setGeometry(10, 60, 280, 80)
```

```
        self.text_edit.setEnabled(False)

    def event_connect(self, err_code):
        if err_code == 0:
            self.text_edit.append("로그인 성공")

    def btn1_clicked(self):
        code = self.code_edit.text()
        self.text_edit.append("종목코드: " + code)

        # SetInputValue
        self.kiwoom.dynamicCall("SetInputValue(QString, QString)", "종목코드", code)

        # CommRqData
        self.kiwoom.dynamicCall("CommRqData(QString, QString, int, QString)", "opt10001_req",
"opt10001", 0, "0101")

    def receive_trdata(self, screen_no, rqname, trcode, recordname, prev_next, data_len, err_
code, msg1, msg2):
        if rqname == "opt10001_req":
            name = self.kiwoom.dynamicCall("CommGetData(QString, QString, QString, int,
QString)", trcode, "", rqname, 0, "종목명")
            volume = self.kiwoom.dynamicCall("CommGetData(QString, QString, QString, int,
QString)", trcode, "", rqname, 0, "거래량")

            self.text_edit.append("종목명: " + name.strip())
            self.text_edit.append("거래량: " + volume.strip())

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

Open API+의 TR을 사용하는 코드를 작성하기 전에 먼저 TR 처리 순서를 간략히 살펴보겠습니다. Open API+의 TR 처리 순서는 그림 12.39와 같습니다.

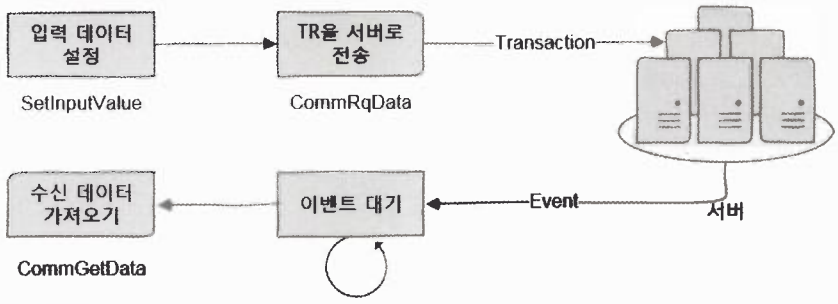


그림 12.39 Open API+의 TR 처리 순서

1. SetInputValue 메서드를 사용해 TR 입력 값을 설정합니다.
2. CommRqData 메서드를 사용해 TR을 서버로 송신합니다.
3. 서버로부터 이벤트가 발생할 때까지 이벤트 루프를 사용해 대기합니다.
4. CommGetData 메서드를 사용해 수신 데이터를 가져옵니다.

KOA Studio를 참조하면 opt10001 TR에 대한 C++ 예제 코드는 그림 12.40과 같습니다. 먼저 SetInputValue 메서드를 호출해 '종목코드'라는 문자열과 실제 종목 코드 값(예: 039490)을 입력합니다. 그다음 CommRqData 메서드를 호출해 TR을 서버에 전송하면 됩니다.

```
/*****************************************************************************/
/// ########## Open API 함수를 이용한 전문처리 C++용 샘플코드 예제입니다.

[ opt10001 : 주식기본정보요청 ]

1. Open API 조회 함수 입력값을 설정합니다.
        종목코드 = 전문 조회할 종목코드
        SetInputValue("종목코드"      , "입력값 1");


2. Open API 조회 함수를 호출해서 전문을 서버로 전송합니다.
        CommRqData( "RQName"      , "opt10001"      , "0"     , "화면번호");
/*****************************************************************************/
```

그림 12.40 opt10001 TR에 대한 C++ 예제 코드

파이썬 코드에서는 사용자가 [조회] 버튼을 클릭했을 때 사용자가 입력한 종목 코드값으로 TR 데이터를 구성한 후 해당 TR을 서버로 송신하도록 코드를 수정하면 됩니다. 먼저 btn1_clicked 메서드를 다음과 같이 수정합니다.

```
def btn1_clicked(self):
    code = self.code_edit.text()
    self.text_edit.append("종목코드: " + code)
```

```
        # SetInputValue
        self.kiwoom.dynamicCall("SetInputValue(QString, QString)", "종목코드", code)

        # CommRqData
        self.kiwoom.dynamicCall("CommRqData(QString, QString, int, QString)", "opt10001_req",
"opt10001", 0, "0101")
```

위 코드를 참조하면 QLineEdit 위젯을 통해 입력받은 종목 코드를 SetInputValue 메서드의 두 번째 인자로 전달하는 것을 확인할 수 있습니다.

TR 구성이 완료되면 CommRqData 메서드를 사용해 TR을 서버로 송신하면 됩니다. CommRqData의 첫 번째 인자는 사용자가 TR을 구분하기 위한 용도로 사용합니다. 여기서는 "opt10001_req"라는 문자열을 첫 번째 인자로 사용했습니다. 두 번째 인자는 요청하는 TR 이름으로서 "opt10001"을 입력하면 됩니다. 세 번째 인자로는 단순 조회 TR일 경우 0을 입력합니다. 네 번째 인자는 4자리의 화면 번호인데 여기서는 개발 가이드를 참조해서 기본값인 "0101"을 사용했습니다.

예제 12.7의 btn1_clicked 메서드에서 SetInputValue와 CommRqData 호출까지 진행했습니다. TR을 서버로 송신했으므로 OnReceiveTrData라는 이벤트를 처리할 메서드를 구현하고 이를 이벤트와 연결하면 됩니다.

이를 위해 먼저 키움증권 개발 가이드에서 OnReceiveTrData 이벤트의 원형을 살펴봅시다. 그림 12.41을 참조하면 OnReceiveTrData 이벤트는 서버와 통신한 후 서버로부터 데이터를 전달받은 시점에 발생합니다. 또한 OnReceiveTrData 이벤트에는 총 9개의 인자가 전달되는 것을 확인할 수 있습니다.

1) OnReceiveTrData

원형	void OnReceiveTrData(LPCTSTR sScrNo, LPCTSTR sRQName, LPCTSTR sTrCode, LPCTSTR sRecordName, LPCTSTR sPreNext, LONG nDataLength, LPCTSTR sErrorCode, LPCTSTR sMessage, LPCTSTR sSplmMsg)
설명	서버통신 후 데이터를 받은 시점을 알려준다.
입력값	sScrNo – 화면번호 sRQName – 사용자구분 명 sTrCode – Tran 명 sRecordName – Record 명 sPreNext – 연속조회 유무 nDataLength – 1.0.0.1 버전 이후 사용하지 않음. sErrorCode – 1.0.0.1 버전 이후 사용하지 않음. sMessage – 1.0.0.1 버전 이후 사용하지 않음. sSplmMsg - 1.0.0.1 버전 이후 사용하지 않음.
반환값	없음
비고	sRQName – CommRqData의 sRQName과 매핑되는 이름이다. sTrCode – CommRqData의 sTrCode과 매핑되는 이름이다.

그림 12.41 OnReceiveTrData 이벤트

OnReceiveTrData 이벤트의 인자가 9개이므로 해당 이벤트를 처리할 메서드를 다음과 같이 구현해 봅시다. OnReceiveTrData 메서드의 첫 번째 인자는 당연히 self이고 나머지 인자는 OnReceiveTrData 이벤트의 원형을 참조해서 인자를 순서대로 받을 수 있도록 구현하면 됩니다.

```
def receive_trdata(self, screen_no, rqname, trcode, recordname, prev_next, data_len, err_code,
msg1, msg2):
    pass
```

OnReceiveTrData 이벤트가 발생했다는 것은 서버로부터 데이터를 전달받았음을 의미하므로 OnReceiveTrData 메서드에서 CommGetData 메서드를 호출해서 데이터를 가져오면 됩니다.

먼저 개발 가이드에서 CommGetData 메서드를 살펴봅시다. 그림 12.42를 참조하면 CommGetData 메서드를 사용해 TR 데이터, 실시간 데이터, 체결잔고 데이터 등을 구해올 수 있습니다.

9) CommGetData

원형	BSTR CommGetData(LPCTSTR sJongmokCode, LPCTSTR sRealType, LPCTSTR sFieldName, long nIndex, LPCTSTR sInnerFieldName)		
설명	Tran 데이터, 실시간 데이터, 체결잔고 데이터를 반환한다.		
입력값	<비고>		
반환값	요청 데이터		
비고	① Tran 데이터 sJongmokCode : Tran명 sRealType : 사용안함 sFieldName : 레코드명 nIndex : 반복인덱스 sInnerFieldName: 아이템명	① 실시간 데이터 sJongmokCode : Key Code sRealType : Real Type sFieldName : Item Index nIndex : 사용안함 sInnerFieldName:사용안함	① 체결 데이터 sJongmokCode : 체결구분 sRealType : "-1" sFieldName : 사용안함 nIndex : ItemIndex sInnerFieldName:사용안함
	Ex)		

그림 12.42 CommGetData 메서드(개발 가이드)

CommGetData 메서드는 Open API+에서 제공하는 메서드이므로 이를 파이썬 코드에서 사용하려면 dynamicCall 메서드를 사용해야 합니다. 그리고 Open API+에는 수많은 TR이 있으므로 CommGetData의 첫 번째 인자와 세 번째 인자에 TR 명과 Request 명을 입력해 어떤 TR에 대한 데이터를 얻고자 하는지 알려줘야 합니다.

다음 코드에서 receive_trdata 메서드는 OnReceiveTrData 이벤트가 발생할 때마다 자동으로 호출됩니다. 어떤 TR 요청에 의해 OnReceiveTrData 이벤트가 발생했는지 확인하기 위해 먼저 사용자 Request 명(rqname)을 확인합니다. 앞서 CommRqData를 통해 서버에 TR을 송신할 때 Request 명으로 'opt10001_req'를 설정했으므로 receive_trdata 메서드에서는 rqname 값이 'opt10001_req'인지 확인합니다.

```
def receive_trdata(self, screen_no, rqname, trcode, recordname, prev_next, data_len, err_code,
msg1, msg2):
    if rqname == "opt10001_req":
        name = self.kiwoom.dynamicCall("CommGetData(QString, QString, QString, int, QString)",
trcode, "", rqname, 0, "종목명")
        volume = self.kiwoom.dynamicCall("CommGetData(QString, QString, QString, int, QString)",
trcode, "", rqname, 0, "거래량")

        self.text_edit.append("종목명: " + name.strip())
        self.text_edit.append("거래량: " + volume.strip())
```

rqname이 'opt10001_req'이면 CommGetData 메서드를 호출해 '종목명'과 '거래량'에 해당하는 값을 가져옵니다. 이때 앞서 설명한 것처럼 CommGetData의 첫 번째 인자와 세 번째 인자에 trcode와 rqname을 전달하는 것을 확인할 수 있습니다.

CommGetData 메서드를 호출해 '종목명'과 '거래량' 값을 가져왔다면 strip 메서드를 호출해 문자열의 공백을 제거합니다. 그런 다음 QTextEdit 객체에 해당 문자열을 추가합니다.

OnReceiveTrData 이벤트를 처리할 receive_trdata 메서드를 구현했으므로 이제 이벤트와 이벤트 처리용 메서드를 연결하면 됩니다. 다음과 같이 MyWindow 클래스의 생성자에서 이벤트와 이벤트를 처리할 메서드를 연결합니다.

```
class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        # 생략

        # OpenAPI+ Event
        self.kiwoom.OnReceiveTrData.connect(self.receive_trdata)
```

계좌 정보 가져오기

키움증권 Open API+는 다수의 메서드를 제공합니다. 이 가운데 GetLoginInfo 메서드를 이용하면 계좌 개수, 계좌 번호, 사용자 ID, 사용자명 등의 정보를 가져올 수 있습니다. 이번 절에서는 GetLoginInfo 메서드를 사용해 그림 12.43과 같이 로그인을 수행한 후 해당 계정의 계좌 번호를 가져오는 프로그램을 구현해 보겠습니다.

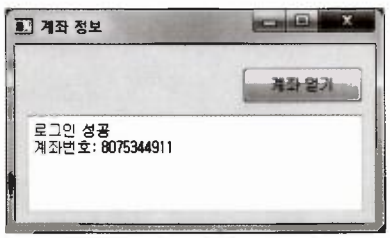


그림 12.43 계좌 정보 가져오기

그림 12.43의 프로그램을 구현하기에 앞서 먼저 키움증권의 개발 가이드에서 GetLoginInfo 메서드에 대해 살펴봅시다. 그림 12.44를 참조하면 GetLoginInfo 메서드의 인자는 한 개인데 해당 위치에 정해진 문자열을 입력함으로써 계좌 개수, 계좌 번호, 사용자 ID 등을 구할 수 있음을 확인할 수 있습니다.

4) GetLoginInfo

원형	BSTR GetLoginInfo(BSTR sTag)
설명	로그인한 사용자 정보를 반환한다.
입력값	BSTR sTag : 사용자 정보 구분 TAG값 (비고)
반환값	TAG값에 따른 데이터 반환
비고	BSTR sTag에 들어 갈 수 있는 값은 아래와 같음 "ACCOUNT_CNT" – 전체 계좌 개수를 반환한다. "ACCNO" – 전체 계좌를 반환한다. 계좌별 구분은 ';'이다. "USER_ID" - 사용자 ID를 반환한다. "USER_NAME" – 사용자명을 반환한다. "KEY_BSECGB" – 키보드보안 해지여부. 0:정상, 1:해지 "FIREW_SECGB" – 방화벽 설정 여부. 0:미설정, 1:설정, 2:해지 Ex) openApi.GetLoginInfo("ACCOUNT_CNT");

그림 12.44 GetLoginInfo 메서드(개발 가이드)

예제 12.8은 그림 12.43에 나온 프로그램의 전체 코드인데, 기본 UI를 구성하는 코드는 예제 12.5와 유사합니다. 다만 버튼이 클릭됐을 때 GetLoginInfo 메서드를 호출하는 부분이 추가됐습니다.

예제 12.8 계좌 정보를 가져오는 프로그램(소스코드: book/ch12/12.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import  *
from PyQt5.QAxContainer import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # Kiwoom Login
        self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")
        self.kiwoom.dynamicCall("CommConnect()")

        # OpenAPI+ Event
        self.kiwoom.OnEventConnect.connect(self.event_connect)

        self.setWindowTitle("계좌 정보")
        self.setGeometry(300, 300, 300, 150)
```

```
        btn1 = QPushButton("계좌 얻기", self)
        btn1.move(190, 20)
        btn1.clicked.connect(self.btn1_clicked)

        self.text_edit = QTextEdit(self)
        self.text_edit.setGeometry(10, 60, 280, 80)

    def btn1_clicked(self):
        account_num = self.kiwoom.dynamicCall("GetLoginInfo(QString)", ["ACCNO"])
        self.text_edit.append("계좌번호: " + account_num.rstrip(';'))

    def event_connect(self, err_code):
        if err_code == 0:
            self.text_edit.append("로그인 성공")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

GetLoginInfo 메서드는 이벤트 방식으로 동작하는 것도 아니므로 단순히 dynamicCall 메서드를 통해 호출하면 값을 바로 얻을 수 있습니다. 다만 메서드의 인자가 한 개인데 실제 인자를 전달하는 부분에서 다음 코드와 같이 반드시 리스트 형태로 값을 전달해야 합니다.

```
account_num = self.kiwoom.dynamicCall("GetLoginInfo(QString)", ["ACCNO"])
```

만약 위와 같이 리스트를 사용하지 않고 다음과 같이 문자열을 직접 전달하려고 하면 에러가 발생합니다. 이 부분은 중요하니 반드시 기억해두기 바랍니다.

```
account_num = self.kiwoom.dynamicCall("GetLoginInfo(QString)", "ACCNO")
```

함수의 인자로 'ACCNO'를 전달하면 서버는 전체 계좌를 반환하는데, 각 계좌 번호에는 세미콜론(;)이 붙어 있습니다. 따라서 QTextEdit에 계좌 번호를 출력할 때 문자열의 rstrip 메서드를 사용해 문자열 끝의 세미콜론을 제거했습니다.

```
self.text_edit.append("계좌번호: " + account_num.rstrip(';'))
```

종목 코드 및 한글 종목명 가져오기

증권사 API를 사용할 때 종목 코드는 가장 기본이 되는 데이터입니다. 키움증권의 Open API+도 종목 코드를 사용해 TR을 요청하는 경우가 많습니다. 따라서 종목 코드를 API를 통해 먼저 가져와야 각 종목에 대한 반복적인 작업 등을 자동화할 수 있습니다.

이번 절에서는 그림 12.45와 같이 유가증권시장의 종목 코드 및 각 종목에 해당하는 한글 종목명을 출력하는 프로그램을 구현해보겠습니다.

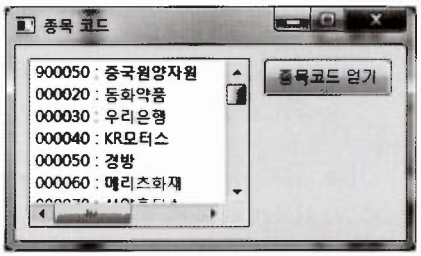


그림 12.45 종목 코드 및 한글 종목명을 출력하는 프로그램

키움증권의 개발 가이드를 참조하면 종목 코드 목록을 가져오는 메서드는 GetCodeListByMarket입니다. 그림 12.46을 참조하면 GetCodeListByMarket 메서드에 인자로 시장 구분 코드를 문자열로 전달하면 메서드의 리턴 값으로 해당 시장에 속한 종목의 종목 코드 목록이 반환됩니다.

원형	BSTR GetCodeListByMarket(LPCTSTR sMarket)
설명	시장구분에 따른 종목코드를 반환한다.
입력값	sMarket – 시장구분
반환값	종목코드 리스트, 종목간 구분은 ';'이다.
비고	sMarket – 0:장내, 3:ELW, 4:뮤추얼펀드, 5:신주인수권, 6:리츠, 8:ETF, 9:하이일드펀드, 10:코스닥, 30:제3시장

그림 12.46 GetCodeListByMarket 메서드

종목 코드로부터 한글 종목명을 구하려면 그림 12.47과 같이 GetMasterCodeName을 사용하면 됩니다.

16) GetMasterCodeName

원형	BSTR GetMarsterCodeName(LPCTSTR strCode)
설명	종목코드의 한글명을 반환한다.
입력값	strCode – 종목코드
반환값	종목한글명
비고	장내외, 지수선옵, 주식선옵 검색 가능.

그림 12.47 GetMarsterCodeName 메서드

예제 12.9는 그림 12.45에 나온 프로그램의 전체 코드입니다. 기본 코드 구성은 앞의 예제와 유사하며, 종목 코드 및 종목 한글명을 출력하기 위해 QListWidget이라는 위젯을 사용했습니다.

예제 12.9 종목 코드 및 종목 한글명 가져오기(소스코드: book/ch12/13.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import  *
from PyQt5.QAxContainer import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.kiwoom = QAxWidget("KHOPENAPI.KHOpenAPICtrl.1")
        self.kiwoom.dynamicCall("CommConnect()")

        self.setWindowTitle("종목 코드")
        self.setGeometry(300, 300, 300, 150)

        btn1 = QPushButton("종목코드 얻기", self)
        btn1.move(190, 10)
        btn1.clicked.connect(self.btn1_clicked)

        self.listWidget = QListWidget(self)
        self.listWidget.setGeometry(10, 10, 170, 130)

    def btn1_clicked(self):
        ret = self.kiwoom.dynamicCall("GetCodeListByMarket(QString)", ["0"])
        kospi_code_list = ret.split(';')
        kospi_code_name_list = []

        for x in kospi_code_list:
            name = self.kiwoom.dynamicCall("GetMasterCodeName(QString)", [x])
            kospi_code_name_list.append(x + " : " + name)

        self.listWidget.addItems(kospi_code_name_list)

if __name__ == "__main__":
    app = QApplication(sys.argv)
```

```
myWindow = MyWindow()
myWindow.show()
sys.exit(app.exec_())
```

생성자에서는 QListWidget 객체를 생성한 후 setGeometry 메서드를 통해 위젯의 크기 및 출력 위치를 조절합니다.

```
class MyWindow(QMainWindow):
    def __init__(self):
        # 코드 생략
        self.listWidget = QListWidget(self)
        self.listWidget.setGeometry(10, 10, 170, 130)
```

이번에는 [종목코드 얻기] 버튼을 클릭할 때 호출되는 메서드인 btn1_clicked를 구현해 보겠습니다. 먼저 GetCodeListByMarket 메서드를 통해 유가증권시장의 종목 코드 목록을 가져옵니다.

```
ret = self.kiwoom.dynamicCall("GetCodeListByMarket(QString)", ["0"])
```

GetCodeListByMarket 메서드의 리턴 값은 문자열이고 해당 문자열 내에서 각 종목은 세미콜론(;)으로 구분됩니다. 종목 코드를 더 효과적으로 관리하기 위해 세미콜론을 기준으로 문자열을 분리해서 파이썬 리스트를 생성합니다.

```
kospi_code_list = ret.split(';')
```

kospi_code_list에는 각 종목의 종목 코드가 저장돼 있습니다. 파이썬의 반복문을 사용해 각 종목 코드에 대한 한글 종목명을 가져온 후 종목 코드와 한글 종목명을 하나의 문자열로 구성해서 새로운 리스트(kospi_code_name_list)에 추가합니다.

```
kospi_code_name_list = []
for x in kospi_code_list:
    name = self.kiwoom.dynamicCall("GetMasterCodeName(QString)", [x])
    kospi_code_name_list.append(x + " : " + name)
```

마지막으로 종목 코드와 한글 종목명이 저장된 파이썬 리스트를 addItems 메서드의 인자로 전달해서 파이썬 리스트에 있는 항목을 QListWidget에 추가합니다.

```
self.listWidget.addItems(kospi_code_name_list)
```

3부

데이터 분석 기초

3부에서는 pandas라는 라이브러리를 이용한 데이터 분석 기초를 다루고 Zipline을 이용해 백테스팅하는 방법을 소개합니다. 그리고 matplotlib를 통해 데이터를 시각화하는 방법과 PyQt를 이용한 GUI(Graphical User Interface) 개발을 배웁니다.

13

pandas를 이용한 데이터 분석 기초

10~12장에서 국내 증권사의 API를 사용하는 방법을 배웠습니다. 국내 증권사의 API를 이용하면 과거 시점의 주식 시세를 얻어오는 것은 물론 프로그래밍을 통해 매수/매도 주문도 넣을 수 있었습니다. 하지만 증권사의 API가 자동으로 과거 데이터를 분석해주고 유용한 정보를 제공해주지는 않습니다. API를 이용해 힘들게 얻어온 주식 데이터로부터 투자에 유용한 정보를 추출하려면 어떻게 해야 할까요?

데이터로부터 의미 있는 정보를 추출하는 기술은 최근 들어 매우 주목받고 있습니다. 사실 데이터 분석 분야는 예전부터 의미가 있었고 중요했습니다. 하지만 최근 들어 분석할 데이터의 양(volume)이 커지고, 데이터의 입출력 속도(velocity)가 빨라지고, 데이터의 종류가 다양해(variety)지면서 기존보다 데이터를 분석하기 어려워졌고 이로 인해 데이터 분석 분야가 업계의 주목을 받는 것입니다.

Volume, Velocity, Variety의 세 가지 'V'를 가진 데이터를 흔히 '빅데이터'라고 부릅니다. 여러분이 관심 있어 하는 주식 데이터도 대표적인 빅데이터라고 말할 수 있습니다. 빅데이터 분석에는 다양한 프로그래밍 언어와 기술이 사용되고 있는데 파이썬도 그중 하나입니다.

최근 들어 파이썬이 오픈소스 기반의 통계언어인 R과 더불어 빅데이터 분석 분야에서 인기가 높아진 데는 여러 가지 이유가 있지만, 이번 장에서 배우게 될 pandas라는 라이브러리 덕이 큽니다. 여러분도 이번 장을 통해 pandas의 기초를 배워보기 바랍니다.

01 pandas Series

파이썬이 인기 있는 이유 중 하나는 파이썬의 기본 자료구조인 리스트, 튜플, 딕셔너리가 사용하기 편리하고 데이터를 다루는 데 효과적이기 때문입니다. pandas 역시 효과적인 데이터 분석을 위한 고수준의 자료구조와 데이터 분석 도구를 제공합니다. pandas의 Series는 1차원 데이터를 다루는 데 효과적인 자료구조이며, DataFrame은 행과 열로 구성된 2차원 데이터를 다루는 데 효과적인 자료구조입니다.

pandas를 이해하려면 가장 먼저 pandas의 핵심 자료구조인 Series와 DataFrame을 알아야 합니다. 이번 절에서는 먼저 Series에 대해 간단히 살펴봅니다.

파이썬 리스트, 튜플, 딕셔너리

이미 1부 파이썬 문법편을 통해 파이썬의 기본 자료구조인 리스트, 튜플, 딕셔너리에 대해 공부했습니다. 이번 절에서는 pandas의 자료구조인 Series를 살펴보기 전에 간단히 파이썬의 자료구조를 복습해 보겠습니다.

여러분이 카카오와 네이버 주식을 보유했다면 이를 리스트라는 자료구조로 저장할 수 있으며, 다음 예제와 같이 0번과 1번 인덱스를 통해 리스트에 저장된 원소에 접근할 수 있습니다.

```
mystock = ['kakao', 'naver']
print(mystock[0])
print(mystock[1])
```

파이썬의 리스트를 사용하는 것이 기존에 변수를 사용했던 것에 비해 좋은 점은 리스트를 이용하면 반복문을 통해 쉽게 데이터를 관리할 수 있다는 것입니다. 변수는 규칙성이 없으므로 반복문을 사용하기 어렵습니다. 그렇지만 리스트를 이용하면 다음과 같이 반복문을 통해 각 데이터에 접근할 수 있습니다.

```
for stock in mystock:
    print(stock)
```

파이썬 리스트는 반복문을 사용할 수 있다는 장점 외에도 다양한 메서드를 제공합니다. 대표적으로 insert, append, index 등이 있었지요? 기억이 잘 안 난다면 잠깐 3장을 복습한 후 계속해서 다음 부분을 읽기 바랍니다.

파이썬의 튜플은 어땠습니까? 튜플은 리스트의 []와 달리 ()를 사용했으며 수정이 가능한 리스트와 달리 수정할 수 없었습니다. 그 대신 리스트에 비해 속도가 빠르다는 장점이 있었습니다. 그래서 원소를 한 번 넣은 후에 수정할 필요가 없으며 속도가 중요한 상황에 리스트 대신 튜플을 사용했습니다.

마지막으로 파이썬의 딕셔너리를 생각해봅시다. 딕셔너리는 리스트와 튜플과는 조금 달리 키-값(key-value)이라는 쌍으로 데이터를 저장했습니다. 따라서 키를 가지고 아주 빠르게 값을 찾을 수 있는 자료구조였습니다.

다음과 같이 딕셔너리를 사용하는 간단한 코드를 작성해봅시다.

```
exam_dic = {'key1': 'room1', 'key2':'room2'}
print(exam_dic['key1'])
print(exam_dic['key2'])
```

예제 코드에서 exam_dic은 그림 13.1과 같은 구조를 띱니다. 'key1'이라는 문자열은 자신에게 연결된 'room1'이라는 값을 바인딩하고 있으며, 'key2'는 'room2'를 바인딩합니다. 리스트나 튜플이 인덱스를 통해 값에 접근하는 것과 달리 딕셔너리는 키를 사용해 빠르게 값에 접근할 수 있습니다.

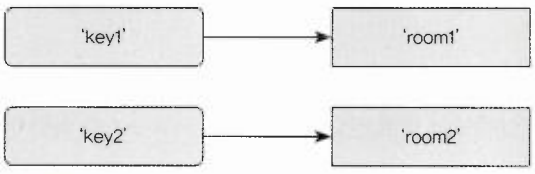


그림 13.1 파이썬 딕셔너리의 키-값 쌍 구조

그렇다면 표 13.1과 같이 카카오의 5일간의 종가 데이터가 있을 때 이를 파이썬으로 어떻게 표현할 수 있을까요?

표 13.1 카카오의 5일간 종가 데이터

날짜	종가
2016-02-19	92,600
2016-02-18	92,400
2016-02-17	92,100
2016-02-16	94,300
2016-02-15	92,300

표 13.1의 데이터를 살펴보면 각 날짜에 해당하는 종가 데이터가 존재합니다. 표 13.1을 통해 알 수 있는 것은 각 날짜에 해당하는 종가입니다. 먼저 파이썬의 리스트 자료구조를 사용해 날짜순으로 데이터를 저장하는 경우를 생각해봅시다.

```
kakao_daily_closing_prices = [92300, 94300, 92100, 92400, 92600]

for price in kakao_daily_closing_prices:
    print(price)
```

파이썬의 리스트로도 충분히 종가를 저장할 수 있지만 종가에 대응되는 날짜에 대한 데이터는 제대로 저장할 수 없습니다. 표 13.1과 같이 어떤 두 데이터가 서로 연관된 경우에는 리스트 자료구조보다는 '키-값 쌍' 구조를 가진 파이썬 딕셔너리를 사용하는 것이 좋습니다.

다음 코드와 같이 표 13.1의 데이터를 딕셔너리로 표현하면 날짜와 종가 데이터 각각을 모두 저장할 수 있으며, 날짜에 해당하는 종가도 서로 연관 지어 저장할 수 있습니다. 이처럼 각 데이터에 효과적인 자료구조를 사용하는 것은 상당히 중요한 일입니다.

```
kakao_daily_closing_prices = {'2016-02-19': 92600,
                              '2016-02-18': 92400,
                              '2016-02-17': 92100,
                              '2016-02-16': 94300,
                              '2016-02-15': 92300}
print(kakao_daily_closing_prices['2016-02-19'])
```

Series 기초

앞서 pandas에는 Series와 DataFrame이라는 두 종류의 자료구조가 있다고 했습니다. pandas의 Series는 1차원 배열과 같은 자료구조입니다. 파이썬 리스트와 튜플도 1차원 배열과 같은 자료구조인데 왜 pandas에서는 Series라는 자료구조를 추가로 만든 것일까요?

사실 pandas의 Series는 어떤 면에서는 파이썬의 리스트와 비슷하고 어떤 면에서는 파이썬의 딕셔너리와 닮은 알쏭달쏭한 자료구조입니다. 지금부터 간단히 코드를 작성해가면서 Series라는 자료구조를 공부해 봅시다.

먼저 Series를 사용하기에 앞서 pandas라는 모듈에서 직접 Series와 DataFrame을 로컬 네임스페이스로 임포트합시다.

```
from pandas import Series, DataFrame
```

이미 앞서 여러 번 설명한 것처럼 모듈을 임포트하는 방식에는 크게 세 가지가 있는데, 그중 하나를 선택해서 사용하면 됩니다. pandas 모듈 내의 Series를 사용할 때도 꼭 위와 같은 방식으로 임포트할 필요는 없습니다. 다음과 같은 방식도 사용 가능한데 이 경우에는 Series를 사용하고자 할 때 pandas.Series와 같이 이름을 적어줘야 합니다. 이와 달리 Series를 직접 로컬 네임스페이스로 임포트한 경우에는 pandas는 생략하고 바로 Series라고만 적으면 됩니다.

```
import pandas
print(pandas.Series)
```

보통 pandas 라이브러리를 사용할 때 Series와 DataFrame이라는 이름은 자주 사용하기 때문에 Series와 DataFrame을 로컬 네임스페이스로 직접 임포트하는 방식을 더 많이 사용합니다.

표 13.1의 카카오 종가 데이터를 이번에는 Series를 사용해서 저장해 봅시다. pandas의 Series는 사실 클래스입니다. 생성자로 다음과 같이 파이썬 리스트를 넘겨주면 해당 값에 포함하는 Series 객체를 생성해줍니다.

```
from pandas import Series, DataFrame

kakao = Series([92600, 92400, 92100, 94300, 92300])
print(kakao)
```

Series 객체를 생성할 때 따로 인덱스를 지정하지 않았다면 기본적으로 0부터 시작하는 정숫값으로 인덱싱됩니다.

```
0    92600
1    92400
2    92100
3    94300
4    92300
dtype: int64
```

그림 13.2는 kakao라는 변수가 바인딩하고 있는 Series 객체의 내부 구조를 그림으로 나타낸 것입니다. 그림 13.2처럼 Series 객체는 일차원 배열과 달리 값뿐만 아니라 각 값에 연결된 인덱스 값도 동시에 저장합니다.

Series	
Index	Value
0	92600
1	92400
2	92100
3	94300
4	92300

그림 13.2 Series 객체의 내부 구조

앞의 예제 코드에서처럼 Series 객체를 생성할 때 인덱스 값을 따로 지정하지 않으면 기본적으로 Series 객체는 0부터 시작하는 정숫값을 사용해 인덱싱합니다. 따라서 다음과 같이 정수 인덱스 값을 통해 데이터에 접근할 수 있습니다.

```
print(kakao[0])
print(kakao[2])
print(kakao[4])
```

출력 결과를 살펴보면 Series 객체를 생성할 때 인자로 넘겨준 리스트 내의 데이터 순서를 기준으로 인덱싱돼 있음을 알 수 있습니다.

```
92600
92100
92300
```

지금까지 배운 내용만으로는 Series 자료구조가 파이썬의 리스트와 별반 달라 보이지 않습니다. Series 객체에 데이터가 저장되며, 저장된 데이터는 0부터 순서대로 인덱싱됩니다. 특히 리스트와 마찬가지로 표 13.1에서 날짜 데이터가 저장되지 않았습니다. 그렇지만 이 정도로 아직 실망하긴 이릅니다.

Series 객체는 파이썬 리스트와 달리 인덱싱 값을 지정할 수 있습니다. 예를 들어, 표 13.1의 데이터에 대한 Series 객체를 생성하려면 다음 예제 코드처럼 작성하면 됩니다. index에 인덱싱할 값을 넘겨주면 해당 값으로 인덱싱됩니다.

```
kakao2 = Series([92600, 92400, 92100, 94300, 92300], index=['2016-02-19',
                                                            '2016-02-18',
                                                            '2016-02-17',
                                                            '2016-02-16',
                                                            '2016-02-15'])
print(kakao2)
```

kakao2 객체의 출력 값을 보면 데이터가 어떻게 저장돼 있는지 확인할 수 있습니다. Series 객체 내의 데이터는 1차원으로 저장돼 있지만 파이썬의 딕셔너리와 유사하게 값과 이에 대응되는 인덱스 값이 서로 연결돼 있습니다.

```
2016-02-19    92600
2016-02-18    92400
2016-02-17    92100
2016-02-16    94300
2016-02-15    92300
dtype: int64
```

kakao2 객체(정확히는 kakao2가 바인딩하고 있는 Series 객체)를 그림으로 나타내면 그림 13.3과 같습니다. 그림 13.3과 그림 13.2를 비교해보면 Value 부분은 같지만 각 Value에 대한 인덱스 값이 다른 것을 확인할 수 있습니다.

Series	
Index	Value
'2016-02-19'	92600
'2016-02-18'	92400
'2016-02-17'	92100
'2016-02-16'	94300
'2016-02-15'	92300

그림 13.3 Series 객체의 내부 구조(2)

kakao2라는 Series 객체는 인덱스 값으로 날짜에 해당하는 문자열을 지정했기 때문에 정숫값으로 인덱싱하는 대신 날짜를 의미하는 문자열을 사용해 각 날짜에 대한 종가를 바로 얻어올 수 있습니다.

```
print(kakao2['2016-02-19'])
print(kakao2['2016-02-18'])
```

위 코드의 출력 값을 확인해보면 인덱스 값으로 사용한 날짜에 대한 종가가 정상적으로 출력된 것을 확인할 수 있습니다.

```
92600
92400
```

pandas의 Series 객체에는 그림 13.2나 13.3과 같이 인덱스와 값이 저장돼 있는데 Series 객체의 index와 values라는 이름의 속성을 통해 접근할 수 있습니다. 예를 들어, kakao2 객체의 인덱스 값과 저장된 종가를 각각 출력하는 코드는 다음과 같이 작성할 수 있습니다.

```
for date in kakao2.index:
    print(date)

for closing_price in kakao2.values:
    print(closing_price)
```

위 코드를 실행한 결과는 다음과 같습니다.

```
2016-02-19
2016-02-18
2016-02-17
2016-02-16
2016-02-15
92600
92400
92100
```

```
94300
92300
```

이번에는 Series 객체의 덧셈 연산에 대해 알아보겠습니다. 먼저 여러분이 네이버, SK, KT에 대해 10, 20, 30주의 주식을 보유하고 있다고 가정해 봅시다. 그리고 여러분의 친구는 KT, 네이버, SK에 대해 각각 10, 30, 20주를 보유하고 있습니다. 여러분과 여러분의 친구가 보유하고 있는 네이버, SK, KT의 주식은 몇 주일까요?

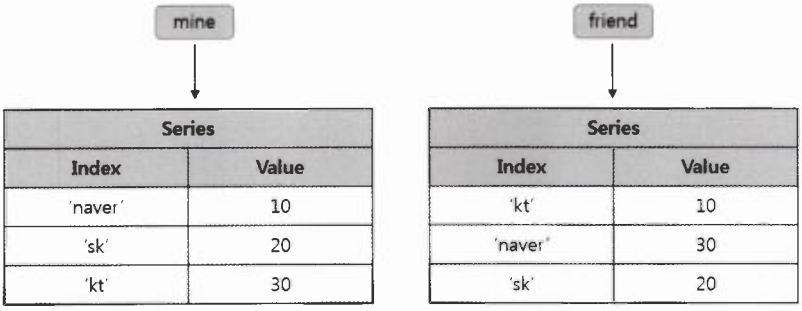
이 같은 경우는 우리 주변에서도 자주 발생합니다. 일반적으로 어떤 데이터를 여러 명이 나눠서 취합하게 되면 데이터를 서로 다르게 인덱싱하는 경우가 종종 발생합니다. pandas의 Series는 서로 다르게 인덱싱된 데이터에 대해서도 알아서 덧셈 연산을 처리해 줍니다.

예를 들어, 앞서 예시로 들었던 네이버, SK, KT의 주식 보유 현황은 Series로 다음과 같이 표현할 수 있습니다.

```
from pandas import Series, DataFrame

mine   = Series([10, 20, 30], index=['naver', 'sk', 'kt'])
friend = Series([10, 30, 20], index=['kt', 'naver', 'sk'])
```

이 코드를 그림으로 나타내면 그림 13.4와 같습니다. 그림 13.4를 참조하면 mine이라는 변수와 friend라는 변수가 바인딩하고 있는 Series 객체의 인덱싱 순서가 서로 다른 것을 확인할 수 있습니다.



Series	
Index	Value
'naver'	10
'sk'	20
'kt'	30

Series	
Index	Value
'kt'	10
'naver'	30
'sk'	20

그림 13.4 Series 객체의 구조(3)

pandas의 Series는 그림 13.4와 같이 인덱싱이 서로 다른 경우에도 알아서 인덱싱이 같은 값끼리 덧셈 연산을 수행합니다. 다음과 같이 mine과 friend를 더해서 merge를 만들어 봅시다.

```
merge = mine + friend
print(merge)
```

위 코드의 출력 값을 확인해보면 같은 인덱스를 갖는 값끼리 정상적으로 덧셈이 수행된 것을 확인할 수 있습니다.

```
kt       40
naver    40
sk       40
dtype: int64
```

Python Console

이번 절에서는 PyCharm에서 Python Console을 사용하는 방법을 다루겠습니다. pandas를 설명하다가 갑자기 웬 PyCharm이냐 하시는 분도 있겠지만 pandas를 효과적으로 사용하기 위한 개발 환경을 잠시 소개한다고 생각하면 되겠습니다.

사실 pandas 기초를 공부할 때는 PyCharm과 같은 IDE보다는 Python IDLE이나 그림 13.5와 같은 Jupyter QtConsole(IPython Qt 버전)이 더 적합합니다. 왜냐하면 Python IDLE이나 Jupyter QtConsole과 같은 대화식 인터프리터로 바로바로 결괏값을 확인할 수 있기 때문입니다.

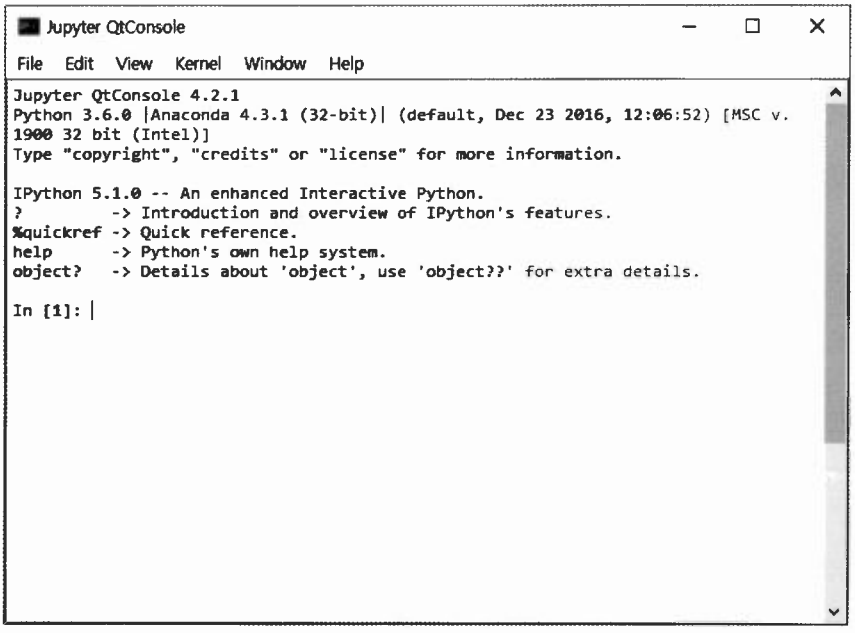


그림 13.5 Jupyter QtConsole

그러나 이미 PyCharm에 익숙해진 분들은 PyCharm을 계속 사용하고 싶을 것입니다. 이럴 때는 그림 13.6과 같이 PyCharm에서 [Tools] → [Python Console] 메뉴를 차례로 클릭합니다.

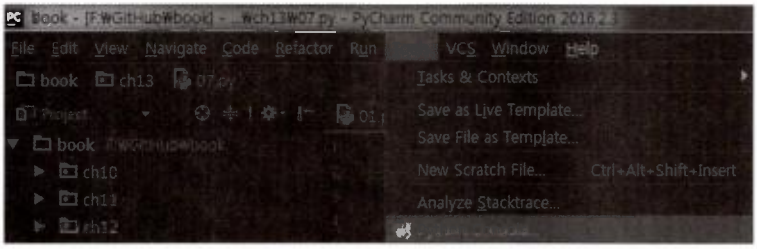


그림 13.6 PyCharm의 Python Console 실행 메뉴

그림 13.7과 같이 PyCharm 하단에 IPython 콘솔이 실행되는 것을 확인할 수 있습니다.



그림 13.7 IPython console 실행 화면

PyCharm이 제공하는 편리한 기능 중 하나는 PyCharm IDE에서 작성한 코드를 바로 Python Console에서 실행할 수 있다는 것입니다. 그림 13.8과 같이 실행하려는 코드를 선택한 후 마우스 오른쪽 버튼을 누릅니다. 그리고 [Execute Selection in Console] 메뉴를 클릭하면 선택된 코드가 Python Console에서 실행되는 것을 확인할 수 있습니다.

PyCharm에서 Python Console도 사용하는 법을 배웠으니 이번 절에서 배운 내용을 직접 타이핑하면서 복습해보기 바랍니다.

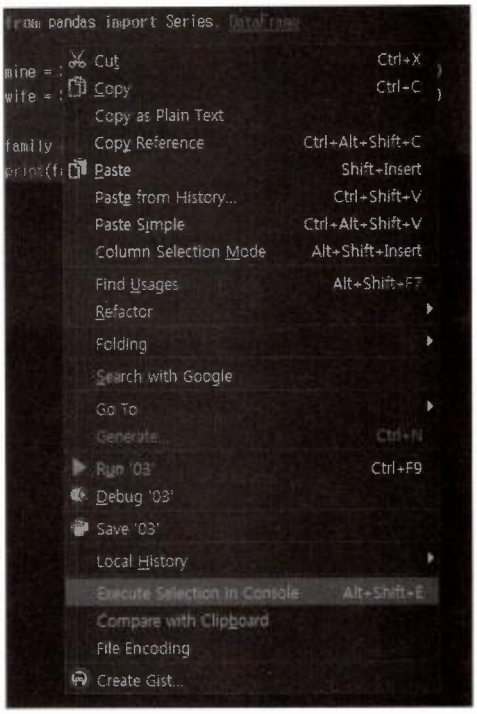
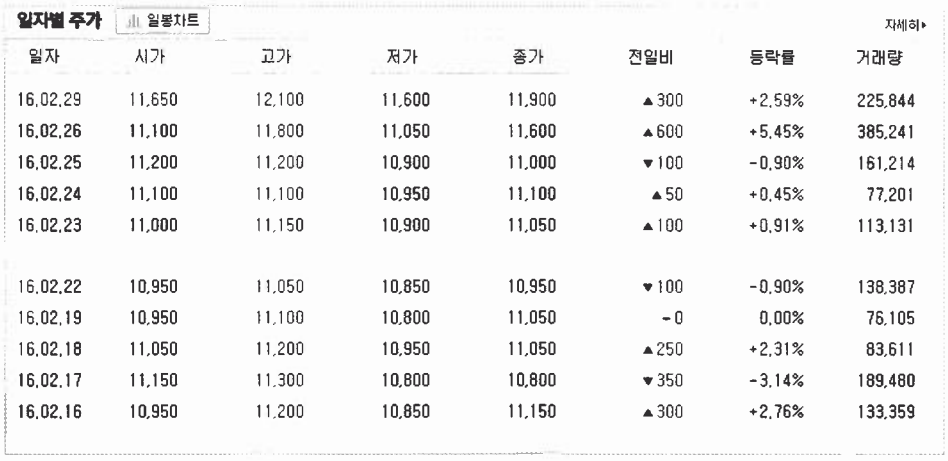


그림 13.8 Python Console에서 코드 실행하기

02 pandas DataFrame

이번 절에서는 DataFrame이라는 pandas의 자료구조를 살펴보겠습니다. pandas의 Series가 1차원 형태의 자료구조라면 DataFrame은 여러 개의 칼럼(Column)으로 구성된 2차원 형태의 자료구조입니다. 아직까지는 2차원 형태의 자료구조가 어떻게 생겼고 그것이 왜 필요한지 감이 쉽게 오지는 않지요?

그림 13.9는 대신증권 종목에 대한 2주간의 일자별 주가 데이터입니다. 로우(row)는 일자로 구성돼 있으며 칼럼에는 시가, 고가, 저가, 종가, 전일비, 등락률, 거래량이 있습니다. 우리는 일상생활에서 그림 13.9와 같은 형태의 데이터를 많이 사용하고 있습니다.



일자별 주가 | 일봉차트 | 자세히▸

일자	시가	고가	저가	종가	전일비	등락률	거래량
16.02.29	11,650	12,100	11,600	11,900	▲300	+2.59%	225,844
16.02.26	11,100	11,800	11,050	11,600	▲600	+5.45%	385,241
16.02.25	11,200	11,200	10,900	11,000	▼100	-0.90%	161,214
16.02.24	11,100	11,100	10,950	11,100	▲50	+0.45%	77,201
16.02.23	11,000	11,150	10,900	11,050	▲100	+0.91%	113,131
16.02.22	10,950	11,050	10,850	10,950	▼100	-0.90%	138,387
16.02.19	10,950	11,100	10,800	11,050	-0	0.00%	76,105
16.02.18	11,050	11,200	10,950	11,050	▲250	+2.31%	83,611
16.02.17	11,150	11,300	10,800	10,800	▼350	-3.14%	189,480
16.02.16	10,950	11,200	10,850	11,150	▲300	+2.76%	133,359

그림 13.9 2차원 형태의 데이터(출처: 다음 증권)

pandas의 DataFrame을 사용하면 그림 13.9와 같이 로우와 칼럼으로 구성된 2차원 구조의 데이터를 쉽게 저장하고 조작할 수 있습니다.

DataFrame 생성

DataFrame 객체를 생성하는 가장 쉬운 방법은 파이썬의 딕셔너리를 사용하는 것입니다. 딕셔너리를 통해 각 칼럼에 대한 데이터를 저장한 후 딕셔너리를 DataFrame 클래스의 생성자 인자로 넘겨주면 DataFrame 객체가 생성됩니다.

예제 13.1을 참조하면 raw_data 딕셔너리는 'col0', 'col1', 'col2'라는 키 값을 가지며 각 키는 리스트 타입의 값을 가집니다. 이를 DataFrame의 생성자로 넘겨주면 DataFrame 객체가 생성되는데 생성된 객체를 data라는 변수가 바인딩합니다.

예제 13.1 딕셔너리를 이용한 DataFrame 객체 생성(소스코드: book/ch13/08.py)

```
from pandas import Series, DataFrame

raw_data = {'col0': [1, 2, 3, 4],
            'col1': [10, 20, 30, 40],
            'col2': [100, 200, 300, 400]}

data = DataFrame(raw_data)
print(data)
```

예제 13.1을 실행한 결과는 다음과 같습니다. 주의해서 살펴볼 부분은 col0, col1, col2라는 세 개의 칼럼이 존재한다는 점입니다. 여기서 'col0', 'col1', 'col2'라는 문자열은 DataFrame의 각 칼럼을 인덱싱하는 데 사용됩니다. 로우 방향으로는 Series와 유사하게 정숫값으로 자동으로 인덱싱된 것을 확인할 수 있습니다.

```
   col0  col1  col2
0     1    10   100
1     2    20   200
2     3    30   300
3     4    40   400
```

그림 13.8을 참조해서 예제 13.1의 코드를 Python Console에서 실행해보기 바랍니다. 먼저 'col0', 'col1', 'col2'를 사용해 각 칼럼을 선택해 봅시다. 파이썬 딕셔너리에서 키를 통해 값에 접근했던 것과 같이 DataFrame 객체의 각 칼럼에 접근할 수 있습니다.

```
In[3]: data['col0']
Out[3]:
0    1
1    2
2    3
3    4
Name: col0, dtype: int64

In[4]: data['col1']
Out[4]:
0    10
1    20
2    30
3    40
Name: col1, dtype: int64

In[5]: data['col2']
Out[5]:
0    100
1    200
2    300
3    400
Name: col2, dtype: int64
```

type 함수를 사용해 DataFrame에 있는 각 칼럼의 타입을 확인해 봅시다. type의 리턴 값을 살펴보면 DataFrame에 있는 각 칼럼은 Series 객체임을 알 수 있습니다. 즉, DataFrame을 인덱스가 같은 여러 개의 Series 객체로 구성된 자료구조로 생각해도 좋습니다.

```
In[6]: type(data['col0'])
Out[6]: pandas.core.series.Series
```

그림 13.10은 data라는 변수가 바인딩하고 있는 DataFrame 객체의 내부 구조를 그림으로 나타낸 것입니다. DataFrame 객체는 'col0', 'col1', 'col2'라는 세 개의 Series 객체로 구성되는데, Series 객체의 인덱스는 서로 동일합니다.

data

DataFrame			
	Series ('col0')	Series ('col1')	Series ('col2')
Index	Value	value	Value
0	1	10	100
1	2	20	200
2	3	30	300
3	4	40	400

그림 13.10 DataFrame 객체의 내부 구조

data라는 변수가 바인딩하는 DataFrame 객체를 그림 13.11과 같이 표현해 볼 수도 있습니다. DataFrame에는 3개의 Series 객체가 있습니다. 이는 'col0', 'col1', 'col2'라는 키에 각각 대응되는 값이고 이것들을 하나의 파이썬 딕셔너리 객체로 생각하는 것입니다. 따라서 'col0', 'col1', 'col2'라는 키를 통해 값에 해당하는 Series 객체에 접근할 수 있는 것입니다.

입력 데이터와의 구조적 유사성(2차원) 관점에서 바라보면 DataFrame이 그림 13.10과 같이 구성된다고 생각하는 것이 편하지만 DataFrame의 각 칼럼에 대한 접근 방식 관점에서 바라보면 DataFrame이 그림 13.11과 같이 구성된다고 생각하는 것이 더 편리합니다. 두 그림을 모두 기억해 둔다면 DataFrame이라는 자료구조를 더욱 쉽게 이해할 수 있을 것입니다.

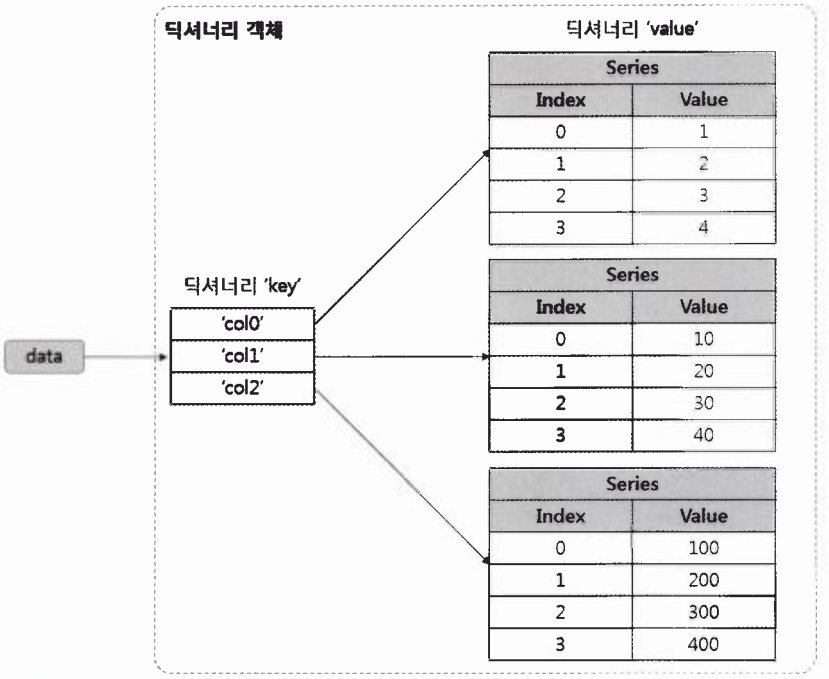


그림 13.11 DataFrame 객체의 구조(2)

이번에는 그림 13.9의 데이터에서 16.02.23~16.02.29일 사이의 일자별 주가 중 시가, 고가, 저가, 종가를 DataFrame으로 저장해 봅시다. 참고로 시가, 고가, 저가, 종가는 영어로 open, high, low, close라고 부르며 약어로 OHLC라고 하기도 합니다.

DataFrame 객체를 생성하기 위해 예제 13.2의 코드와 같이 파이썬 딕셔너리 형태로 각 칼럼에 대응하는 데이터를 표현한 후 이를 DataFrame 클래스의 생성자로 넘겨줍니다.

예제 13.2 딕셔너리를 이용한 DataFrame 객체 생성(2)(소스코드: book/ch13/09.py)

```
from pandas import Series, DataFrame

daeshin = {'open':  [11650, 11100, 11200, 11100, 11000],
           'high':  [12100, 11800, 11200, 11100, 11150],
           'low' :  [11600, 11050, 10900, 10950, 10900],
           'close': [11900, 11600, 11000, 11100, 11050]}

daeshin_day = DataFrame(daeshin)
print(daeshin_day)
```

예제 13.2를 실행해 생성한 DataFrame 객체 값을 확인해 봅시다. 출력 값을 살펴보면 시가, 고가, 저가, 종가 데이터가 일단 2차원 형태로 잘 저장된 것을 확인할 수 있습니다. 다만 그림 13.9와 비교해보면 칼럼의 순서가 그림과 다른 것을 알 수 있습니다. 그리고 Series 객체의 인덱스가 날짜 대신 정숫값으로 돼 있습니다.

```
    close   high    low   open
0   11900   12100   11600   11650
1   11600   11800   11050   11100
2   11000   11200   10900   11200
3   11100   11100   10950   11100
4   11050   11150   10900   11000
```

DataFrame 객체에서 칼럼의 순서는 DataFrame 객체를 생성할 때 columns라는 키워드를 지정할 수 있습니다.

```
daeshin_day = DataFrame(daeshin, columns=['open', 'high', 'low', 'close'])
```

새롭게 생성된 DataFrame 객체를 출력해보면 그림 13.9와 동일한 순서로 칼럼이 구성된 것을 확인할 수 있습니다.

```
    open   high    low  close
0   11650   12100   11600   11900
1   11100   11800   11050   11600
2   11200   11200   10900   11000
3   11100   11100   10950   11100
4   11000   11150   10900   11050
```

DataFrame에서 인덱스 역시 DataFrame 객체를 생성하는 시점에 index를 통해 지정할 수 있습니다. 먼저 인덱싱에 사용할 값을 만든 후 이를 DataFrame 객체 생성 시점에 지정하면 됩니다.

```
date = ['16.02.29', '16.02.26', '16.02.25', '16.02.24', '16.02.23']
daeshin_day = DataFrame(daeshin, columns=['open', 'high', 'low', 'close'], index=date)
```

새롭게 생성된 DataFrame 객체의 출력 값을 확인해 봅시다. 이번에는 그림 13.9와 일자도 일치하는 것을 확인할 수 있습니다.

```
          open   high    low  close
16.02.29  11650   12100   11600   11900
16.02.26  11100   11800   11050   11600
16.02.25  11200   11200   10900   11000
16.02.24  11100   11100   10950   11100
16.02.23  11000   11150   10900   11050
```

DataFrame 칼럼, 로우 선택

앞서 생성한 DataFrame 객체인 daeshin_day에는 'open', 'high', 'low', 'close'라는 네 개의 칼럼이 존재합니다. 만약 종가를 기준으로만 데이터를 분석한다면 'close' 칼럼에 대한 데이터만 DataFrame 객체로부터 얻어내야 합니다. 앞서 설명한 것처럼 DataFrame 객체의 종가 칼럼은 'close'를 사용해 접근할 수 있습니다.

```
close = daeshin_day['close']
print(close)
```

```
16.02.29    11900
16.02.26    11600
16.02.25    11000
16.02.24    11100
16.02.23    11050
Name: close, dtype: int64
```

이번에는 16년 02월 24일의 시가, 고가, 저가, 종가 데이터를 얻어오는 경우를 생각해 봅시다. 해당 데이터는 DataFrame 객체에서 한 로우에 해당합니다. DataFrame의 2차원 구조만 생각한다면 인덱스를 사용해 다음과 같은 코드도 가능할 것 같기도 합니다.

```
print(daeshin_day['16.02.24'])
```

그러나 위 코드를 실행해보면 그림 13.12와 같은 에러가 발생합니다. 앞서 칼럼에 접근할 때 'close'라는 값을 사용한 것처럼 '16.02.04'와 같은 값을 사용해서 접근하면 pandas는 이 값을 칼럼의 키 값으로 판단합니다. 그러나 daeshin_day라는 DataFrame 객체에는 '16.02.04'라는 키 값을 갖는 칼럼이 없으므로 에러가 발생하는 것입니다.

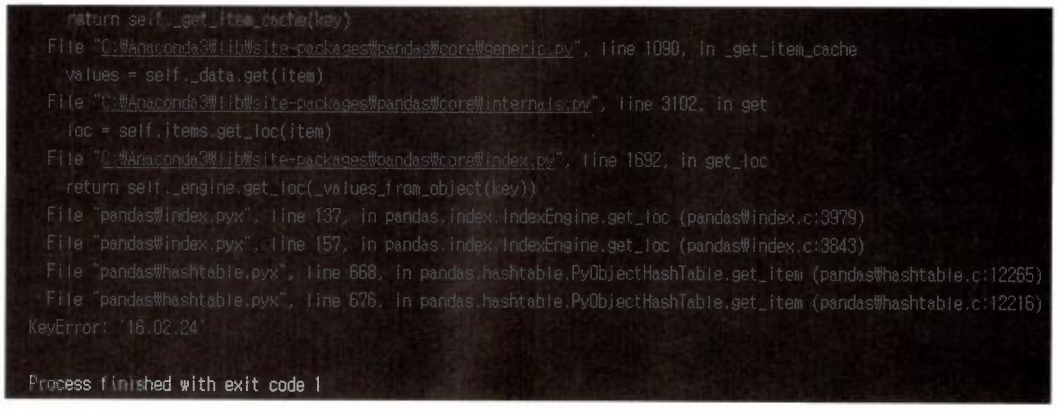


그림 13.12 DataFrame 키 에러

DataFrame 객체의 로우에 접근하려면 ix 메서드를 사용해 인덱스 값을 넘겨주면 됩니다. 예를 들어, 인덱스 값 '16.02.24'를 전달하면 해당 값을 가진 로우 데이터를 얻어올 수 있습니다.

```
day_data = daeshin_day.ix['16.02.24']
print(day_data)
print(type(day_data))
```

위 파이썬 코드의 실행 결과를 확인해보면 정상적으로 16년 02월 24일자 대신증권 종목의 시가, 고가, 저가, 종가를 얻어온 것을 확인할 수 있습니다. 그리고 반환된 객체는 pandas의 Series 타입임을 확인할 수 있습니다.

```
open     11100
high     11100
low      10950
close    11100
Name: 16.02.24, dtype: int64
<class 'pandas.core.series.Series'>
```

정리해보면 DataFrame 객체의 칼럼에 접근하려면 칼럼 이름을 지정하면 되고, 로우에 접근하려면 ix 메서드를 통해 인덱스 값을 지정하면 됩니다.

DataFrame 객체의 칼럼 이름과 인덱스 값을 확인하려면 각각 columns와 index 속성을 사용하면 됩니다.

```
print(daeshin_day.columns)
print(daeshin_day.index)

Index(['open', 'high', 'low', 'close'], dtype='object')
Index(['16.02.29', '16.02.26', '16.02.25', '16.02.24', '16.02.23'], dtype='object')
```

03 주식 데이터 받기

이미 여러분은 국내 증권사의 API를 이용해 주식 데이터를 받아오는 법을 배웠습니다. 실시간 데이터라면 증권사의 API를 사용하는 것이 편리하지만 일봉 데이터 정도는 데이터의 양이 많지 않기 때문에 야후(www.yahoo.com)나 구글 (www.google.com) 같은 인터넷 웹사이트를 통해 구할 수도 있습니다.

특히 pandas에서는 야후나 구글에서 제공하는 데이터를 쉽게 DataFrame 객체로 가져오는 기능을 제공합니다. 따라서 분 단위나 틱 단위의 데이터가 아니라 일봉 기준의 데이터가 필요하다면 증권사를 거치지 않고 pandas만으로 주식 데이터를 쉽게 얻어올 수 있습니다.

pandas는 현재도 계속 개발되고 있는 라이브러리입니다. 데이터를 읽어오는 데 사용하는 서브 패키지가 pandas 0.17.0 버전부터는 pandas에서 분리되어 pandas-datareader라는 이름의 독립 패키지가 됐습니다. 따라서 pandas-datareader라는 패키지를 추가로 설치해야 합니다.

먼저 윈도우의 [모든 프로그램] 메뉴에서 그림 13.13과 같이 [Anaconda Prompt]를 메뉴를 선택해 아나콘다 프롬프트를 실행합니다.



그림 13.13 Anaconda Prompt 실행 메뉴

다음으로 그림 13.14와 같이 conda 명령을 통해 pandas-datareader라는 패키지를 설치합니다. 이처럼 아나콘다 배포판은 패키지를 쉽게 설치하고 관리할 수 있는 conda 명령어를 제공합니다.



그림 13.14 conda install

그림 13.15와 같이 conda는 pandas-datareader 패키지를 설치하기 위해 추가로 설치해야 하는 패키지 및 기존 패키지 중 업데이트가 필요한 패키지 목록을 보여줍니다. pandas-datareader 패키지를 설치하려면 키보드의 Y 키를 눌러 설치를 시작합니다.

만약 conda install이 없었다면 패키지에 필요한 추가 패키지 설치 및 업데이트를 수동으로 진행해야 합니다. 아나콘다 배포판을 사용하니 여러모로 편리하지요?

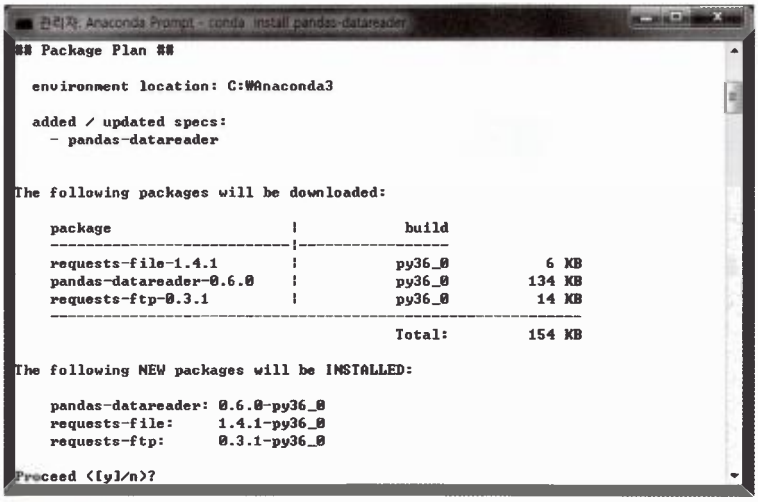


그림 13.15 pandas-datareader 패키지 설치

pandas-datareader 패키지를 사용하기에 앞서 에디터를 바꿔 보겠습니다. 아무래도 PyCharm 같은 IDE 스타일보다는 대화식으로 결과를 바로 볼 수 있는 인터프리터가 pandas-datareader를 공부하는 데 더 편리합니다. 윈도우의 시작 버튼을 누른 다음 [모든 프로그램] → [Anaconda3] → [Anaconda Navigator]를 차례로 클릭한 후 그림 13.16과 같이 qtconsole에서 [Launch]를 클릭합니다.

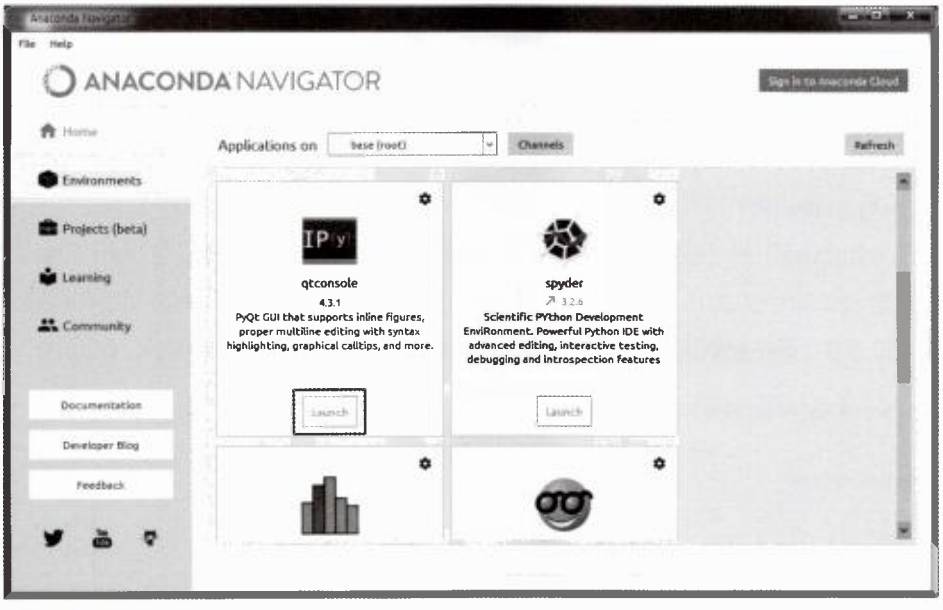


그림 13.16 Jupyter QTConsole

pandas_datareader 패키지의 data 모듈은 인터넷상에서 제공하는 다양한 데이터를 DataFrame으로 만들어주는 기능을 제공합니다. 해당 모듈을 사용하기 위해 그림 13.17과 같이 임포트합니다.

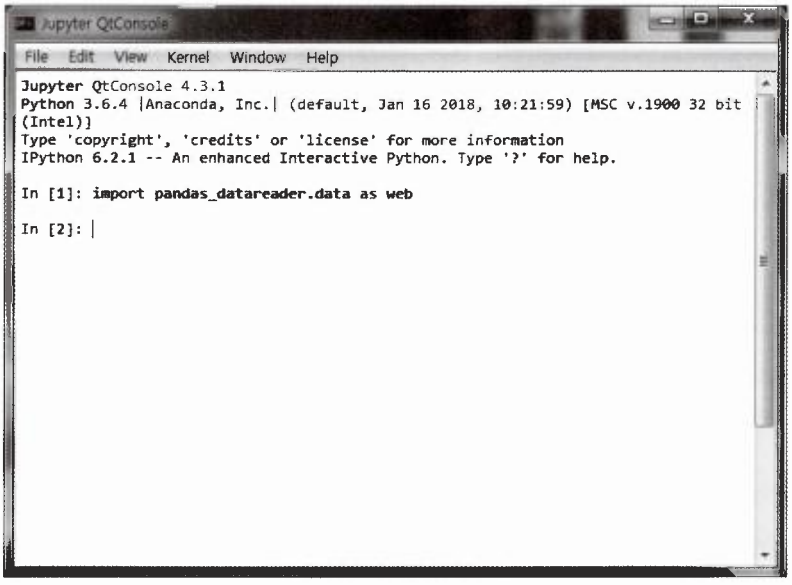


그림 13.17 pandas_datareader 패키지의 data 모듈 임포트

DataReader 사용하기

pandas_datareader 패키지의 DataReader라는 함수는 웹 상의 데이터를 DataFrame 객체로 만드는 기능을 제공합니다. 이번 절에서는 DataReader를 이용해 야후 파이낸스로부터 GS 종목의 일봉 데이터를 받아와 보겠습니다.

먼저 pandas_datareader 패키지의 data 모듈을 임포트한 후 추가로 datetime이라는 모듈을 임포트합니다. 참고로 datetime은 파이썬에서 날짜를 쉽게 표현하는 데 사용하는 모듈입니다. datetime 모듈을 이용해 조회 시작 날짜인 2016-02-19와 조회 종료일인 2016-03-04에 해당하는 변수를 만듭니다. 여기서 주의할 점은 2월과 3월을 입력할 때 02, 03이 아니라 2, 3으로 입력해야 한다는 점입니다.

```
In [1]: import pandas_datareader.data as web

In [2]: import datetime

In [3]: start = datetime.datetime(2016, 2, 19)

In [4]: end = datetime.datetime(2016, 3, 4)
```

조회할 시작일과 종료일을 설정했으니 DataReader 함수를 이용해 데이터를 조회해 봅시다. 조회할 종목인 GS의 종목코드는 '078930'입니다. 따라서 다음과 같이 DataReader 함수의 인자를 입력합니다.

DataReader 함수의 첫 번째 인자로는 조회할 종목에 대한 정보를 넣고, 두 번째 인자로는 데이터를 가져올 소스(구글)에 대한 정보를 입력했습니다. 세 번째와 네 번째 인자에는 조회 기간의 시작일과 종료일을 입력합니다.

```
In [5]: gs = web.DataReader("KRX:078930", "google", start, end)
```

DataReader 함수는 DataFrame 객체를 반환합니다. gs가 바인딩하고 있는 값을 확인해보면 DataFrame 객체에 값이 저장돼 있는 것을 확인할 수 있습니다.

```
In [6]: gs
Out[6]:
             Open     High     Low      Close    Volume
Date
2016-02-19   50300    51100    49950    50600    301800
2016-02-22   50300    50800    49850    50400    153900
2016-02-23   50800    53000    50800    52800    431900
2016-02-24   52000    53500    51900    53500    259600
2016-02-25   53900    54700    52700    53900    170800
2016-02-26   54300    54500    53200    53300    159400
2016-02-29   53000    53900    52800    53000    154600
2016-03-02   53900    55200    53500    55100    229900
2016-03-03   55300    55800    54300    55000    217000
2016-03-04   54700    54800    53200    54200    237900
```

그림 13.18은 다음 증권 사이트로부터 해당 기간의 일자별 주가를 출력한 것입니다. 두 데이터를 서로 비교해보면 값이 일치하는 것을 확인할 수 있습니다. 다만 거래량 데이터에서 약간 차이가 나는 것을 확인할 수 있습니다.

구글 파이낸스는 종가와 수정 종가 중 종가만 제공합니다. 수정 종가는 유상/무상증자, 액면분할 등이 있을 때 이를 고려해 주식 가격을 조정(adjustment)해둔 가격입니다. 보통 데이터 분석에는 이전 데이터와의 관계를 분석하는 것이 중요하기 때문에 데이터의 연속성을 위해 수정 종가를 사용해야 합니다.

일자별 주가 | 일봉차트 | 자세히▸

일자	시가	고가	저가	종가	전일비	등락률	거래량
16.03.04	54,700	54,800	53,200	54,200	▼800	-1.45%	237,791
16.03.03	55,300	55,800	54,300	55,000	▼100	-0.18%	217,063
16.03.02	53,900	55,200	53,500	55,100	▲2,100	+3.96%	229,999
16.02.29	53,000	53,900	52,800	53,000	▼300	-0.56%	154,692
16.02.26	54,300	54,500	53,200	53,300	▼600	-1.11%	159,453
16.02.25	53,900	54,700	52,700	53,900	▲400	+0.75%	170,839
16.02.24	52,000	53,500	51,900	53,500	▲700	+1.33%	259,616
16.02.23	50,800	53,000	50,800	52,800	▲2,400	+4.76%	431,934
16.02.22	50,300	50,800	49,850	50,400	▼200	-0.40%	153,961
16.02.19	50,300	51,100	49,950	50,600	▼300	-0.59%	301,800

그림 13.18 GS 종목의 일자별 주가(출처: 다음 증권)

차트 그리기

앞 절에서 DataReader 함수를 이용해 야후 파이낸스로부터 일봉 데이터를 DataFrame 객체로 저장하는 방법을 배웠습니다. 데이터를 적당한 자료구조로 저장했다면 데이터를 분석하면 되는데 이때 가장 중요한 점은 데이터를 시각적으로 표현해보는 것입니다. 주식의 경우에도 차트 형태로 시각화한 후 차트를 가지고 주식 데이터를 분석하는 경우가 많습니다. 이번 절에서는 파이썬으로 주식 차트를 그리는 방법을 배워보겠습니다.

차트를 그리기 위해서는 먼저 차트로 표현하고자 하는 데이터가 있어야 합니다. 앞서 배운 것처럼 DataReader 함수를 이용해 GS 종목 데이터를 얻어옵니다.

```
In [1]: import pandas_datareader.data as web

In [2]: import datetime

In [3]: start = datetime.datetime(2016, 2, 19)

In [4]: end = datetime.datetime(2016, 3, 4)

In [5]: gs = web.DataReader("KRX:078930", "google", start, end)
```

DataFrame 객체에서 info 메서드를 호출하면 DataFrame 객체를 요약해서 볼 수 있습니다. gs라는 DataFrame 객체에는 현재 11개의 엔트리만이 존재합니다. 그래프로 그리기에는 11개라는 데이터의 수는 너무 적습니다.

```
In [9]: gs.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 11 entries, 2016-02-19 to 2016-03-04
Data columns (total 5 columns):
Open 11 non-null float64
High 11 non-null float64
Low 11 non-null float64
Close 11 non-null float64
Volume 11 non-null int64
dtypes: float64(4), int64(1)
memory usage: 480.0 bytes
```

DataReader 함수를 사용해 더 많은 데이터를 받아오기 위해 다음과 같이 함수를 다시 호출합니다.

```
In [10]: gs = web.DataReader("KRX:078930", "google")
```

DataReader 함수를 호출할 때 명시적으로 조회 기간을 입력하지 않으면 2010/1/1일부터 데이터를 조회한 날까지의 데이터를 얻어옵니다. info 메서드를 호출하면 이번에는 2,022개의 엔트리가 있음을 확인할 수 있습니다. 이 정도면 그래프로 충분히 그려볼 수 있겠습니다.

```
In [11]: gs.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2022 entries, 2010-01-04 to 2018-03-09
Data columns (total 5 columns):
Open 2022 non-null float64
High 2022 non-null float64
Low 2022 non-null float64
Close 2022 non-null float64
Volume 2022 non-null int64
dtypes: float64(4), int64(1)
memory usage: 94.8 KB
```

데이터 분석을 위해 pandas라는 패키지를 사용하는 것처럼 파이썬에서 그래프를 그릴 때는 matplotlib라는 패키지를 주로 사용합니다. 특히 그래프는 matplotlib 패키지의 pyplot이라는 모듈을 사용합니다. 해당 모듈을 사용하기 위해 다음과 같이 모듈을 임포트합니다.

```
In [12]: import matplotlib.pyplot as plt
```

pyplot 모듈에는 그래프를 그리는 plot 함수가 있습니다. 다음과 같이 plot 함수를 호출하면 되는데, 이때 그래프로 표현하려는 데이터를 인자로 전달하면 됩니다. DataFrame에서 'Close'라는 칼럼 데이터를 넘겨줍니다.

```
In [15]: plt.plot(gs['Close'])
Out[15]: [<matplotlib.lines.Line2D at 0xafd5290>]
```

그림 13.19와 같이 Jupyter QtConsole 내에 그래프가 출력됩니다.



그림 13.19 GS 수정 종가 그래프

그래프를 별도의 창으로 출력하게 하려면 그림 13.20과 같이 '%matplotlib qt'를 한 번 입력한 후 그래프를 그리면 됩니다. 다시 Jupyter QtConsole 내에 출력되도록 설정하려면 '%matplotlib inline'을 입력하면 됩니다.

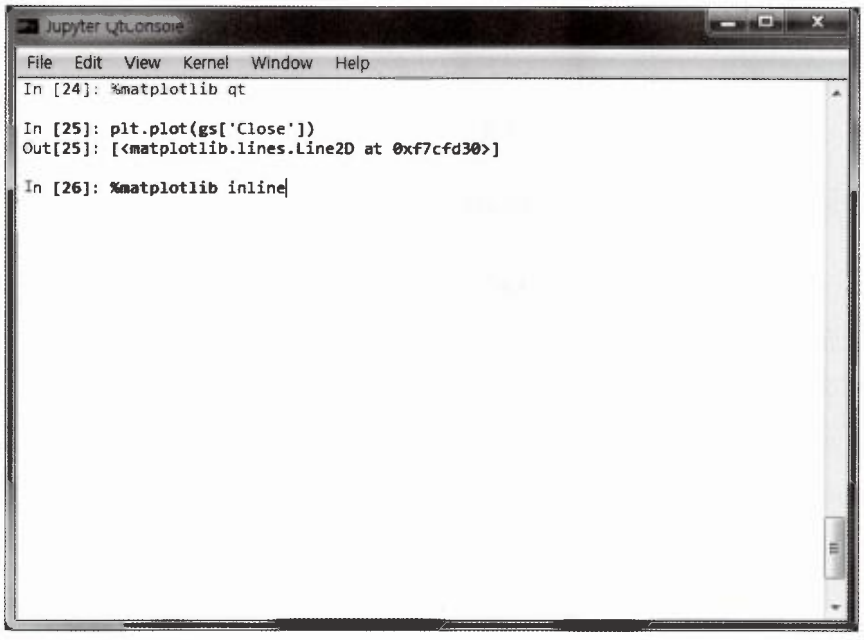


그림 13.20 matplotlib inline 설정

04 이동평균선 구하기

주식 분석 기법은 기본적 분석(fundamental analysis)과 기술적 분석(technical analysis)으로 나눌 수 있습니다. 두 분석 기법은 이론적 입장이 서로 달라 서로 다른 진영의 분석 기법에 대해 종종 비방하기도 합니다. 하지만 분명한 것은 두 분석 기법 모두 의미가 있고 실전에서는 상호 보완적으로 유용하게 활용될 수 있다는 점입니다. 한 예로 많은 사람이 기본적 분석을 통해 투자할 종목을 선정하고 기술적 분석을 통해 매매 시점을 포착하곤 합니다.

초보자들이 가장 처음으로 배우는 기술적 분석지표로 이동평균선이 있습니다. 증권사에 상관없이 HTS에서 5일, 20일, 60일, 120일 이동 평균선이 차트로 제공되고 있으며 시중에 나와 있는 수많은 주식투자서들이 이동평균선에 기반을 두고 매매 시점을 찾는 기법을 소개하고 있습니다.

물론 이동평균선을 활용한 매매법으로 바로 수익을 내는 것은 어려울 수 있지만 이동평균선과 같은 여러 가지 보조 지표를 활용해 새로운 알고리즘을 만드는 데 효과적으로 이용할 수 있을 것입니다. 이번 절에서는 pandas를 이용해 이동평균을 계산하고 이를 차트로 그리는 방법을 배워보겠습니다.

이동평균선

pandas를 이용해 이동평균을 구하고 이를 선으로 그려보기에 앞서 먼저 이동평균선의 정의에 대해 알아봅시다. 네이버의 시사상식 사전을 참고하면 이동평균선의 의미는 다음과 같습니다.

> "일정 기간 동안의 주가를 산술 평균한 값인 주가이동평균을 차례로 연결해 만든 선"

이동평균선의 정의를 잘 살펴보면 이동평균선을 그리려면 먼저 여러 개의 주가이동평균 값이 필요합니다. 여기서 주가이동평균이란 일정 기간의 주가를 산술 평균한 값입니다. 이 기간을 5일로 사용한 것이 5일 주가이동평균이고 이를 연결한 것이 5일 주가이동평균 선입니다.

주가이동평균을 계산하는 데 사용되는 값은 수정 종가(Adjusted Closing Price)입니다. 만약 5일 주가이동평균을 구한다면 주가이동평균 계산일을 기준으로 최근 5일의 수정 종가를 모두 합산한 후 그 값을 5로 나누어 계산합니다. 무슨 말인지 어려워하는 독자분들을 위해 예를 들어 설명해 보겠습니다.

표 13.2는 GS의 16-02-19~16-03-04 기간의 수정 종가 및 이를 사용한 5일 주가이동평균 값을 계산한 것입니다. 5일 주가이동평균 값을 계산하려면 최소 5일의 수정 종가가 필요합니다. 따라서 표 13.2의 데이터 중에서 16-02-19~16-02-24일까지는 5일 주가이동평균을 계산할 수 없습니다.

16-02-25일부터 5일 주가이동평균 계산이 가능한데 16-02-25의 주가이동평균은 16-02-19~16-02-25일 기간의 수정 종가를 모두 더한 후 5로 나누면 됩니다. 계산된 주가이동평균 값이 맞는지 궁금하신 분들은 파이썬 IDLE나 Jupyter QtConsole을 사용해서 직접 계산해보기 바랍니다. 계속해서 16-02-26일에 대한 5일 주가이동평균은 16-02-22~16-02-26 기간의 수정 종가를 사용해서 계산하면 됩니다.

표 13.2 GS 종목 수정 종가 및 5일 주가이동평균

일자	수정 종가	5일 주가이동평균
16-02-19	50,600	-
16-02-22	50,400	-
16-02-23	52,800	-
16-02-24	53,500	-
16-02-25	53,900	(50,600+50,400+52,800+53,500+53,900)/5 = 52,240
16-02-26	53,300	(50,400+52,800+53,500+53,900+53,300)/5 = 52,780
16-02-29	53,000	(52,800+53,500+53,900+53,300+53,000)/5 = 53,300

일자	수정 종가	5일 주가이동평균
16-03-02	55,100	(53,500+53,900+53,300+53,000+55,100)/5 = 53,760
16-03-03	55,000	(53,900+53,300+53,000+55,100+55,000)/5 = 54,060
16-03-04	54,200	(53,300+53,000+55,100+55,000+54,200)/5 = 54,120

표 13.2를 통해 계산해본 것처럼 주가이동평균은 일별로 하나의 값으로 계산됩니다. 따라서 매일 계산된 주가이동평균을 하나의 선으로 연결해 나타내면 5일 주가이동평균선을 그릴 수 있습니다.

참고로 주가이동평균선은 시장의 전반적인 주가 흐름을 판단하고 향후 주가의 추이를 전망하는 데 자주 사용되는 주식시장의 대표적인 기술지표입니다. 특히 5일 20일 60일 120일 이동평균선이 서로 교차하게 되는데 이를 크로스(cross)라고 부르고 이 시점이 매우 중요한 매매 시점이 됩니다.

예를 들어, 그림 13.21은 네이버 증권에서 에이모션이라는 종목의 차트를 캡처한 것입니다. 보통 주식 관련 서적을 보면 거래량 분석과 이동평균선 기반의 매매 기법의 효용성을 보여주기 위해 그림 13.21과 같은 차트를 자주 사용합니다.

그림 13.21을 살펴보면 에이모션 종목은 2014-11-13일에 거래량이 큰 폭으로 늘어났습니다. 이 시점에 매수해서 2015-02-13일까지 종목을 계속 보유했다면 약 60% 정도의 이익을 얻었을 것입니다.

이번에는 주가이동평균선을 이용해 매매 시점을 분석해보겠습니다. 그림 13.21의 차트에서 2014-11-26은 20일 주가이동평균선이 60일 주가이동평균선을 뚫고 지나가는 크로스 지점입니다. 만약 이 근처에서 주식을 매수해서 2015-02-13일까지 종목을 계속 보유했다면 약 40%의 이익을 얻었을 것입니다.



그림 13.21 에이모션 일봉 차트(출처: 네이버 증권)

그러나 여기서 조심해야 할 점이 있습니다. 기존의 주식 관련 서적에서는 특정 차트의 예를 들어 마치 거래량 분석과 주가이동평균선 기반의 분석 기법이 항상 효과적인 것처럼 설명합니다. 그러나 어떤 매매 기법이 정말로 효과적인지 확인하려면 약 2,000여 개의 유가증권/코스닥 시장 종목 중 몇 개가 이런 패턴이 발생했고 이때 몇 개가 실제로 상승했는지를 가지고 매매 기법을 평가해야 합니다.

여러분은 시중에 있는 주식 관련 서적들이 내세우는 알고리즘을 무턱대고 믿기보다는 프로그래밍을 통해 직접 알고리즘을 검증해보고 수익이 날 확률이 얼마 정도인지 꼭 확인해보기 바랍니다.

그렇다면 검증되지도 않은 주가이동평균선을 왜 배워야 할까요? 사실 돈 잘 버는 알고리즘이란 것들도 가만히 살펴보면 우리가 알고 있는 알고리즘과 큰 원리는 비슷한데 약간씩 튜닝된 경우가 많습니다. 그래서 시장에서 지금까지 많이 사용돼 왔던 기술 지표 등을 공부하고 이를 이해하고 응용하는 것이 의미가 있습니다.

pandas를 이용한 주가이동평균 계산

주가이동평균을 계산하기 위해 먼저 DataReader를 통해 일자별 데이터를 DataFrame 객체로 저장합니다.

```
In [84]: import pandas as pd

In [85]: import pandas_datareader.data as web

In [86]: gs = web.DataReader("KRX:078930", "google", "2014-01-01", "2016-03-06")
```

앞의 예에서는 datetime 모듈을 이용해 조회 기간을 입력했는데 예제에서는 조회 기간의 시작일과 종료일을 직접 입력했습니다. 이 경우에는 datetime과 달리 날짜에 0을 넣거나 빼도 잘 인식합니다.

데이터가 잘 받아졌는지 확인하기 위해 tail 메서드를 호출해서 끝부분에 위치하는 5개의 데이터를 확인합니다.

```
In [87]: gs.tail()
Out[87]:
              Open     High      Low    Close  Volume
Date
2016-02-26  54300.0  54500.0  53200.0  53300.0  159181
2016-02-29  53000.0  53900.0  52800.0  53000.0  154363
2016-03-02  53900.0  55200.0  53500.0  55100.0  229860
2016-03-03  55300.0  55800.0  54300.0  55000.0  216838
2016-03-04  54700.0  54800.0  53200.0  54200.0  237737
```

수정 종가에 대해 5일 주가 이동평균을 계산하려면 다음과 같이 코드를 작성합니다.

```
In [88]: ma5 = gs['Close'].rolling(window=5).mean()
```

ma5 객체의 타입을 보면 pandas의 Series임을 확인할 수 있습니다. 앞서 설명한 것처럼 이동평균은 각 일자에 계산되는 값이므로 결괏값이 매우 많습니다. tail 메서드를 사용해 뒤에서부터 10개의 값만 확인해 봅시다.

```
In [91]: type(ma5)
Out[91]: pandas.core.series.Series

In [92]: ma5.tail(10)
Out[92]:
Date
2016-02-19    50510.0
```

```
2016-02-22    50600.0
2016-02-23    51060.0
2016-02-24    51640.0
2016-02-25    52240.0
2016-02-26    52780.0
2016-02-29    53300.0
2016-03-02    53760.0
2016-03-03    54060.0
2016-03-04    54120.0
Name: Close, dtype: float64
```

계산된 5일 주가이동평균 값을 좀 더 쉽게 관리하기 위해 DataFrame 객체의 새로운 칼럼으로 추가해 봅시다. DataFrame에 칼럼을 추가하려면 insert 메서드를 사용하면 됩니다. insert 메서드의 첫 번째 인자로는 칼럼이 추가될 위치를, 두 번째 인자로는 칼럼의 이름을, 세 번째 인자로는 DataFrame 객체에 추가할 데이터를 전달하면 됩니다.

```
In [164]: ma5 =  gs['Close'].rolling(window=5).mean()

In [165]: gs.insert(len(gs.columns), "MA5", ma5)

In [166]: gs.tail(5)
Out[166]:
             Open     High      Low    Close  Volume      MA5
Date
2016-02-26  54300.0  54500.0  53200.0  53300.0  159181  52780.0
2016-02-29  53000.0  53900.0  52800.0  53000.0  154363  53300.0
2016-03-02  53900.0  55200.0  53500.0  55100.0  229860  53760.0
2016-03-03  55300.0  55800.0  54300.0  55000.0  216838  54060.0
2016-03-04  54700.0  54800.0  53200.0  54200.0  237737  54120.0
```

rolling_mean 함수의 두 번째 인자로 20, 60, 120의 값을 전달하면 20, 60, 120일 이동평균 값 역시 쉽게 계산할 수 있습니다. 20, 60, 120일 이동평균 값 역시 DataFrame 객체의 칼럼으로 추가합니다.

```
In [167]: ma20 = gs['Close'].rolling(window=20).mean()

In [168]: ma60 = gs['Close'].rolling(window=60).mean()

In [169]: ma120 = gs['Close'].rolling(window=120).mean()

In [170]: gs.insert(len(gs.columns), "MA20", ma20)
```

```
In [171]: gs.insert(len(gs.columns), "MA60", ma60)

In [172]: gs.insert(len(gs.columns), "MA120", ma120)
```

주가이동평균선 그리기

이번에는 앞서 계산한 5, 20, 60, 120일 이동평균 값을 사용해 주가이동평균선을 그려보겠습니다. 기본적으로 차트를 그리는 방법은 13.3.2절에서 배운 내용과 같습니다.

먼저 matplotlib 패키지의 pyplot 모듈을 임포트합니다.

```
In [170]: import matplotlib.pyplot as plt
```

plot 함수를 사용해 그래프를 그리면 됩니다. 이때 각 그래프가 서로 구분될 수 있도록 범례를 추가해 보겠습니다. 범례를 만드는 가장 쉬운 방법은 그래프를 그릴 때 그래프에 label 인자를 사용하는 것입니다.

먼저 종가 그래프를 그리기 위해 label 인자에 'Close'를 전달합니다.

```
In [182]: plt.plot(gs.index, gs['Close'], label="Close")
Out[182]: [<matplotlib.lines.Line2D at 0xc82cff0>]
```

위와 마찬가지로 5, 20, 60, 120일 이동평균선 그래프도 그려봅시다.

```
In [188]: plt.plot(gs.index, gs['MA5'], label="MA5")
Out[188]: [<matplotlib.lines.Line2D at 0xc8b0550>]

In [189]: plt.plot(gs.index, gs['MA20'], label="MA20")
Out[189]: [<matplotlib.lines.Line2D at 0xc8b09d0>]

In [190]: plt.plot(gs.index, gs['MA60'], label="MA60")
Out[190]: [<matplotlib.lines.Line2D at 0xc8b0ef0>]

In [191]: plt.plot(gs.index, gs['MA120'], label="MA120")

Out[191]: [<matplotlib.lines.Line2D at 0xc8b83f0>]
```

범례를 표시하기 위해 legend 함수를 호출합니다. 이때 loc 인자를 통해 범례 표시 위치를 지정할 수 있습니다. 범례가 적절한 위치에 자동으로 출력되게 하려면 loc='best' 옵션을 사용하면 됩니다. 또한 그래프의 값을 좀 더 편리하게 확인하기 위한 격자(grid)를 표시하려면 grid 함수를 호출하면 됩니다.

```
In [192]: plt.legend(loc='best')
Out[192]: <matplotlib.legend.Legend at 0xc8b04f0>
In [202]: plt.grid()
```

오타가 없다면 그림 13.22와 같은 모습의 그래프가 출력되는 것을 확인할 수 있습니다.

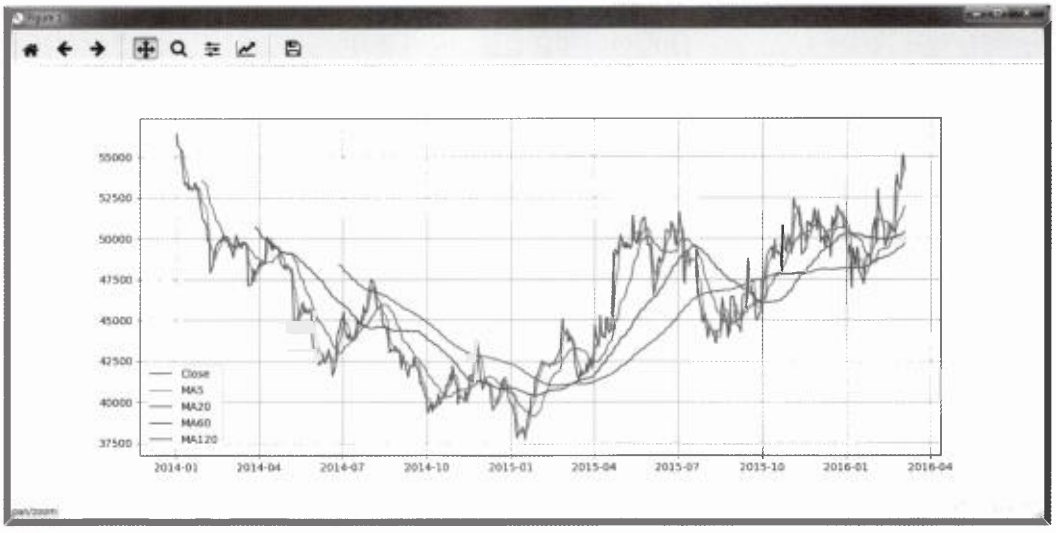


그림 13.22 5, 20, 60, 120일 주가이동평균선 그래프

예제 13.3은 수정 종가 및 주가이동평균선(5, 20, 60, 120일)을 그래프로 출력하는 전체 코드입니다.

예제 13.3 수정 종가, 주가이동평균선 그래프 그리기(소스코드: book/ch13/15.py)

```
import pandas as pd
import pandas_datareader.data as web
import matplotlib.pyplot as plt

# Get GS Data from Google
gs = web.DataReader("KRX:078930", "google", "2014-01-01", "2016-03-06")

# Moving average
ma5 = gs['Close'].rolling(window=5).mean()
ma20 = gs['Close'].rolling(window=20).mean()
ma60 = gs['Close'].rolling(window=60).mean()
ma120 = gs['Close'].rolling(window=120).mean()
```

```
# Insert columns
gs.insert(len(gs.columns), "MA5", ma5)
gs.insert(len(gs.columns), "MA20", ma20)
gs.insert(len(gs.columns), "MA60", ma60)
gs.insert(len(gs.columns), "MA120", ma120)

# Plot
plt.plot(gs.index, gs['Close'], label='Close')
plt.plot(gs.index, gs['MA5'], label='MA5')
plt.plot(gs.index, gs['MA20'], label='MA20')
plt.plot(gs.index, gs['MA60'], label='MA60')
plt.plot(gs.index, gs['MA120'], label='MA120')

plt.legend(loc="best")
plt.grid()
plt.show()
```

14

pandas와 Zipline을 이용한 백테스팅

14장에서는 pandas와 알고리즘 트레이딩 라이브러리인 Zipline을 이용해 백테스팅(backtesting) 환경을 꾸미는 방법을 배워보겠습니다. 알고리즘 트레이딩이 기존의 전통적인 투자 관리 기법과 다른 점이 바로 백테스팅을 통해 알고리즘을 검증한다는 점입니다.

여러분만의 투자 알고리즘을 개발했다면 이를 이용한 실전 투자를 하기 앞서 가장 먼저 해야 할 일이 바로 백테스팅입니다. 여기서 백테스팅이란 개발된 알고리즘을 과거 데이터를 사용해 검증해보는 것을 의미합니다. 과거 데이터를 사용하기 때문에 back이라는 영어 단어와 알고리즘을 테스트하기 때문에 testing이라는 영어 단어가 사용됩니다.

물론 개발된 알고리즘을 테스트하는 가장 좋은 방법은 미래의 데이터를 사용하는 겁니다. 그러나 현시점보다 미래의 데이터를 얻을 방법은 없으며 알고리즘을 검증하지 않고 막연히 잘 될 것 같다는 직감을 갖고 실전 투자에 임하는 것은 매우 위험합니다.

여러분이 개발한 알고리즘뿐 아니라 시장에서 널리 통용되고 있는 투자 기법에 대해 백테스팅을 해보는 것도 상당히 유용합니다. 기존 알고리즘에 대한 백테스팅을 수행해봄으로써 기존 알고리즘을 더욱

정확히 이해할 수 있게 됩니다. 이를 통해 기존의 알고리즘에 대한 개선점 등을 찾을 수 있고 이를 여러분만의 알고리즘 트레이딩 시스템에 적용할 수도 있을 것입니다.

01 Zipline 시작하기

Zipline은 알고리즘 트레이딩 라이브러리로서 백테스팅 기능을 제공합니다. Zipline은 Quantopian사에 의해 깃허브(https://github.com/quantopian/zipline)를 통해 오픈소스로 개발되고 있습니다. 여기서 오픈소스란 소스코드가 공개돼 있고 누구나 개발에 참여할 수 있는 개발 환경을 의미합니다.

Zipline은 무료로 사용할 수 있으며, 오픈소스이고 무엇보다 사용하기 편리합니다. 또한 pandas, Matplotlib, Scipy 같은 다양한 파이썬 라이브러리와 쉽게 연동해서 사용할 수 있으므로 매우 유용한 도구입니다.

Zipline 설치

먼저 Jupyter QtConsole을 실행한 후 그림 14.1과 같이 Zipline 모듈을 임포트합니다. 모듈이 설치돼 있다면 에러가 발생하지 않겠지만 설치돼 있지 않다면 그림 14.1과 같이 에러가 발생합니다.

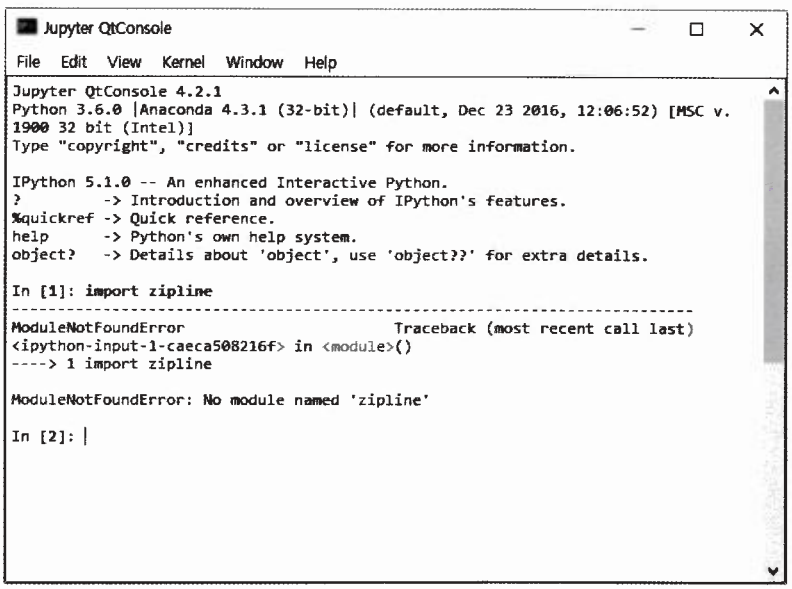


그림 14.1 Zipline 모듈 설치 여부 확인

Zipline 모듈이 설치돼 있지 않다면 먼저 모듈 설치부터 해야겠지요? 여러분은 아나콘다 배포판을 사용하기 때문에 지금까지는 conda 명령어를 통해 패키지나 모듈을 쉽게 설치할 수 있었습니다. 그러나 일부 패키지나 모듈은 특정 버전의 파이썬만 지원하거나 conda 명령어를 통한 패키지 관리를 지원하지 않는 경우가 있습니다.

이번에 설치해볼 Zipline도 이 경우에 속합니다. Zipline의 깃허브 저장소(https://github.com/quantopian/zipline)에 접속해보면 그림 14.2와 같이 현재 파이썬 2.7과 파이썬 3.4만 지원함을 알 수 있습니다. 따라서 파이썬 3.6용 아나콘다 배포판을 설치한 분들은 Zipline을 사용할 수 없습니다.

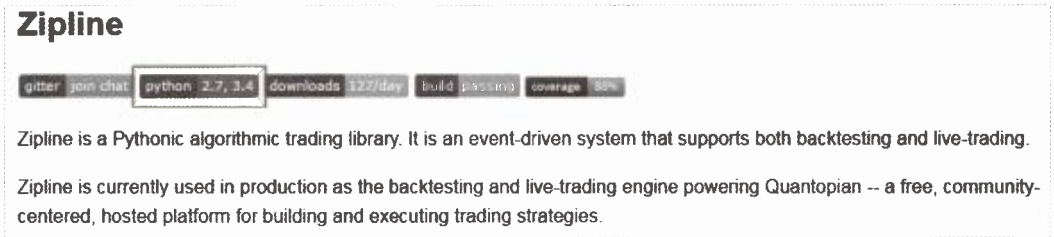


그림 14.2 Zipline 파이썬 버전 확인

귀찮은 일이긴 하지만 Zipline이 파이썬 2.7 또는 파이썬 3.4만 지원하기 때문에 Zipline을 위해서는 파이썬 3.4를 사용해야 합니다. 그러나 이미 파이썬 3.6 버전의 배포판을 설치해서 잘 사용해오고 있는데 Zipline 패키지 하나 때문에 아나콘다 배포판을 다른 버전으로 재설치하자니 여간 불편한 것이 아닙니다. 다행히도 아나콘다를 이용하면 현재 설치된 파이썬 개발 환경은 그대로 유지한 채로 별도의 파이썬 개발 환경을 만들 수 있습니다.

먼저 그림 14.3과 같이 아나콘다 프롬프트를 실행합니다. 그리고 PC에 설치된 conda의 버전을 확인합니다.

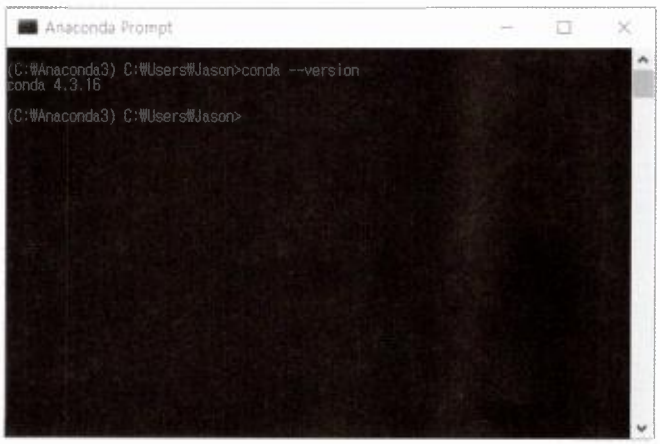


그림 14.3 아나콘다 프롬프트

파이썬 3.4 개발 환경을 추가로 생성하기 전에 conda부터 업데이트하겠습니다. 그림 14.4와 같이 'conda update conda' 명령어를 아나콘다 프롬프트에 입력해 conda를 업데이트합니다.

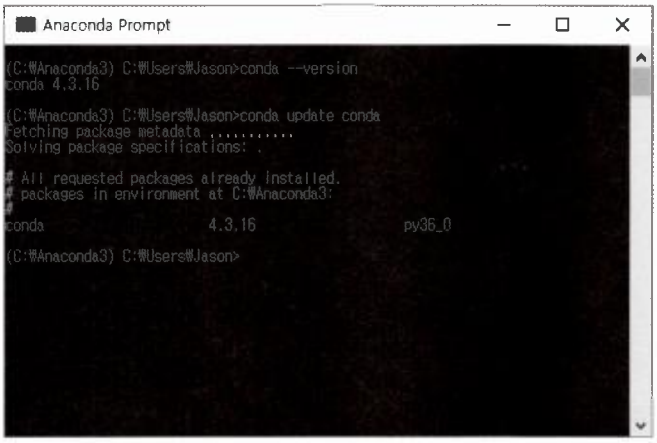


그림 14.4 Conda 업데이트

conda를 업데이트했다면 이번에는 파이썬 3.4용 개발 환경을 만들어 보겠습니다. 새로운 개발 환경을 만들기 위해서는 conda create 명령을 사용합니다. -n 옵션을 통해 개발 환경의 이름을 지정할 수 있으며, 사용할 파이썬 버전과 초기에 설치될 패키지를 입력하면 됩니다. 설치할 패키지 이름 대신 anaconda라고 입력하면 아나콘다를 설치할 때 설치되는 기본 패키지가 한 번에 모두 설치됩니다.

conda create 명령 역시 그림 14.5와 같이 아나콘다 프롬프트 상에서 수행하면 됩니다. 해당 명령어가 수행되면 설치될 패키지 목록이 출력되고 설치를 진행할 것인지 물어봅니다. 이때 'y'를 입력해 패키지 다운로드 및 설치를 진행합니다. 패키지를 다운로드하고 설치하기 때문에 설치 과정에 약간의 시간이 소요됩니다.



그림 14.5 conda create를 이용한 개발 환경 생성

conda create 명령어를 통해 파이썬 개발 환경을 추가로 생성하면 아나콘다의 설치 디렉터리인 C:\Anaconda3\envs에 -n 옵션을 통해 지정한 이름(py34)으로 디렉터리가 생성되고 해당 디렉터리 안에 그림 14.6과 같이 지정한 버전의 파이썬(파이썬 3.4)과 패키지가 설치됩니다.

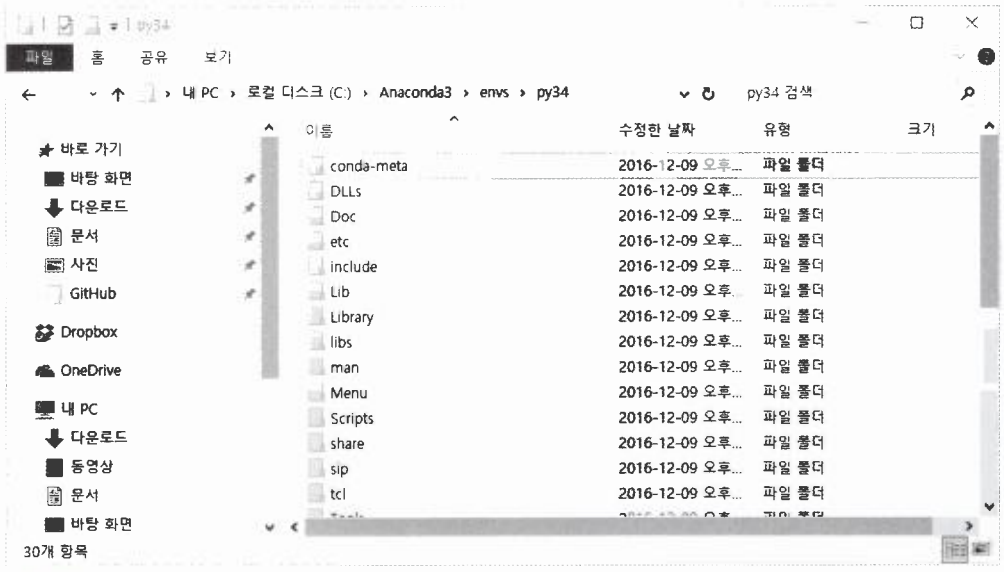


그림 14.6 conda create를 통해 생성된 개발 환경

conda create를 통해 새로운 개발 환경을 생성했다면 이번에는 Visual C++ 2010이라는 개발 툴을 설치해보겠습니다. Visual C++ 2010은 마이크로소프트에서 개발한 IDE인데 C/C++ 코드에 대한 컴파일러인 Visual C++ 10.0을 포함하고 있습니다. Visual C++ 컴파일러는 C/C++ 코드로 구현된 파이썬 패키지를 설치할 때 필요합니다.

Visual C++ 컴파일러를 설치할 때 주의할 점은 파이썬 버전에 따라 특정 버전의 컴파일러를 설치해야 한다는 점입니다. 파이썬 2.6부터 3.2까지는 Visual C++ 9.0 컴파일러, 파이썬 3.3과 3.4는 Visual C++ 10.0 컴파일러, 파이썬 3.5는 Visual C++ 14.0 컴파일러를 설치하면 됩니다.

다음 URL을 통해 vc_web.exe 파일을 내려받은 후 더블클릭해서 설치를 시작합니다.

- http://microsoft-visual-cpp-express.soft32.com/

그림 14.7에서 체크박스에 체크하지 않고 [Next]를 눌러 설치를 시작합니다.

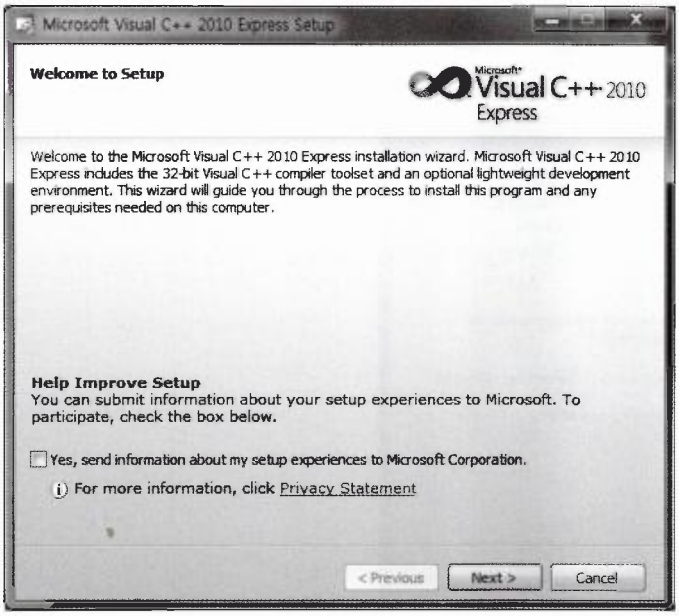


그림 14.7 Visual C++ 2010 Express 설치 1

그림 14.8과 같이 라이선스에 동의한 후 [Next]를 누릅니다.

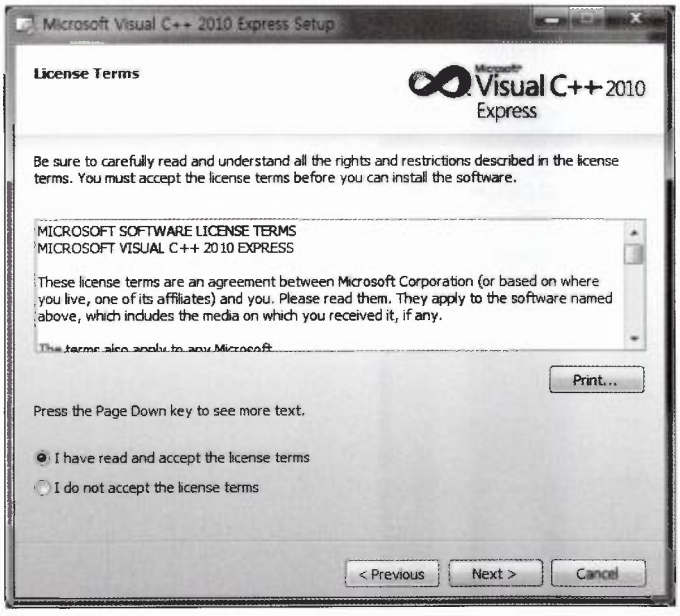


그림 14.8 Visual C++ 2010 Express 설치 2

그림 14.9와 같이 두 개의 체크박스에 체크 표시를 해제한 후 [Next]를 눌러 다음 설치 과정을 진행합니다.

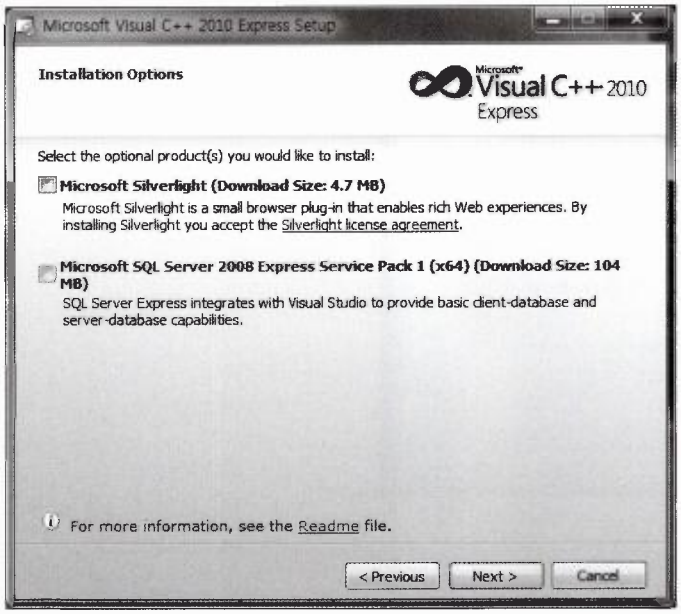


그림 14.9 Visual C++ 2010 Express 설치 3

그림 14.10에서 [Install] 버튼을 누르면 설치가 시작됩니다.

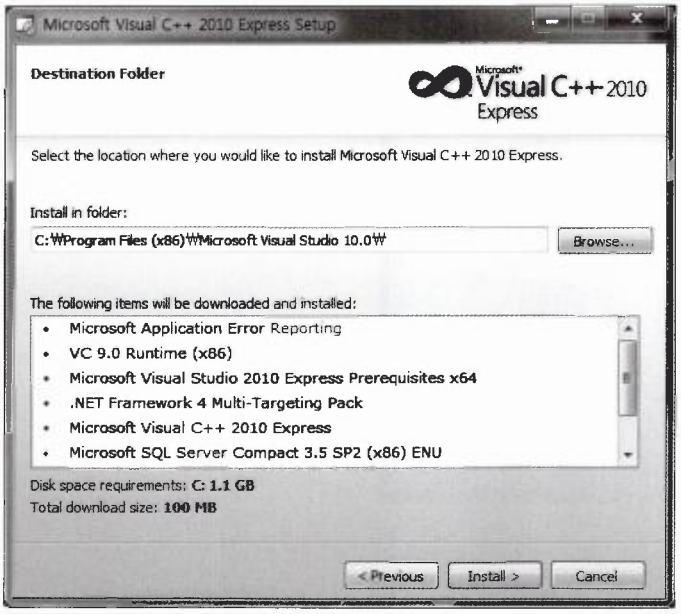


그림 14.10 Visual C++ 2010 Express 설치 4

Visual C++ 2010 Express 개발 툴의 설치가 완료되면 앞에서 만든 py34 개발 환경에 zipline 패키지를 설치하면 됩니다. 윈도우의 [시작] → [모든 프로그램] 메뉴를 보면 아나콘다 디렉터리에 기존 개발 환경과 추가로 생성한 개발 환경인 py34가 모두 존재하는 것을 확인할 수 있습니다. 그림 14.11과 같이 [Anaconda Prompt (py34)] 메뉴를 클릭합니다.

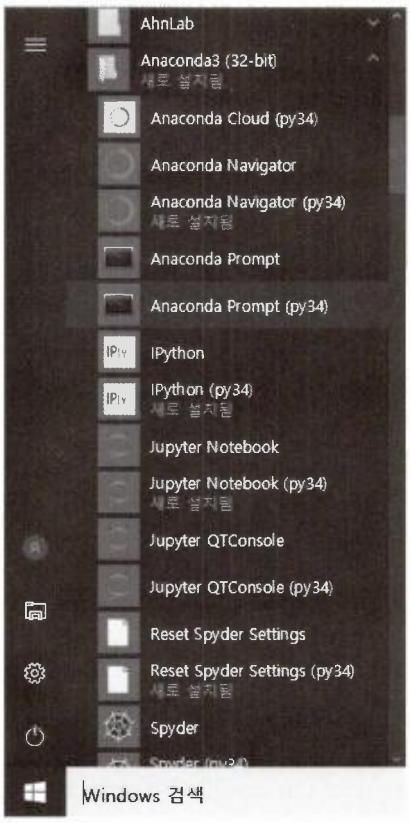


그림 14.11 Anaconda Prompt (py34) 메뉴 선택

Anaconda Prompt (py34)가 실행되면 그림 14.22와 같이 프롬프트에 'pip install zipline' 명령을 입력해 Zipline 패키지 설치를 시작합니다.

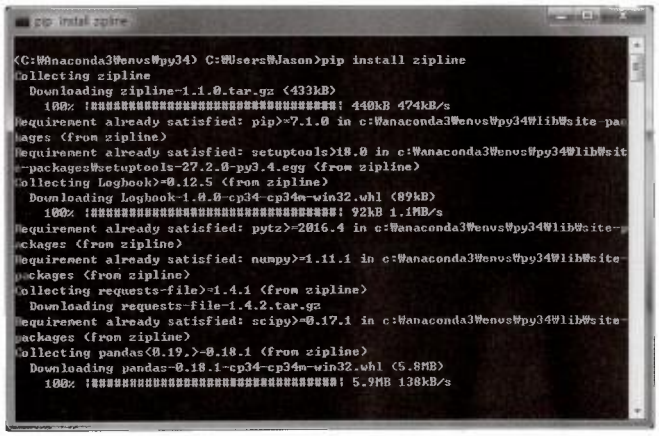


그림 14.12 pip를 이용한 Zipline 설치

Zipline 패키지가 정상적으로 설치됐는지 확인하기 위해 Python 3.4 개발환경에서 해당 모듈을 임포트해 봅시다. 윈도우의 [시작] → [모든 프로그램] 메뉴에서 [Jupyter QtConsole (py34)]를 선택합니다. 실행된 Jupyter QtConsole을 보면 그림 14.13과 같이 파이썬 3.4.5 버전임을 확인할 수 있습니다.

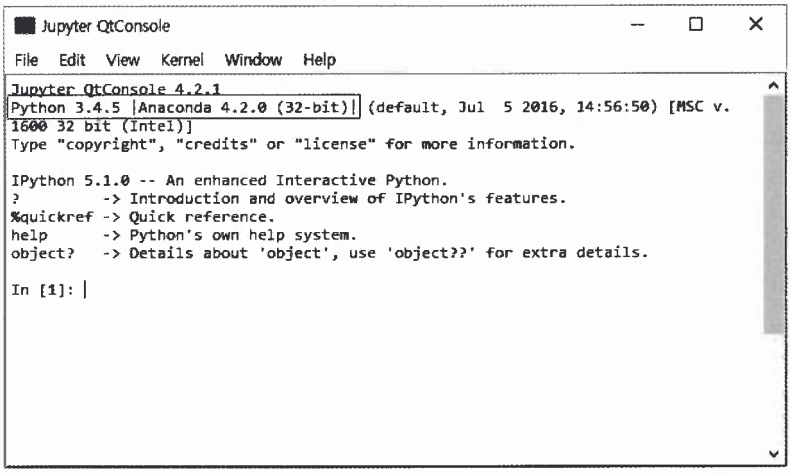


그림 14.13 파이썬 3.4용 Jupyter QtConsole

Jupyter QtConsole에서 import zipline을 실행하면 그림 14.14와 같이 정상적으로 모듈이 임포트됨을 확인할 수 있습니다.

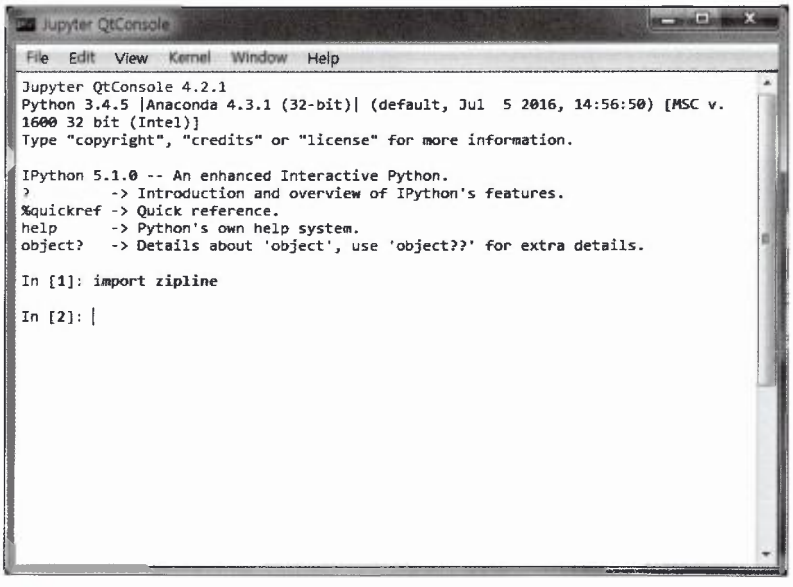


그림 14.14 Zipline 모듈 임포트

Zipline 기초

이번 절에서는 매우 간단한 알고리즘을 통해 Zipline을 이용한 백테스팅 기초를 배우겠습니다. 이번 절에서 살펴볼 알고리즘은 Zipline 패키지에서 제공하는 예제 중 하나로, 매일 일정 수량의 애플 주식을 매수하는 알고리즘입니다.

먼저 Jypyter QtConsole (py34)를 실행합니다. Jupyter QtConsole이 실행되면 그림 14.13과 같이 파이썬 버전이 3.4.x 버전인지 확인합니다.

백테스팅을 수행하려면 알고리즘과 과거 데이터가 있어야 합니다. 먼저 애플 종목의 과거 데이터를 pandas_datareader 패키지를 사용해 가져와 봅시다. 다음 코드를 Jupyter QtConsole에 입력합니다.

```
In [1]: import pandas_datareader.data as web

In [2]: import datetime

In [3]: import matplotlib.pyplot as plt

In [4]: start = datetime.datetime(2010, 1, 1)

In [5]: end = datetime.datetime(2016, 3, 19)

In [6]: data = web.DataReader("AAPL", "google", start, end)
```

애플 종목에 대한 데이터는 "AAPL"이라는 이름을 통해 구글 파이낸스로부터 쉽게 얻어올 수 있습니다. 데이터를 잘 얻었다면 종가(Close)를 그래프로 그려 봅시다.

```
In [23]: plt.plot(data.index, data['Close'])
Out[23]: [<matplotlib.lines.Line2D at 0x6b982b0>]

In [24]: plt.show()
```

그림 14.15와 같이 각 일자에 대한 수정 종가가 그래프로 잘 그려진 것을 확인할 수 있습니다.

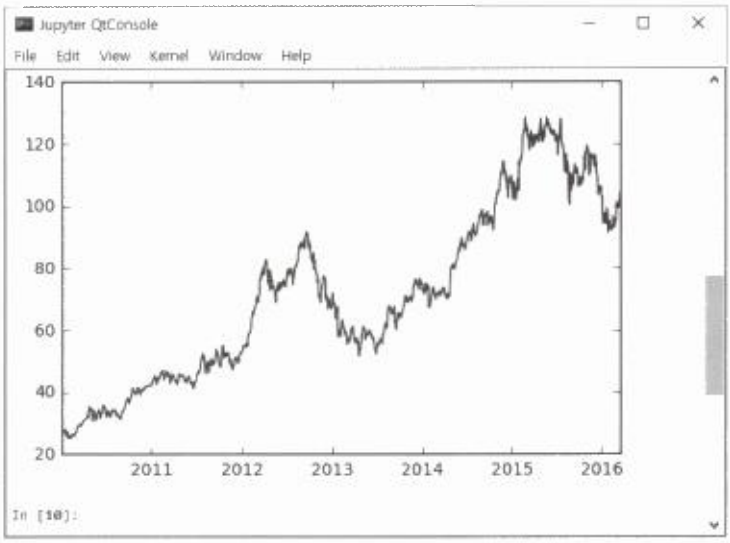


그림 14.15 애플 수정 종가를 나타내는 그래프

Jupyter QtConsole에서 matplotlib 패키지를 이용해 그래프를 그리면 그림 14.15와 같이 Jupyter QtConsole에 출력됩니다. 그래프가 별도의 창으로 출력되도록 변경하려면 다음과 같은 명령어를 입력하면 됩니다. 또는 %matplotlib qt라고 입력해도 됩니다.

```
In [10]: %matplotlib
```

다시 그래프를 그려보면 이번에는 그림 14.16과 같이 별도의 창으로 그래프가 출력되는 것을 확인할 수 있습니다.

```
In [11]: plt.plot(data.index, data["Close"])
Out[11]: [<matplotlib.lines.Line2D at 0x13f2ac50>]
```

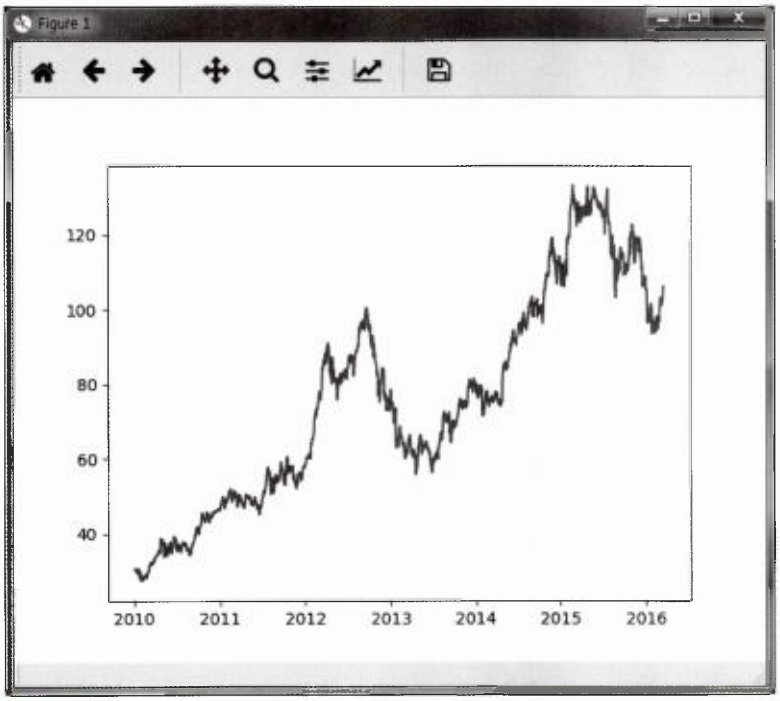


그림 14.16 애플 수정 종가를 나타내는 그래프(새로운 창으로 출력)

백테스팅에 사용할 데이터가 준비되면 알고리즘을 구현하면 됩니다. 앞서 설명한 것처럼 이번 절에서는 단순히 매일 정해진 수량의 주식을 매수하는 알고리즘을 구현해 보겠습니다.

Zipline을 이용해 알고리즘을 백테스팅하려면 먼저 initialize와 handle_data라는 함수를 구현해야 합니다. Zipline 백테스팅 시뮬레이터는 시뮬레이션을 수행하기 전에 항상 initialize 함수를 호출합니다. 따라서 여러분은 시뮬레이션에 사용할 초기 투자 금액이나 거래 수수료와 같은 값들을 initialize 함수에서 설정할 수 있습니다.

실제 거래 알고리즘은 handle_data라는 이름의 함수에서 구현하면 됩니다. Zipline 시뮬레이터는 시뮬레이션을 수행하는 동안 거래일마다 handle_data를 호출합니다. 그림 14.17은 Zipline 백테스팅의 주요 시뮬레이션 과정을 그림으로 나타낸 것입니다.

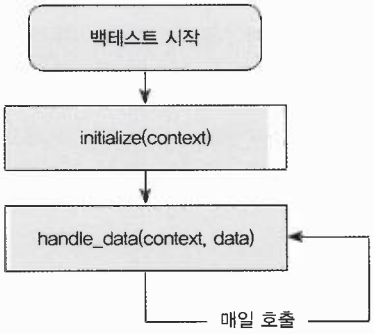


그림 14.17 Zipline 백테스팅 시뮬레이션

먼저 initialize 함수를 구현해 봅시다. 이번 절에서는 Zipline 시뮬레이션 동안 초기 투자 금액이나 거래 수수료를 기본값으로 그대로 사용할 것이므로 initialize 함수 내에서 따로 초기화 코드를 작성할 필요는 없습니다.

```
In [16]: def initialize(context):
    ...:     pass
    ...:

In [17]:
```

handle_data 함수를 구현하기 전에 zipline.api 모듈로부터 사용할 함수를 임포트합니다. order는 Zipline 백테스팅 시뮬레이션에서 주문을 실행하는 함수이며, symbol은 참조할 데이터에 대한 심볼을 등록하는 함수입니다. order 함수를 사용해 심볼이 'AAPL'인 주식 한 주를 매수하도록 코드를 구현합니다.

```
In [18]: from zipline.api import order, symbol

In [19]: def handle_data(context, data):
    ...:     order(symbol('AAPL'), 1)
    ...:

In [20]:
```

백테스팅을 위한 알고리즘 구현을 완료했다면 데이터와 알고리즘을 사용해 시뮬레이션을 진행하면 됩니다. 백테스팅에는 주로 수정 종가가 사용되는데 현재 data라는 DataFrame 객체에는 Open, High, Low, Close, Volume 칼럼이 있습니다.

```
In [21]: data.head()
Out[21]:
             Open   High    Low  Close     Volume
Date
2010-01-04  30.49  30.64  30.34  30.57  123432050
2010-01-05  30.66  30.80  30.46  30.63  150476004
2010-01-06  30.63  30.75  30.11  30.14  138039594
2010-01-07  30.25  30.29  29.86  30.08  119282324
2010-01-08  30.04  30.29  29.87  30.28  111969081

In [22]:
```

기존의 DataFrame 객체에서 'Close' 칼럼만 선택해 새로운 DataFrame 객체를 생성합니다. head 메서드를 통해 새로 생성된 DataFrame 객체를 확인하면 'Adj Close' 칼럼만 존재하는 것을 확인할 수 있습니다.

```
In [22]: data = data[['Close']]

In [23]: data.head()
Out[23]:
            Close
Date
2010-01-04  30.57
2010-01-05  30.63
2010-01-06  30.14
2010-01-07  30.08
2010-01-08  30.28

In [24]:
```

앞서 handle_data 함수를 구현할 때 symbol("AAPL")을 사용했는데 이것은 DataFrame 객체에서 선택할 칼럼 인덱스를 의미합니다. 따라서 DataFrame 객체의 칼럼 인덱스 값을 'Close'에서 'AAPL'로 변경해야 합니다. 다음과 같이 columns 속성을 사용해 칼럼의 인덱스 값을 'AAPL'로 변경합니다.

```
In [24]: data.columns = ["AAPL"]

In [25]: data.head()
Out[25]:
             AAPL
Date
2010-01-04  30.57
2010-01-05  30.63
2010-01-06  30.14
2010-01-07  30.08
2010-01-08  30.28

In [26]:
```

마지막으로 DataFrame 객체의 인덱스 값을 협정 세계시(UTC)로 변경합니다.

```
In [26]: data = data.tz_localize("UTC")

In [27]: data.head()
Out[27]:
```

```
                            AAPL
Date
2010-01-04 00:00:00+00:00  30.57
2010-01-05 00:00:00+00:00  30.63
2010-01-06 00:00:00+00:00  30.14
2010-01-07 00:00:00+00:00  30.08
2010-01-08 00:00:00+00:00  30.28

In [28]:
```

이로써 백테스팅을 위한 모든 준비를 마쳤습니다. 지금부터는 백테스팅을 실제로 수행해보겠습니다. Zipline에서 백테스팅에는 TradingAlgorithm이라는 클래스의 인스턴스를 사용합니다. 먼저 해당 클래스를 사용하기 위해 임포트합니다.

```
In [28]: from zipline.algorithm import TradingAlgorithm
```

TradingAlgorithm 클래스의 인스턴스를 생성할 때 앞서 구현한 초기화 함수, 알고리즘이 구현된 함수를 인자로 전달하면 됩니다.

```
In [31]: algo = TradingAlgorithm(initialize=initialize, handle_data=handle_data)
```

TradingAlgorithm 객체가 정상적으로 생성되면 run 메서드를 호출해 시뮬레이션을 수행할 수 있습니다. 이때 run 메서드의 인자로 시뮬레이션에 사용될 데이터인 DataFrame 객체를 전달하면 됩니다. run 메서드가 수행된 후 시뮬레이션의 결괏값은 result에 저장됩니다.

```
In [32]: result = algo.run(data)
```

시뮬레이션이 정상적으로 수행됐다면 result의 값을 확인해봅니다. 참고로 result는 pandas의 DataFrame 객체로서 그림 14.18과 같이 head 메서드를 통해 간단히 값을 확인해 봅시다.

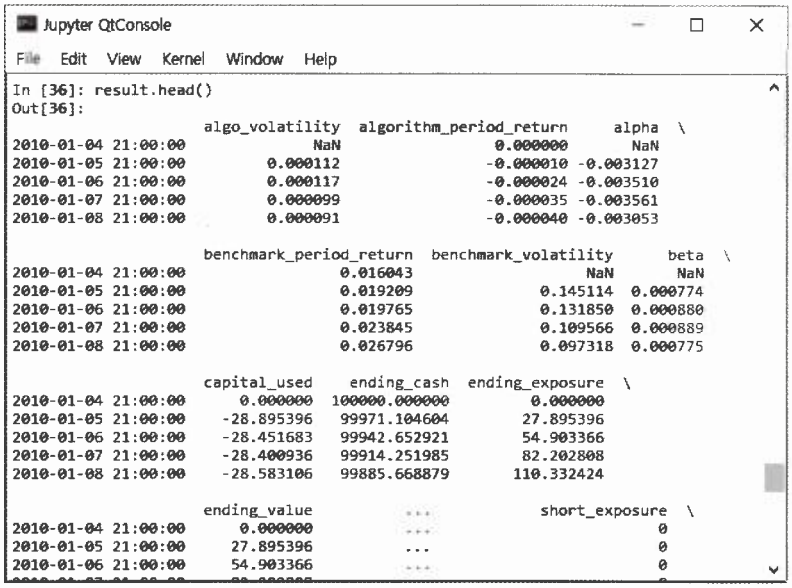


그림 14.18 일별 매수 알고리즘을 시뮬레이션한 결과

result에는 일자별로 다양한 데이터가 저장돼 있습니다. 그중 'starting_cash'는 각 거래일의 개장 전의 보유 현금이고 'ending_cash'는 시장 종료 후 보유 현금입니다. 'ending_value'는 각 거래일의 시장 종료 후의 보유 주식에 대한 평가 금액입니다.

'starting_cash', 'ending_cash', 'ending_value'는 DataFrame 객체의 칼럼 인덱스이므로 이를 이용해 시뮬레이션 초기 시점의 값을 확인해 봅시다. 거래는 2010-01-04일부터 시작됐는데 해당 일자에는 주식 1주의 매수 주문이 체결되지 않았기 때문에 'starting_cash'와 'ending_cash'가 시뮬레이션 초기 금액인 100,000달러입니다. 주문이 체결되지 않았기 때문에 평가 금액인 'ending_value'는 0임을 확인할 수 있습니다.

```
In [38]: result[['starting_cash', 'ending_cash', 'ending_value']].head()
Out[38]:
                     starting_cash    ending_cash  ending_value
2010-01-04 21:00:00  100000.000000  100000.000000      0.000000
2010-01-05 21:00:00  100000.000000   99971.104604     27.895396
2010-01-06 21:00:00   99971.104604   99942.652921     54.903366
2010-01-07 21:00:00   99942.652921   99914.251985     82.202808
2010-01-08 21:00:00   99914.251985   99885.668879    110.332424

In [39]:
```

2010-01-05에는 매수 1주가 체결됐고 이에 따라 'ending_cash'가 매수 금액 및 수수료(0.03 달러)만큼 빠진 99,971.104604달러임을 확인할 수 있습니다. 참고로 이날의 평가 금액은 2010-01-05일의 종가인 27.895396달러와 같음을 확인할 수 있습니다.

여러분이 투자하는 동안 관심 있는 것은 아마도 보유 현금과 주식 평가 금액을 합산한 금액일 것입니다. 특정 시점에서 이 금액이 초기 투자 금액보다 높다면 여러분은 주식 투자를 통해 수익을 낸 것이 됩니다. 이 값은 앞서 살펴본 'ending_cash'와 'ending_value'를 합산해서 계산할 수도 있지만 'portfolio_value' 칼럼을 통해 바로 확인할 수 있습니다.

```
In [40]: result.portfolio_value
```

시뮬레이션 기간이 길기 때문에 수치로 확인하기보다 그래프로 그려보겠습니다.

```
In [41]: plt.plot(result.index, result.portfolio_value)
Out[41]: [<matplotlib.lines.Line2D at 0x96cf870>]

In [42]: plt.show()

In [43]:
```

그림 14.19를 살펴보면 거래일마다 1주씩 매수하는 알고리즘으로는 2010년부터 2011년 중반까지는 큰 수익을 내지 못했습니다. 그러나 그 후에는 가파른 상승세를 보이는 것을 확인할 수 있습니다. 물론 이러한 수익은 알고리즘이 좋아서가 아니라 해당 기간 동안 애플 주식의 주가가 상승했기 때문입니다.



그림 14.19 일일 매수 알고리즘을 시뮬레이션한 결과(애플 주식)

지금까지 작성한 소스코드는 예제 14.1과 같습니다.

예제 14.1 일일 매수 알고리즘에 대한 백테스팅(소스코드: book/ch14/02.py)

```
import pandas_datareader.data as web
import datetime
import matplotlib.pyplot as plt
from zipline.api import order, symbol
from zipline.algorithm import TradingAlgorithm

# data
start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2016, 3, 19)
data = web.DataReader("AAPL", "google", start, end)

data = data[['Close']]
data.columns = ['AAPL']
data = data.tz_localize('UTC')
```

```
def initialize(context):
    pass

def handle_data(context, data):
    order(symbol('AAPL'), 1)

algo = TradingAlgorithm(initialize=initialize, handle_data=handle_data)
result = algo.run(data)

plt.plot(result.index, result.portfolio_value)
plt.show()
```

PyCharm에서 예제 14.1의 코드를 실행하려면 PyCharm 프로젝트에서 사용하는 파이썬 인터프리터의 버전을 3.4.5로 변경해야 합니다. PyCharm에서 프로젝트를 생성한 후 [File] → [Settings] 메뉴를 차례로 선택합니다.

그림 14.20에서 톱니바퀴 모양의 버튼을 클릭한 후 [Add Local] 메뉴를 선택합니다.

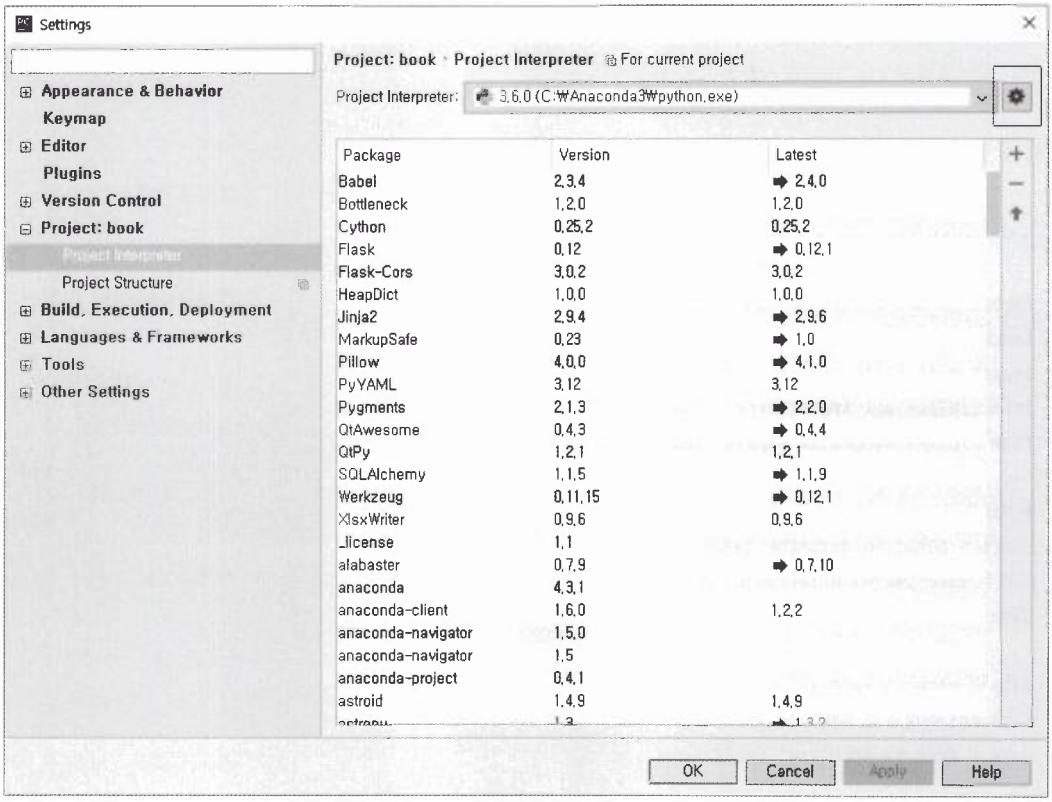


그림 14.20 PyCharm 프로젝트 인터프리터 설정

그림 14.21과 같이 Select Python Interpreter 창에서 C:\Anacona3\envs\py34 디렉터리의 python.exe를 선택한 후 [OK] 버튼을 클릭합니다.

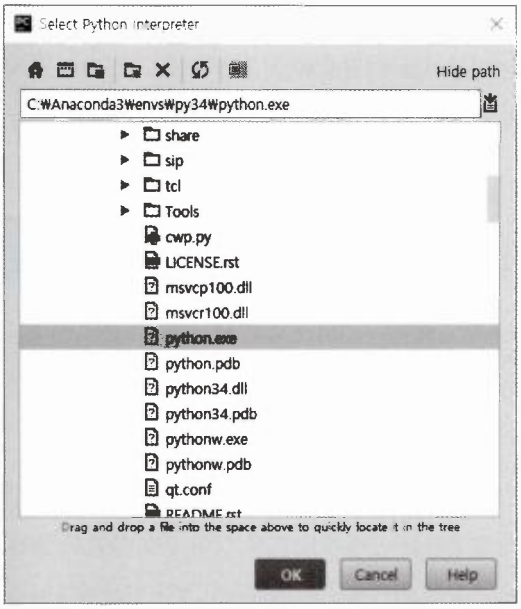


그림 14.21 파이썬 인터프리터 선택(PyCharm)

그림 14.22와 같이 현재 프로젝트에 대한 인터프리터 설정을 파이썬 3.6.0에서 파이썬 3.4.5로 변경한 후 [OK] 버튼을 클릭합니다.

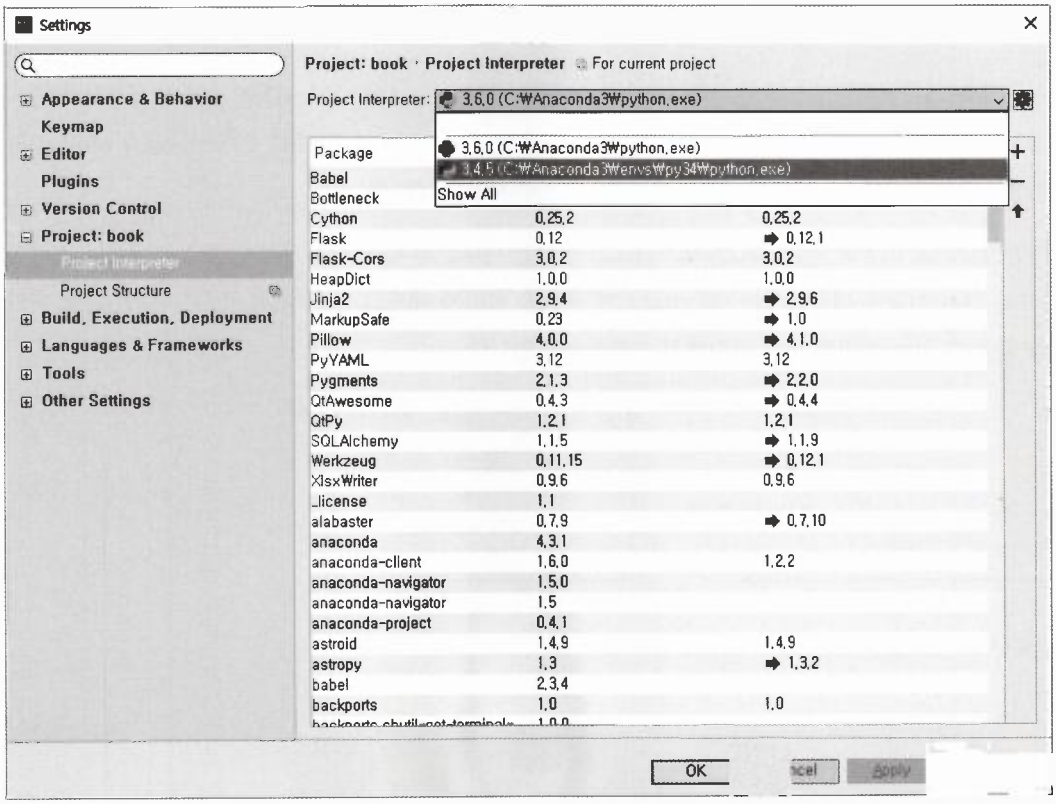


그림 14.22 프로젝트 인터프리터 변경(PyCharm)

PyCharm에서는 프로젝트 단위로 파이썬 인터프리터 버전을 설정할 수 있습니다. Zipline은 현재 파이썬 3.4.x 버전에서만 실행되므로 Zipline을 사용하는 프로젝트에서는 파이썬 인터프리터를 3.4.x 버전으로 변경해서 코드를 실행해야 합니다.

02 이동평균선 전략 백테스트

이번 절에서는 시장에서 주식 매수 시점과 매도 시점을 알아내는 전략으로 많이 사용되는 골든크로스(golden cross)와 데드크로스(dead cross)를 백테스팅해보겠습니다. 참고로 골든크로스와 데드크로스는 주가 이동평균선을 이용하는 대표적인 기술적 분석 지표입니다.

그림 14.23은 골든크로스와 데드크로스를 나타냅니다. 골든크로스는 단기 이동평균선이 중 · 장기 이동평균선을 아래에서 위로 상향 돌파하면서 생기는 교차점입니다. 이동평균선이 일정 기간의 주가의 평균값을 의미하므로 골든크로스가 발생했다는 것은 최근 주가의 흐름이 강세로 접어들었음을 의미합니다.

데드크로스란 단기 이동평균선이 중 · 장기 이동평균선을 위에서 아래로 하향 돌파하면서 생기는 교차점을 의미합니다. 데드크로스는 골든크로스와 반대로 최근 주가의 흐름이 약세로 접어들었음을 의미합니다.

골든크로스와 데드크로스를 계산할 때 단기 이동평균선으로는 5일 주가 이동평균선을 사용하며 중 · 장기 이동평균선으로 20일 주가 이동평균선을 사용합니다. 또는 20일 주가 이동평균선과 60일 주가 이동평균선을 각각 단기와 중 · 장기 이동평균선으로 사용하기도 합니다.

이동평균선을 이용한 골든크로스/데드크로스 전략은 대부분의 주식 서적에서 단골로 소개하는 전략입니다. 심지어 일부 책들에서는 골든크로스에 주식을 샀다가 데드크로스점에서 주식을 팔면 항상 이익을 얻을 수 있는 것처럼 과대 포장해서 설명하기도 합니다.

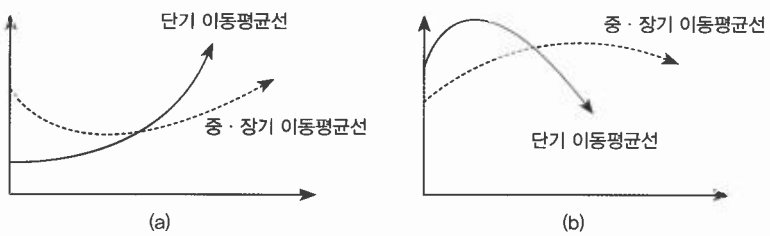


그림 14.23 이동평균선 전략 (a) 골든크로스 (b) 데드크로스

시스템 트레이딩을 공부하는 여러분들은 골든크로스나 데디스크로스를 맹신하기보다는 Zipline과 과거 데이터를 이용해 직접 평가를 해보시기 바랍니다.

Zipline을 이용한 이동평균선 전략 테스트

골든크로스와 데드크로스의 개념을 간단히 알아봤으니, 본격적으로 Zipline을 이용해 골든크로스와 데드크로스 전략을 백테스트해보겠습니다. 앞서 설명한 것처럼 백테스트를 수행하려면 알고리즘과 과거 데이터가 있어야 합니다. 알고리즘은 이미 존재하므로 여러분은 과거 데이터만 준비하면 됩니다.

Jypyter QtConsole(py34)를 실행한 후 pandas_datareader 패키지를 이용해 2010-01-01부터의 애플 주가를 읽어옵니다.

```
In [1]: import pandas_datareader.data as web

In [2]: import datetime

In [3]: start = datetime.datetime(2010, 1, 1)

In [4]: end = datetime.datetime(2016, 3, 29)

In [5]: data = web.DataReader("AAPL", "google", start, end)
```

정상적으로 데이터가 읽어와졌는지 확인하기 위해 그래프를 그려봅시다. 그래프를 Jupyter QtConsole에 바로 출력하려면 %matplotlib inline 명령을 사용하면 됩니다.

```
In [6]: %matplotlib inline

In [7]: import matplotlib.pyplot as plt

In [8]: plt.plot(data.index, data['Close'])
```

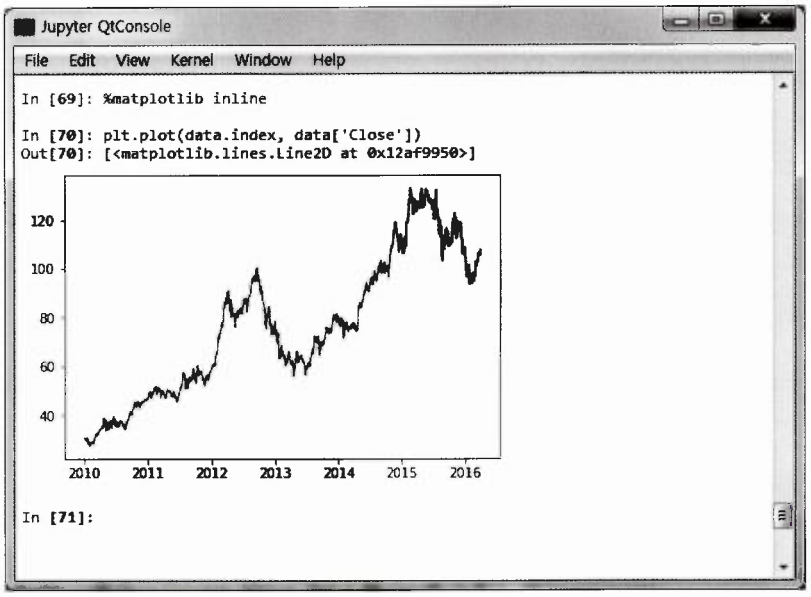


그림 14.24 QtConsole에서 곧바로 matplotlib를 실행

종가로 백테스팅하기 위해 DataFrame 객체에서 해당 칼럼만 가져오고 칼럼의 이름도 'AAPL'로 변경합니다.

```
In [13]: data = data[['Close']]

In [14]: data.columns = ['AAPL']

In [15]: data = data.tz_localize('UTC')

In [16]: data.head()
Out[16]:
                           AAPL
Date
2010-01-04 00:00:00+00:00  30.57
2010-01-05 00:00:00+00:00  30.63
2010-01-06 00:00:00+00:00  30.14
2010-01-07 00:00:00+00:00  30.08
2010-01-08 00:00:00+00:00  30.28

In [17]:
```

앞서 설명한 것처럼 Zipline을 이용해 알고리즘을 백테스트하려면 initialize와 handle_data라는 이름의 함수를 구현해야 합니다. 두 함수를 구현하기에 앞서 필요한 함수를 zipline.api 모듈로부터 임포트합니다.

```
In [17]: from zipline.api import order_target, record, symbol
```

Zipline 모듈이 정상적으로 임포트되면 백테스트의 초기화를 담당하는 initialize 함수부터 구현합니다. initialize 함수는 인자로 context를 사용합니다. 여기서 context는 네임스페이스로서 백테스트를 수행할 때 계속해서 유지해야 할 변수를 저장하는 용도로 사용합니다.

예를 들어, 다음 코드와 같이 변수 i와 sym을 context라는 네임스페이스에 추가할 수 있습니다. 해당 변수들은 시뮬레이션 동안 handle_data 함수에서 참조할 수 있습니다. 뒤에서 다시 설명하겠지만, i라는 변수는 거래일 수를 계산하는 데 사용되는 변수이고 sym은 참조할 데이터에 대한 심볼을 저장하는 데 사용됩니다.

```
In [18]: def initialize(context):
    ...:     context.i = 0
    ...:     context.sym = symbol('AAPL')

In [19]:
```

초기화 함수를 작성했으니 이번에는 백테스트 시뮬레이션 동안 거래일마다 호출되는 함수인 handle_data를 구현해 보겠습니다. handle_data 함수는 인자로 context와 data를 사용하는데 context는 TradingAlgorithm 클래스의 인스턴스이고 data는 BarData 클래스의 인스턴스입니다.

5일 주가 이동 평균과 20일 주가 이동 평균은 시뮬레이션 시작일로부터 최소 20일이 지난 후부터 계산할 수 있습니다. 따라서 다음과 같이 context.i 값이 20보다 작을 때는 handle_data 함수가 종료되도록 구현합니다.

```
In [19]: def handle_data(context, data):
    ...:     context.i += 1
    ...:     if context.i < 20:
    ...:         return
    ...:

In [20]:
```

주가 이동 평균을 계산하려면 일정 기간의 데이터가 필요합니다. Zipline에서는 BarData 클래스의 history 메서드를 사용하면 일정 기간의 데이터를 쉽게 얻을 수 있습니다. 예를 들어, data.

history(context.sym, 'price', 5, '1d')를 호출하면 handle_data 함수가 호출되는 날짜를 기준으로 5일간의 수정 종가 데이터가 저장된 Series 객체를 얻을 수 있습니다.

Series 객체의 데이터에 대한 평균 값을 구하려면 mean 함수를 호출하면 됩니다. 따라서 5일 주가 이동 평균값과 20일 주가 이동 평균값은 다음과 같이 구할 수 있습니다.

```
In [20]: def handle_data(context, data):
    ...:     context.i += 1
    ...:     if context.i < 20:
    ...:         return
    ...:
    ...:     ma5 = data.history(context.sym, 'price', 5, '1d').mean()
    ...:     ma20 = data.history(context.sym, 'price', 20, '1d').mean()
    ...:

In [21]:
```

이번에는 5일 주가 이동 평균값이 20일 주가 이동 평균값보다 큰 상태일 때 1주를 매수하고, 반대면 1주를 매도하는 코드를 추가해 봅시다.

```
In [21]: def handle_data(context, data):
    ...:     context.i += 1
    ...:     if context.i < 20:
    ...:         return
    ...:
    ...:     ma5 = data.history(context.sym, 'price', 5, '1d').mean()
    ...:     ma20 = data.history(context.sym, 'price', 20, '1d').mean()
    ...:
    ...:     if ma5 > ma20:
    ...:         order_target(context.sym, 1)
    ...:     else:
    ...:         order_target(context.sym, -1)
    ...:

In [22]:
```

handle_data 함수는 시뮬레이션 동안 거래일마다 호출되는데, 5일 이동 평균값, 20일 이동 평균값을 저장해두면 향후 전략을 검증하거나 코드를 디버깅할 때 유용하게 활용할 수 있습니다. Zipline에서는 데이터를 저장하기 위해 record 함수를 제공합니다.

```
In [22]: def handle_data(context, data):
    ...:     context.i += 1
    ...:     if context.i < 20:
    ...:         return
    ...:
    ...:     ma5 = data.history(context.sym, 'price', 5, '1d').mean()
    ...:     ma20 = data.history(context.sym, 'price', 20, '1d').mean()
    ...:
    ...:     if ma5 > ma20:
    ...:         order_target(context.sym, 1)
    ...:     else:
    ...:         order_target(context.sym, -1)
    ...:
    ...:     record(AAPL=data.current(context.sym, "price"), ma5=ma5, ma20=ma20)
    ...:

In [15]:
```

백테스팅을 실행하기 위해 TradingAlgorithm 클래스를 임포트합니다.

```
In [15]: from zipline.algorithm import TradingAlgorithm
```

TradingAlgorithm 클래스의 인스턴스를 생성한 후 run 메서드를 호출합니다.

```
In [16]: algo = TradingAlgorithm(initialize=initialize, handle_data=handle_data)
In [17]: result = algo.run(data)
```

백테스팅 시뮬레이션이 끝나면 결괏값은 result라는 변수가 바인딩하는 객체에 저장됩니다. result가 바인딩하고 있는 객체는 pandas의 DataFrame 타입입니다.

```
In [18]: type(result)
Out[18]: pandas.core.frame.DataFrame
```

DataFrame의 요약 정보를 보기 위해 그림 14.25와 같이 info 메서드를 호출해 봅시다. 그림 14.25를 참조하면 result 객체는 41개의 칼럼으로 구성되는데 앞서 record 함수 호출을 통해 저장했던 AAPL, ma5, ma20 칼럼도 존재하는 것을 확인할 수 있습니다.

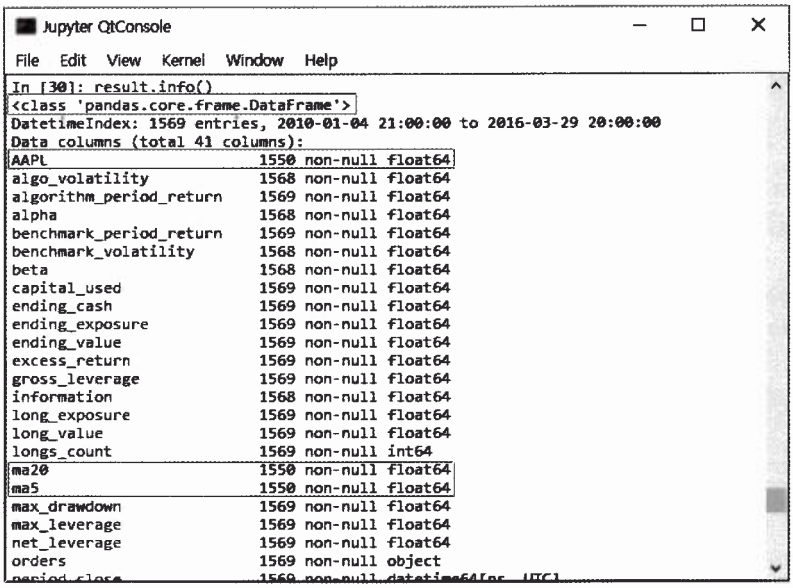


그림 14.25 백테스팅 결과를 저장하는 DataFrame 객체

result 객체에 저장된 ma5, ma20 칼럼의 데이터를 사용해 그림 14.26과 같이 주가 이동 평균선을 그래프로 그려봅시다.

```
In [31]: result[['ma5', 'ma20']].plot()
```

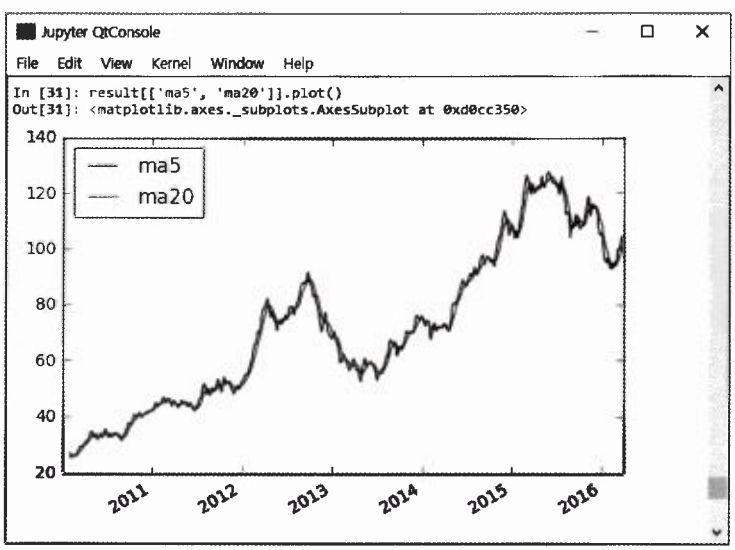


그림 14.26 주가 이동 평균선 그래프(1)

Zipline은 초기 설정을 따로 하지 않으면 100,000달러를 초기 투자 금액으로 사용합니다. DataFrame 객체의 'portfolio_value' 값을 보면 시뮬레이션 종료 시점에서의 보유 현금과 주식 평가 금액의 합산 값을 확인할 수 있습니다.

```
In [36]: result['portfolio_value'].plot()
```

그림 14.27은 portfolio_value 값을 그래프로 그려 본 것입니다. y축은 포트폴리오 값으로 초기 금액인 100,000달러에서 시작함을 확인할 수 있습니다.

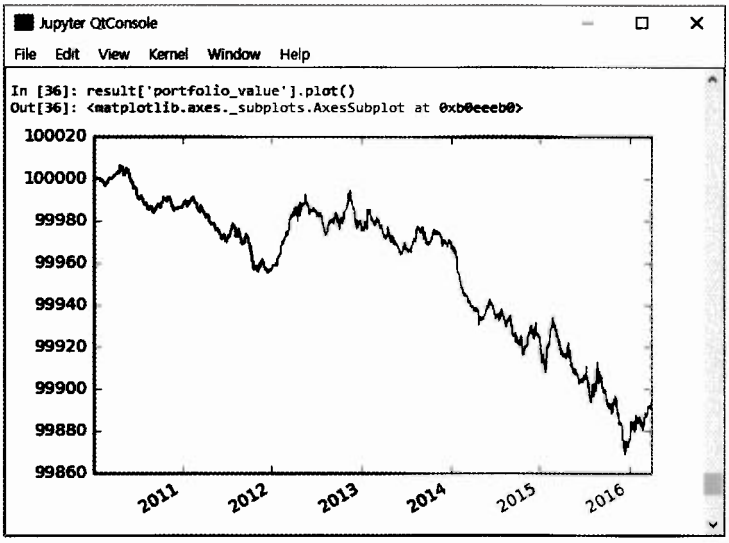


그림 14.27 이동평균선 전략의 포트폴리오

Jupyter QtConsole에서 시뮬레이션 끝부분의 포트폴리오 값을 확인하면 99,892달러임을 확인할 수 있습니다. 시뮬레이션 시작 지점에서 100,000달러를 투자했는데 5년 후는 99,982달러임을 확인할 수 있습니다.

```
In [37]: result['portfolio_value'].tail()
Out[37]:
2016-03-22 20:00:00 99892.033263
2016-03-23 20:00:00 99891.452968
2016-03-24 20:00:00 99891.000539
2016-03-28 20:00:00 99890.528443
2016-03-29 20:00:00 99892.977463
Name: portfolio_value, dtype: float64

In [38]:
```

지금까지 작성한 전체 코드는 예제 14.2와 같습니다.

예제 14.2 이동평균선 알고리즘에 대한 백테스팅(소스코드: book/ch14/03.py)

```
import pandas_datareader.data as web
import datetime
import matplotlib.pyplot as plt
from zipline.api import order_target, record, symbol
from zipline.algorithm import TradingAlgorithm

start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2016, 3, 29)
data = web.DataReader("AAPL", "google", start, end)

#plt.plot(data.index, data['Close'])
#plt.show()

data = data[['Close']]
data.columns = ['AAPL']
data = data.tz_localize('UTC')

#print(data.head())

def initialize(context):
    context.i = 0
    context.sym = symbol('AAPL')

def handle_data(context, data):
    context.i += 1
    if context.i < 20:
        return

    ma5 = data.history(context.sym, 'price', 5, '1d').mean()
    ma20 = data.history(context.sym, 'price', 20, '1d').mean()

    if ma5 > ma20:
        order_target(context.sym, 1)
    else:
        order_target(context.sym, -1)

    record(AAPL=data.current(context.sym, "price"), ma5=ma5, ma20=ma20)
```

```
algo = TradingAlgorithm(initialize=initialize, handle_data=handle_data)
result = algo.run(data)
```

이동평균선 전략 보완하기

앞 절에서 구현한 이동평균선 전략은 5일 이동 평균값이 20일 이동 평균값보다 큰 기간 동안 주식을 1주씩 매수하고 반대로 단기 5일 이동 평균값이 20일 이동 평균값보다 작은 기간 동안 주식을 1주씩 매도했습니다. 이와 달리 일반적인 골든크로스/데드크로스 전략은 골든크로스 지점에서 주식을 매수하고 매수한 주식을 데드크로스 지점에서 매도합니다. 앞서 구현한 전략과의 차이는 매수/매도가 일정 구간 동안 이뤄지는 것이 아니라 골든크로스/데드크로스 시점에서만 발생한다는 것입니다.

이번 절에서는 기존에 구현했던 예제 14.2의 코드에 몇 가지 조건을 추가해서 일반적인 골든크로스/데드크로스 전략을 구현해보겠습니다.

먼저 initialize 함수에서는 주식 보유 여부에 대한 정보를 저장하는 용도의 변수인 context.hold를 추가합니다. 시뮬레이션 초기화 시점에서는 주식을 보유하지 않고 있기 때문에 context.hold의 초깃값은 False가 됩니다.

```
def initialize(context):
    context.i = 0
    context.sym = symbol('AAPL')
    context.hold = False
```

다음으로 handle_data 함수를 수정해 보겠습니다. 주식을 보유하지 않는 상태에서 골든크로스가 발생하면 100주를 매수하고 주식을 보유한 상태에서 데드크로스가 발생하면 100주를 매도하도록 구현합니다. 기존과 달리 매수 여부를 저장하고 있는 변수인 context.hold를 사용하고 골든크로스 지점과 데드크로스 지점에서만 매매가 이뤄집니다.

```
def handle_data(context, data):
    # 코드 생략

    if ma5 > ma20 and context.hold == False:
        order_target(context.sym, 100)
        context.hold = True
    elif ma5 < ma20 and context.hold == True:
        order_target(context.sym, -100)
        context.hold = False
```

마지막으로 buy와 sell이라는 변수를 추가해서 각 시뮬레이션 거래일마다 매수 또는 매도 여부를 저장한 후 이를 record 함수를 사용해 시뮬레이션 결과에 추가합니다. 업데이트된 handle_data 함수는 다음과 같습니다.

```
def handle_data(context, data):
    context.i += 1
    if context.i < 20:
        return

    buy = False
    sell = False

    ma5 = data.history(context.sym, 'price', 5, '1d').mean()
    ma20 = data.history(context.sym, 'price', 20, '1d').mean()

    if ma5 > ma20 and context.hold == False:
        order_target(context.sym, 100)
        context.hold = True
        buy = True
    elif ma5 < ma20 and context.hold == True:
        order_target(context.sym, -100)
        context.hold = False
        sell = True

    record(AAPL=data.current(context.sym, "price"), ma5=ma5, ma20=ma20, buy=buy, sell=sell)
```

Zipline 시뮬레이션을 수행하는 전체 코드는 예제 14.3과 같습니다.

예제 14.3 이동평균선 알고리즘에 대한 백테스팅(2)(소스코드: book/ch14/04.py)

```
import pandas_datareader.data as web
import datetime
import matplotlib.pyplot as plt
from zipline.api import order_target, record, symbol
from zipline.algorithm import TradingAlgorithm

start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2016, 3, 29)
data = web.DataReader("AAPL", "google", start, end)
```

```
data = data[['Close']]
data.columns = ['AAPL']
data = data.tz_localize('UTC')

def initialize(context):
    context.i = 0
    context.sym = symbol('AAPL')
    context.hold = False

def handle_data(context, data):
    context.i += 1
    if context.i < 20:
        return

    buy = False
    sell = False

    ma5 = data.history(context.sym, 'price', 5, '1d').mean()
    ma20 = data.history(context.sym, 'price', 20, '1d').mean()

    if ma5 > ma20 and context.hold == False:
        order_target(context.sym, 100)
        context.hold = True
        buy = True
    elif ma5 < ma20 and context.hold == True:
        order_target(context.sym, -100)
        context.hold = False
        sell = True

    record(AAPL=data.current(context.sym, "price"), ma5=ma5, ma20=ma20, buy=buy, sell=sell)

algo = TradingAlgorithm(initialize=initialize, handle_data=handle_data)
result = algo.run(data)
```

정상적으로 시뮬레이션이 수행되면 result 객체의 'portfolio_value' 칼럼의 값을 그래프로 그려보기 바랍니다. 다음 코드를 예제 14.3의 마지막 부분에 추가하면 됩니다.

```
plt.plot(result.index, result.portfolio_value)
plt.show()
```

그림 14.28은 보완된 이동평균선 전략의 포트폴리오 추세를 보여줍니다. 중간에 잠깐 수익이 나는 시점이 있기도 했지만 여전히 마이너스의 수익률을 보입니다. 주식 서적에서 좋은 전략처럼 소개하는 것치고는 수익률이 높지 않은 것을 확인할 수 있습니다.

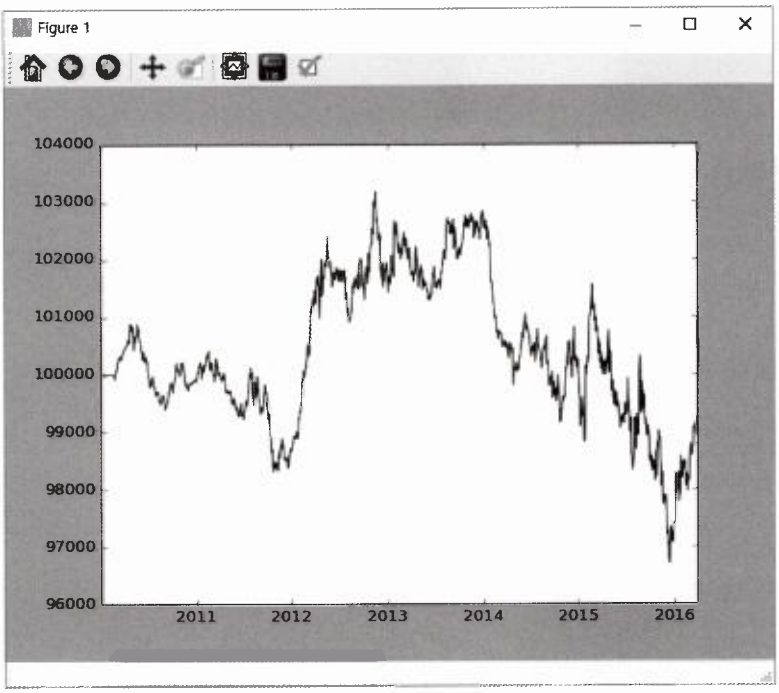


그림 14.28 보완된 알고리즘의 포트폴리오 결과

이번에는 백테스팅 기간 동안의 매수/매도 시점을 그래프로 그려보겠습니다. 먼저 result 객체의 'ma5'와 'ma20'이라는 칼럼 값을 그래프로 그린 후 범례를 추가합니다.

```
plt.plot(result.index, result.ma5)
plt.plot(result.index, result.ma20)
plt.legend(loc='best')
```

매수 시점은 '^'로 표시하고 매도 시점에는 'v'로 표시합니다.

```
plt.plot(result.ix[result.buy == True].index, result.ma5[result.buy == True], '^')
plt.plot(result.ix[result.sell == True].index, result.ma5[result.sell == True], 'v')
```

그래프를 출력할 준비가 되면 plt.show()를 호출해서 그래프를 화면에 출력합니다.

```
plt.show()
```

그림 14.29에 나타난 그래프에 5일선과 20일 이동 평균선이 출력됐고 그 위로 매수/매도 시점도 함께 표시된 것을 확인할 수 있습니다.

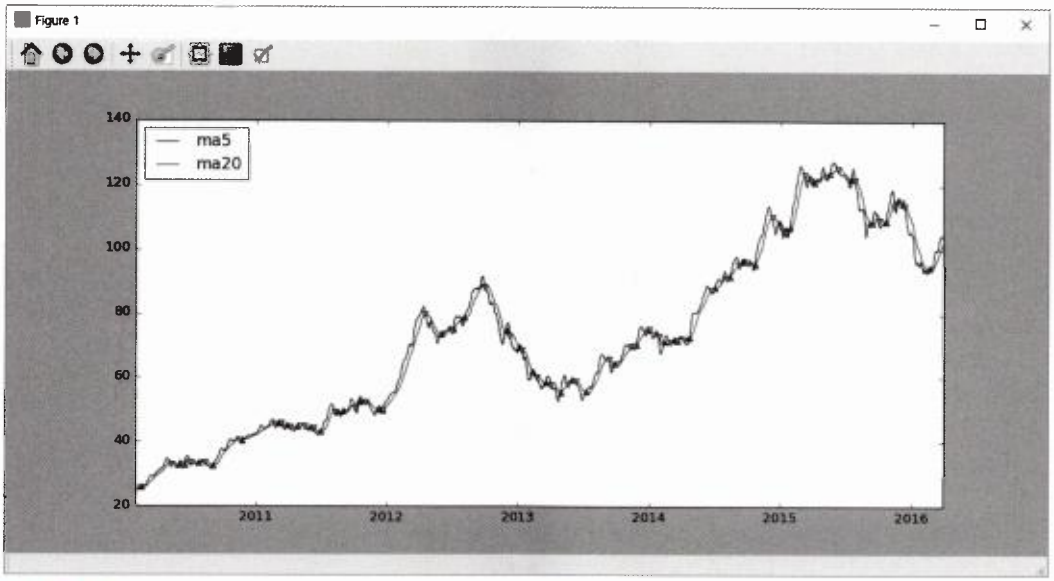


그림 14.29 매수/매도 시점 표시

03 유가증권시장/코스닥시장 백테스팅

이번 절에서는 Zipline을 이용해 유가증권시장/코스닥시장 종목에 대한 백테스트를 수행할 때 고려해야 하는 사항을 살펴보겠습니다. 초기 투자 금액이나 거래 수수료 설정 등이 여기에 해당합니다.

거래 수수료 설정

Zipline은 수수료를 따로 설정하지 않으면 기본값으로 주당 0.03달러를 사용합니다. 코스피의 GS 종목을 매 거래일마다 한 주씩 매수하면서 거래 수수료가 어떻게 계산되는지 살펴보겠습니다. 먼저 다음 코드를 PyCharm에서 실행해 백테스팅을 수행합니다.

```
import pandas_datareader.data as web
import datetime
from zipline.api import order_target, record, symbol
from zipline.algorithm import TradingAlgorithm
import matplotlib.pyplot as plt
```

```
start = datetime.datetime(2016, 1, 1)
end = datetime.datetime(2016, 1, 31)
data = web.DataReader("KRX:078930", "google", start, end)

data = data[['Close']]
data.columns = ['GS']
data = data.tz_localize('UTC')

def initialize(context):
    context.i = 0
    context.sym = symbol('GS')

def handle_data(context, data):
    order_target(context.sym, 1)

algo = TradingAlgorithm(initialize=initialize, handle_data=handle_data)
result = algo.run(data)

print(result[['starting_cash', 'ending_cash', 'ending_value']])
```

실행 결과는 다음과 같습니다.

```
                     starting_cash     ending_cash     ending_value
2016-01-04 21:00:00      100000           100000          0
2016-01-05 21:00:00      100000           50399           49600
2016-01-06 21:00:00      50399            50399           47600
# 중간 생략
2016-01-29 21:00:00      50399            50399           50400
```

위 결과를 참조하면 2016-01-05일에 종목 매수가 이뤄진 것을 확인할 수 있습니다. 2016-01-05일의 ending_cash 값이 50399인 것은 100,000원에서 해당일의 종가인 49,600원을 빼고 다시 기본 수수료인 0.03원을 뺐기 때문입니다.

여기서 한 가지 주의해야 할 점은 Zipline 모듈은 기본적으로 달러를 기준으로 동작하는데 유가증권시장/코스닥시장은 원 단위라는 점입니다. 이 문제를 가장 쉽게 해결하려면 유가증권시장/코스닥시장의 데이터를 사용하는 경우 Zipline 모듈의 값도 원 단위로 간주하면 됩니다.

Zipline에서 거래 수수료를 설정하려면 set_comminsion 메서드를 initialize 메서드에서 호출하면 됩니다. Zipline은 PerTrade, PerShare, PerDollar라는 세 종류의 commission 모델을 제공합니다. PerTrade는 거래를 할 때 거래 금액과는 상관없이 일정 수수료를 지급하는 모델입니다. PerShare는 주당 정해진 금액의 수수료를 지급하는 모델입니다. PerDollar는 달러당 일정 수수료를 내는 모델입니다.

유가증권시장/코스닥시장에서는 주식을 매수할 때는 세금을 내지 않지만 매도 시에는 매도 금액의 0.3%에 해당하는 금액인 주식거래세를 내야 합니다. 여기에 추가로 약 0.015% 정도의 증권사 수수료가 매수와 매도 시점에서 모두 발생합니다. 아직까지는 zipline에서 유가증권/코스닥 시장과 정확히 일치하는 수수료 모델을 만들기는 어렵지만 총 0.33%의 부대 비용이 발생하는 것을 매수/매도 시점에 나눠서 0.165%씩 설정하면 그나마 비슷하게 설정할 수 있습니다.

수수료 설정에 사용되는 함수나 모듈을 임포트합니다.

```
from zipline.api import set_commission, commission
```

앞서 설명한 것처럼 initialize 함수에서 set_commission 메서드를 호출해 수수료를 설정합니다. 0.165%이므로 1원을 기준으로 하면 0.00165원이 수수료입니다. PerDollar는 원래는 달러를 기준으로 수수료를 설정하는 것이지만 우리는 원 단위로 생각하면 되기 때문에 1원에 대한 수수료로 간주하면 됩니다.

```
def initialize(context):
    set_commission(commission.PerDollar(cost=0.00165))
```

수정된 코드로 다시 백테스팅을 수행합니다. 결과는 다음과 같습니다.

```
                           starting_cash     ending_cash     ending_value
2016-01-04 21:00:00        100000.00         100000.00       0
2016-01-05 21:00:00        100000.00         50318.16        49600
2016-01-06 21:00:00        50318.16          50318.16        47600
# 중간 생략
2016-01-29 21:00:00        50318.16          50318.16        50400
```

백테스팅 결과를 확인해보면 우리가 의도했던 수수료가 설정된 것을 확인할 수 있습니다. 초기 투자 금액인 100,000원에서 주식 한 주당 금액인 49,600원을 빼면 50,400원입니다. 여기서 수수료 81.84원(=49,600x0.00165)을 빼면 50318.16원이 남게 됩니다.

초기 투자 금액 설정

Zipline은 기본 거래 단위가 달러입니다. 따라서 초기 투자 금액인 100,000달러는 원화 기준으로는 약 1억 원이 넘는 꽤 큰 금액입니다. 그러나 100,000이라는 값을 원 단위로 해석하면 상당히 적은 금액입니다.

이번 절에서는 Zipline에서 초기 투자 금액을 설정하는 방법을 배워보겠습니다. 초기 투자 금액은 TradingAlgorithm 클래스의 생성자로 sim_params 값을 지정해 쉽게 변경할 수 있습니다.

예제 14.4 Zipline 초기 투자 금액 설정(소스코드: book/ch14/07.py)

```
import pandas_datareader.data as web
import datetime
from zipline.api import order, record, symbol
from zipline.algorithm import TradingAlgorithm
from zipline.api import set_commission, commission
from zipline.utils.factory import create_simulation_parameters
import matplotlib.pyplot as plt

start = datetime.datetime(2016, 1, 1)
end = datetime.datetime(2016, 1, 31)
data = web.DataReader("KRX:078930", "google", start, end)

data = data[['Close']]
data.columns = ['GS']
data = data.tz_localize('UTC')

def initialize(context):
    context.i = 0
    context.sym = symbol('GS')
    set_commission(commission.PerDollar(cost=0.00165))

def handle_data(context, data):
    #order_target(context.sym, 1)
    order(context.sym, 1)

algo = TradingAlgorithm(sim_params=create_simulation_parameters(capital_base=100000000), ini-
tialize=initialize, handle_data=handle_data)
result = algo.run(data)

print(result[['starting_cash', 'ending_cash', 'ending_value']])
```

백테스팅을 수행한 결과는 다음과 같습니다. starting_cash 값이 TradingAlgorithm 클래스의 생성자에서 설정한 100,000,000원임을 확인할 수 있습니다.

```
                           starting_cash      ending_cash        ending_value
2016-01-04 21:00:00    100000000.0000    100000000.0000    0
2016-01-05 21:00:00    100000000.0000    99950318.1600     49600
2016-01-06 21:00:00    99950318.1600     99902639.6200     95200
2016-01-07 21:00:00    99902639.6200     99855612.1525     140850
# 생략
```

15

matplotlib를 이용한 데이터 시각화

많은 양의 데이터를 분석할 때는 데이터 값 자체를 살펴보기보다는 그래프나 차트 형태로 그려보는 것이 큰 도움이 될 때가 많습니다. 예를 들어, 주식 투자자들도 주식 데이터를 분석할 때 여러 거래일의 시가, 종가, 고가, 저가의 수치를 일일이 살펴보지는 않습니다. 대신 HTS가 제공하는 봉 차트를 살펴봄으로써 전체적인 추세를 확인합니다.

파이썬을 이용하면 HTS에서 제공하는 다양한 차트를 여러분도 손쉽게 그릴 수 있습니다. 파이썬은 데이터 시각화를 위한 다양한 라이브러리를 제공하는데, 이번 장에서는 그중 matplotlib라는 라이브러리에 대해 다룰 예정입니다.

matplotlib는 파이썬에서 2차원 그래프를 그릴 때 가장 널리 사용되는 라이브러리입니다. 특히 출판물에 사용할 정도의 품질로 도표를 쉽게 만들 수 있어 과학이나 공학 분야에서도 자주 사용됩니다. matplotlib 라이브러리를 익혀둔다면 주식 투자 외에도 다양한 분야에서 활용할 수 있을 것입니다.

01 pyplot 기초

Mathworks에서 개발되어 공학이나 과학 분야에서 자주 사용되는 프로그래밍 언어로 매트랩(MATLAB)이 있습니다. 매트랩은 공학 및 과학 문제 해결에 최적화된 프로그래밍 환경으로서 다양한 분야에서 활용되고 있습니다.

matplotlib의 pyplot 모듈은 매트랩과 비슷한 형태로 그래프를 그리는 기능을 제공합니다. 참고로 앞 장에서 그린 그래프에는 pyplot 모듈을 사용했습니다. pyplot 모듈만 제대로 익혀도 웬만한 데이터는 모두 효과적으로 도식화할 수 있습니다.

간단한 그래프 그리기

간단한 형태의 그래프를 그려보면서 pyplot 모듈을 배워보겠습니다. pyplot 모듈을 사용하려면 먼저 해당 모듈을 임포트해야 합니다. pyplot 모듈은 보통 다음과 같은 형태로 임포트합니다.

```
In [1]: import matplotlib.pyplot as plt
```

만약 임포트 에러가 발생한다면 matplotlib 라이브러리가 설치되지 않은 것입니다. matplotlib 라이브러리는 기본적으로 설치돼 있지 않기 때문에 직접 설치하거나 아나콘다와 같은 배포판을 사용해야 합니다.

matplotlib 라이브러리의 pyplot 모듈은 그래프를 그리는 데 사용되는 다양한 함수를 포함하고 있습니다. 그중 plot 함수는 값을 서로 연결해서 라인 형태의 그래프를 그리는 함수입니다. 잘 이해가 되지 않는 분들은 먼저 다음 코드를 실행해 보기 바랍니다.

```
In [17]: plt.plot([1, 2, 3, 4])
Out[17]: [<matplotlib.lines.Line2D at 0x5262c50>]
In [18]: plt.show()
```

show 함수를 호출하면 그림 15.1과 같은 그래프가 출력됩니다. 그림 15.1을 살펴보면 y축 값 1, 2, 3, 4를 지나는 직선이 그려진 것을 확인할 수 있습니다. 참고로 plot 함수를 호출할 때 x 축 값을 따로 지정하지 않으면 자동으로 실숫값이 할당됩니다. plot 함수에서 y축 값이 네 개였으므로 이에 대응하는 x축 값은 0.0, 1.0, 2.0, 3.0이 됩니다.

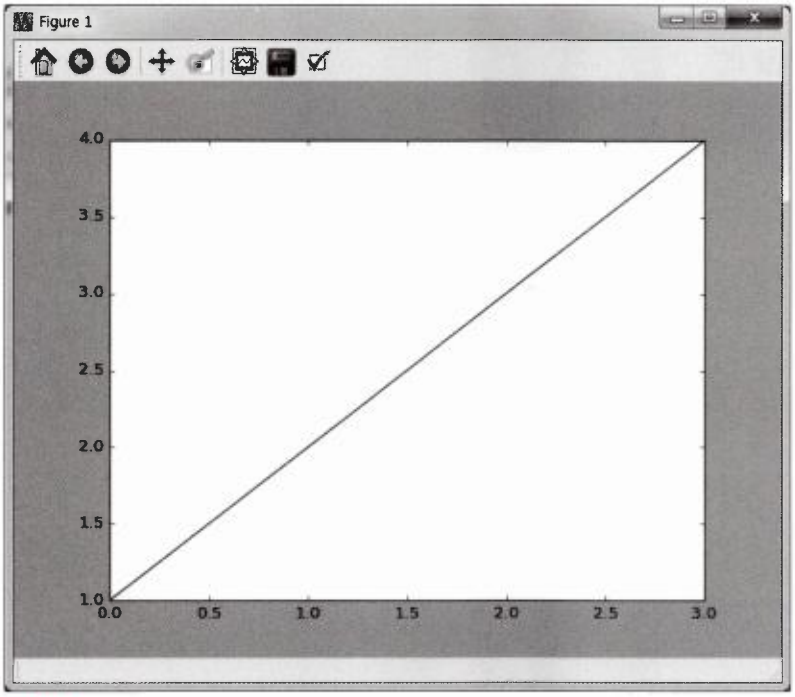


그림 15.1 plot을 이용해 그래프 그리기(소스코드: book/ch15/01.py)

plot 함수는 x축 값과 y축 값을 동시에 받을 수도 있습니다. 예를 들어, 중학교 수학시간에 배우는 $y=x^2$ $(x>=0)$ 그래프는 다음과 같이 그릴 수 있습니다.

```
In [4]: x = range(0, 100)
In [5]: y = [v*v for v in x]
In [6]: plt.plot(x, y)
Out[6]: [<matplotlib.lines.Line2D at 0x523e490>]
In [7]: plt.show()
```

x는 0~99 범위의 정숫값인데 y를 구하는 방법이 조금 독특합니다. 위와 같은 표현을 리스트 내장이라고 하는데, 리스트 내장은 리스트를 쉽게 생성할 수 있기 때문에 자주 사용됩니다. 만약 리스트 내장을 사용하지 않는다면 y는 다음과 같이 for 문을 통해서도 구성할 수 있습니다. 그러나 이 경우 코드가 리스트 내장보다 길어진다는 단점이 있습니다. 아무래도 코드가 길어진다면 나중에 읽고 해석해야 하는 코드가 많다는 것이므로 가독성 측면에서는 좋지 않겠지요?

```
In [8]: y = []
In [9]: for v in x:
   ...:     y.append(v*v)
```

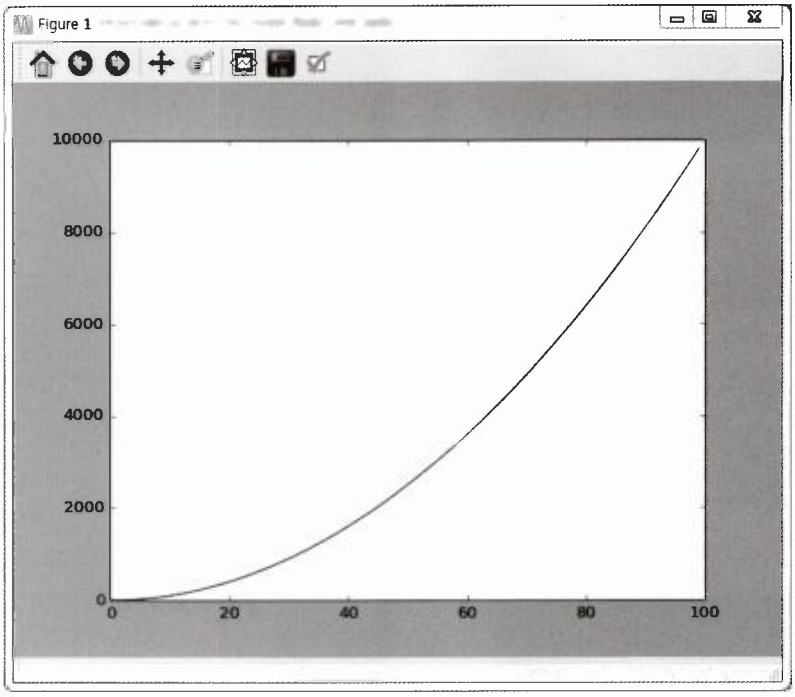


그림 15.2 $y=x^2$ (x >=0) 그래프(소스코드: book/ch15/02.py)

plot 함수를 이용하면 그래프의 색상이나 형태도 쉽게 변경할 수 있습니다. plot 함수에서 따로 인자를 설정하지 않으면 기본 포맷은 파란색을 사용해 구간 사이를 직선으로 연결하는 스타일인 'b-'가 사용됩니다. 만약 빨간색의 원을 그리려면 'ro'를 사용하면 됩니다.

```
In [11]: x = range(0, 100)
In [12]: y = [v*v for v in x]
In [13]: plt.plot(x, y, 'ro')
Out[13]: [<matplotlib.lines.Line2D at 0x5287570>]
```

그림 15.3은 plot 함수에서 'ro' 인자를 통해 그래프의 색상과 스타일을 변경한 것입니다. 'ro' 인자에서 'r'은 red를 의미하고 'o'는 그래프의 마커 모양을 의미합니다.

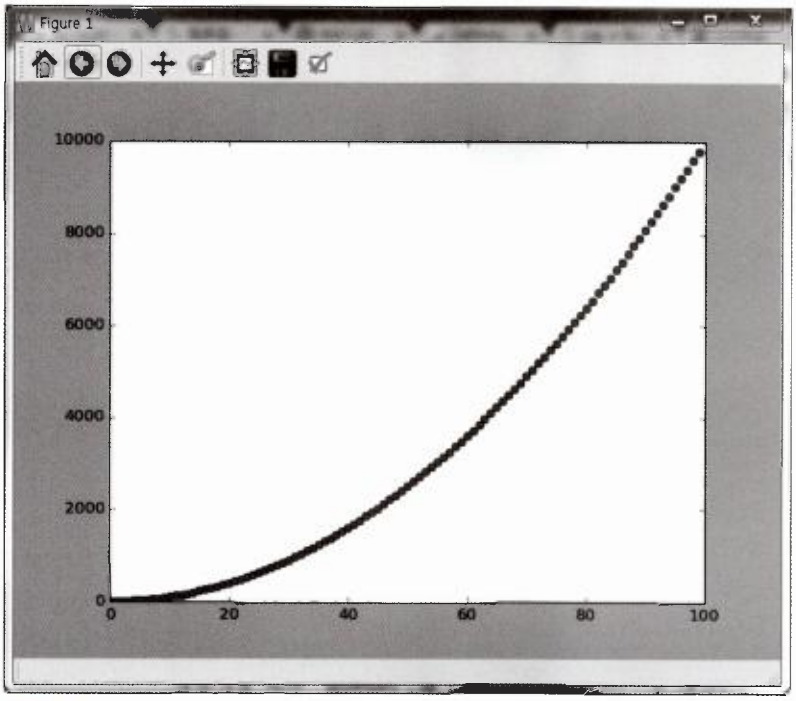


그림 15.3 'ro' 스타일의 그래프(소스코드: book/ch15/02.py, 주석 해제 후 동작)

자주 사용하는 마커 패턴으로는 '--', 's', '^', '+' 등이 있습니다. 색상과 마커 패턴을 조합한 'r--'는 빨간색 대시라인을 의미하고, 'bs'는 파란색 사각형, 'g^'는 녹색 삼각형, 'g+'는 녹색 플러스 모양을 의미합니다.

matplotlib에서 자주 사용되는 색상은 표 15.1과 같습니다.

표 15.1 matplotlib의 주요 색상

문자	색상
b	blue(파란색)
g	green(녹색)
r	red(빨간색)
c	cyan(청록색)
m	magenta(마젠타색)
y	yellow(노란색)
k	black(검은색)
w	white(흰색)

matplotlib에서 자주 사용되는 마커는 표 15.2와 같습니다.

표 15.2 matplotlib의 주요 마커

마커	의미
o	circle(원)
v	triangle_down(역삼각형)
^	triangle_up(삼각형)
s	square(네모)
+	plus(플러스)
.	point(점)

Figure와 subplots

이번 절에서는 그림 15.4와 같이 한 화면에 여러 개의 그래프를 그리는 방법을 배워 보겠습니다.

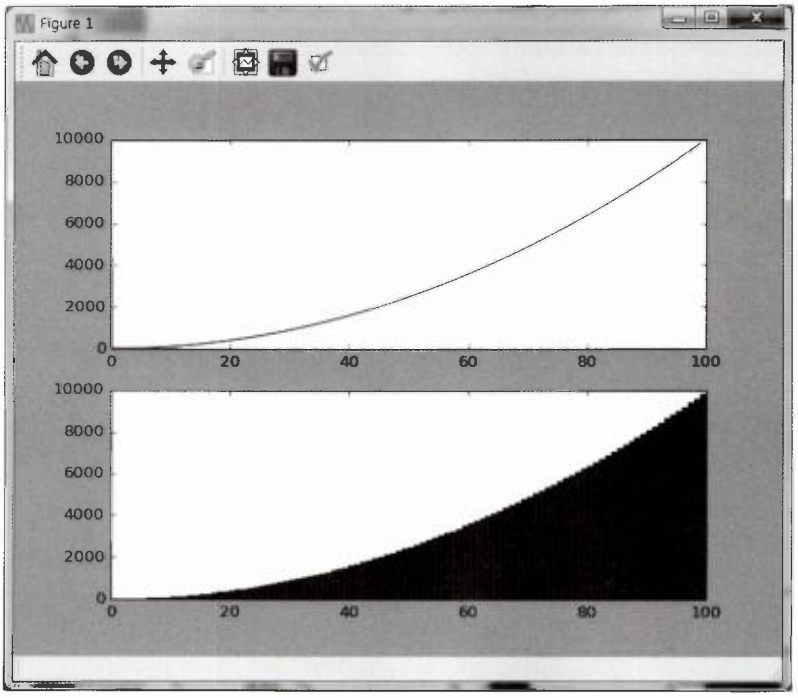


그림 15.4 한 화면에 여러 개의 그래프 그리기(소스코드: book/ch15/04.py)

한 화면에 여러 개의 그래프를 그리려면 figure 함수를 통해 Figure 객체를 먼저 만든 후 add_subplot 메서드를 통해 그리려는 그래프 개수만큼 subplot을 만들면 됩니다.

subplot의 개수는 add_subplot 메서드의 인자를 통해 조정할 수 있습니다. 다음 코드에서 (2, 1, 1)은 2x1(행x열)의 subplot을 생성한다는 의미이고 세 번째 인자 1은 생성된 두 개의 subplot 중 첫 번째 subplot을 의미합니다. 마찬가지로 (2, 1, 2)는 2x1 subplot에서 두 번째 subplot을 의미합니다.

```
In [46]: fig = plt.figure()
In [48]: ax1 = fig.add_subplot(2, 1, 1)
In [49]: ax2 = fig.add_subplot(2, 1, 2)
In [50]: plt.show()
```

그림 15.5는 위의 코드를 실행한 결과입니다. Figure 안에 두 개의 subplot이 존재하는데, 행 방향으로 2개, 열 방향으로는 1개의 subplot이 위치함을 확인할 수 있습니다.

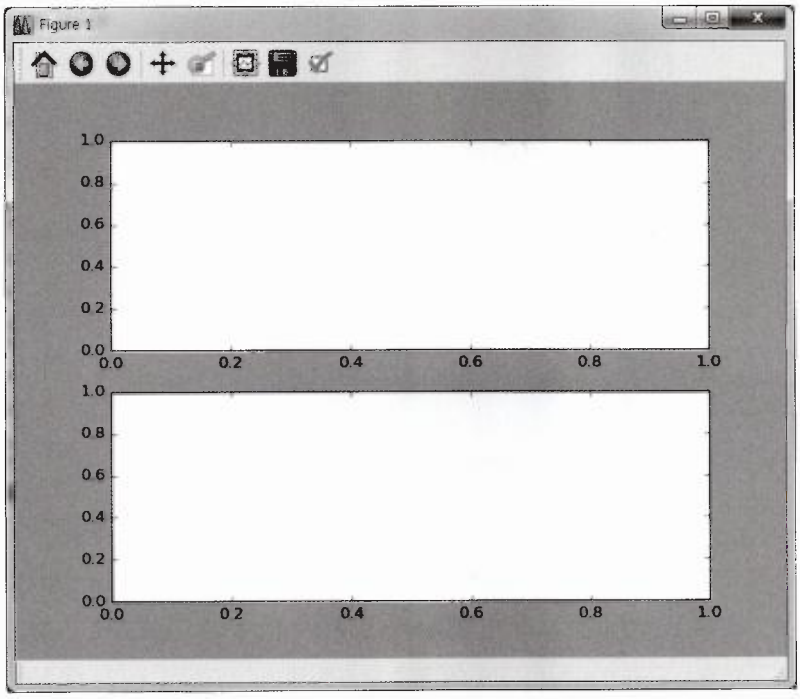


그림 15.5 2x1 subplot(소스코드: book/ch15/03.py)

add_subplot 메서드의 인자를 변경해 1x2 형태의 subplot을 만들어 봅시다. 다음 코드를 실행하면 그림 15.6과 같이 1x2 형태의 subplot이 생성되는 것을 확인할 수 있습니다. 아직까지는 생성된 subplot에 그래프를 그리지 않았기 때문에 아무것도 출력되지는 않지만 add_subplot을 통해 subplot을 구성하는 것은 확실히 이해하셨을 것입니다.

```
In [52]: fig = plt.figure()
In [53]: ax1 = fig.add_subplot(1, 2, 1)
In [54]: ax2 = fig.add_subplot(1, 2, 2)
In [55]: plt.show()
```

참고로 add_subplot 메서드를 호출하면 AxesSubplot 객체가 생성됩니다. 위 코드에서는 ax1, ax2라는 변수를 통해 생성된 객체를 바인딩했습니다. matplotlib 라이브러리의 도움말 파일을 보다 보면 Axes라는 용어가 종종 사용되는데, Axes가 하나의 subplot과 유사한 개념이라고 이해하면 됩니다.

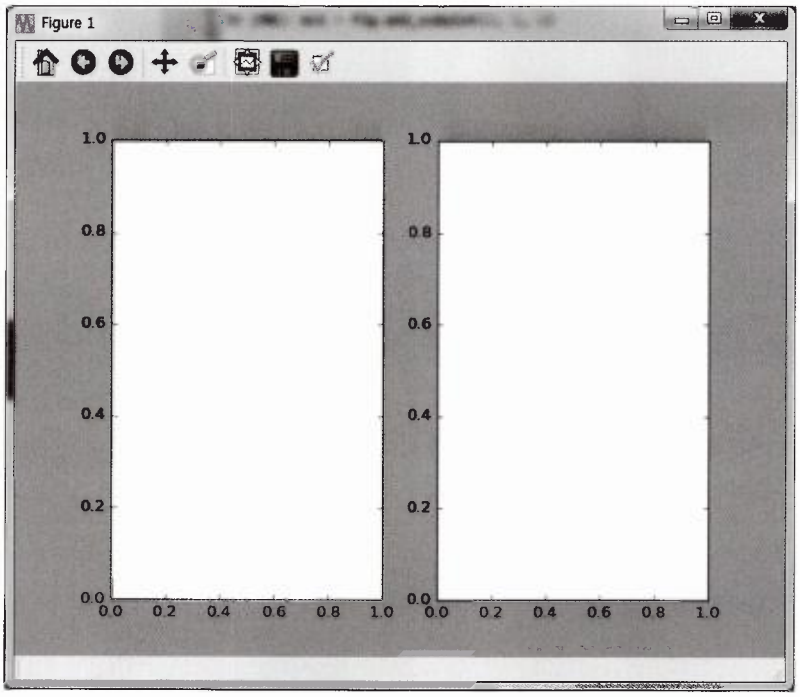


그림 15.6 1x2 subplots

다음은 그림 15.4의 그래프를 그리는 코드입니다. 2x1의 subplot을 생성했고 첫 번째 subplot (ax1)에는 plot 함수를 호출해서 그래프를 그렸습니다. 두 번째 subplot (ax2)에는 bar라는 함수를 호출해서 그래프를 그렸습니다. 참고로 bar 함수는 pyplot 모듈에 있는 함수로 막대그래프를 그리는 데 사용합니다.

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax1 = fig.add_subplot(2, 1, 1)
ax2 = fig.add_subplot(2, 1, 2)
```

```
x = range(0, 100)
y = [v*v for v in x]

ax1.plot(x, y)
ax2.bar(x, y)

plt.show()
```

이번에는 그림 15.7과 같이 sine과 cosine 그래프를 그려보겠습니다. 첫 번째 subplot에는 sine 그래프를 그리고 두 번째 subplot에는 cosine 그래프를 그리겠습니다.

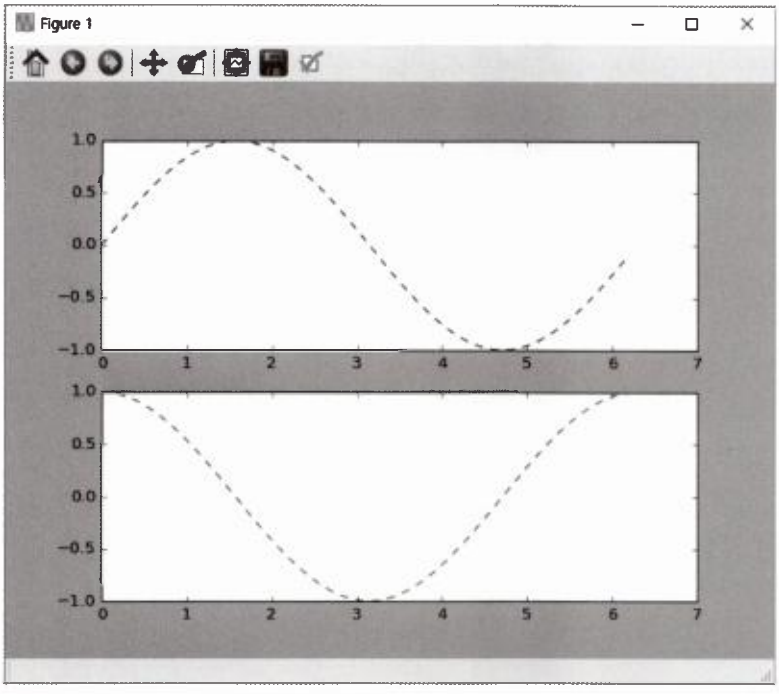


그림 15.7 sine과 cosine 그래프-1(소스코드: book/ch15/05.py)

0~2π 범위의 x에 대해 y=sin(x)와 y=cos(x)를 그리려면 x와 π가 있어야 합니다. x 값이 실수 범위이기 때문에 range를 사용할 수는 없고 numpy 모듈의 arange를 사용해야 합니다. 다음과 같이 numpy 모듈을 먼저 임포트한 후 arange 함수를 호출해서 0.0~2π 사이에서 0.1 간격으로 x 값을 만들어보기 바랍니다. 참고로 π는 numpy 모듈의 pi를 사용하면 됩니다.

```
In [63]: import numpy as np

In [64]: x = np.arange(0.0, 2*np.pi, 0.1)
```

x 값을 살펴보면 다음과 같이 0.1씩 증가하는 것을 확인할 수 있습니다.

```
Out[74]:
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
        2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
        3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3,
        4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,  5.4,
        5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2])
```

그림 15.7의 그래프를 그리는 전체 코드는 다음과 같습니다. 앞서 설명한 것처럼 두 개의 subplot을 추가하기 위해 add_subplot 메서드를 사용했습니다. 그리고 각 subplot에서 plot 함수를 호출해 y=sin(x)와 y=cos(x) 그래프를 그렸습니다. sine 그래프에는 'b--'를 사용해 파란색의 대시라인 마커로 표현했고, cosine 그래프에는 'r--'를 사용해 빨간색의 대시라인 마커로 표현했습니다.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0.0, 2 * np.pi, 0.1)
sin_y = np.sin(x)
cos_y = np.cos(x)

fig = plt.figure()
ax1 = fig.add_subplot(2, 1, 1)
ax2 = fig.add_subplot(2, 1, 2)

ax1.plot(x, sin_y, 'b--')
ax2.plot(x, cos_y, 'r--')

plt.show()
```

라벨 및 범례 표시하기

그래프를 그릴 때 가장 주의해야 할 점은 그래프의 x축과 y축의 값이 어떤 데이터인지를 표시하는 것입니다. 앞에서 그린 그림 15.7의 그래프를 보면 x축과 y축의 값이 어떤 의미를 갖는 데이터인지 알 수 없습니다. 물론 그래프를 그린 본인은 알겠지만 해당 그래프를 보는 다른 사람들은 전혀 알 수가 없을 것입니다.

그림 15.8은 그림 15.7의 그래프에서 x축과 y축에 라벨을 추가한 것입니다. 이처럼 그래프를 그릴 때는 항상 각 축에 라벨을 붙여두는 것이 좋습니다.

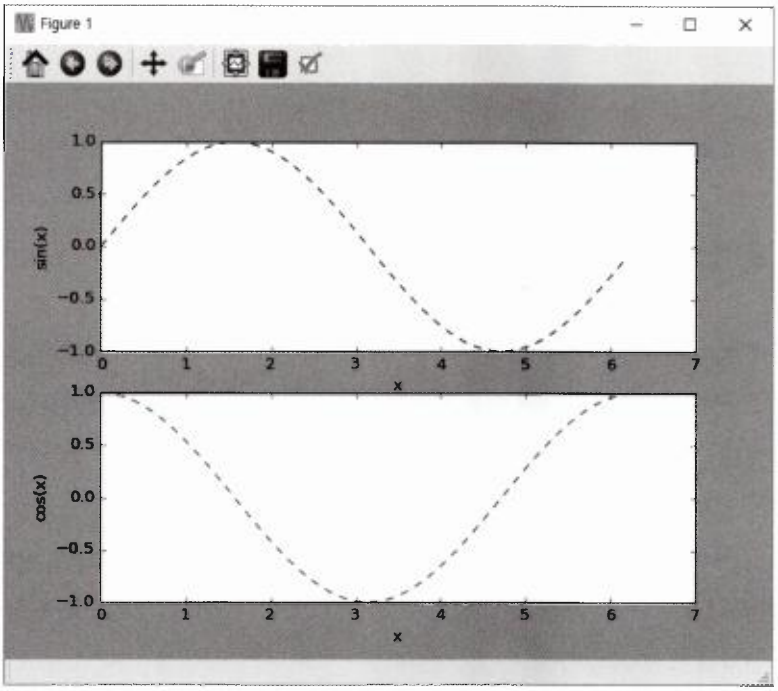


그림 15.8 sine과 cosine 그래프-2(소스코드: book/ch15/06.py)

그림 15.8에는 두 개의 subplot이 있으므로 각 subplot에 대해 x축과 y축 라벨을 설정해야 합니다. subplot은 앞서 설명한 것처럼 Axes 객체입니다. Axes 객체는 set_xlabel과 set_ylabel 메서드를 통해 x축과 y축에 라벨을 설정할 수 있습니다.

다음은 set_xlabel과 set_ylabel 메서드를 이용해 subplot의 x축과 y축에 라벨을 추가한 코드입니다. matplotlib 라이브러리는 아주 쉽게 새로운 기능을 추가할 수 있음을 확인할 수 있습니다.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0.0, 2 * np.pi, 0.1)
sin_y = np.sin(x)
cos_y = np.cos(x)

fig = plt.figure()
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)

ax1.plot(x, sin_y, 'b--')
```

```
ax2.plot(x, cos_y, 'r--')

ax1.set_xlabel('x')
ax1.set_ylabel('sin(x)')

ax2.set_xlabel('x')
ax2.set_ylabel('cos(x)')

plt.show()
```

시중에 판매되고 있는 주식 차트 분석 책들을 보면 동일 업종의 주가를 참조하는 기법들이 소개되곤 합니다. 이번에는 그림 15.9와 같이 국내 전자 분야의 대표 기업인 LG전자와 삼성전자의 주가를 하나의 그래프로 그려보겠습니다. 이처럼 하나의 공간에 두 개의 그래프를 그리는 경우 각 그래프가 어떤 데이터를 의미하는지 범례를 통해 구분하는 것이 좋습니다.

그래프와는 상관 없는 이야기지만 그림 15.9를 보면 2010년 이후에 삼성전자의 주가는 큰 상승세를 보인 반면 LG전자는 약간의 하락세를 보이는 것을 확인할 수 있습니다. 동일한 업종이라고 해서 항상 주가 그래프가 유사한 것은 아님을 알 수 있습니다.

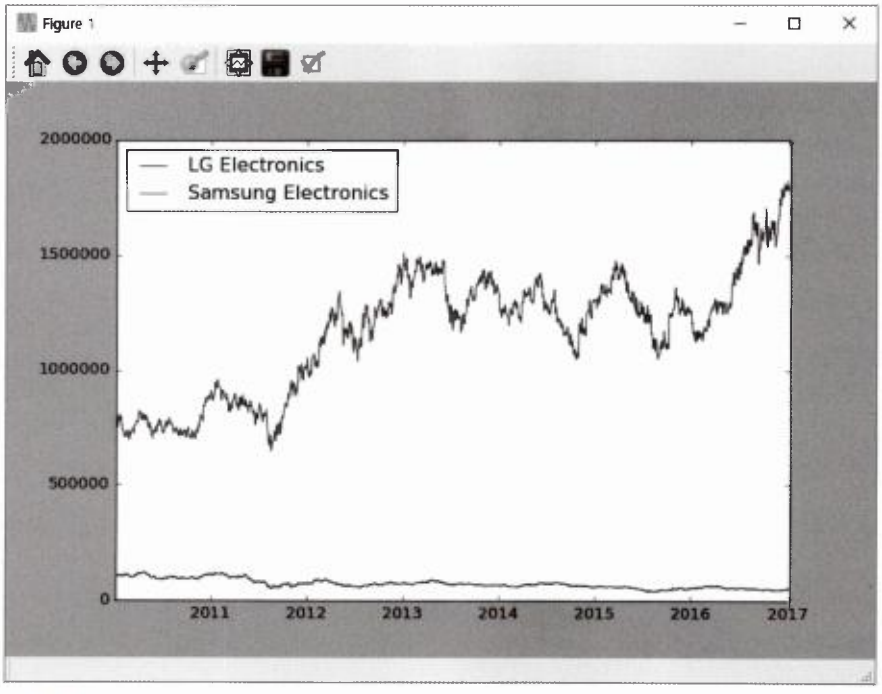


그림 15.9 범례를 이용한 동종 업종 주가 그래프(소스코드: book/ch15/07.py)

그림 15.9의 그래프를 그리려면 먼저 LG전자와 삼성전자의 종가 데이터가 필요합니다. 종가는 13장에서 설명한 pandas_datareader 모듈의 DataReader 함수를 통해 구하면 됩니다. DataReader 함수를 통해 데이터를 얻어올 때 따로 기간을 설정하지 않으면 2010-01-01일 이후부터 데이터를 얻어오게 됩니다.

```
In [1]: import pandas_datareader.data as web

In [2]: lg = web.DataReader("KRX:066570", "google")

In [3]: samsung = web.DataReader("KRX:005930", "google")
```

lg나 saumsung은 DataFrame 객체를 바인딩하고 있는데, 이 가운데 'Close'라는 칼럼에 종가 데이터가 저장돼 있습니다. 그래프를 그리기 위해 먼저 matplotlib.pyplot 모듈을 임포트한 후 각각의 종가를 plot 함수를 통해 그립니다. 이때 label이라는 인자를 통해 라벨을 지정합니다. label을 통해 전달하는 문자열이 범례에 표시됩니다.

```
In [5]: import matplotlib.pyplot as plt
In [14]: plt.plot(lg.index, lg['Close'], label='LG Electronics')
Out[14]: [<matplotlib.lines.Line2D at 0x91cc8d0>]

In [15]: plt.plot(samsung.index, samsung['Close'], label='Samsung Electronics')
Out[15]: [<matplotlib.lines.Line2D at 0x91cc7b0>]
```

범례는 legend 함수를 통해 추가할 수 있습니다. 이때 loc라는 함수 인자를 통해 범례가 표시되는 위치를 지정할 수 있습니다.

```
In [16]: plt.legend(loc='upper left')
Out[16]: <matplotlib.legend.Legend at 0x91a7c50>

In [17]: plt.show()
```

legend 함수를 호출할 때 loc='upper left'를 사용했으므로 그림 15.9와 같이 범례가 왼쪽 상단에 출력됩니다. 범례의 위치는 표 15.3과 같이 'location string'을 입력하거나 'location string'에 대응되는 'location code'를 사용해도 됩니다. 범례 표시 위치를 직접 지정하지 않고, 'best' 옵션을 사용하면 그래프를 가리지 않도록 적절한 위치에 범례를 자동으로 표시합니다.

표 15.3 legend loc 파라미터 옵션

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

지금까지 작성한 전체 코드는 다음과 같습니다.

```
import pandas_datareader.data as web
import matplotlib.pyplot as plt

lg = web.DataReader("KRX:066570", "google")
samsung = web.DataReader("KRX:005930", "google")

plt.plot(lg.index, lg['Close'], label='LG Electronics')
plt.plot(samsung.index, samsung['Close'], label='Samsung Electronics')

plt.legend(loc='upper left')
plt.show()
```

matplotlib 구성

그림 15.10은 matplotlib의 Figure 객체의 구성을 그림으로 나타낸 것입니다. matplotlib로 그래프를 그리려면 Figure 객체와 하나 이상의 subplot(Axes) 객체가 필요합니다. Axes 객체는 다시 두 개의 Axis 객체를 포함합니다. 여기서 Axis 객체가 y축과 x축을 나타냅니다.

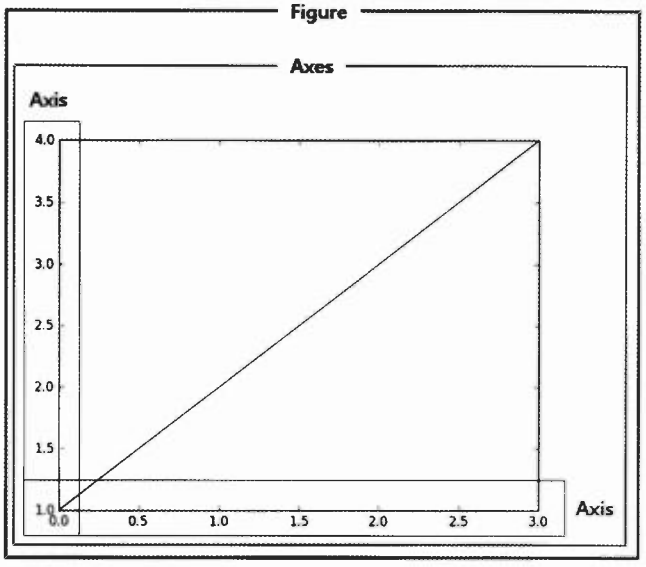


그림 15.10 Figure의 구성

matplotlib에서 그래프는 Figure 객체 내에 존재합니다. 따라서 그래프를 그리려면 다음 코드에서처럼 figure 함수를 사용해 Figure 객체를 생성해야 합니다.

```
In [22]: import matplotlib.pyplot as plt

In [23]: fig = plt.figure()

In [24]: type(fig)
Out[24]: matplotlib.figure.Figure

In [25]:
```

Figure 객체를 생성한 후 plt.show 함수를 호출하면 그림 15.11과 같이 비어있는 Figure 객체가 화면에 출력됩니다. 그러나 Figure 객체 안에는 Axes 객체가 아직 포함되지 않았기 때문에 그래프를 그릴 수는 없습니다.

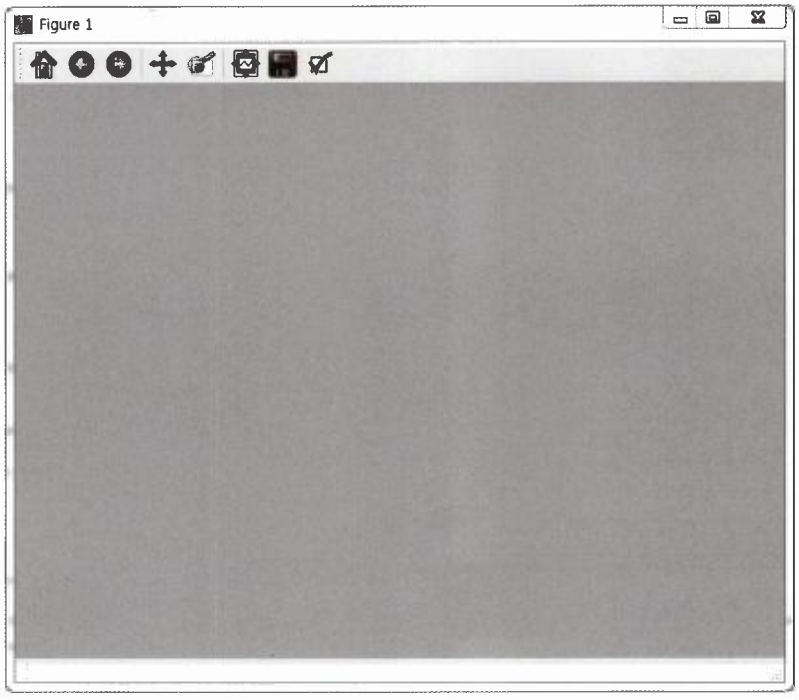


그림 15.11 비어있는 Figure 객체

빈 Figure 객체에 Axes 객체(또는 subplot)를 생성하려면 add_subplot 메서드를 사용하면 됩니다. add_subplot(1, 1, 1)은 Figure 객체에 하나의 AxesSubplot 객체를 생성합니다. 다음 코드에서 ax의 타입을 보면 AxesSubplot인 것을 확인할 수 있습니다.

```
In [35]: fig = plt.figure()

In [36]: ax = fig.add_subplot(1, 1, 1)

In [37]: type(ax)
Out[37]: matplotlib.axes._subplots.AxesSubplot

In [38]: plt.show()
```

plt.show 함수를 호출해보면 이번에는 Figure 객체에 AxesSubplot 객체가 생성된 것을 확인할 수 있습니다.

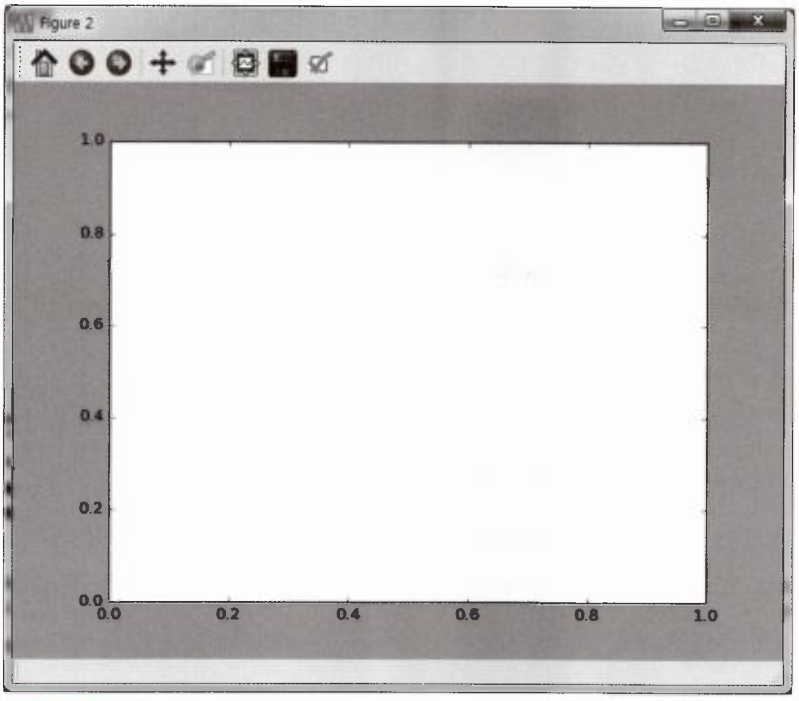


그림 15.12 Figure와 AxesSubplot 객체

Figure 객체를 생성하고 해당 Figure 객체에 여러 개의 AxesSubplot 객체를 생성하는 두 가지 작업을 한번에 하고자 할 때는 다음과 같이 plt.subplots를 사용하면 됩니다. plt.subplots(2, 2)가 호출되면 Figure 객체와 2x2 그리드 형태의 AxesSubplot 객체가 생성됩니다.

```
In [44]: fig, ax_list = plt.subplots(2, 2)

In [45]: fig
Out[45]: <matplotlib.figure.Figure at 0xa8a8730>

In [46]: ax_list
Out[46]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x0A777470>,
<matplotlib.axes._subplots.AxesSubplot object at 0x0A8E08B0>],
[<matplotlib.axes._subplots.AxesSubplot object at 0x0A90AD50>,
<matplotlib.axes._subplots.AxesSubplot object at 0x0A9C1550>]], dtype=object)

In [47]:
```

2x2 구조의 AxesSubplot 객체에 접근하려면 ax_list[0][0], ax_list[0][1], ax_list[1][0], ax_list[1][1]을 통해 각 행과 열에 위치에 존재하는 AxesSubplot 객체에 접근할 수 있습니다.

```
In [48]: ax_list[0][0]
Out[48]: <matplotlib.axes._subplots.AxesSubplot at 0xa777470>

In [49]: ax_list[0][1]
Out[49]: <matplotlib.axes._subplots.AxesSubplot at 0xa8e08b0>

In [50]: ax_list[1][0]
Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0xa90ad50>

In [51]: ax_list[1][1]
Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0xa9c1550>
```

ax_list[0][0]이 바인딩하고 있는 왼쪽 상단의 AxesSubplot 객체에 그래프를 그린 후 그래프를 화면에 출력해 보겠습니다. 다음 코드를 실행하면 그림 15.13과 같이 왼쪽 상단에만 그래프가 출력되는 것을 확인할 수 있습니다.

```
In [56]: ax_list[0][0].plot([1, 2, 3, 4])
Out[56]: [<matplotlib.lines.Line2D at 0xb2b9770>]

In [57]: plt.show()
```

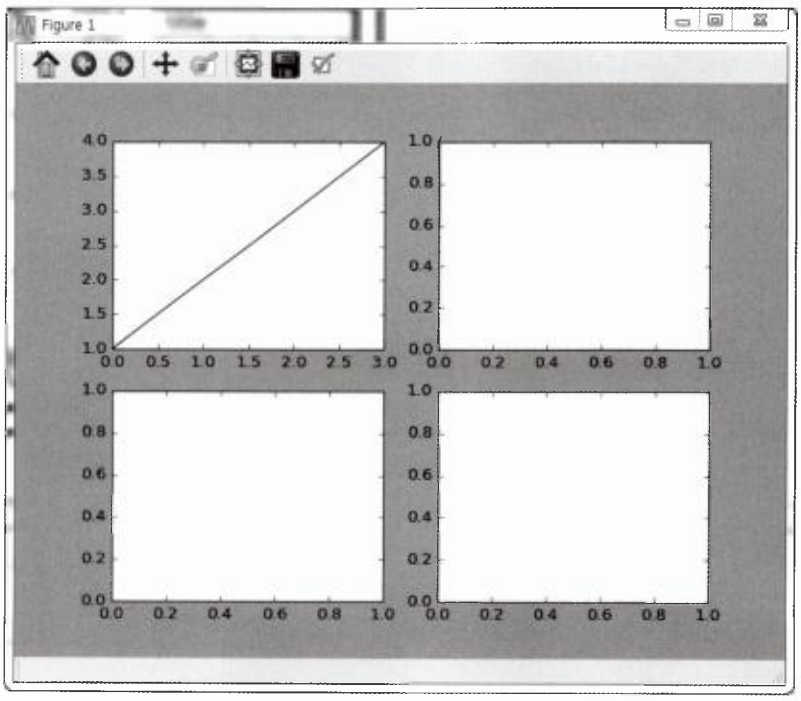


그림 15.13 (0, 0) AxesSubplot에 그래프 출력(소스코드: book/ch15/08.py)

한 가지 재미있는 점은 이전 장에서는 Figure 객체나 AxesSubplot 객체를 명시적으로 생성하지 않고도 plt.plot 함수를 통해 그래프를 그렸었다는 것입니다. 이는 plt.plot 함수가 자동으로 가장 최근에 생성된 Figure 객체를 찾고 해당 Figure 객체 내의 AxesSubplot에 그래프를 그리기 때문입니다. 만약 Figure와 AxesSubplot 객체가 없다면 Figure 객체와 AxesSubplot를 하나 생성합니다. 이 때문에 지금까지 Figure와 AxesSubplot 객체를 명시적으로 만들지 않고도 그래프를 그릴 수 있었던 것입니다.

02 다양한 그래프 그리기

주식 투자를 할 때 아마도 가장 처음 배우는 내용 중 하나가 봉 차트를 읽는 방법일 겁니다. 봉 차트는 일본에서 개발됐는데 그 모양이 양초와 비슷해서 캔들 스틱 차트라고도 합니다. 캔들 스틱 차트는 투자자들이 거래일의 시가, 고가, 저가, 종가를 한눈에 알아볼 수 있기 때문에 주식 투자에서 자주 사용되는 그래프 중 하나입니다. 이번 절에서는 주식 투자에서 많이 활용되는 캔들 스틱 차트, Bar 차트, Pie 차트를 그리는 법을 배워보겠습니다.

수정 종가와 거래량 한번에 그리기

이번 절에서는 그림 15.14와 같이 상단에는 수정 종가 그래프를, 하단에는 거래량 그래프를 그려보겠습니다. 참고로 그림 15.14는 SK 하이닉스 종목의 2010-01-01일 이후의 데이터를 사용해서 그린 것입니다.

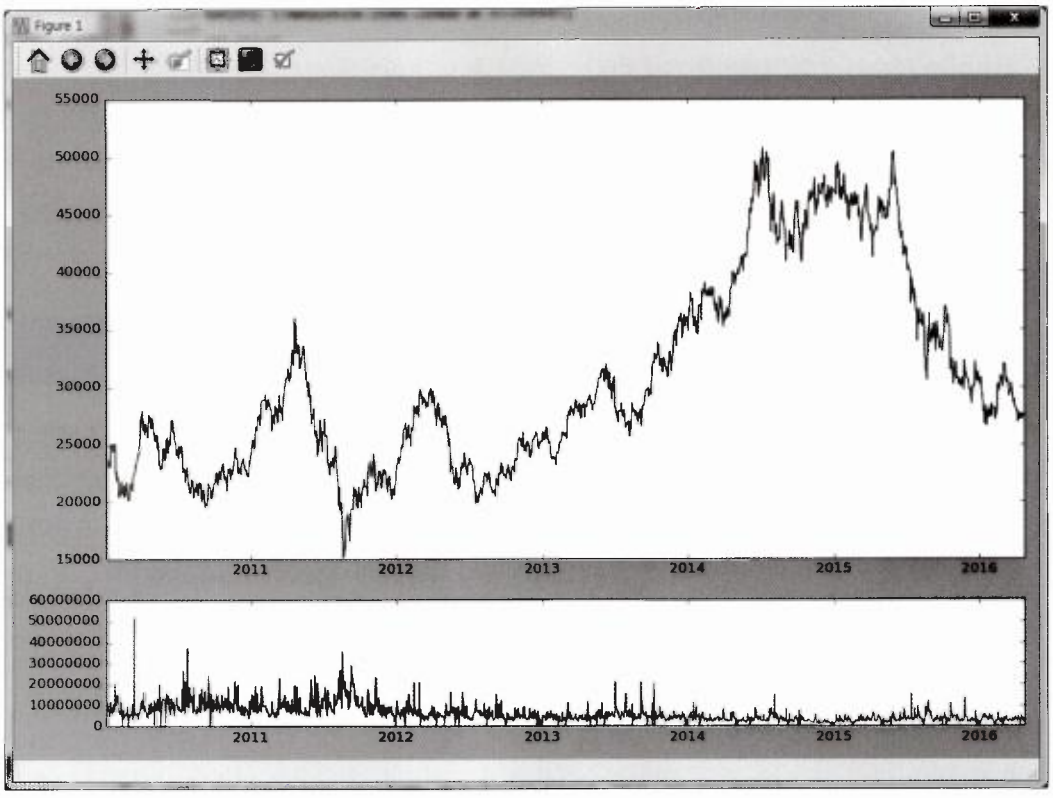


그림 15.14 수정 종가와 거래량을 나타내는 그래프(소스코드: book/ch15/09.py)

그림 15.14를 살펴보면 Figure 객체 안에 수정 종가 그래프를 위한 AxesSubplot과 거래량 그래프를 위한 AxesSubplot이 위치한 것을 볼 수 있습니다. 이미 이전 절에서도 한 Figure 객체 안에 여러 개의 subplot이 존재하는 경우에 대해 배웠습니다. 다만 그림 15.14에서는 위쪽에 위치하는 AxesSubplot이 아래쪽의 AxesSubplot에 비해 더 큰 공간을 차지하고 있다는 점에서 이전과는 조금 다릅니다.

다음 코드는 그림 15.14의 수정 종가와 거래량 그래프를 만드는 코드입니다. SK 하이닉스 데이터는 pandas_datareader.data 모듈을 이용해 구글 파이낸스로부터 가져왔습니다.

그래프를 그리는 데는 matplotlib 라이브러리의 pyplot 모듈을 사용했습니다. Figure 객체와 AxesSubplot 객체를 만든 후 plot 함수를 사용해 그래프를 그리는 기본 틀은 그대로 유지됐습니다. 다만 AxesSubplot 객체를 만들 때 add_subplot이 아닌 subplot2grid 함수를 사용했다는 점이 다릅니다.

```
import matplotlib.pyplot as plt
import pandas_datareader.data as web

sk_hynix = web.DataReader("KRX:000660", "google")

fig = plt.figure(figsize=(12, 8))

top_axes = plt.subplot2grid((4,4), (0,0), rowspan=3, colspan=4)
bottom_axes = plt.subplot2grid((4,4), (3,0), rowspan=1, colspan=4)
bottom_axes.get_yaxis().get_major_formatter().set_scientific(False)

top_axes.plot(sk_hynix.index, sk_hynix['Close'], label='Close')
bottom_axes.plot(sk_hynix.index, sk_hynix['Volume'])

plt.tight_layout()
plt.show()
```

pandas_datareader.data 모듈을 이용해 DataFrame 객체를 구하는 방법은 많이 실습해봤기 때문에 여기서는 따로 다루지 않겠습니다. SK 하이닉스의 데이터를 얻기 위해 "KRX:000660"이라는 종목 코드를 사용했다는 점만 확인하면 됩니다.

전체 코드를 이해하기에 앞서 Figure 객체와 크기가 서로 다른 두 개의 AxesSubplot 객체를 생성하는 코드만 따로 분리해서 실행해보겠습니다. 다음 코드를 실행하면 그림 15.15와 같이 Figure 객체에 크기가 다른 두 개의 AxesSubplot 객체가 생성되는 것을 확인할 수 있습니다.

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12, 8))
top_axes = plt.subplot2grid((4,4), (0,0), rowspan=3, colspan=4)
bottom_axes = plt.subplot2grid((4,4), (3,0), rowspan=1, colspan=4)

plt.show()
```

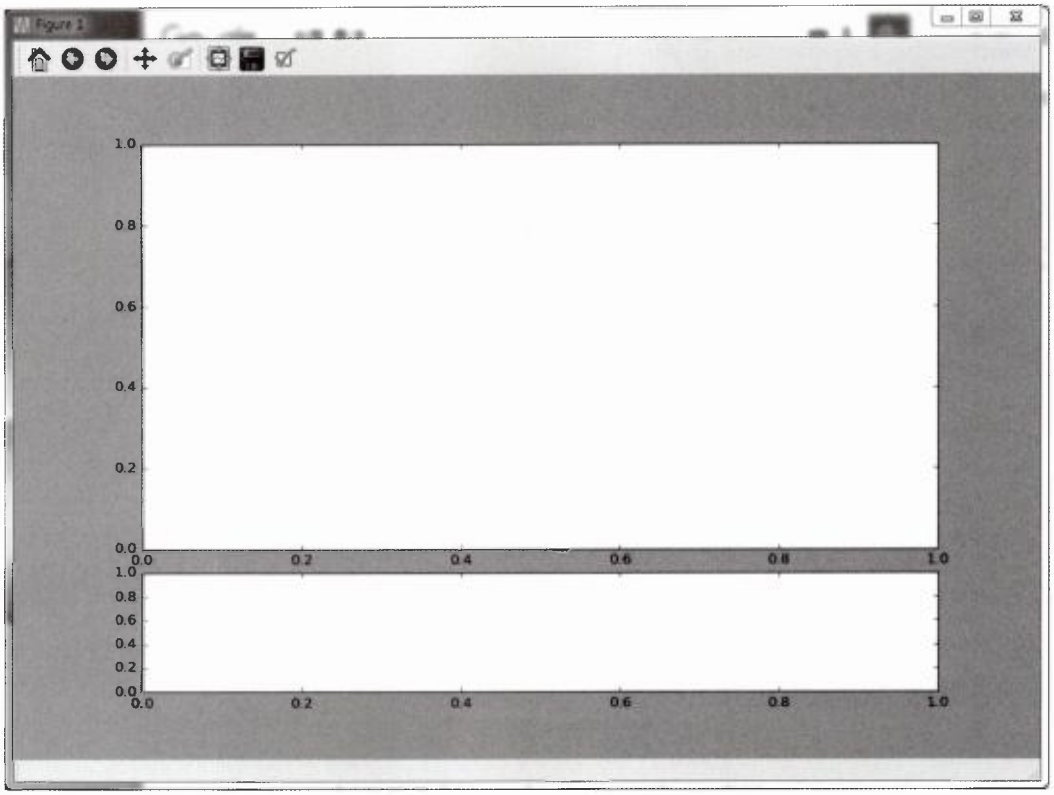


그림 15.15 subplot2grid를 이용해 AxesSubplot 객체를 생성(소스코드: book/ch15/10.py)

Figure 객체를 생성하는 코드를 살펴보면 figsize라는 인자를 통해 Figure 객체의 크기를 조정한 것을 볼 수 있습니다. 일반적으로 주식 데이터를 그래프로 그릴 때는 높이보다 너비를 조금 더 길게 설정하는 것이 보기 편리합니다.

AxesSubplot 객체는 앞서 설명한 add_subplot이나 subplots를 사용하지 않고 subplot2grid를 사용해 생성했습니다. subplot2grid는 subplot의 위치나 크기를 조절할 때 사용합니다.

subplot2grid 함수에서 첫 번째 인자인 (4, 4)는 그림 15.16 (a)와 같이 4x4 grid 모양을 의미합니다. 두 번째 인자인 (0, 0)은 4x4 grid에서 (0, 0)에 위치하는 grid를 의미합니다. 세 번째 인자인 rowspan=3은 (0, 0) grid가 행 방향으로 3개만큼 걸치는 것을 의미합니다. 네 번째 인자인 colspan=4는 (0, 0) grid가 열 방향으로 4개만큼 걸치는 것을 의미합니다. 즉, 그림 15.16 (b)에서 위쪽 박스에 해당하는 AxesSubplot이 생성됩니다.

두 번째로 호출되는 subplot2grid 함수에서 첫 번째 인자는 (4, 4)로 역시 4x4 grid를 의미합니다. 두 번째 인자인 (3, 0)은 4x4 grid에서 (3, 0)에 위치하는 grid를 의미합니다. rowspan=1은 행 방향으로 1개만큼 걸치는 것을, colspan=4는 열 방향으로 4개만큼 걸치는 것을 의미합니다. 즉, 그림 15.16 (b)에서 아래쪽 박스에 해당하는 AxesSubplot이 생성됩니다.

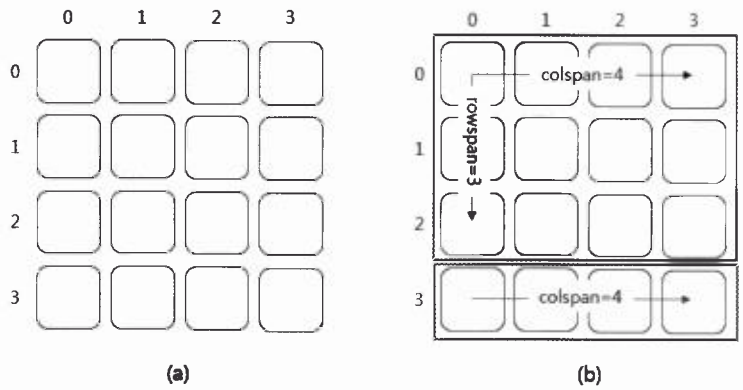


그림 15.16 subplot2grid를 이용한 AxesSubplot 생성

생성된 AxesSubplot에 plot 함수를 통해 그래프를 그릴 수 있습니다. 이때 거래량 그래프가 출력되는 bottom_axes라는 AxesSubplot 객체에는 다음과 같은 설정을 추가했습니다. 이는 거래량 값으로서 큰 값이 발생할 때 그 값을 오일러 상수(e)의 지수 형태로 표현되지 않게 해줍니다.

```
bottom_axes.get_yaxis().get_major_formatter().set_scientific(False)
```

마지막으로 tight_layout 함수는 subplot들이 Figure 객체의 영역 내에서 자동으로 최대 크기로 출력되게 해줍니다. 여백을 두는 것보다 조금 더 그래프를 크게 그리는 편이 해석할 때 편리하기 때문에 자동으로 크기 조절을 해주는 것입니다.

```
plt.tight_layout()
```

캔들 스틱 차트 그리기

이번 절에서는 그림 15.17과 같은 캔들 스틱 차트를 그려보겠습니다. 캔들 스틱 차트는 일본에서 개발된 봉 차트와도 비슷한데 보통은 일본식 봉 차트나 미국에서 사용하는 캔들 스틱 차트나 같은 것이라 보면 됩니다.



그림 15.17 캔들 스틱 차트(봉 차트)

캔들 스틱 차트나 봉 차트를 그리기 전에 먼저 차트를 구성하는 기본 단위인 캔들 스틱 또는 봉의 의미를 간단히 살펴보겠습니다. matplotlib에서 캔들 스틱 차트를 그리는 함수 이름도 candle로 시작하지만 국내에서는 캔들보다는 봉 차트라는 용어를 더 많이 사용합니다.

그림 15.18처럼 봉 차트를 구성하는 기본 단위인 '일봉'은 거래일 동안의 시가(장시작 가격), 고가(장중 최고가), 저가(장중 최저가), 종가(장마감 가격)의 네 가지 값을 한 개의 봉으로 표현합니다.

일봉 중 종가가 시가보다 높은 경우를 양봉이라고 하며, 반대로 종가가 시가보다 낮은 경우를 음봉이라고 합니다. 증권사 HTS마다 다르지만 양봉은 보통 빨간색으로 표시하고 음봉은 파란색으로 표시합니다.

거래일 하루 동안 주가는 쉼 없이 변하지만 봉 차트에서는 시가, 고가, 저가, 종가의 네 가지 값으로 해당 거래일의 추가 추이를 나타냅니다. 예를 들어, 종가와 고가 사이의 간격이 크다는 것은 하루 중에 주가가 크게 올랐다가 결국은 다시 떨어지는 것을 의미합니다. 비슷하게 종가와 저가 사이의 간격이 크다는 것은 장중에 주가가 크게 떨어졌다가 장 종료 시점에는 일정 부분 상승한 채로 장이 종료됐음을 의미합니다.

봉 차트에서는 시가와 종가 사이를 몸통이라고 부르고 몸통 윗부분을 머리, 몸통 아래를 꼬리라고 부릅니다. 봉을 그릴 때 몸통은 두껍게 그리고 고가와 저가를 나타내는 머리나 꼬리는 얇은 실선으로 표시합니다.

많은 투자 관련 책에서 봉 차트를 바탕으로 주가가 약세시점에서 양봉 3개가 나타나는 것을 '적삼병'이라고 부르고 이를 주가 상승 장세의 시점으로 봅니다. 반대로 주가가 상승 시점에 있다가 음봉 3개가 연달아서 나타나면 이를 '흑삼병'이라고 부르고 주가가 하락세로 전환되거나 하락세가 지속될 시점으로 판단합니다.

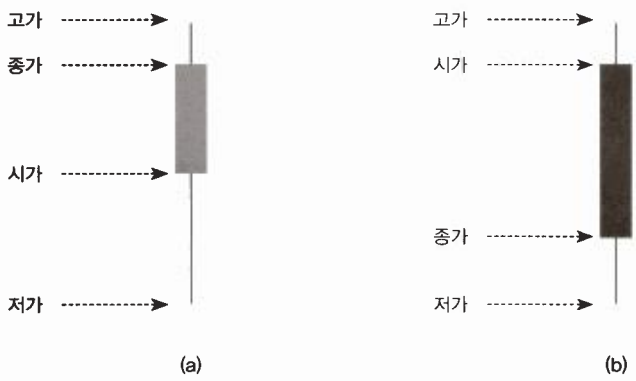


그림 15.18 봉 차트의 양봉(a)과 음봉(b)

봉 차트와 봉의 의미에 대해 간단히 공부했으니 이번에는 파이썬으로 봉 차트를 그려보겠습니다. 파이썬에서 봉 차트를 그리려면 역시 matplotlib 모듈을 사용하면 됩니다. 참고로 matplotlib에서는 봉 차트를 캔들 스틱 차트라고 부릅니다.

봉 차트를 그리려면 한 달 정도의 데이터가 필요합니다. 너무 많은 기간의 데이터는 그래프로 그린 후 보기가 어렵기 때문에 본 예제에서는 딱 한 달 정도의 값만 사용하겠습니다. 이번에 봉 차트로 그려볼 종목은 'SK 하이닉스'입니다. pandas_datareader.data 모듈을 사용해 2016년 3월 한 달간의 주가를 다운로드합니다.

```
In [1]: import pandas_datareader.data as web

In [2]: import datetime

In [3]: start = datetime.datetime(2016, 3, 1)

In [4]: end = datetime.datetime(2016, 3, 31)

In [5]: skhynix = web.DataReader("KRX:000660", "google", start, end)
```

코드가 에러 없이 실행됐다면 skhynix라는 변수는 DataFrame 객체를 바인딩할 것입니다. 데이터를 정상적으로 얻어왔는지 확인해 보겠습니다.

```
In [6]: skhynix.head()
Out[6]:
              Open     High      Low    Close   Volume
Date
2016-03-02  31050.0  31300.0  30650.0  30950.0  3533588
2016-03-03  31550.0  32000.0  31100.0  31950.0  3790108
2016-03-04  31800.0  32100.0  31550.0  31900.0  1905325
2016-03-07  32150.0  32250.0  31500.0  31900.0  3437058
2016-03-08  31200.0  31500.0  30300.0  30500.0  4583668

In [7]:
```

앞서 설명한 것처럼 봉을 구성하려면 시가(Open), 고가(High), 저가(Low), 종가(Close)가 필요한데 DataFrame에는 이미 해당 데이터로 구성된 칼럼이 존재합니다. 그래프를 그릴 데이터가 준비됐기 때문에 여러분은 이를 봉 차트로 그리기만 하면 됩니다.

봉 차트는 matplotlib.finance 모듈의 candlestick2_ohlc 함수를 이용하면 바로 그릴 수 있습니다. 해당 모듈을 사용하기 위해 먼저 모듈을 임포트합니다.

```
In [8]: import matplotlib.pyplot as plt

In [9]: import matplotlib.finance as matfin
```

Figure 객체와 AxesSubplot 객체를 만든 후 candlestick2_ohlc 함수를 호출합니다. matplotlib.finance 모듈의 candlestick2_ohlc 함수 호출 부분을 살펴보면 첫 번째 인자로 AxesSubplot 객체가 사용됩니다. 두 번째부터 다섯 번째 인자로는 함수의 이름에서 알 수 있듯이 시가, 고가, 저가, 종가가 사용됩니다. width는 봉의 몸통의 너비를 조절하는 파라미터이고 colorup은 양봉을 표현하는 색상을, colordown은 음봉을 표현하는 색상을 지정하는 데 사용하는 파라미터입니다.

```
In [10]: fig = plt.figure(figsize=(12, 8))

In [11]: ax = fig.add_subplot(111)

In [12]: matfin.candlestick2_ohlc(ax, skhynix['Open'], skhynix['High'], skhynix['Low'], skhy-
nix['Close'], width=0.5, colorup='r', colordown='b')

In [13]: plt.show()
```

그림 15.19와 같이 봉 차트가 그려지지만 봉 차트 그래프의 x 축에 날짜가 아니라 0부터 시작하는 정숫값이 출력된 것을 확인할 수 있습니다. 따라서 봉 차트는 그렸지만 각 봉에 해당하는 거래일 정보는 얻을 수 없습니다.

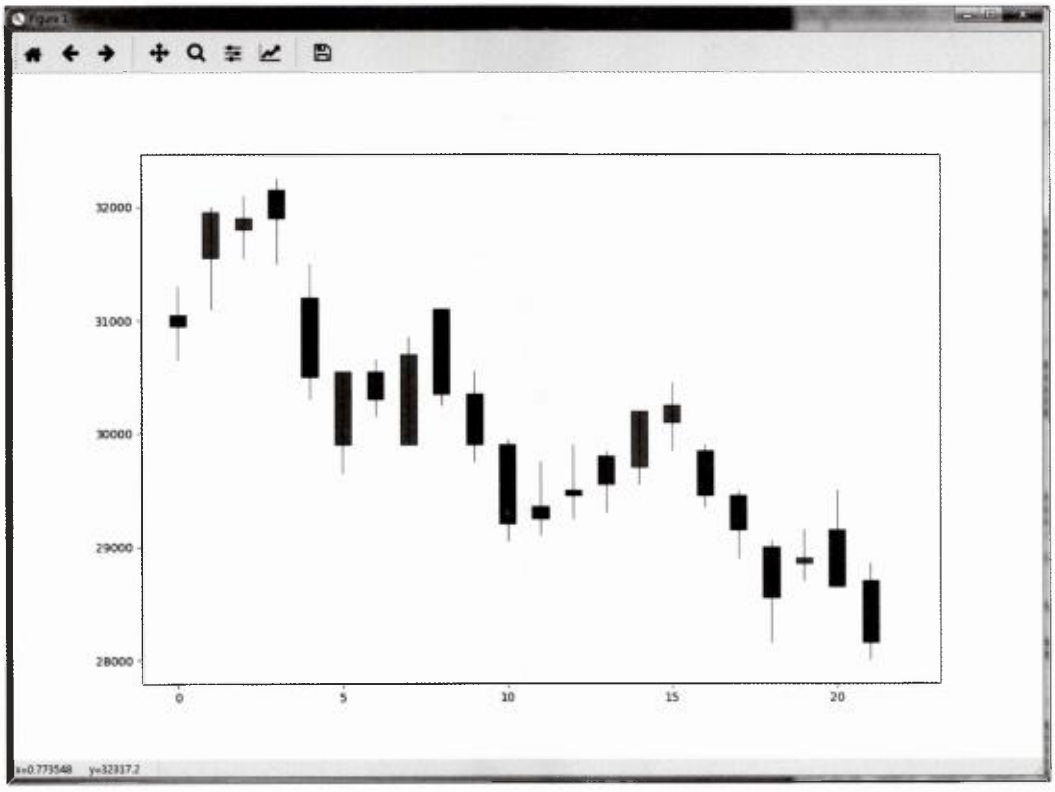


그림 15.19 봉차트 (candlestick2_ohlc 함수 사용)

지금까지 작성한 전체 코드는 예제 15.1과 같습니다. 다음 코드를 앞에서처럼 Jupyter QtConsole에서 입력하거나 PyCharm을 통해 실행하면 됩니다.

예제 15.1 candlestick2_ohlc를 이용한 봉 차트 그리기-1(소스코드: book/ch15/11.py)

```
import pandas_datareader.data as web
import datetime
import matplotlib.pyplot as plt
import matplotlib.finance as matfin
```

```
start = datetime.datetime(2016, 3, 1)
end = datetime.datetime(2016, 3, 31)
skhynix = web.DataReader("KRX:000660", "google", start, end)

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)

matfin.candlestick2_ohlc(ax, skhynix['Open'], skhynix['High'], skhynix['Low'], skhynix['Close'],
width=0.5, colorup='r', colordown='b')
plt.show()
```

이번에는 그림 15.19에 있는 봉 차트의 문제점을 해결해 보겠습니다. 구글 파이낸스로부터 데이터를 받는 경우 장이 열리지 않았던 날은 데이터가 없습니다. 그러나 야후 파이낸스와 같은 일부 사이트에는 2016-03-01(공휴일)에도 데이터가 존재하는데 이 경우 거래량은 0의 값을 가집니다. 만약 웹 사이트에서 받은 데이터가 이와 같다면 다음과 같이 거래량이 0이라는 조건을 사용해 쉽게 해당 로우를 선택하지 않을 수 있습니다.

```
In [24]: skhynix = skhynix[skhynix['Volume'] > 0]
```

수정된 코드를 포함한 전체 코드는 예제 15.2와 같습니다.

예제 15.2 candlestick2_ohlc를 이용한 봉 차트 그리기-2(소스코드: book/ch15/12.py)

```
import pandas_datareader.data as web
import datetime
import matplotlib.pyplot as plt
import matplotlib.finance as matfin

start = datetime.datetime(2016, 3, 1)
end = datetime.datetime(2016, 3, 31)
skhynix = web.DataReader("KRX:000660", "google", start, end)
skhynix = skhynix[skhynix['Volume'] > 0]

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)

matfin.candlestick2_ohlc(ax, skhynix['Open'], skhynix['High'], skhynix['Low'], skhynix['Close'],
width=0.5, colorup='r', colordown='b')
plt.show()
```

이번에는 그래프의 x축에 정수 인덱스가 아닌 날짜가 출력되도록 변경해 봅시다. 사실 DataReader 함수를 통해 얻어온 DataFrame 객체에는 이미 날짜 정보가 저장돼 있습니다. 다음 코드를 Jupyter QtConsole에서 실행해보기 바랍니다.

```
In [1]: import pandas_datareader.data as web

In [2]: import datetime

In [3]: start = datetime.datetime(2016, 3, 1)

In [4]: end = datetime.datetime(2016, 3, 31)

In [5]: skhynix = web.DataReader("KRX:000660", "google", start, end)

In [6]: skhynix.index
Out[6]:
DatetimeIndex(['2016-03-02', '2016-03-03', '2016-03-04', '2016-03-07',
               '2016-03-08', '2016-03-09', '2016-03-10', '2016-03-11',
               '2016-03-14', '2016-03-15', '2016-03-16', '2016-03-17',
               '2016-03-18', '2016-03-21', '2016-03-22', '2016-03-23',
               '2016-03-24', '2016-03-25', '2016-03-28', '2016-03-29',
               '2016-03-30', '2016-03-31'],
              dtype='datetime64[ns]', name='Date', freq=None)
```

skhynix.index[0]은 '2016-03-02 00:00:00'을 의미합니다. 객체 타입은 pandas의 Timestamp입니다. 문제는 Timestamp 객체의 '연도-월-일 시:분:초'를 그래프의 x축에 출력하기에는 공간이 부족하다는 점입니다. 따라서 출력에 필요한 최소 부분만 뽑아내는 작업이 필요합니다.

```
In [8]: skhynix.index[0]
Out[8]: Timestamp('2016-03-02 00:00:00')

In [9]: type(skhynix.index[0])
Out[9]: pandas.tslib.Timestamp
```

pandas의 Timestamp 객체는 strftime 메서드에 출력 포맷을 전달해서 문자열로 변환할 수 있습니다. 예를 들어, '%d'를 사용하면 날짜만 문자열로 변환됩니다. '%Y-%m-%d'를 사용하면 '연도-월-일'이 문자열로 변환됩니다.

```
In [11]: skhynix.index[0].strftime('%d')
Out[11]: '02'

In [12]: skhynix.index[0].strftime('%Y-%m-%d')
Out[12]: '2016-03-02'
```

앞서 언급한 것처럼 그래프의 x축에 긴 문자열을 출력하기에는 공간이 부족합니다. 따라서 연도, 월, 시간 등은 제외하고 '일자'만 문자열로 만들어서 x축에 출력하겠습니다. 이를 위해 먼저 name_list라는 비어있는 리스트를 만들고, 반복문을 사용해 각 날짜에서 '일자'만 문자열로 변환한 후 변환된 문자열을 리스트에 추가합니다.

```
In [13]: name_list = []

In [14]: for day in skhynix.index:
    ...:     name_list.append(day.strftime('%d'))
    ...:

In [15]: name_list
Out[15]:
['02',
'03',
'04',
'05',
#중간 생략
'31']

In [16]:
```

matplotlib에서 x축과 y축에 표시되는 값을 ticker라고 부릅니다. ticker를 설정하려면 ticker의 위치와 각 위치에서 출력될 값이 필요합니다. 앞서 name_list에 출력될 값은 준비해 뒀기 때문에 이번에는 위치 값을 생성해 보겠습니다.

여러분이 그릴 그래프에서 x축은 사실 날짜를 의미합니다. 어차피 날짜 값은 준비돼 있으므로 위치에는 일정하게 증가하는 정숫값을 사용하면 됩니다. 따라서 다음과 같이 range 내장 함수를 사용해 위치 값을 생성합니다. range(0, 23)은 0~22까지의 정숫값과 같습니다.

```
In [16]: day_list = range(len(skhynix))

In [17]: day_list
Out[17]: range(0, 23)
```

x축 ticker의 위치와 각 위치에서 출력될 값을 모두 준비했습니다. 이제 x축의 ticker 위치와 출력 값을 설정하면 됩니다. 위치를 설정하는 함수는 set_major_locator이고, 출력되는 값을 설정하는 함수는 set_major_formatter입니다.

먼저 matplotlib.ticker 모듈을 임포트한 후 x축에 대해서만 설정해 봅시다. set_major_locator나 set_major_formatter 함수에 인자를 전달할 때 ticker.FixedLocator와 ticker.FixedFormatter를 사용했습니다. 각각 고정 위치와 고정된 포매팅이라는 것을 의미합니다.

```
import matplotlib.ticker as ticker

ax.xaxis.set_major_locator(ticker.FixedLocator(day_list))
ax.xaxis.set_major_formatter(ticker.FixedFormatter(name_list))
```

지금까지 설명한 전체 코드는 예제 15.3과 같습니다.

예제 15.3 candlestick2_ohlc를 이용한 봉 차트 그리기-3(소스코드: book/ch15/13.py)

```
import pandas_datareader.data as web
import datetime
import matplotlib.pyplot as plt
import matplotlib.finance as matfin
import matplotlib.ticker as ticker

start = datetime.datetime(2016, 3, 1)
end = datetime.datetime(2016, 3, 31)
skhynix = web.DataReader("KRX:000660", "google", start, end)

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)

day_list = range(len(skhynix))
name_list = []
for day in skhynix.index:
    name_list.append(day.strftime('%d'))

ax.xaxis.set_major_locator(ticker.FixedLocator(day_list))
ax.xaxis.set_major_formatter(ticker.FixedFormatter(name_list))
```

```
matfin.candlestick2_ohlc(ax, skhynix['Open'], skhynix['High'], skhynix['Low'], skhynix['Close'],
width=0.5, colorup='r', colordown='b')
plt.show()
```

예제 15.3을 실행하면 그림 15.20과 같은 그래프가 출력됩니다. 그림 15.20에서 x축을 살펴보면 ticker 부분이 정수 인덱스가 아니라 각 데이터에 대응되는 날짜가 잘 출력됨을 확인할 수 있습니다.

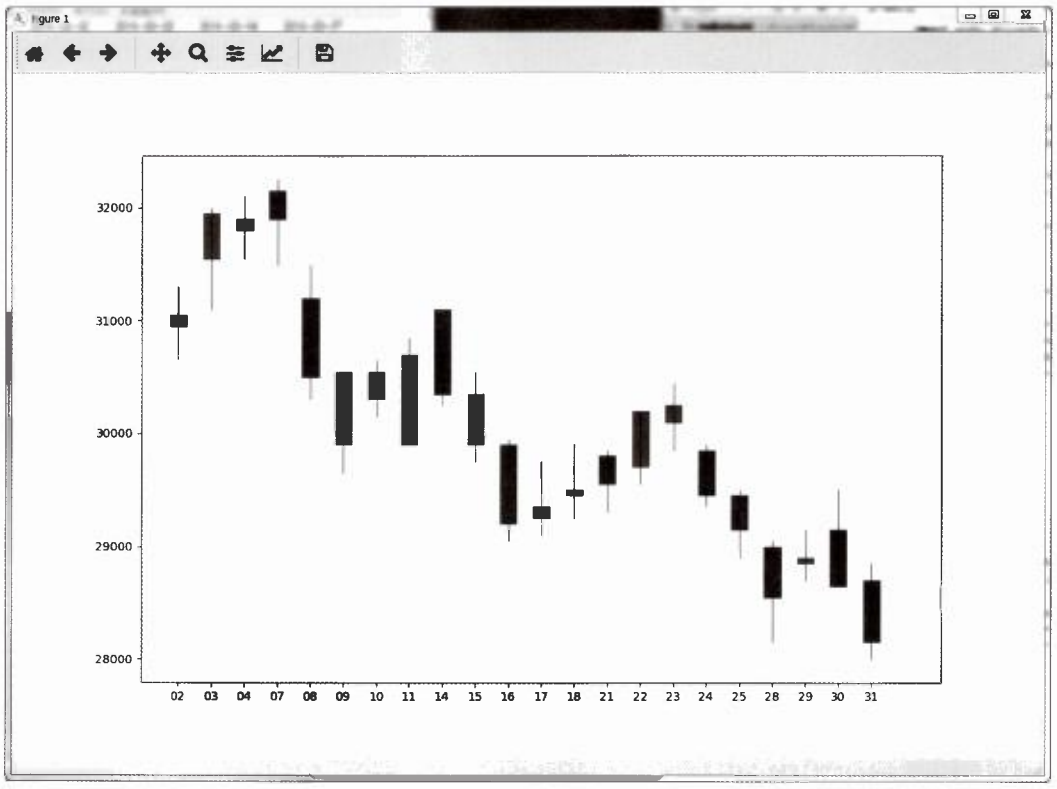


그림 15.20 봉차트-3(candlestick2_ohlc 함수 사용)

이번에는 x축에 각 날짜를 모두 표시하지 않고 매주 월요일에만 '연도-월-일' 포맷으로 출력해 보겠습니다. 기본적인 방식은 예제 15.3과 같습니다. set_major_locator 함수의 인자로 월요일에 해당하는 데이터의 인덱스 값을 넘겨줍니다.

예를 들어, 2016년 03월에는 7, 14, 21, 28일이 월요일입니다. skhynix라는 DataFrame 객체에는 2016-03-01일부터 데이터가 존재하므로 2016-03-07일 데이터에 대한 인덱스는 4가 됩니다(인덱스는 0부터 시작하며 2016-03-05일과 2016-03-06일은 장이 열리지 않았습니다).

그러나 각 날짜에 대해 인덱스를 일일이 계산하는 것은 쉽지 않습니다. 다행히 Timestamp 객체에는 'dayofweek'라는 속성이 있는데 이 속성을 사용하면 거래일의 요일을 쉽게 확인할 수 있습니다. 즉, 다음과 같은 코드를 사용해 거래일의 요일이 월요일인 경우에 대해 데이터의 인덱스와 문자열로 변환한 날짜 정보를 리스트로 각각 구성할 수 있습니다.

```
In [33]: day_list = []

In [34]: name_list = []

In [38]: for i, day in enumerate(skhynix.index):
    ...:     if day.dayofweek == 0:
    ...:         day_list.append(i)
    ...:         name_list.append(day.strftime('%Y-%m-%d') + '(Mon)')
In [36]: day_list
Out[36]: [4, 9, 14, 19]

In [37]: name_list
Out[37]: ['2016-03-07(Mon)', '2016-03-14(Mon)', '2016-03-21(Mon)', '2016-03-28(Mon)']

In [38]:
```

거래일이 월요일인 날에 대한 day_list와 name_list를 새로 생성했다면 그래프를 다시 그려보기 바랍니다. 그래프 해석을 좀 더 편하게 하기 위해 plt.grid 함수를 호출하면 그림 15.21과 같이 그리드가 추가로 표시됩니다. 그림 15.21에서 x축의 ticker를 살펴보면 월요일에 해당하는 데이터에 날짜 정보가 출력됨을 확인할 수 있습니다.

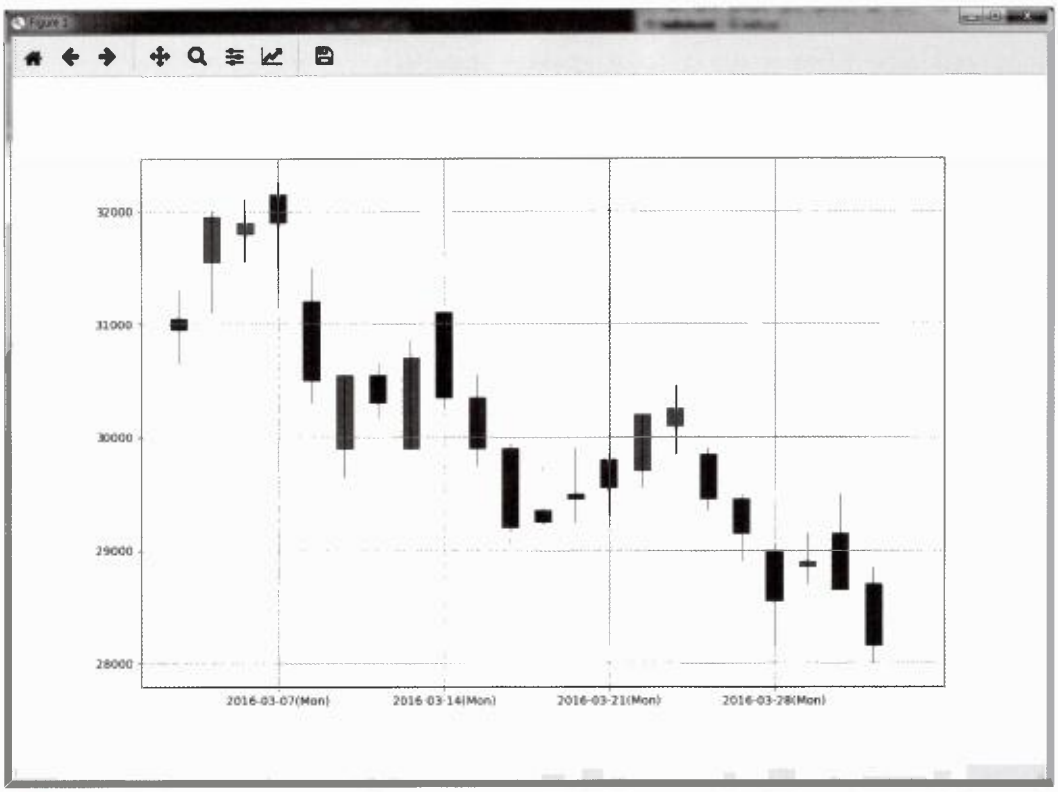


그림 15.21 봉차트-4(candlestick2_ohlc 함수 사용)

그림 15.21에 사용된 전체 코드는 예제 15.4와 같습니다.

예제 15.4 candlestick2_ohlc를 이용한 봉 차트 그리기-4(소스코드: book/ch15/14.py)

```
import pandas_datareader.data as web
import datetime
import matplotlib.pyplot as plt
import matplotlib.finance as matfin
import matplotlib.ticker as ticker

start = datetime.datetime(2016, 3, 1)
end = datetime.datetime(2016, 3, 31)
skhynix = web.DataReader("KRX:000660", "google", start, end)
```

```
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)

day_list = []
name_list = []
for i, day in enumerate(skhynix.index):
    if day.dayofweek == 0:
        day_list.append(i)
        name_list.append(day.strftime('%Y-%m-%d') + '(Mon)')

ax.xaxis.set_major_locator(ticker.FixedLocator(day_list))
ax.xaxis.set_major_formatter(ticker.FixedFormatter(name_list))

matfin.candlestick2_ohlc(ax, skhynix['Open'], skhynix['High'], skhynix['Low'], skhynix['Close'],
width=0.5, colorup='r', colordown='b')
plt.grid()
plt.show()
```

bar 차트 그리기

이번 절에서는 그림 15.22와 같은 bar 차트를 만들어 보겠습니다. bar 차트는 우리말로 막대 그래프라고 합니다. bar 차트는 데이터를 분류한 후 각 데이터를 비교하는 데 많이 사용하는 그래프입니다.

그림 15.22는 특정 거래일의 유가증권시장 등락률 상위 10개 업종에 대한 업종명과 등락률을 bar 차트로 그린 것입니다. 확실히 bar 차트를 사용하니 각 데이터를 쉽게 비교할 수 있음을 확인할 수 있습니다.

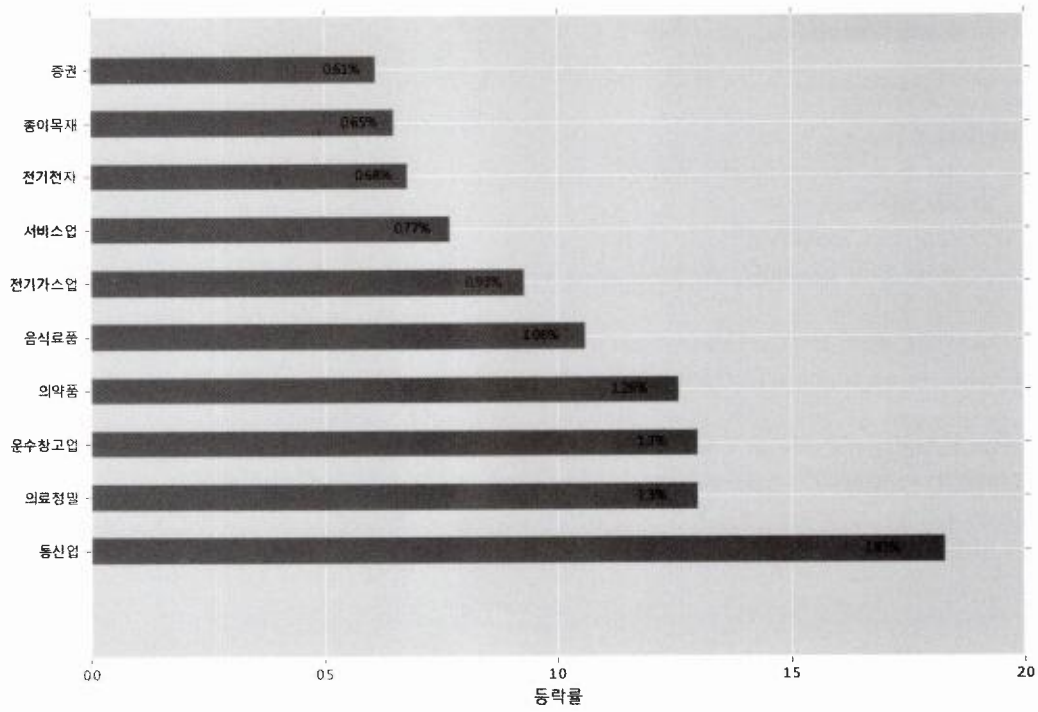


그림 15.22 matplotlib를 이용해서 그린 bar 차트

bar 차트를 그리기 위해 먼저 필요한 모듈을 임포트합니다. matplotlib.pyplot과 numpy 모듈은 기존에도 자주 사용했지만 font_manager와 rc 모듈은 처음으로 임포트하는 모듈입니다. 해당 모듈은 그래프를 그릴 때 한글 폰트를 설정하는 데 사용합니다.

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import font_manager, rc
```

그림 15.22를 참조하면 업종명이 한글로 출력됨을 확인할 수 있습니다. matplotlib는 한글 폰트를 설정하지 않으면 기본적으로 한글이 제대로 출력되지 않습니다. 따라서 데이터에 한글이 포함돼 있다면 다음과 같이 사용할 한글 폰트를 설정해야 한글이 정상적으로 출력됩니다.

```
font_name = font_manager.FontProperties(fname="c:/Windows/Fonts/malgun.ttf").get_name()
rc('font', family=font_name)
```

폰트를 설정하려면 폰트 이름을 알아야 합니다. 윈도우 7부터는 '맑은 고딕'이라는 폰트가 기본 폰트로 사용되고 있기 때문에 이번 예제에서는 'c:/Windows/Fonts/malgun.ttf'라는 경로를 사용했습니다. 폰트의 설치 경로로부터 폰트 이름을 먼저 알아낸 후 rc 함수를 통해 폰트를 설정했습니다.

한글 폰트를 설정하고 나면 이제부터는 bar 차트를 그리기 위한 데이터를 준비해야 합니다. 그림 15.22에서는 bar 차트를 수평 방향으로 그렸지만 수직 방향으로도 그릴 수 있습니다. 일단 bar 차트의 방향과는 상관 없이 한쪽 축에 표시될 label 값과 각 label에 해당하는 수치가 필요합니다.

예를 들어, 그림 15.22에서 label들은 업종명이 될 것이고 수치는 각 업종의 등락률이 됩니다. 저자는 다음 금융 웹 페이지(http://finance.daum.net/quote/upjong.daum)를 참조해서 다음과 같이 데이터를 구성했습니다.

```
industry = ['통신업', '의료정밀', '운수창고업', '의약품', '음식료품', '전기가스업', '서비스업',
'전기전자', '종이목재', '증권']
fluctuations = [1.83, 1.30, 1.30, 1.26, 1.06, 0.93, 0.77, 0.68, 0.65, 0.61]
```

데이터까지 준비됐으므로 이제 그래프를 그리면 됩니다. 그래프를 그리려면 Figure 객체와 AxesSubplot 객체가 필요합니다.

```
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)
```

수평 방향의 bar 차트는 matplotlib.pyplot 모듈의 barh 함수를 사용해 그릴 수 있습니다. barh 함수의 첫 번째 인자는 각 bar가 그려질 위치이고, 두 번째 인자는 각 bar에 대한 수치입니다. 이 값들은 파이썬 리스트 형태로 전달하면 됩니다. align은 bar 차트에서 bar의 정렬 위치를 설정하고 height는 수평 bar 차트의 높이를 설정합니다.

그림 15.23과 같이 수평 방향의 bar 차트에서는 y축에 ticker를 표시하면 됩니다. matplotlib.pyplot 모듈의 yticks 함수를 사용해 ticker의 위치와 각 위치에서의 label을 설정합니다.

```
ypos = np.arange(10)
rects = plt.barh(ypos, fluctuations, align='center', height=0.5)
plt.yticks(ypos, industry)
```

각 수치의 의미를 알려주기 위해 label을 설정한 후 마지막으로 그래프를 화면에 출력합니다.

```
plt.xlabel('등락률')
plt.show()
```

지금까지 설명한 코드를 실행하면 그림 15.23과 같은 모양의 bar 차트가 출력됩니다. 그림 15.22와 비교해보면 그래프의 스타일이 조금 다르며 각 bar의 끝 부분에 수치가 표시되지 않았음을 확인할 수 있습니다.

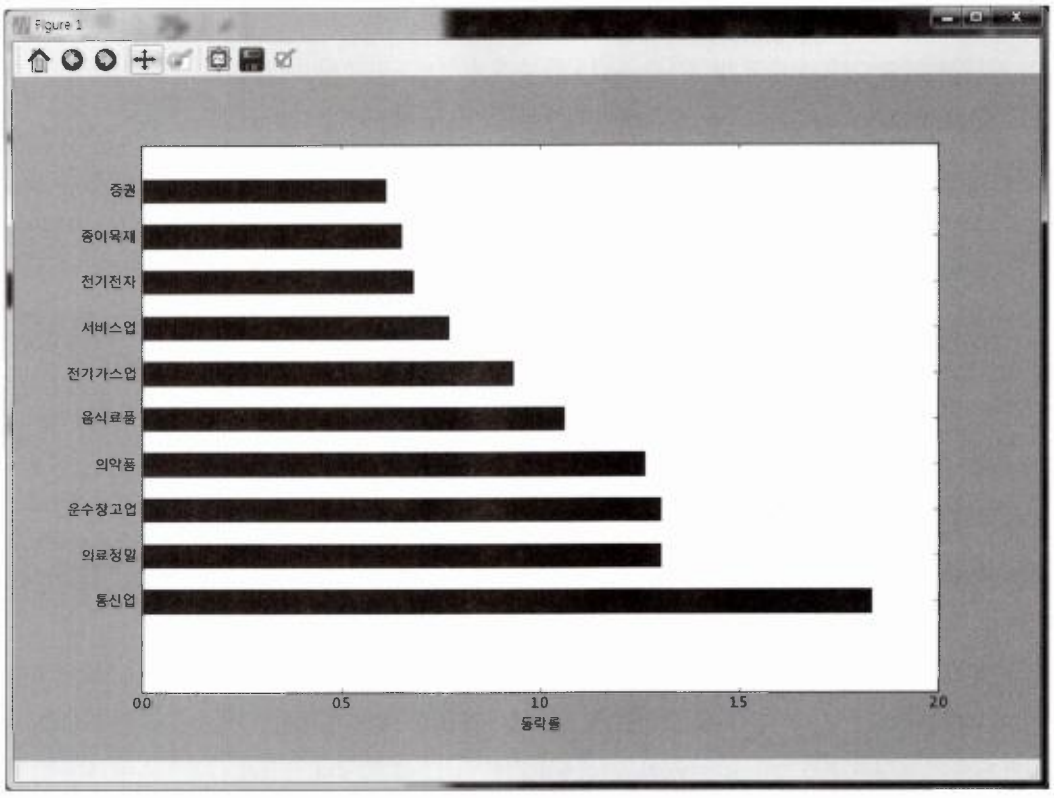


그림 15.23 barh를 통해 그린 bar차트

그래프의 출력 스타일과 수치 값이 bar 차트에 출력되도록 코드를 업데이트해보겠습니다. 먼저 스타일을 변경하려면 matplotlib.style이라는 모듈을 임포트한 후 style을 설정합니다. 참고로 그래프의 style을 변경하는 코드는 그래프를 출력(plt.show 함수 호출)하기 전에 수행해야 합니다.

```
from matplotlib import style

style.use('ggplot')
```

이번에는 bar 차트의 각 bar에 등락률 데이터를 출력해 보겠습니다. 이를 위해 text라는 함수를 사용했습니다. text 함수의 인자를 살펴보면 첫 번째는 text가 출력되는 x축 위치를 의미하고 두 번째는 text가 출력될 y축 위치를 의미합니다. 세 번째는 실제로 표시될 값을 전달하면 됩니다. ha는 수평 방향으로의 정렬을 의미하고 va는 수직 방향으로의 정렬을 의미합니다.

첫 번째 인자를 보면 rect.get_width를 사용하는데, 여기서 rect는 bar 차트에서 각 bar에 해당합니다. 즉, 먼저 각 bar의 너비(길이)를 알아낸 후 그 너비의 95% 지점이 텍스트가 출력될 x축 위치가 됩

니다. 두 번째 인자를 살펴보면 먼저 bar가 출력된 y축 위치를 rect.get_y를 통해 얻은 후 bar 높이의 절반을 더함으로써 y축 위치를 계산합니다. 코드가 잘 이해되지 않는다면 0.95나 2.0을 다른 값으로 변경해보면서 텍스트의 출력 위치를 확인해보기 바랍니다.

```
for i, rect in enumerate(rects):
    ax.text(0.95 * rect.get_width(), rect.get_y() + rect.get_height() / 2.0, str(fluctua-
tions[i]) + '%', ha='right', va='center')
```

예제 15.5는 지금까지 설명한 bar 차트 그리기의 전체 코드입니다. 코드를 보면서 bar 차트를 그리는 전체 맥락을 한 번 더 살펴보기 바랍니다.

예제 15.5 barh를 사용한 bar 차트(소스코드: book/ch15/15.py)

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import font_manager, rc
from matplotlib import style

font_name = font_manager.FontProperties(fname="c:/Windows/Fonts/malgun.ttf").get_name()
rc('font', family=font_name)
style.use('ggplot')

industry = ['통신업', '의료정밀', '운수창고업', '의약품', '음식료품', '전기가스업', '서비스업',
'전기전자', '종이목재', '증권']
fluctuations = [1.83, 1.30, 1.30, 1.26, 1.06, 0.93, 0.77, 0.68, 0.65, 0.61]

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)

ypos = np.arange(10)
rects = plt.barh(ypos, fluctuations, align='center', height=0.5)
plt.yticks(ypos, industry)

for i, rect in enumerate(rects):
    ax.text(0.95 * rect.get_width(), rect.get_y() + rect.get_height() / 2.0, str(fluctua-
tions[i]) + '%', ha='right', va='center')

plt.xlabel('등락률')
plt.show()
```

수평 방향 bar 차트를 barh 함수를 통해 쉽게 그릴 수 있는 것처럼 수직 방향 bar 차트는 matplotlib.pyplot 모듈의 bar 함수를 사용하면 됩니다. 다음 코드는 예제 15.5에서 수평 방향 차트를 수직 방향으로 그리기 위해 변경이 필요한 코드입니다.

```
pos = np.arange(10)
rects = plt.bar(pos, fluctuations, align='center', width=0.5)
plt.xticks(pos, industry)

for i, rect in enumerate(rects):
    ax.text(rect.get_x() + rect.get_width() / 2.0, 0.95 * rect.get_height(), str(fluctuations[i]) + '%', ha='center')

plt.ylabel('등락률')
```

예제 15.5에서 위의 코드 부분만 변경한 후 실행해 보면 그림 15.24와 같이 수직 방향으로 bar 차트가 출력됨을 확인할 수 있습니다.

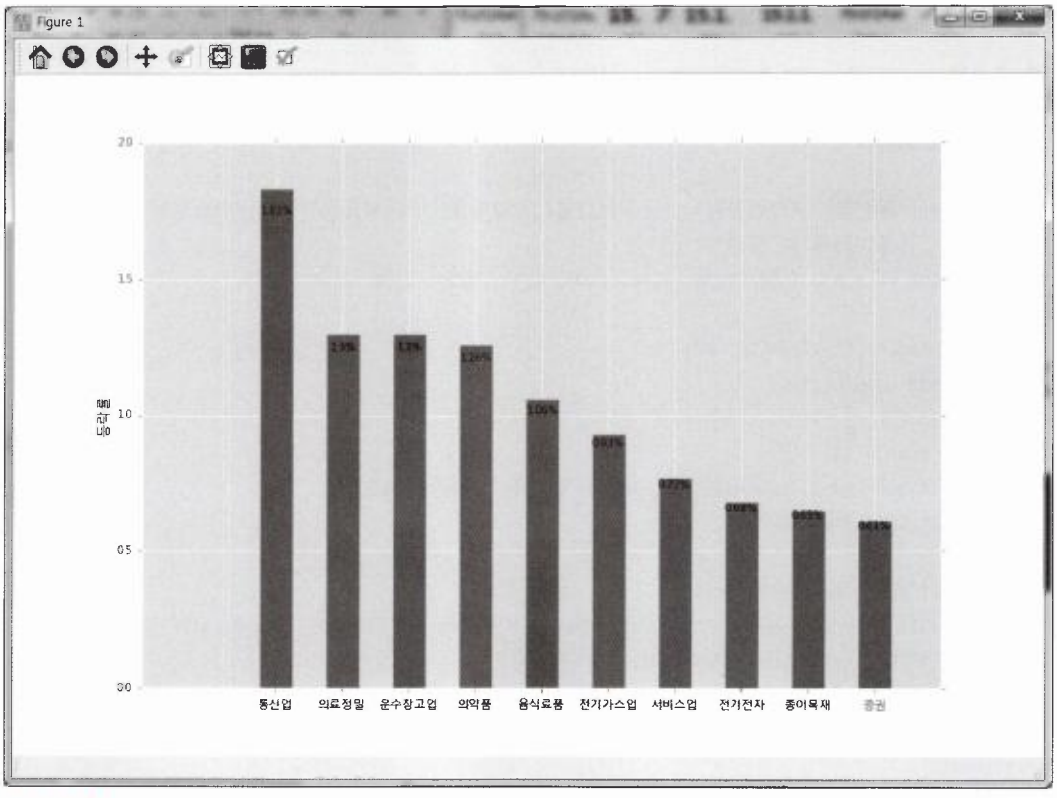


그림 15.24 bar를 통해 그린 수직 방향 bar차트(소스코드: book/ch15/16.py)

pie 차트 그리기

이번 절에서는 그림 15.25와 같은 형태의 그래프를 그려보겠습니다. 이런 형태의 그래프를 pie 차트 또는 원형 차트라고 부릅니다. 참고로 차트의 모양이 먹는 파이와 비슷하게 생겼기 때문에 pie 차트(파이 차트)라고 부릅니다. pie 차트는 각 범주가 데이터에서 차지하는 비율을 나타내는 데 자주 사용됩니다.

그림 15.25는 저자의 가상 포트폴리오를 pie 차트로 그린 것입니다. 삼성전자 주식이 약 50%를 차지하고 SK하이닉스는 20%, LG전자, 네이버, 카카오가 각각 10%를 차지합니다. pie 차트는 각 범주의 상대적 크기를 통해 각 범주가 데이터에서 차지하는 비율을 한눈에 확인할 수 있기 때문에 다양한 형태의 데이터를 시각화하는 데 효과적입니다.

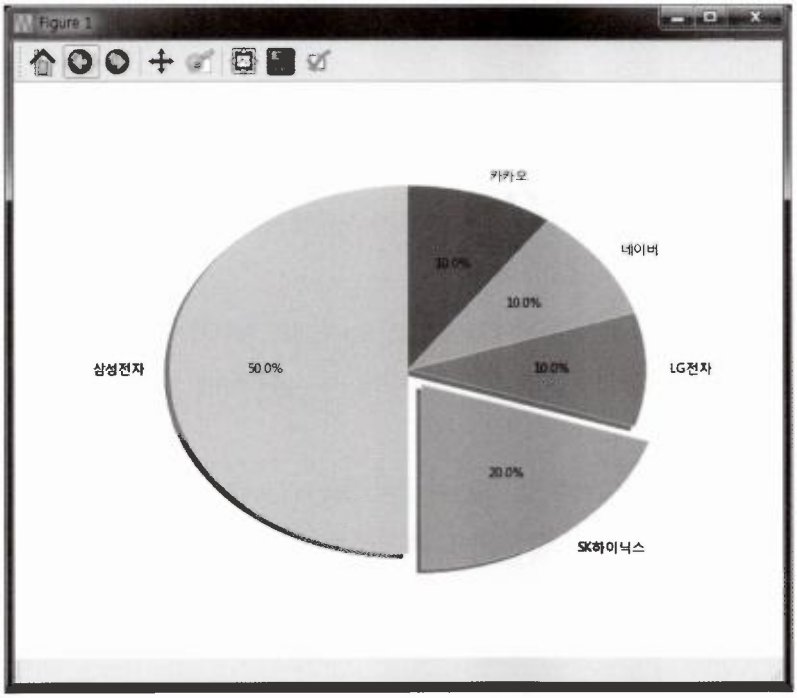


그림 15.25 matplotlib를 이용한 pie 차트 그리기

matplotlib를 사용하면 pie 차트 역시 간단하게 그래프로 그려볼 수 있습니다. 먼저 사용할 모듈을 임포트합니다.

```
import matplotlib.pyplot as plt
import numpy as np
```

```
from matplotlib import font_manager, rc
from matplotlib import style
```

그림 15.25의 pie 차트에서 일부 데이터가 한글로 출력되므로 한글 폰트 및 그래프 스타일을 설정합니다. 참고로 'ggplot'은 스타일의 한 종류일 뿐이고 matplotlib는 'ggplot' 외에도 여러 가지 스타일을 제공합니다.

```
font_name = font_manager.FontProperties(fname="c:/Windows/Fonts/malgun.ttf").get_name()
rc('font', family=font_name)
style.use('ggplot')
```

pie 차트를 그리는 데 필요한 데이터는 범주와 각 범주가 데이터에서 차지하는 비율입니다. 다음과 같이 간단히 데이터를 준비합니다.

```
labels = ['삼성전자', 'SK하이닉스', 'LG전자', '네이버', '카카오']
ratio = [50, 20, 10, 10, 10]
```

pie 차트는 matplotlib.pyplot 모듈의 pie 함수를 사용해 그릴 수 있습니다. 이번 예제에서는 명시적으로 Figure나 AxesSubplot 객체를 생성하지는 않겠습니다. 앞서 설명한 것처럼 명시적으로 객체를 생성하지 않아도 matplotlib.pyplot 모듈의 그래프 출력 함수가 호출될 때 자동으로 객체가 생성됩니다.

pie 함수의 첫 번째 인자는 각 범주가 데이터에서 차지하는 비율입니다. pie 차트에서 가장 중요한 데이터가 바로 이 값입니다. labels를 통해 범주를 전달할 수 있으며, 'shadow=True'를 통해 pie 차트에 그림자를 설정할 수 있습니다.

'startangle'은 첫 번째 pie의 시작 각도를 의미합니다. 이 값이 '90'이라는 것은 첫 번째 pie 데이터('삼성전자')가 90도 각도에서부터 그려짐을 의미합니다. 잘 이해가 되지 않는다면 이 값을 45, 135로 각각 변경해본 후 그래프를 그려보기 바랍니다.

```
plt.pie(ratio, labels=labels, shadow=True, startangle=90)
plt.show()
```

지금까지 설명한 코드를 실행하면 그림 15.26과 같은 모양의 pie 차트가 그려집니다. 그림 15.25와 비교하면 pie 차트에서 각 pie의 색상이 다르고 'SK하이닉스' 종목이 확대되지 않았음을 확인할 수 있습니다. 차트의 스타일도 중요하지만 차트의 색상을 잘 선택하지 않으면 차트가 예쁘게 출력되지 않음을 알 수 있습니다.

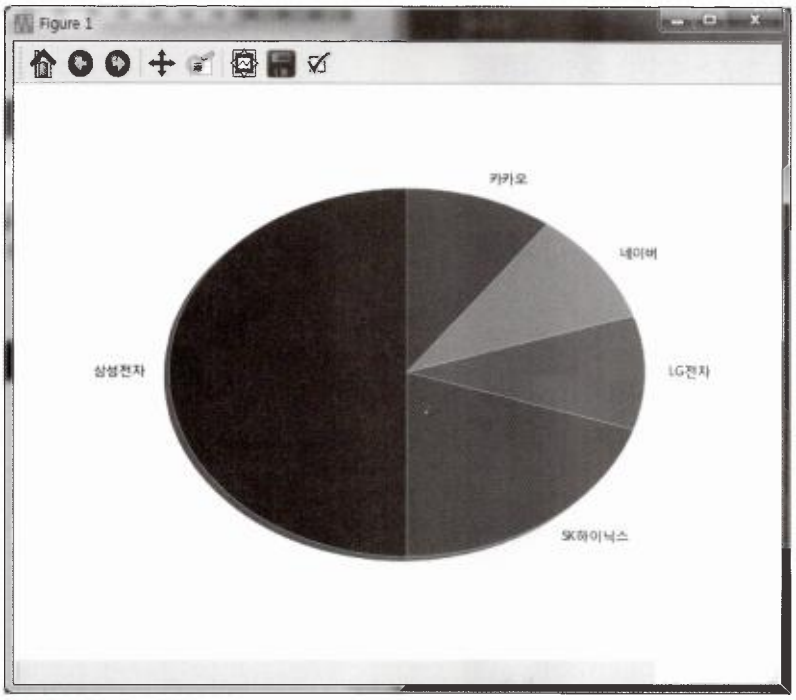


그림 15.26 pie 차트 - 1(소스코드: book/ch15/17.py)

차트 색상을 설정하려면 각 범주에 해당하는 색상을 리스트 형태로 전달하면 됩니다. 그리고 특정 pie를 확대하려면 각 범주에 확대 값을 리스트 형태로 전달하면 됩니다. 색상 변경과 확대를 위해 새로 설정한 데이터는 다음과 같습니다.

```
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'red']
labels = ['삼성전자', 'SK하이닉스', 'LG전자', '네이버', '카카오']
ratio = [50, 20, 10, 10, 10]
explode = (0.0, 0.1, 0.0, 0.0, 0.0)
```

확대를 위해 explode 인자에 확대 값을 전달했고, 색상 변경을 위해 colors 인자에 색상 값 리스트를 전달했습니다. 마지막으로 각 범주가 데이터에서 차지하는 비율을 출력하기 위해 autopct라는 키워드 인자를 사용했습니다.

```
plt.pie(ratio, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True,
startangle=90)
plt.show()
```

예제 15.6은 그림 15.26의 그래프를 그리는 전체 코드입니다. matplotlib 라이브러리를 사용하니 pie 차트 역시 쉽게 그릴 수 있음을 확인할 수 있습니다.

예제 15.6 pie 차트-2(소스코드: book/ch15/18.py)

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import font_manager, rc
from matplotlib import style

font_name = font_manager.FontProperties(fname="c:/Windows/Fonts/malgun.ttf").get_name()
rc('font', family=font_name)
style.use('ggplot')

colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'red']
labels = ['삼성전자', 'SK하이닉스', 'LG전자', '네이버', '카카오']
ratio = [50, 20, 10, 10, 10]
explode = (0.0, 0.1, 0.0, 0.0, 0.0)

plt.pie(ratio, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True,
startangle=90)
plt.show()
```

16

PyQt를 이용한 GUI 프로그래밍

이미 12장을 통해 간단한 GUI(Graphical User Interface) 프로그래밍을 해봤습니다. 이번 장에서는 앞에서 살펴본 PyQt 내용을 간단히 복습한 후 PyQt에 대해 좀 더 자세히 살펴보겠습니다.

사실 알고리즘 트레이딩을 하는 데 GUI 프로그래밍이 꼭 필요한 것은 아닙니다. GUI나 TUI(Text-based User Interface)는 데이터를 효과적으로 표현하기 위한 수단일 뿐 프로그램의 핵심 로직이나 알고리즘과는 관계가 없기 때문입니다. 그렇지만 여러분이 자주 사용하는 HTS가 텍스트 기반으로 동작한다면 조금 불편하겠지요? 이번 장을 통해 PyQt를 제대로 배워둔다면 여러분들이 만들려는 프로그램을 쉽게 개발할 수 있을 것입니다.

01 PyQt 기초

PyQt는 파이썬에서 GUI 프로그래밍을 할 때 사용하는 대표적인 패키지입니다. 현재 사용 중인 아나콘다 배포판(Anaconda 4.3.1)에는 현재 PyQt5 버전이 기본 패키지로 포함돼 있습니다. 이번 절에서는 아나콘다에서 PyQt를 설치 및 업데이트하는 방법과 12장에서 다룬 PyQt의 기초 부분에 대해 간단히 복습해 보겠습니다.

아나콘다에서 PyQt 설치 및 업데이트

PyQt는 아나콘다 배포판을 설치하면 기본적으로 설치되는 패키지입니다. 따라서 아나콘다 배포판 버전에 따라 설치된 PyQt 패키지 버전이 다를 수 있습니다. 먼저 그림 16.1과 같이 윈도우의 프로그램 메뉴에서 [Anaconda Prompt]를 클릭해 Anaconda Prompt를 실행합니다.

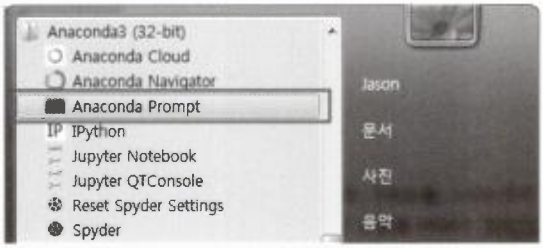


그림 16.1 Anaconda Prompt 실행

아나콘다 배포판에서 현재 설치된 패키지 목록을 확인하려면 conda list 명령을 사용하면 됩니다. 참고로 conda list 명령 이후에 아무것도 입력하지 않으면 배포판에 설치된 모든 패키지의 정보가 출력됩니다.

PyQt 패키지에 대한 정보만 확인하려면 그림 16.2와 같이 conda list pyqt를 입력하면 됩니다. 이때 패키지의 이름은 대소문자를 구분하지는 않습니다. 즉, pyqt라고 입력하거나 PyQt라고 입력해도 정상적으로 PyQt 패키지 정보를 확인할 수 있습니다.

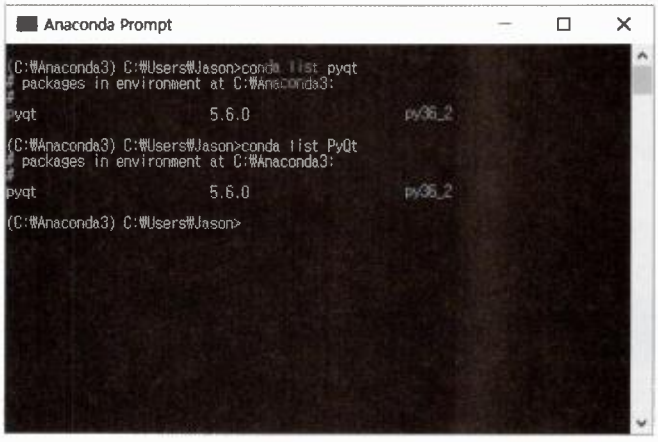


그림 16.2 conda list pyqt를 이용한 패키지 정보 확인

그림 16.2를 참조하면 현재 저자의 개발 환경에서는 파이썬 3.6용으로 빌드된 PyQt 5.6.0 버전의 패키지가 설치돼 있음을 확인할 수 있습니다. 아나콘다 배포판을 개발하고 있는 Continuum에서는 개별 패키지에 대해 지속적으로 업데이트하고 있습니다. 따라서 아나콘다 배포판 사용자는 배포판을 새로 설치하지 않고도 특정 패키지만 업데이트할 수 있습니다.

Anaconda Prompt에서 패키지를 업데이트하려면 conda update 명령을 사용하면 됩니다. 예를 들어, PyQt 패키지를 업데이트하려면 conda update pyqt를 입력하면 됩니다. 그림 16.3을 참조하면 현재는 PyQt를 업데이트할 내용이 없는 것을 확인할 수 있습니다.



그림 16.3 conda update를 사용한 PyQt 패키지 업데이트

PyQt 복습하기

이번 절에서는 12.2절에서 설명한 PyQt 기초를 복습하겠습니다. PyQt가 잘 기억나지 않는다면 잠깐 12.2절을 먼저 읽어보기 바랍니다.

PyQt는 따로 한 권 분량의 책이 있을 만큼 복잡하고 규모가 큰 라이브러리입니다. 따라서 이 책에서 PyQt를 모두 다룰 수는 없습니다. 대신 알고리즘 트레이딩 프로그램 UI를 만들 수 있을 만큼만 공부하겠습니다.

PyQt는 분명히 복잡한 라이브러리이긴 하지만 PyQt 기반의 GUI 프로그램의 핵심 구성 요소는 그림 16.4와 같이 정리할 수 있습니다. 먼저 화면에 출력되는 UI 부분이 있습니다. 이는 PyQt가 기본적으로 제공하는 위젯(Widget) 클래스의 객체를 생성해서 만들 수 있습니다. 수많은 위젯 중에서 여러분이 사용할 위젯을 선택하고 각 위젯 클래스의 사용법을 익혀야 합니다.

두 번째는 이벤트 루프입니다. PyQt에서는 QApplication 객체에서 exec_ 메서드를 호출해 이벤트 루프를 생성합니다. 여러분은 이벤트 루프를 생성만 할 뿐 실제로 이벤트 루프에서 여러분이 관여하는 일은 많지 않습니다.

세 번째는 이벤트를 처리할 함수 또는 메서드를 구현하는 부분입니다. 버튼과 같은 위젯을 클릭하면 해당 위젯은 'clicked'라는 시그널(signal)을 발생시킵니다. 여러분은 'clicked'라는 시그널이 발생했을 때 호출되는 함수 또는 메서드를 구현해야 합니다. PyQt에서는 특정 시그널이 발생했을 때 호출되는 함수 또는 메서드를 슬롯(slot)이라고 합니다. 참고로 다른 프로그래밍 언어에서는 콜백 함수(callback function)라고 부르기도 합니다.

PyQt에서는 위젯에서 발생하는 시그널에 대해 어떤 슬롯으로 처리할지에 대해 미리 등록함으로써 특정 위젯에서 시그널이 발생했을 때 이벤트 루프가 미리 연결된 슬롯을 자동으로 호출하게 돼 있습니다.

GUI 프로그램의 전반적인 실행 흐름에 대해 정리해 보겠습니다. 그림 16.4에서 사용자가 버튼 위젯을 클릭하면 'clicked'라는 시그널이 발생합니다. GUI 프로그램의 시작과 함께 생성돼 있던 이벤트 루프는 해당 시그널에 등록된 슬롯을 호출함으로써 사용자가 발생시킨 이벤트를 처리합니다. 이벤트 루프는 사용자가 프로그램을 종료한다는 시그널을 보내기 전까지는 계속해서 이와 같은 방식으로 시그널과 연결된 슬롯을 호출함으로써 이벤트를 처리하는 역할을 합니다.

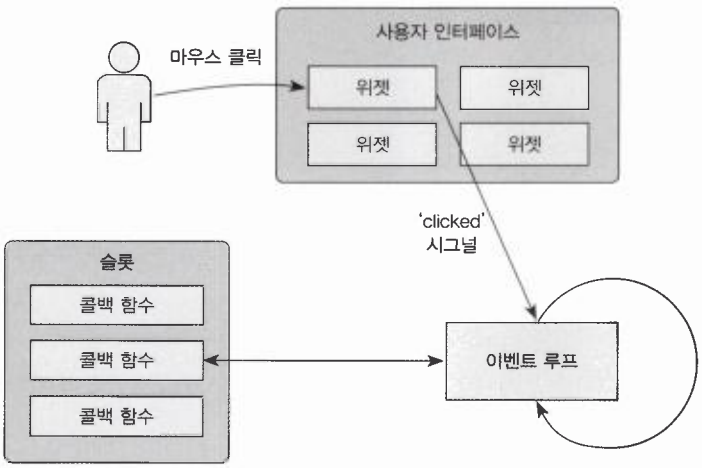


그림 16.4 PyQt의 UI-시그널-슬롯-이벤트 루프의 관계

PyQt의 전반적인 동작 방식을 복습했으니, 이번에는 간단한 코드를 통해 이벤트 루프를 생성하는 QApplication 객체를 살펴보겠습니다. 앞서 설명한 것처럼 그림 16.4의 이벤트 루프는 QApplication 클래스의 객체를 생성한 후 exec_ 메서드를 호출하는 순간 생성됩니다.

한번 생성된 이벤트 루프는 사용자가 윈도우를 닫을 때까지 실행되면서 위젯에서 발생한 시그널을 처리하는 슬롯을 호출하는 역할을 합니다. 이벤트 루프의 생성과 종료 시점을 확인해보기 위해 다음과 같이 코드를 작성해 보겠습니다.

```
import sys
from PyQt5.QtWidgets import *

app = QApplication(sys.argv)
label = QLabel("Hello, PyQt")
label.show()

print("Before event loop")
app.exec_()
print("After event loop")
```

작성된 코드에서는 이벤트 루프를 생성하는 코드의 앞쪽과 뒤쪽에 문자열이 출력되게 했습니다. 코드를 실행하면 그림 16.5와 같이 QLabel 위젯이 나타나고 콘솔 창에는 'Before event loop'라는 문자열이 출력됩니다.

여기서 한 가지 중요한 점은 파이썬 코드는 파일의 위에서부터 아래쪽으로 순차적으로 실행되는데 app.exec_ 앞쪽의 코드는 실행됐지만 그다음 줄의 코드는 아직 실행되지 않았다는 점입니다. 이는 exec_ 메서드가 호출되면서 이벤트 루프가 생성됐고 이 때문에 프로그램이 계속해서 이벤트 루프 안에서 실행되고 있기 때문입니다.



그림 16.5 이벤트 루프 생성(소스코드: book/ch16/01.py)

이제 화면에 출력된 QLabel 위젯을 닫아 보겠습니다. 윈도우의 'X' 부분을 눌러 QLabel 위젯을 종료하면 exec_ 메서드 호출 시점에 생성됐던 이벤트 루프가 종료되면서 그다음 줄의 코드가 실행됩니다. 따라서 그림 16.6과 같이 콘솔 창에 'After event loop'라는 문자열이 출력됨을 확인할 수 있습니다.



그림 16.6 이벤트 루프 종료

이번에는 시그널과 슬롯을 연결하는 부분에 대한 코드를 작성해 보겠습니다. PyQt는 위젯의 종류에 따라 발생 가능한 기본 시그널이 정의돼 있습니다. 예를 들어, QPushButton 위젯은 마우스 클릭을 했을 때 'clicked'라는 시그널이 발생합니다.

시그널이 발생했을 때 호출되는 함수 또는 메서드를 슬롯이라 부른다고 했습니다. 슬롯을 구현했다면 시그널과 슬롯을 연결만 해주면 됩니다. 시그널이 발생했을 때 연결된 슬롯을 호출하는 역할은 앞서 살펴본 이벤트 루프가 알아서 처리해줍니다.

다음 코드에서는 QPushButton 위젯을 생성하고 해당 위젯에서 'clicked' 시그널이 발생하면 호출되는 슬롯인 clicked_slot 함수를 구현했고 시그널과 슬롯을 이벤트 루프를 생성하기 전에 미리 연결했습니다.

```
import sys
from PyQt5.QtWidgets import *

def clicked_slot():
    print('clicked')

app = QApplication(sys.argv)

btn = QPushButton("Hello, PyQt")
btn.clicked.connect(clicked_slot)
btn.show()

app.exec_()
```

위 코드를 실행해 보면 그림 16.7과 같이 QPushButton 객체가 화면에 출력됩니다. 버튼을 클릭하면 콘솔 창에 'clicked'라는 문자열이 출력됨을 확인할 수 있습니다. 이는 QPushButton 위젯에서 'clicked' 이벤트가 발생하면 'clicked_slot'이라는 이름의 함수가 호출되도록 미리 연결해 뒀기 때문입니다.

여기서 한 가지 주의할 점은 슬롯의 구현과 시그널-슬롯의 연결이 모두 이벤트 루프를 생성하는 app.exec_ 메서드 호출보다 먼저 수행돼야 한다는 점입니다.



그림 16.7 간단한 시그널-슬롯 연결의 예

지금까지 이벤트 루프 생성과 시그널-슬롯의 연결에 대해 복습했습니다. 이번에는 위젯을 이용해 UI를 구성하는 부분에 대해 복습하겠습니다. 이번에 구현할 프로그램은 그림 16.8과 같습니다.



그림 16.8 윈도우 및 이벤트 처리

PyQt에서 위젯은 UI를 구성하는 핵심 요소입니다. PyQt에서는 모든 위젯이 최상위 위젯을 의미하는 윈도우가 될 수 있습니다. 그러나 대부분의 프로그램에서는 QMainWindow나 QDialog 클래스를 사용해 윈도우를 생성합니다.

예제 16.1 QMainWindow를 이용한 윈도우 구성(소스코드: book/ch16/03.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setWindowTitle("Review")
```

```
        btn1 = QPushButton("Click me", self)
        btn1.move(20, 20)
        btn1.clicked.connect(self.btn1_clicked)

    def btn1_clicked(self):
        QMessageBox.about(self, "message", "clicked")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

그림 16.8 프로그램의 전체 코드는 예제 16.1과 같습니다. 코드가 기존 코드에 비해 복잡해 보이지만 각 부분을 나눠서 이해하면 쉽게 파악할 수 있습니다. 가장 먼저 살펴볼 부분은 QApplication 객체를 생성하고 이벤트 루프를 만드는 부분입니다.

먼저 QApplication 객체인 app을 생성하고 exec_ 메서드를 호출해서 이벤트 루프를 생성합니다. 그리고 이벤트 루프를 생성하기 전에 MyWindow라는 클래스의 객체를 생성하는 것을 확인할 수 있습니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

앞서 배운 내용을 상기해보면 이벤트 루프를 생성하기 전에는 UI를 구성해야 하고 UI를 구성하는 기본 단위인 위젯에 대해 시그널과 슬롯을 연결해야 했습니다.

UI가 간단하다면 몇 줄의 코드만으로도 구현할 수 있겠지만 UI가 복잡한 경우에는 클래스를 사용하는 것이 좋습니다. 특히 클래스를 사용하면 기존 위젯 클래스를 상속함으로써 여러분만의 클래스를 새롭게 정의할 수 있습니다.

MyWindow 클래스의 객체는 이벤트 루프가 실행되기 전에 생성되며 이때 자동으로 생성자가 호출됩니다. 따라서 코드를 분석할 때는 먼저 생성자 부분부터 살펴보는 것이 좋습니다. 다음 코드를 살펴보면 MyWindow 클래스는 QMainWindow 클래스를 상속받고 있습니다.

자식 클래스인 MyWindow가 부모 클래스인 QMainWindow를 상속받으면 QMainWindow 클래스에 정의된 모든 메서드나 프로퍼티를 사용할 수 있습니다. 클래스의 상속은 우리 생활에서 보면 부모님을 잘 만나서 태어날 때부터 '금수저'를 물고 태어나는 것과 비슷합니다.

```
class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()
```

MyWindow 클래스는 단순히 부모 클래스를 상속할 뿐만 아니라 자기 자신의 UI를 변경하는 메서드인 setupUI를 호출합니다 물론 QPushButton과 같은 객체 생성 코드를 모두 MyWindow 클래스의 생성자에 넣을 수도 있지만, 따로 메서드로 구성해서 코드의 가독성을 좋게 합니다. 여기서 중요한 코드는 'clicked'라는 시그널과 이를 처리하는 슬롯인 self.btn1_clicked를 연결하는 코드입니다.

```
def setupUI(self):
    self.setWindowTitle("Review")

    btn1 = QPushButton("Click me", self)
    btn1.move(20, 20)
    btn1.clicked.connect(self.btn1_clicked)
```

QPushButton에서 'clicked' 시그널이 발생할 때 자동으로 호출되는 슬롯인 self.btn1_clicked는 다음과 같이 MyWindow 클래스의 메서드로 구현합니다.

```
def btn1_clicked(self):
    QMessageBox.about(self, "message", "clicked")
```

02 Qt Designer

지금까지는 UI 구성을 위해 직접 파이썬 코드를 작성했습니다. 간단한 UI를 구성하는 경우에는 이처럼 직접 코드를 작성하는 것도 편리합니다. 하지만 복잡한 UI를 구성할 때는 UI 구성을 위한 전용 툴인 Qt Designer를 사용하는 것이 더 편리합니다.

Qt Designer는 이름에서 알 수 있듯이 Qt의 컴포넌트를 이용해 GUI를 설계하는 전용 툴입니다. 설계라는 단어가 들어 있어 복잡해 보이지만 Qt Designer는 프로그래머가 GUI 구성을 위지위그(WYSIWYG: What You See Is What You Get) 방식으로 수행할 수 있도록 지원합니다.

Qt Designer 소개

앞 절에서 배운 GUI 프로그래밍을 정리해보면 GUI 프로그램의 코드는 다음과 같이 크게 세 부분으로 구성돼 있습니다.

1. GUI 레이아웃
2. 시그널-슬롯 연결 및 슬롯 처리 함수(메서드)
3. 이벤트 루프

위의 세 가지 중 이벤트가 발생했을 때 실제로 프로그램에서 수행해야 하는 작업을 프로그래머가 함수나 메서드로 구현하는 것은 필수적입니다. 이런 것까지 프로그램이 알아서 판단하고 처리한다면 그것은 인공지능 프로그램과 같을 것입니다.

하지만 버튼과 같은 위젯을 생성하고 화면에 출력될 윈도우의 크기를 조절하는 것과 같은 사소한 작업을 모두 파이썬 코드로 구현하는 것은 조금 귀찮고 따분한 작업일지도 모릅니다. 마이크로소프트의 파워포인트에서 도형을 사용해 그림을 그리는 것처럼 코드 한 줄 작성하지 않고도 쉽게 GUI 화면을 구성할 수 있는 툴(또는 프로그램)이 있으면 좋겠다는 생각을 많이들 할 겁니다. 이번 절에서 배울 Qt Designer가 바로 위지위그 방식으로 UI 구성을 도와주는 프로그램입니다. 여러분도 이 툴을 사용하면 귀찮은 GUI 작업에서 해방될 수 있습니다.

그림 16.9는 Qt Designer를 실행한 모습입니다. 왼쪽에 있는 위젯을 드래그 앤드 드롭해서 쉽게 레이아웃을 구성할 수 있습니다.

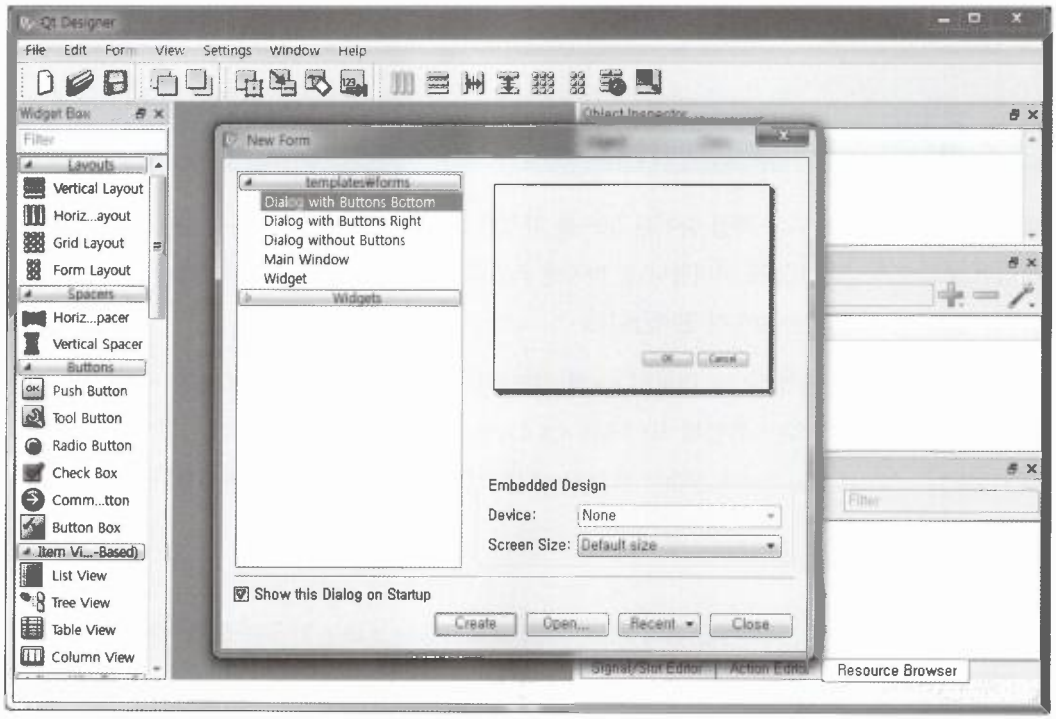


그림 16.9 Qt Designer

Qt Designer를 이용한 UI 구성

이번 절에서는 Qt Designer를 이용해 아주 간단한 형태의 UI를 구성해 보겠습니다. Qt Designer는 하나의 독립적인 프로그램으로 아나콘다 배포판에 포함돼 있습니다. 보통은 PyQt를 설치할 때 함께 설치됩니다.

그림 16.10과 같이 아나콘다 배포판에는 `C:\Anaconda3\Library\bin` 디렉터리에 `designer.exe` 파일이 존재합니다. 이 파일이 바로 Qt Designer 실행 파일입니다. 윈도우에서는 해당 파일을 더블클릭하면 Qt Designer를 실행할 수 있습니다.

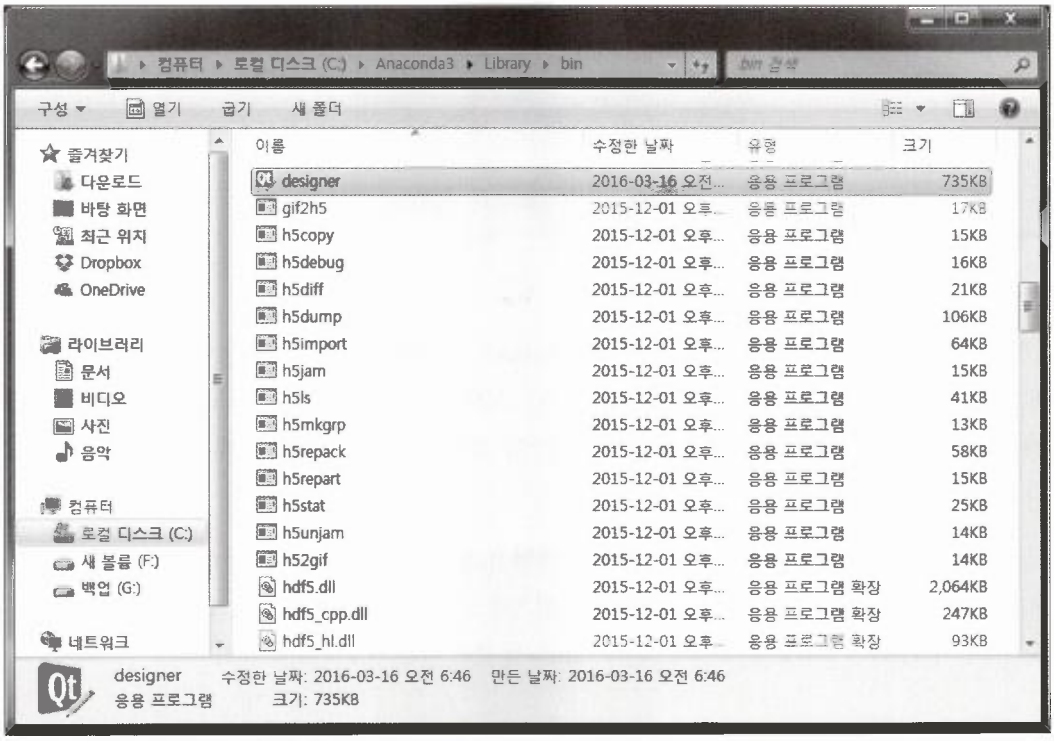


그림 16.10 Qt Designer 실행 파일 경로

Qt Designer를 실행했다면 가장 먼저 할 일은 Dialog나 Main Window를 생성하는 일입니다. 그림 16.11과 같이 [Main Window]를 선택한 후 [Create] 버튼을 눌러 윈도우를 생성합니다.

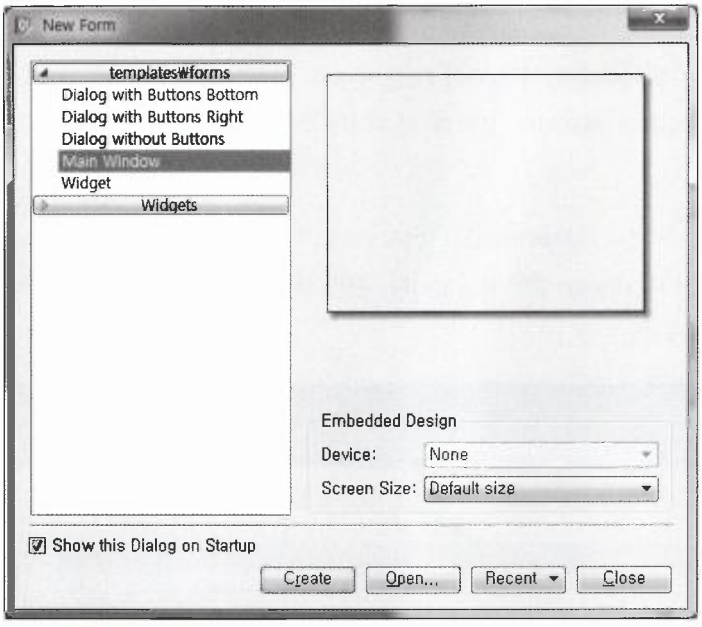


그림 16.11 Main Window 생성

윈도우를 생성하면 그림 16.12와 같이 Qt Designer의 중앙에 윈도우가 나타납니다. 윈도우의 오른쪽 아래를 클릭해 윈도우 크기를 직접 조절할 수도 있습니다. 이때 한 줄의 코드도 작성하지 않았지만 `QMainWindow` 클래스를 상속받아 `MyWindow` 클래스를 정의하고 해당 클래스에 대한 객체를 생성한 것과 비슷한 윈도우가 나타났습니다.

그림 16.12를 보면 생성된 윈도우의 왼쪽에는 위젯 Box가 보이고, 오른쪽에는 Object Inspector나 Property Editor가 보입니다. 앞으로 여러분은 왼쪽에 있는 위젯들을 윈도우로 드래그 앤드 드롭해서 위치시킨 후 Object Inspector나 Property Editor를 통해 값을 변경하게 될 것입니다.

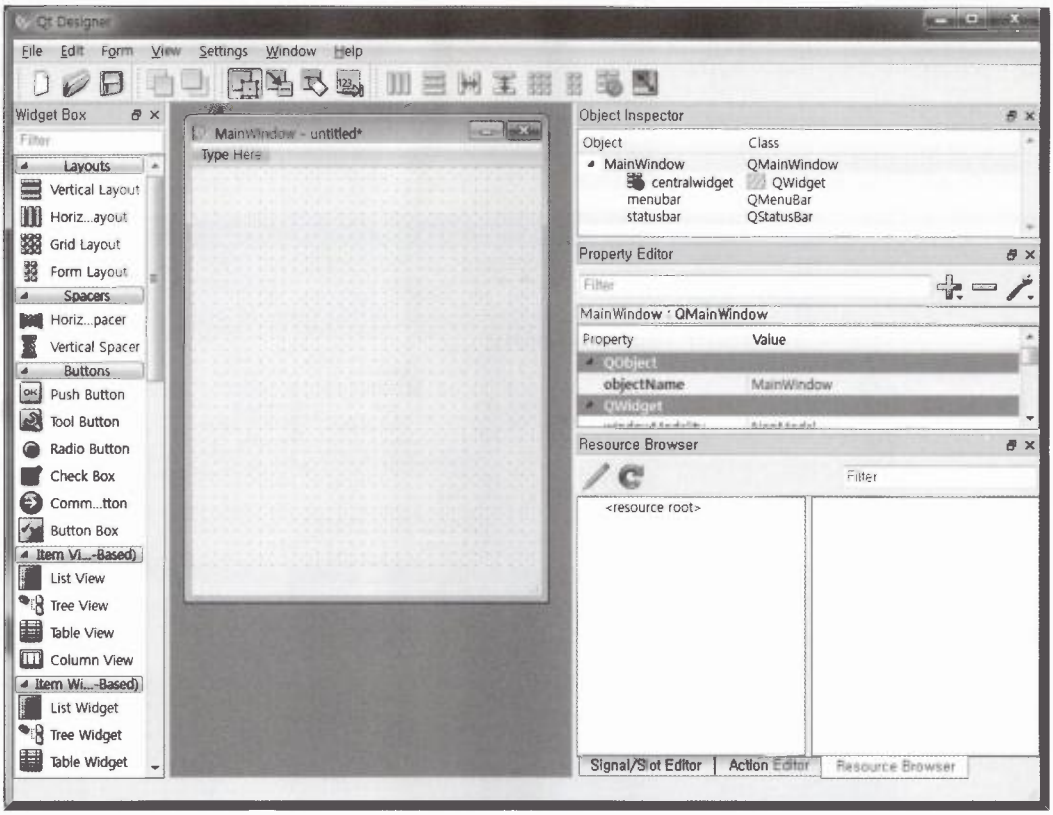


그림 16.12 Qt Designer 구성

아직 감흥이 없는 분들을 위해 위젯 Box에서 Push Button 위젯과 Label 위젯을 선택한 후 윈도우로 드래그 앤드 드롭해서 그림 16.13과 같이 위젯들을 배치해 보겠습니다.

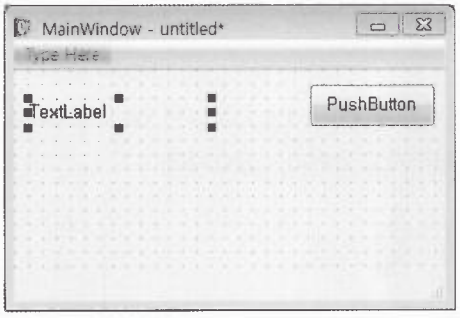


그림 16.13 Qt Designer를 사용한 레이아웃 디자인

위젯이 배치됐다면 이번에는 Property Editor를 이용해 각 위젯의 속성값을 변경해 보겠습니다. 그림 16.14와 같이 MainWindow의 PushButton을 클릭한 상태에서 Property Editor의 text 항목의 속성값을 '클릭'으로 변경합니다. Property Editor에서 속성값을 변경하면 바로 MainWindow에 반영되는 것을 확인할 수 있습니다.

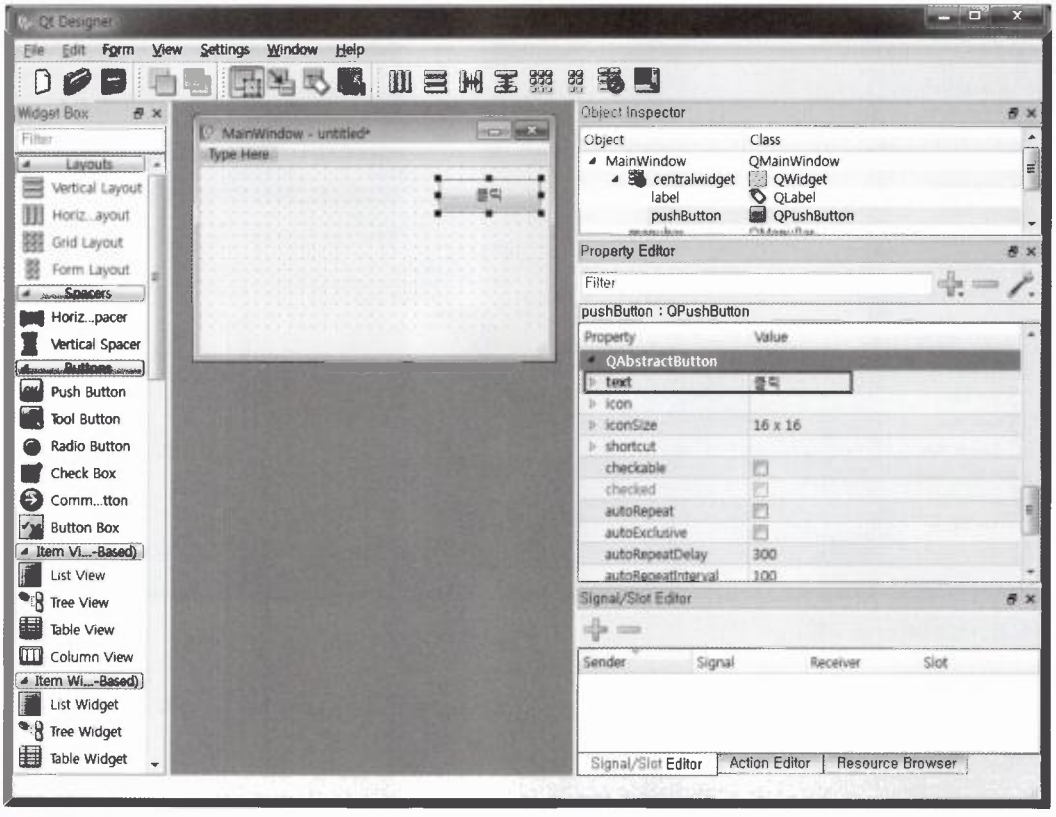


그림 16.14 Property Editor를 이용한 속성 변경-1

이번에는 그림 16.15와 같이 MainWindow에서 Label을 선택한 후 Property Editor의 text 속성값을 공백 상태로 비워 둡니다. 공백 상태로 두면 Label 객체에 아무것도 출력되지 않습니다.

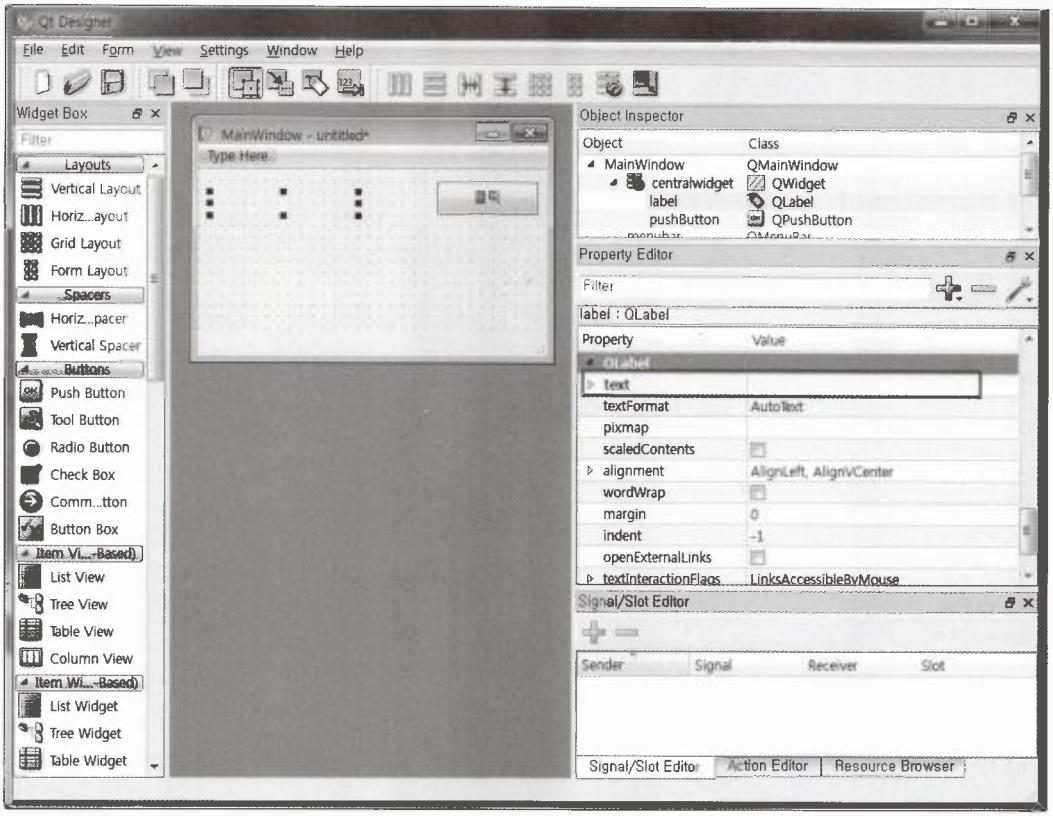


그림 16.15 Property Editor를 이용한 속성 변경-2

지금까지 작업한 내용에 대한 결과물을 미리 보려면 단축키인 Ctrl + R을 누르면 됩니다. Ctrl + R을 누르면 그림 16.16과 같이 작업 중인 UI의 최종 결과물인 윈도우가 출력되는 것을 확인할 수 있습니다. 다만 아직 시그널과 슬롯은 따로 처리하지 않았기 때문에 버튼을 클릭해도 어떤 동작도 수행되지 않습니다.

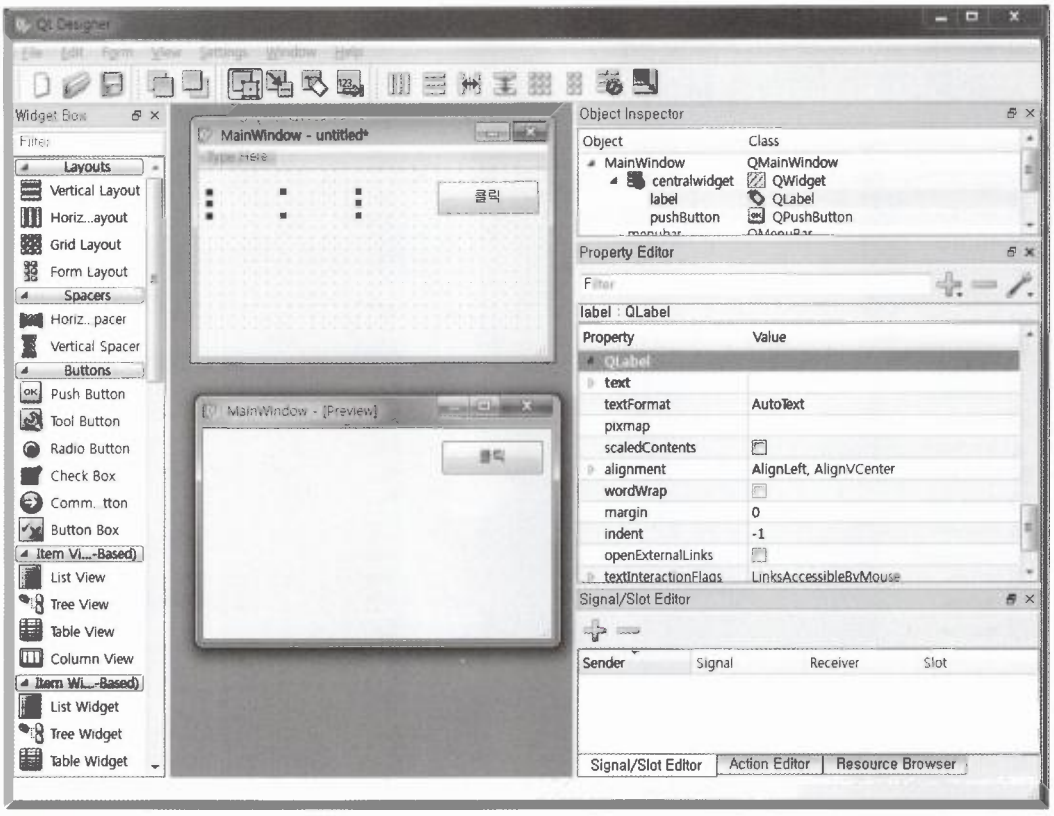


그림 16.16 미리 보기(단축키: Ctrl + R)

구성한 UI가 마음에 든다면 이를 저장합니다. Qt Designer의 툴바에서 디스켓 모양의 아이콘을 클릭하거나 [File] → [Save] 메뉴를 차례로 클릭하면 'Save Form As' 윈도우가 나타납니다. 그림 16.17과 같이 윈도우 바탕 화면에 'main_window'라는 이름으로 저장합니다.



그림 16.17 UI 파일 저장

윈도우 메모장을 실행한 후 바탕 화면에 저장된 main_window.ui 파일을 열어보면 지금까지 구성한 UI 레이아웃이 XML이라는 형식으로 저장된 것을 확인할 수 있습니다.

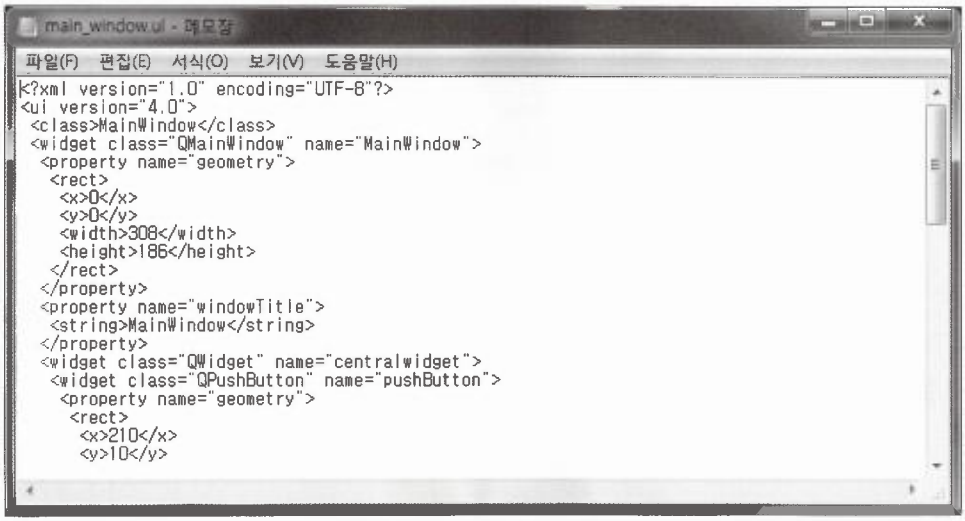


그림 16.18 Qt Designer UI 파일의 내용(XML 형식)

UI 파일을 파이썬 코드로 변환하기

앞 절에서 Qt Designer를 통해 main_window.ui라는 파일을 만들었습니다. 생성된 UI 파일을 사용해 GUI를 실행하는 방법에는 두 가지가 있습니다. 이번 절에서는 그중 첫 번째 방법에 대해 알아보겠습니다.

앞서 설명한 것처럼 Qt Designer의 결과 파일인 UI 파일은 XML이라는 언어로 구성돼 있습니다. 파이썬은 XML 코드를 바로 실행할 수 없으므로 XML 코드를 파이썬 코드로 변환하는 것이 필요합니다.

PyQt5는 XML로 구성된 UI 파일을 파이썬 코드로 자동으로 변환하는 pyuic.py라는 프로그램을 제공합니다. 확장자를 보면 알겠지만 이 프로그램 역시 파이썬으로 구현돼 있습니다. pyuic.py 파일은 XML 파일을 입력받은 후 이를 파이썬 코드로 변환합니다.

먼저 윈도우 바탕 화면에 저장된 main_window.ui라는 파일을 C:\Anaconda3\Lib\site-packages\PyQt5\uic라는 디렉터리로 복사합니다.

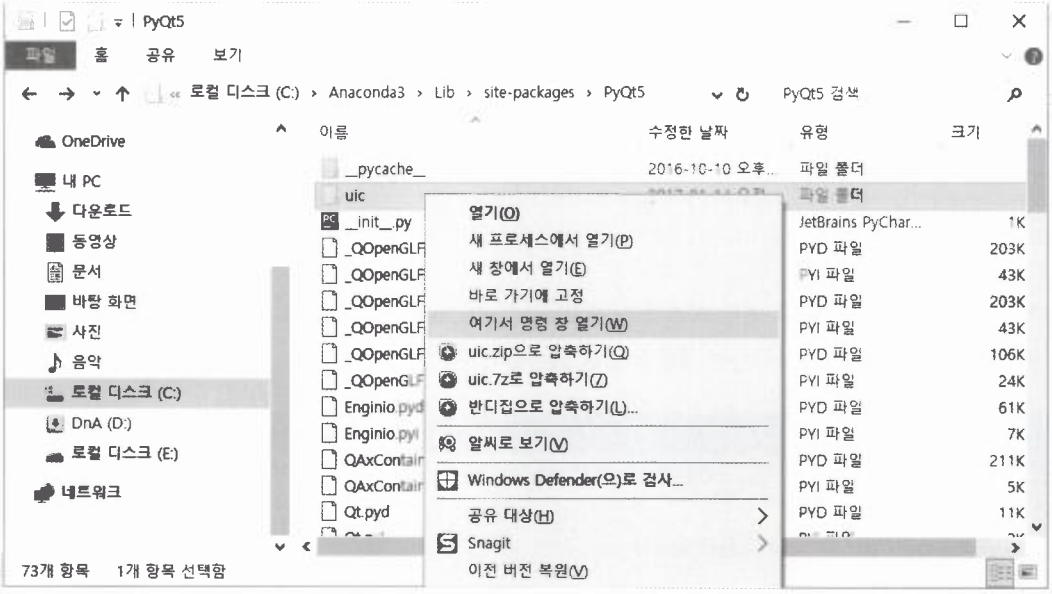


그림 16.19 uic 디렉터리에서 명령 창 열기

그림 16.19와 같이 C:\Anaconda3\Lib\site-packages\PyQt5 디렉터리에서 uic 디렉터리를 선택한 후 Shift 키를 누른 상태에서 마우스 오른쪽 버튼을 클릭합니다. 그림 16.19와 같은 메뉴가 나타나면 [여기서 명령 창 열기] 메뉴를 선택합니다.

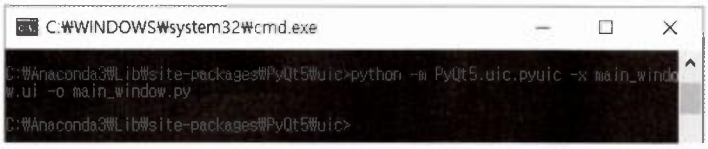


그림 16.20 pyuic를 이용해 UI 파일을 파이썬 코드로 변환

명령 창이 실행되면 그림 16.20과 같이 명령어를 입력해 main_window.ui 파일을 main_window.py 파일로 변환합니다. 에러 없이 정상적으로 변환이 수행되면 C:\Anaconda3\Lib\site-packages\PyQt5\uic 디렉터리에 main_window.py 파일이 생성됩니다. 생성된 main_window.py 파일을 열어보면 그림 16.21과 같이 파이썬 코드로 구성된 것을 확인할 수 있습니다.

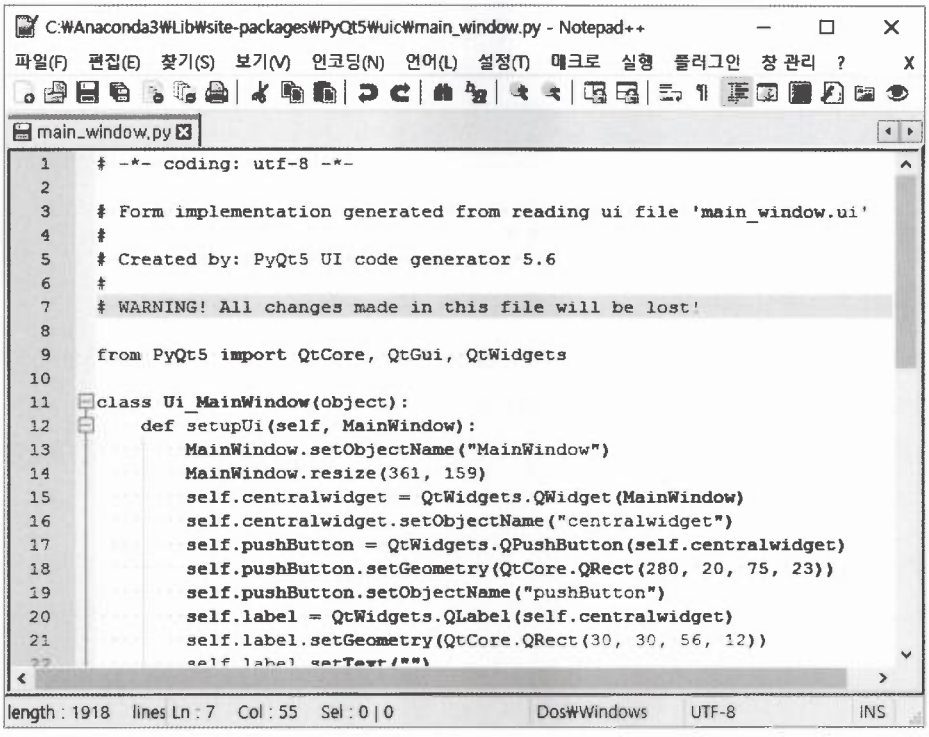


그림 16.21 UI 파일로부터 생성된 파이썬 코드

UI 파일로부터 생성된 파이썬 코드(main_window.py)를 실행하기 위해 그림 16.22와 같이 명령 창에서 명령을 수행합니다. 그림 16.22를 살펴보면 앞서 Qt Designer를 통해 구성했던 UI가 그대로 화면에 출력됨을 확인할 수 있습니다.

이처럼 Qt Designer를 이용하면 UI 구성을 위한 코드를 작성하지 않고도 편리하게 UI를 구성하고 프로그램을 동작시킬 수 있습니다. 물론 아직은 버튼에 대한 이벤트(signal)를 처리하는 메서드 (slot)를 구현하지 않았기 때문에 버튼을 클릭해도 아무런 동작을 수행하지 않습니다.

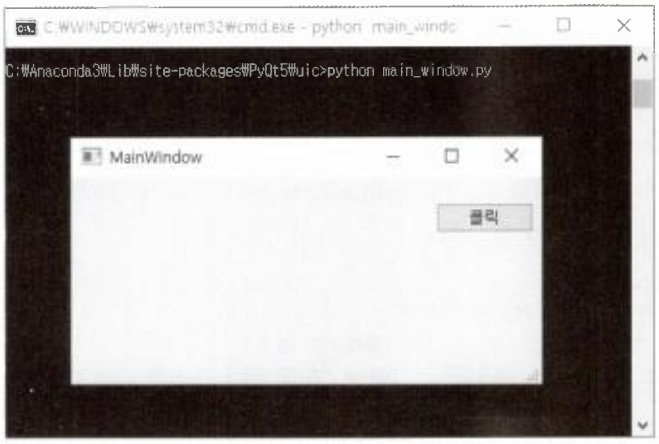


그림 16.22 UI 파일로부터 생성된 파이썬 코드를 실행

UI 파일을 파이썬 코드에서 로드하기

앞 절에서는 UI 파일을 pyuic.py 파일을 사용해 파이썬 코드로 변환한 후 이를 실행하는 방법을 알아봤습니다. 이 같은 방법은 편리하긴 하지만 UI가 변경될 때마다 UI 파일을 파이썬으로 변경해야 하는 문제가 있습니다. 만약 UI 파일로부터 변환된 파이썬 코드에 이벤트 처리를 위한 코드를 함께 구현했다면 문제는 더 심각해질 것입니다.

이번 절에서는 UI 파일을 파이썬 코드로 변환하지 않고 사용하는 방법을 알아보겠습니다. 앞 절에서 main_window.ui 파일을 윈도우 바탕 화면에 저장했습니다. 예제 16.2의 코드를 run.py라는 파일로 저장한 후 main_window.ui 파일이 있는 경로인 윈도우 바탕 화면에 저장합니다.

예제 16.2 ui 파일을 사용한 UI 구성(소스코드: book/ch16/04.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5 import uic

form_class = uic.loadUiType("main_window.ui")[0]
```

```
class MyWindow(QMainWindow, form_class):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

예제 16.2를 실행하면 그림 16.23과 같은 윈도우가 화면에 출력됩니다. 이는 앞서 UI 파일을 파이썬 코드로 변환한 후 변환된 파이썬 코드를 실행했을 때와 같습니다.



그림 16.23 UI 파일 로드를 통한 GUI 프로그램 실행

예제 16.2를 살펴보면 QApplication 클래스를 이용해 이벤트 루프를 생성하는 부분과 MyWindow라는 클래스를 정의한 부분이 있습니다. MyWindow 클래스를 살펴보면 QMainWindow와 form_class 클래스로부터 다중 상속을 받았음을 확인할 수 있습니다. 여기서 form_class는 main_window.ui 파일을 로드해서 만든 클래스입니다.

MyWindow 클래스는 form_class를 상속받았기 때문에 form_class에 정의돼 있던 속성이나 메서드를 모두 상속받게 됩니다. 따라서 다음과 같이 setupUi 메서드를 호출함으로써 Qt Designer를 통해 구성한 UI를 화면에 출력할 수 있습니다. 참고로 setupUi라는 메서드 이름은 정해진 이름이기 때문에 그대로 사용해야 합니다.

```
self.setupUi(self)
```

이번 절에서 살펴본 방법은 UI 파일을 파이썬 코드로 변환하지 않고 파이썬 코드에서 UI 파일을 로드했습니다. 이 방식은 추후에도 UI 파일을 Qt Designer를 통해 쉽게 수정할 수 있으며, UI 파일이 변경되더라도 파이썬 코드의 변경을 최소화할 수 있습니다.

이벤트 처리하기

Qt Designer를 사용하면 더 쉽고 편리하게 UI를 구성할 수 있음을 배웠습니다. 하지만 사용자 이벤트를 처리하는 메서드나 함수는 여전히 여러분이 직접 구현해야 합니다. 이번 절에서는 그림 16.23의 윈도우에서 버튼을 클릭할 때 발생하는 이벤트를 처리하는 프로그램을 작성해 보겠습니다.

예제 16.3은 예제 16.2의 기본 윈도우 구성 코드에 시그널(self.pushButton.clicked)과 슬롯(btn_clicked)을 연결하는 코드와 슬롯 구현부를 추가한 것입니다. 여기서 중요한 점은 여전히 UI의 구성은 main_window.ui 파일을 사용한다는 점입니다.

예제 16.3 슬롯 구현 및 시그널-슬롯 연결(소스코드: book/ch16/05.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5 import uic

form_class = uic.loadUiType("main_window.ui")[0]

class MyWindow(QMainWindow, form_class):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.pushButton.clicked.connect(self.btn_clicked)

    def btn_clicked(self):
        QMessageBox.about(self, "message", "clicked")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

예제 16.3을 살펴보면 버튼 객체에 접근하기 위해 self.pushButton을 사용했습니다. 그런데 self.pushButton이라는 변수는 예제 16.3에서 어떤 객체도 바인딩하고 있지 않습니다. 그 이유는 버튼 객체에 대한 생성 및 바인딩은 setupUi 메서드에서 수행되기 때문입니다.

그렇다면 우리는 어떻게 객체를 바인딩하고 있는 변수를 알 수 있을까요? 그림 16.24와 같이 Qt Designer를 실행한 후 main_window.ui 파일을 엽니다. 버튼을 클릭한 상태로 Property Editor를 살

펴보면 객체의 이름이 pushButton임을 확인할 수 있습니다. 물론 이 변수는 여러분이 언제든지 변경할 수 있지만 중요한 것은 이 값이 파이썬 코드에서 해당 객체에 접근하는 데 사용된다는 점입니다.

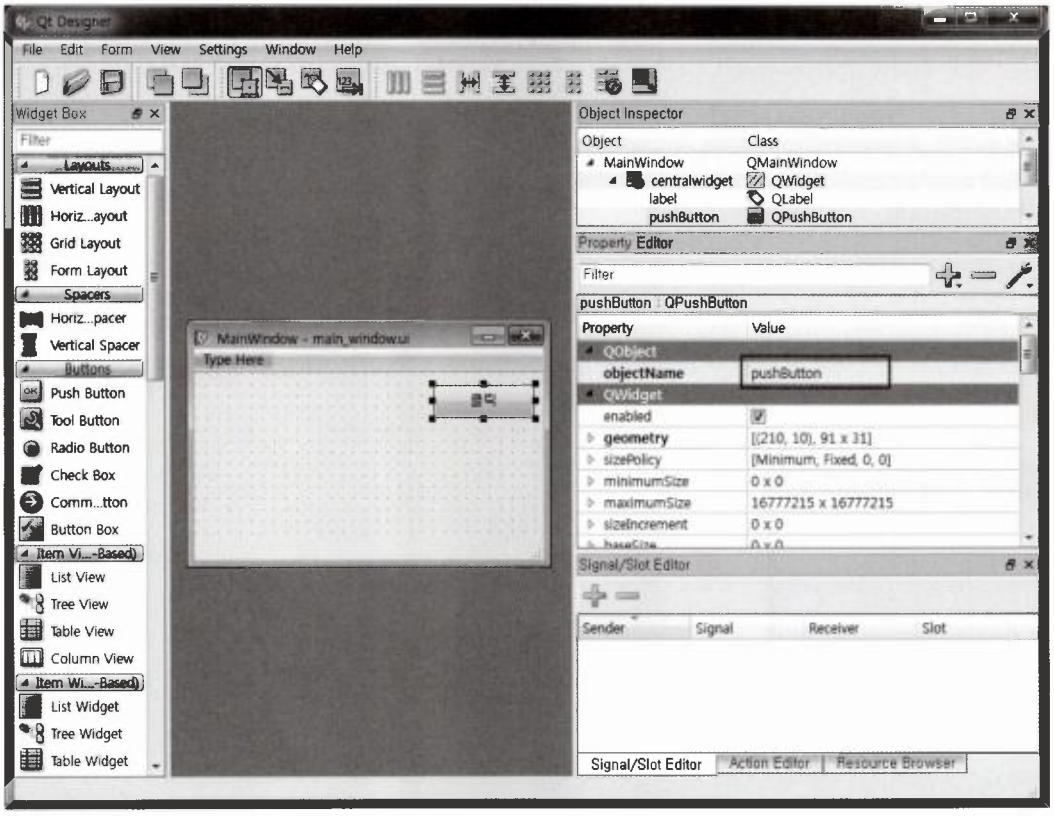


그림 16.24 버튼 객체의 QObject 값 확인

예제 16.3을 살펴보면 self.pushButton이라는 변수가 바인딩하고 있는 객체인 버튼에서 'clicked'라는 시그널이 발생하면 self.btn_clicked라는 메서드가 호출되는 것을 확인할 수 있습니다. btn_clicked 메서드는 QMessageBox를 통해 화면에 메시지 박스를 출력하도록 구현돼 있습니다.

예제 16.3을 실행한 후 버튼을 클릭하면 그림 16.25와 같이 메시지 박스가 화면에 출력되는 것을 확인할 수 있습니다.



그림 16.25 슬롯을 통한 시그널 처리

03 기본 위젯

PyQt에서 버튼, 메뉴 등과 같이 UI를 구성하는 데 사용되는 요소를 위젯이라고 합니다. GUI 프로그래밍은 여러 위젯을 활용해 최적의 UI를 구성한 후 각 위젯에서 발생하는 이벤트를 적절히 처리하는 것입니다. 이번 절에서는 PyQt에서 자주 사용하는 위젯을 살펴보겠습니다.

QPushButton

QPushButton은 버튼을 생성하기 위해 사용하는 위젯입니다. UI를 구성할 때 버튼은 가장 기본이 되는 위젯으로서 사용자로부터 '예', '아니오' 같은 이벤트를 받는 데 사용합니다. 그림 16.26은 윈도우에 QPushButton 위젯으로 생성한 버튼이 있습니다. 사용자가 '닫기'라는 버튼을 클릭하면 윈도우가 종료됩니다.



그림 16.26 QPushButton 위젯

예제 16.4 QPushButton 사용 예(소스코드: book/ch16/06.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        btn1 = QPushButton("닫기", self)
        btn1.move(20, 20)
        btn1.clicked.connect(QCoreApplication.instance().quit)
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

예제 16.4는 그림 16.26에 나온 프로그램의 소스코드입니다. 전체적인 코드 구성은 예제 16.1과 비슷합니다. 단지 QPushButton 객체에서 'clicked'라는 시그널이 발생하면 이를 처리하는 슬롯으로 QCoreApplication.instance().quit과 연결한 부분이 다릅니다.

참고로 QCoreApplication.instance()를 이용하면 예제 16.4에서 app이라는 변수가 바인딩하고 있는 동일한 객체를 얻어올 수 있습니다. app가 바인딩하고 있는 객체는 QApplication 클래스의 인스턴스인데 해당 객체는 quit이라는 메서드를 포함합니다. 버튼을 클릭했을 때 QApplication이 제공하는 quit 메서드를 호출하도록 연결함으로써 윈도우가 종료하는 것입니다.

예제 16.5 QPushButton 사용 예-2(소스코드: book/ch16/07.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

app = None

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        btn1 = QPushButton("닫기", self)
        btn1.move(20, 20)
        btn1.clicked.connect(app.quit)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

예제 16.4에서 QCoreApplication.instance()가 잘 이해되지 않는다면 예제 16.5를 살펴보기 바랍니다. 예제 16.5는 예제 16.4와 UI를 구성하는 코드는 같지만 app라는 변수를 전역적으로 선언함으로써 클래스 내에서도 app이라는 변수를 사용할 수 있게 했습니다.

예제 16.5를 보면 QPushButton에서 'clicked' 이벤트가 발생할 때 'app.quit' 메서드가 호출되도록 연결하는 코드를 명확히 이해할 수 있습니다. 다만 이러한 코딩 스타일은 app이라는 변수가 여러 곳에서 사용될 수 있기 때문에 프로그램의 유지보수 관점에서 볼 때 좋은 코딩 스타일은 아닙니다.

QLabel

QLabel 위젯은 텍스트나 이미지를 출력하는 데 사용합니다. 그림 16.33은 QLabel과 QPushButton 위젯을 통해 생성한 윈도우입니다. 사용자가 'Click'이라는 버튼을 클릭하면 QLabel 위젯을 통해 '버튼이 클릭되었습니다.'라는 메시지를 출력합니다. 사용자가 'Clear' 버튼을 클릭하면 QLabel 위젯에 출력한 메시지가 지워집니다.



그림 16.27 QLabel을 이용한 텍스트 출력

예제 16.6 QLabel을 이용한 텍스트 출력(소스코드: book/ch16/08.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 400, 300, 150)

        textLabel = QLabel("Message: ", self)
        textLabel.move(20, 20)

        self.label = QLabel("", self)
        self.label.move(80, 20)
        self.label.resize(150, 30)
```

```
        btn1 = QPushButton("Click", self)
        btn1.move(20, 60)
        btn1.clicked.connect(self.btn1_clicked)

        btn2 = QPushButton("Clear", self)
        btn2.move(140, 60)
        btn2.clicked.connect(self.btn2_clicked)

    def btn1_clicked(self):
        self.label.setText("버튼이 클릭되었습니다.")

    def btn2_clicked(self):
        self.label.clear()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

예제 16.6은 그림 16.27 프로그램에 대한 코드입니다. 이벤트 루프를 생성하고 MyWindow 클래스를 정의한 부분은 이전 예제와 같습니다. 다만 setupUI 메서드에서 위젯을 생성하고 시그널과 슬롯을 연결하는 부분과 시그널 처리를 위한 슬롯 구현이 수정됐습니다.

그림 16.27을 참조하면 화면에 텍스트를 출력하는 QLabel 객체가 두 개 사용됐습니다. 윈도우에서 항상 'Message'라는 텍스트를 출력하는 객체는 textLabel이라는 변수로 바인딩합니다. 이와 달리 버튼을 클릭할 때 '버튼이 클릭되었습니다.'라는 메시지를 출력하는 QLabel 객체는 self.label이라는 변수로 바인딩하고 있습니다.

어떤 객체에는 self를 붙이고 어떤 객체에는 self를 붙이지 않고 바인딩합니다. 저자가 십여 년 전에 프로그래밍을 처음 공부할 때는 이 부분이 매우 이해하기 어려웠습니다. 생성한 위젯을 클래스 내의 다른 메서드(setupUI가 아닌)에서 참조할 때는 변수 이름에 self를 붙여야 하고 그렇지 않은 경우에는 self를 붙이지 않는다고 생각하면 쉽습니다.

예제 16.6에서 self.label은 setupUI 메서드뿐 아니라 btn1_clicked와 btn2_clicked라는 메서드에서도 사용됩니다. 따라서 self를 붙여서 변수 이름을 생성했습니다. 이와 달리 textLabel은 setupUI 메서드에서만 사용하고 더는 참조되지 않기 때문에 self를 붙이지 않아도 되는 것입니다.

QLabel 클래스의 인스턴스를 생성한 후에는 move 메서드를 통해 해당 객체가 출력되는 위치를 조정할 수 있고, resize 메서드 호출을 통해 크기를 조정할 수 있습니다.

그림 16.27에서 윈도우 하단에는 두 개의 버튼이 존재합니다. 'Click'이라고 표시된 버튼을 클릭하면 QLabel 객체에 '버튼이 클릭되었습니다.'라는 텍스트가 출력됩니다. 'Clear'라고 표시된 버튼을 클릭하면 QLabel 객체에 출력된 텍스트가 지워집니다. 버튼 클릭이라는 이벤트를 적절히 처리해주기 위해 각 버튼 위젯의 'clicked'라는 시그널을 self.btn_clicked, self.btn2_clicked라는 슬롯과 각각 연결했습니다.

btn1 객체에서 'clicked' 시그널이 발생하는 경우 btn1_clicked라는 슬롯(메서드)이 호출됩니다. 따라서 해당 메서드 내에서 QLabel 객체에 텍스트를 출력하면 됩니다. QLabel 객체를 바인딩하고 있는 self.label 변수로 setText 메서드를 호출해서 텍스트를 QLabel 객체에 출력하면 됩니다.

```
def btn1_clicked(self):
    self.label.setText("버튼이 클릭되었습니다.")
```

btn2 객체에서 'clicked'라는 시그널이 발생하면 QLabel 객체에 출력된 문자열을 지웁니다. 이 경우 역시 btn1 객체와 마찬가지로 self.label 변수를 통해 QLabel 객체에 접근한 후 clear 메서드를 호출해서 문자열을 지울 수 있습니다.

```
def btn2_clicked(self):
    self.label.clear()
```

QLineEdit와 QStatusBar

QLineEdit는 한 줄의 텍스트를 입력할 수 있는 위젯으로, 사용자로부터 간단한 텍스트를 입력받을 때 사용합니다. 그림 16.28은 QLineEdit를 이용해 사용자로부터 주식 종목 코드를 입력받는 프로그램입니다.



그림 16.28 QLineEdit 사용 예

예제 16.7은 그림 16.28에 나온 프로그램의 소스코드입니다.

예제 16.7 QLineEdit 사용 예(소스코드: book/ch16/09.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 400, 300, 150)

        # Label
        label = QLabel("종목코드", self)
        label.move(20, 20)

        # LineEdit
        self.lineEdit = QLineEdit("", self)
        self.lineEdit.move(80, 20)
        self.lineEdit.textChanged.connect(self.lineEditChanged)

        # StatusBar
        self.statusBar = QStatusBar(self)
        self.setStatusBar(self.statusBar)

    def lineEditChanged(self):
        self.statusBar.showMessage(self.lineEdit.text())

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

여섯 개의 숫자로 구성된 종목 코드는 QLineEdit를 이용해 사용자로부터 입력받을 수 있습니다. 이때 중요한 점은 QLineEdit 객체의 시그널 중 적당한 시그널을 사용해 사용자 이벤트를 처리해야 한다는 점입니다.

표 16.1은 QLineEdit 위젯에서 자주 사용되는 시그널을 나타냅니다. textChanged는 사용자의 입력으로 텍스트가 변경되면 발생합니다. returnPressed는 사용자가 텍스트를 입력한 후 QLineEdit 객체에서 엔터 키를 눌렀을 때 발생합니다.

표 16.1 자주 사용되는 QLineEdit 시그널

시그널	시그널 발생 시점
textChanged()	QLineEdit 객체에서 텍스트가 변경될 때
returnPressed()	QLineEdit 객체에서 사용자가 엔터 키를 눌렀을 때

예제 16.7에서는 사용자가 종목 코드를 입력하는 순간에 시그널을 발생시키기 위해 textChanged를 사용했습니다. 다음과 같이 textChanged라는 시그널과 self.lineEditChanged라는 슬롯을 연결하면 사용자가 종목 코드를 입력할 때 self.lineEditChanged라는 메서드가 계속해서 호출됩니다.

```
self.lineEdit = QLineEdit("", self)
self.lineEdit.move(80, 20)
self.lineEdit.textChanged.connect(self.lineEditChanged)
```

사용자가 입력한 종목 코드를 다시 사용자에게 보여주기 위한 용도로 QStatusBar 위젯을 생성했습니다.

```
# StatusBar
self.statusBar = QStatusBar(self)
self.setStatusBar(self.statusBar)
```

lineEditChanged 메서드에서는 self.statusBar라는 변수로 showMessage 메서드를 호출함으로써 상태 표시줄에 텍스트를 출력할 수 있습니다. 이때 출력될 문자열은 사용자가 QLineEdit 객체에 입력한 텍스트입니다. QLineEdit 객체에 입력된 문자열을 구하기 위해 self.lineEdit 변수를 통해 QLineEdit 객체에 접근한 후 text 메서드를 통해 사용자가 입력한 문자열을 얻어옵니다.

```
def lineEditChanged(self):
    self.statusBar.showMessage(self.lineEdit.text())
```

QRadioButton과 QGroupBox

이번 절에서는 QRadioButton과 QGroupBox 위젯에 대해 알아보겠습니다. QRadioButton은 사용자로부터 여러 가지 옵션 중 하나를 입력받을 때 주로 사용하며, QGroupBox는 제목이 있는 네모 박스 형태의 경계선을 만드는 데 사용합니다.

그림 16.29에서 일봉, 주봉, 월봉이 QRadioButton 위젯 객체에 해당하고 '시간 단위'로 표시된 네모 박스가 QGroupBox 위젯 객체입니다. 보통 그림 16.29와 같이 QRadioButton과 QGroupBox 위젯을 동시에 사용하는 경우가 많습니다. 그림 16.29의 프로그램은 사용자가 QRadioButton 위젯 중 하나를 선택할 때 QStatusBar 객체에 텍스트를 출력합니다.



그림 16.29 QRadioButton과 QGroupBox 사용 예

예제 16.8은 그림 16.29에 나온 프로그램의 전체 코드입니다.

예제 16.8 QRadioButton, QGroupBox 사용 예(소스코드: book/ch16/10.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)

        groupBox = QGroupBox("시간 단위", self)
        groupBox.move(10, 10)
        groupBox.resize(280, 80)

        self.radio1 = QRadioButton("일봉", self)
```

```
        self.radio1.move(20, 20)
        self.radio1.setChecked(True)
        self.radio1.clicked.connect(self.radioButtonClicked)

        self.radio2 = QRadioButton("주봉", self)
        self.radio2.move(20, 40)
        self.radio2.clicked.connect(self.radioButtonClicked)

        self.radio3 = QRadioButton("월봉", self)
        self.radio3.move(20, 60)
        self.radio3.clicked.connect(self.radioButtonClicked)

        self.statusBar = QStatusBar(self)
        self.setStatusBar(self.statusBar)

    def radioButtonClicked(self):
        msg = ""
        if self.radio1.isChecked():
            msg = "일봉"
        elif self.radio2.isChecked():
            msg = "주봉"
        else:
            msg = "월봉"
        self.statusBar.showMessage(msg + "선택 됨")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

그림 16.29의 UI를 구성하는 코드부터 살펴보겠습니다. 먼저 QGroupBox 위젯을 이용해 제목이 '시간 단위'인 네모 박스를 만듭니다. 이때 출력 위치를 설정하기 위해 move 메서드를 호출했고 크기를 조절하기 위해 resize 메서드를 호출했습니다. 참고로 resize 메서드의 첫 번째 인자는 위젯의 너비(width)이고 두 번째 인자는 높이(height)입니다.

```
groupBox = QGroupBox("시간 단위", self)
groupBox.move(10, 10)
groupBox.resize(280, 80)
```

라디오 버튼도 다른 위젯과 비슷하게 QRadioButton 클래스의 생성자를 호출함으로써 객체를 생성할 수 있습니다. 첫 번째 인자로 출력될 문자열을 전달하고, 두 번째 인자로 위젯이 출력될 부모 위젯(parent widget)을 전달합니다.

라디오 버튼 역시 move 메서드를 통해 위젯의 출력 위치를 설정할 수 있습니다. setChecked 메서드는 라디오 버튼의 초기 상태를 설정할 수 있습니다. True로 설정하는 경우 해당 라디오 버튼은 선택된 상태로 출력됩니다.

```
self.radio1 = QRadioButton("일봉", self)
self.radio1.move(20, 20)
self.radio1.setChecked(True)
```

라디오 버튼도 'clicked'라는 시그널을 가집니다. 따라서 라디오 버튼이 클릭됐을 때 호출되는 슬롯을 설정할 수 있습니다.

```
self.radio1.clicked.connect(self.radioButtonClicked)
```

그림 16.29에는 총 3개의 라디오 버튼이 있는데, 세 개의 라디오 버튼 객체는 모두 self.radioButtonClicked라는 동일한 슬롯을 사용하고 있습니다. 따라서 사용자가 세 개의 라디오 버튼 중 하나를 클릭하면 self.radioButtonClicked라는 슬롯이 호출됩니다.

self.radioButtonClicked 슬롯에는 사용자가 '일봉', '주봉', '월봉'에 해당하는 라디오 버튼을 클릭하면 선택된 시간 단위가 QStatusBar 객체에 출력되도록 구현합니다. QStatusBar에 텍스트를 출력하는 데는 showMessage 메서드를 사용했습니다.

세 개의 라디오 버튼 모두가 self.radioButtonClicked라는 슬롯과 연결돼 있으므로 슬롯이 호출될 때 각 라디오 버튼 객체에 대해 isChecked 메서드를 호출해 어떤 라디오 버튼이 선택됐는지 확인해야 합니다.

예를 들어, 사용자가 '일봉'으로 표시된 라디오 버튼을 선택하는 경우 해당 라디오 버튼 객체를 바인딩하고 있는 self.radio1의 isChecked 메서드 호출의 결과가 True를 리턴합니다. 따라서 다음과 같은 if 문을 사용해 어떤 라디오 버튼을 선택했는지 확인할 수 있습니다.

```
def radioButtonClicked(self):
    msg = ""
    if self.radio1.isChecked():
        msg = "일봉"
    elif self.radio2.isChecked():
        msg = "주봉"
```

```
        else:
            msg = "월봉"
        self.statusBar.showMessage(msg + "선택 됨")
```

QCheckBox

이번 절에서는 QRadioButton과 비슷하게 사용자로부터 옵션을 선택받을 때 사용하는 QCheckBox 위젯을 알아보겠습니다. QCheckBox는 여러 옵션 중 하나만 선택할 수 있었던 QRadioButton과 달리 여러 옵션을 동시에 선택할 수도 있습니다.

그림 16.30은 QCheckBox를 사용해 이동평균선에 대한 옵션을 선택하는 프로그램입니다. 사용자는 '5일 이동평균선', '20일 이동평균선', '60일 이동평균선'이라는 옵션에 대해 동시에 선택할 수 있습니다. QCheckBox 위젯과 QRadioButton 위젯 중 어떤 위젯을 사용할지는 프로그래머가 상황에 맞춰 선택하면 됩니다.



그림 16.30 QCheckBox

예제 16.9는 그림 16.30에 나온 프로그램의 전체 코드입니다.

예제 16.9 QCheckBox 사용 예(소스코드: book/ch16/11.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)

        self.checkBox1 = QCheckBox("5일 이동평균선", self)
```

```
        self.checkBox1.move(10, 20)
        self.checkBox1.resize(150, 30)
        self.checkBox1.stateChanged.connect(self.checkBoxState)

        self.checkBox2 = QCheckBox("20일 이동평균선", self)
        self.checkBox2.move(10, 50)
        self.checkBox2.resize(150, 30)
        self.checkBox2.stateChanged.connect(self.checkBoxState)

        self.checkBox3 = QCheckBox("60일 이동평균선", self)
        self.checkBox3.move(10, 80)
        self.checkBox3.resize(150, 30)
        self.checkBox3.stateChanged.connect(self.checkBoxState)

        self.statusBar = QStatusBar(self)
        self.setStatusBar(self.statusBar)

    def checkBoxState(self):
        msg = ""
        if self.checkBox1.isChecked() == True:
            msg += "5일 "
        if self.checkBox2.isChecked() == True:
            msg += "20일 "
        if self.checkBox3.isChecked() == True:
            msg += "60일 "
        self.statusBar.showMessage(msg)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

CheckBox를 사용하려면 QCheckBox 클래스의 인스턴스를 생성합니다. 인스턴스를 생성할 때는 QCheckBox에 표시될 텍스트와 QCheckBox가 출력될 부모 위젯을 인자로 전달합니다. QCheckBox는 QMainWindow 객체 안에 출력되기 때문에 부모 위젯으로 self를 사용합니다.

QCheckBox 클래스는 QRadioButton 클래스와 마찬가지로 move와 resize라는 메서드를 가집니다. move 메서드를 통해 QCheckBox 위젯이 출력될 위치를 결정할 수 있고, resize를 통해 크기를 지정할 수 있습니다.

```
self.checkBox1 = QCheckBox("5일 이동평균선", self)
self.checkBox1.move(10, 20)
self.checkBox1.resize(150, 30)
```

'5일 이동평균선'이라는 텍스트를 출력하는 QCheckBox 위젯을 self.checkBox1이라는 인스턴스 변수가 바인딩하고 있습니다. 같은 방식으로 '20일 이동평균선'과 '60일 이동평균선'이라는 텍스트를 가진 CheckBox를 생성합니다.

```
self.checkBox2 = QCheckBox("20일 이동평균선", self)
self.checkBox2.move(10, 50)
self.checkBox2.resize(150, 30)

self.checkBox3 = QCheckBox("60일 이동평균선", self)
self.checkBox3.move(10, 80)
self.checkBox3.resize(150, 30)
```

QCheckBox가 선택될 때 어떤 QCheckBox가 선택됐는가에 대한 정보를 출력할 용도로 QStatusBar 위젯도 생성합니다.

```
self.statusBar = QStatusBar(self)
self.setStatusBar(self.statusBar)
```

화면에 출력될 위젯을 모두 생성했으니 이번에는 각 위젯에서 발생하는 이벤트를 처리하기 위해 시그널과 슬롯을 연결하는 작업을 해보겠습니다.

앞서 설명한 것처럼 PyQt의 위젯은 각 위젯의 특성에 맞는 시그널을 갖고 있습니다. QCheckBox는 CheckBox가 선택되지 않은 상태에서 선택되거나 반대로 선택된 상태에서 선택 해제가 된 경우에 stateChanged라는 시그널을 발생시킵니다. 따라서 사용자가 QCheckBox 위젯의 상태를 변경할 때 이를 처리하는 함수인 슬롯을 stateChanged 시그널과 연결하면 됩니다.

```
self.checkBox1.stateChanged.connect(self.checkBoxState)
self.checkBox2.stateChanged.connect(self.checkBoxState)
self.checkBox3.stateChanged.connect(self.checkBoxState)
```

그림 16.30에 나온 3개의 QCheckBox 객체 중 하나라도 상태가 변경되면 checkBoxState 메서드가 호출됩니다. 따라서 checkBoxState 메서드가 호출될 때 어떤 QCheckBox의 상태가 변경되어 이벤트가 발생했는지 알아내야 합니다.

QCheckBox는 QRadioButton과 마찬가지로 isChecked 메서드를 통해 QCheckBox 위젯 객체가 현재 선택된 상태인지 아닌지 확인할 수 있습니다. isChecked 메서드를 이용해 self.checkBox1, self.

checkBox2, self.checkBox3이 바인딩하고 있는 QCheckBox 객체 중 어떤 QCheckBox가 선택됐는지 확인합니다. QCheckBox 위젯은 동시에 여러 개가 선택될 수도 있으므로 if, elif, else 같은 형태가 아니라 각 위젯에 대해 if 문으로 상태를 비교해야 합니다.

```
def checkBoxState(self):
    msg = ""
    if self.checkBox1.isChecked() == True:
        msg += "5일 "
    if self.checkBox2.isChecked() == True:
        msg += "20일 "
    if self.checkBox3.isChecked() == True:
        msg += "60일 "
    self.statusBar.showMessage(msg)
```

QSpinBox

이번 절에서는 사용자로부터 정숫값을 입력받을 때 사용하는 QSpinBox 위젯에 대해 알아보겠습니다. QSpinBox라는 위젯 이름에서 'spin'이라는 영어 단어는 '회전', '돌다' 등의 의미를 가집니다.

그림 16.31은 QSpinBox를 사용한 프로그램입니다. QSpinBox 위젯을 보면 값을 증가시키는 데 사용하는 화살표, 값을 감소시킬 때 사용하는 화살표, 값이 출력되는 부분으로 구성됨을 알 수 있습니다. QSpinBox에는 값을 증가/감소시키기 위한 화살표가 존재하기 때문에 사용자가 값을 입력할 때보다 편리하게 입력할 수 있습니다.



그림 16.31 QSpinBox

설명만으로는 위젯의 특성을 제대로 파악하기 어려우므로 예제 16.10을 먼저 실행해 보기 바랍니다. QSpinBox 위젯을 이용해 정숫값을 입력할 때는 QSpinBox 위젯에 직접 값을 입력하거나 초깃값으로부터 시작해 화살표를 통해 값을 증가시키거나 감소시켜서 입력해도 됩니다.

예제 16.10 QSpinBox 사용 예(소스코드: book/ch16/12.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)

        label = QLabel("매도수량: ", self)
        label.move(10, 20)

        self.spinBox = QSpinBox(self)
        self.spinBox.move(70, 25)
        self.spinBox.resize(80, 22)
        self.spinBox.valueChanged.connect(self.spinBoxChanged)

        self.statusBar = QStatusBar(self)
        self.setStatusBar(self.statusBar)

    def spinBoxChanged(self):
        val = self.spinBox.value()
        msg = '%d 주를 매도합니다.' % (val)
        self.statusBar.showMessage(msg)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

QSpinBox 위젯 동작에 대해 어느 정도 이해되면 지금부터는 위젯을 구성하는 부분과 위젯이 제공하는 시그널에 대해 알아보겠습니다.

QSpinBox 위젯 역시 다른 위젯과 마찬가지로 생성자를 호출해 QSpinBox 클래스에 대한 인스턴스를 생성함으로써 화면에 출력할 수 있습니다. QSpinBox는 QCheckBox와 달리 텍스트를 출력하지는 않기 때

문에 생성자의 인자로 부모 위젯만 전달합니다. 현재 QSpinBox는 QMainWindow라는 위젯 위에 출력돼야 하므로 self라는 값을 생성자의 인자로 전달하면 됩니다.

QSpinBox 위젯의 출력 위치와 크기 조절에는 다른 위젯과 마찬가지로 move와 resize라는 메서드를 사용합니다.

```
self.spinBox = QSpinBox(self)
self.spinBox.move(70, 25)
self.spinBox.resize(80, 22)
```

QSpinBox 위젯을 생성했다면 QSpinBox가 제공하는 시그널에 대해 알아봅시다. QSpinBox는 사용자로부터 값을 입력받기 때문에 값이 변할 때마다 값이 변경됐다는 사실을 프로그램에 알려줘야 합니다. QSpinBox는 valueChanged라는 시그널을 가지는데, 해당 시그널이 바로 QSpinBox의 값이 변경될 때 자동으로 발생하는 시그널입니다.

값이 변경될 때 valueChanged라는 시그널이 발생하므로 해당 시그널과 이를 처리할 슬롯을 연결하면 사용자 이벤트를 처리할 수 있습니다.

```
self.spinBox.valueChanged.connect(self.spinBoxChanged)
```

QSpinBox 위젯 값이 변경될 때 호출되는 메서드인 spinBoxChanged에서는 현재 QSpinBox 위젯 값을 얻어와야 합니다. value 메서드를 호출해서 얻어온 값은 정수 타입이기 때문에 먼저 문자열로 변경한 후 QStatusBar 객체에 출력합니다.

```
def spinBoxChanged(self):
    val = self.spinBox.value()
    msg = '%d 주를 매도합니다.' % (val)
    self.statusBar.showMessage(msg)
```

QSpinBox 클래스는 value 메서드 외에도 몇 가지 유용한 메서드를 제공합니다. 대표적으로 많이 사용되는 메서드에 대해서만 간략히 살펴보겠습니다.

첫 번째는 QSpinBox의 초깃값을 설정하는 것입니다. 예를 들어, 매수 수량의 초깃값을 0으로 설정하는 것보다 10으로 설정해 주는 편이 사용자에게 더 편리할 수 있습니다.

두 번째는 QSpinBox의 화살표를 눌렀을 때 증가 또는 감소하는 값을 설정하는 것입니다. 유가증권시장이나 코스닥시장에서 소형주는 1주 단위로도 거래할 수 있지만 대형주는 10주 단위로 거래할 수도 있습니다. 따라서 QSpinBox의 화살표를 눌렀을 때 증가 또는 감소하는 값을 지정해두면 사용자가 더욱 쉽게 값을 입력할 수 있습니다.

마지막은 QSpinBox를 통해 입력할 수 있는 값의 범위를 지정해주는 것입니다. 매도할 때 0주를 매도하는 것은 불가능하므로 매도 수량의 최솟값은 0보다 클 것입니다. 사용하는 시나리오에 따라서 최솟값과 최댓값을 지정해주면 사용자가 더욱 편리하게 값을 입력할 수 있습니다.

QSpinBox 클래스에서 초깃값을 설정할 때는 setValue 메서드를 사용합니다. QSpinBox에서 화살표를 눌렀을 때 증가 또는 감소하는 값은 setSingleStep 메서드를 사용합니다. 마지막으로 최솟값과 최댓값을 설정할 때는 setMinimum과 setMaximum 메서드를 사용합니다.

예제 16.10에서 QSpinBox와 관련된 코드를 다음과 같이 수정하면 QSpinBox가 10부터 시작해서 10 단위로 증가 또는 감소하는 것을 확인할 수 있습니다. 지금 설명한 부분은 어쩌면 프로그래머에게는 지루한 작업일 수 있습니다. 하지만 이러한 사소한 배려가 프로그램을 사용하는 사용자가 느끼는 UI/UX에 큰 차이를 줍니다.

```
self.spinBox = QSpinBox(self)
self.spinBox.move(70, 25)
self.spinBox.resize(80, 22)

self.spinBox.setValue(10)
self.spinBox.setSingleStep(10)
self.spinBox.setMinimum(1)
self.spinBox.setMaximum(10000)
self.spinBox.valueChanged.connect(self.spinBoxChanged)
```

QTableWidget

이번 절에서는 그림 16.32에서 사용한 QTableWidget에 대해 알아보겠습니다. 일상생활에서 그림 16.32와 같이 행과 열로 구성된 2차원 포맷 형태의 데이터를 많이 사용합니다. 여러분이 자주 사용하는 마이크로소프트의 엑셀도 데이터를 2차원 포맷 형태로 표현하는 대표적인 프로그램입니다. 이처럼 데이터를 행과 열로 표시하면 데이터를 쉽게 파악하는 데 도움이 됩니다.

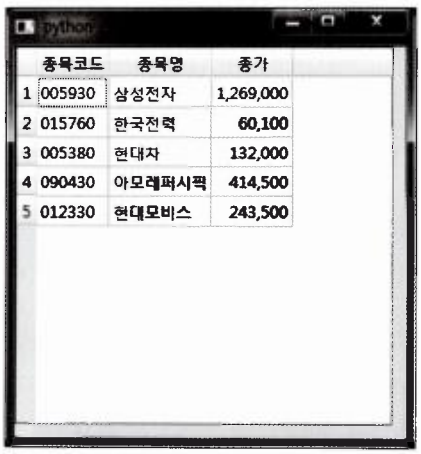


그림 16.32 QTableWidget

먼저 그림 16.33과 같이 간단한 형태의 프로그램을 작성하면서 QTableWidget 클래스의 주요 기능을 파악해 보겠습니다.

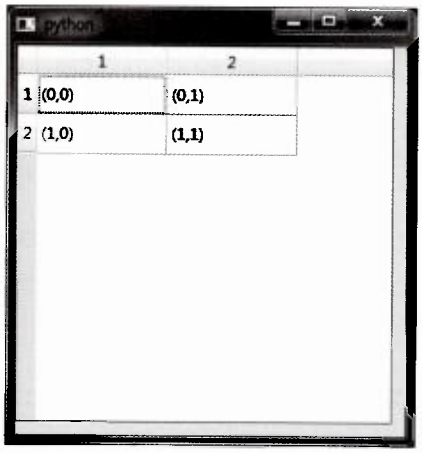


그림 16.33 QTableWidget - 2

예제 16.11은 그림 16.33에 나온 프로그램의 전체 코드입니다.

예제 16.11 QTableWidget 사용 예(소스코드: book/ch16/14.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)

        self.tableWidget = QTableWidget(self)
        self.tableWidget.resize(290, 290)
        self.tableWidget.setRowCount(2)
        self.tableWidget.setColumnCount(2)
        self.setTableWidgetData()

    def setTableWidgetData(self):
        self.tableWidget.setItem(0, 0, QTableWidgetItem("(0,0)"))
        self.tableWidget.setItem(0, 1, QTableWidgetItem("(0,1)"))
        self.tableWidget.setItem(1, 0, QTableWidgetItem("(1,0)"))
        self.tableWidget.setItem(1, 1, QTableWidgetItem("(1,1)"))
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

먼저 setupUI 메서드에서 QTableWidget 클래스의 인스턴스를 생성하는 코드를 살펴보겠습니다. QTableWidget 클래스의 인스턴스를 생성하기 위해 생성자를 호출할 때 QTableWidget 클래스가 표시될 부모 클래스를 지정합니다.

QTableWidget 클래스의 크기는 resize 메서드를 이용해 조절할 수 있습니다. QTableWidget 클래스는 데이터를 출력하기에 앞서 행과 열의 개수를 미리 지정해야 합니다. 행의 개수를 지정할 때는 setRowCount 메서드를 사용하고, 열의 개수를 지정할 때는 setColumnCount 메서드를 사용합니다.

```
self.tableWidget = QTableWidget(self)
self.tableWidget.resize(290, 290)
self.tableWidget.setRowCount(2)
self.tableWidget.setColumnCount(2)
```

2x2 (행 x 열) 크기의 QTableWidget 객체가 생성되면 (0, 0), (0, 1), (1, 0), (1, 1) 위치에 아이템을 넣을 수 있습니다. 각 위치에는 setItem 메서드를 사용해 QTableWidgetItem 객체를 삽입하면 됩니다.

setItem 메서드는 3개의 인자를 받는데 첫 번째 인자는 행에 대한 인덱스이고, 두 번째 인자는 열에 대한 인덱스입니다. 마지막으로 세 번째 인자로는 QTableWidgetItem 객체를 전달합니다.

```
def setTableWidgetData(self):
    self.tableWidget.setItem(0, 0, QTableWidgetItem("(0,0)"))
    self.tableWidget.setItem(0, 1, QTableWidgetItem("(0,1)"))
    self.tableWidget.setItem(1, 0, QTableWidgetItem("(1,0)"))
    self.tableWidget.setItem(1, 1, QTableWidgetItem("(1,1)"))
```

QTableWidget 클래스에 대해 간단히 알아봤으니 이번에는 그림 16.32에 나온 프로그램의 코드를 살펴보겠습니다.

예제 16.12 QTableWidget 사용 예 - 2(소스코드: book/ch16/15.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
```

```
kospi_top5 = {
    'code': ['005930', '015760', '005380', '090430', '012330'],
    'name': ['삼성전자', '한국전력', '현대차', '아모레퍼시픽', '현대모비스'],
    'cprice': ['1,269,000', '60,100', '132,000', '414,500', '243,500']
}
column_idx_lookup = {'code': 0, 'name': 1, 'cprice': 2}

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)

        self.tableWidget = QTableWidget(self)
        self.tableWidget.resize(290, 290)
        self.tableWidget.setRowCount(5)
        self.tableWidget.setColumnCount(3)
        self.tableWidget.setEditTriggers(QAbstractItemView.NoEditTriggers)

        self.setTableWidgetData()

    def setTableWidgetData(self):
        column_headers = ['종목코드', '종목명', '종가']
        self.tableWidget.setHorizontalHeaderLabels(column_headers)

        for k, v in kospi_top5.items():
            col = column_idx_lookup[k]
            for row, val in enumerate(v):
                item = QTableWidgetItem(val)
                if col == 2:
                    item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)

                self.tableWidget.setItem(row, col, item)

        self.tableWidget.resizeColumnsToContents()
        self.tableWidget.resizeRowsToContents()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
```

```
    mywindow.show()
    app.exec_()
```

먼저 UI를 구성하는 코드를 살펴보면 예제 16.11과 크게 다르지는 않습니다. 다만 QTableWidget의 행과 열의 개수가 다르고 setEditTriggers 메서드를 사용해 사용자가 QTableWidget의 아이템 항목을 수정할 수 없도록 설정했습니다.

```
self.tableWidget = QTableWidget(self)
self.tableWidget.resize(290, 290)
self.tableWidget.setRowCount(5)
self.tableWidget.setColumnCount(3)
self.tableWidget.setEditTriggers(QAbstractItemView.NoEditTriggers)
```

QTableWidget에 아이템을 채우는 코드를 살펴보겠습니다. 먼저 column에 대한 라벨을 설정했습니다. 참고로 row 방향에 대한 라벨을 설정할 때는 setVerticalHeaderLabels 메서드를 사용합니다.

```
column_headers = ['종목코드', '종목명', '종가']
self.tableWidget.setHorizontalHeaderLabels(column_headers)
```

5x3 크기의 QTableWidget에 아이템을 채우는 코드는 (행, 열) 위치에 대한 인덱스를 구하고 해당 위치에 문자열을 넣는 방식으로 구현했습니다. 이때 열(column)에 대한 인덱스는 미리 정의된 column_idx_lookup이라는 딕셔너리를 사용해 계산합니다.

QTableWidget에 아이템으로 삽입하려면 먼저 데이터를 QTableWidgetItem 객체로 만들어야 합니다. 그런 다음 setItem 메서드를 이용해 아이템을 원하는 위치에 삽입하면 됩니다. 이때 '종가'에 해당하는 열 데이터는 setTextAlignment 메서드를 사용해 우측 정렬했습니다.

```
for k, v in kospi_top5.items():
    col = column_idx_lookup[k]
    for row, val in enumerate(v):
        item = QTableWidgetItem(val)
        if col == 2:
            item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)
        self.tableWidget.setItem(row, col, item)
```

마지막으로 QTableWidget의 행과 열 크기를 각 위치에 저장된 아이템 길이에 맞춰 조정합니다.

```
self.tableWidget.resizeColumnsToContents()
self.tableWidget.resizeRowsToContents()
```

04 Layout

GUI 프로그래밍에서 위젯을 배치하는 것을 레이아웃(layout)이라고 합니다. 앞서 여러분이 작성했던 코드에서는 resize 메서드를 이용해 위젯의 크기를 변경했고, move나 setGeometry 메서드를 이용해 위젯의 출력 위치를 설정했습니다.

그림 16.34는 resize와 move 메서드로 직접 위치와 크기를 설정해 레이아웃을 조정한 프로그램의 예입니다.



그림 16.34 위치 및 크기 지정을 통한 레이아웃 설정

예제 16.13은 그림 16.34에 나온 프로그램의 전체 코드입니다. QTextEdit와 QPushButton 위젯의 위치와 크기를 직접 설정하는 것을 확인할 수 있습니다. 참고로 MyWindow 클래스를 정의할 때 QMainWindow 클래스 대신 QWidget 클래스를 상속했습니다.

예제 16.13 위치 및 크기 조절을 통한 레이아웃 설정(소스코드: book/ch16/16.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)

        self.textEdit = QTextEdit(self)
        self.textEdit.resize(280, 250)
        self.textEdit.move(10, 10)

        self.pushButton= QPushButton('저장', self)
        self.pushButton.resize(280, 25)
```

```
        self.pushButton.move(10, 270)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

예제 16.13과 같은 방식으로 레이아웃을 구성하면 사용자가 윈도우의 크기를 변경할 때 문제가 발생합니다. 예를 들어, 그림 16.35는 사용자가 윈도우를 오른쪽 아래 방향으로 드래그해서 크기를 변경한 것입니다. 윈도우 크기는 변경됐지만 윈도우 내부에 존재하는 위젯의 출력 위치 및 크기는 변경되지 않기 때문에 전체적인 UI 구성이 어색해 보입니다.

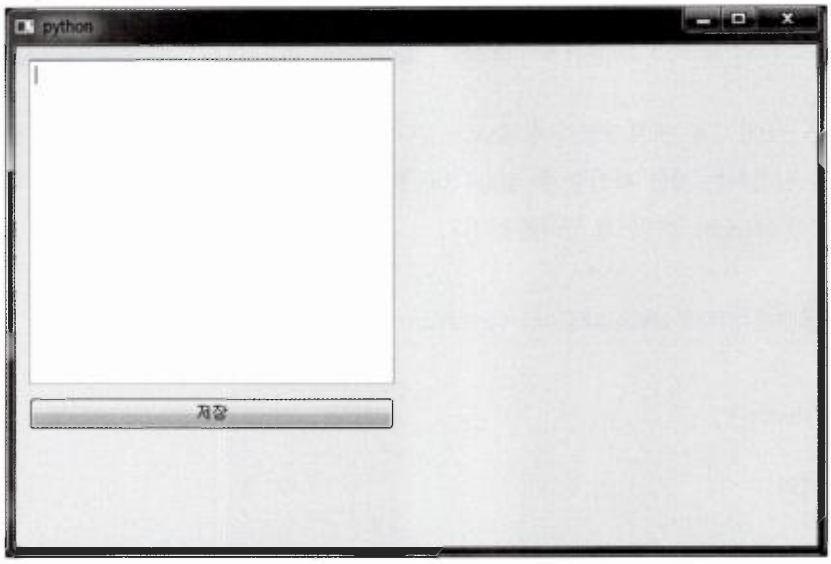


그림 16.35 잘못된 레이아웃의 예

이러한 문제는 PyQt에서만 있는 것이 아니고 GUI 프로그래밍에서 공통으로 발생하는 문제입니다. 이러한 문제를 해결하는 고전적인 방법은 윈도우의 크기가 변경될 때 발생하는 시그널과 이를 처리하는 슬롯을 연결한 후, 해당 슬롯 내에서 윈도우 내부의 위젯의 크기와 위치를 다시 계산한 후 재설정하는 것입니다. 그러나 이러한 방식은 프로그래머에게 매우 귀찮은 작업입니다.

QVBoxLayout

앞서 위젯의 크기와 출력 위치를 명시적으로 설정하는 방법은 윈도우 크기가 바뀔 때 문제가 있음을 확인했습니다. PyQt에서는 이러한 문제를 쉽게 해결하고자 레이아웃 매니저를 제공합니다. PyQt는 레이아웃 매니저로 QVBoxLayout, QHBoxLayout, QBoxLayout, QGridLayout, QLayout이라는 다섯 가지 클래스를 제공합니다. 이번 절에서는 QVBoxLayout 클래스에 대해 먼저 알아보겠습니다.

QVBoxLayout 클래스는 이름을 통해 짐작할 수 있듯이 위젯을 수직 방향으로 나열합니다. 그림 16.34를 참조하면 QTextEdit 위젯이 윈도우 상단에 위치하고 QPushButton 위젯이 QTextEdit 아래쪽에 위치합니다. 이러한 형태의 레이아웃일 때 QVBoxLayout 클래스를 사용합니다. 참고로 QVBoxLayout을 사용하면 앞에서 살펴본 위젯의 크기 변경 문제가 자동으로 해결됩니다.

예제 16.14는 QVBoxLayout 클래스를 이용해 그림 16.34에 나온 프로그램의 레이아웃을 조정한 것입니다. 예제 16.13과 달리 QTextEdit와 QPushButton 위젯을 생성할 때 부모 위젯을 명시적으로 전달하지 않으며, resize나 move 메서드도 호출하지 않는 것을 확인할 수 있습니다. 그 대신 QTextEdit와 QPushButton 위젯을 QVBoxLayout 객체에 추가한 후 현재 윈도우의 레이아웃으로 QVBoxLayout 객체로 설정했습니다.

예제 16.14 QVBoxLayout을 통한 레이아웃 구성(소스코드: book/ch16/17.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)

        self.textEdit = QTextEdit()
        self.pushButton= QPushButton('저장')

        layout = QVBoxLayout()
        layout.addWidget(self.textEdit)
        layout.addWidget(self.pushButton)
```

```
        self.setLayout(layout)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

예제 16.14를 실행한 후 화면에 출력된 윈도우의 크기를 변경해 보기 바랍니다. 이번에는 그림 16.36과 같이 윈도우의 크기를 자유롭게 변경해도 내부 위젯이 일정한 비율을 가지며 크기가 자동으로 바뀌는 것을 확인할 수 있습니다.



그림 16.36 QVBoxLayout을 이용한 레이아웃 구성

QHBoxLayout

이번 절에서 다룰 QHBoxLayout은 행(row) 방향으로 위젯을 배치할 때 사용하는 레이아웃 매니저입니다. QHBoxLayout을 이용하면 그림 16.37과 같은 형태의 UI를 효과적으로 구성할 수 있습니다.



그림 16.37 QHBoxLayout을 이용한 레이아웃 구성

예제 16.15는 그림 16.37에 나온 프로그램의 전체 코드입니다.

예제 16.15 QHBoxLayout을 이용한 레이아웃 구성(소스코드: book/ch16/18.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 100)

        self.pushButton1= QPushButton("Button1")
        self.pushButton2= QPushButton("Button2")
        self.pushButton3= QPushButton("Button3")

        layout = QHBoxLayout()
        layout.addWidget(self.pushButton1)
        layout.addWidget(self.pushButton2)
        layout.addWidget(self.pushButton3)

        self.setLayout(layout)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

예제 16.15를 살펴보면 1) 위젯 만들기 2) 레이아웃 매니저 만들기 3) addWidget 메서드를 이용한 위젯 등록 4) 레이아웃 매니저 등록 단계로 구성돼 있습니다. 이는 앞에서 배운 QVBoxLayout과 동일하며 앞으로 다룰 QGridLayout도 같습니다.

레이아웃 매니저를 사용할 때 주의할 점은 레이아웃 매니저에 추가되는 위젯을 생성할 때는 부모 위젯을 지정할 필요가 없으며, 위젯의 크기나 위치를 명시적으로 설정하지 않아야 한다는 점입니다.

QGridLayout

QGridLayout 클래스는 격자 형태의 UI를 구성하는 데 사용합니다. 예를 들어, 그림 16.38에서는 2x3(행x열) 크기의 격자 구조에 QLabel, QLineEdit, QPushButton 위젯을 배치했습니다. QGridLayout은 QHBoxLayout이나 QVBoxLayout과 달리 격자에서 위젯을 배치할 좌표를 입력받습니다.



그림 16.38 QGridLayout을 이용한 레이아웃 구성

예제 16.16은 그림 16.38에 나온 프로그램의 전체 코드입니다. 앞에서 여러 번 설명한 것처럼 코드의 구성은 1) 위젯 만들기 2) 레이아웃 매니저 만들기 3) addWidget 메서드를 이용한 위젯 등록 4) 레이아웃 매니저 등록 단계로 구성됩니다. 다만 QGridLayout에서는 addWidget 메서드에서 행과 열의 인덱스를 차례로 입력받습니다.

예제 16.16 QGridLayout을 이용한 레이아웃 구성(소스코드: book/ch16/19.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 100)
```

```
        self.label1 = QLabel("ID: ")
        self.label2 = QLabel("Password: ")
        self.lineEdit1 = QLineEdit()
        self.lineEdit2 = QLineEdit()
        self.pushButton1= QPushButton("Sign In")

        layout = QGridLayout()

        layout.addWidget(self.label1, 0, 0)
        layout.addWidget(self.lineEdit1, 0, 1)
        layout.addWidget(self.pushButton1, 0, 2)

        layout.addWidget(self.label2, 1, 0)
        layout.addWidget(self.lineEdit2, 1, 1)

        self.setLayout(layout)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

그림 16.39는 그림 16.38에서 사용한 2x3(행x열) 격자를 그림으로 표현한 것입니다. 각 행과 열에 대해 0부터 시작하는 정숫값을 통해 인덱싱하면 됩니다.

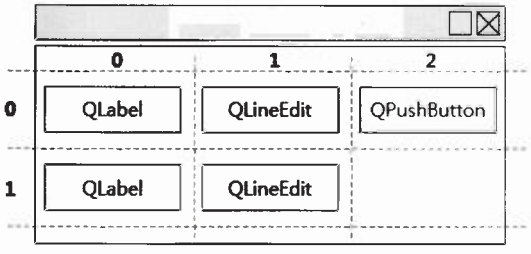


그림 16.39 QGridLayout에서의 격자 셀

레이아웃 중첩

이번 절에서는 QVBoxLayout과 QHBoxLayout을 중첩해서 그림 16.40과 같이 조금 복잡한 형태의 레이아웃을 구성해 보겠습니다. 레이아웃 매니저 클래스에서 addWidget 메서드를 통해 레이아웃에 위젯을 추가할 수 있는 것처럼, addLayout 메서드를 호출하면 레이아웃에 다른 레이아웃 매니저를 추가할 수 있습니다.

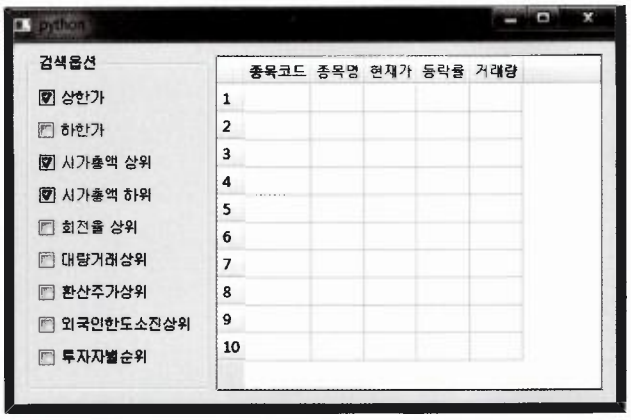


그림 16.40 레이아웃 중첩

그림 16.41은 그림 16.40의 프로그램에 대한 대략적인 레이아웃 구성입니다. 가장 바깥쪽에는 QHBoxLayout이 위치합니다. QHBoxLayout 내부에는 두 개의 QVBoxLayout이 존재하는데 하나는 윈도우의 왼쪽 부분에, 나머지 하나는 윈도우의 오른쪽 부분에 자동으로 배치됩니다. 이처럼 레이아웃 매니저는 다른 레이아웃 매니저를 포함할 수도 있으며 이러한 중첩 구조를 이용하면 복잡한 레이아웃도 구성할 수 있습니다.

왼쪽의 QVBoxLayout에는 QGroupBox와 QCheckBox 위젯이 추가되고, 오른쪽의 QVBoxLayout에는 QTableWidget 객체가 추가됩니다. 왼쪽의 QVBoxLayout에서는 QCheckBox 위젯을 addWidget 메서드를 사용해 직접 추가하는 것이 아니라 QGroupBox 위젯이 QCheckBox를 포함하는 형태로 구성했습니다.

QGroupBox 내부에는 9개의 QCheckBox가 존재하는데, 칼럼 방향으로 배치하기 위해 또 다른 QVBoxLayout (2)를 사용했습니다. 결과적으로 QCheckBox 위젯들은 내부의 QVBoxLayout (2) 안에 존재하고 QGroupBox는 QVBoxLayout (2)를 포함합니다.

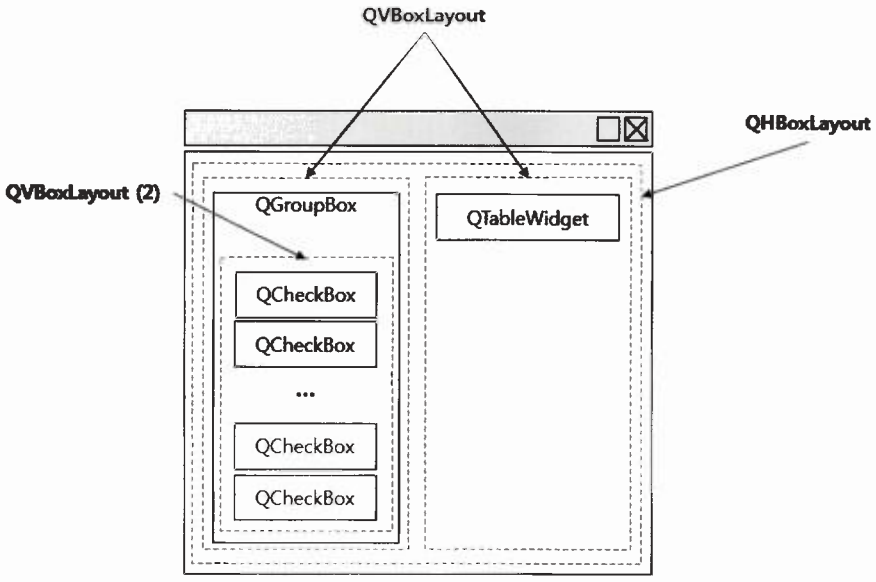


그림 16.41 레이아웃 중첩을 통한 UI 구성

예제 16.17은 그림 16.40에 나온 프로그램의 전체 코드입니다.

예제 16.17 레이아웃 중첩을 통한 UI 구성(소스코드: book/ch16/20.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 500, 300)

        groupBox = QGroupBox("검색옵션")
        checkBox1 = QCheckBox("상한가")
        checkBox2 = QCheckBox("하한가")
        checkBox3 = QCheckBox("시가총액 상위")
        checkBox4 = QCheckBox("시가총액 하위")
        checkBox5 = QCheckBox("회전율 상위")
        checkBox6 = QCheckBox("대량거래상위")
```

```
        checkBox7 = QCheckBox("환산주가상위")
        checkBox8 = QCheckBox("외국인한도소진상위")
        checkBox9 = QCheckBox("투자자별순위")

        tableWidget = QTableWidget(10, 5)
        tableWidget.setHorizontalHeaderLabels(["종목코드", "종목명", "현재가", "등락률", "거래
량"])
        tableWidget.resizeColumnsToContents()
        tableWidget.resizeRowsToContents()

        leftInnerLayOut = QVBoxLayout()
        leftInnerLayOut.addWidget(checkBox1)
        leftInnerLayOut.addWidget(checkBox2)
        leftInnerLayOut.addWidget(checkBox3)
        leftInnerLayOut.addWidget(checkBox4)
        leftInnerLayOut.addWidget(checkBox5)
        leftInnerLayOut.addWidget(checkBox6)
        leftInnerLayOut.addWidget(checkBox7)
        leftInnerLayOut.addWidget(checkBox8)
        leftInnerLayOut.addWidget(checkBox9)
        groupBox.setLayout(leftInnerLayOut)

        leftLayOut = QVBoxLayout()
        leftLayOut.addWidget(groupBox)

        rightLayOut = QVBoxLayout()
        rightLayOut.addWidget(tableWidget)

        layout = QHBoxLayout()
        layout.addLayout(leftLayOut)
        layout.addLayout(rightLayOut)

        self.setLayout(layout)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

그림 16.40에서 윈도우 왼쪽에는 QGroupBox와 QCheckBox 위젯이 위치합니다. 먼저 사용할 위젯을 생성합니다. 앞서 QGroupBox와 QCheckBox 위젯을 생성했던 것과 달리 크기나 좌표를 설정하지 않습니다.

```
groupBox = QGroupBox("검색옵션")
checkBox1 = QCheckBox("상한가")
checkBox2 = QCheckBox("하한가")
checkBox3 = QCheckBox("시가총액 상위")
checkBox4 = QCheckBox("시가총액 하위")
checkBox5 = QCheckBox("회전율 상위")
checkBox6 = QCheckBox("대량거래상위")
checkBox7 = QCheckBox("환산주가상위")
checkBox8 = QCheckBox("외국인한도소진상위")
checkBox9 = QCheckBox("투자자별순위")
```

QGroupBox에 여러 개의 QCheckBox를 추가할 때 단순히 위젯을 추가하는 것이 아니라 먼저 QVBoxLayout을 사용해 QCheckBox 위젯을 추가합니다.

```
leftInnerLayOut = QVBoxLayout()
leftInnerLayOut.addWidget(checkBox1)
leftInnerLayOut.addWidget(checkBox2)
leftInnerLayOut.addWidget(checkBox3)
leftInnerLayOut.addWidget(checkBox4)
leftInnerLayOut.addWidget(checkBox5)
leftInnerLayOut.addWidget(checkBox6)
leftInnerLayOut.addWidget(checkBox7)
leftInnerLayOut.addWidget(checkBox8)
leftInnerLayOut.addWidget(checkBox9)
```

QVBoxLayout에는 QCheckBox 위젯이 포함돼 있으므로 QGroupBox에 QVBoxLayout 객체만 추가하면 QCheckBox를 QGroupBox 안쪽으로 배치할 수 있습니다.

```
groupBox.setLayout(leftInnerLayOut)
```

QGroupBox를 윈도우의 왼쪽에 배치하기 위해 다음과 같이 QVBoxLayout을 사용해 위젯을 추가합니다.

```
leftLayOut = QVBoxLayout()
leftLayOut.addWidget(groupBox)
```

그림 16.40에서 윈도우의 오른쪽에는 QTableWidget이 존재합니다. 먼저 QTableWidget 객체를 생성한 후 칼럼의 라벨을 설정합니다.

```
tableWidget = QTableWidget(10, 5)
tableWidget.setHorizontalHeaderLabels(["종목코드", "종목명", "현재가", "등락률", "거래량"])
tableWidget.resizeColumnsToContents()
tableWidget.resizeRowsToContents()
```

QTableWidget 객체를 윈도우의 오른쪽에 위치시키기 위해 먼저 QVBoxLayout 클래스의 객체를 생성한 후 QTableWidget 객체를 추가합니다.

```
rightLayOut = QVBoxLayout()
rightLayOut.addWidget(tableWidget)
```

마지막으로 leftLayout과 rightLayout을 윈도우의 왼쪽과 오른쪽에 배치하기 위해 QHBoxLayout 클래스의 객체를 생성한 후 레이아웃을 추가합니다.

```
layout = QHBoxLayout()
layout.addLayout(leftLayOut)
layout.addLayout(rightLayOut)
```

이것으로 UI 구성을 모두 마쳤습니다. 현재 윈도우에 레이아웃을 layout 객체로 설정합니다.

```
self.setLayout(layout)
```

05 다이얼로그

다이얼로그(Dialog)는 '대화'라는 뜻을 가진 영어 단어입니다. GUI 프로그래밍에서 다이얼로그는 사용자와의 상호작용을 위해 사용되는 윈도우를 의미합니다. 예를 들어, 그림 16.42는 윈도우와 로그인에 사용되는 다이얼로그입니다.

사용자가 메인 윈도우에서 [Sign In] 버튼을 클릭하면 로그인을 위한 새로운 다이얼로그가 생성됩니다. 사용자가 다이얼로그 윈도우에서 아이디와 패스워드를 입력하면 해당 정보가 메인 윈도우에 출력됩니다.

대부분의 GUI 프로그램은 그림 16.42와 같이 메인 윈도우와 다수의 다이얼로그로 구성됩니다. PyQt는 다양한 형태의 Dialog 클래스를 지원하므로 여러분은 용도에 맞는 다이얼로그 클래스를 사용하거나 상속을 받은 후 새로운 클래스를 정의해서 사용하면 됩니다.

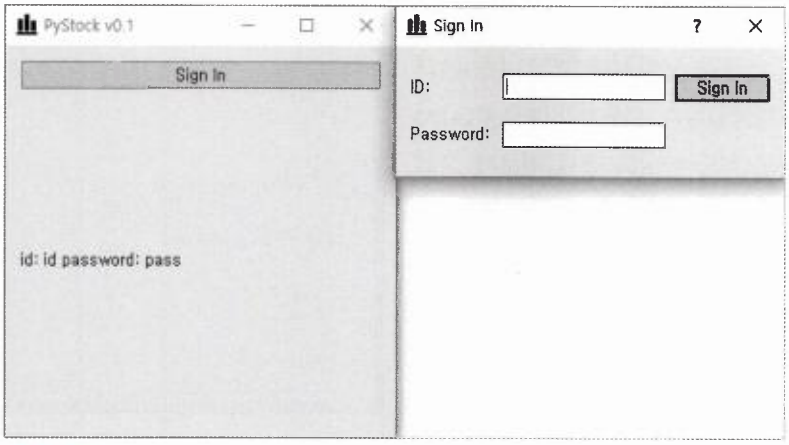


그림 16.42 윈도우와 다이얼로그

QFileDialog

이번 절에서는 PyQt가 제공하는 QFileDialog 클래스에 대해 알아보겠습니다. QFileDialog는 그림 16.43과 같이 사용자가 파일이나 디렉터리를 선택할 수 있게 하는 다이얼로그 창입니다. 이러한 기능의 창은 자주 사용되기 때문에 PyQt는 QFileDialog 클래스를 통해 이러한 기능을 지원하고 있습니다.

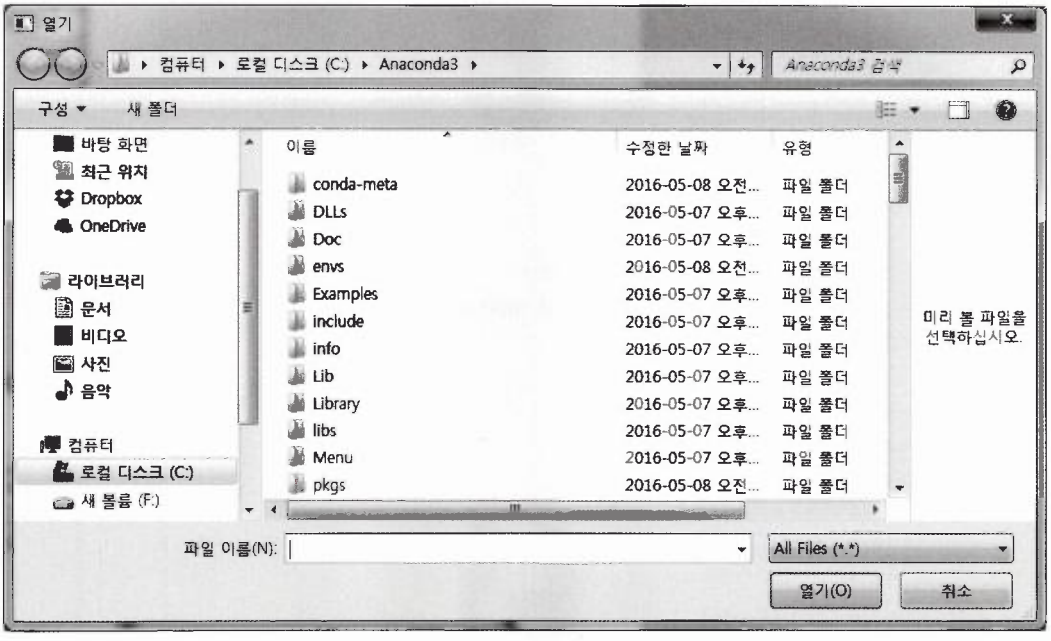


그림 16.43 QFileDialog

그림 16.44는 QFileDialog를 사용한 프로그램입니다. 사용자가 [File Open] 버튼을 클릭하면 그림 16.43의 QFileDialog 창이 화면에 나타납니다. 사용자가 QFileDialog 창을 통해 파일을 선택하면 해당 파일의 절대 경로가 그림 16.44와 같이 메인 윈도우에 출력됩니다.

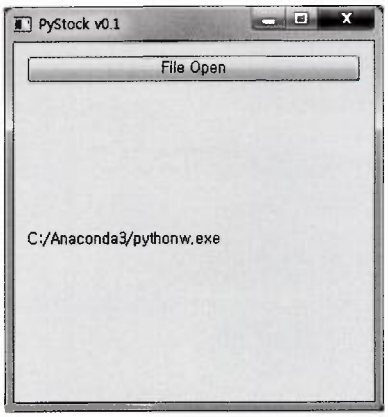


그림 16.44 QFileDialog를 사용한 프로그램

예제 16.18은 그림 16.44에 나온 프로그램의 전체 코드입니다.

예제 16.18 QFileDialog 사용 예(소스코드: book/ch16/21.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)
        self.setWindowTitle("PyStock v0.1")

        self.pushButton = QPushButton("File Open")
        self.pushButton.clicked.connect(self.pushButtonClicked)
        self.label = QLabel()

        layout = QVBoxLayout()
        layout.addWidget(self.pushButton)
        layout.addWidget(self.label)

        self.setLayout(layout)

    def pushButtonClicked(self):
        fname = QFileDialog.getOpenFileName(self)
```

```
        self.label.setText(fname[0])

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyWindow()
    window.show()
    app.exec_()
```

그림 16.44에서 버튼을 클릭하면 pushButtonClicked라는 슬롯이 호출됩니다. pushButtonClicked 구현을 살펴보면 QFileDialog 클래스의 getOpenFileName 메서드를 호출합니다. getOpenFileName 메서드가 호출되면 그림 16.43과 같은 QFileDialog 창이 화면에 출력되고, 사용자가 파일을 선택한 후 [열기] 버튼을 클릭하면 해당 파일의 절대 경로가 반환되어 fname에 저장됩니다.

```
def pushButtonClicked(self):
fname = QFileDialog.getOpenFileName(self)
```

반환된 파일 경로는 튜플 타입으로 fname이라는 변수가 바인딩하고 있습니다. self.label 객체의 setText 메서드를 통해 튜플의 0번째 값을 화면에 출력합니다.

```
self.label.setText(fname[0])
```

QInputDialog

QInputDialog는 그림 16.45와 같이 사용자로부터 간단한 텍스트, 정수, 실수를 받을 때 사용하는 다이얼로그입니다. QInputDialog의 구성을 살펴보면 상단에는 QLabel 객체가 있고 다음으로 사용자로부터 입력을 받기 위한 QLineEdit 객체가 존재합니다. 마지막으로 OK와 Cancel 버튼이 있습니다.



그림 16.45 QInputDialog를 사용하는 프로그램

예제 16.19는 그림 16.45에 나온 프로그램의 전체 코드입니다.

예제 16.19 QInputDialog 사용 예(소스코드: book/ch16/22.py)

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)
        self.setWindowTitle("PyStock v0.1")

        self.pushButton = QPushButton("Input number")
        self.pushButton.clicked.connect(self.pushButtonClicked)
        self.label = QLabel()

        layout = QVBoxLayout()
        layout.addWidget(self.pushButton)
        layout.addWidget(self.label)

        self.setLayout(layout)

    def pushButtonClicked(self):
        text, ok = QInputDialog.getInt(self, '매수 수량', '매수 수량을 입력하세요.')
        if ok:
            self.label.setText(str(text))

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyWindow()
    window.show()
    app.exec_()
```

예제 16.19에서 pushButtonClicked 메서드를 살펴보면 QInputDialog 클래스의 getInt 메서드를 호출합니다. getInt 메서드의 첫 번째 인자는 부모 위젯을 의미하고, 두 번째 인자는 QInputDialog 창에 표시될 텍스트입니다. 세 번째 인자는 QInputDialog 창 내부에 출력될 텍스트입니다.

getInt 메서드는 (text, ok)라는 튜플 형태로 값을 반환합니다. text에는 사용자가 입력한 값이 반환되고, ok는 사용자가 OK 버튼을 누른 경우에 True 값을 가집니다.

```
def pushButtonClicked(self):
    text, ok = QInputDialog.getInt(self, '매수 수량', '매수 수량을 입력하세요.')
```

사용자가 QInputDialog에서 OK 버튼을 입력했다면 사용자로부터 입력받은 값을 먼저 문자열로 변경한 후 self.label 객체에 출력합니다.

```
if ok:
    self.label.setText(str(text))
```

QInputDialog 클래스는 getDouble, getText, getItem이라는 메서드도 갖고 있습니다. 사용자로부터 실수를 입력받을 때는 getInt 대신 getDouble을 사용하고, 문자열을 입력받을 때는 getText를 사용합니다.

사용자로부터 단순한 텍스트, 정수, 실수를 입력받는 것이 아니라 그림 16.46과 같이 다양한 옵션 중 하나를 선택받고자 한다면 getItem 메서드를 사용합니다.

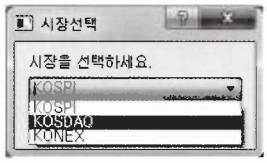


그림 16.46 QInputDialog.getItem() 메서드를 이용한 다이얼로그

getItem 메서드를 사용하는 경우 pushButtonClicked 메서드를 다음과 같이 구현합니다. getItem 메서드의 첫 번째 인자는 부모 위젯, 두 번째는 다이얼로그의 타이틀 텍스트, 세 번째는 다이얼로그 내부의 텍스트, 네 번째 인자는 선택할 아이템 리스트, 다섯 번째 인자는 초기 아이템 인덱스, 여섯 번째 인자는 아이템 수정 가능 여부입니다.

```
def pushButtonClicked(self):
    items = ("KOSPI", "KOSDAQ", "KONEX")
    item, ok = QInputDialog.getItem(self, "시장선택", "시장을 선택하세요.", items, 0, False)
    if ok and item:
        self.label.setText(item)
```

메인 윈도우와 다이얼로그의 상호작용

이번 절에서는 그림 16.42의 프로그램을 만들어 보겠습니다. 그림 16.42는 한 개의 메인 윈도우와 다이얼로그로 구성됩니다. 사용자가 다이얼로그에서 정보를 입력하면 해당 정보가 메인 윈도우 내의 위젯에 출력됩니다.

예제 16.20은 그림 16.49에 나온 프로그램의 전체 코드입니다. 먼저 코드를 실행한 후 프로그램의 동작 방식을 파악해 보기 바랍니다. 아이콘 파일이 없는 경우에는 setWindowIcon 메서드를 호출하는 코드를 주석으로 처리하면 됩니다.

예제 16.20 메인 윈도우와 다이얼로그의 상호작용(소스코드: book/ch16/24.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *

class LogInDialog(QDialog):
    def __init__(self):
        super().__init__()
        self.setupUI()

        self.id = None
        self.password = None

    def setupUI(self):
        self.setGeometry(1100, 200, 300, 100)
        self.setWindowTitle("Sign In")
        self.setWindowIcon(QIcon('icon.png'))

        label1 = QLabel("ID: ")
        label2 = QLabel("Password: ")

        self.lineEdit1 = QLineEdit()
        self.lineEdit2 = QLineEdit()
        self.pushButton1= QPushButton("Sign In")
        self.pushButton1.clicked.connect(self.pushButtonClicked)

        layout = QGridLayout()
        layout.addWidget(label1, 0, 0)
        layout.addWidget(self.lineEdit1, 0, 1)
        layout.addWidget(self.pushButton1, 0, 2)
        layout.addWidget(label2, 1, 0)
        layout.addWidget(self.lineEdit2, 1, 1)

        self.setLayout(layout)

    def pushButtonClicked(self):
        self.id = self.lineEdit1.text()
        self.password = self.lineEdit2.text()
        self.close()
```

```
class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(800, 200, 300, 300)
        self.setWindowTitle("PyStock v0.1")
        self.setWindowIcon(QIcon('icon.png'))

        self.pushButton = QPushButton("Sign In")
        self.pushButton.clicked.connect(self.pushButtonClicked)
        self.label = QLabel()

        layout = QVBoxLayout()
        layout.addWidget(self.pushButton)
        layout.addWidget(self.label)

        self.setLayout(layout)

    def pushButtonClicked(self):
        dlg = LogInDialog()
        dlg.exec_()
        id = dlg.id
        password = dlg.password
        self.label.setText("id: %s password: %s" % (id, password))

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyWindow()
    window.show()
    app.exec_()
```

예제 16.20에는 두 개의 클래스가 정의돼 있습니다. MyWindow 클래스는 QWidget 클래스를 상속받아 정의한 클래스로 지금까지 많이 사용했습니다. 이와 달리 LogInDialog 클래스는 QDialog 클래스를 상속받아 새로 정의한 클래스입니다.

LogInDialog 클래스에서 내부 위젯의 구성은 지금까지 설명한 내용과 동일합니다. 내부 위젯을 배치하는 데 QGridLayout을 사용했습니다. 지금까지 설명하지 않은 내용으로는 다이얼로그 창에 아이콘을 설정하는 부분입니다. 아이콘 파일은 인터넷을 통해 png 형태의 파일을 내려받은 후 현재 소스코드와

같은 디렉터리에 icon.png라는 이름으로 저장하면 됩니다. 저자는 https://www.iconfinder.com/에서 아이콘을 내려받았습니다.

```
self.setWindowIcon(QIcon('icon.png'))
```

다이얼로그 창에서 사용자가 아이디와 비밀번호를 입력한 후 [Sign In] 버튼을 누르면 pushButton Clicked 메서드가 호출됩니다. 이 메서드에서는 self.lineEdit1과 self.lineEdit2 위젯에서 입력된 텍스트를 받아온 후 인스턴스 변수인 self.id, self.password에 저장합니다. 이처럼 위젯에서 입력받은 값을 인스턴스 변수에 저장해둠으로써 메인 윈도우에서 해당 값들을 읽어갈 수 있게 해줍니다. 마지막으로 close 메서드를 호출해 다이얼로그 창을 닫습니다.

```
def pushButtonClicked(self):
    self.id = self.lineEdit1.text()
    self.password = self.lineEdit2.text()
    self.close()
```

로그인 다이얼로그에 대해 알아봤으니 이번에는 메인 윈도우 역할을 수행하는 MyWindow 클래스에 대해 살펴보겠습니다. MyWindow 클래스 역시 대부분은 지금까지 사용한 MyWindow 클래스와 동일합니다. 다만 메인 윈도우의 버튼 위젯이 클릭될 때 호출되는 pushButtonClicked 메서드에 차이가 있습니다.

pushButtonClicked 메서드에서는 앞서 정의한 LogInDialog 클래스에 대한 인스턴스를 생성합니다. 그리고 인스턴스의 메서드인 exec_를 호출합니다. exec_ 메서드는 다이얼로그 창을 Modal 형태로 출력합니다. 참고로 Modal 다이얼로그는 해당 다이얼로그 창을 닫을 때까지 부모 윈도우로 이동할 수 없습니다.

사용자가 로그인 다이얼로그 창을 닫으면 dlg.exec_() 이후의 구문이 실행됩니다. 앞서 다이얼로그 창에서 인스턴스 변수인 dlg.id와 dlg.password에 위젯을 통해 입력받은 값을 저장해뒀으므로 해당 변수를 통해 아이디와 비밀번호를 메인 윈도우로 얻어올 수 있습니다. 메인 윈도우로 얻어온 아이디와 비밀번호를 메인 윈도우의 self.label 객체를 통해 화면에 출력합니다.

```
def pushButtonClicked(self):
    dlg = LogInDialog()
    dlg.exec_()
    id = dlg.id
    password = dlg.password
    self.label.setText("id: %s password: %s" % (id, password))
```

06 PyQt와 matplotlib 연동

이번 절에서는 그림 16.47과 같이 PyQt와 matplotlib를 연동해서 윈도우 내부에 그래프를 그려보겠습니다. 15장에서는 matplotlib를 이용해 그래프를 독립적인 윈도우로 출력했지만 시스템 트레이딩 프로그램을 개발하다 보면 그림 16.47과 같이 윈도우 내부에 PyQt 위젯과 그래프를 배치해야 하는 경우도 종종 발생합니다.

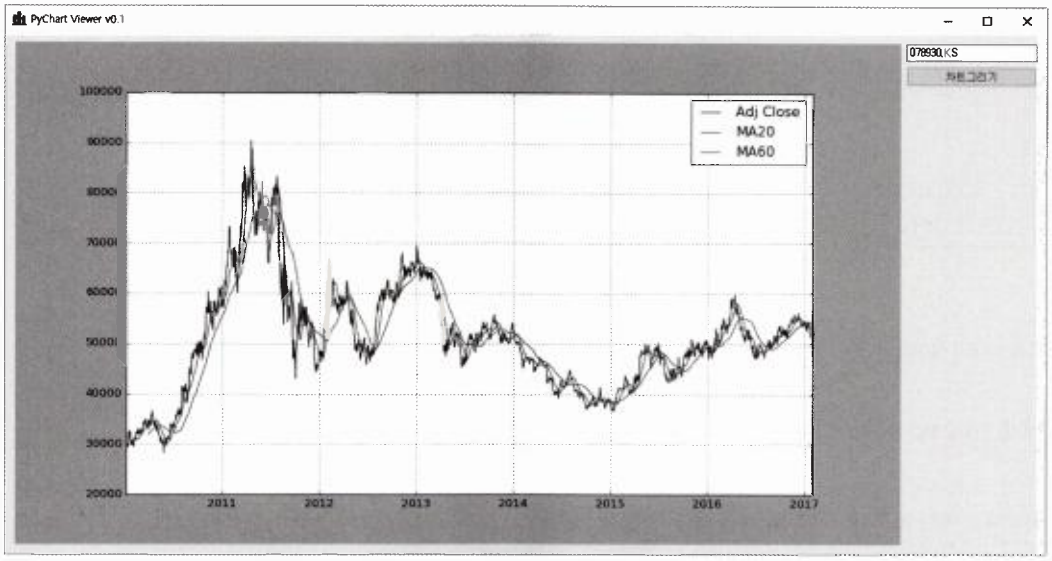


그림 16.47 PyQt와 matplotlib를 이용한 그래프 그리기

기본 레이아웃 구성

그림 16.48은 그림 16.47 프로그램의 기본 레이아웃을 그린 것입니다. 메인 윈도우의 크기가 변할 때 위젯의 크기도 변경될 수 있도록 레이아웃 매니저를 사용했습니다. 메인 윈도우의 가장 바깥쪽에는 QHBoxLayout을 사용했고 그 안쪽에는 두 개의 QVBoxLayout이 위치합니다.

왼쪽의 QVBoxLayout에는 matplotlib을 통해 그래프가 그려질 FigureCanvas 객체가 위치합니다. 오른쪽의 QVBoxLayout에는 사용자로부터 종목 코드를 입력받기 위한 QLineEdit 위젯과 '차트 그리기'라는 라벨을 갖는 QPushButton 위젯이 위치합니다.

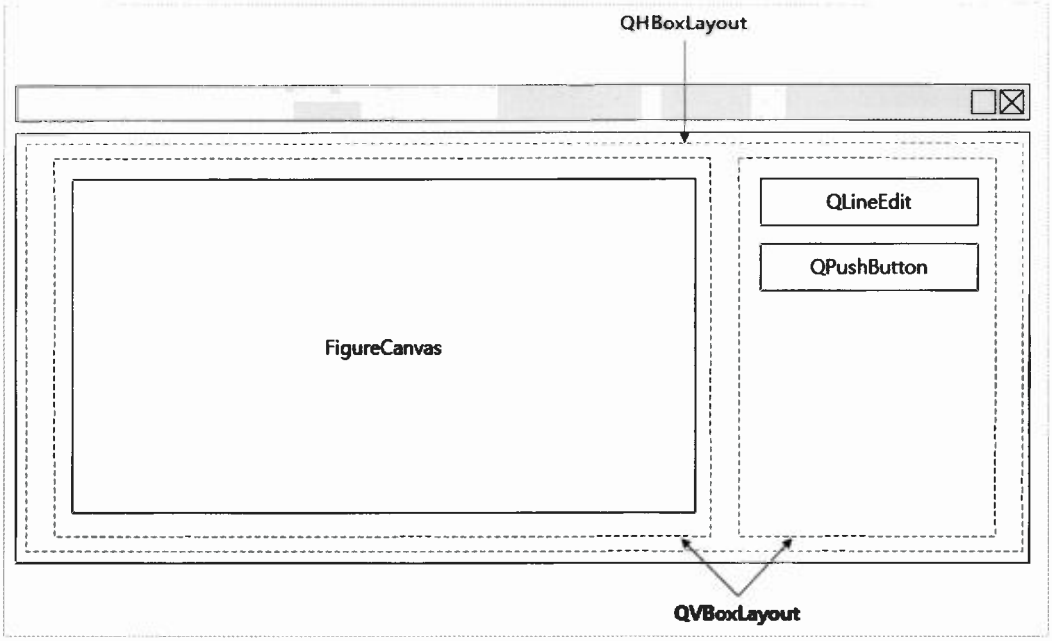


그림 16.48 레이아웃 구성

예제 16.21은 그림 16.48의 레이아웃을 구성하는 전체 코드입니다.

예제 16.21 메인 레이아웃 구성(소스코드: book/ch16/25.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(600, 200, 1200, 600)
        self.setWindowTitle("PyChart Viewer v0.1")
        self.setWindowIcon(QIcon('icon.png'))
```

```
        self.lineEdit = QLineEdit()
        self.pushButton = QPushButton("차트그리기")
        self.pushButton.clicked.connect(self.pushButtonClicked)

        self.fig = plt.Figure()
        self.canvas = FigureCanvas(self.fig)

        leftLayout = QVBoxLayout()
        leftLayout.addWidget(self.canvas)

        # Right Layout
        rightLayout = QVBoxLayout()
        rightLayout.addWidget(self.lineEdit)
        rightLayout.addWidget(self.pushButton)
        rightLayout.addStretch(1)

        layout = QHBoxLayout()
        layout.addLayout(leftLayout)
        layout.addLayout(rightLayout)
        layout.setStretchFactor(leftLayout, 1)
        layout.setStretchFactor(rightLayout, 0)

        self.setLayout(layout)

    def pushButtonClicked(self):
        print(self.lineEdit.text())

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyWindow()
    window.show()
    app.exec_()
```

matplotlib를 이용해 PyQt 내에 그래프를 그리려면 FigureCanvasQTAgg 클래스를 사용합니다. 이를 위해 다음과 같이 클래스를 임포트합니다.

```
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
```

다음과 같이 FigureCanvas 객체를 생성합니다.

```
self.fig = plt.Figure()
self.canvas = FigureCanvas(self.fig)
```

메인 윈도우의 오른쪽에 위치하는 QLineEdit와 QPushButton 객체를 상단으로 배치하기 위해 addStretch 메서드를 호출합니다. 참고로 addStretch 메서드는 크기 조절이 가능한 공백을 추가합니다.

```
rightLayout.addStretch(1)
```

메인 윈도우의 크기가 변경될 때 QLineEdit와 QPushButton 객체가 있는 rightLayout은 크기를 변경하지 않고 그래프가 출력되는 leftLayout만 크기 조절이 가능하도록 설정합니다.

```
layout.setStretchFactor(leftLayout, 1)
layout.setStretchFactor(rightLayout, 0)
```

예제 16.21에서 위의 두 줄을 제거한 후 실행해 보면 그림 16.49와 같은 레이아웃이 나타납니다.

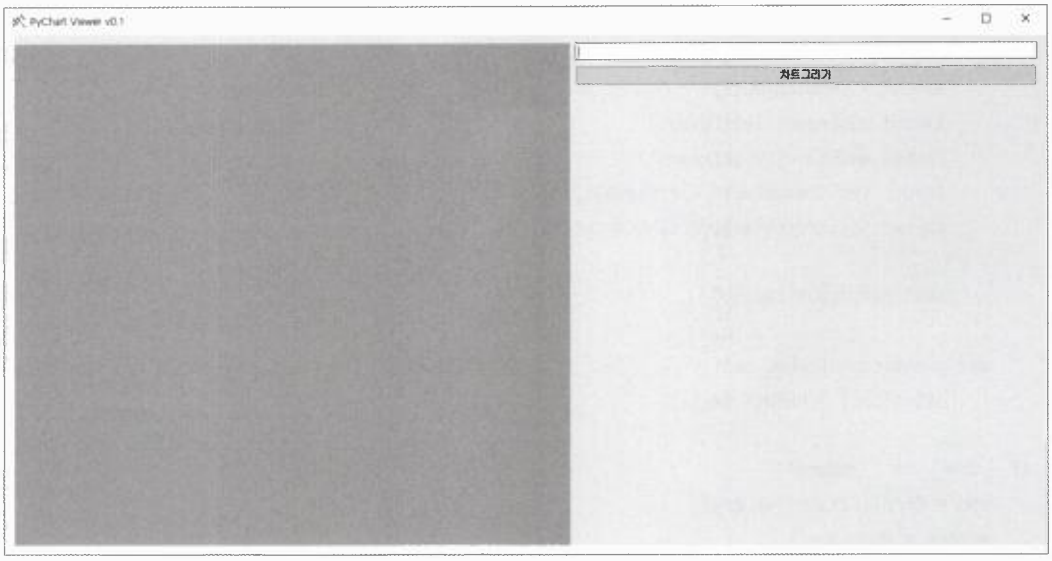


그림 16.49 메인 레이아웃-2

그래프 그리기

이번 절에서는 앞서 개발한 메인 레이아웃에 이벤트를 추가해보겠습니다. 메인 윈도우에서 QPushButton을 클릭했을 때 pushButtonClicked 메서드가 호출됩니다. pushButtonClicked 메서드에서는 QLineEdit에 입력된 종목 코드를 얻어온 후 pandas 모듈을 이용해 20일 이동평균 값과 60일 이동평균 값을 계산한 후 그래프로 출력합니다.

예제 16.22는 그림 16.47 프로그램의 전체 코드입니다. 프로그램을 실행한 후 KRX:078930을 입력하고 버튼을 클릭하면 그래프가 출력됩니다. 참고로 078930은 GS의 종목 코드입니다.

예제 16.22 그래프 그리기 이벤트 추가(소스코드: book/ch16/26.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
import pandas_datareader.data as web
import pandas as pd
from pandas import Series, DataFrame

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setupUI()

    def setupUI(self):
        self.setGeometry(600, 200, 1200, 600)
        self.setWindowTitle("PyChart Viewer v0.1")
        self.setWindowIcon(QIcon('icon.png'))

        self.lineEdit = QLineEdit()
        self.pushButton = QPushButton("차트그리기")
        self.pushButton.clicked.connect(self.pushButtonClicked)

        self.fig = plt.Figure()
        self.canvas = FigureCanvas(self.fig)

        leftLayout = QVBoxLayout()
        leftLayout.addWidget(self.canvas)

        # Right Layout
        rightLayout = QVBoxLayout()
        rightLayout.addWidget(self.lineEdit)
        rightLayout.addWidget(self.pushButton)
        rightLayout.addStretch(1)
```

```
        layout = QHBoxLayout()
        layout.addLayout(leftLayout)
        layout.addLayout(rightLayout)
        layout.setStretchFactor(leftLayout, 1)
        layout.setStretchFactor(rightLayout, 0)

        self.setLayout(layout)

    def pushButtonClicked(self):
        code = self.lineEdit.text()
        df = web.DataReader(code, "google")
        df['MA20'] = df['Close'].rolling(window=20).mean()
        df['MA60'] = df['Close'].rolling(window=60).mean()

        ax = self.fig.add_subplot(111)
        ax.plot(df.index, df['Close'], label='Close')
        ax.plot(df.index, df['MA20'], label='MA20')
        ax.plot(df.index, df['MA60'], label='MA60')
        ax.legend(loc='upper right')
        ax.grid()

        self.canvas.draw()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyWindow()
    window.show()
    app.exec_()
```

pushButtonClicked 메서드에 대해 조금 더 자세히 살펴보겠습니다. QLineEdit 객체로부터 사용자가 입력한 종목 코드를 얻어온 후 pandas 모듈을 통해 주가 이동 평균값을 계산합니다.

```
code = self.lineEdit.text()
df = web.DataReader(code, "google")
df['MA20'] = df['Close'].rolling(window=20).mean()
df['MA60'] = df['Close'].rolling(window=60).mean()
```

그래프를 그리기 위해 add_subplot 메서드를 호출해 AxesSubplot 객체를 생성한 후 plot 메서드를 통해 그래프를 그립니다. 마찬가지로 범례와 그리드도 그립니다.

```
ax = self.fig.add_subplot(111)
ax.plot(df.index, df['Close'], label='Close')
ax.plot(df.index, df['MA20'], label='MA20')
ax.plot(df.index, df['MA60'], label='MA60')
ax.legend(loc='upper right')
ax.grid()
```

마지막으로 FigureCanvas 객체에 그래프를 그리기 위해 draw 메서드를 호출합니다.

```
self.canvas.draw()
```

4부

실전
알고리즘 트레이딩

4부에서는 1~3부에서 배운 내용을 바탕으로 실전 프로그램을 개발합니다. 실전 프로그램에서는 거래량 기반의 투자 알고리즘과 배당률 기반 투자 알고리즘을 구현하고, 이를 통해 실제 매수/매도까지 자동으로 진행하는 프로그램을 개발함으로써 개발에 대한 자신감을 가질 수 있습니다.

17

주가 데이터 저장하기

기술적 분석(Technical Analysis)과 기본적 분석(Fundamental Analysis)으로 주가를 예측하기 위해서는 과거 시점의 데이터가 필요합니다. 기술적 분석에는 가격과 거래량이 중요하며 거래일 기준으로 시가, 고가, 저가, 종가, 거래량 등의 데이터가 사용됩니다. 기업의 내재적 가치를 분석해서 주가를 예측하는 방법인 기본적 분석에는 최근 몇 년 치의 기업 재무실적이나 PER, PBR, ROE 같은 기본적인 투자 지표에 대한 데이터가 사용됩니다.

데이터를 조작하는 관점에서는 DataFrame 형태로 데이터를 제공하는 야후 Finance나 구글 Finance가 편리합니다. 그러나 해당 사이트에서는 국내의 모든 종목에 대한 데이터를 제공하지 않으며 틱 단위의 데이터 역시 제공하지 않습니다. 국내 증권사 API를 이용하면 DataFrame 형태는 아니지만 모든 종목에 대한 데이터를 내려받을 수 있으며 일봉 데이터의 경우 꽤 오랜 기간 동안의 데이터를 내려받을 수 있습니다. 다만 틱 데이터는 데이터의 양이 많기 때문에 최근 거래일로부터 일정 기간 동안의 데이터만 제공합니다.

결국 기술적 분석이나 기본적 분석을 위해서는 가장 먼저 자신에게 필요한 과거 데이터부터 준비해야 합니다. 특히 틱 데이터의 경우에는 증권사 API를 이용해 데이터를 여러분의 PC에 주기적으로 받아 두

는 것이 필요합니다. 이번 절에서는 증권사 API를 이용해 일봉 데이터를 저장하는 것과 내려받은 데이터를 효과적으로 저장하기 위해 데이터베이스(database)를 사용하는 방법을 다룹니다.

01 SQLite

이번 절에서 배울 SQLite는 데이터베이스 관리 시스템(DBMS; Database Management System)입니다. DBMS는 그림 17.1과 같이 여러 사용자 또는 응용 프로그램이 데이터를 저장하거나 조회할 수 있게 도와주는 시스템 또는 소프트웨어입니다. 대표적인 DBMS로는 MySQL, PostgreSQL, Oracle, MS-SQL 등이 있습니다.

SQLite는 Oracle이나 PostgreSQL과 같은 큰 규모의 DBMS와 달리 서버 설치와 설정 면에서 'Lite'한 DBMS입니다. SQLite는 응용 프로그램에서 내부 데이터를 저장할 때도 사용하기 적합해서 안드로이드폰이나 아이폰 애플리케이션에서 많이 사용되고 있습니다. 일단 SQLite를 통해 DBMS 사용법을 익힌 후에 개발하려는 시스템의 규모에 따라 향후에는 PostgreSQL, MySQL 등을 사용하는 편이 좋습니다.

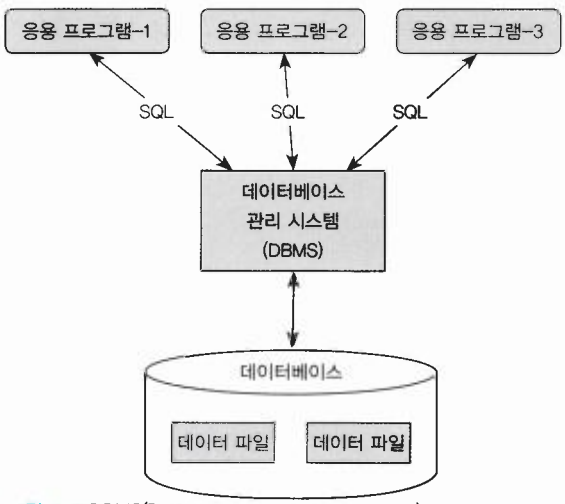


그림 17.1 DBMS(Database Management System)

sqlite3 모듈 기초

sqlite3 모듈은 파이썬 표준 라이브러리(파이썬이 설치될 때 기본적으로 설치되는 모듈)로 SQLite에 대한 인터페이스를 제공합니다. 쉽게 설명하면 표준 라이브러리인 sqlite3 모듈을 이용하면 따로 모듈을 설치할 필요 없이 데이터베이스를 쉽게 이용할 수 있습니다.

그림 17.2와 같이 Jupyter QtConsole을 실행한 후 sqlite3 모듈을 임포트합니다. 그리고 version과 sqlite_version을 확인해보기 바랍니다. 여기서 version은 sqlite3 모듈 자체의 버전을 의미하고, sqlite_version은 SQLite의 버전을 의미합니다.

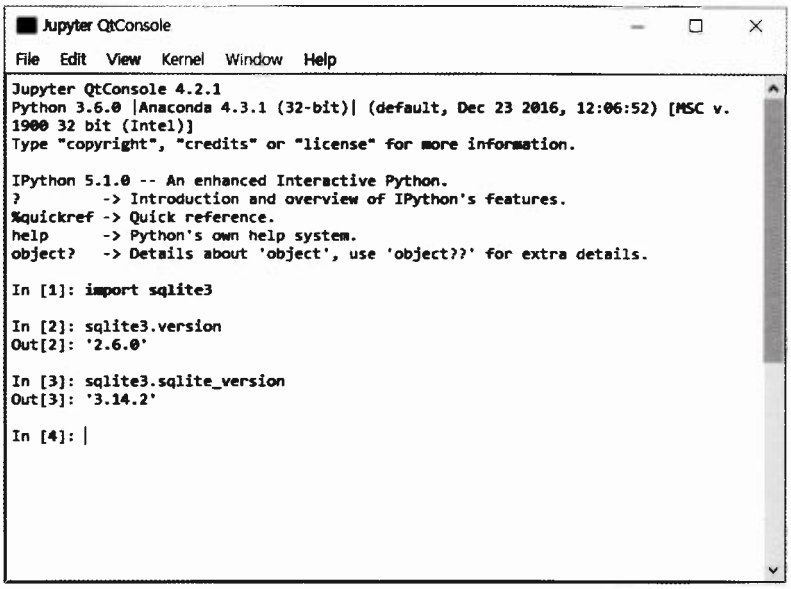


그림 17.2 sqlite3 모듈 임포트 및 버전 확인

sqlite3 모듈이 정상적으로 임포트됐다면 이번에는 데이터베이스를 생성해 보겠습니다. 데이터베이스 생성은 sqlite3 모듈의 connect 함수를 사용합니다. 이때 함수의 인자로 생성할 데이터베이스 파일의 경로 및 이름을 지정하면 됩니다. 윈도우 c:\ 드라이브의 최상위 경로에는 권한 문제 때문에 데이터베이스가 생성되지 않을 수도 있는데 이때는 다른 경로를 사용하면 됩니다.

다음 예에서 저자는 'c:/Users/Jason'이라는 경로에 데이터베이스를 생성했는데 여기서 'Jason'은 저자의 윈도우 로그인 아이디입니다. 여러분은 본인의 윈도우 로그인 아이디에 맞춰 적당히 경로를 수정하면 됩니다.

connect 함수가 에러 없이 정상적으로 호출됐다면 con 타입도 살펴보기 바랍니다. type 내장 함수를 통해 확인해보면 Connection 객체임을 확인할 수 있습니다. 사실 con의 타입이 중요한 것은 아니고 con이라는 객체를 통해 앞으로 데이터베이스를 조작할 수 있다는 것을 기억하는 것이 중요합니다.

```
In [1]: con = sqlite3.connect("c:/Users/Jason/kospi.db")
In [2]: type(con)
Out[3]: sqlite3.Connection
```

파일 다루기에서 open 내장 함수를 사용해서 파일을 생성했던 것처럼 connect 함수를 호출하면 새로운 데이터베이스 파일이 생성됩니다. 그림 17.3과 같이 connect 함수의 인자로 사용했던 경로('c:/Users/Jason')를 윈도우의 파일 탐색기를 통해 접근해보면 kospi.db라는 파일이 생성됐음을 확인할 수 있습니다.



그림 17.3 생성된 kospi.db 파일

이번에는 생성된 데이터베이스에 데이터를 넣어보겠습니다. 그림 17.1을 참조하면 DBMS는 SQL(Structured Query Language)이라는 데이터베이스 관리 전용 프로그래밍 언어를 통해 사용할 수 있습니다. 파이썬도 배우고 있는데 여기에 또 SQL까지 배울 생각을 하니 벌써 머리가 아프지요? 그래도 SQL은 생각보다는 어렵지 않으니 천천히 따라오기 바랍니다.

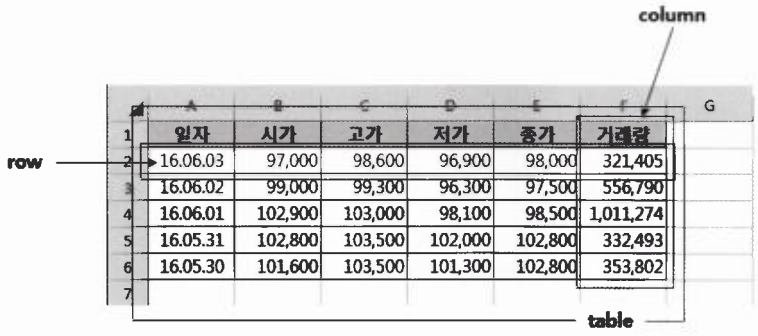
DBMS에서는 보통 SQL 구문을 호출해서 데이터를 조작하게 되는데 SQL 구문을 호출하려면 Cursor 객체가 필요합니다. 다음과 같이 Connection 객체인 con을 사용해 Cursor 객체를 생성합니다. 이것으로 데이터베이스 조작을 위한 기본적인 준비를 마쳤습니다.

```
In [4]: cursor = con.cursor()
```

데이터베이스 파일에 데이터를 저장하는 데는 다양한 방법이 있지만 가장 일반적인 방법은 그림 17.4와 같이 2차원 형태로 데이터를 저장하는 것입니다. 그림 17.4는 카카오 종목에 대한 5일 치 시가, 고가, 저가, 종가, 거래량 데이터를 마이크로소프트 엑셀로 정리해본 것입니다. 여러분도 이러한 형태의 데이터에 익숙할 것입니다.

데이터베이스를 사용할 때 가장 자주 사용하는 용어 세 가지에 대해 먼저 정리해보겠습니다. 그림 17.4에서 각 열에 위치하는 일자, 시가, 고가, 저가, 종가, 거래량 데이터를 칼럼(column)이라고 부르

고 각 행에 해당하는 데이터 묶음을 로우(row)라고 합니다. 마지막으로 그림 17.4와 같이 2차원 형태의 데이터 구조를 테이블(table)이라고 합니다.



	A	B	C	D	E	F	G
1	일자	시가	고가	저가	종가	거래량	
2	16.06.03	97,000	98,600	96,900	98,000	321,405	
3	16.06.02	99,000	99,300	96,300	97,500	556,790	
4	16.06.01	102,900	103,000	98,100	98,500	1,011,274	
5	16.05.31	102,800	103,500	102,000	102,800	332,493	
6	16.05.30	101,600	103,500	101,300	102,800	353,802	
7							

그림 17.4 데이터베이스 용어(테이블, 칼럼, 로우)

이번에는 그림 17.4의 데이터를 파이썬을 통해 데이터베이스에 저장해보겠습니다. 데이터를 넣기 전에 가장 먼저 해야 하는 일은 테이블을 만드는 것입니다. 테이블을 만들 때는 각 칼럼에 어떤 데이터가 저장되는지에 대해 함께 기술해야 합니다.

테이블을 만들 때는 다음과 같이 SQL 구문을 사용합니다. 'CREATE TABLE'은 테이블을 만든다는 SQL 구문이고 kakao는 테이블의 이름입니다. 함수의 인자와 같은 부분은 각각 칼럼 이름과 칼럼 데이터 타입을 기술하는 부분입니다. 일자 데이터는 데이터 타입으로 text를 사용했는데 이는 파이썬의 문자열과 같은 의미입니다. 나머지 데이터는 모두 int로 지정했는데 이는 파이썬의 정수형과 같습니다.

```
CREATE TABLE kakao(Date text, Open int, High int, Low int, Closing int, Volumn int)
```

DBMS에는 데이터베이스를 관리할 때 위와 같이 SQL 구문을 사용합니다. 파이썬에서 SQL 구문을 실행하려면 앞서 생성한 Cursor 객체의 execute 메서드를 사용하면 됩니다. 다음 코드를 보면 execute 메서드의 인자로 kakao라는 테이블을 생성하는 SQL 구문이 입력된 것을 확인할 수 있습니다.

```
In [5]: cursor.execute("CREATE TABLE kakao(Date text, Open int, High int, Low int, Closing int,
Volumn int)")
Out[5]: <sqlite3.Cursor at 0x4aa2d60>
```

위 코드를 실행하면 그림 17.5와 같은 형태의 테이블이 만들어집니다. SQL 구문을 사용하는 것이 복잡해 보이지만 단계별로 나눠서 생각해보면 여러분이 엑셀로 데이터를 만드는 과정과 그 절차가 매우 비슷하다는 것을 확인할 수 있습니다.

Date	Open	High	Low	Closing	Volumn

그림 17.5 테이블 생성

kakao라는 테이블을 만들었으니 이번에는 테이블에 데이터를 넣어보겠습니다. 역시 SQL 구문을 만든 후 해당 구문을 cursor.execute 메서드를 사용해서 호출하면 됩니다. 로우(Row)에 데이터를 삽입하는 SQL 구문은 다음과 같습니다. 'INSERT INTO kakao VALUES'는 kakao라는 테이블 안으로(into) VALUES를 삽입(insert)한다는 의미입니다. 테이블로 입력할 로우 데이터는 VALUES() 부분에 지정하면 됩니다.

```
INSERT INTO kakao VALUES("16.06.03", 97000, 98600, 96900, 98000, 321405)
```

SQL 구문이 실제로 수행되게 하려면 cursor.execute를 통해 실행하면 됩니다.

```
In [8]: cursor.execute("INSERT INTO kakao VALUES('16.06.03', 97000, 98600, 96900, 98000,
321405)")
Out[8]: <sqlite3.Cursor at 0x4aa2d60>
```

execute 메서드를 사용해 SQL 구문을 실행할 때 한 가지 주의할 점이 있습니다. SQL 구문을 실행하는 코드를 살펴보면 SQL 구문 전체를 큰따옴표를 사용해서 문자열로 만들었습니다. 따라서 '16.06.03'과 같이 SQL 구문에 문자열이 있는 경우 작은따옴표를 사용해야 합니다. 만약 그림 17.6과 같이 내부에서도 큰따옴표를 사용하는 경우 에러가 발생합니다.

```
In [6]: cursor.execute("INSERT INTO kakao VALUES("16.06.03", 97000, 98600, 96900,
98000, 321405)")
  File "<ipython-input-6-0159a3c949f5>", line 1
    cursor.execute("INSERT INTO kakao VALUES("16.06.03", 97000, 98600, 96900,
98000, 321405)")
                                                ^
SyntaxError: invalid syntax
```

그림 17.6 SQL 구문 실행 에러

이번에는 16.06.02에 해당하는 데이터를 로우로 추가해 보겠습니다.

```
In [9]: cursor.execute("INSERT INTO kakao VALUES('16.06.02', 99000, 99300, 96300, 97500,
556790)")
Out[9]: <sqlite3.Cursor at 0x4aa2d60>
```

지금까지 작업한 내용을 실제로 데이터베이스에 반영하려면 commit 메서드를 호출해야 합니다. 그리고 close 메서드를 통해 DB 연결을 닫습니다.

```
In [10]: con.commit()
In [11]: con.close()
```

데이터베이스 브라우저

앞서 kospi.db라는 데이터베이스 파일을 만들고 해당 데이터베이스 안으로 kakao라는 테이블을 생성했습니다. kokao 테이블에는 날짜, 시가, 고가, 저가, 종가, 거래량의 데이터가 일자별로 저장돼 있습니다. 이번 절에서는 데이터베이스 브라우저를 통해 kospi.db 파일에 제대로 데이터가 입력됐는지 확인해 보겠습니다.

데이터베이스 브라우저는 데이터베이스에 저장된 데이터를 효과적으로 보여주는 프로그램입니다. SQLite에는 DB Browser for SQLite라는 프로그램이 있습니다. 먼저 프로그램의 웹 페이지(http://sqlitebrowser.org/)로 이동한 후 그림 17.7과 같이 32비트 윈도우용 설치 파일을 내려받습니다. 물론 윈도우가 64비트인 경우에는 64비트 설치 파일을 다운로드하면 됩니다.

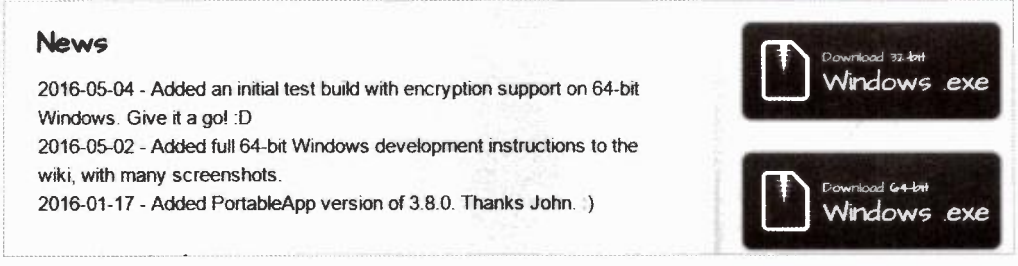


그림 17.7 윈도우 설치 파일 다운로드

내려받은 설치 파일을 더블클릭해서 프로그램 설치를 시작합니다. 그림 17.8에서 [다음] 버튼을 눌러 설치를 시작합니다.

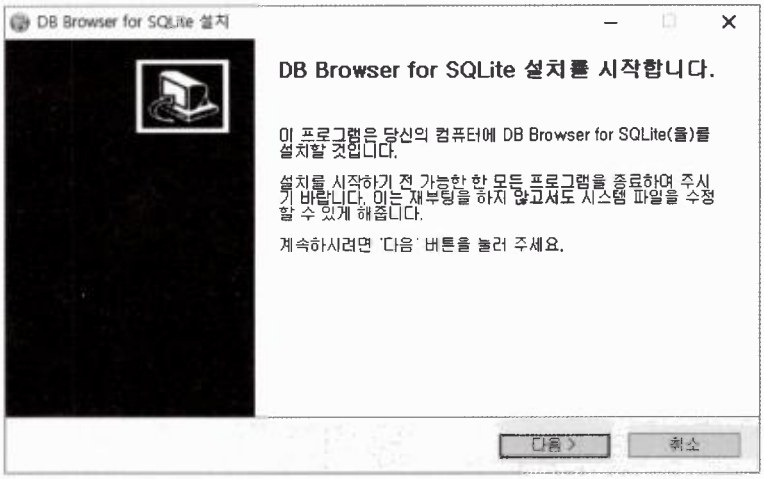


그림 17.8 DB Browser for SQLite 설치 – 1

사용권 계약에 [동의함]을 누르고 다음 설치 과정으로 넘어갑니다.

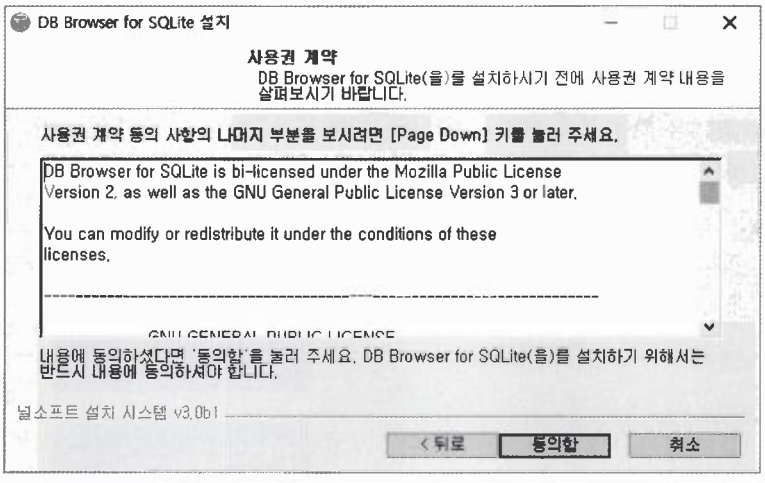


그림 17.9 DB Browser for SQLite 설치 – 2

프로그램 설치 위치를 선택합니다. 기본 설치 폴더를 변경하지 않고 [다음] 버튼을 눌러 설치를 계속합니다.

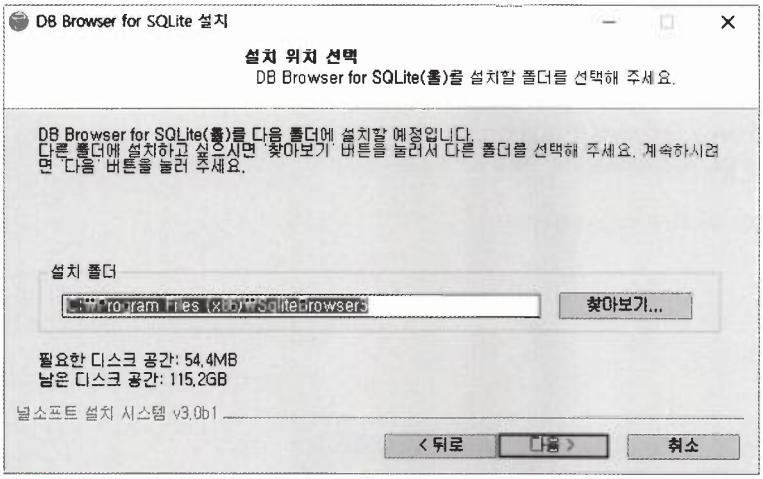


그림 17.10 DB Browser for SQLite 설치 - 3

[설치] 버튼을 눌러 설치를 시작합니다.

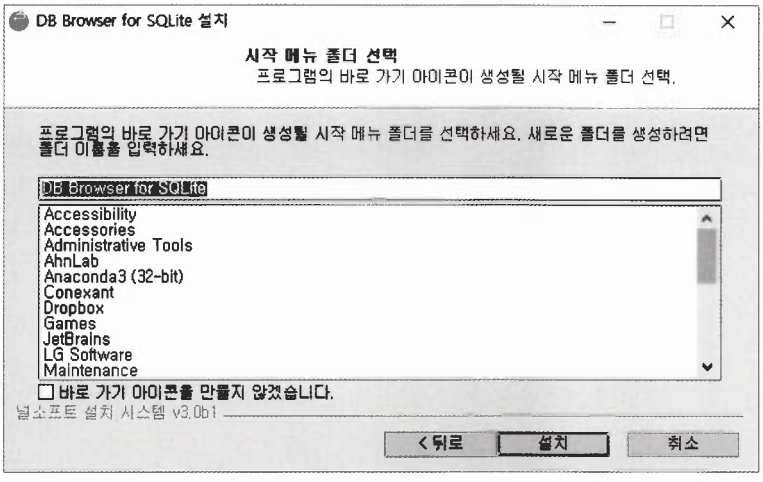


그림 17.11 DB Browser for SQLite 설치 - 4

설치가 완료되면 [마침] 버튼을 눌러 설치를 종료합니다.

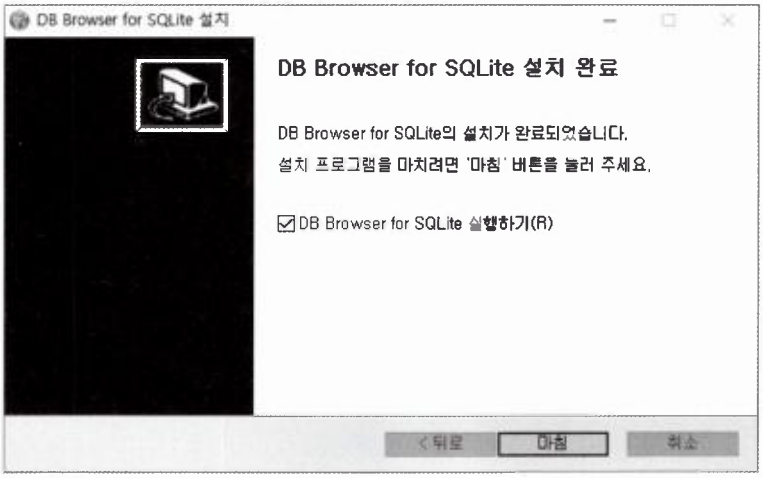


그림 17.12 DB Browser for SQLite 설치 – 5

설치가 완료되면 그림 17.13과 같이 DB Browser for SQLite가 실행됩니다. 데이터베이스 파일을 열기 위해 [Open Database] 버튼을 클릭합니다.

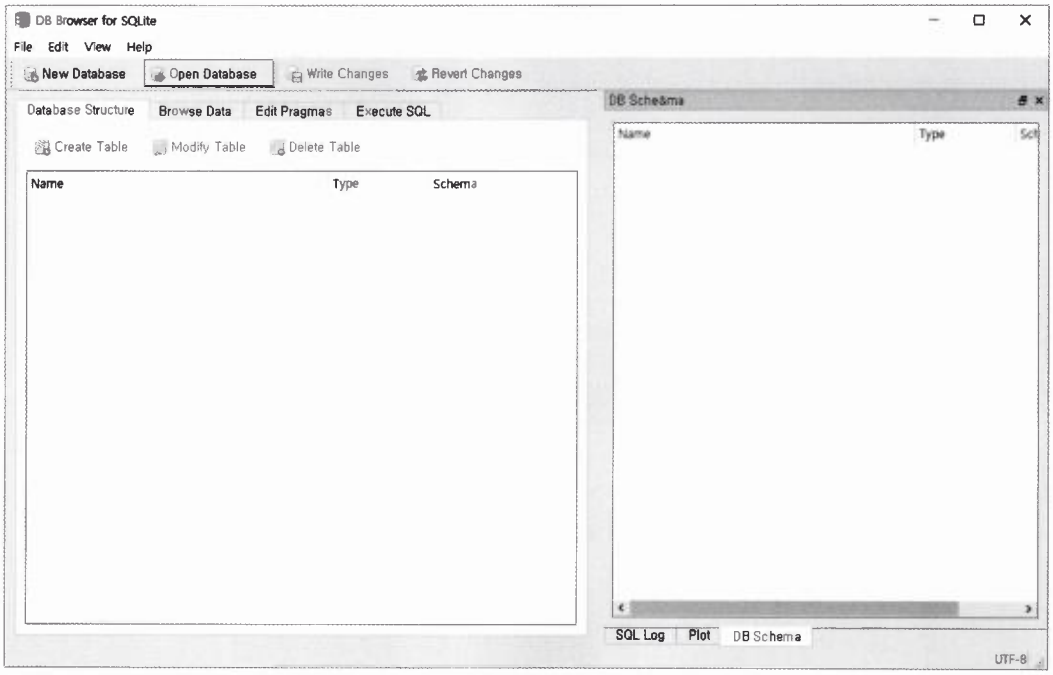


그림 17.13 DB Browser for SQLite를 이용한 데이터베이스 열기

앞서 데이터베이스 파일을 생성했던 디렉터리 경로로 이동한 후 kospi.db 파일을 선택한 후 [열기] 버튼을 클릭합니다.



그림 17.14 kospi.db 파일 열기

kospi.db 파일이 정상적으로 열렸다면 그림 17.15와 같이 [Browse Data] 탭을 클릭합니다. 그림 17.4의 엑셀 파일과 같은 형태로 kakao라는 테이블에 데이터가 잘 저장돼 있음을 확인할 수 있습니다. 물론 여러분이 데이터베이스를 사용할 때 DB Browser for SQLite와 같은 프로그램을 사용해서 데이터베이스 파일을 연 후 데이터를 확인하지는 않습니다. DB Browser for SQLite는 데이터베이스의 구조를 파악하거나 데이터가 잘 저장돼 있는지 확인할 때 사용합니다.

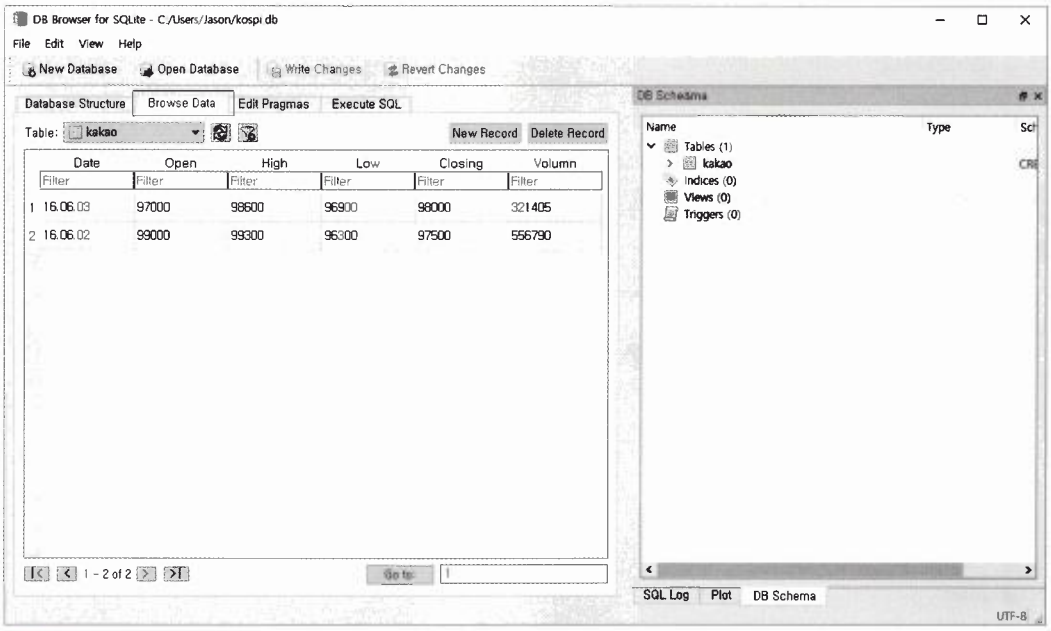


그림 17.15 Browse Data 탭

데이터베이스로부터 데이터 읽기

이번 절에서는 앞서 생성한 kospi.db라는 데이터베이스 파일로부터 데이터를 읽어와 보겠습니다. 먼저 Jypyter QtConsole을 실행한 후 다음과 같이 Cursor 객체까지 생성합니다. 참고로 kospi.db 파일의 경로가 다른 경우에는 적절히 수정해야 합니다.

```
In [1]: import sqlite3

In [2]: con = sqlite3.connect("c:/Users/Jason/kospi.db")

In [3]: cursor = con.cursor()
```

데이터베이스에서 데이터를 읽으려면 먼저 테이블을 선택하는 SQL 구문을 실행해야 합니다. "SELECT * FROM kakao"라는 SQL 구문은 'kakoa라는 테이블로부터 모든 것(*)을 선택하라'는 의미입니다. 항상 느끼는 것이지만 SQL 구문의 구조가 생각보다 어렵지 않지요?

```
In [4]: cursor.execute("SELECT * From kakao")
Out[4]: <sqlite3.Cursor at 0x4a9f3a0>
```

선택한 테이블로부터 로우 단위로 데이터를 읽으려면 fetchone 메서드를 호출합니다. 다음 코드와 같이 fetchone을 호출할 때마다 로우 단위로 데이터를 얻을 수 있음을 확인할 수 있습니다. kakao라는 테이블에는 두 개의 로우만 존재하기 때문에 fetchone을 세 번 호출하면 마지막 호출에 대해서는 아무런 값도 반환되지 않음을 확인할 수 있습니다. 참고로 fetchone 메서드의 리턴 값은 튜플입니다.

```
In [5]: cursor.fetchone()
Out[5]: ('16.06.03', 97000, 98600, 96900, 98000, 321405)

In [6]: cursor.fetchone()
Out[6]: ('16.06.02', 99000, 99300, 96300, 97500, 556790)

In [7]: cursor.fetchone()

In [8]:
```

테이블로부터 로우 단위로 데이터를 읽을 수도 있지만 한 번에 모든 로우를 읽을 수도 있습니다. 한 번에 모든 로우를 읽기 위해서는 fetchall 메서드를 사용하면 됩니다. 앞서 선택한 테이블에서는 모든 데이터를 읽어 버렸기 때문에 다시 한 번 SELECT 구문을 실행한 후 fetchall 메서드를 호출하면 됩니다.

```
In [8]: cursor.execute("SELECT * From kakao")
Out[8]: <sqlite3.Cursor at 0x4a9f3a0>

In [9]: cursor.fetchall()
Out[9]:
[('16.06.03', 97000, 98600, 96900, 98000, 321405),
('16.06.02', 99000, 99300, 96300, 97500, 556790)]
```

이번에는 리턴 값을 변수로 바인딩해둔 후 해당 변수를 통해 데이터에 접근해 보겠습니다. 보통 데이터베이스로부터 읽어온 데이터는 이 같은 방식으로 처리합니다. fetchall 메서드의 반환값을 kakao라는 변수로 바인딩합니다. kakao는 리스트 타입이기 때문에 인덱싱을 통해 데이터에 접근할 수 있습니다.

```
In [10]: cursor.execute("SELECT * From kakao")
Out[10]: <sqlite3.Cursor at 0x4a9f3a0>

In [11]: kakao = cursor.fetchall()

In [12]: kakao[0][0]
Out[12]: '16.06.03'
```

```
In [13]: kakao[0][1]
Out[13]: 97000

In [14]: kakao[0][2]
Out[14]: 98600

In [15]:
```

02 pandas와 SQLite

13장에서 배운 pandas의 DataFrame은 2차원 형태의 자료구조였습니다. pandas는 데이터 분석에 유용한 라이브러리로서 주식 데이터를 DataFrame 형태로 저장하고 다루는 방법은 매우 유용했습니다. 이번 절에서는 pandas의 DataFrame 객체를 데이터베이스에 저장하는 방법과 데이터베이스에 저장된 데이터를 DataFrame 객체로 로드하는 방법을 알아보겠습니다.

DataFrame 객체를 SQLite DB에 저장하기

DataFrame 객체를 SQLite DB에 저장하기 위해 먼저 간단한 형태의 DataFrame 객체를 생성해 보겠습니다. Jupyter QtConsole을 실행한 후 다음과 같이 입력하면 됩니다. 참고로 DataFrame 객체를 쉽게 생성하는 방법은 데이터를 파이썬 딕셔너리로 표현한 후 DataFrame 생성자를 호출하는 것입니다.

```
In [1]: import pandas as pd

In [2]: from pandas import Series, DataFrame

In [3]: raw_data = {'col0': [1, 2, 3, 4], 'col1': [10, 20, 30, 40], 'col2':[100, 200, 300, 400]}

In [4]: df = DataFrame(raw_data)

In [5]: df
Out[5]:
   col0  col1  col2
0     1    10   100
1     2    20   200
2     3    30   300
3     4    40   400

In [6]:
```

sqlite3 모듈을 임포트한 후 connect 함수를 호출해 기존에 생성해둔 kospi.db를 엽니다.

```
In [6]: import sqlite3
```

```
In [7]: con = sqlite3.connect("c:/Users/Jason/kospi.db")
```

앞서 Cursor 객체를 생성한 후 SQL 구문을 execute 메서드로 호출하는 방식을 통해 데이터베이스를 조작했습니다. pandas의 DataFrame 클래스는 DataFrame 객체 내의 데이터를 데이터베이스로 저장하기 위해 to_sql이라는 메서드를 제공합니다. 따라서 명시적으로 SQL 구문을 사용하지 않고도 to_sql 메서드 호출을 통해 데이터베이스에 데이터를 저장할 수 있습니다. 다음과 같이 to_sql 메서드를 호출해 'test'라는 이름의 테이블에 df 객체의 내용을 저장합니다.

```
In [10]: df.to_sql('test', con)
```

to_sql 메서드가 제대로 동작했는지 확인해보기 위해 DB Browser for SQLite 프로그램을 실행해 'kospi.db' 파일을 열어봅니다. [Browse Data] 탭에서 [Table] 항목을 'test'로 선택하면 그림 17.16과 같이 df 객체의 데이터가 저장돼 있음을 확인할 수 있습니다.

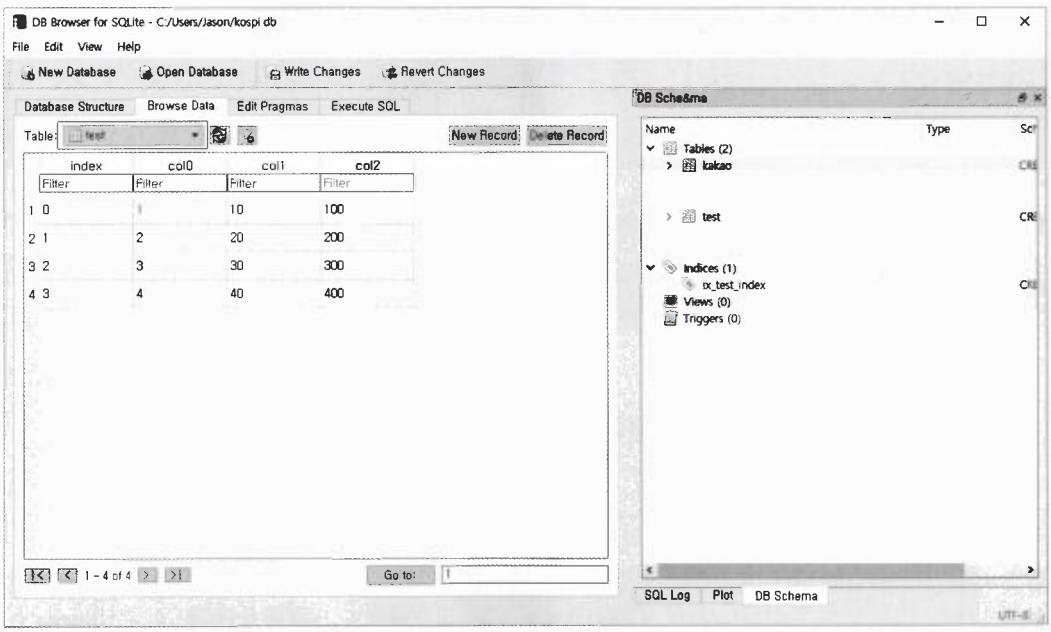


그림 17.16 test 테이블 확인

pandas 웹 페이지(http://pandas.pydata.org/)에서 to_sql 메서드를 살펴보면 다음과 같이 정의돼 있습니다.

```
DataFrame.to_sql(name, con, flavor='sqlite', schema=None, if_exists='fail', index=True, index_
label=None, chunksize=None, dtype=None)
```

to_sql 메서드의 파라미터에 대한 자세한 내용은 표 17.1을 참고하기 바랍니다.

표 17.1 DataFrame.to_sql 메서드 파라미터

파라미터	설명
name	SQL 테이블 이름으로 파이썬 문자열로 형태로 나타낸다.
con	Cursor 객체
flavor	DBMS를 지정할 수 있다. 'sqlite' 또는 'mysql'을 사용할 수 있다. 기본값은 'sqlite'다.
schema	Schema를 지정할 수 있다. 기본값은 None이다.
if_exists	데이터베이스에 테이블이 존재할 때 수행 동작을 지정한다. 'fail', 'replace', 'append' 중 하나를 사용할 수 있는데 기본값은 'fail'이다. 'fail'은 데이터베이스에 테이블이 있다면 아무 동작도 수행하지 않는다. 'replace'는 테이블이 존재하면 기존 테이블을 삭제하고 새로 테이블을 생성한 후 데이터를 삽입한다. 'append'는 테이블이 존재하면 데이터만 추가한다.
index	DataFrame의 index를 데이터베이스에 칼럼으로 추가할지 여부를 지정한다. 기본값은 True다.
index_label	인덱스 칼럼에 대한 라벨을 지정할 수 있다. 기본값은 None이다.
chunksize	한 번에 써지는 로우의 크기를 정숫값으로 지정할 수 있다. 기본값은 None으로 DataFrame 내의 모든 로우가 한 번에 써진다.
dtype	칼럼에 대한 SQL 타입을 파이썬 딕셔너리로 전달할 수 있다.

to_sql 메서드에는 여러 파라미터가 있지만 대부분 기본값을 가지고 있어 따로 지정할 필요가 없는 경우가 많습니다. 다만 DataFrame 객체에 많은 로우가 있어 이를 데이터베이스 한 번에 쓰는 경우 패킷 크기 제약으로 에러가 발생할 수 있습니다. 이 경우 다음과 같이 한 번에 데이터베이스로 저장될 로우의 개수를 지정하면 됩니다.

```
In [11]: df.to_sql('test', con, chunksize=1000)
```

SQLite DB에서 테이블 로드하기

이번 절에서는 pandas를 이용해 데이터베이스의 테이블을 DataFrame 객체로 읽어오는 방법을 알아보겠습니다. pandas는 데이터베이스의 테이블을 DataFrame 객체로 읽어오기 위해 read_sql 함수를 제공합니다.

먼저 Jupyter QtConsole을 실행합니다. 그리고 필요한 모듈을 임포트한 후 앞서 생성한 kospi.db 파일을 엽니다.

```
In [1]: import pandas as pd

In [2]: from pandas import Series, DataFrame

In [3]: import sqlite3

In [4]: con = sqlite3.connect("c:/Users/Jason/kospi.db")
```

read_sql 함수를 호출해 kakao 테이블 데이터를 df라는 DataFrame 객체로 로드합니다. read_sql 함수의 첫 번째 인자로는 SQL 구문을 입력하고, 두 번째 인자로는 Connection 객체를 전달합니다. index_col 인자는 DataFrame 객체에서 인덱스로 사용될 칼럼을 지정하는 것으로 기본값은 None입니다. None을 입력하면 자동으로 0부터 시작하는 정숫값이 인덱스로 할당됩니다.

```
In [23]: df = pd.read_sql("SELECT * From kakao", con, index_col=None)

In [24]: df
Out[24]:
       Date   Open   High    Low  Closing  Volumn
0  16.06.03  97000  98600  96900    98000  321405
1  16.06.02  99000  99300  96300    97500  556790
```

이번에는 앞서 생성했던 'test'라는 테이블을 로드합니다. 그림 17.17과 같이 read_sql 함수를 사용해 'test' 테이블을 로드하면 DataFrame 객체에 추가로 인덱스 칼럼이 생성된 것을 확인할 수 있습니다.

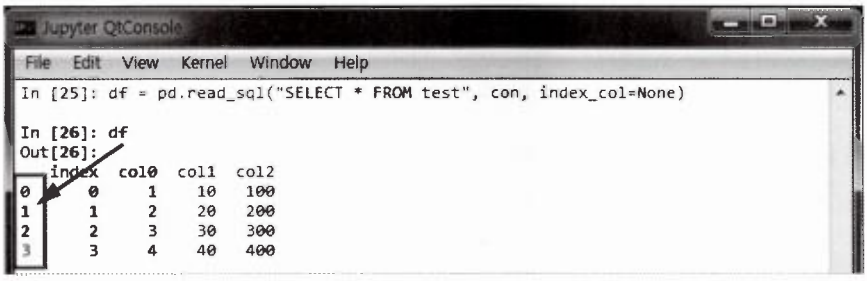


그림 17.17 read_sql 함수를 사용해 데이터베이스 테이블을 로드

이처럼 테이블에 이미 DataFrame의 인덱스에 해당하는 칼럼이 존재하는 경우에는 해당 칼럼의 이름을 index_col 인자로 전달하면 됩니다. 이번에는 추가로 인덱스 칼럼이 생성되지 않았음을 확인할 수 있습니다.

```
In [27]: df = pd.read_sql("SELECT * From test", con, index_col='index')

In [28]: df
Out[28]:
       col0  col1  col2
index
0         1    10   100
1         2    20   200
2         3    30   300
3         4    40   400
```

pandas를 이용한 주가 데이터 저장

이번 절에서는 pandas의 DataReader를 통해 특정 종목의 일봉 데이터를 DataFrame 형태로 다운로드한 후 이를 데이터베이스로 저장해보겠습니다.

먼저 관련 모듈을 임포트합니다. DataReader를 통해 읽은 데이터를 SQLite로 저장하기 위해 sqlite3 모듈도 미리 임포트합니다.

```
import pandas as pd
import pandas_datareader.data as web
import datetime
import sqlite3
```

DataReader를 통해 GS 종목(종목 코드: '078930')의 일별 데이터를 DataFrame 형태로 다운로드합니다.

```
start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2016, 6, 12)
df = web.DataReader("KRX:078930", "google", start, end)
```

df.head()를 이용해 df 객체에 저장된 일부 값을 확인해 봅니다. 다음과 같이 시가, 고가, 저가, 종가, 거래량이 거래일별로 존재하는 것을 확인할 수 있습니다.

```
              Open     High      Low    Close  Volume
Date
2010-01-04  33500.0  34050.0  33500.0  33750.0  236460
2010-01-05  33950.0  34500.0  33600.0  33900.0  439670
2010-01-06  33900.0  34900.0  33900.0  34250.0  533210
2010-01-07  34350.0  35100.0  34300.0  34600.0  543270
2010-01-08  34700.0  35100.0  34200.0  34450.0  551230
```

SQLite 데이터베이스에 테이블로 생성한 후 DataFrame 객체를 저장합니다. 테이블 이름은 종목별로 구분될 수 있도록 6자리 숫자의 종목 코드를 그대로 사용하겠습니다.

```
con = sqlite3.connect("c:/Users/Jason/kospi.db")
df.to_sql('078930', con, if_exists='replace')
```

이번에는 데이터베이스에 저장된 데이터를 다시 로드해서 pandas의 DataFrame 객체로 생성해보겠습니다. 앞에서 배운 것처럼 pandas의 read_sql 함수를 사용하면 됩니다. 이때 read_sql 함수의 첫 번째 인자로 SQL 쿼리를 전달합니다. 종목 코드가 숫자 형태로 구성돼 있으므로 한 번 작은따옴표로 감싸줘야 합니다.

```
readed_df = pd.read_sql("SELECT * From '078930'", con, index_col = 'Date')
```

readed_df 객체에 데이터가 잘 저장돼 있는지 readed_df.head()를 호출해서 확인해 봅니다.

```
                     Open     High      Low    Close  Volume
Date
2010-01-04 00:00:00  33500.0  34050.0  33500.0  33750.0  236460
2010-01-05 00:00:00  33950.0  34500.0  33600.0  33900.0  439670
2010-01-06 00:00:00  33900.0  34900.0  33900.0  34250.0  533210
2010-01-07 00:00:00  34350.0  35100.0  34300.0  34600.0  543270
2010-01-08 00:00:00  34700.0  35100.0  34200.0  34450.0  551230
```

예제 17.1은 지금까지 설명한 내용의 전체 코드입니다. pandas와 SQLite를 이용해 몇 줄 안 되는 코드로 과거 주식 데이터를 얻어오고 이를 저장할 수 있음을 확인할 수 있습니다.

예제 17.1 pandas와 SQLite(소스코드: book/ch17/01.py)

```
import pandas as pd
import pandas_datareader.data as web
import datetime
import sqlite3

start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2016, 6, 12)
df = web.DataReader("KRX:078930", "google", start, end)

con = sqlite3.connect("c:/Users/Jason/kospi.db")
df.to_sql('078930', con, if_exists='replace')

readed_df = pd.read_sql("SELECT * From '078930'", con, index_col = 'Date')
```

03 증권사 API를 이용한 주가 데이터 저장

이번 절에서는 증권사의 API를 이용해 주가 데이터를 다운로드하고 이를 데이터베이스로 저장하는 방법을 알아보겠습니다. 참고로 이번 절에서는 키움증권의 OpenAPI+를 사용하겠습니다.

종목 코드 리스트 얻어 오기

증권사의 API를 이용해 각 종목의 일봉 데이터를 내려받으려면 먼저 해당 종목의 종목 코드가 필요합니다. 종목 코드는 KRX 홈페이지와 같은 웹을 통해 내려받을 수도 있고 증권사의 API를 사용해도 내려받을 수 있습니다.

키움 OpenAPI+ 개발 가이드를 참조하면 그림 17.18과 같이 GetCodeListByMarket 메서드를 사용하면 각 시장에 속하는 종목의 종목 코드 리스트를 얻을 수 있습니다. 예를 들어, 유가증권시장에 속하는 종목의 종목 코드 리스트를 받으려면 GetCodeListByMarket 메서드의 인자로 "0"을 전달하면 됩니다.

14) GetCodeListByMarket

원형	BSTR GetCodeListByMarket(LPCTSTR sMarket)
설명	시장구분에 따른 종목코드를 반환한다.
입력값	sMarket – 시장구분
반환값	종목코드 리스트, 종목간 구분은 ';'이다.
비고	sMarket – 0:장내, 3:ELW, 4:뮤추얼펀드, 5:신주인수권, 6:리츠, 8:ETF, 9:하이일드펀드, 10:코스닥, 30:제3시장

그림 17.18 GetCodeListByMarket 메서드(출처: 키움 OpenAPI+ 개발 가이드)

예제 17.2는 키움증권 OpenAPI+를 이용해 코스닥시장의 종목 코드 리스트를 얻어오는 코드입니다. 여기서 Kiwoom 클래스는 파이썬에서 키움 OpenAPI+를 쉽게 사용할 수 있게 해주는 역할을 하는 클래스입니다. Kiwoom 클래스에 대한 인스턴스를 생성한 후 인스턴스 메서드를 호출함으로써 더욱 편리하게 키움 OpenAPI+를 사용할 수 있습니다.

예제 17.2 종목 코드 리스트 얻어 오기(소스코드: book/ch17/02.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QAxContainer import *
from PyQt5.QtCore import *
```

```
class Kiwoom(QAxWidget):
    def __init__(self):
        super().__init__()
        self._create_kiwoom_instance()
        self._set_signal_slots()

    def _create_kiwoom_instance(self):
        self.setControl("KHOPENAPI.KHOpenAPICtrl.1")

    def _set_signal_slots(self):
        self.OnEventConnect.connect(self._event_connect)

    def comm_connect(self):
        self.dynamicCall("CommConnect()")
        self.login_event_loop = QEventLoop()
        self.login_event_loop.exec_()

    def _event_connect(self, err_code):
        if err_code == 0:
            print("connected")
        else:
            print("disconnected")

        self.login_event_loop.exit()

    def get_code_list_by_market(self, market):
        code_list = self.dynamicCall("GetCodeListByMarket(QString)", market)
        code_list = code_list.split(';')
        return code_list[:-1]

if __name__ == "__main__":
    app = QApplication(sys.argv)
    kiwoom = Kiwoom()
    kiwoom.comm_connect()
    code_list = kiwoom.get_code_list_by_market('10')
    for code in code_list:
        print(code, end=" ")
```

Kiwoom 클래스를 이용해 종목 코드 리스트를 얻어 오기 전에 먼저 Kiwoom 클래스에 대해 간단히 살펴보겠습니다. 앞서 키움증권의 OpenAPI+가 제공하는 메서드를 호출하려면 QAxWidget 클래스의 인스턴스가 필요했습니다. Kiwoom 클래스에서는 QAxWidget 클래스를 상속받음으로써 Kiwoom 클래스의 인스턴스가 QAxWidget 클래스가 제공하는 메서드를 호출할 수 있게 했습니다.

```
class Kiwoom(QAxWidget):
    def __init__(self):
        super().__init__()
```

키움증권의 OpenAPI+를 사용하려면 먼저 COM 오브젝트를 생성해야 합니다. COM 오브젝트를 생성하려면 "KHOPENAPI.KHOpenAPICtrl.1"이라는 ProgID를 QAxWidget 클래스의 생성자로 전달하거나 setControl 메서드를 사용하면 됩니다.

Kiwoom 클래스에서는 OpenAPI+에 대한 객체를 생성하는 용도로 _create_kiwoom_instance 메서드를 구현한 후 Kiwoom 클래스의 생성자에서 _create_kiwoom_instance를 호출합니다. 메서드의 이름에 언더스코어(_)를 붙인 것은 _create_kiwoom_instance라는 메서드가 주로 Kiwoom 클래스의 메서드에서 호출되기 때문입니다.

```
class Kiwoom(QAxWidget):
    def __init__(self):
        super().__init__()
        self._create_kiwoom_instance()

    def _create_kiwoom_instance(self):
        self.setControl("KHOPENAPI.KHOpenAPICtrl.1")
```

키움증권의 OpenAPI+를 사용하려면 가장 먼저 로그인해야 합니다. Kiwoom 클래스에서는 로그인을 위해 comm_connect라는 메서드를 구현했습니다. 해당 메서드에서는 CommConnect 함수를 dynamicCall 함수를 통해 호출합니다.

```
def comm_connect(self):
    self.dynamicCall("CommConnect()")
    self.login_event_loop = QEventLoop()
    self.login_event_loop.exec_()
```

comm_connect 메서드가 제공하는 또 하나의 중요한 기능은 이벤트 루프를 만드는 것입니다. 앞서 대신증권의 CybosPlus나 이베스트증권의 xingAPI에서는 while 문을 사용해 이벤트가 발생할 때까지 프로그램이 종료되지 않게 했습니다. 13장에서 설명한 키움증권 기초 편에서는 PyQt를 이용해 프로그램을 작성했기 때문에 이미 내부에 이벤트 루프가 있어 따로 이벤트 루프를 만들 필요가 없었습니다.

그러나 만약 PyQt를 이용해 GUI 형태로 프로그램을 만들지 않는다면 명시적으로 이벤트 루프를 생성해야 합니다. 위의 코드에서는 PyQt의 QEventLoop 클래스의 인스턴스를 생성한 후 exec_ 메서드를 호출해 이벤트 루프를 생성했습니다.

comm_connect 메서드를 호출하면 키움증권의 로그인 창이 나타납니다. 로그인 정보를 입력하면 키움증권 서버로 로그인 정보가 전송됩니다. 키움증권은 로그인 요청을 받으면 OnEventConnect 이벤트를 발생시킵니다. Kiwoom 클래스에서는 comm_connect 메서드 내에서 이벤트 루프를 생성했기 때문에 키움증권 서버로부터 OnEventConnect 이벤트가 발생할 때까지 종료하지 않은 상태로 남아 있게 됩니다.

OnEventConnect 이벤트가 발생하면 Kiwoom 클래스의 _event_connect 메서드가 호출됩니다. 해당 메서드에서는 err_code 값에 따라 연결 상태를 화면에 출력한 후 comm_connect 메서드 호출 시 생성됐던 이벤트 루프를 종료합니다.

```
def _event_connect(self, err_code):
    if err_code == 0:
        print("connected")
    else:
        print("disconnected")

    self.login_event_loop.exit()
```

마지막으로 키움증권 서버로부터 발생한 이벤트(signal)와 이를 처리할 메서드(slot)를 연결하기 위해 _set_signal_slots 메서드를 추가합니다.

```
def _set_signal_slots(self):
    self.OnEventConnect.connect(self._event_connect)
```

그림 17.18의 GetCodeListByMarket 메서드를 호출하는 용도로 Kiwoom 클래스에도 get_code_list_by_market 메서드를 정의합니다. get_code_list_by_market에서는 OpenAPI+가 제공하는 GetCodeListByMarket 메서드를 dynamicCall 메서드를 통해 호출합니다.

```
def get_code_list_by_market(self, market):
    code_list = self.dynamicCall("GetCodeListByMarket(QString)", market)
    code_list = code_list.split(';')
    return code_list[:-1]
```

OpenAPI+의 GetCodeListByMarket 메서드는 종목 코드를 리턴하는데 다음과 같이 6자리 숫자로 구성된 종목 코드는 ';' 기호를 통해 서로 구분됩니다. 따라서 파이썬 문자열의 split 메서드를 이용해 종목 코드를 파이썬 리스트로 변환합니다.

```
900090;900080;900110;900130;900180;000250;
```

지금까지 Kiwoom 클래스에 대해 알아봤습니다. 이전과 다른 부분은 QEventLoop 클래스를 사용해 이벤트 루프를 생성한다는 점과 OpenAPI+의 메서드를 쉽게 사용하기 위한 메서드가 제공된다는 점이었습니다.

이번에는 지금까지 살펴본 Kiwoom 클래스를 이용해 종목 코드 리스트를 얻어와 보겠습니다. Kiwoom 클래스는 QAxWidget 클래스를 상속받았기 때문에 Kiwoom 클래스에 대한 인스턴스를 생성하려면 먼저 QApplication 클래스의 인스턴스를 생성해야 합니다. 그런 다음 Kiwoom 클래스에 대한 인스턴스를 생성할 수 있습니다.

```
app = QApplication(sys.argv)
kiwoom = Kiwoom()
```

정상적으로 kiwoom 객체가 생성됐다면 comm_connect 메서드를 호출해 로그인을 수행합니다.

```
kiwoom.comm_connect()
```

로그인 창이 나타나서 사용자가 로그인 정보를 입력해 정상적으로 로그인이 수행될 때까지 더는 파이썬 코드는 실행되지 않고 대기합니다. 이것은 comm_connect 메서드에서 QEventLoop 클래스를 통해 이벤트 루프를 생성했기 때문입니다.

정상적으로 로그인이 이뤄졌다면 comm_connect 코드 아래에 위치하는 다음 코드가 실행됩니다. 그림 17.18을 참조하면 코스닥시장은 '10'이므로 해당 값을 get_code_list_by_market 메서드의 인자로 사용해 메서드를 호출합니다.

```
code_list = kiwoom.get_code_list_by_market('10')
for code in code_list:
    print(code, end=" ")
```

리턴받은 종목 코드 리스트로부터 한 종목씩 화면에 출력해보면 그림 17.19와 같이 정상적으로 종목 코드를 받아왔음을 확인할 수 있습니다.



그림 17.19 코스닥시장 종목 코드 리스트

이번에는 종목 코드로부터 종목명을 얻어와 보겠습니다. 키움 OpenAPI+ 개발 가이드를 참조하면 종목 코드로 한글 종목명을 얻어오는 메서드는 GetMasterCodeName입니다. 그림 17.20을 참조하면 GetMasterCodeName 메서드는 종목 코드를 인자로 받음을 확인할 수 있습니다.

16) GetMasterCodeName

원형	BSTR GetMarsterCodeName(LPCTSTR strCode)
설명	종목코드의 한글명을 반환한다.
입력값	strCode – 종목코드
반환값	종목한글명
비고	장내외, 지수선옵, 주식선옵 검색 가능.

그림 17.20 GetMasterCodeName 메서드(출처: 키움 OpenAPI+ 개발 가이드)

OpenAPI+가 제공하는 GetMasterCodeName 메서드를 쉽게 사용하기 위해 앞에서 작성한 Kiwoom 메서드에 get_master_code_name 메서드를 구현합니다. 다음 코드를 참조하면 Kiwoom 클래스의 get_master_code_name 메서드에서는 self.dymaicCall 메서드를 통해 OpenAPI+가 제공하는 GetMasterCodeName을 호출한다는 것을 확인할 수 있습니다.

```
def get_master_code_name(self, code):
    code_name = self.dynamicCall("GetMasterCodeName(QString)", code)
    return code_name
```

한글 종목명을 얻어오는 메서드를 Kiwoom 클래스에 구현했으므로 이를 사용하는 코드를 구현해 보겠습니다. comm_connect 메서드를 호출해 로그인을 수행한 후 get_master_code_name 메서드를 호출하면 됩니다. 이때 인자로 문자열 타입의 종목 코드를 전달합니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    kiwoom = Kiwoom()

    kiwoom.comm_connect()
    print(kiwoom.get_master_code_name("000660"))
```

프로그램을 실행하면 다음과 같이 'SK하이닉스'라는 종목명이 정상적으로 출력되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch17/03.py
connected
SK하이닉스
```

```
[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

Process finished with exit code 0
```

일봉 데이터 연속조회

이번 절에서는 키움증권의 OpenAPI+를 통해 일봉 데이터를 얻어와 보겠습니다. 키움증권의 KOA Studio를 참고하면 일봉 데이터 요청에는 'opt10081'이라는 TR을 사용하면 됩니다.

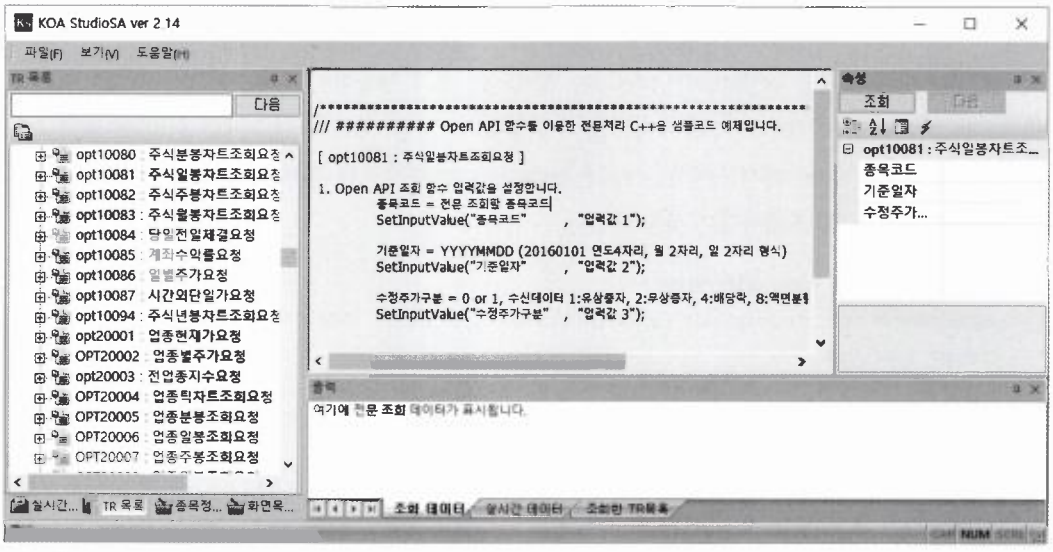


그림 17.21 opt10081을 이용한 주식 일봉 차트 조회 요청

OpenAPI+가 제공하는 TR을 사용하려면 먼저 SetInputValue 메서드를 통해 요청하는 TR에 필요한 데이터를 설정한 후 CommRqData 메서드를 호출해 TR을 키움증권 서버로 전송하면 됩니다. TR을 요청받은 서버는 이를 처리한 후 데이터가 준비되면 이벤트를 발생시켜 사용자가 데이터를 가져갈 수 있게 알려줍니다.

이번 절에서는 Kiwoom 클래스에 TR을 쉽게 사용할 수 있도록 몇 개의 메서드를 추가해 보겠습니다. 이번 절에서 작성할 전체 코드는 예제 17.3과 같습니다. 코드가 길어 보이지만 Kiwoom 클래스의 대부분은 앞에서 작성했습니다.

예제 17.3 opt10081TR을 이용한 일봉 차트 데이터 연속 조회(소스코드: book/ch17/04.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QAxContainer import *
from PyQt5.QtCore import *
import time

TR_REQ_TIME_INTERVAL = 0.2

class Kiwoom(QAxWidget):
    def __init__(self):
        super().__init__()
        self._create_kiwoom_instance()
        self._set_signal_slots()

    def _create_kiwoom_instance(self):
        self.setControl("KHOPENAPI.KHOpenAPICtrl.1")

    def _set_signal_slots(self):
        self.OnEventConnect.connect(self._event_connect)
        self.OnReceiveTrData.connect(self._receive_tr_data)

    def comm_connect(self):
        self.dynamicCall("CommConnect()")
        self.login_event_loop = QEventLoop()
        self.login_event_loop.exec_()

    def _event_connect(self, err_code):
        if err_code == 0:
            print("connected")
        else:
            print("disconnected")

        self.login_event_loop.exit()

    def get_code_list_by_market(self, market):
        code_list = self.dynamicCall("GetCodeListByMarket(QString)", market)
        code_list = code_list.split(';')
        return code_list[:-1]

    def get_master_code_name(self, code):
```

```
        code_name = self.dynamicCall("GetMasterCodeName(QString)", code)
        return code_name

    def set_input_value(self, id, value):
        self.dynamicCall("SetInputValue(QString, QString)", id, value)

    def comm_rq_data(self, rqname, trcode, next, screen_no):
        self.dynamicCall("CommRqData(QString, QString, int, QString)", rqname, trcode, next,
screen_no)
        self.tr_event_loop = QEventLoop()
        self.tr_event_loop.exec_()

    def _comm_get_data(self, code, real_type, field_name, index, item_name):
        ret = self.dynamicCall("CommGetData(QString, QString, QString, int, QString)", code,
                               real_type, field_name, index, item_name)
        return ret.strip()

    def _get_repeat_cnt(self, trcode, rqname):
        ret = self.dynamicCall("GetRepeatCnt(QString, QString)", trcode, rqname)
        return ret

    def _receive_tr_data(self, screen_no, rqname, trcode, record_name, next, unused1, unused2,
unused3, unused4):
        if next == '2':
            self.remained_data = True
        else:
            self.remained_data = False

        if rqname == "opt10081_req":
            self._opt10081(rqname, trcode)

        try:
            self.tr_event_loop.exit()
        except AttributeError:
            pass

    def _opt10081(self, rqname, trcode):
        data_cnt = self._get_repeat_cnt(trcode, rqname)

        for i in range(data_cnt):
            date = self._comm_get_data(trcode, "", rqname, i, "일자")
            open = self._comm_get_data(trcode, "", rqname, i, "시가")
```

```
            high = self._comm_get_data(trcode, "", rqname, i, "고가")
            low = self._comm_get_data(trcode, "", rqname, i, "저가")
            close = self._comm_get_data(trcode, "", rqname, i, "현재가")
            volume = self._comm_get_data(trcode, "", rqname, i, "거래량")
            print(date, open, high, low, close, volume)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    kiwoom = Kiwoom()
    kiwoom.comm_connect()

    # opt10081 TR 요청
    kiwoom.set_input_value("종목코드", "039490")
    kiwoom.set_input_value("기준일자", "20170224")
    kiwoom.set_input_value("수정주가구분", 1)
    kiwoom.comm_rq_data("opt10081_req", "opt10081", 0, "0101")

    while kiwoom.remained_data == True:
        time.sleep(TR_REQ_TIME_INTERVAL)
        kiwoom.set_input_value("종목코드", "039490")
        kiwoom.set_input_value("기준일자", "20170224")
        kiwoom.set_input_value("수정주가구분", 1)
        kiwoom.comm_rq_data("opt10081_req", "opt10081", 2, "0101")
```

가장 먼저 Kiwoom 클래스에 추가할 메서드는 set_input_value입니다. set_input_value 메서드는 OpenAPI+에서 제공하는 SetInputValue 메서드를 dynamicCall 메서드를 통해 호출하는 역할을 수행합니다.

7) SetInputValue

원형	void SetInputValue(BSTR sID, BSTR sValue)
설명	Tran 입력 값을 서버통신 전에 입력한다.
입력값	sID – 아이템명 sValue – 입력 값
반환값	없음
비고	Ex) openApi.SetInputValue("종목코드", "000660"); openApi.SetInputValue("계좌번호", "5015123401");

그림 17.22 SetInputValue 메서드(출처: 키움 OpenAPI+ 개발가이드)

```
def set_input_value(self, id, value):
    self.dynamicCall("SetInputValue(QString, QString)", id, value)
```

이번에는 comm_rq_data 메서드를 추가해 보겠습니다. 이를 위해 OpenAPI+ 개발 가이드에서 CommRqData 메서드의 원형을 살펴보겠습니다. 그림 17.23과 같이 네 개의 메서드 인자를 받는 것을 확인할 수 있습니다.

3) **CommRqData**

원형	LONG CommRqData(BSTR sRQName, BSTR sTrCode, long nPrevNext, BSTR sScreenNo)
설명	Tran을 서버로 송신한다.
입력값	BSTR sRQName
	BSTR sTrCode long nPrevNext BSTR sScreenNo
반환값	OP_ERR_SISE_OVERFLOW – 과도한 시세조회로 인한 통신불가 OP_ERR_RQ_STRUCT_FAIL – 입력 구조체 생성 실패 OP_ERR_RQ_STRING_FAIL – 요청전문 작성 실패 OP_ERR_NONE – 정상처리
비고	sRQName – 사용자구분 명 sTrCode - Tran명 입력 nPrevNext - 0:조회, 2:연속 sScreenNo - 4자리의 화면번호 Ex) openApi.CommRqData("RQ_1", "OPT00001", 0, "0101");

그림 17.23 CommRqData 메서드(출처: 키움 OpenAPI+ 개발가이드)

다음 코드와 같이 Kiwoom 클래스에 comm_rq_data 메서드를 구현합니다. comm_rq_data는 self.dynamicCall 메서드를 이용해 OpenAPI+의 CommRqData 메서드를 호출하는 코드와 QEventLoop 클래스의 인스턴스를 생성한 후 이벤트 루프를 만드는 코드로 구성됩니다.

CommRqData를 통해 키움증권 서버에 TR을 요청하면 데이터가 바로 반환되는 것이 아닙니다. 키움증권 서버는 TR을 처리한 후 이벤트를 통해 알려주기 때문에 키움증권이 이벤트를 줄 때까지 대기하고 있어야 합니다. 따라서 CommRqData를 호출한 후에 이벤트 루프를 만들어주는 코드가 반드시 있어야 합니다. 이 부분은 앞서 로그인을 수행할 때 이벤트 루프를 만든 것과 동일합니다.

```
def comm_rq_data(self, rqname, trcode, next, screen_no):
    self.dynamicCall("CommRqData(QString, QString, int, QString)", rqname, trcode, next, screen_
no)
    self.tr_event_loop = QEventLoop()
    self.tr_event_loop.exec_()
```

키움증권 서버에 TR을 요청하면 한 번에 여러 개의 데이터가 반환됩니다. 예를 들어, opt10081을 통해 일봉 데이터를 요청하는 경우 약 900개의 데이터(거래일 기준)가 반환됩니다. 데이터를 받는 입장에서

는 총 몇 개의 데이터가 왔는지 알고 있어야 for나 while과 같은 반복문을 통해 데이터를 읽을 수 있습니다. 이러한 용도로 사용하는 메서드가 _get_repeat_cnt입니다. 역시 OpenAPI+의 개발 가이드를 참조해서 다음과 같이 Kiwoom 클래스에 구현합니다.

```
def _get_repeat_cnt(self, trcode, rqname):
    ret = self.dynamicCall("GetRepeatCnt(QString, QString)", trcode, rqname)
    return ret
```

키움증권 서버로부터 TR 처리에 대한 이벤트가 발생했을 때 실제로 데이터를 가져오려면 CommGetData 메서드를 사용해야 합니다. 역시 OpenAPI+의 개발 가이드를 참조해서 다음과 같이 _comm_get_data 메서드를 Kiwoom 클래스에 구현합니다. OpenAPI+의 CommGetData 메서드의 반환값은 문자열 타입으로 양쪽에 공백이 있습니다. 따라서 공백을 제거하기 위해 반환받은 문자열에 strip 메서드를 사용했습니다.

```
def _comm_get_data(self, code, real_type, field_name, index, item_name):
    ret = self.dynamicCall("CommGetData(QString, QString, QString, int, QString)", code,
                           real_type, field_name, index, item_name)
    return ret.strip()
```

마지막으로 이벤트가 발생했을 때 이를 처리하는 메서드를 구현하겠습니다. OpenAPI+ 개발 가이드를 참조하면 TR 요청의 결과로 OnReceiveTrData 이벤트가 발생합니다. 개발 가이드를 참조해서 다음과 같이 동일한 이름으로 메서드를 정의합니다.

```
def _receive_tr_data(self, screen_no, rqname, trcode, record_name, next, unused1, unused2, un-
used3, unused4):
    pass
```

위에서 구현한 _receive_tr_data라는 메서드가 'OnReceiveTrData' 이벤트가 발생할 때 호출되게 하려면 시그널과 슬롯을 연결해 주면 되지요? 다음과 같이 _set_signal_slots 메서드를 수정합니다.

```
def _set_signal_slots(self):
    self.OnEventConnect.connect(self._event_connect)
    self.OnReceiveTrData.connect(self._receive_tr_data)
```

Kiwoom 클래스 내부에 구현한 _receive_tr_data 메서드가 호출됐다는 것은 키움증권 서버로부터 'OnReceiveTrData' 이벤트가 발생했음을 의미합니다. 따라서 _receive_tr_data 메서드 내에서 서버로부터 데이터를 받아오면 됩니다. 앞서 메서드의 원형만 구현했던 _receive_tr_data 메서드를 다음과 같이 수정합니다.

```
def _receive_tr_data(self, screen_no, rqname, trcode, record_name, next, unused1, unused2, un-
used3, unused4):
    if next == '2':
        self.remained_data = True
    else:
        self.remained_data = False

    if rqname == "opt10081_req":
        self._opt10081(rqname, trcode)

    try:
        self.tr_event_loop.exit()
    except AttributeError:
        pass
```

_receive_tr_data 메서드를 구현할 때는 다음의 세 가지를 고려해야 합니다.

첫 번째는 여러 종류의 TR을 요청해도 모두 _receive_tr_data 메서드 내에서 처리해야 한다는 것입니다. 따라서 rqname이라는 인자를 통해 요청한 TR을 구분한 후 TR에 따라 적당한 데이터를 가져오도록 코드를 작성해야 합니다.

위 코드를 살펴보면 if 구문을 사용해 rqname 값이 'opt10081_req'일 때 _opt10081 메서드를 호출하는 것을 확인할 수 있습니다. 이처럼 TR을 요청할 때 여러분이 TR을 구분하기 위한 적당한 문자열을 사용하고, 데이터를 받을 때는 해당 문자열과 같은지 확인함으로써 올바른 데이터를 얻을 수 있습니다.

두 번째는 연속조회에 대한 처리입니다. 연속조회는 아직 데이터가 더 존재한다는 것을 의미합니다. 예를 들어, 10년 치 일봉 차트 데이터가 존재한다면 데이터의 개수가 수천 개에 이를 것입니다. 앞서 TR을 한 번 요청하면 900개의 데이터가 반환된다고 했으므로 10년 치 데이터를 한 번의 TR 요청으로는 받을 수 없습니다. 이런 경우에는 연속해서 같은 TR을 또 요청해서 남아있는 데이터를 받아야 합니다.

키움증권 서버는 OnReceiveTrData 이벤트가 발생할 때 PrevNext라는 인자 값을 통해 연속조회가 필요한 경우 PrevNext 값을 2로 리턴합니다. 따라서 PrevNext 값을 보고 그 값이 2이면 아직 데이터가 더 있다는 사실을 알 수 있고, 이 경우 동일한 TR을 한 번 더 요청해서 남은 데이터를 가져와야 합니다. 위 코드에서는 _receive_tr_data 메서드의 인자인 next의 값이 '2'일 때 self.remained_data라는 변수에 True를 저장합니다. 이 값을 보고 TR을 더 요청하는 코드는 조금 후에 다시 설명하겠습니다.

세 번째 고려사항은 이벤트 루프에 대한 처리입니다. 앞서 CommRqData를 호출할 때 이벤트 루프를 생성했습니다. 이벤트 루프 덕분에 키움증권 서버가 'OnReceiveTrData' 이벤트를 보낼 때까지 대기할 수 있었습니다.

Kiwoom 클래스의 _receive_tr_data 메서드가 호출됐다는 것은 여러분이 구현한 프로그램이 정상적으로 대기 상태에서 머무르다가 서버로부터 발생한 'OnReceiveTrData' 이벤트를 받았음을 의미합니다. 따라서 _receive_tr_data 메서드에서는 더는 필요하지 않은 이벤트 루프를 종료하면 됩니다.

```
try:
    self.tr_event_loop.exit()
except AttributeError:
    pass
```

_receive_tr_data 메서드에서 데이터를 얻어오는 부분을 살펴보겠습니다. opt10081_req에 대한 데이터는 _opt10081 메서드에서 처리합니다. 이는 각 TR에 대한 데이터를 얻어가는 코드를 _receive_tr_data에 모두 구현하기보다는 메서드로 따로 빼서 구현한 것입니다.

앞서 TR을 요청하면 약 900개의 데이터가 반환된다고 했는데 데이터가 그보다 적다면 더 적은 수가 리턴될 것입니다. 예를 들어, 상장한 지 얼마 되지 않은 종목이 여기에 해당할 것입니다. 따라서 데이터를 얻어오기 전에 먼저 데이터의 개수를 _get_repeat_cnt 메서드를 통해 얻어옵니다.

```
def _opt10081(self, rqname, trcode):
    data_cnt = self._get_repeat_cnt(trcode, rqname)
```

데이터의 총 개수를 얻었다면 반복문을 통해 데이터를 하나씩 얻어오면 됩니다. 일봉 데이터는 거래일 단위로 '종목코드', '현재가', '거래량', '거래대금', '일자', '시가', '고가', '저가', '수정주가구분', '수정비율', '대업종구분', '소업종구분', '종목정보', '수정주가이벤트', '전일종가'를 한 번에 제공합니다. 여기서는 '일자', '시가', '고가', '저가', '현재가', '거래량'만 필요하므로 다음과 같이 코드를 작성하면 됩니다.

```
def _opt10081(self, rqname, trcode):
    data_cnt = self._get_repeat_cnt(trcode, rqname)

    for i in range(data_cnt):
        date = self._comm_get_data(trcode, "", rqname, i, "일자")
        open = self._comm_get_data(trcode, "", rqname, i, "시가")
        high = self._comm_get_data(trcode, "", rqname, i, "고가")
        low = self._comm_get_data(trcode, "", rqname, i, "저가")
        close = self._comm_get_data(trcode, "", rqname, i, "현재가")
        volume = self._comm_get_data(trcode, "", rqname, i, "거래량")
        print(date, open, high, low, close, volume)
```

이로써 opt10081 TR을 사용할 수 있도록 Kiwoom 클래스를 업데이트했습니다. 이번에는 Kiwoom 클래스를 사용해 실제로 일봉 데이터를 얻어오는 코드를 작성해 보겠습니다. 먼저 QApplication 클래스 인스턴스와 Kiwoom 클래스 인스턴스를 생성한 후 로그인을 수행하는 코드는 앞의 예제와 같습니다.

```
app = QApplication(sys.argv)
kiwoom = Kiwoom()
kiwoom.comm_connect()
```

연속조회가 가능한 TR의 처리는 다음과 같이 코드를 작성하면 됩니다. '종목코드'에 실제로 조회할 종목 코드를 입력하고, '기준일자' 항목에는 조회를 시작할 일자에 대해 '20170224'와 같은 형식으로 입력하면 됩니다. '수정주가구분'에는 1을 입력해서 수정주가를 요청했습니다.

comm_rq_data 메서드를 호출할 때 주의할 점은 첫 번째 인자의 이름입니다. Kiwoom 클래스의 _receive_tr_data 메서드에서 'opt10081_req'라는 문자열로 TR을 구분했기 때문에 같은 이름으로 TR을 요청해야 합니다.

```
# opt10081 TR 요청
kiwoom.set_input_value("종목코드", "039490")
kiwoom.set_input_value("기준일자", "20170224")
kiwoom.set_input_value("수정주가구분", 1)
kiwoom.comm_rq_data("opt10081_req", "opt10081", 0, "0101")

while kiwoom.remained_data == True:
    time.sleep(TR_REQ_TIME_INTERVAL)
    kiwoom.set_input_value("종목코드", "039490")
    kiwoom.set_input_value("기준일자", "20170224")
    kiwoom.set_input_value("수정주가구분", 1)
    kiwoom.comm_rq_data("opt10081_req", "opt10081", 2, "0101")
```

TR을 요청할 때는 먼저 한 번 요청을 한 후 while 문을 사용해 인스턴스의 remained_data 값이 True인 경우 연속조회 데이터가 존재한다는 것을 의미하므로 다시 한 번 동일한 TR을 요청하도록 구현합니다. 키움증권은 1초에 최대 5번의 TR 요청만 허용하고 있으므로 time 모듈의 sleep 함수를 통해 0.2초를 대기한 후 다음 TR을 요청하도록 구현했습니다.[1]

연속조회를 할 때 주의할 점은 comm_rq_data 메서드의 세 번째 인자 값이 처음 요청할 때는 0이지만 두 번째부터는 2를 전달해야 한다는 점입니다. 모든 코드가 이해됐다면 예제 17.3의 코드를 실행해 봅니다. 그림 17.24와 같이 일봉 차트 데이터가 정상적으로 출력되는 것을 확인할 수 있습니다.

1 조회횟수 제한(1초당 5회)은 종종 수정되므로 키움증권 고객문의 게시판을 참고하세요.

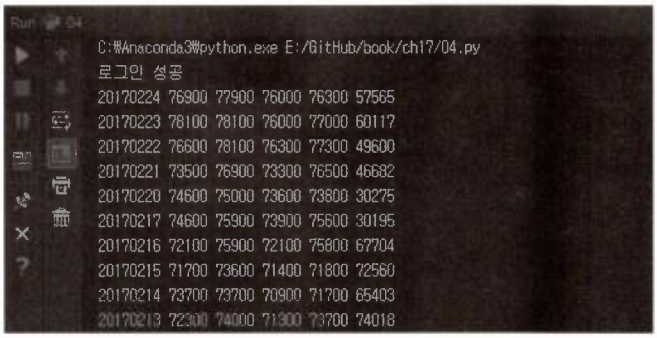


그림 17.24 일봉 차트 조회 요청 결과

SQLite DB에 일봉 데이터 저장하기

이번 절에서는 앞서 개발한 Kiwoom 클래스를 바탕으로 일 단위로 시가, 고가, 저가, 종가, 거래량 데이터를 얻어온 후 이를 SQLite 데이터베이스로 저장해보겠습니다. 이번 절에서 사용할 전체 코드는 예제 17.4와 같습니다.

예제 17.4 일봉 데이터를 SQLite에 저장하기 (소스코드: book/ch17/05.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QAxContainer import *
from PyQt5.QtCore import *
import time
import pandas as pd
import sqlite3

TR_REQ_TIME_INTERVAL = 0.2

class Kiwoom(QAxWidget):
    def __init__(self):
        super().__init__()
        self._create_kiwoom_instance()
        self._set_signal_slots()

    def _create_kiwoom_instance(self):
        self.setControl("KHOPENAPI.KHOpenAPICtrl.1")

    def _set_signal_slots(self):
```

```
        self.OnEventConnect.connect(self._event_connect)
        self.OnReceiveTrData.connect(self._receive_tr_data)

    def comm_connect(self):
        self.dynamicCall("CommConnect()")
        self.login_event_loop = QEventLoop()
        self.login_event_loop.exec_()

    def _event_connect(self, err_code):
        if err_code == 0:
            print("connected")
        else:
            print("disconnected")

        self.login_event_loop.exit()

    def get_code_list_by_market(self, market):
        code_list = self.dynamicCall("GetCodeListByMarket(QString)", market)
        code_list = code_list.split(';')
        return code_list[:-1]

    def get_master_code_name(self, code):
        code_name = self.dynamicCall("GetMasterCodeName(QString)", code)
        return code_name

    def set_input_value(self, id, value):
        self.dynamicCall("SetInputValue(QString, QString)", id, value)

    def comm_rq_data(self, rqname, trcode, next, screen_no):
        self.dynamicCall("CommRqData(QString, QString, int, QString)", rqname, trcode, next,
screen_no)
        self.tr_event_loop = QEventLoop()
        self.tr_event_loop.exec_()

    def _comm_get_data(self, code, real_type, field_name, index, item_name):
        ret = self.dynamicCall("CommGetData(QString, QString, QString, int, QString)", code,
                               real_type, field_name, index, item_name)
        return ret.strip()

    def _get_repeat_cnt(self, trcode, rqname):
        ret = self.dynamicCall("GetRepeatCnt(QString, QString)", trcode, rqname)
        return ret
```

```
    def _receive_tr_data(self, screen_no, rqname, trcode, record_name, next, unused1, unused2,
unused3, unused4):
        if next == '2':
            self.remained_data = True
        else:
            self.remained_data = False

        if rqname == "opt10081_req":
            self._opt10081(rqname, trcode)

        try:
            self.tr_event_loop.exit()
        except AttributeError:
            pass

    def _opt10081(self, rqname, trcode):
        data_cnt = self._get_repeat_cnt(trcode, rqname)

        for i in range(data_cnt):
            date = self._comm_get_data(trcode, "", rqname, i, "일자")
            open = self._comm_get_data(trcode, "", rqname, i, "시가")
            high = self._comm_get_data(trcode, "", rqname, i, "고가")
            low = self._comm_get_data(trcode, "", rqname, i, "저가")
            close = self._comm_get_data(trcode, "", rqname, i, "현재가")
            volume = self._comm_get_data(trcode, "", rqname, i, "거래량")

            self.ohlcv['date'].append(date)
            self.ohlcv['open'].append(int(open))
            self.ohlcv['high'].append(int(high))
            self.ohlcv['low'].append(int(low))
            self.ohlcv['close'].append(int(close))
            self.ohlcv['volume'].append(int(volume))

if __name__ == "__main__":
    app = QApplication(sys.argv)
    kiwoom = Kiwoom()
    kiwoom.comm_connect()
    kiwoom.ohlcv = {'date': [], 'open': [], 'high': [], 'low': [], 'close': [], 'volume': []}

    # opt10081 TR 요청
    kiwoom.set_input_value("종목코드", "039490")
    kiwoom.set_input_value("기준일자", "20170224")
```

```
        kiwoom.set_input_value("수정주가구분", 1)
        kiwoom.comm_rq_data("opt10081_req", "opt10081", 0, "0101")

        while kiwoom.remained_data == True:
            time.sleep(TR_REQ_TIME_INTERVAL)
            kiwoom.set_input_value("종목코드", "039490")
            kiwoom.set_input_value("기준일자", "20170224")
            kiwoom.set_input_value("수정주가구분", 1)
            kiwoom.comm_rq_data("opt10081_req", "opt10081", 2, "0101")

        df = pd.DataFrame(kiwoom.ohlcv, columns=['open', 'high', 'low', 'close', 'volume'], index-
    =kiwoom.ohlcv['date'])

        con = sqlite3.connect("c:/Users/Jason/stock.db")
        df.to_sql('039490', con, if_exists='replace')
```

Kiwoom 클래스의 대부분은 앞서 구현한 바와 같기 때문에 업데이트된 부분만 설명하겠습니다. 앞서 _opt10081 메서드에서는 시가, 고가, 저가, 종가, 거래량 데이터를 화면에 출력했지만 이번에는 이 데이터를 SQLite 데이터베이스로 저장하려고 합니다. 따라서 일봉 데이터를 pandas의 DataFrame 객체로 만들 필요가 있습니다.

파이썬 자료구조를 DataFame 객체로 만드는 가장 쉬운 방법은 파이썬 딕셔너리를 사용하는 것이었습니다. 다음과 같이 Kiwoom 클래스의 인스턴스를 생성한 후 ohlcv라는 딕셔너리를 추가합니다.

```
app = QApplication(sys.argv)
kiwoom = Kiwoom()
kiwoom.comm_connect()
kiwoom.ohlcv = {'date': [], 'open': [], 'high': [], 'low': [], 'close': [], 'volume': []}
```

데이터를 저장할 기본적인 자료구조가 준비됐으므로 이제 _opt10081 메서드에서 데이터를 출력했던 코드를 자료구조 안으로 저장하도록 코드를 수정하면 됩니다. print 코드 대신 다음과 같이 딕셔너리의 각 키에 대응되는 파이썬 리스트에 값을 추가합니다. 이때 날짜를 제외한 시가, 고가, 저가, 종가, 거래량 데이터는 정수 타입으로 변환한 후 리스트에 삽입합니다.

```
self.ohlcv['date'].append(date)
self.ohlcv['open'].append(int(open))
self.ohlcv['high'].append(int(high))
self.ohlcv['low'].append(int(low))
self.ohlcv['close'].append(int(close))
self.ohlcv['volume'].append(int(volume))
```

Kiwoom 클래스의 인스턴스를 통해 일봉 데이터를 내려받으면 ohlcv라는 딕셔너리에 일봉 데이터가 저장됩니다. ohlcv 딕셔너리를 먼저 DataFrame 객체로 변환합니다. 이때 인덱스를 지정하고 칼럼 순서도 지정합니다.

```
df = pd.DataFrame(kiwoom.ohlcv, columns=['open', 'high', 'low', 'close', 'volume'], index=ki-
woom.ohlcv['date'])
```

sqlite3 모듈의 connect 메서드를 호출해 데이터베이스 파일을 엽니다. 그런 다음 DataFrame 객체의 to_sql 메서드를 호출해 DataFrame 객체를 데이터베이스로 저장하면 됩니다.

```
con = sqlite3.connect("c:/Users/Jason/stock.db")
df.to_sql('039490', con, if_exists='replace')
```

코드를 실행한 후 DB Browser for SQLite 프로그램을 통해 'stock.db'라는 파일을 엽니다. 그림 17.25와 같이 정상적으로 데이터가 저장된 것을 확인할 수 있습니다.

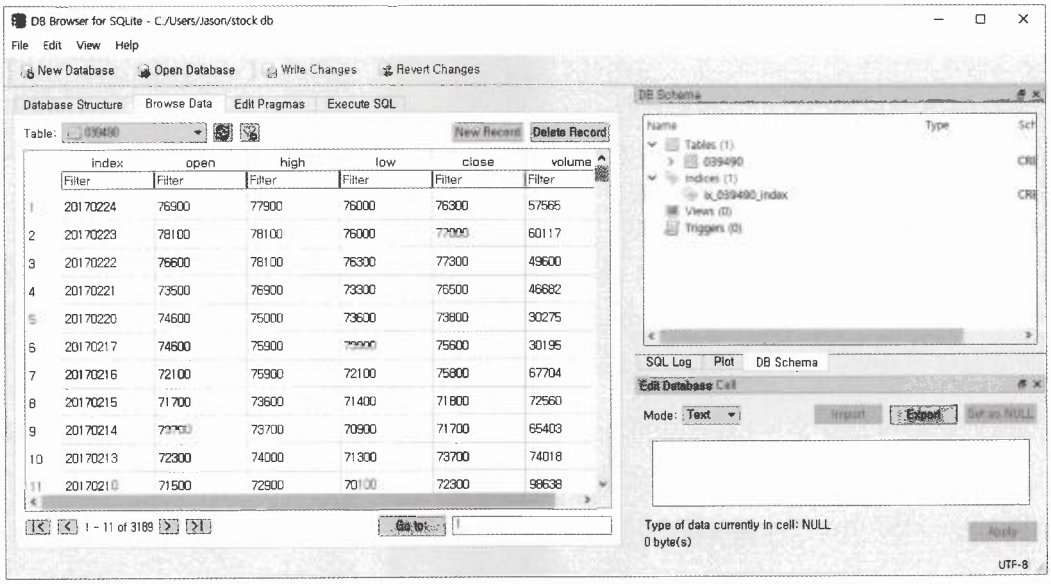


그림 17.25 DB Browser for SQLite를 통한 테이블 확인

18

실전 프로그램 개발

지금까지 파이썬 기본 문법, 증권사 API 기본 사용법, 데이터 분석 기초, 백테스팅, GUI 프로그래밍, 데이터베이스에 대해 배웠습니다. 사실 이러한 주제는 별도로 다룬 책이 있을 만큼 규모가 큰 주제입니다. 따라서 각 분야를 제대로 이해하려면 각각을 자세히 다룬 책을 통해 더 깊이 있게 공부해야 합니다.

그러나 직장인이 모든 분야를 다 공부한 후에 실전에 사용할 시스템 트레이딩 프로그램을 개발하기에는 시간이 부족합니다. 또한 그러한 방식으로는 미처 프로그램을 완성하기도 전에 지쳐서 포기하고 말 것입니다. 당장은 아는 것이 많지 않더라도 간단한 프로그램부터 개발해볼 필요가 있습니다.

이번 장에서는 키움증권의 OpenAPI+를 이용해 시스템 트레이딩의 기초가 되는 실전 프로그램을 개발하겠습니다. 실전 프로그램을 직접 개발해보면서 프로그램 개발의 전반적인 과정을 이해할 수 있으며, 이를 통해 앞으로는 자신만의 투자 알고리즘을 이용해 더욱 복잡한 프로그램도 직접 개발할 수 있을 것입니다.

01 개발 1일 차

이번 장에서는 한 프로그램의 전체 개발 과정을 여러 단계로 나눠 설명하겠습니다. 편의상 일별로 개발을 진행한다고 가정하겠습니다.

개발 1일 차에는 그림 18.1의 PyTrader 프로그램을 개발하고자 합니다. PyTrader는 키움 OpenAPI+를 이용한 자동 로그인을 지원합니다. PyTrader는 PyQt를 통해 개발됐으며 윈도우의 상태 표시줄에 현재 시각과 키움증권 연결 여부를 매초 출력하는 기능이 구현돼 있습니다.

그뿐만 아니라 윈도우의 작업 스케줄러를 이용해 PyTrader 프로그램을 정해진 시간에 자동으로 실행되게 할 것입니다. 또한 키움증권의 버전 처리를 자동으로 수행하기 위해 키움 번개3을 자동으로 로그인하고 종료하는 파이썬 스크립트도 작성하겠습니다.

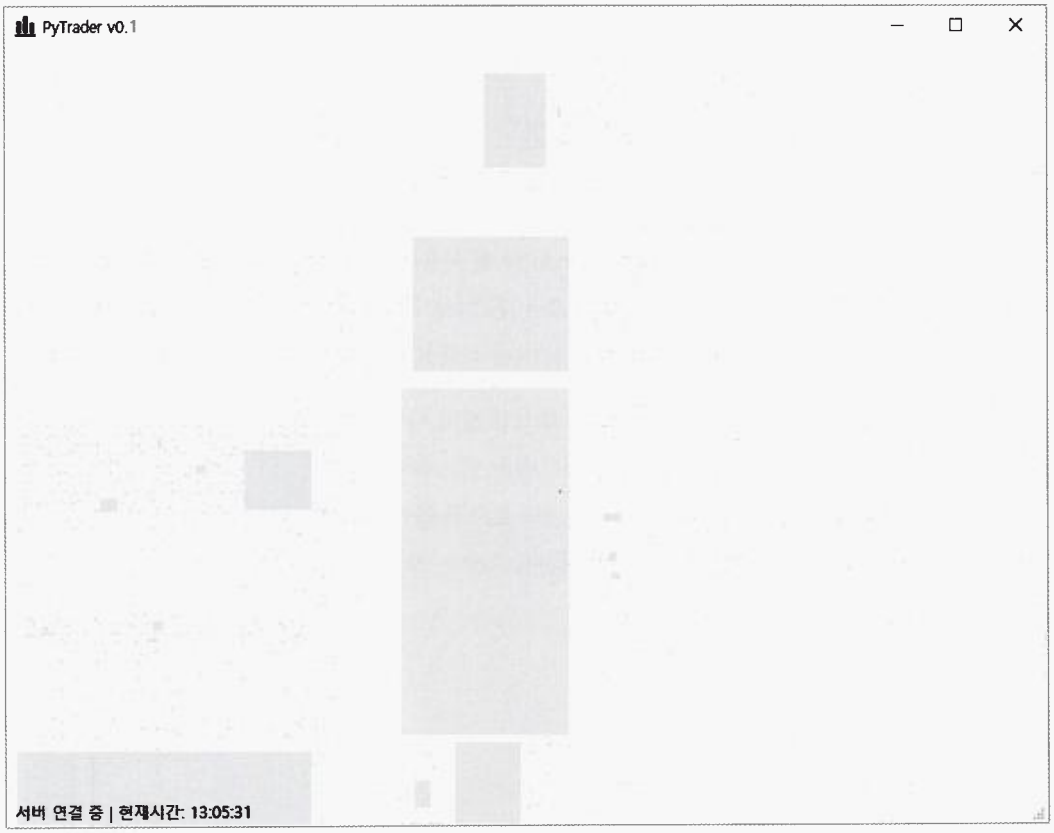


그림 18.1 개발 1일차에 개발한 프로그램의 실행 결과

자동 버전 처리 스크립트

키움 OpenAPI+도 하나의 프로그램이기 때문에 주기적으로 업데이트됩니다. 이 경우 PC에 설치된 키움 OpenAPI+ 모듈을 업데이트해야 합니다. 키움증권에서는 이러한 과정을 버전 처리라고 부릅니다.

OpenAPI+를 이용해 개발한 프로그램에서 로그인을 시도할 때 버전 처리가 필요하면 버전 처리 메시지 창이 그림 18.2와 같이 나타납니다. 이 경우 먼저 여러분이 개발한 프로그램을 완전히 종료한 후 버전 처리 메시지 창의 [확인] 버튼을 눌러야 정상적으로 버전 처리가 진행됩니다. 이러한 버전 처리는 자동화의 큰 걸림돌입니다. 또한 버전 처리가 주기적인 것이 아니라 언제 발생하는지 알기 어려워 자동화하기도 쉽지 않습니다.

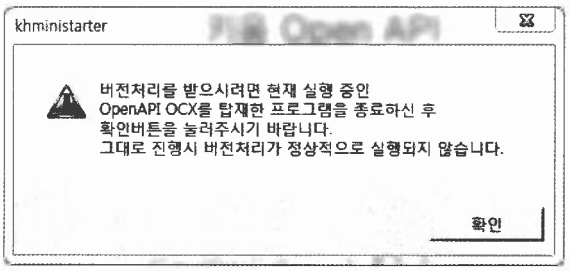


그림 18.2 버전 처리 메시지 창

버전 처리를 가장 쉽게 할 수 있는 방법은 OpenAPI+를 사용하기 위해 함께 설치했던 번개 HTS를 사용하는 것입니다. 번개 HTS를 실행하면 자동으로 버전 처리가 완료되기 때문에 OpenAPI+를 사용해서 개발한 프로그램을 실행하기 전에 먼저 번개 HTS를 실행하기만 하면 버전 처리 문제가 해결됩니다.

번개를 실행하면 그림 18.3과 같이 번개 HTS 로그인 창이 나타납니다. 번개 HTS를 실행하는 작업 자체는 앞으로 배울 윈도우 작업 스케줄러를 이용해서 처리하면 되지만 그림 18.3의 로그인 창에 비밀번호와 인증비밀번호를 입력한 후 [로그인] 버튼을 누르는 작업도 자동으로 진행해야 할 것입니다.

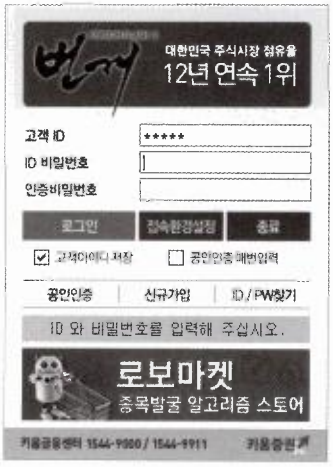


그림 18.3 키움번개 로그인 창

다행히도 이러한 형태의 자동화는 pywinauto 패키지를 이용하면 해결할 수 있습니다. pywinauto 패키지는 그림 18.3과 같은 윈도우 대화상자에 자동으로 마우스나 키보드 이벤트를 보낼 수 있습니다. 즉, 여러분이 비밀번호를 입력하기 위해 마우스로 클릭한 후 비밀번호를 키보드로 입력하고 다시 인증비밀번호 입력을 위해 마우스를 클릭한 후 인증비밀번호를 키보드로 입력하는 일련의 과정을 모두 파이썬 코드로 처리할 수 있습니다.

이를 위해 먼저 pywinauto 패키지를 설치해 봅시다. 아나콘다는 pywinauto 설치를 아직 지원하지 않으므로 그림 18.4와 같이 윈도우 명령 프롬프트를 실행한 후 'pip install pywinauto'를 입력해 패키지를 설치합니다.

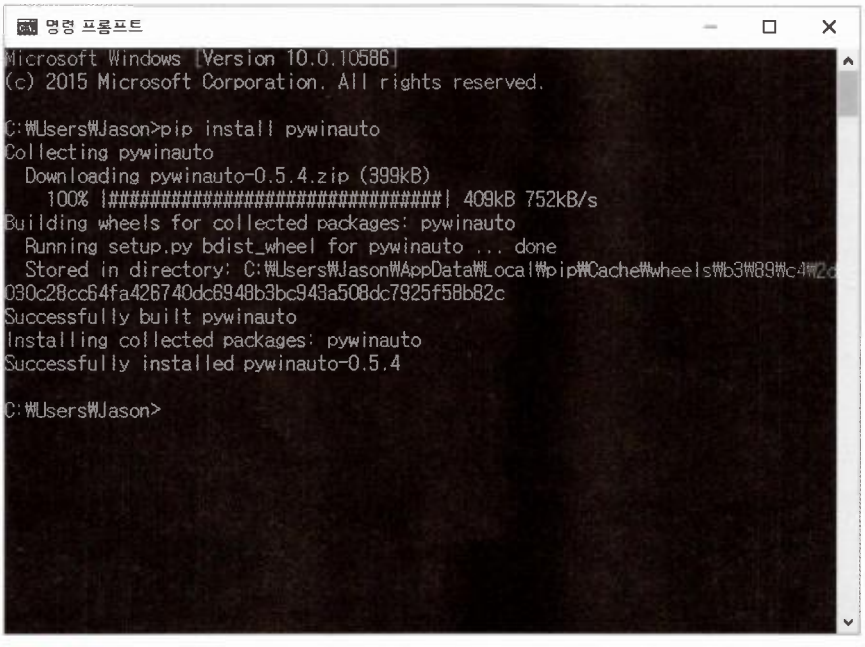


그림 18.4 pywinauto 모듈 설치

pywinauto 패키지를 이용해 코드를 작성하기 위에 먼저 PyCharm과 같은 파이썬 IDE를 관리자 권한으로 실행합니다. 앞으로 작성할 코드에서는 키움 번개3을 실행하는 코드가 포함돼 있으므로 반드시 IDE도 관리자 권한으로 실행해야 합니다.

PyCharm이 실행되면 파이썬 파일을 하나 생성한 후 다음과 같이 코드를 작성하고 실행합니다. 코드를 실행하면 그림 18.3과 같이 키움 번개 로그인 창이 출력되는 것을 확인할 수 있습니다.

```
from pywinauto import application
from pywinauto import timings
import time
import os

app = application.Application()
app.start("C:/KiwoomFlash3/bin/nkministarter.exe")

title = "번개3 Login"
dlg = timings.WaitUntilPasses(20, 0.5, lambda: app.window_(title=title))
```

이번에는 pywinauto 패키지를 이용해 마우스와 키보드 입력을 자동화하겠습니다. 앞서 작성한 코드에 다음 코드를 추가합니다.

```
pass_ctrl = dlg.Edit2
pass_ctrl.SetFocus()
pass_ctrl.TypeKeys('xxxx')

cert_ctrl = dlg.Edit3
cert_ctrl.SetFocus()
cert_ctrl.TypeKeys('yyyy')

btn_ctrl = dlg.Button0
btn_ctrl.Click()
```

앞서 키움 번개 로그인 대화상자를 dlg라는 변수로 바인딩했으므로 dlg로부터 비밀번호와 인증비밀번호를 입력하는 데 사용되는 컨트롤을 구합니다. 그리고 마우스 클릭 함수인 SetFocus와 키보드 입력 함수인 TypeKeys를 호출해 비밀번호와 인증비밀번호(공인 인증 비밀번호)를 입력합니다. 즉, 이어지는 코드에서 'xxxx' 부분을 지우고 여러분의 키움 번개3 로그인 비밀번호를 입력하고 'yyyy' 부분에 인증비밀번호를 입력하면 됩니다.

지금까지 작성한 코드를 실행하면 자동으로 로그인이 이뤄진 후 그림 18.5와 같이 키움 번개가 정상적으로 실행되는 것을 확인할 수 있습니다. 이 과정에서 업데이트가 있는 경우 업데이트 파일을 다운로드하기 때문에 자동으로 버전 처리가 완료되는 것입니다.



그림 18.5 키움 번개3

그림 18.5의 키움 번개3은 버전 처리를 위해 실행한 프로그램이고 우리는 실제 거래를 OpenAPI+를 통해 개발한 프로그램에서 처리할 것입니다. 따라서 키움 번개3은 필요 없으므로 바로 종료하겠습니다.

윈도우에서는 taskkill 명령을 이용해 특정 프로그램을 종료할 수 있습니다. 파이썬에서 윈도우 명령을 실행하려면 os 모듈의 system 함수를 사용하면 됩니다. 참고로 로그인 후에 업데이트를 수행하는 과정에 시간이 소요되기 때문에 time.sleep 함수를 호출해 약 50초 정도 대기하게 했습니다.

```
time.sleep(50)
os.system("taskkill /im nkmini.exe")
```

지금까지 작성한 전체 코드는 예제 18.1과 같습니다. 예제 18.1 코드를 실행하면 자동으로 버전 처리를 할 수 있습니다.

예제 18.1 키움 번개2 자동 로그인 및 종료 스크립트(소스코드: book/ch18/day01/autorun.py)

```
from pywinauto import application
from pywinauto import timings
import time
import os

app = application.Application()
app.start("C:/KiwoomFlash3/bin/nkministarter.exe")

title = "번개3 Login"
dlg = timings.WaitUntilPasses(20, 0.5, lambda: app.window_(title=title))

pass_ctrl = dlg.Edit2
pass_ctrl.SetFocus()
pass_ctrl.TypeKeys('xxxx')

cert_ctrl = dlg.Edit3
cert_ctrl.SetFocus()
cert_ctrl.TypeKeys('yyyy')

btn_ctrl = dlg.Button0
btn_ctrl.Click()

time.sleep(50)
os.system("taskkill /im nkmini.exe")
```

만약 PyCharm을 관리자 권한으로 실행하지 않은 상태에서 예제 18.1의 코드를 실행하면 그림 18.6과 같은 에러가 발생합니다. 앞서 여러 번 설명했지만 윈도우 7 이상의 운영 체제에서는 그림 18.6과 같은 에러 메시지가 발생하면 프로그램을 관리자 권한으로 실행하면 된다는 점을 꼭 알고 계시길 바랍니다.



그림 18.6 관리자 권한으로 실행하지 않은 경우 발생하는 에러

그림 18.3에 나온 윈도우 대화상자의 이름과 각 컨트롤의 이름을 알아내기 위해 SWAPY(https://github.com/pywinauto/SWAPY)라는 실행 프로그램을 사용했습니다. SWAPY는 깃허브를 통해 코드와 실행 파일을 제공하는데, 단순히 실행 파일을 다운로드한 후 실행하기만 하면 그림 18.6과 같이 현재 윈도우 상에 실행 중인 프로그램의 구조를 살펴볼 수 있습니다.

이 책을 쓸 때도 예제 18.1의 코드를 작성하기 위해 먼저 키움 번개3 프로그램을 실행한 상태에서 SWAPY를 실행해 윈도우 대화상자 및 각 컨트롤에 대한 정보를 얻은 후 이를 코드로 구현했습니다.

번개3은 종료할 때 종료확인창을 화면에 출력합니다. 아쉽게도 이 창은 예제 18.1의 코드로는 닫을 수 없고 번개3의 환경설정을 통해 해결해야 합니다. 그림 18.5 번개3 실행화면에서 '호' 메뉴 왼쪽에 위치하는 메뉴를 클릭합니다. 그다음 환경설정 메뉴를 선택한 후 '번개3 종료시 종료확인창 보기' 체크 박스의 체크를 해제하면 됩니다.

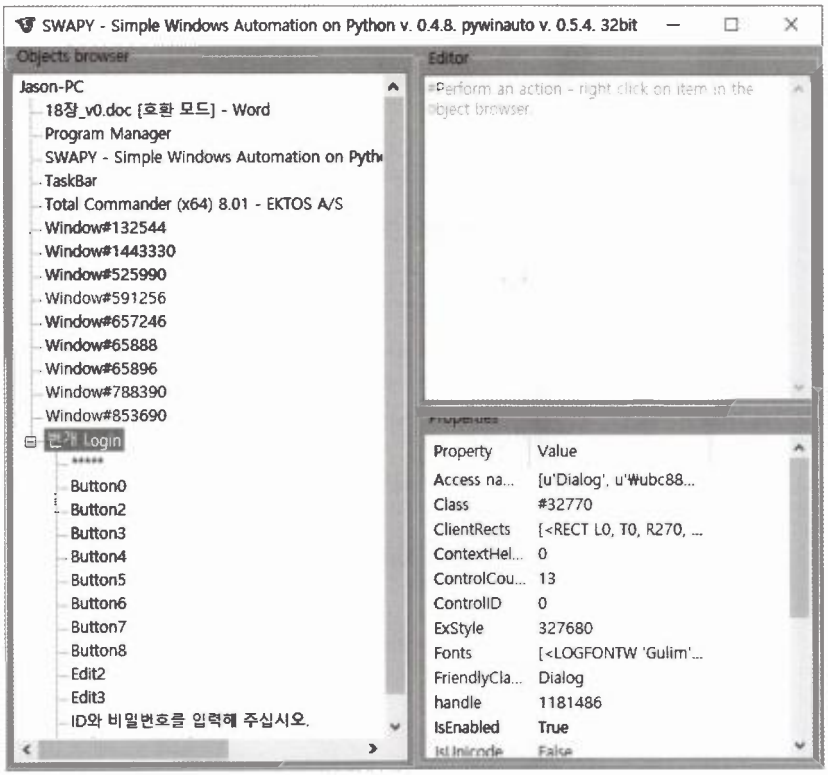


그림 18.7 SWAPY 프로그램

윈도우 작업 스케줄러

앞서 자동 버전 처리를 수행하는 파이썬 스크립트를 구현했습니다. 이번에는 윈도우의 작업 스케줄러를 이용해 정해진 시간에 파이썬 스크립트가 자동으로 실행되게 하는 방법을 알아보겠습니다.

윈도우 10에서 작업 스케줄러를 실행하려면 그림 18.8과 같이 화면의 왼쪽 아래에 위치하는 '웹 및 Windows 검색' 창에 작업 스케줄러라고 입력하면 됩니다.

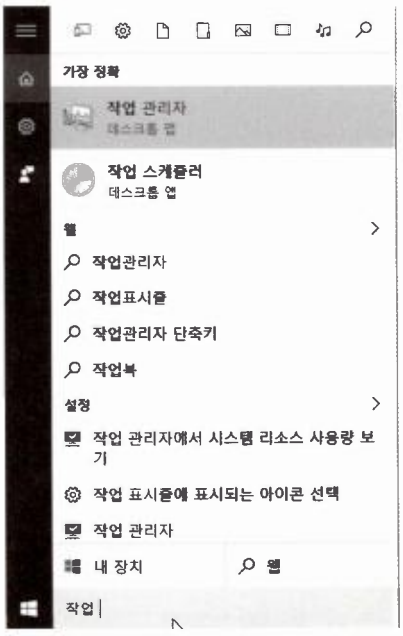


그림 18.8 작업 스케줄러 실행(윈도우 10)

작업 스케줄러가 실행되면 그림 18.9와 같이 [작업 만들기] 메뉴를 선택합니다.

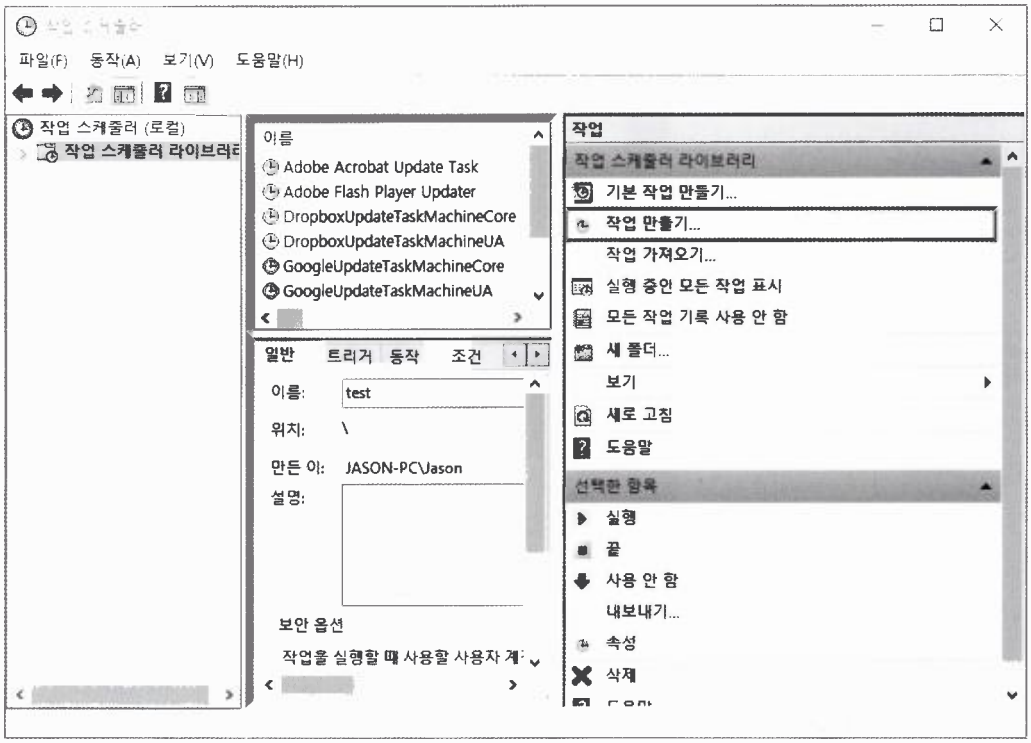


그림 18.9 새 작업 만들기

[새 작업 만들기] 창의 [일반] 탭에서는 작업 이름을 입력합니다. 여기서는 KiwoomAuto라고 입력했습니다. 또한 해당 스크립트를 실행할 때 관리자 권한으로 실행했던 것처럼 [가장 높은 수준의 권한으로 실행]을 체크합니다.

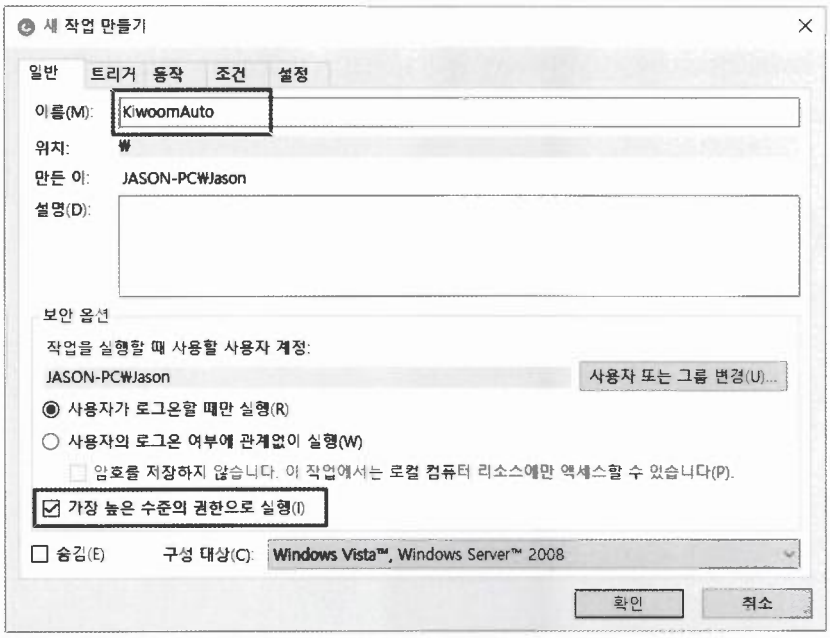


그림 18.10 [일반] 탭 설정

[새 작업 만들기] 창에서 [트리거] 탭을 선택합니다. 트리거란 '방아쇠'라는 뜻인데, 이곳에 앞에서 지정한 'KiwoomAuto'가 언제 시작되는지 설정합니다. 보통은 시간을 입력해서 정해진 시간에 작업이 수행되도록 설정합니다. 트리거를 입력하기 위해 그림 18.11과 같이 [새로 만들기] 버튼을 클릭합니다.

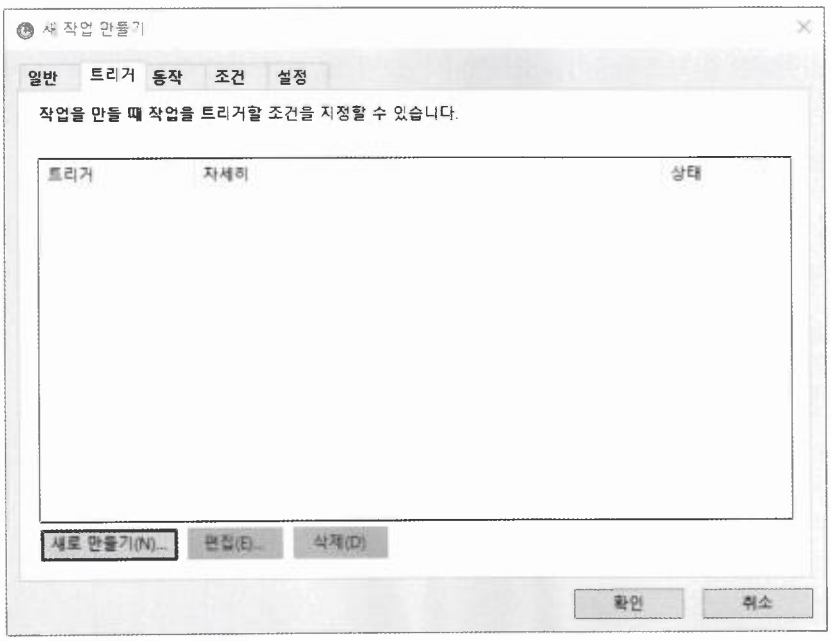


그림 18.11 트리거 설정(1)

그림 18.12와 같이 설정 항목에서 [한 번] 또는 [매일]을 선택합니다. 여기서는 테스트를 위해 [한 번]을 선택했지만 매일 반복되게 하려면 [매일]을 선택하면 됩니다. 시작 시간도 원하는 대로 선택합니다. [확인] 버튼을 눌러 트리거 생성을 끝냅니다.

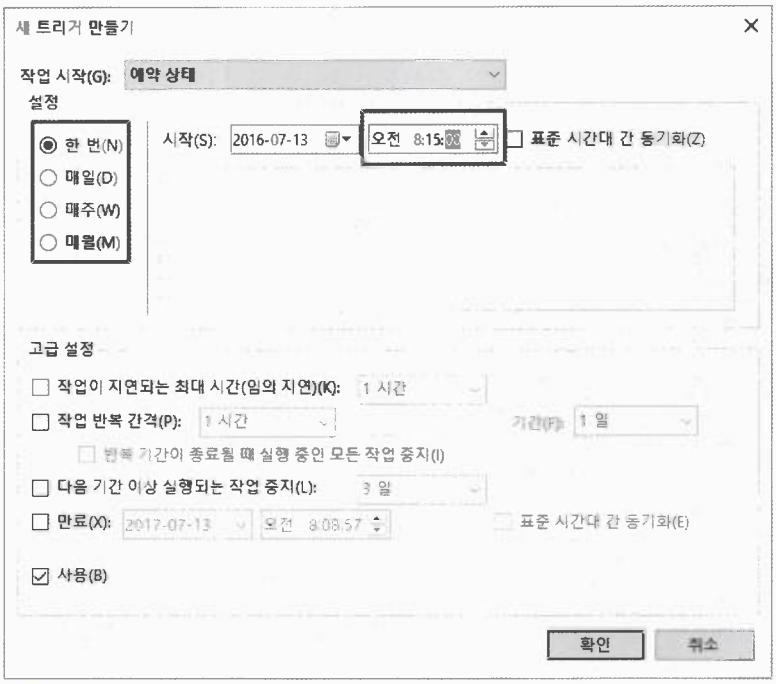


그림 18.12 트리거 설정(2)

이번에는 [동작] 탭으로 이동한 후 새로 만들기를 클릭합니다. 그림 18.13과 같이 먼저 [찾아보기] 버튼을 클릭해 아나콘다 설치 디렉터리의 pythonw.exe를 선택합니다. 참고로 python.exe 대신 pythonw.exe를 선택한 것은 스크립트 실행 시 콘솔 창이 나타나지 않게 하기 위해서입니다.

[인수 추가(옵션)] 항목에는 앞서 작성한 예제 18.1의 파이썬 스크립트의 위치를 지정하면 됩니다. 여기서는 스크립트가 현재 바탕 화면 경로에 test.py라는 이름으로 있으므로 다음과 같이 입력했습니다.

```
C:\Users\Jason\Desktop\test.py
```

[확인]을 눌러 새 동작 만들기를 종료합니다.

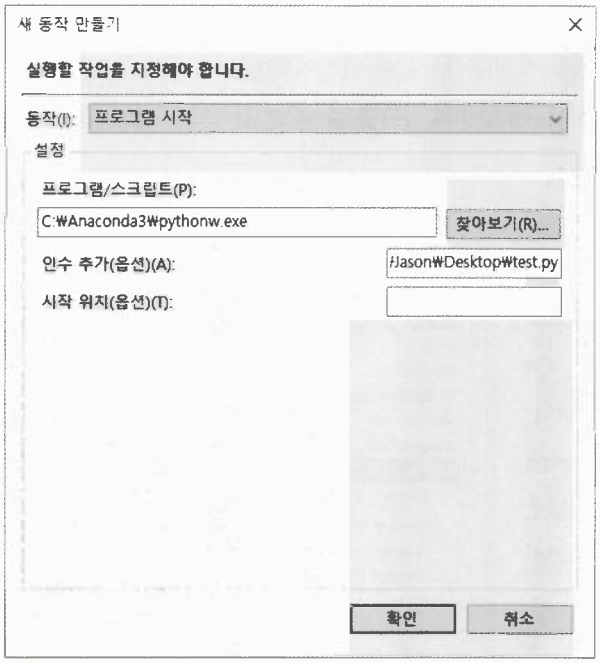


그림 18.13 새 동작 만들기

이번에는 [조건] 탭으로 이동합니다. 노트북의 경우 배터리로 잠시 테스트해볼 수 있는데 이 경우에는 정상적으로 작업이 실행되지 않을 수 있습니다. 그림 18.14와 같이 [컴퓨터의 AC 전원이 켜져 있는 경우에만 작업 시작]을 선택 해제합니다. [확인]을 눌러 조건 설정도 완료합니다.

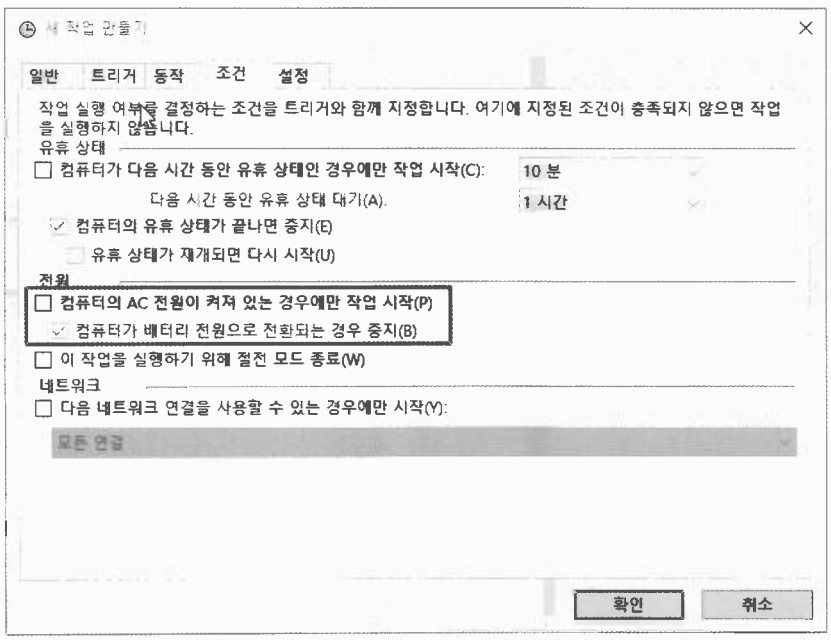


그림 18.14 조건 항목 설정

지금까지 설명한 과정을 잘 따라 했다면 설정한 시간에 키움 번개2 로그인 스크립트가 자동으로 실행되는 것을 확인할 수 있었을 것입니다. 경우에 따라 키움 번개2를 업데이트하는 데 시간이 오래 걸릴 수 있는데 이 경우 예제 18.1의 코드에서 time.sleep 함수의 인자를 좀 더 큰 값으로 지정해 충분히 업데이트될 수 있게 설정하면 됩니다.

PyTrader 구현

이번 절에서는 그림 18.1의 PyTrader 프로그램을 만들어 보겠습니다. 가장 먼저 할 일은 Qt Designer를 이용해 메인 윈도우를 만드는 것입니다. 아직은 윈도우 내에 위젯이 없으므로 Qt Designer에서 메인 윈도우만 만든 후 pytrader.ui라는 이름으로 저장합니다. 이 부분이 잘 기억나지 않는다면 잠깐 PyQt를 다룬 16장을 참고하기 바랍니다.

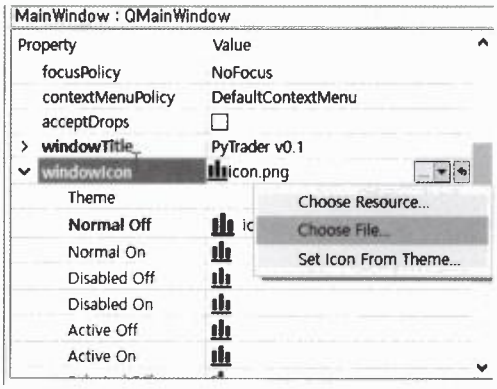


그림 18.15 Main Window의 타이틀 및 아이콘 설정

윈도우 타이틀을 변경하려면 그림 18.15와 같이 Qt Designer의 [Property Window]에서 [windowTitle] 항목을 변경하면 됩니다. 윈도우 아이콘을 변경하려면 [Property Window]에서 '…'라고 표시된 버튼을 클릭한 후 [Choose File] 메뉴를 선택해 아이콘 파일의 경로를 지정하면 됩니다.

프로그램에 사용할 아이콘은 무료 아이콘을 검색할 수 있는 MyIconFinder(http://www.myiconfinder.com/)나 Flaticon(http://www.flaticon.com/) 같은 사이트에서 'stock'이라는 키워드로 검색한 후 PNG 포맷의 48x48 크기의 파일을 내려받으면 됩니다. 저자는 PyTrader에 쓸 아이콘을 MyIconFinder에서 검색했고, 그림 18.16과 같은 PNG 파일을 내려받아 사용했습니다.

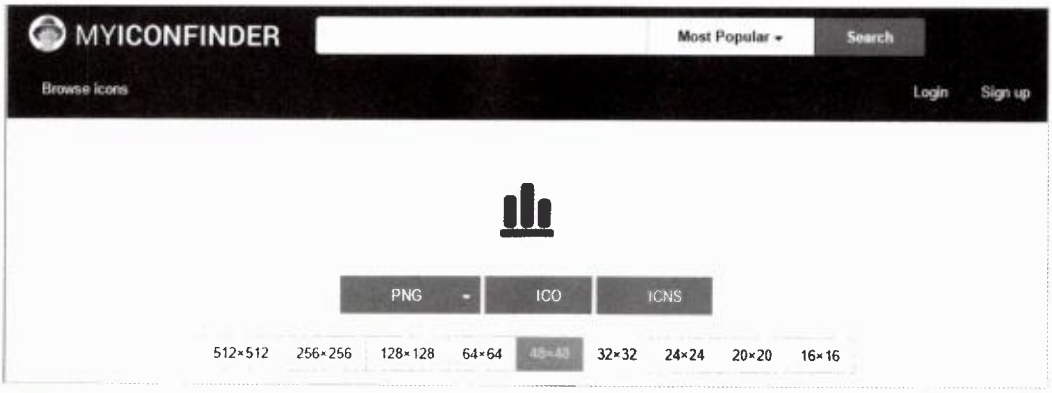


그림 18.16 아이콘 검색 및 다운로드

Qt Designer에서는 UI를 디자인한 후 Ctrl + R 키를 누르면 그림 18.17과 같이 현재까지 작업이 진행된 UI의 실행 화면을 확인할 수 있습니다. 매우 편리한 기능이니 단축키를 잘 기억해 두길 바랍니다.

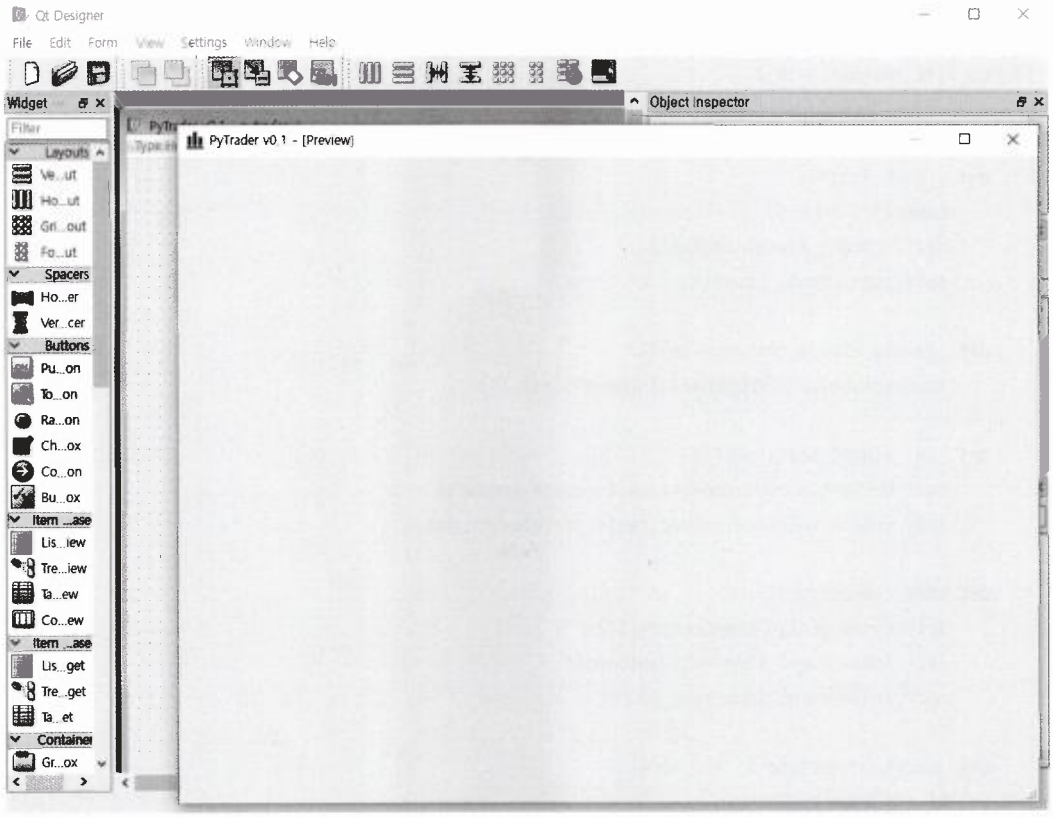


그림 18.17 Qt Designer를 이용한 UI 구성 및 실행

PyTrader에서 UI는 Qt Designer를 통해 생성한 pytrader.ui 파일을 불러와서 사용할 것입니다. 이와 유사하게 키움 OpenAPI+와 관련된 코드는 17장에서 구현한 Kiwoom 클래스를 사용하겠습니다.

예제 18.2는 Kiwoom.py의 소스코드입니다. 17장에서 구현한 Kiwoom 클래스에 get_connect_state 메서드를 추가로 구현했습니다.

예제 18.2 Kiwoom 클래스(소스코드: book/ch18/day01/Kiwoom.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QAxContainer import *
```

```
from PyQt5.QtCore import *
import time
import pandas as pd
import sqlite3

TR_REQ_TIME_INTERVAL = 0.2

class Kiwoom(QAxWidget):
    def __init__(self):
        super().__init__()
        self._create_kiwoom_instance()
        self._set_signal_slots()

    def _create_kiwoom_instance(self):
        self.setControl("KHOPENAPI.KHOpenAPICtrl.1")

    def _set_signal_slots(self):
        self.OnEventConnect.connect(self._event_connect)
        self.OnReceiveTrData.connect(self._receive_tr_data)

    def comm_connect(self):
        self.dynamicCall("CommConnect()")
        self.login_event_loop = QEventLoop()
        self.login_event_loop.exec_()

    def _event_connect(self, err_code):
        if err_code == 0:
            print("connected")
        else:
            print("disconnected")

        self.login_event_loop.exit()

    def get_code_list_by_market(self, market):
        code_list = self.dynamicCall("GetCodeListByMarket(QString)", market)
        code_list = code_list.split(';')
        return code_list[:-1]

    def get_master_code_name(self, code):
        code_name = self.dynamicCall("GetMasterCodeName(QString)", code)
        return code_name
```

```
    def get_connect_state(self):
        ret = self.dynamicCall("GetConnectState()")
        return ret

    def set_input_value(self, id, value):
        self.dynamicCall("SetInputValue(QString, QString)", id, value)

    def comm_rq_data(self, rqname, trcode, next, screen_no):
        self.dynamicCall("CommRqData(QString, QString, int, QString)", rqname, trcode, next,
screen_no)
        self.tr_event_loop = QEventLoop()
        self.tr_event_loop.exec_()

    def _comm_get_data(self, code, real_type, field_name, index, item_name):
        ret = self.dynamicCall("CommGetData(QString, QString, QString, int, QString)", code,
                               real_type, field_name, index, item_name)
        return ret.strip()

    def _get_repeat_cnt(self, trcode, rqname):
        ret = self.dynamicCall("GetRepeatCnt(QString, QString)", trcode, rqname)
        return ret

    def _receive_tr_data(self, screen_no, rqname, trcode, record_name, next, unused1, unused2,
unused3, unused4):
        if next == '2':
            self.remained_data = True
        else:
            self.remained_data = False

        if rqname == "opt10081_req":
            self._opt10081(rqname, trcode)

        try:
            self.tr_event_loop.exit()
        except AttributeError:
            pass

    def _opt10081(self, rqname, trcode):
        data_cnt = self._get_repeat_cnt(trcode, rqname)

        for i in range(data_cnt):
            date = self._comm_get_data(trcode, "", rqname, i, "일자")
```

```
        open = self._comm_get_data(trcode, "", rqname, i, "시가")
        high = self._comm_get_data(trcode, "", rqname, i, "고가")
        low = self._comm_get_data(trcode, "", rqname, i, "저가")
        close = self._comm_get_data(trcode, "", rqname, i, "현재가")
        volume = self._comm_get_data(trcode, "", rqname, i, "거래량")

        self.ohlcv['date'].append(date)
        self.ohlcv['open'].append(int(open))
        self.ohlcv['high'].append(int(high))
        self.ohlcv['low'].append(int(low))
        self.ohlcv['close'].append(int(close))
        self.ohlcv['volume'].append(int(volume))
```

pytrader.ui 파일과 Kiwoom.py 파일이 준비되면 pytrader.py 파일에 모듈을 임포트하고 ui 파일을 불러오는 코드를 구현합니다.

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5 import uic
from Kiwoom import *

form_class = uic.loadUiType("pytrader.ui")[0]

class MyWindow(QMainWindow, form_class):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

지금까지 작성한 pytrader.py 파일을 실행하면 그림 18.18과 같이 PyTrader 윈도우 창이 화면에 나타납니다. 아직 OpenAPI+ 로그인, 연결 상태 확인, 현재 시간 출력 기능은 추가하지 않았기 때문에 단순히 윈도우 창만이 화면에 나타날 것입니다.

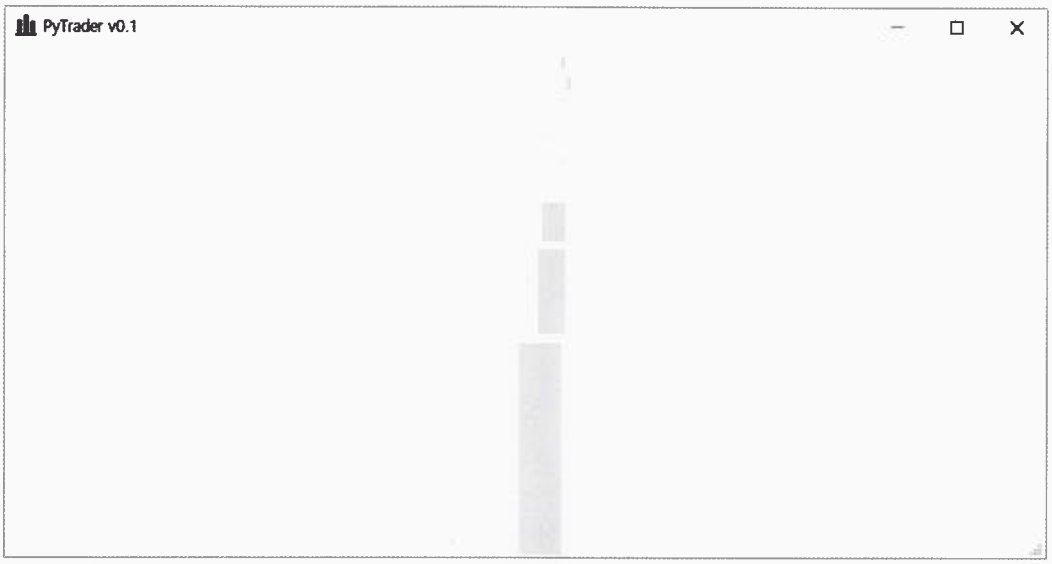


그림 18.18 PyTrader 실행 화면

PyTrader 프로그램이 실행될 때 키움 로그인이 진행되도록 생성자에서 키움 객체를 생성한 후 comm_connect 메서드를 호출합니다.

```
class MyWindow(QMainWindow, form_class):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

        self.kiwoom = Kiwoom()
        self.kiwoom.comm_connect()
```

MyWindow 클래스에서 Kiwoom 클래스를 사용하려면 먼저 Kiwoom 모듈(Kiwoom.py)을 임포트해야 합니다. 이때 Kiwoom.py 파일이 pytrader.py 파일과 동일한 디렉터리에 위치해야 합니다.

```
from Kiwoom import *
```

생성자에 코드를 추가했다면 다시 pytrader.py 파일을 실행해 봅시다. 이번에는 그림 18.19와 같이 키움 Open API+ 로그인 창이 나타나는 것을 확인할 수 있습니다. 고객 ID, 비밀번호를 입력하고 로그인 버튼을 누르면 정상적으로 로그인이 수행되고 그림 18.18과 같이 PyTrader 윈도우가 다시 표시됩니다.

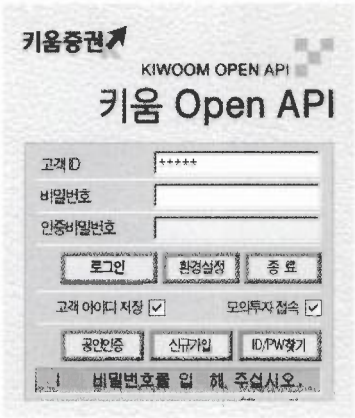


그림 18.19 키움 Open API+ 로그인 창

이번에는 StatusBar 위젯에 서버 연결 상태 및 현재 시간을 출력하는 기능을 구현해 보겠습니다. 서버 연결 상태를 확인하려면 Kiwoom 클래스에 추가한 get_connect_state 메서드를 사용하면 됩니다.

현재 시간을 출력하는 기능은 일정한 단위로 현재 시간을 얻어온 후 이를 StatusBar에 출력하면 됩니다. 이를 위해서는 주기적으로 이벤트를 발생시키는 Timer가 필요합니다. 예를 들어, Timer가 1초에 한 번 이벤트(시그널)를 발생시키면 이 이벤트를 처리하는 메서드(슬롯)에서 현재 시간을 얻어온 후 이를 StatusBar에 출력하면 됩니다.

Qt의 QTimer 클래스를 사용하면 정해진 시간마다 이벤트를 발생시킬 수 있습니다. pytrader.py 파일에 들어있는 MyWindow 클래스의 생성자에서 QTimer 객체를 생성하고 1초에 한 번씩 이벤트가 발생하게 하는 코드를 추가합니다.

```
self.timer = QTimer(self)
self.timer.start(1000)
self.timer.timeout.connect(self.timeout)
```

QTimer 클래스는 start 메서드를 제공하는데 이 메서드에 인자로 1000을 지정하면 1초에 한 번씩 주기적으로 timeout 시그널이 발생합니다. 따라서 timeout 시그널이 발생할 때 이를 처리할 슬롯으로 self.timeout을 설정하면 됩니다.

시그널이 1초마다 발생하게 했으니 이번에는 시그널을 처리할 슬롯을 구현하면 됩니다. timeout 메서드에서는 먼저 현재 시간을 구하고 이를 '시간:분:초' 형태의 문자열로 변환해 둡니다. 그리고 Kiwoom 클래스의 get_connect_state 메서드를 호출해서 서버 연결 상태를 확인한 후 StatusBar에 출력할 문자열을 준비합니다. 마지막으로 현재 시간과 상태 출력 메시지를 StatusBar 위젯에 출력하면 됩니다.

```
def timeout(self):
    current_time = QTime.currentTime()
    text_time = current_time.toString("hh:mm:ss")
    time_msg = "현재시간: " + text_time

    state = self.kiwoom.GetConnectState()
    if state == 1:
        state_msg = "서버 연결 중"
    else:
        state_msg = "서버 미 연결 중"

    self.statusbar.showMessage(state_msg + " | " + time_msg)
```

지금까지 구현한 전체 코드는 예제 18.3과 같습니다. UI 부분은 pytrader.ui 파일로 분리돼 있고 키움 OpenAPI+ 처리에 대한 코드는 Kiwoom 모듈을 사용하기 때문에 코드가 생각보다 많지 않음을 확인할 수 있습니다.

예제 18.3 pytrader.py 파일(소스코드: book/ch18/day01/pytrader.py)

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5 import uic
from Kiwoom import *

form_class = uic.loadUiType("pytrader.ui")[0]

class MyWindow(QMainWindow, form_class):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

        self.kiwoom = Kiwoom()
        self.kiwoom.comm_connect()

        self.timer = QTimer(self)
        self.timer.start(1000)
        self.timer.timeout.connect(self.timeout)

    def timeout(self):
        current_time = QTime.currentTime()
```

```
        text_time = current_time.toString("hh:mm:ss")
        time_msg = "현재시간: " + text_time

        state = self.kiwoom.get_connect_state()
        if state == 1:
            state_msg = "서버 연결 중"
        else:
            state_msg = "서버 미 연결 중"

        self.statusbar.showMessage(state_msg + " | " + time_msg)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    myWindow = MyWindow()
    myWindow.show()
    app.exec_()
```

예제 18.3을 실행하면 그림 18.1과 같이 윈도우 창이 나타나고 현재 시간과 연결 상태가 출력되는 것을 확인할 수 있습니다.

키움 OpenAPI+ 자동 로그인

개발 1일 차에 구현한 pytrader.py 파일을 실행하면 그림 18.19와 같은 키움 Open API 로그인 창이 나타납니다. 윈도우 7 이상에서는 로그인 창이 바로 실행될 때 사용자 계정 컨트롤과 관련된 메시지 박스가 나타나는 경우가 있습니다. 이러한 메시지 박스는 프로그램의 자동 실행을 어렵게 만들기 때문에 사용자 계정 컨트롤 기능을 비활성화하는 것이 좋습니다.

화면에 출력된 사용자 계정 컨트롤 메시지 박스에서 '알림이 표시될 때 변경'이라는 곳을 클릭하면 그림 18.20과 같은 화면이 나타납니다. 사용자 계정 컨트롤 기능을 비활성화하려면 그림 18.20에서 '알리지 않음' 단계를 선택한 후 [확인]을 클릭합니다.

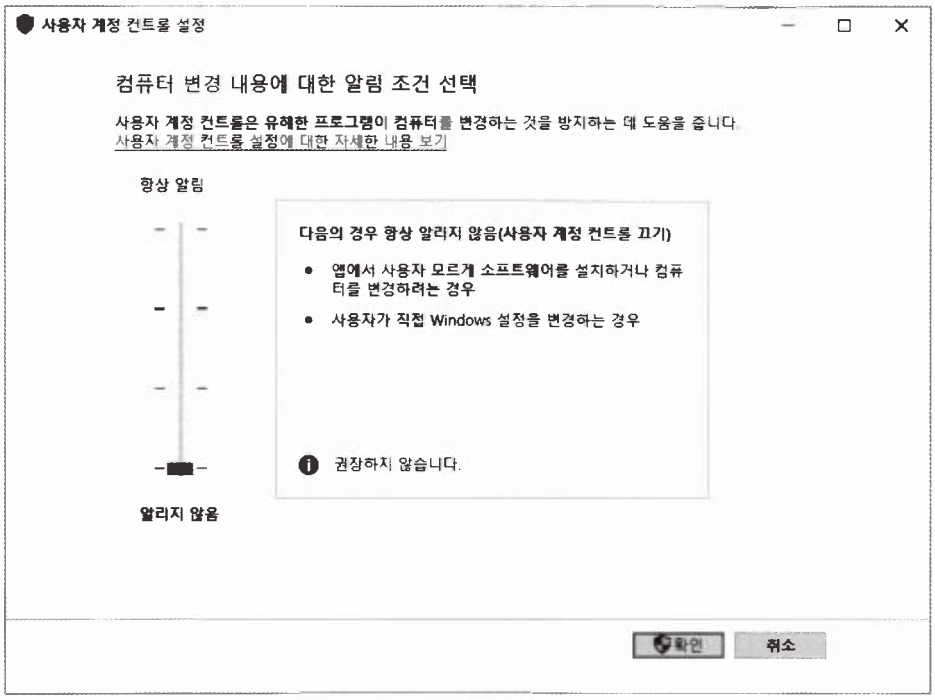


그림 18.20 사용자 계정 컨트롤 설정

사용자 계정 컨트롤 설정을 변경한 후 다시 pytrader.py 파일을 실행해 보기 바랍니다. 이번에는 사용자 계정 컨트롤과 관련된 메시지 박스가 출력되지 않고 바로 키움 OpenAPI+ 로그인 창이 화면에 출력될 것입니다.

이번에는 키움증권 로그인 과정을 자동화하겠습니다. 키움증권은 사용자의 편의를 위해 자동 로그인 기능을 지원합니다. 자동 로그인 설정을 위해 pytrader.py 파일을 실행해 로그인을 진행합니다. 정상적으로 로그인되면 그림 18.21과 같이 윈도우 오른쪽 아래의 아이콘 중 키움증권 아이콘에서 마우스 오른쪽 버튼을 클릭한 후 '계좌비밀번호 저장'을 선택합니다.



그림 18.21 키움 Open API 자동 로그인 설정

자동 로그인 설정은 계좌 단위로 가능한데, 계좌를 선택한 후 등록 부분에 계좌 비밀번호를 입력하고 [등록] 버튼을 눌러 비밀번호를 등록합니다. 비밀번호가 잘 등록되면 [AUTO]라는 체크박스를 체크합니다.

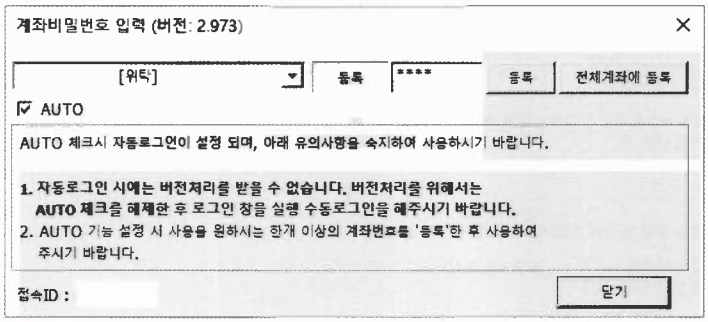


그림 18.22 자동 로그인 설정

설정을 모두 완료했다면 앞서 실행한 PyTrader 프로그램을 종료합니다. 그리고 다시 pytrader.py 파일을 실행해 보면 이번에는 로그인 창이 화면에 나타나지 않고 바로 로그인이 수행된 후 PyTrader 프로그램이 실행되는 것을 확인할 수 있습니다.

지금까지 배운 내용을 정리해 보겠습니다. 윈도우 작업 스케줄러를 이용해 매일 08:00에 키움 번개 자동 로그인 스크립트(test.py)를 실행합니다. 해당 스크립트가 실행되면 키움 번개에 로그인되면서 업데이트가 진행되기 때문에 버전 처리가 완료됩니다. 그리고 키움 번개 HTS는 자동으로 종료됩니다.

08:15 경에는 윈도우 작업 스케줄러를 이용해 pytrader.py 파일을 실행합니다. 이때 이미 그림 18.22와 같이 자동 로그인을 설정해둬야 합니다. 그림 18.23과 같은 순서대로 프로그램이 실행되게 해두면 버전 처리 및 로그인을 자동화할 수 있습니다.

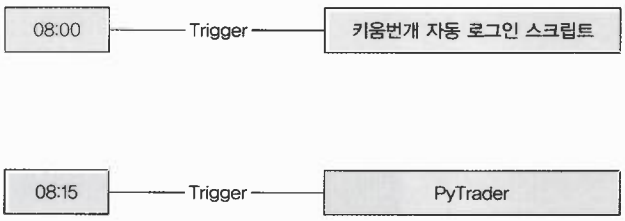


그림 18.23 버전 처리 및 자동 로그인 실행 시나리오

PyTrader 개발과 관련해서 키움 OpenAPI+와 관련된 코드는 모두 Kiwoom.py 파일에 구현했고 UI는 Qt Designer를 사용해 pytrader.ui 파일에 구현했습니다. 그 덕분에 pytrader.py 파일에는 이벤트를 처리하는 코드만 있어서 코드가 짧고 가독성이 좋다는 것을 확인했습니다.

여기까지가 개발 1일 차 내용입니다. 처음이라 조금 어려웠을 수도 있으니 직접 따라 하면서 코드도 입력해보고 윈도우 작업 스케줄러도 사용해보기 바랍니다.

02 개발 2일 차

개발 2일 차에서는 1일 차에서 개발한 프로그램에 주문 기능을 추가하겠습니다. 앞서 2부 증권사 API 편에서도 기초적인 API만 다뤘을 뿐 주문 기능은 다루지 않았습니다. 사실 주문 기능 API도 기본적으로는 다른 API와 사용법이 유사합니다. 다만 주문 API는 장중에만 동작하기 때문에 직장인이 이를 테스트하는 것이 현실적으로는 쉽지 않아 2부에서는 이를 다루지 않았던 것입니다.

먼저 2일 차의 최종 결과물 UI를 살펴봅시다. 그림 18.24를 보면 UI에 '수동주문'이라는 부분이 추가된 것을 확인할 수 있습니다. 여러분이 개발하는 프로그램의 최종 결과물에서는 프로그램이 자동으로 매수/매도를 진행할 테지만 현시점에서는 주문 관련 API를 테스트해보기 위해 수동주문과 관련된 UI를 추가했습니다.

PyTrader v0.2의 기능 부분을 살펴봅시다. PyTrader v0.2는 로그인이 수행되면 계좌를 조회해서 이를 QComboBox 위젯에 출력합니다. QComboBox 위젯을 사용하기 때문에 하나 이상의 계좌가 조회된 경우에도 효과적으로 이를 출력할 수 있습니다.

수동주문은 사용자가 입력한 종목코드를 바탕으로 주문 종류, 주문 방식, 수량, 가격에 따라 주문을 수행합니다. 그림 18.24를 보면 사용자는 주문과 관련된 데이터를 위젯을 통해 입력할 수 있음을 확인할 수 있습니다. 주문에 대한 데이터를 입력한 후 사용자가 [현금주문] 버튼을 클릭하면 주문 요청이 키움증권 서버로 전송됩니다.

PyTrader v0.2에는 사용자가 종목코드를 입력하면 종목코드로부터 종목명을 가져오고 이를 위젯에 출력하는 기능이 구현돼 있습니다. 이러한 기능은 사용자가 종목명을 일일이 조회하지 않고도 종목코드에 해당하는 종목명을 확인할 수 있게 해줍니다.

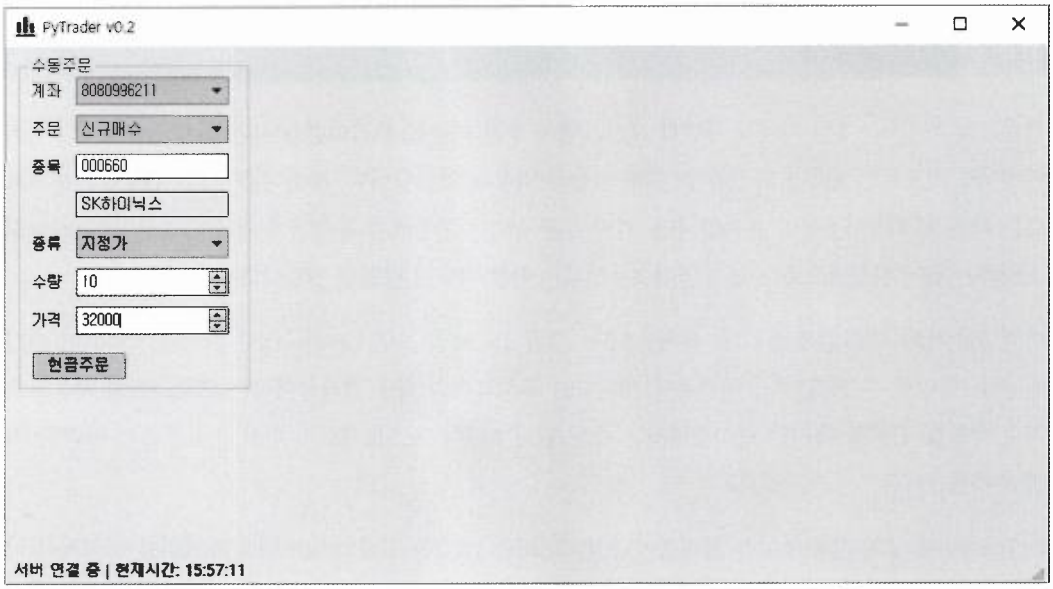


그림 18.24 PyTrader v0.2

UI 구성

먼저 UI 구성부터 업데이트하겠습니다. Qt Designer를 실행한 후 1일 차에 작성했던 pytrader.ui 파일을 엽니다. 윈도우 타이틀부터 PyTrader v0.2로 변경합니다. 참고로 매일 0.1씩 버전을 올리겠습니다. 윈도우 타이틀은 그림 18.25와 같이 Property Editor의 windowTitle 항목에서 변경하면 됩니다.

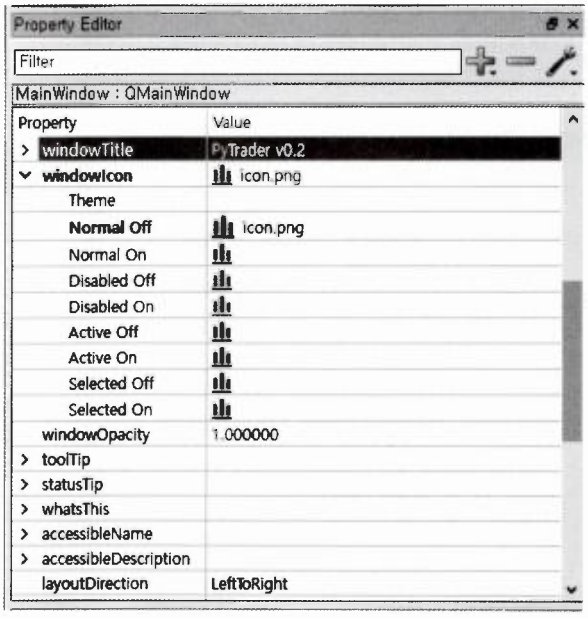


그림 18.25 Property Editor에서 윈도우 타이틀 변경

이번에는 수동주문과 관련된 위젯을 배치해 봅시다. 먼저 QGroupBox 위젯을 MainWindow 안으로 배치합니다. 이를 위해서는 그림 18.26과 같이 Qt Designer 왼쪽에 있는 Widget Box에서 Containers 항목의 Group Box를 선택한 후 이를 적당한 위치로 드래그 앤드 드롭하면 됩니다. Group Box의 이름도 '수동주문'으로 변경합니다.



그림 18.26 Group Box 위젯

이번에는 Qt Designer의 Widget Box에서 Display Widgets 항목에 있는 Label을 선택합니다. 참고로 여기서 Label은 PyQt의 QLabel 클래스에 해당합니다.



그림 18.27 QLabel 위젯

Label을 Main Window의 Group Box 안으로 드래그 앤드 드롭해서 그림 18.28과 같이 UI를 구성합니다. 각 위젯의 위치는 적당히 조절하면 됩니다.

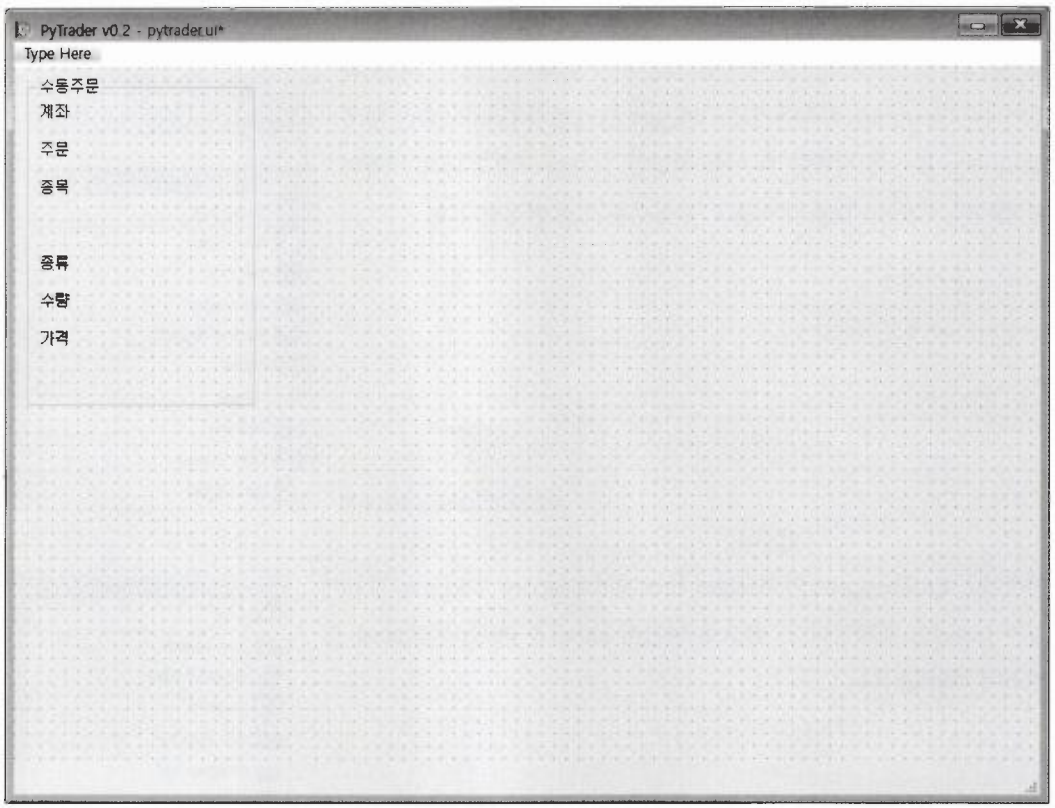


그림 18.28 Label 배치

이번에는 그림 18.29와 같이 Input Widgets 항목에서 Combo Box를 선택하기 바랍니다. Combo Box는 QComboBox 클래스를 의미합니다.



그림 18.29 Combo Box

Combo Box를 선택한 후 드래그 앤드 드롭해서 그림 18.30과 같이 세 개의 Combo Box를 Main Window에 배치합니다. 16장의 QComboBox에서 배웠듯이 QComboBox를 이용하면 여러 옵션이 있는 경우 이를 효과적으로 표현할 수 있습니다.

예를 들어, 계좌 번호가 여러 개 존재하는 경우 이를 효과적으로 표현하기 위해 QComboBox를 사용할 수 있습니다. 그림 18.30을 보면 계좌, 주문, 종류 항목은 각자 세부적으로 여러 가지 옵션이 존재하기 때문에 QComboBox를 사용했습니다.

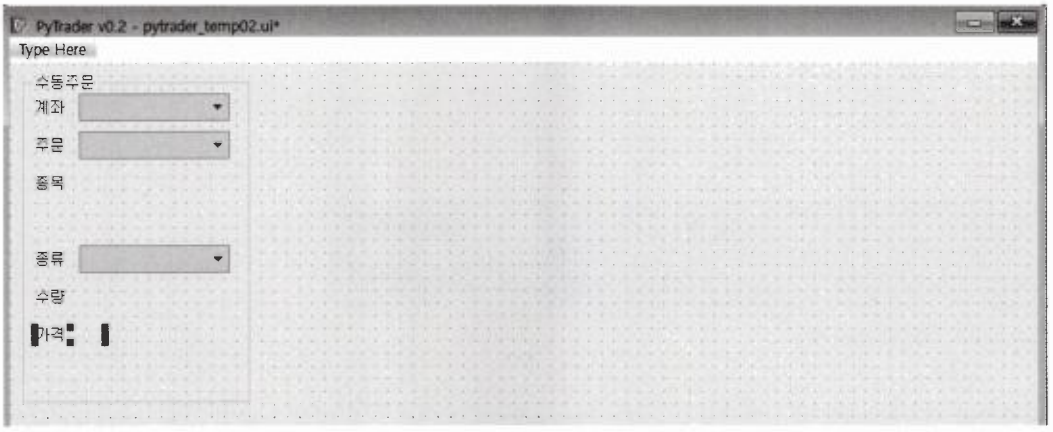


그림 18.30 QComboBox 배치

주문에 대한 Combo Box 위젯을 더블클릭하면 그림 18.31과 같이 세부 항목을 추가할 수 있는 창이 나타납니다. 왼쪽 아래의 + 기호를 클릭해 '신규매수', '신규매도', '매수취소', '매도취소'의 네 가지 세부 항목을 입력합니다. 그런 다음 [OK] 버튼을 클릭해 'Edit Combobox' 창을 닫습니다.

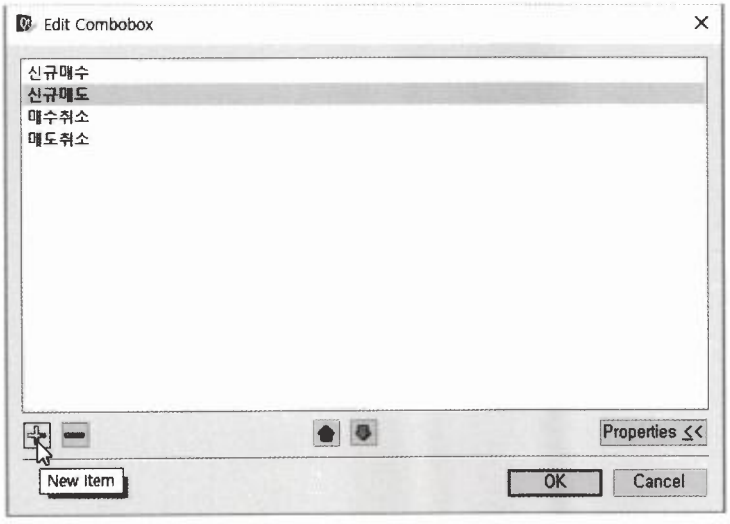


그림 18.31 주문 세부 항목 추가

마찬가지로 종류에 대한 Combo Box 위젯을 더블클릭한 후 Edit Combobox라는 창에서 '지정가'와 '시장가'를 추가합니다.

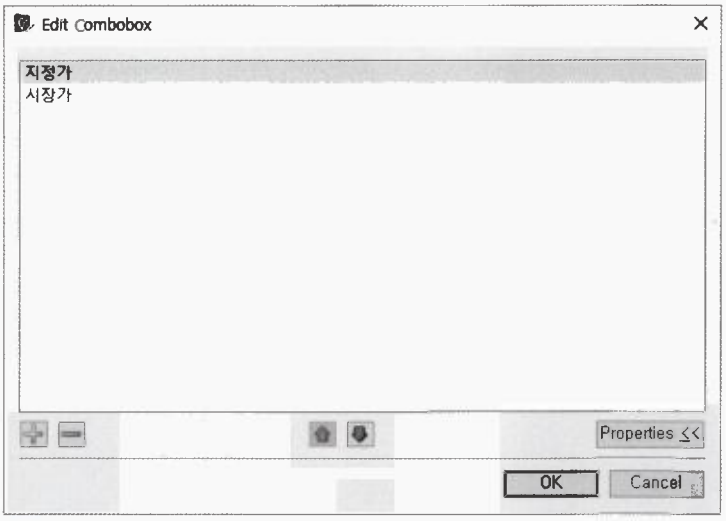


그림 18.32 종류 세부 항목 추가

지금까지 구현한 UI를 테스트해봅시다. Qt Designer에서 Ctrl + R 키를 누르면 그림 18.33과 같이 UI가 실행됩니다. 아직 시그널과 슬롯을 지정하지 않아 이벤트는 동작하지 않지만 간단히 UI의 결과물을 확인할 수 있습니다. Combo Box 위젯을 클릭하면 앞서 입력한 세부 항목을 선택할 수 있음을 확인할 수 있습니다.

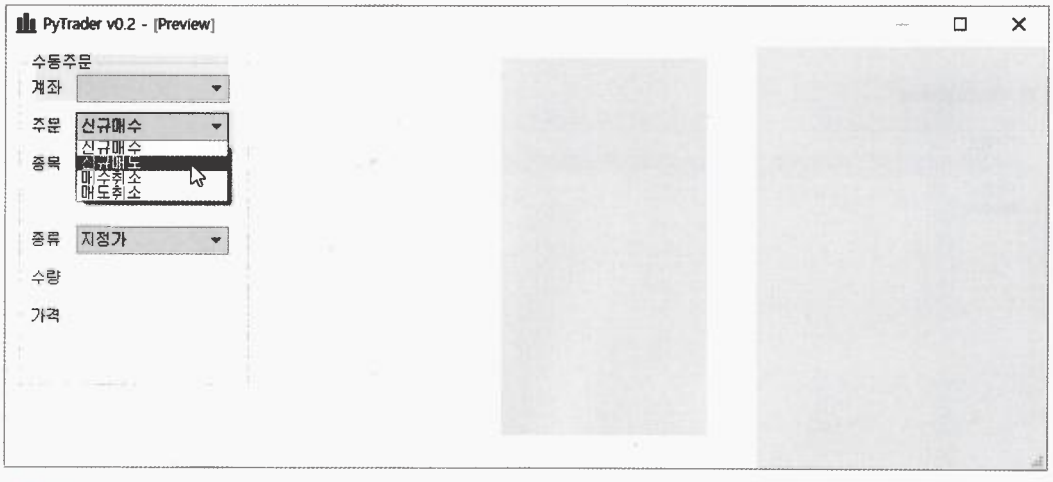


그림 18.33 수동주문 관련 UI 테스트

이번에는 사용자로부터 종목코드를 입력받기 위한 위젯을 배치해 보겠습니다. Qt Designer의 Widget Box에서 Input Widgets 항목에 있는 Line Edit를 선택한 후 그림 18.34와 같이 배치합니다. 참고로 Line Edit는 `QLineEdit` 클래스에 해당합니다.

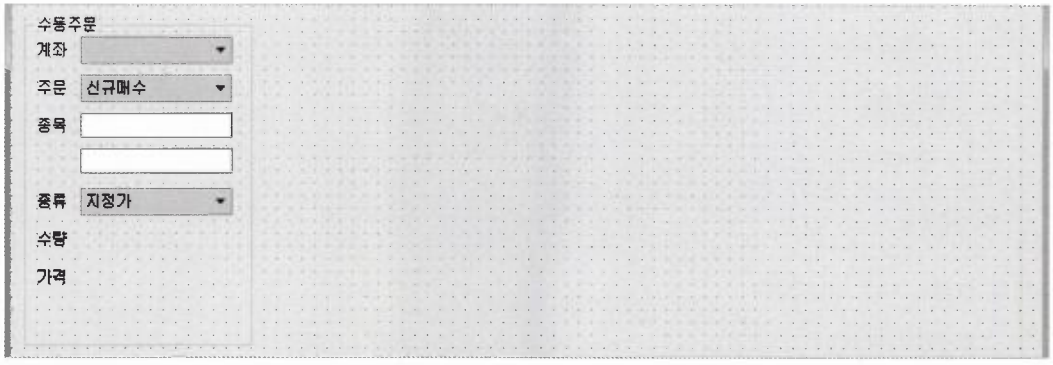


그림 18.34 Line Edit 위젯 추가

종목에서 두 번째 Line Edit 위젯은 사용자로부터 입력받은 종목코드를 바탕으로 종목명을 조회한 후 출력하는 역할을 합니다. 사용자가 알아보기 쉽게 위젯의 배경 색상을 변경해 봅시다.

먼저 해당 위젯을 선택한 후 Property Editor의 styleSheet 항목에서 [···] 버튼을 클릭합니다. 그림 18.35와 같이 Edit Style Sheet 창이 나타나면 Add Color에서 아래쪽 방향의 세모를 클릭해 `background-color` 항목을 선택한 후 원하는 색상을 선택하면 됩니다.

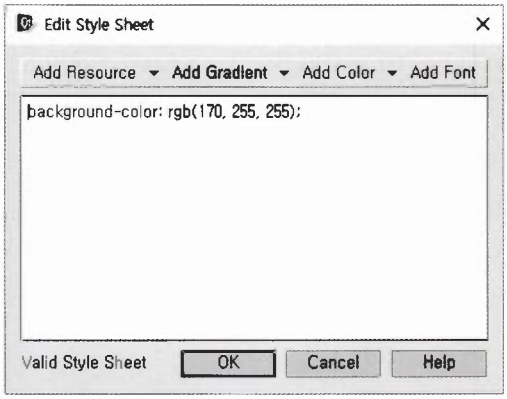


그림 18.35 Edit Style Sheet

이번에는 Input Widgets로부터 Spin Box를 선택한 후 수량과 가격 옆으로 배치해 봅시다. 그리고 Buttons 항목에서 Push Button을 선택한 후 맨 아래쪽에 배치하고 이름도 '현금주문'으로 변경합니다. 버튼에 표시될 텍스트는 버튼을 더블클릭해서 설정할 수 있습니다.

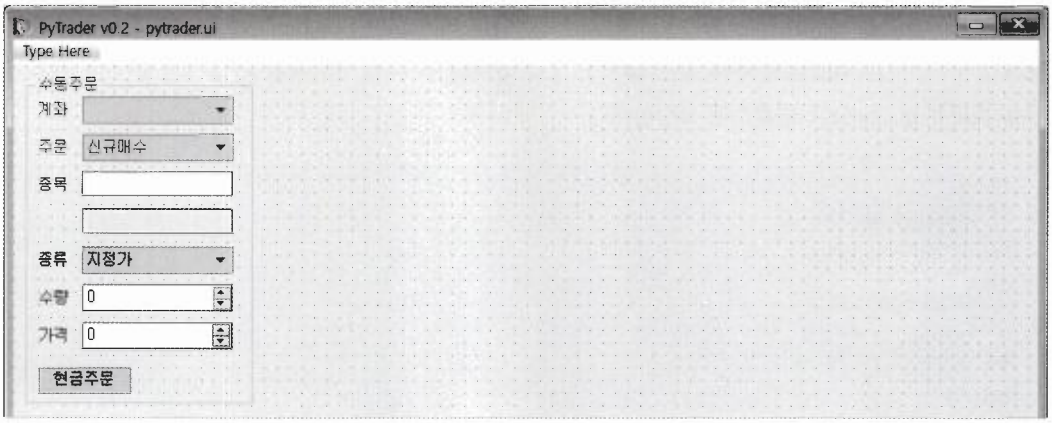


그림 18.36 Spin Box와 Push Button

Spin Box는 QSpinBox에 해당하는데, 먼저 Spin Box 위젯을 선택한 후 Property Editor를 통해 세부 옵션을 변경할 수 있습니다. QSpinBox에서는 위젯을 통해 입력 가능한 최솟값과 최댓값을 지정할 수 있으며, 그림 18.37과 같이 수량과 가격 부분에 적당한 값을 입력하면 됩니다. 만약 수량에 100을 입력해 둔다면 해당 QSpinBox 위젯으로는 100을 초과하는 숫자를 입력받을 수 없게 됩니다. 따라서 되도록 수량과 가격에 가능한 한 큰 값을 지정합니다.

singleStep 옵션에서는 QSpinBox에서 업/다운 버튼을 클릭할 때 한 번에 증가하는 값의 범위를 지정할 수 있습니다. 이 값 역시 적당한 값을 입력하면 되며, 여기서는 50을 입력했습니다. 즉, 가격을 입력하는 Spin Box는 한 번에 50원씩 증가하거나 감소하게 됩니다.

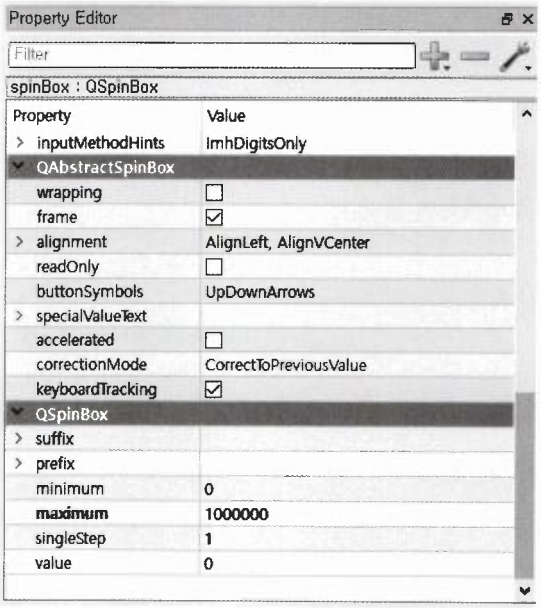


그림 18.37 QSpinBox 세부 옵션

Kiwoom.py 파일 업데이트

이번에는 키움 OpenAPI+ 처리를 담당하는 Kiwoom.py 파일을 업데이트해 보겠습니다. 먼저 주문과 관련된 API를 처리하는 메서드를 구현해 봅시다. 개발 가이드를 참조하면 그림 18.38과 같이 SendOrder 메서드를 사용하면 주식 주문에 대한 정보를 서버로 전송할 수 있습니다. 다만, 증권사 서버에 주문 요청을 했다고 해서 즉시 체결되는 것이 아니므로 이벤트 루프를 사용해 대기하고 있어야 합니다.

주문이 체결되면 증권사 서버는 그림 18.38과 같이 OnReceiveChejanData라는 이벤트를 발생시킵니다. 앞서 OnReceiveTrData 메서드 내에서 CommGetData 메서드를 호출해 데이터를 얻어왔던 것과 동일하게 체결과 관련해서는 OnReceiveChejanData라는 메서드 내에서 GetChejanData라는 메서드를 호출해서 체결잔고 데이터를 얻어오면 됩니다.

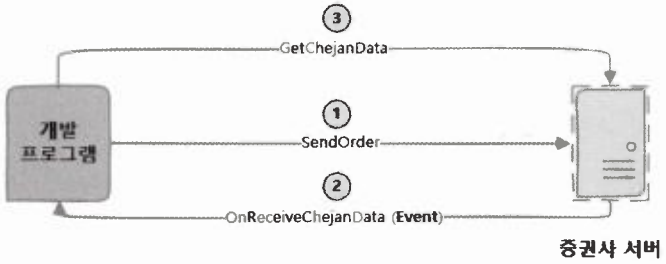


그림 18.38 주문 관련 API 호출 순서

먼저 SendOrder 메서드를 self.dynamicCall을 통해 호출하는 send_order 메서드를 Kiwoom 클래스에 구현해봅시다. 그림 18.39를 참조하면 SendOrder 메서드는 총 9개의 인자를 받는다는 것을 확인할 수 있습니다.

5) SendOrder

원형	LONG SendOrder(BSTR sRQName, BSTR sScreenNo, BSTR sAccNo, LONG nOrderType, BSTR sCode, LONG nQty, LONG nPrice, BSTR sHogaGb, BSTR sOrgOrderNo)
설명	주식 주문을 서버로 전송한다.

그림 18.39 SendOrder 메서드 원형 및 설명(출처: 키움 OpenAPI+ 개발가이드)

Kiwoom 클래스에 send_order 메서드를 추가합니다.

```
def send_order(self, rqname, screen_no, acc_no, order_type, code, quantity, price, hoga, order_
no):
    self.dynamicCall("SendOrder(QString, QString, QString, int, QString, int, int, QString,
QString)",
                     [rqname, screen_no, acc_no, order_type, code, quantity, price, hoga, order_
no])
```

체결잔고 데이터를 가져오는 메서드인 GetChejanData를 사용하는 get_chejan_data 메서드를 Kiwoom 클래스에 추가합니다.

```
def get_chejan_data(self, fid):
    ret = self.dynamicCall("GetChejanData(int)", fid)
    return ret
```

주문체결 시점에서 키움증권 서버가 발생시키는 OnReceiveChejanData 이벤트를 처리하는 메서드를 구현해 봅시다. 먼저 _set_signal_slots 메서드에 시그널과 슬롯을 연결하는 코드를 추가합니다.

```
self.OnReceiveChejanData.connect(self._receive_chejan_data)
```

OnReceiveChejanData 이벤트가 발생할 때 호출되는 _receive_chejan_data는 다음과 같이 구현합니다. 현시점에서는 주문체결 데이터를 출력할 UI가 존재하지 않으므로 단순히 print 함수를 사용해 화면에 결과를 출력해 보겠습니다.

```
def _receive_chejan_data(self, gubun, item_cnt, fid_list):
    print(gubun)
    print(self.get_chejan_data(9203))
    print(self.get_chejan_data(302))
    print(self.get_chejan_data(900))
    print(self.get_chejan_data(901))
```

get_chejan_data 메서드는 함수 인자인 FID 값을 통해 서로 다른 데이터를 얻을 수 있습니다. 주요 FID 값에 대한 설명은 표 18.1을 참조하면 되며 더 자세한 FID 정보는 개발 가이드 8.19절을 참조합니다.

표 18.1 주문체결 FID

FID	설명
9203	주문번호
302	종목명

FID	설명
900	주문수량
901	주문가격
902	미체결수량
904	원주문번호
905	주문구분
908	주문/체결시간
909	체결번호
910	체결가
911	체결량
10	현재가, 체결가, 실시간종가

그림 18.24를 참조하면 계좌 정보가 QComboBox에 출력되는 것을 확인할 수 있습니다. OpenAPI+에서 계좌 정보 및 로그인 사용자 정보를 얻어오는 메서드는 GetLoginInfo입니다. 다음과 같이 dynamicCall 메서드로 GetLoginInfo 메서드를 호출하는 get_login_info 메서드를 Kiwoom 클래스에 추가합니다.

```
def get_login_info(self, tag):
    ret = self.dynamicCall("GetLoginInfo(QString)", tag)
    return ret
```

이로써 개발 2일 차에서 키움 OpenAPI+와 관련된 모든 내용을 업데이트했습니다.

pytrader.py 파일 업데이트

이번 절에서는 pytrader.py 파일을 업데이트하겠습니다. 가장 먼저 추가할 기능은 QLineEdit 위젯에 사용자가 코드명을 입력하면 해당 코드에 대한 종목명을 출력하는 것입니다. pytrader.py 파일에 위젯에 대한 코드를 구현할 때 가장 먼저 할 일은 위젯의 이름을 파악하는 것입니다. Qt Designer에서 그림 18.40과 같이 종목에 해당하는 QLineEdit 객체를 클릭한 후 Property 윈도우를 살펴보면 objectName 항목이 'lineEdit'임을 확인할 수 있습니다. 동일한 방법으로 아래쪽에 위치한 QLineEdit 객체의 이름을 확인하면 'lineEdit_2'임을 확인할 수 있습니다.

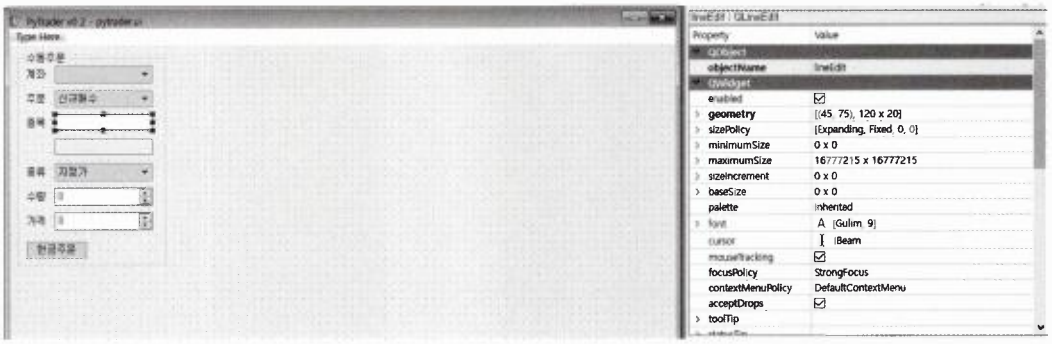


그림 18.40 objectName 확인

사용자가 lineEdit 객체에 종목코드를 입력하면 PyTrader는 사용자가 입력한 종목코드를 읽은 후 키움증권의 OpenAPI+를 사용해 종목명을 알아내야 합니다. 이를 위해 먼저 lineEdit 객체가 변경될 때 호출되는 슬롯을 지정합니다.

```
self.lineEdit.textChanged.connect(self.code_changed)
```

위에서 시그널 슬롯을 설정했기 때문에 lineEdit 객체로부터 textChanged라는 이벤트가 발생하면 MyWindow 클래스의 code_changed 메서드가 호출될 것입니다.

MyWindow 클래스에 code_changed 메서드를 다음과 같이 구현합니다. 먼저 사용자가 입력한 종목코드를 얻어옵니다. 종목코드에 대한 종목명은 Kiwoom 클래스에 구현된 get_master_code_name 메서드를 호출해서 알아냅니다. 마지막으로 종목명을 lineEdit_2 객체에 출력합니다.

```
def code_changed(self):
    code = self.lineEdit.text()
    name = self.kiwoom.get_master_code_name(code)
    self.lineEdit_2.setText(name)
```

이번에는 계좌 정보를 QComboBox 위젯에 출력하는 코드를 구현해 보겠습니다. 이를 위해서는 먼저 전체 계좌 개수와 계좌 번호를 키움증권으로부터 얻어와야 합니다. Kiwoom 클래스의 get_login_info 메서드를 사용해 다음과 같이 필요한 데이터를 얻어옵니다.

```
accouns_num = int(self.kiwoom.get_login_info("ACCOUNT_CNT"))
accounts = self.kiwoom.get_login_info("ACCNO")
```

계좌 번호를 QComboBox 위젯에 출력하려면 그림 18.41과 같이 먼저 Qt Designer에서 QComboBox를 선택한 후 Property 창에서 objectName을 확인합니다. 예제에서는 objectName이 comboBox이므로 self.comboBox를 통해 해당 객체에 접근할 수 있습니다.

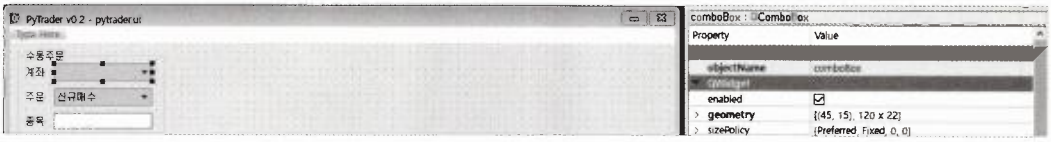


그림 18.41 QComboBox objectName 확인

계좌가 여러 개인 경우 각 계좌는 ';'를 통해 구분됩니다. 따라서 먼저 얻어온 전체 계좌에 대해 split 메서드를 호출해 리스트로 분리한 후 슬라이싱을 통해 출력할 계좌 번호를 선택합니다. 계좌를 QComboBox에 출력하기 위해 addItems 메서드를 호출합니다.

```
accounts_list = accounts.split(';')[0:accouns_num]
self.comboBox.addItems(accounts_list)
```

마지막으로 현금주문에 대한 코드를 구현해 봅시다. 현금주문은 UI 창에서 [현금주문] 버튼을 클릭할 때 수행됩니다. 따라서 먼저 버튼의 이름을 확인한 후 해당 버튼에 대한 시그널과 슬롯을 연결해야 합니다. 다음 코드를 MyWindow 클래스의 생성자에 추가합니다.

```
self.pushButton.clicked.connect(self.send_order)
```

pushButton 객체가 클릭될 때 호출되는 send_order 메서드를 구현해 봅시다. send_order 메서드에서는 사용자가 위젯을 통해 입력한 정보를 얻어온 후 이를 이용해 Kiwoom 클래스에 구현돼 있는 send_order 메서드를 호출하면 됩니다.

여기서 주의할 점은 사용자는 QComboBox를 통해 '신규매수', '신규매도', '매수취소', '매도취소'와 같은 세부 항목을 선택하지만 실제 키움증권의 API에는 세부 항목에 대응되는 정숫값이 전달돼야 한다는 점입니다. 마찬가지로 호가에서도 지정가일 때는 '00'이라는 문자열을 전달해야 하고, 시장가일 때는 '03'이라는 문자열을 전달해야 합니다.

```
def send_order(self):
    order_type_lookup = {'신규매수': 1, '신규매도': 2, '매수취소': 3, '매도취소': 4}
    hoga_lookup = {'지정가': "00", '시장가': "03"}

    account = self.comboBox.currentText()
    order_type = self.comboBox_2.currentText()
    code = self.lineEdit.text()
    hoga = self.comboBox_3.currentText()
    num = self.spinBox.value()
    price = self.spinBox_2.value()
```

```
        self.kiwoom.send_order("send_order_req", "0101", account, order_type_lookup[order_type],
    code, num, price, hoga_lookup[hoga], "")
```

매수 테스트

이로써 2일 차에 대한 구현이 모두 끝났습니다. 이번에는 구현한 프로그램을 실행해 실제로 종목 매수를 진행해 봅시다. 먼저 pytrader.py 파일을 관리자 권한으로 실행해 봅시다. 앞서 키움증권 자동 로그인 설정을 했다면 프로그램 시작과 동시에 자동 로그인이 수행될 것입니다.

로그인이 실행되면 그림 18.42와 같이 PyTrader가 실행됩니다. 먼저 거래할 계좌 번호를 선택한 후 주문을 신규매수로 선택합니다. 매수하고자 하는 종목코드를 입력하면 자동으로 해당 종목명이 출력되는 기능이 잘 동작하는 것을 확인할 수 있습니다.

종류에 시장가를 선택하고 적당한 수량을 입력한 후 [현금주문] 버튼을 클릭해보기 바랍니다. 장중(09:00~15:30)이라면 정상적으로 매수가 진행될 것입니다. 참고로 모의투자 서버를 사용한다고 하더라도 주문과 관련된 API는 장중에만 테스트할 수 있습니다.

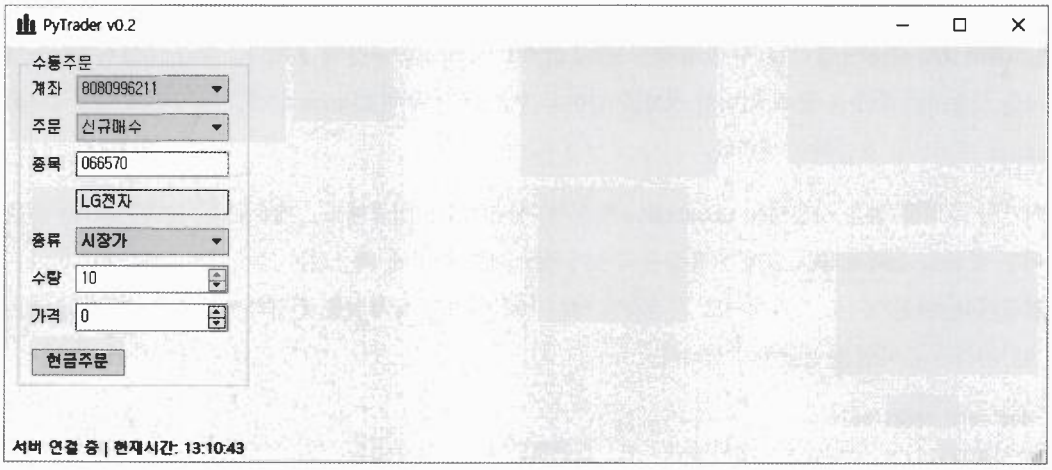


그림 18.42 신규매수 테스트

매수 API가 정상적으로 동작했는지 확인하는 가장 쉬운 방법은 HTS를 사용하는 것입니다. 키움증권의 HTS 중 하나인 영웅문을 사용해 그림 18.43과 같이 모의투자로 접속하면 해당 계좌에서 매수한 종목 목록을 쉽게 확인할 수 있습니다.

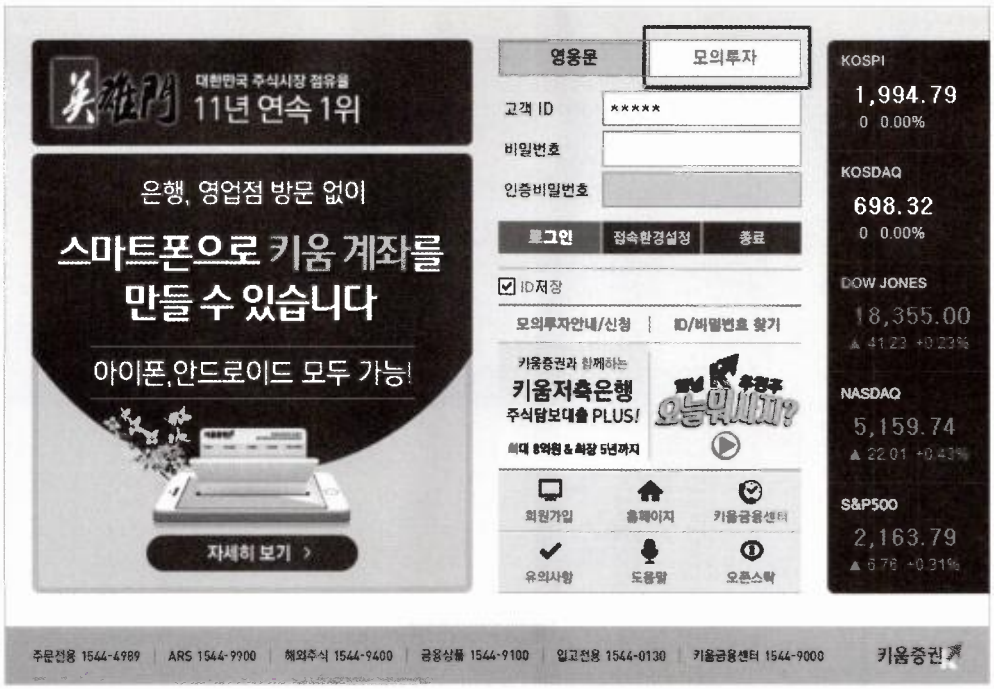


그림 18.43 영웅문을 이용한 모의투자 접속

또 다른 방법은 KOA Studio를 사용하는 것입니다. 그림 18.44처럼 TR 목록에서 opt10085를 선택한 후 오른쪽 계좌번호에 여러분의 모의투자 계좌번호를 입력한 후 [조회] 버튼을 클릭하면 화면 하단 출력부에 계좌 수익률 데이터가 출력됩니다. 출력된 데이터를 살펴보면 여러분이 API를 통해 매수한 종목이 출력되는 것을 확인할 수 있을 것입니다.

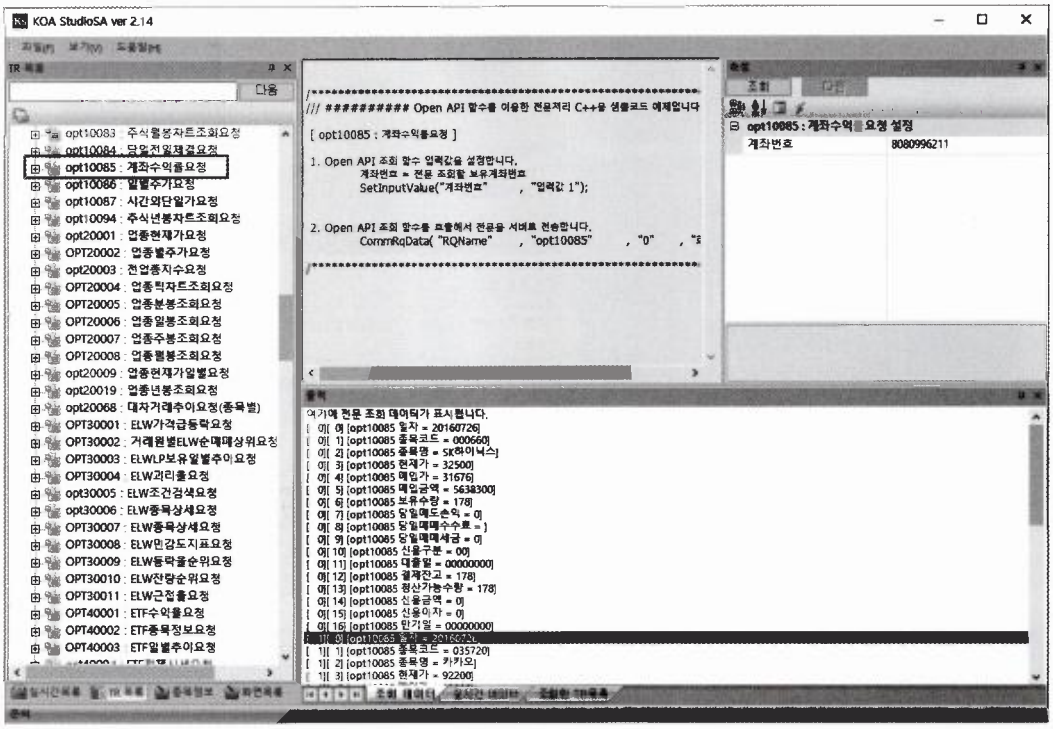


그림 18.44 KOA Studio opt10085를 이용한 매수 종목 확인

03 개발 3일 차

개발 3일 차에는 보유 주식 종목 현황 출력과 잔고확인 기능을 구현해 보겠습니다. 보유 주식 종목 현황 출력 기능이란 여러분이 매수한 종목에 대해 종목명, 보유량, 매입가, 현재가, 평가손익, 수익률 등을 보여주는 기능을 말합니다. 예를 들어, 그림 18.45는 키움증권의 HTS인 영웅문4에서 볼 수 있는 주식종합 화면 중 일부인데 이 같은 형태로 보유 종목에 대한 현황을 출력할 것입니다.

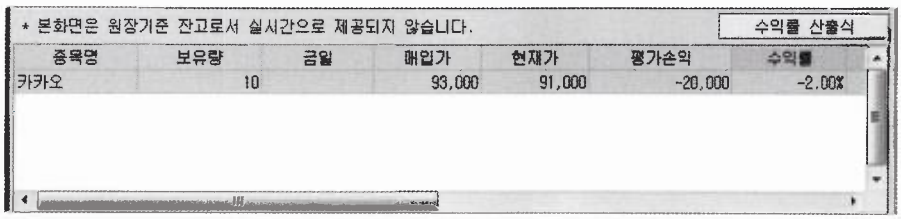


그림 18.45 영웅문의 주식종합 잔고확인 화면(보유 주식 종목 현황)

잔고확인 기능이란 선택된 계좌의 예수금, 총 매입 금액, 총평가 금액, 총 손익 금액, 총 수익률, 추정 자산 등을 출력하는 기능을 말합니다. 이 역시 그림 18.46과 같이 영웅문의 주식종합 화면에서 확인할 수 있는데 여러분도 이와 비슷한 형태로 해당 정보가 출력되도록 구현할 것입니다.

예수금	99,442,576	총평가	910,000	총수익률	-2.15%	조회
총매입	930,000	총손익	-20,000	추정자산	100,346,666	다음

그림 18.46 영웅문의 주식종합 잔고확인 화면(잔고 현황)

UI 구성

먼저 UI부터 구성해 보겠습니다. 3일 차 구현을 위해 2일 차까지 업데이트된 UI 파일을 준비합니다. UI 파일이 없는 분은 아래의 깃허브 저장소에서 관련 파일을 내려받을 수 있습니다(/ch18/day02/pytrader.ui).

- 깃허브 저장소: https://github.com/pystockhub/book

Qt Designer를 실행한 후 2일 차까지 작업해둔 pytrader.ui 파일을 엽니다. 먼저 윈도우 타이틀을 PyTrader v0.3으로 변경합니다. 윈도우 타이틀을 변경했다면 본격적인 UI 수정에 앞서 Ctrl + R 키를 눌러 수정한 UI를 확인해 봅시다. 그림 18.47과 같이 문제 없이 PyTrader v0.3이 실행되는 것을 확인할 수 있습니다.

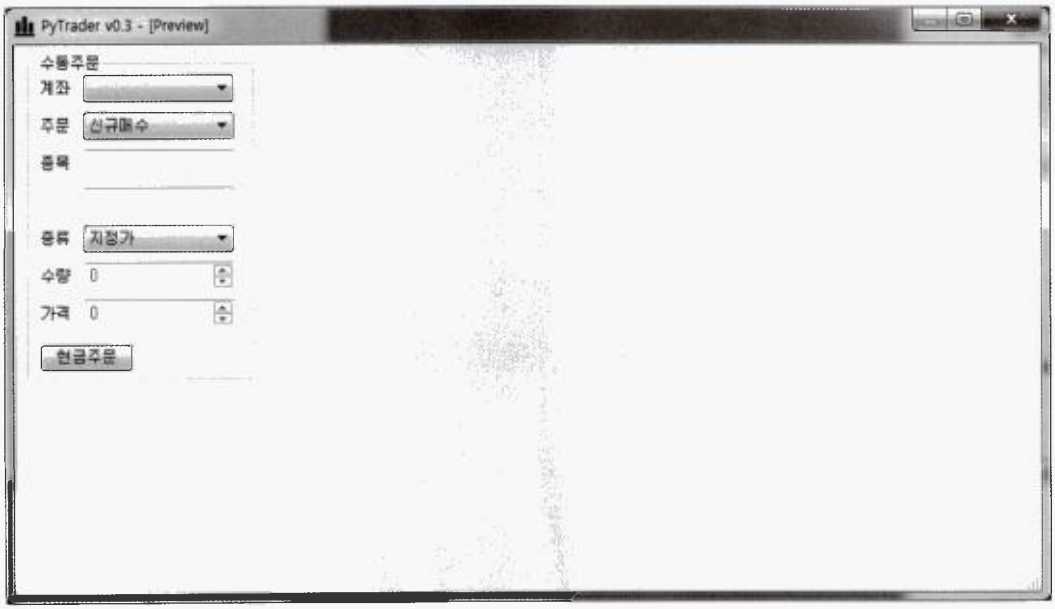


그림 18.47 윈도우 타이틀을 변경한 후 실행한 모습

Qt Designer의 Widget Box에서 Containers 항목 중 Group Box를 사용해 그림 18.48과 같이 화면을 구성해 봅시다. Group Box 위젯을 드래그 앤드 드롭한 후 title 항목을 '잔고 및 보유종목 현황'으로 변경합니다.

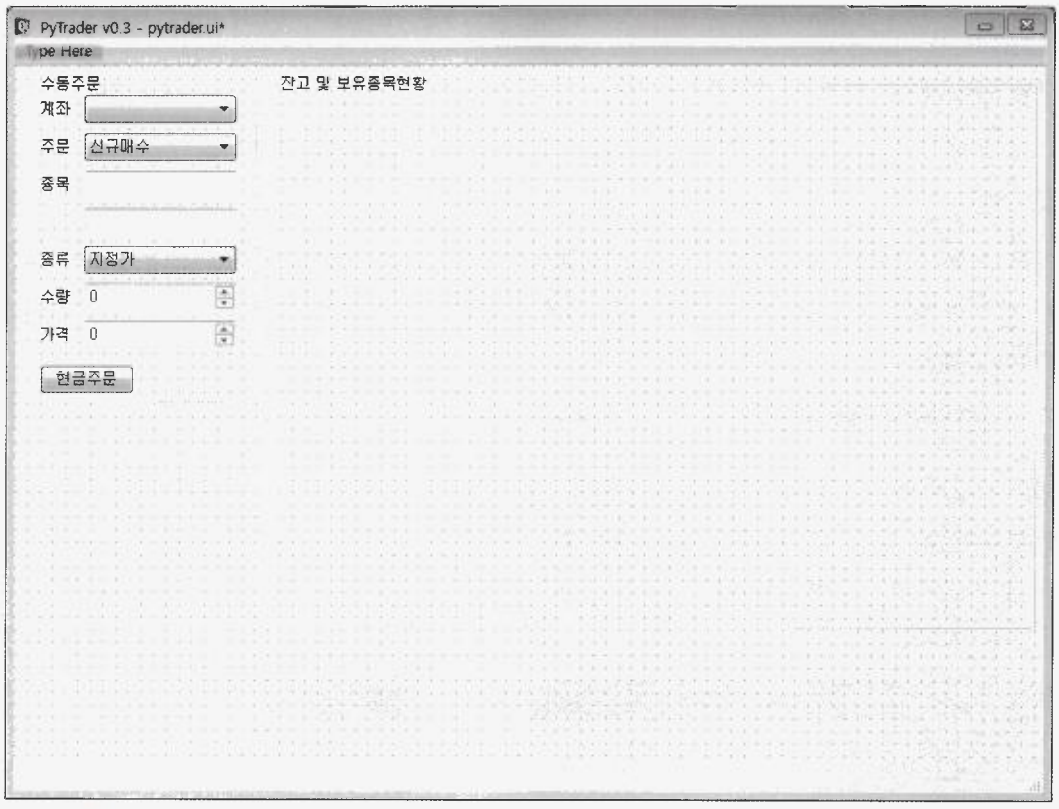


그림 18.48 Group Box 위젯 배치

이번에는 Item Widgets 항목 중 Table Widget을 선택한 후 그림 18.49와 같이 앞서 생성한 Group Box 안으로 배치합니다. 그리고 위젯의 크기 및 위치를 적당히 조절합니다.

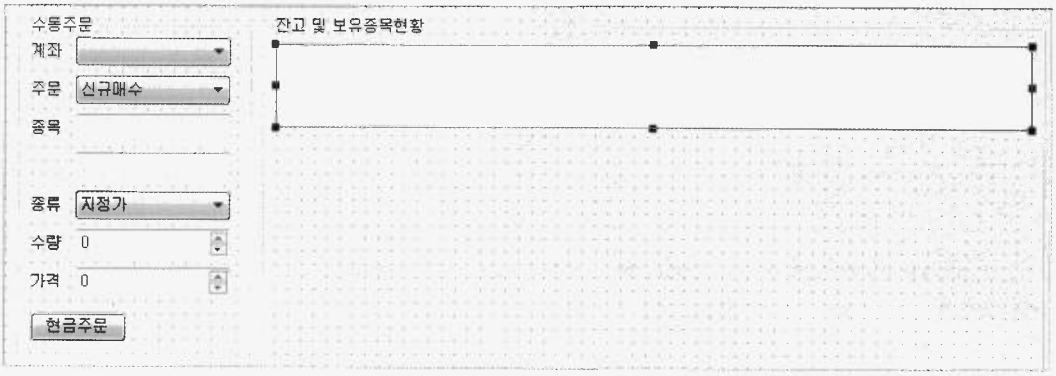


그림 18.49 Table Widget 배치

Table Widget에 칼럼(Column)을 추가하기 위해 그림 18.50과 같이 마우스 오른쪽 버튼을 클릭한 후 Edit Items 항목을 선택합니다.

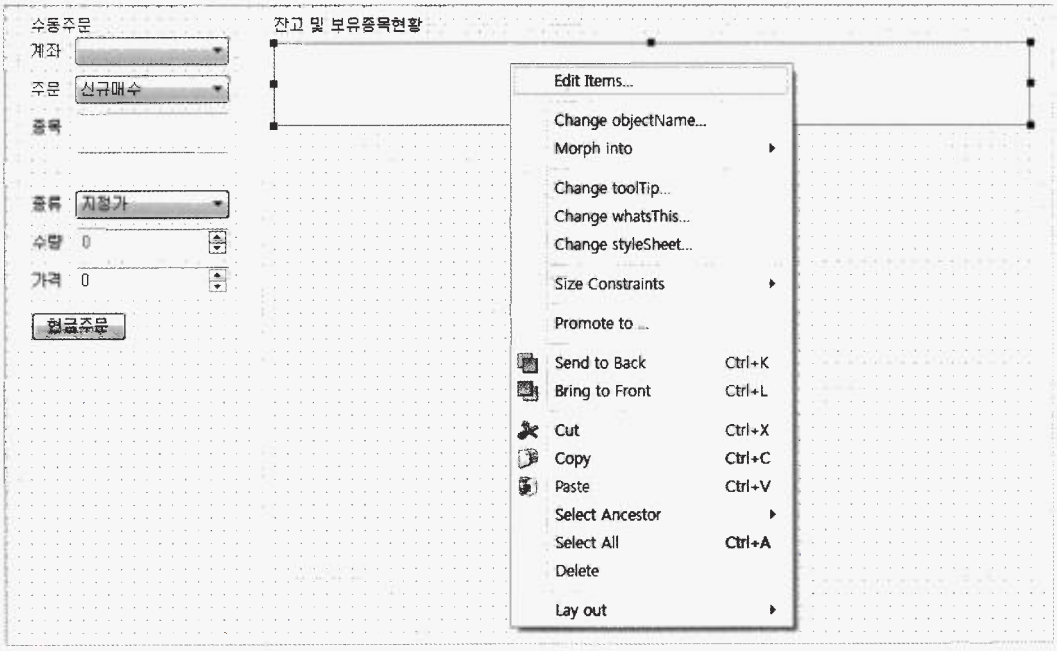


그림 18.50 Edit Items 항목 선택(Table Widget)

그림 18.51과 같이 Columns 탭에서 + 버튼을 클릭해 예수금(d+2), 총매입, 총평가, 총손익, 총수익률(%), 추정자산 항목을 추가합니다. 입력을 마치면 [OK] 버튼을 눌러 해당 창을 닫습니다.

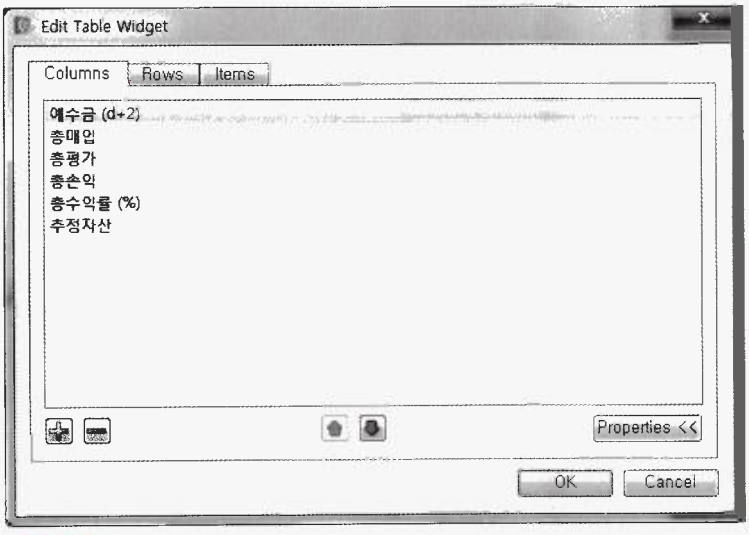


그림 18.51 칼럼 항목 추가(Table Widget)

Table Widget을 선택한 후 그림 18.52와 같이 Property Editor의 QTableWidget 항목에서 rowCount 값을 1로 변경합니다. QTableWidget 객체는 rowCount 값을 제대로 설정하지 않으면 아이템을 추가할 수 없으므로 반드시 값을 설정해야 합니다.

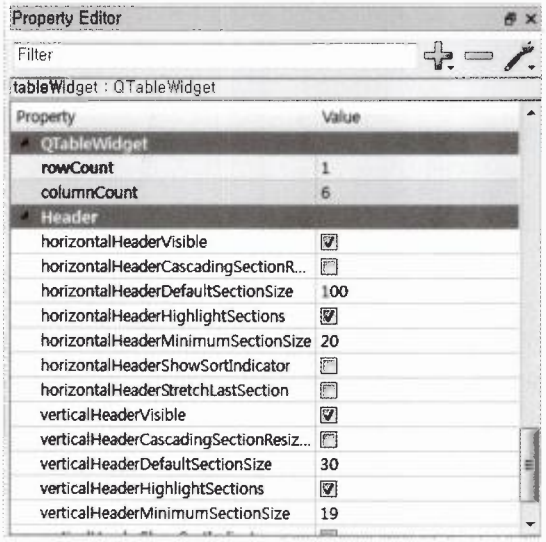


그림 18.52 rowCount 조정(Table Widget)

이번에는 보유종목 현황을 조회하는 UI를 구성해 봅시다. 앞서 잔고와 같은 방식으로 그림 18.53과 같이 구성합니다.

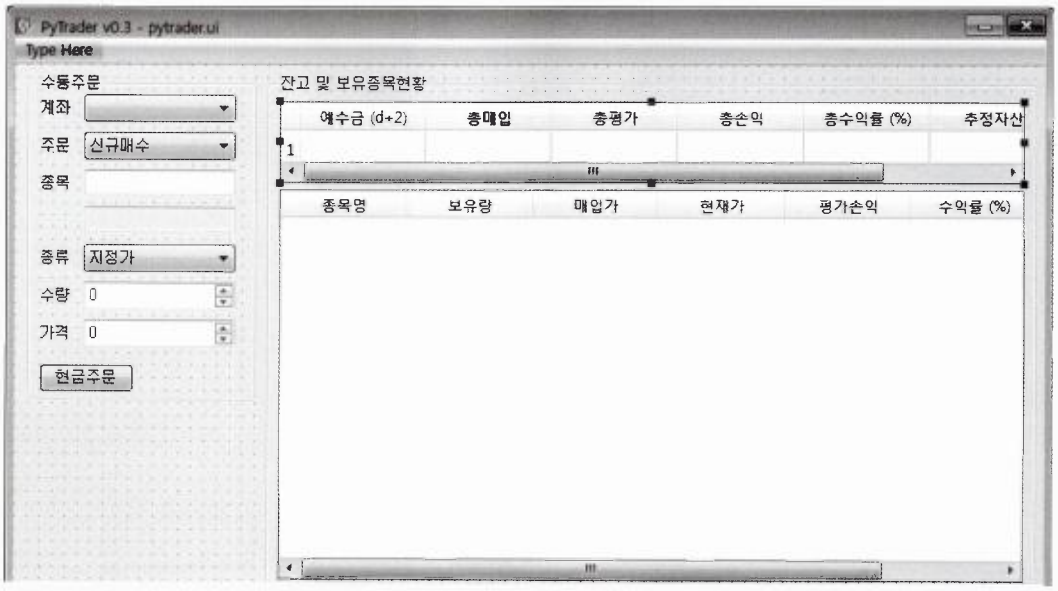


그림 18.53 보유종목 현황을 확인하기 위한 Table Widget 추가

보유종목 현황에 쓸 Table Widget을 선택한 후 그림 18.54와 같이 6개의 칼럼을 추가합니다.

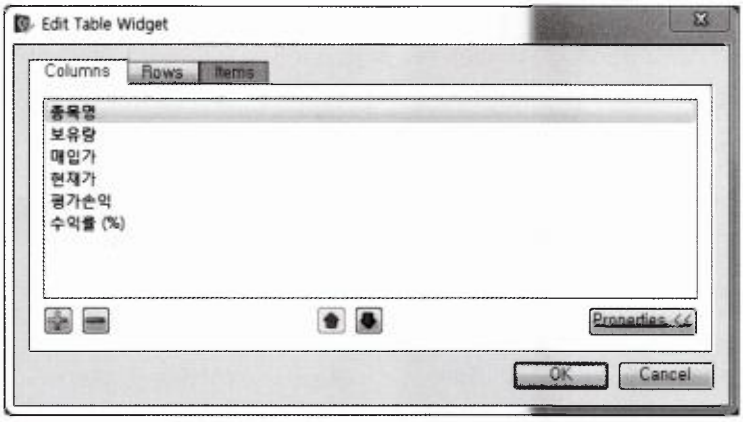


그림 18.54 보유종목 현황 칼럼 추가(Table Widget)

이번에는 Buttons 항목에서 Check Box와 Push Button을 사용해 그림 18.55와 같이 위젯을 배치합니다. Check Box의 경우 Property Editor에서 text 항목을 '실시간 조회'로 변경하고 버튼의 text 항목을 '조회'로 변경합니다.

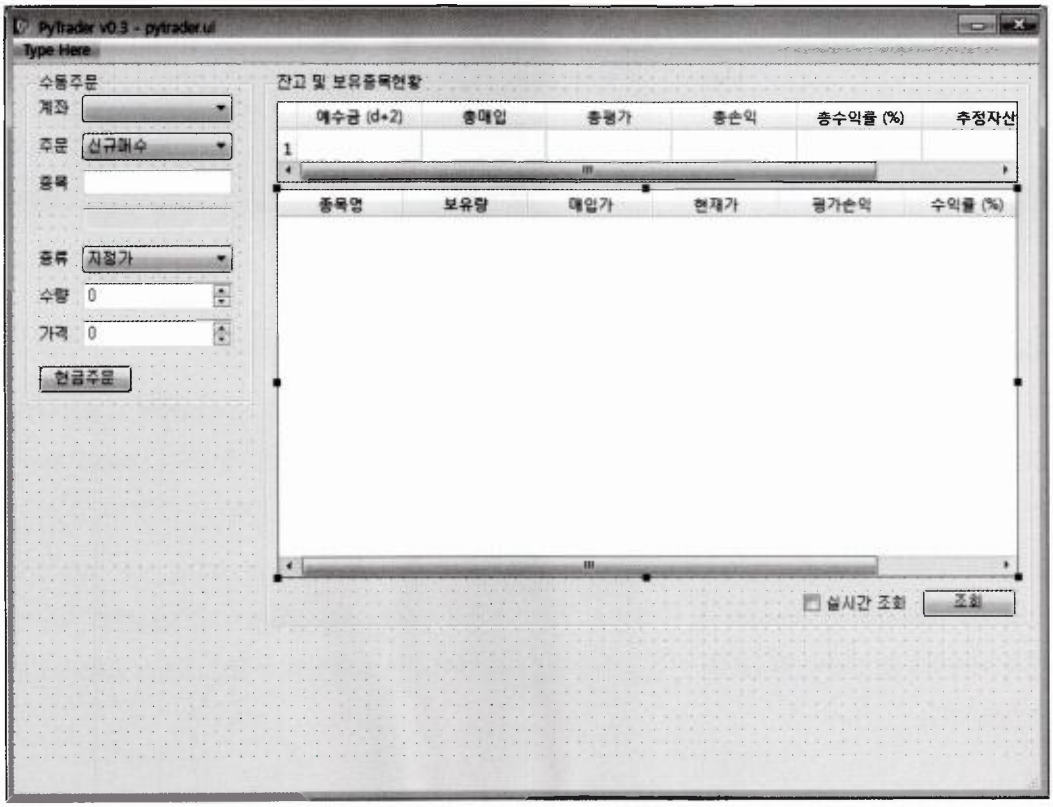


그림 18.55 Check Box 및 Push Button 배치

이로써 3일 차의 UI 구성을 끝마쳤습니다. 지금까지 작업한 내용을 확인하기 위해 Qt Designer에서 Ctrl + R 키를 눌러 실행해 보면 그림 18.56과 같은 UI를 확인할 수 있습니다. 참고로 보유종목 현황에 대한 QTableWidget 객체는 행의 개수를 Qt Designer를 통해 설정하지 않았습니다. 이는 아직 QTableWidget 객체에 입력될 데이터의 개수를 모르기 때문입니다. 이 부분은 pytrader.py 파일에서 데이터를 추가할 때 설정하겠습니다.

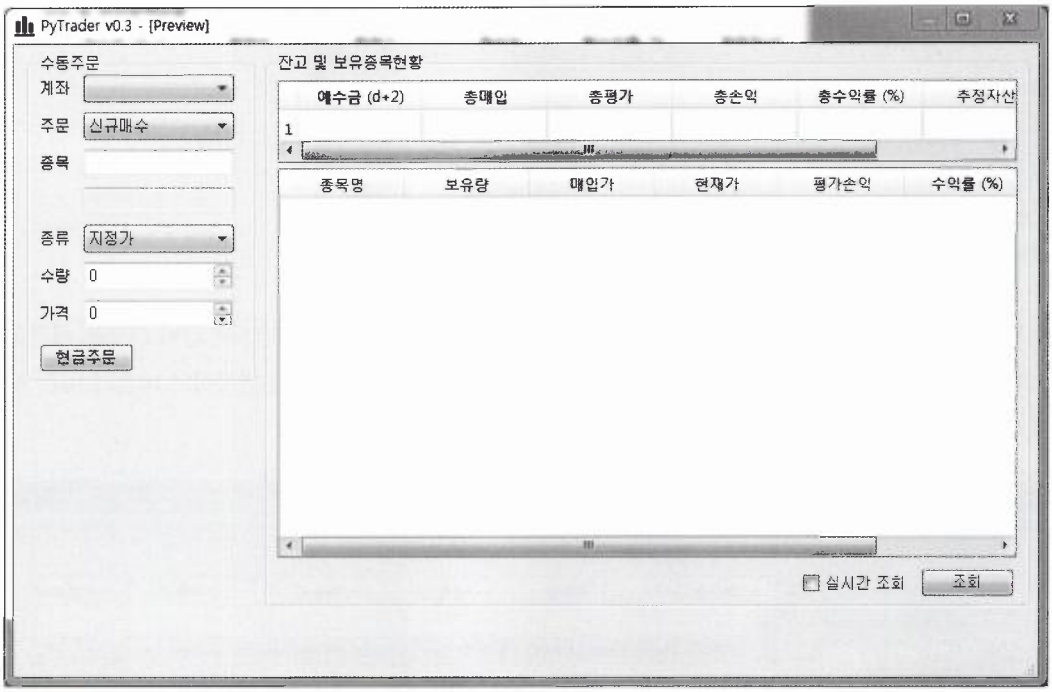


그림 18.56 개발 3일 차의 최종 UI 구성

Kiwoom.py 파일 업데이트

이번에는 잔고 및 보유종목 현황 조회 기능을 구현하기 위해 Kiwoom 클래스를 업데이트해 보겠습니다. KOA Studio를 참고하면 잔고 및 보유종목 현황 출력 기능에 필요한 대부분의 데이터는 opw00018이라는 TR을 통해 얻을 수 있음을 확인할 수 있습니다.

본격적인 코드 작성에 앞서 먼저 KOA Studio를 통해 opw00018의 출력 값을 확인해 봅시다. 이를 위해 그림 18.57과 같이 KOA Studio를 실행한 후 TR 목록에서 opw00018을 선택합니다. 오른쪽의 속성 창에서 계좌번호를 입력한 후 [조회] 버튼을 누르면 '조회 데이터'란에 데이터가 출력되는 모습을 확인할 수 있습니다.

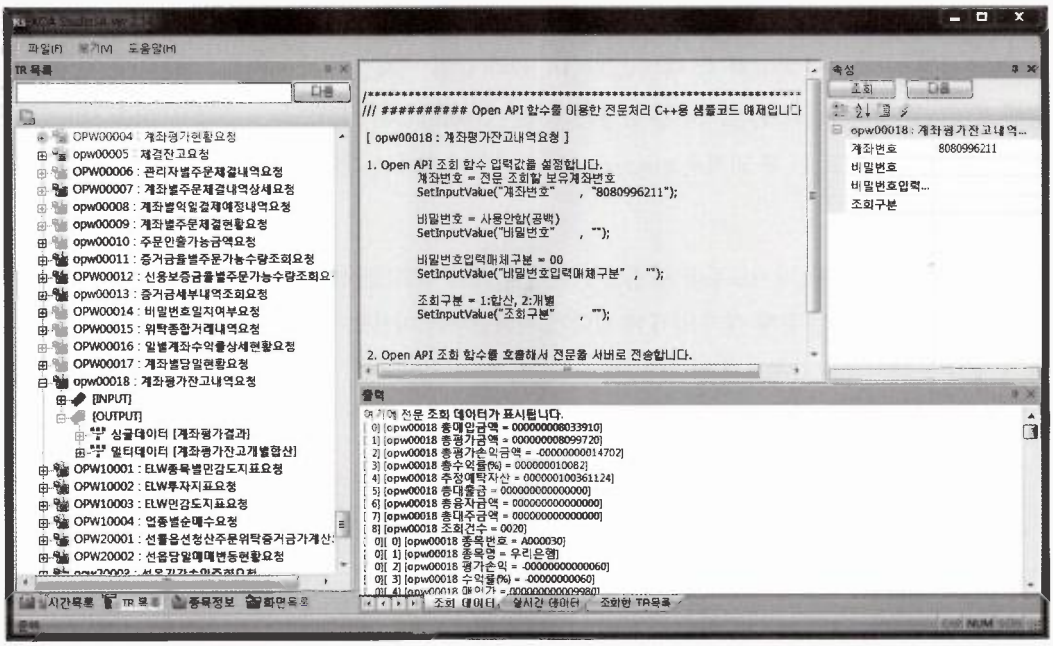


그림 18.57 opw00018 TR 사용(KOA Studio)

opw00018 TR을 사용하면 잔고 및 보유종목 현황에 관련된 대부분의 데이터를 얻을 수 있지만 예수금 정보는 얻을 수 없습니다. 예수금 정보를 얻기 위해서는 그림 18.58과 같이 opw00001 TR을 사용해야 합니다. KOA Studio에서 opw00001 TR을 통해 데이터를 조회해 봅시다.

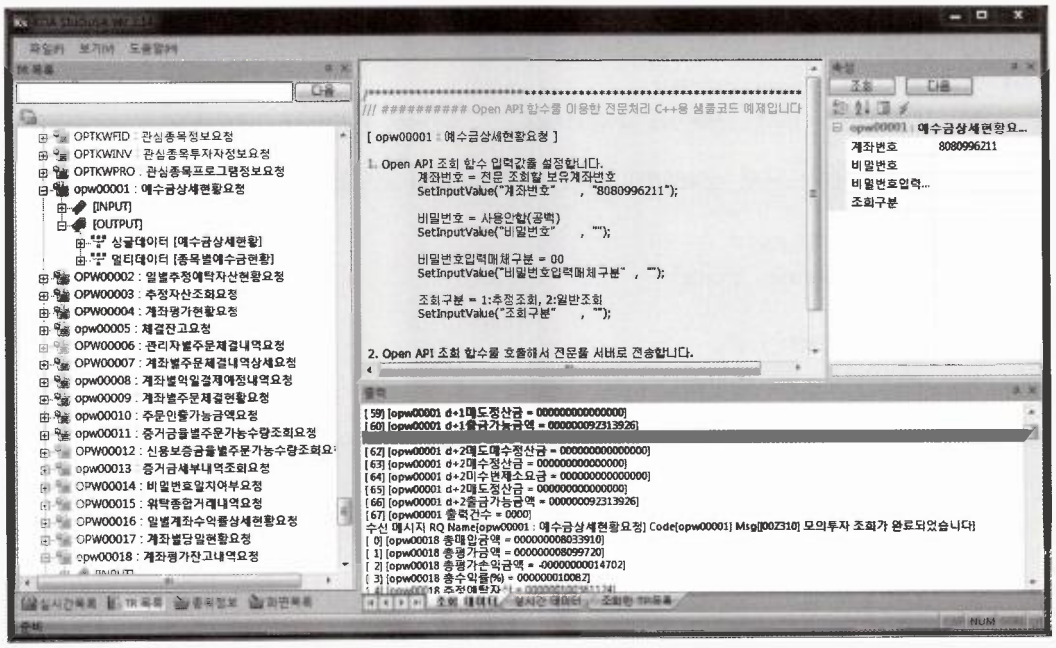


그림 18.58 opw00001 TR 사용(KOA Studio)

Kiwoom 클래스에 코드를 추가하기 전에 TR 요청 및 데이터를 가져오는 전반적인 과정을 먼저 복습해 봅시다. 특정 TR을 사용하고자 할 때는 그림 18.59와 같이 먼저 SetInputValue 메서드를 호출해서 입력 데이터를 설정했습니다. 데이터를 설정한 후에는 CommRqData 메서드를 호출해서 TR을 서버로 전송하면 됩니다. 참고로 이 두 과정은 Kiwoom.py 파일에서 수행하는 것이 아니라 pytrader.py 파일에서 처리할 것입니다.

TR이 서버로 전송된 후 서버로부터 이벤트가 발생할 때까지 TR을 전송한 프로그램은 기다려야 합니다. 보통 이벤트 루프를 통해 대기 상태에 있다가 서버로부터 이벤트가 발생하면 CommGetData 메서드를 통해 수신 데이터를 가져오게 됩니다.

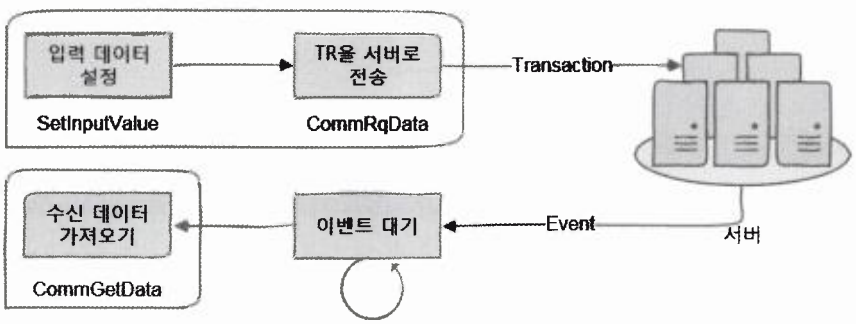


그림 18.59 TR 요청 및 수신 데이터 가져오기

먼저 OnReceiveTrData 이벤트가 발생할 때 수신 데이터를 가져오는 함수인 _opw00001을 Kiwoom 클래스에 추가합니다.

```
def _opw00001(self, rqname, trcode):
    self.d2_deposit = self._comm_get_data(trcode, "", rqname, 0, "d+2추정예수금")
```

_receive_tr_data 메서드에서 _opw00001 메서드를 호출하도록 코드를 수정합니다.

```
if rqname == "opt10081_req":
    self._opt10081(rqname, trcode)

elif rqname == "opw00001_req":
    self._opw00001(rqname, trcode)
```

opw00001 TR을 사용하기 위한 코드를 Kiwoom 클래스에 모두 업데이트했습니다. 'd+2추정예수금'을 잘 얻어오는지 확인해보기 위해 Kiwoom.py 파일의 __main__ 부분을 다음과 같이 수정합니다

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    kiwoom = Kiwoom()
    kiwoom.comm_connect()

    kiwoom.set_input_value("계좌번호", "8087711111")
    kiwoom.set_input_value("비밀번호", "0000")
    kiwoom.comm_rq_data("opw00001_req", "opw00001", 0, "2000")

    print(kiwoom.d2_deposit)
```

얻어온 'd+2추정예수금'을 확인하면 '000000499225300'과 같이 문자열의 앞쪽에 0이 존재하는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch18/day03/Kiwoom.py
connected
000000499225300
```

보통 금액은 천의 자리마다 콤마를 사용해서 표시합니다. 이를 위해 Kiwoom 클래스에 change_format 이라는 정적 메서드(static method)를 추가합니다. change_format 메서드는 입력된 문자열에 대해 lstrip 메서드를 통해 문자열 왼쪽에 존재하는 '-' 또는 '0'을 제거합니다. 그리고 format 함수를 통해 천의 자리마다 콤마를 추가한 문자열로 변경합니다.

```
@staticmethod
def change_format(data):
    strip_data = data.lstrip('-0')
    if strip_data == '' or strip_data == '.00':
        strip_data = '0'

    format_data = format(int(strip_data), ',d')
    if data.startswith('-'):
        format_data = '-' + format_data

    return format_data
```

_opw00001에서 얻어온 'd+2추정예수금'의 포맷을 변경하기 위해 change_format 메서드를 호출합니다. 정적 메서드를 호출하려면 정적 메서드 이름 앞에 클래스 이름을 붙여야 합니다.

```
def _opw00001(self, rqname, trcode):
    d2_deposit = self._comm_get_data(trcode, "", rqname, 0, "d+2추정예수금")
    self.d2_deposit = Kiwoom.change_format(d2_deposit)
```

이번에는 opw00018 TR을 위한 코드를 추가하겠습니다. opw00018 TR은 싱글 데이터를 통해 계좌에 대한 평가 잔고 데이터를 제공하며 멀티 데이터를 통해 보유 종목별 평가 잔고 데이터를 제공합니다.

먼저 총매입금액, 총평가금액, 총평가손익금액, 총수익률, 추정예탁자산을 _comm_get_data 메서드를 통해 얻어옵니다. 얻어온 데이터는 change_format 메서드를 통해 포맷을 문자열로 변경합니다.

```
def _opw00018(self, rqname, trcode):
    total_purchase_price = self._comm_get_data(trcode, "", rqname, 0, "총매입금액")
    total_eval_price = self._comm_get_data(trcode, "", rqname, 0, "총평가금액")
    total_eval_profit_loss_price = self._comm_get_data(trcode, "", rqname, 0, "총평가손익금액")
    total_earning_rate = self._comm_get_data(trcode, "", rqname, 0, "총수익률(%)")
    estimated_deposit = self._comm_get_data(trcode, "", rqname, 0, "추정예탁자산")

    print(Kiwoom.change_format(total_purchase_price))
    print(Kiwoom.change_format(total_eval_price))
    print(Kiwoom.change_format(total_eval_profit_loss_price))
    print(Kiwoom.change_format(total_earning_rate))
    print(Kiwoom.change_format(estimated_deposit))
```

_receive_tr_data 메서드에서 _opw00018 메서드를 호출하도록 다음과 같이 코드를 수정합니다.

```
if rqname == "opt10081_req":
    self._opt10081(rqname, trcode)
elif rqname == "opw00001_req":
    self._opw00001(rqname, trcode)
elif rqname == "opw00018_req":
    self._opw00018(rqname, trcode)
```

opw00018 TR을 통해 싱글 데이터를 잘 얻어오는지 테스트하기 위해 Kiwoom.py 파일의 __main__ 부분을 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    kiwoom = Kiwoom()
    kiwoom.comm_connect()

    account_number = kiwoom.get_login_info("ACCNO")
    account_number = account_number.split(';')[0]

    kiwoom.set_input_value("계좌번호", account_number)
    kiwoom.comm_rq_data("opw00018_req", "opw00018", 0, "2000")
```

Kiwoom.py 파일을 실행하면 다음과 같이 정상적으로 데이터를 얻어오는 것을 확인할 수 있습니다.

```
connected
1,908,300
1,924,200
-3,252
101
499,996,748
```

만약 Kiwoom.py 파일을 실행했을 때 그림 18.60과 같은 에러가 발생한다면 그림 18.22를 참조해서 계좌 비밀번호를 등록한 후 자동 로그인 설정을 하면 됩니다.

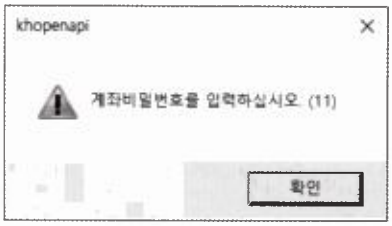


그림 18.60 계좌 비밀번호 에러

이번에는 멀티 데이터를 통해 보유 종목별로 평가 잔고 데이터를 얻어와 보겠습니다. 다음 코드를 _opw00018에 추가합니다. 멀티 데이터는 먼저 _get_repeat_cnt 메서드를 호출해 보유 종목의 개수를 얻어옵니다. 그런 다음 해당 개수만큼 반복하면서 각 보유 종목에 대한 상세 데이터를 _comm_get_data를 통해 얻어옵니다. 참고로 opw00018 TR을 사용하는 경우 한 번의 TR 요청으로 최대 20개의 보유 종목에 대한 데이터를 얻을 수 있습니다.

```
# multi data
rows = self._get_repeat_cnt(trcode, rqname)
for i in range(rows):
    name = self._comm_get_data(trcode, "", rqname, i, "종목명")
    quantity = self._comm_get_data(trcode, "", rqname, i, "보유수량")
    purchase_price = self._comm_get_data(trcode, "", rqname, i, "매입가")
    current_price = self._comm_get_data(trcode, "", rqname, i, "현재가")
    eval_profit_loss_price = self._comm_get_data(trcode, "", rqname, i, "평가손익")
    earning_rate = self._comm_get_data(trcode, "", rqname, i, "수익률(%)")

    quantity = Kiwoom.change_format(quantity)
    purchase_price = Kiwoom.change_format(purchase_price)
    current_price = Kiwoom.change_format(current_price)
    eval_profit_loss_price = Kiwoom.change_format(eval_profit_loss_price)
    earning_rate = Kiwoom.change_format2(earning_rate)

    print(name, quantity, purchase_price, current_price, eval_profit_loss_price, earning_rate)
```

수익률에 대한 포맷 변경은 change_format2라는 정적 메서드를 사용합니다.

```
@staticmethod
def change_format2(data):
    strip_data = data.lstrip('-0')

    if strip_data == '':
        strip_data = '0'

    if strip_data.startswith('.'):
        strip_data = '0' + strip_data

    if data.startswith('-'):
        strip_data = '-' + strip_data

    return strip_data
```

Kiwoom.py 파일을 실행하면 다음과 같이 각 보유 종목에 대한 평가 잔고 데이터가 출력되는 것을 확인할 수 있습니다.

```
SK하이닉스 10 47,100 47,300 2,000 0.42
키움증권 10 77,200 77,600 4,000 0.52
LG전자 10 61,100 61,700 6,000 0.98
AJ네트웍스 10 5,430 5,460 300 0.55
```

이번에는 opw00018 TR을 통해 얻어온 데이터를 인스턴스 변수에 저장해 보겠습니다. 먼저 Kiwoom 클래스에 다음 메서드를 추가합니다.

```
def reset_opw00018_output(self):
    self.opw00018_output = {'single': [], 'multi': []}
```

_opw00018 메서드는 예제 18.4와 같이 수정합니다. 싱글 데이터는 1차원 리스트로 데이터를 저장하며, 멀티 데이터는 2차원 리스트로 데이터를 저장합니다.

예제 18.4 _opw00018 메서드

```
def _opw00018(self, rqname, trcode):
    # single data
    total_purchase_price = self._comm_get_data(trcode, "", rqname, 0, "총매입금액")
    total_eval_price = self._comm_get_data(trcode, "", rqname, 0, "총평가금액")
    total_eval_profit_loss_price = self._comm_get_data(trcode, "", rqname, 0, "총평가손익금액")
```

```
        total_earning_rate = self._comm_get_data(trcode, "", rqname, 0, "총수익률(%)")
        estimated_deposit = self._comm_get_data(trcode, "", rqname, 0, "추정예탁자산")

        self.opw00018_output['single'].append(Kiwoom.change_format(total_purchase_price))
        self.opw00018_output['single'].append(Kiwoom.change_format(total_eval_price))
        self.opw00018_output['single'].append(Kiwoom.change_format(total_eval_profit_loss_price))
        self.opw00018_output['single'].append(Kiwoom.change_format(total_earning_rate))
        self.opw00018_output['single'].append(Kiwoom.change_format(estimated_deposit))

        # multi data
        rows = self._get_repeat_cnt(trcode, rqname)
        for i in range(rows):
            name = self._comm_get_data(trcode, "", rqname, i, "종목명")
            quantity = self._comm_get_data(trcode, "", rqname, i, "보유수량")
            purchase_price = self._comm_get_data(trcode, "", rqname, i, "매입가")
            current_price = self._comm_get_data(trcode, "", rqname, i, "현재가")
            eval_profit_loss_price = self._comm_get_data(trcode, "", rqname, i, "평가손익")
            earning_rate = self._comm_get_data(trcode, "", rqname, i, "수익률(%)")

            quantity = Kiwoom.change_format(quantity)
            purchase_price = Kiwoom.change_format(purchase_price)
            current_price = Kiwoom.change_format(current_price)
            eval_profit_loss_price = Kiwoom.change_format(eval_profit_loss_price)
            earning_rate = Kiwoom.change_format2(earning_rate)

            self.opw00018_output['multi'].append([name, quantity, purchase_price, current_price,
eval_profit_loss_price, earning_rate])
```

opw00018 TR을 사용할 때 한 가지 주의 사항은 실 서버로 접속할 때와 모의투자 서버로 접속할 때 제공되는 데이터의 형식이 다르다는 점입니다. 예를 들어, 실 서버에서 수익률은 소수점 표시 없이 전달되지만 모의투자에서는 소수점을 포함해서 데이터가 전달됩니다. 따라서 접속 서버를 구분해서 데이터를 다르게 처리할 필요가 있습니다. 이를 위해 Kiwoom 클래스에 다음 메서드를 추가합니다.

```
def get_server_gubun(self):
    ret = self.dynamicCall("KOA_Functions(QString, QString)", "GetServerGubun", "")
    return ret
```

_opw00018 메서드에서 모의투자일 때는 '총수익률(%)'의 값을 100으로 나눈 후 출력되도록 코드를 다음과 같이 수정합니다.

```
total_earning_rate = Kiwoom.change_format(total_earning_rate)

if self.get_server_gubun():
    total_earning_rate = float(total_earning_rate) / 100
    total_earning_rate = str(total_earning_rate)

self.opw00018_output['single'].append(total_earning_rate)
```

pytrader.py 파일 업데이트

이번 절에서는 Kiwoom 클래스를 사용해 잔고 및 보유종목 현황 데이터를 요청하고 얻어온 데이터를 UI에 출력하는 코드를 작성해 보겠습니다.

그림 18.61은 PyTrader v0.3을 실행한 화면입니다. PyTrader v0.3이 실행된 후 사용자가 [조회] 버튼을 클릭하면 잔고 및 보유종목 현황이 QTableWidget 객체를 통해 출력됩니다. 물론 출력 데이터는 opw00001 TR과 opw00018 TR을 통해 키움증권 서버로부터 가져온 것입니다.



그림 18.61 PyTrader를 실행한 모습

사용자가 버튼을 클릭했을 때 특정 메서드가 호출되는 것을 구현하기 위해 먼저 Qt Designer에서 버튼의 이름을 확인합니다. 예제의 경우 그림 18.62와 같이 버튼 객체의 이름이 'pushButton_2'임을 확인할 수 있습니다.



그림 18.62 버튼 객체의 이름 확인

버튼 객체의 이름을 확인했으므로 MyWindow 클래스의 생성자에 시그널과 슬롯을 연결하는 코드를 추가합니다. 다음 코드는 pushButton_2라는 객체가 클릭될 때 check_balance라는 메서드가 호출되게 합니다.

```
self.pushButton_2.clicked.connect(self.check_balance)
```

이번에는 버튼이 클릭됐을 때 호출되는 check_balance 메서드를 구현해 봅시다. 먼저 opw00018 TR을 위한 입력 데이터 설정(SetInputValue)과 TR을 요청(CommRqData)하는 코드를 구현합니다. 앞서 설명한 것처럼 opw00018 TR은 최대 20개의 보유 종목에 대한 데이터를 리턴합니다. 따라서 보유 종목이 20개 이상인 경우를 고려해서 반복문을 통해 연속적으로 데이터를 요청하도록 구현해야 합니다.

```
def check_balance(self):
    self.kiwoom.reset_opw00018_output()
    account_number = self.kiwoom.get_login_info("ACCNO")
    account_number = account_number.split(';')[0]

    self.kiwoom.set_input_value("계좌번호", account_number)
    self.kiwoom.comm_rq_data("opw00018_req", "opw00018", 0, "2000")
```

```
    while self.kiwoom.remained_data:
        time.sleep(0.2)
        self.kiwoom.set_input_value("계좌번호", account_number)
        self.kiwoom.comm_rq_data("opw00018_req", "opw00018", 2, "2000")
```

이번에는 예수금 데이터를 얻기 위해 opw00001 TR을 요청하는 코드를 구현해 봅시다. opw00001 TR은 연속적으로 데이터를 요청할 필요가 없으므로 상당히 간단합니다.

```
# opw00001
self.kiwoom.set_input_value("계좌번호", account_number)
self.kiwoom.comm_rq_data("opw00001_req", "opw00001", 0, "2000")
```

TR 요청을 통해 데이터가 준비되면 데이터를 QTableWidget 객체에 출력하면 됩니다. Qt Designer를 통해 QTableWidget 객체의 이름을 확인해보면 tableWidget임을 확인할 수 있습니다. 따라서 self.tableWidget을 통해 해당 객체에 접근할 수 있습니다.

예수금 데이터를 QTableWidget에 출력하기 위해 먼저 self.kiwoom.d2_deposit에 저장된 예수금 데이터를 QTableWidgetItem 객체로 만듭니다. setItem 메서드를 호출해 QTableWidget 객체에 넣으면 됩니다. 이때 칼럼의 위치로 (0, 0)을 사용합니다.

```
# balance
item = QTableWidgetItem(self.kiwoom.d2_deposit)
item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)
self.tableWidget.setItem(0, 0, item)
```

'총매입', '총평가', '총손익', '총수익률(%)', '추정자산'을 QTableWidget의 칼럼에 추가하는 코드를 구현해 봅시다. 데이터는 self.kiwoom.opw00018_output['single']을 통해 얻어올 수 있습니다. 파이썬 리스트로 데이터가 저장돼 있으므로 반복문을 사용해 데이터를 하나씩 가져온 후 해당 데이터를 QTableWidgetItem 객체로 변환하고 setItem 메서드를 호출해 QTableWidget 객체에 추가합니다.

```
for i in range(1, 6):
    item = QTableWidgetItem(self.kiwoom.opw00018_output['single'][i - 1])
    item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)
    self.tableWidget.setItem(0, i, item)
```

QTableWidget 객체에 데이터를 추가했다면 마지막으로 resizeRowsToContents 메서드를 호출해서 아이템의 크기에 맞춰 행의 높이를 조절합니다.

```
self.tableWidget.resizeRowsToContents()
```

이번에는 보유 종목별 평가 잔고 데이터를 QTableWidget에 추가해 보겠습니다. 앞서 Qt Designer를 통해 UI를 구성할 때 tableWidget_2 객체에는 행의 개수를 따로 설정하지 않았습니다. 따라서 먼저 보유종목의 개수를 확인한 후 행의 개수를 설정해야 합니다.

```
# Item list
item_count = len(self.kiwoom.opw00018_output['multi'])
self.tableWidget_2.setRowCount(item_count)
```

행과 열의 개수를 설정했다면 (열의 개수는 Qt Designer가 자동으로 설정했음) 적당한 위치에 아이템을 추가하면 됩니다. 한 종목에 대한 종목명, 보유량, 매입가, 현재가, 평가손익, 수익률(%)은 한 행에 출력됩니다.

```
for j in range(item_count):
    row = self.kiwoom.opw00018_output['multi'][j]
    for i in range(len(row)):
        item = QTableWidgetItem(row[i])
        item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)
        self.tableWidget_2.setItem(j, i, item)
```

마지막으로 행의 크기를 조절하는 코드를 추가합니다.

```
self.tableWidget_2.resizeRowsToContents()
```

지금까지 설명한 check_balance 메서드의 전체 코드는 예제 18.5와 같습니다.

예제 18.5 check_balance 메서드

```
def check_balance(self):
    self.kiwoom.reset_opw00018_output()
    account_number = self.kiwoom.get_login_info("ACCNO")
    account_number = account_number.split(';')[0]

    self.kiwoom.set_input_value("계좌번호", account_number)
    self.kiwoom.comm_rq_data("opw00018_req", "opw00018", 0, "2000")

    while self.kiwoom.remained_data:
        time.sleep(0.2)
        self.kiwoom.set_input_value("계좌번호", account_number)
        self.kiwoom.comm_rq_data("opw00018_req", "opw00018", 2, "2000")

    # opw00001
```

```
        self.kiwoom.set_input_value("계좌번호", account_number)
        self.kiwoom.comm_rq_data("opw00001_req", "opw00001", 0, "2000")

        # balance
        item = QTableWidgetItem(self.kiwoom.d2_deposit)
        item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)
        self.tableWidget.setItem(0, 0, item)

        for i in range(1, 6):
            item = QTableWidgetItem(self.kiwoom.opw00018_output['single'][i - 1])
            item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)
            self.tableWidget.setItem(0, i, item)

        self.tableWidget.resizeRowsToContents()

        # Item list
        item_count = len(self.kiwoom.opw00018_output['multi'])
        self.tableWidget_2.setRowCount(item_count)

        for j in range(item_count):
            row = self.kiwoom.opw00018_output['multi'][j]
            for i in range(len(row)):
                item = QTableWidgetItem(row[i])
                item.setTextAlignment(Qt.AlignVCenter | Qt.AlignRight)
                self.tableWidget_2.setItem(j, i, item)

        self.tableWidget_2.resizeRowsToContents()
```

잔고 및 보유종목 현황은 사용자가 [조회] 버튼을 클릭할 때 데이터를 얻어온 후 이를 출력하는 방식으로 구현했습니다. 즉, 실시간으로 데이터가 변경되는 것이 아니라 사용자가 [조회] 버튼을 누를 때만 데이터가 갱신됩니다.

이번에는 [실시간 조회] 체크박스를 체크하면 10초에 한 번씩 데이터가 자동으로 갱신되는 기능을 구현해 보겠습니다. 10초에 한 번씩 특정 이벤트가 발생하게 만들려면 앞서 사용해본 타이머를 사용하면 됩니다. MyWindow 클래스의 생성자에 다음과 같이 새로운 타이머 객체를 생성합니다. timeout2 메서드는 10초에 한 번씩 호출됩니다.

```
        # Timer2
        self.timer2 = QTimer(self)
```

```
self.timer2.start(1000*10)
self.timer2.timeout.connect(self.timeout2)
```

이번에는 timeout2 메서드를 구현해 봅시다. 해당 메서드에서는 QCheckBox가 체크됐는지 확인한 후 체크돼 있을 때 데이터를 갱신하면 됩니다. 앞서 데이터 갱신 처리는 check_balance 메서드가 담당하고 있으므로 다음과 같이 간단히 구현할 수 있습니다.

```
def timeout2(self):
    if self.checkBox.isChecked():
        self.check_balance()
```

이로써 개발 3일 차의 모든 코드 작성을 끝마쳤습니다. pytrader.py 파일을 실행해 보면 그림 18.63과 같이 정상적으로 잔고 및 보유종목 현황이 출력되는 모습을 확인할 수 있습니다. 참고로 현재 실시간 조회의 출력 주기는 10초지만 때에 따라 더 짧은 시간을 지정할 수도 있습니다. 다만 이 경우 TR의 요청 주기도 짧아지기 때문에 다른 TR의 요청이 많은 경우에는 주의해야 합니다.

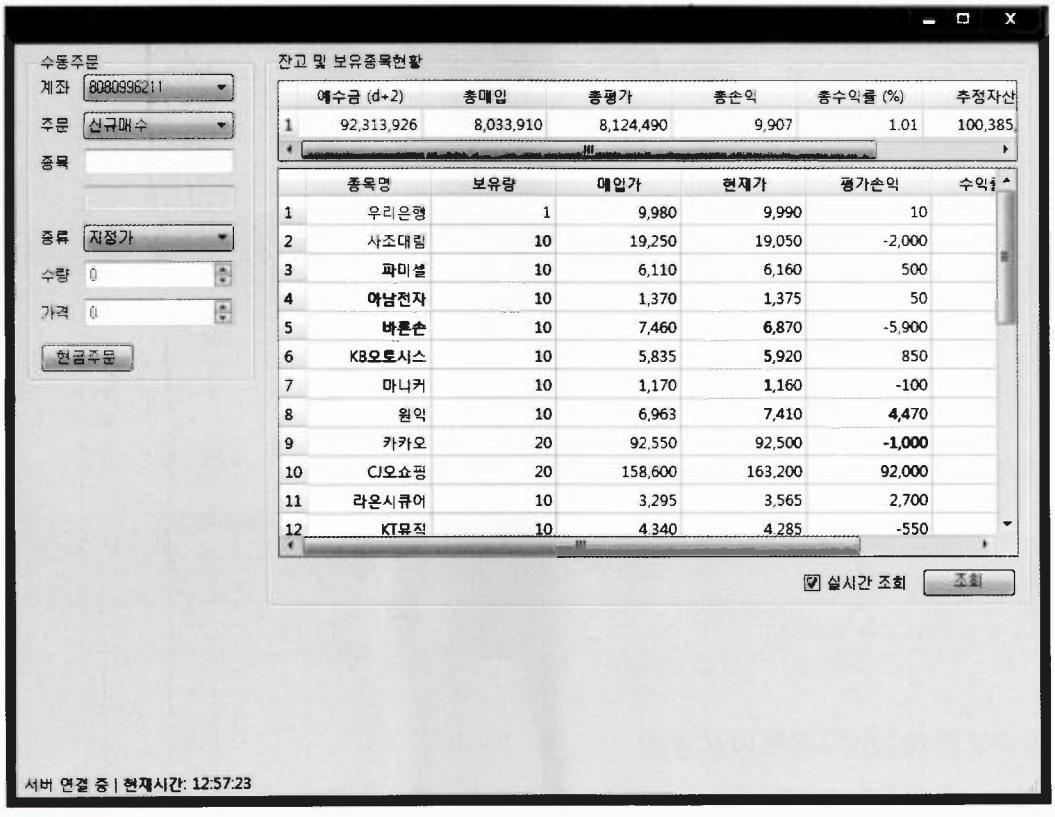


그림 18.63 PyTrader v0.3 실행 화면

04 개발 4일 차

개발 4일 차에는 프로그램에 의해 선정된 매수/매도 종목을 자동으로 매매하고 해당 종목에 대한 정보를 QTableWidget에 출력하는 기능을 구현하겠습니다. 선정된 종목 정보를 출력하는 QTableWidget은 그림 18.64와 같이 UI의 하단부에 위치합니다.

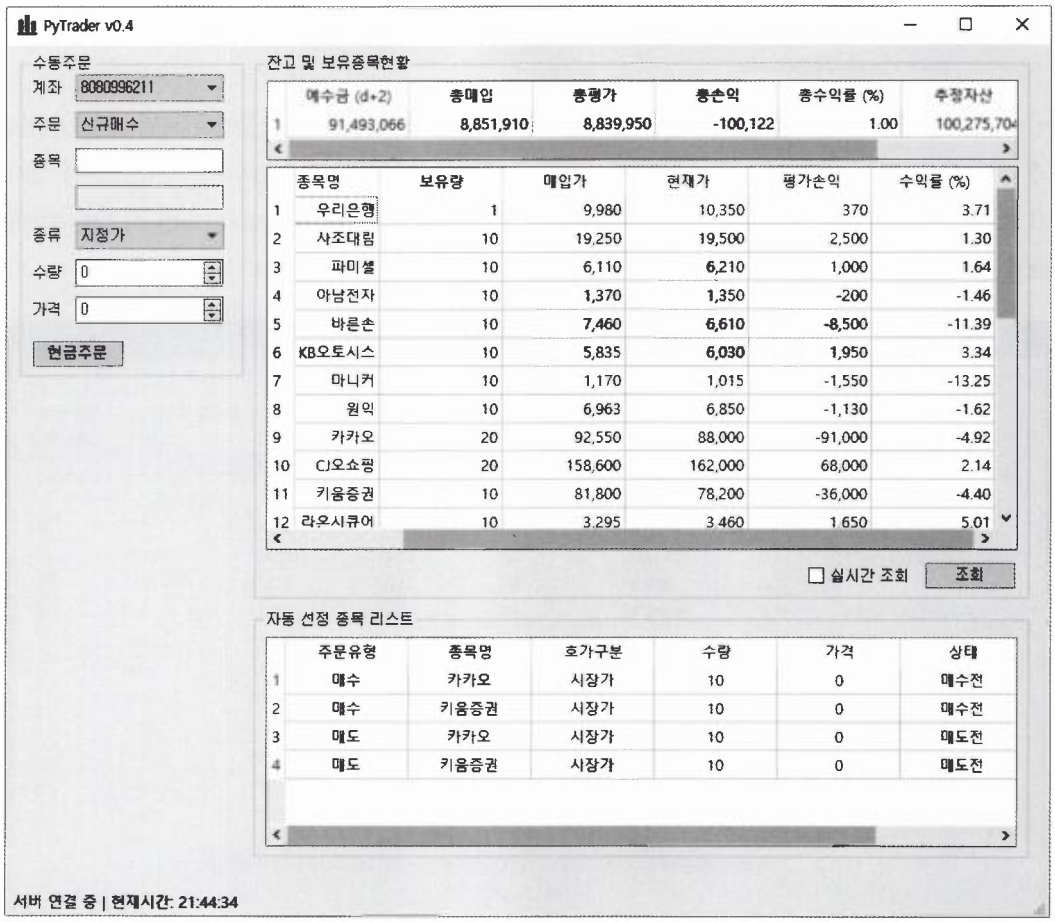


그림 18.64 개발 4일 차 실행 화면

UI 구성 및 매수/매도 목록 파일 생성

먼저 UI부터 구성하겠습니다. Qt Designer를 실행한 후 3일 차까지 작성된 pytrader.ui 파일을 엽니다. 윈도우 타이틀을 PyTrader v0.4로 변경하고 Table Widget과 Group Box를 추가하기 위해 QMainWindow 크기를 적당히 조절합니다.

title이 '자동 선정 종목 리스트'인 Group Box를 추가한 후 해당 Group Box 안으로 Table Widget을 추가합니다. Table Widget의 칼럼에는 그림 18.65와 같이 이름이 각각 '주문유형', '종목명', '호가구분', '수량', '가격', '상태'인 칼럼을 6개 추가합니다.

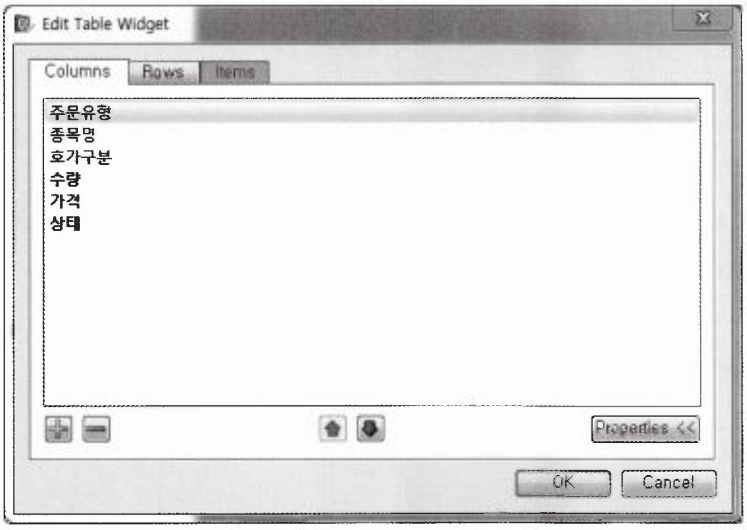


그림 18.65 칼럼 추가

앞서 여러 번 설명했듯이 Table Widget을 사용할 때는 rowCount와 columnCount라는 속성의 값을 잘 설정해야 합니다. columnCount는 그림 18.65와 같이 칼럼을 추가할 때 칼럼의 개수에 맞춰 자동으로 설정되지만 rowCount는 여러분이 직접 설정해야 합니다. 그러나 UI를 구성할 때는 행 단위로 추가되는 데이터의 개수를 알 수 없는 경우가 많은데 이때는 파이썬 코드를 통해 설정해야 합니다.

PyTrader v0.4는 미리 선정된 매수/매도 종목에 대한 정보를 파일로부터 얻습니다. 알고리즘을 기반으로 매수/매도 종목을 선정하는 것은 다음에 알아보기로 하고 4일 차에서는 선정된 매수/매도 종목에 대한 정보가 이미 buy_list.txt라는 파일과 sell_list.txt라는 파일로 저장돼 있다고 가정하고 프로그램을 구현하겠습니다.

이를 위해 먼저 윈도우에서 메모장을 실행한 후 그림 18.66과 같이 매수 종목에 대한 기본 정보를 입력합니다. 여기서 각 종목에 대한 데이터는 세미콜론(;)을 통해 구분하면 되고 종목명 대신 종목코드를 사용합니다. 입력이 끝나면 buy_list.txt라는 이름으로 저장합니다. 참고로 buy_list.txt 파일은 pytrader.py나 pytrader.ui 파일과 같은 디렉터리에 위치해야 합니다.

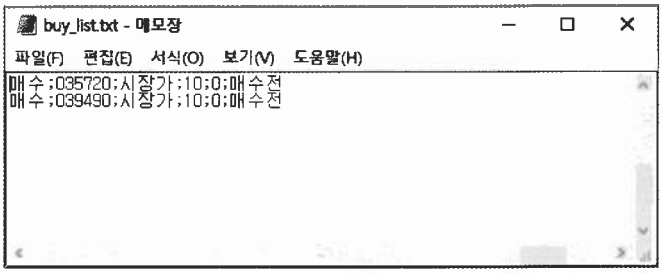


그림 18.66 선정된 매수 종목 정보

매도 종목에 대한 정보는 그림 18.67과 같이 sell_list.txt라는 파일 이름으로 저장합니다. 역시 종목에 대한 세부 데이터는 세미콜론(;)을 통해 구분되도록 데이터를 구성했습니다.

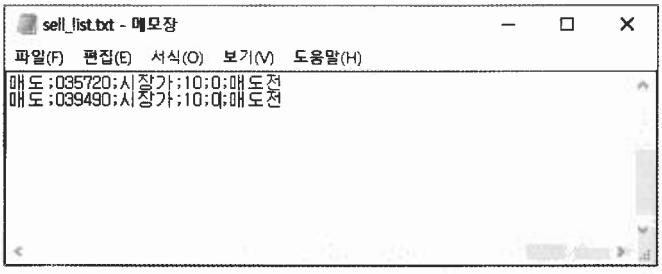


그림 18.67 선정된 매도 종목 정보

선정 종목 정보 출력하기

이번 절에서는 앞서 테스트를 위해 생성했던 buy_list.txt와 sell_list.txt 파일을 읽어서 이를 QTableWidget 객체로 출력하는 기능을 구현하겠습니다. 파일로부터 매매 종목에 대한 정보를 읽고 이를 QTableWidget 객체에 출력하는 기능은 load_buy_sell_list라는 이름의 메서드로 구현하겠습니다. 선정 종목 리스트는 PyTrader 프로그램이 시작되자마자 출력돼야 하므로 MyWindow 클래스의 생성자에서 load_buy_sell_list를 호출해야 합니다.

```
class MyWindow(QMainWindow, form_class):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

        # 중간 코드 생략
        self.load_buy_sell_list()
```

MyWindow 클래스에 load_buy_sell_list 메서드를 추가한 후 buy_list.txt와 sell_list.txt 파일을 열고 파일로부터 데이터를 읽는 코드를 구현합니다.

```
def load_buy_sell_list(self):
    f = open("buy_list.txt", 'rt')
    buy_list = f.readlines()
    f.close()

    f = open("sell_list.txt", 'rt')
    sell_list = f.readlines()
    f.close()
```

파일로부터 매수/매도 리스트를 읽었다면 먼저 데이터의 총 개수를 확인합니다. 매수/매도 종목 각각에 대한 데이터 개수를 확인한 후 이 두 값을 더한 값을 QTableWidget 객체의 setRowCount 메서드로 설정합니다.

```
row_count = len(buy_list) + len(sell_list)
self.tableWidget_4.setRowCount(row_count)
```

먼저 매수 종목부터 QTableWidget 객체에 추가하겠습니다. 다음 코드에서 j는 행(row)에 대한 인덱스 값이며, i는 열(column)에 대한 인덱스 값입니다. 기본적으로 파일에서 한 줄씩 읽어온 문자열을 ';'에 대해 분리한 후 데이터를 각 칼럼에 추가합니다.

칼럼에 추가되는 데이터 중 종목명은 종목코드로부터 구해야 합니다. 종목코드는 split_row_data 리스트의 1번째에 저장돼 있으므로 이 값을 이용해 종목코드를 종목명으로 변경합니다. 종목코드로부터 종목명을 구하려면 Kiwoom 클래스의 get_master_code_name 메서드를 이용하면 됩니다.

```
# buy list
for j in range(len(buy_list)):
    row_data = buy_list[j]
    split_row_data = row_data.split(';')
    split_row_data[1] = self.kiwoom.get_master_code_name(split_row_data[1].rsplit())

    for i in range(len(split_row_data)):
        item = QTableWidgetItem(split_row_data[i].rstrip())
        item.setTextAlignment(Qt.AlignVCenter | Qt.AlignCenter)
        self.tableWidget_4.setItem(j, i, item)
```

매도 종목에 대해서도 동일한 방식으로 데이터를 QTableWidget 객체에 추가합니다.

```
# sell list
for j in range(len(sell_list)):
    row_data = sell_list[j]
    split_row_data = row_data.split(';')
    split_row_data[1] = self.kiwoom.get_master_code_name(split_row_data[1].rstrip())

    for i in range(len(split_row_data)):
        item = QTableWidgetItem(split_row_data[i].rstrip())
        item.setTextAlignment(Qt.AlignVCenter | Qt.AlignCenter)
        self.tableWidget_4.setItem(len(buy_list) + j, i, item)
```

QTableWidget에서 행의 크기를 조절하기 위해 resizeRowsToContents 메서드를 호출합니다.

```
self.tableWidget_4.resizeRowsToContents()
```

예제 18.6은 지금까지 구현한 load_buy_sell_list 메서드의 전체 코드입니다. 코드의 전체적인 흐름을 다시 한 번 살펴보기 바랍니다.

예제 18.6 load_buy_sell_list 메서드

```
def load_buy_sell_list(self):
    f = open("buy_list.txt", 'rt')
    buy_list = f.readlines()
    f.close()

    f = open("sell_list.txt", 'rt')
    sell_list = f.readlines()
    f.close()

    row_count = len(buy_list) + len(sell_list)
    self.tableWidget_4.setRowCount(row_count)

    # buy list
    for j in range(len(buy_list)):
        row_data = buy_list[j]
        split_row_data = row_data.split(';')
        split_row_data[1] = self.kiwoom.get_master_code_name(split_row_data[1].rsplit())

        for i in range(len(split_row_data)):
            item = QTableWidgetItem(split_row_data[i].rstrip())
            item.setTextAlignment(Qt.AlignVCenter | Qt.AlignCenter)
            self.tableWidget_4.setItem(j, i, item)
```

```
    # sell list
    for j in range(len(sell_list)):
        row_data = sell_list[j]
        split_row_data = row_data.split(';')
        split_row_data[1] = self.kiwoom.get_master_code_name(split_row_data[1].rstrip())

        for i in range(len(split_row_data)):
            item = QTableWidgetItem(split_row_data[i].rstrip())
            item.setTextAlignment(Qt.AlignVCenter | Qt.AlignCenter)
            self.tableWidget_4.setItem(len(buy_list) + j, i, item)

    self.tableWidget_4.resizeRowsToContents()
```

자동 주문 구현하기

이번 절에서는 장이 시작하면 미리 선정된 종목에 대해 자동으로 주문하는 기능을 구현해 보겠습니다. 사실 자동으로 주문하는 기능은 어떻게 구현하느냐에 따라 매우 복잡해질 수도 있고 간단할 수도 있습니다. PyTrader는 각 거래일의 장 시작에 맞춰 정해진 주문 방식에 따라 주문을 수행하는 간단한 방식을 사용하겠습니다.

자동으로 주문하는 기능은 MyWindow 클래스의 trade_stocks 메서드에 구현하겠습니다. 먼저 미리 생성된 파일로부터 매수/매도 종목을 읽는 코드를 작성합니다. 이 부분은 앞서 load_buy_sell_list 메서드에서 구현한 것과 동일합니다.

```
def trade_stocks(self):
    hoga_lookup = {'지정가': "00", '시장가': "03"}

    f = open("buy_list.txt", 'rt')
    buy_list = f.readlines()
    f.close()

    f = open("sell_list.txt", 'rt')
    sell_list = f.readlines()
    f.close()
```

주문할 때 필요한 계좌 정보를 QComboBox 위젯으로부터 얻어옵니다.

```
account = self.comboBox.currentText()
```

buy_list로부터 데이터를 하나씩 얻어온 후 문자열을 분리해서 주문에 필요한 정보(거래구분, 종목코드, 수량, 가격)를 준비합니다. 읽어온 데이터의 주문 수행 여부가 '매수전'인 경우에만 해당 주문 데이터를 토대로 send_order 메서드를 통해 매수 주문을 수행합니다.

```
# buy list
for row_data in buy_list:
    split_row_data = row_data.split(';')
    hoga = split_row_data[2]
    code = split_row_data[1]
    num = split_row_data[3]
    price = split_row_data[4]

    if split_row_data[-1].rstrip() == '매수전':
        self.kiwoom.send_order("send_order_req", "0101", account, 1, code, num, price, hoga_
lookup[hoga], "")
```

매도 주문 역시 동일한 방식으로 처리하면 됩니다.

```
# sell list
for row_data in sell_list:
    split_row_data = row_data.split(';')
    hoga = split_row_data[2]
    code = split_row_data[1]
    num = split_row_data[3]
    price = split_row_data[4]

    if split_row_data[-1].rstrip() == '매도전':
        self.kiwoom.send_order("send_order_req", "0101", account, 2, code, num, price, hoga_
lookup[hoga], "")
```

매매 주문이 완료되면 buy_list.txt나 sell_list.txt에 저장된 주문 여부를 업데이트합니다. 앞서 매매 주문을 실행했기 때문에 '매수전'과 '매도전'을 '주문완료'로 변경하겠습니다. 데이터를 변경한 후에는 변경된 데이터를 다시 파일에 덮어써서 파일을 업데이트합니다.

```
# buy list
for i, row_data in enumerate(buy_list):
    buy_list[i] = buy_list[i].replace("매수전", "주문완료")

# file update
f = open("buy_list.txt", 'wt')
for row_data in buy_list:
```

```
    f.write(row_data)
f.close()
```

매도 리스트 파일 역시 동일한 방식으로 처리합니다.

```
# sell list
for i, row_data in enumerate(sell_list):
     sell_list[i] = sell_list[i].replace("매도전", "주문완료")

# file update
f = open("sell_list.txt", 'wt')
for row_data in sell_list:
    f.write(row_data)
f.close()
```

지금까지 구현한 trade_stocks 메서드의 전체 코드는 예제 18.7과 같습니다.

예제 18.7 trade_stocks 메서드

```
def trade_stocks(self):
    hoga_lookup = {'지정가': "00", '시장가': "03"}

    f = open("buy_list.txt", 'rt')
    buy_list = f.readlines()
    f.close()

    f = open("sell_list.txt", 'rt')
    sell_list = f.readlines()
    f.close()

    # account
    account = self.comboBox.currentText()

    # buy list
    for row_data in buy_list:
        split_row_data = row_data.split(';')
        hoga = split_row_data[2]
        code = split_row_data[1]
        num = split_row_data[3]
        price = split_row_data[4]

        if split_row_data[-1].rstrip() == '매수전':
```

```
            self.kiwoom.send_order("send_order_req", "0101", account, 1, code, num, price,
hoga_lookup[hoga], "")

        # sell list
        for row_data in sell_list:
            split_row_data = row_data.split(';')
            hoga = split_row_data[2]
            code = split_row_data[1]
            num = split_row_data[3]
            price = split_row_data[4]

            if split_row_data[-1].rstrip() == '매도전':
                self.kiwoom.send_order("send_order_req", "0101", account, 2, code, num, price,
hoga_lookup[hoga], "")

        # buy list
        for i, row_data in enumerate(buy_list):
            buy_list[i] = buy_list[i].replace("매수전", "주문완료")

        # file update
        f = open("buy_list.txt", 'wt')
        for row_data in buy_list:
            f.write(row_data)
        f.close()

        # sell list
        for i, row_data in enumerate(sell_list):
            sell_list[i] = sell_list[i].replace("매도전", "주문완료")

        # file update
        f = open("sell_list.txt", 'wt')
        for row_data in sell_list:
            f.write(row_data)
        f.close()
```

장이 시작할 때 매수/매도 주문을 넣으려면 timeout 메서드에서 시간을 체크하면 됩니다. timeout 메서드를 다음과 같이 수정합니다. timeout 메서드에서는 현재 시간이 09시 00분 00초를 지났고 매수/매도 주문을 수행하지 않았을 때 trade_stocks 메서드를 호출합니다.

```
def timeout(self):
    market_start_time = QTime(9, 0, 0)
```

```
        current_time = QTime.currentTime()

        if current_time > market_start_time and self.trade_stocks_done is False:
            self.trade_stocks()
            self.trade_stocks_done = True

        text_time = current_time.toString("hh:mm:ss")
        time_msg = "현재시간: " + text_time

        state = self.kiwoom.get_connect_state()
        if state == 1:
            state_msg = "서버 연결 중"
        else:
            state_msg = "서버 미 연결 중"

        self.statusbar.showMessage(state_msg + " | " + time_msg)
```

MyWindow 클래스의 생성자에 trade_stocks_done 속성을 추가합니다. 참고로 trade_stocks_done은 생성자에서 QTimer 객체를 생성하는 코드보다 먼저 위치해야 합니다.

```
class MyWindow(QMainWindow, form_class):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

        self.trade_stocks_done = False
```

19

실전 프로그램 개발(2)

이번 장에서는 매수/매도 종목을 자동으로 선정하는 PyMon(Python Monitor)을 개발하겠습니다. PyMon은 매일 장 종료 후 자동으로 실행되어 유가증권 시장과 코스닥 시장에서 미리 구현된 매수/매도 알고리즘에 부합하는 종목을 찾은 후 이를 `buy_list.txt`와 `sell_list.txt`로 저장합니다.

이번 장에서 몇 가지 간단한 매수 알고리즘을 소개하고 이를 직접 구현해 보면서 투자 알고리즘을 어떻게 파이썬 코드로 구현할 수 있는지에 대해 좀 더 자세히 알아보겠습니다.

01 개발 5일 차

PyMon은 PyTrader와 달리 CUI 프로그램입니다. 자동으로 종목을 선정하는 것이 주 역할이기 때문에 굳이 GUI 형태로 구현하지 않았습니다. 그 대신 알고리즘을 어떻게 파이썬 코드를 통해 구현할 것인지에 좀 더 집중하겠습니다.

5일 차에서 개발할 내용은 『주식투자 베스트비법』(청출판, 모닝퍼스 저)에서 소개한 급등주 포착 알고리즘 기반의 종목 선정입니다. PyMon이 실행되면 그림 19.1과 같이 조건과 일치하는 종목에 대한 매수 정보를 buy_list.txt 파일로 출력합니다.

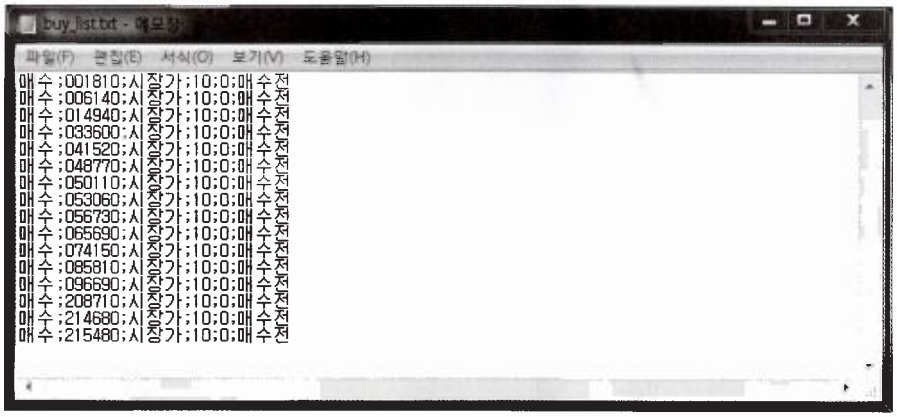


그림 19.1 급등주 포착 알고리즘을 기반으로 선정된 매수 종목 리스트

먼저 pymon.py 파일을 생성한 후 다음과 같이 기본적인 PyMon 클래스를 구현합니다. 이처럼 프로그램을 구현할 때는 먼저 기본적인 골격부터 차근차근 구현해 나가는 것이 중요합니다.

```
import sys
from PyQt5.QtWidgets import *
import Kiwoom

class PyMon:
    def __init__(self):
        self.kiwoom = Kiwoom.Kiwoom()
        self.kiwoom.comm_connect()

    def run(self):
        print("run")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    pymon = PyMon()
    pymon.run()
```

PyMon에서도 키움증권의 API를 사용하기 위해 Kiwoom.py 파일을 사용합니다. 4일 차에서 사용한 Kiwoom.py 파일을 복사해서 pymon.py 파일과 동일한 폴더에 저장합니다. 또는 다음 링크를 통해 Kiwoom.py 파일을 새로 내려받아도 됩니다.

- https://github.com/pystockhub/book/tree/master/ch19/day05

PyCharm에서 pymon.py 파일을 실행하면 그림 19.2와 같이 키움증권에 자동으로 로그인된 후 PyMon 클래스의 run 메서드가 정상적으로 호출되는 것을 확인할 수 있습니다.



그림 19.2 PyMon 실행 결과

유가증권/코스닥 시장 종목코드 가져오기

알고리즘에 부합하는 종목을 선정하려면 먼저 유가증권과 코스닥 시장의 모든 종목에 대한 종목코드를 알고 있어야 합니다. 따라서 PyMon에서 먼저 유가증권과 코스닥 시장에 상장된 종목의 코드를 가져오는 메서드부터 구현해 보겠습니다.

PyMon의 생성자에서 Kiwoom 클래스의 인스턴스를 생성했고, 이를 self.kiwoom이 바인딩하고 있습니다. 따라서 self.kiwoom을 통해 get_code_list_by_market 메서드를 호출함으로써 유가증권시장과 코스닥시장의 종목코드 리스트를 가져올 수 있습니다. 가져온 종목코드 리스트는 get_code_list 메서드뿐 아니라 다른 메서드에서 사용될 것이므로 self.kospi_codes와 self.kosdaq_codes라는 이름으로 각각 바인딩합니다.

```
def get_code_list(self):
    self.kospi_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSPI)
    self.kosdaq_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSDAQ)
```

유가증권과 코스닥 시장의 종목코드를 가져오는 것은 PyMon이 실행될 때 한 번만 수행되면 되기 때문에 PyMon의 생성자에서 get_code_list 메서드를 호출합니다. 정상적으로 종목코드를 가져왔는지 확인하기 위해 run 메서드에서 self.kospi_codes와 self.kosdaq_codes 리스트에 저장된 종목코드 중 일부를 화면에 출력하도록 변경합니다.

예제 19.1 get_code_list 메서드 추가(소스코드: book/ch19/day05/01.py)

```
import sys
from PyQt5.QtWidgets import *
import Kiwoom

MARKET_KOSPI   = 0
MARKET_KOSDAQ  = 10

class PyMon:
    def __init__(self):
        self.kiwoom = Kiwoom.Kiwoom()
        self.kiwoom.comm_connect()
        self.get_code_list()

    def get_code_list(self):
        self.kospi_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSPI)
        self.kosdaq_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSDAQ)

    def run(self):
        print(self.kospi_codes[0:4])
        print(self.kosdaq_codes[0:4])

if __name__ == "__main__":
    app = QApplication(sys.argv)
    pymon = PyMon()
    pymon.run()
```

예제 19.1을 실행하면 그림 19.3과 같이 유가증권시장과 코스닥시장의 일부 종목에 대한 종목코드가 화면에 출력되는 모습을 확인할 수 있습니다.



그림 19.3 유가증권시장과 코스닥시장의 종목코드를 출력한 결과

일별 데이터 가져오기

이번에는 오늘 날짜를 시작으로 과거 거래일별로 시가, 고가, 저가, 종가, 거래량을 가져오는 메서드를 구현해 보겠습니다. 구현할 메서드의 이름은 get_ohlcv이고, 메서드의 인자로 조회할 종목에 대한 종목코드와 기준일자를 받습니다. 앞서 17.3절에서 설명했던 내용을 참조해서 다음과 같이 코드를 작성했습니다.

```
def get_ohlcv(self, code, start):
    self.kiwoom.ohlcv = {'date': [], 'open': [], 'high': [], 'low': [], 'close': [], 'volume':
[]}

    self.kiwoom.set_input_value("종목코드", code)
    self.kiwoom.set_input_value("기준일자", start)
    self.kiwoom.set_input_value("수정주가구분", 1)
    self.kiwoom.comm_rq_data("opt10081_req", "opt10081", 0, "0101")
    time.sleep(0.2)

    df = DataFrame(self.kiwoom.ohlcv, columns=['open', 'high', 'low', 'close', 'volume'],
                   index=self.kiwoom.ohlcv['date'])
    return df
```

TR을 요청할 때 0.2초 간격을 주기 위해 time 모듈을 사용했고 조회된 시가, 고가, 저자, 종가, 거래량을 효율적으로 저장하기 위해 pandas 모듈을 사용했습니다. 따라서 다음과 같이 사용할 모듈을 임포트해야 합니다.

```
import time
from pandas import DataFrame
```

구현한 get_ohlcv 메서드가 정상적으로 동작하는지 테스트하기 위해 run 메서드를 다음과 같이 수정합니다. get_ohlcv의 첫 번째 인자로 키움증권의 종목코드인 '039490'을 입력했고 기준일자로 '20170321'을 입력했습니다.

```
def run(self):
    df = self.get_ohlcv("039490", "20170321")
    print(df)
```

지금까지 작성한 코드는 예제 19.2와 같습니다. 전체적인 코드의 흐름을 파악해 보시기 바랍니다.

예제 19.2 PyMon 클래스에 get_ohlcv 메서드 추가(소스코드: book/ch19/day05/02.py)

```
import sys
from PyQt5.QtWidgets import *
import Kiwoom
import time
from pandas import DataFrame

MARKET_KOSPI   = 0
MARKET_KOSDAQ  = 10

class PyMon:
    def __init__(self):
        self.kiwoom = Kiwoom.Kiwoom()
        self.kiwoom.comm_connect()
        self.get_code_list()

    def get_code_list(self):
        self.kospi_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSPI)
        self.kosdaq_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSDAQ)

    def get_ohlcv(self, code, start):
        self.kiwoom.ohlcv = {'date': [], 'open': [], 'high': [], 'low': [], 'close': [], 'vol-
ume': []}

        self.kiwoom.set_input_value("종목코드", code)
        self.kiwoom.set_input_value("기준일자", start)
        self.kiwoom.set_input_value("수정주가구분", 1)
        self.kiwoom.comm_rq_data("opt10081_req", "opt10081", 0, "0101")
        time.sleep(0.2)

        df = DataFrame(self.kiwoom.ohlcv, columns=['open', 'high', 'low', 'close', 'volume'],
                       index=self.kiwoom.ohlcv['date'])
        return df

    def run(self):
        df = self.get_ohlcv("039490", "20170321")
        print(df)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    pymon = PyMon()
    pymon.run()
```

예제 19.2를 실행해보면 그림 19.4와 같이 키움증권 종목에 대해 2017/03/21일을 시작으로 과거 거래일에 대한 시가, 고가, 저가, 종가, 거래량 값이 출력되는 것을 확인할 수 있습니다. 참고로 comm_rq_data 메서드를 통해 TR을 한 번 요청하면 과거 900일에 대한 데이터를 가져올 수 있습니다.

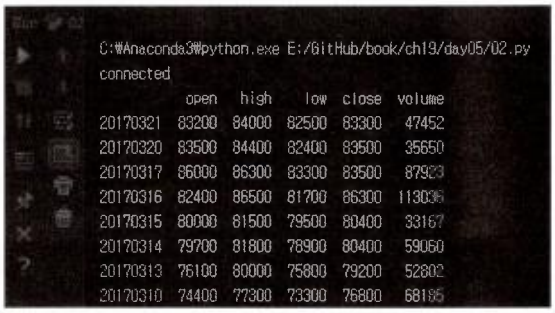


그림 19.4 일자별 시가, 고가, 저가, 종가, 거래량 출력(키움증권)

급등주 포착 알고리즘 구현

알고리즘 구현에 필요한 데이터가 모두 준비됐으니 이번에는 알고리즘을 구현해 보겠습니다. '급등주 포착' 알고리즘은 특정 거래일의 거래량이 이전 시점의 평균 거래량보다 1,000% 이상 급증하는 종목을 매수하는 것입니다.

사실 기존의 주식 관련 책에서 소개하는 기법은 알고리즘 구현 관점에서 조금 불분명한 부분들이 많습니다. 이 책에서는 일단 '이전 시점의 평균 거래량'을 특정 거래일 이전의 20일(거래일 기준) 동안의 평균 거래량으로 정의했습니다. 그리고 '거래량 급증'은 특정 거래일의 거래량이 평균 거래량보다 1,000% 초과일 때 급등한 것으로 정의했습니다.

알고리즘을 구현에 맞춰 상세화했다면 이를 파이썬 코드로 구현하면 됩니다. 급등주 포착을 체크하는 기능이기 때문에 메서드 이름을 check_speedy_rising_volume으로 지정했고 메서드의 인자로 종목코드를 전달받게 했습니다. 코드는 다음과 같습니다.

```
def check_speedy_rising_volume(self, code):
    today = datetime.datetime.today().strftime("%Y%m%d")
    df = self.get_ohlcv(code, today)
    volumes = df['volume']

    if len(volumes) < 21:
        return False
```

```
        sum_vol20 = 0
        today_vol = 0

        for i, vol in enumerate(volumes):
            if i == 0:
                today_vol = vol
            elif 1 <= i <= 20:
                sum_vol20 += vol
            else:
                break

        avg_vol20 = sum_vol20 / 20
        if today_vol > avg_vol20 * 10:
            return True
```

위 코드를 조금씩 나눠서 분석해 보겠습니다. 가장 먼저 오늘 날짜를 가져왔습니다. 파이썬 코드에서 오늘 날짜를 '20170321'과 같은 포맷으로 얻기 위해 datetime 모듈을 사용했습니다.

```
today = datetime.datetime.today().strftime("%Y%m%d")
```

종목코드와 조회 시작일이 준비됐으므로 앞서 구현한 get_ohlcv 메서드를 호출해서 해당 종목에 대한 시가, 고가, 저가, 종가, 거래량 데이터를 DataFrame 객체로 얻어온 후 그중 거래량 칼럼만 바인딩했습니다.

```
df = self.get_ohlcv(code, today)
volumes = df['volume']
```

조회 시작일을 기준으로 이전 20일간 거래량의 평균을 계산해야 하므로 총 21일 치의 데이터가 필요합니다. 만약 최근에 상장된 종목이라면 데이터가 충분하지 않을 수 있으므로 이 경우에는 if 문을 사용해 걸러냈습니다. check_speedy_rising_volume 메서드는 해당 종목이 거래량 기준 급등주인 경우 True를 반환하고 급등주가 아닌 경우에는 False를 반환합니다. 따라서 데이터가 충분하지 않은 종목에 대해서는 False를 반환합니다.

```
if len(volumes) < 21:
    return False
```

조회 시작일의 거래량은 today_vol이라는 변수로 저장하고 조회 시작일을 기준으로 이전 20일의 거래량은 평균을 계산하기 위해 먼저 sum_vol20을 통해 일별 거래량을 누적했습니다. 구현하고 있는 알고리즘은 21일 치 데이터만 필요하므로 for 문에서 break를 통해 반복문을 탈출하도록 구현했습니다. 이를 통해 쓸데없는 데이터에 대해 반복문을 수행하지 않게 할 수 있습니다.

```
sum_vol20 = 0
today_vol = 0

for i, vol in enumerate(volumes):
    if i == 0:
        today_vol = vol
    elif 1 <= i <= 20:
        sum_vol20 += vol
    else:
        break
```

마지막으로 20일의 평균 거래량을 계산한 후 조회 시작일의 거래량과 비교합니다. 만약 조회 시작일의 거래량이 평균 거래량을 1,000% 초과한다면 True를 반환하도록 구현합니다.

```
avg_vol20 = sum_vol20 / 20
if today_vol > avg_vol20 * 10:
    return True
```

구현한 check_speedy_rising_volume 메서드를 테스트하기 위해 run 메서드를 다음과 같이 수정합니다. 코스닥 시장에 대해 급등주 포착 알고리즘에 부합하는 종목이 있는지 확인하고 만약 있다면 이를 화면에 출력하게 했습니다.

```
def run(self):
    num = len(self.kosdaq_codes)

    for i, code in enumerate(self.kosdaq_codes):
        print(i, '/', num)
        if self.check_speedy_rising_volume(code):
            print("급등주: ", code)
```

프로그램을 실행해 보기에 앞서 datetime 모듈을 임포트합니다.

```
import datetime
```

프로그램을 실행해 보면 그림 19.5와 같이 코스닥 종목 중 급등주 포착 알고리즘에 부합하는 종목에 대한 종목코드가 화면에 출력되는 모습을 확인할 수 있습니다.

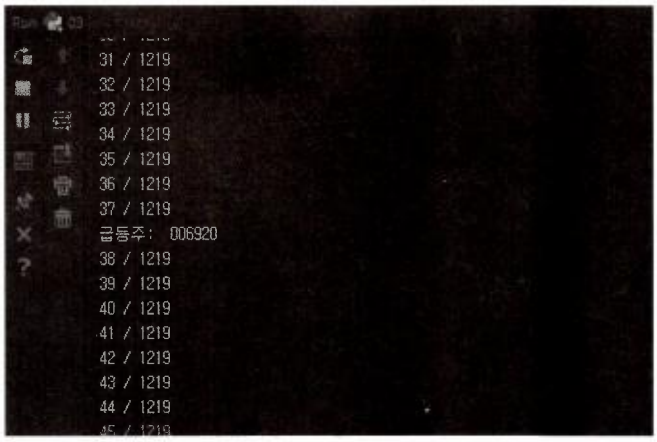


그림 19.5 거래량 급등주를 포착한 결과

마지막으로 선정된 종목에 대한 정보를 파일로 쓰는 기능을 추가하겠습니다. 해당 기능은 update_buy_list라는 이름의 메서드로 구현하겠습니다. 메서드의 인자로 매수할 종목에 대한 종목코드 리스트를 받겠습니다.

다음 코드와 같이 buy_list.txt 파일을 'wt' 모드로 연 후, 메서드 인자로 전달된 매수 종목에 대해 파일에 라인 단위로 출력하면 됩니다. 참고로 매수 수량의 경우 현재 예에서는 예시를 단순화하기 위해 10주씩으로 설정했지만 이런 부분들은 나중에 수정하면 됩니다.

```
def update_buy_list(self, buy_list):
    f = open("buy_list.txt", "wt")
    for code in buy_list:
        f.writelines("매수;%s;시장가;10;0;매수전\n" % (code))
    f.close()
```

매수 종목을 파일에 출력하는 기능이 추가됐으므로 run 메서드도 이에 맞춰 수정합니다. 먼저 buy_list라는 빈 리스트를 생성한 후 check_speedy_rising_volume 메서드의 반환값이 True인 종목의 종목코드를 해당 리스트에 추가합니다. 코스닥 시장의 모든 종목에 대해 거래량 급증을 확인했다면 update_buy_list 메서드를 호출해 거래량 급증 종목을 파일로 출력되게 했습니다.

```
def run(self):
    buy_list = []
    num = len(self.kosdaq_codes)

    for i, code in enumerate(self.kosdaq_codes):
        print(i, '/', num)
```

```
            if self.check_speedy_rising_volume(code):
                buy_list.append(code)

        self.update_buy_list(buy_list)
```

예제 19.3은 5주 차에서 개발한 PyMon 프로그램의 최종 코드입니다. 앞서 설명한 run 메서드를 수정해 유가증권시장과 코스닥시장의 전 종목에 대해 거래량 급등을 확인하도록 수정했습니다.

예제 19.3 PyMon 급등주 포착 알고리즘 구현(소스코드: book/ch19/day05/pymon.py)

```
import sys
from PyQt5.QtWidgets import *
import Kiwoom
import time
from pandas import DataFrame
import datetime

MARKET_KOSPI   = 0
MARKET_KOSDAQ  = 10

class PyMon:
    def __init__(self):
        self.kiwoom = Kiwoom.Kiwoom()
        self.kiwoom.comm_connect()
        self.get_code_list()

    def get_code_list(self):
        self.kospi_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSPI)
        self.kosdaq_codes = self.kiwoom.get_code_list_by_market(MARKET_KOSDAQ)

    def get_ohlcv(self, code, start):
        self.kiwoom.ohlcv = {'date': [], 'open': [], 'high': [], 'low': [], 'close': [], 'vol-
ume': []}

        self.kiwoom.set_input_value("종목코드", code)
        self.kiwoom.set_input_value("기준일자", start)
        self.kiwoom.set_input_value("수정주가구분", 1)
        self.kiwoom.comm_rq_data("opt10081_req", "opt10081", 0, "0101")
        time.sleep(0.2)

        df = DataFrame(self.kiwoom.ohlcv, columns=['open', 'high', 'low', 'close', 'volume'],
                       index=self.kiwoom.ohlcv['date'])
```

```
        return df

    def check_speedy_rising_volume(self, code):
        today = datetime.datetime.today().strftime("%Y%m%d")
        df = self.get_ohlcv(code, today)
        volumes = df['volume']

        if len(volumes) < 21:
            return False

        sum_vol20 = 0
        today_vol = 0

        for i, vol in enumerate(volumes):
            if i == 0:
                today_vol = vol
            elif 1 <= i <= 20:
                sum_vol20 += vol
            else:
                break

        avg_vol20 = sum_vol20 / 20
        if today_vol > avg_vol20 * 10:
            return True

    def update_buy_list(self, buy_list):
        f = open("buy_list.txt", "wt")
        for code in buy_list:
            f.writelines("매수;", code, ";시장가;10;0;매수전")
        f.close()

    def run(self):
        buy_list = []
        num = len(self.kosdaq_codes)

        for i, code in enumerate(self.kosdaq_codes):
            print(i, '/', num)
            if self.check_speedy_rising_volume(code):
                buy_list.append(code)

        self.update_buy_list(buy_list)

if __name__ == "__main__":
```

```
app = QApplication(sys.argv)
pymon = PyMon()
pymon.run()
```

02 개발 6일 차

개발 6일 차에서는 기업 실적 데이터를 바탕으로 종목을 선정하는 프로그램을 개발하겠습니다. 보통 기업 실적은 기업이 분기 또는 매년 결산기에 공시하는 재무제표를 통해 확인할 수 있는데 아쉽게도 키움증권의 OpenAPI+로는 재무제표에 대한 정보는 얻을 수 없습니다.

다행히도 일부 웹 사이트에서는 웹을 통해 기업의 주요 재무정보를 제공합니다. 대표적으로 네이버는 다음과 같은 URL을 통해 기업의 주요 재무정보를 제공합니다. 아래 주소에서 '035720'에 해당하는 부분이 종목코드입니다. 이 값을 변경함으로써 각 종목에 대한 주요 재무정보를 확인할 수 있습니다.

- http://companyinfo.stock.naver.com/v1/company/ajax/cF1001.aspx?cmp_cd=035720&fin_typ=0&freq_typ=Y[1]

먼저 익스플로러나 크롬과 같은 웹 브라우저를 통해 해당 URL로 접속해 보기 바랍니다(페이지 하단에 소개된 단축 URL을 통해 접근하는 것이 편리합니다). 그림 19.6과 같이 매출액, 영업이익, 당기순이익 등 다양한 재무 정보를 확인할 수 있습니다.



주요재무정보

주요재무정보	2011/12 (IFRS연결)	2012/12 (IFRS연결)	2013/12 (IFRS연결)	2014/12 (IFRS연결)	연간 2015/12 (IFRS연결)
매출액	4,213	4,534	2,108	4,989	9,322
영업이익	1,168	1,018	659	1,764	886
세전계속사업이익	1,274	1,009	604	1,688	1,095
당기순이익	1,080	766	614	1,498	788
당기순이익(지배)	1,085	763	614	1,501	757
당기순이익(비지배)	-5	4	0	-3	31
자산총계	5,587	6,119	2,172	27,680	31,885
부채총계	1,025	919	399	3,048	6,030
자본총계	4,562	5,199	1,773	24,632	25,855

그림 19.6 웹 브라우저를 통한 주요 재무정보 확인

1 단축 URL: https://goo.gl/Pp6haJ

참고로 종목코드 '035720'은 카카오입니다. 데이터를 검증하기 위해 다음 금융(finance.daum.net) 사이트에서 카카오의 기업 실적을 확인해보면 그림 19.7과 같습니다. 그림 19.6과 그림 19.7을 비교해 보면 천만 원 단위에서의 올림 처리에 따라 억 단위의 숫자에 약간 차이가 있을 뿐 거의 같음을 확인할 수 있습니다.

기업실적 더보기 ▸

주요재무지표	분기				연간			
	2015/09	2015/12	2016/03	2016/06	2012	2013	2014	2015
	IFRS 연결	IFRS 연결	IFRS 연결	IFRS 연결	IFRS 연결	IFRS 연결	IFRS 연결	IFRS 연결
매출액(억원)	2,295	2,416	2,424	3,765	4,533	2,107	4,988	9,321
영업이익(억원)	161	206	210	266	1,017	658	1,764	885
당기순이익(억원)	147	117	109	132	766	614	1,498	787
EPS(원)	217	183	214	131	5,656	4,532	6,116	1,269
ROE(%)	0.52	0.43	0.45	0.27	15.70	17.73	11.41	3.02
부채비율(%)	14.43	23.32	51.89	48.51	17.68	22.53	12.37	23.32
주당배당금(원)	-	167	-	-	1,110	1,133	173	167

그림 19.7 다음 금융(http://finance.daum.net)을 통한 기업 실적 확인(카카오)

HTML 기초

웹 페이지로부터 데이터를 추출하는 행위를 크롤링(Crawling)이라고 합니다. 앞서 살펴본 재무정보 웹 페이지도 크롤링함으로써 원하는 데이터만 추출해낼 수 있습니다. 웹 페이지에서 크롤링을 잘하려면 웹 페이지를 만드는 데 사용되는 언어인 HTML(HyperText Markup Language)에 대해 조금은 알고 있는 것이 좋습니다. 그래서 이번 절에서는 크롤링에 필요한 기본적인 HTML 지식을 간단히 살펴보겠습니다.

HTML은 하이퍼텍스트 마크업 언어입니다. 웹 페이지는 일반 텍스트와 달리 링크를 클릭하면 다른 페이지로 이동합니다. 이처럼 서로 연결(링크)된 문자 데이터를 하이퍼텍스트라고 합니다. 마크업 언어라는 것은 문서나 데이터의 구조를 명기하는 언어를 의미합니다. 즉, 하이퍼텍스트 마크업 언어라는 것은 웹 페이지(하이퍼텍스트)를 만드는 데 사용되는 언어인 것입니다.

간단한 코드를 통해 HTML을 배워보겠습니다. 먼저 윈도우 메모장을 실행합니다. 그리고 다음과 같은 코드를 입력합니다. 다음은 HTML을 사용해서 웹 페이지를 만든 것입니다.

```
<html>
<table border=1>
    <tr>
```

```
            <td> 항목 </td>
            <td> 2013 </td>
            <td> 2014 </td>
            <td> 2015 </td>
        </tr>

        <tr>
            <td> 매출액 </td>
            <td> 100 </td>
            <td> 200 </td>
            <td> 300 </td>
        </tr>
</table>
</html>
```

메모장에서 [파일] → [다른 이름으로 저장]을 차례로 선택합니다. 파일을 저장하는 위치로 윈도우 바탕 화면을 선택한 후 그림 19.8과 같이 파일명을 'table.html'로, 파일 형식을 '모든 파일'로 변경합니다. 인코딩 설정도 'UTF-8'로 변경한 후 [저장] 버튼을 눌러 파일로 저장합니다.

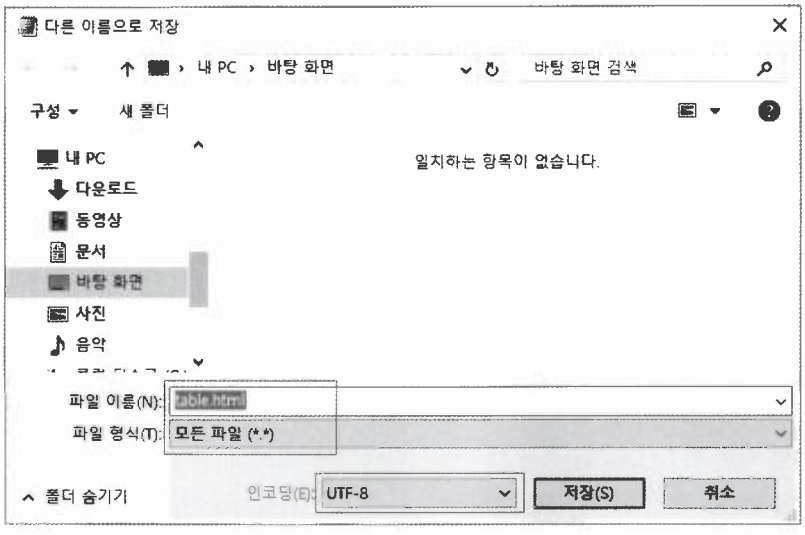


그림 19.8 table.html 파일로 저장

바탕 화면에 저장된 table.html을 클릭해보면 그림 19.9와 같이 웹 브라우저에서 table.html 파일이 열린 것을 확인할 수 있습니다. 앞서 여러분이 HTML로 작성한 내용은 사실 표를 만든 것입니다. 인터넷 익스플로러나 크롬 같은 웹 브라우저가 HTML 코드를 해석해서 그림 19.9와 같이 여러분이 의도한 대로 표를 출력하는 것입니다.

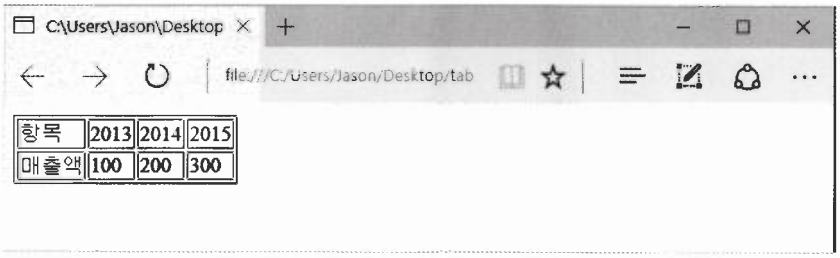


그림 19.9 웹 브라우저에서 table.html 파일 열기

앞에서 작성한 HTML 코드를 살펴보겠습니다. HTML 코드에서 <html>, <table>, <tr>, <td> 같은 것들을 태그라고 부릅니다. HTML에서 태그는 항상 쌍으로 존재합니다. 예를 들어, HTML 코드가 시작함을 의미하는 <html>이 있다면 코드의 끝부분에는 HTML 코드의 끝을 의미하는 </html>이 위치해야 합니다.

표를 만들기 위해서는 <table> … </table> 태그를 사용합니다. 표는 행(row)과 열(column)로 구성되는데, 행을 만들 때는 <tr> … </tr> 태그를 사용하고, 열을 만들 때는 <td> … </td>를 사용합니다. 표가 웹 페이지 안에 존재하므로 <table> … </table> 태그는 다음과 같이 <html> … </html> 태그 안에 위치해야 합니다. 표의 선 두께는 border 속성을 통해 설정할 수 있습니다.

```
<html>
<table border=1>

</table>
</html>
```

그림 19.9를 참조하면 표에는 두 개의 행이 존재합니다. 행이라는 것은 표 안에 존재하는 것이기 때문에 다음과 같이 <tr> … </tr> 태그를 <table> … </table> 안쪽으로 위치시킵니다. 행이 두 개이기 때문에 <tr> … </tr> 태그 쌍이 두 번 위치합니다. 참고로 HTML 코드에서 공백은 의미를 갖지 않지만 코드의 가독성을 위해 적절히 띄어쓰는 것이 좋습니다.

```
<html>
<table border=1>
      <tr>
      </tr>

      <tr>
      </tr>
</table>
</html>
```

그림 19.9에 나온 표의 각 행에는 네 개의 열이 존재합니다. 따라서 <tr> … </tr> 태그 안쪽으로 <td> … </td> 태그 쌍을 네 번 넣어줍니다.

```
<html>
<table border=1>
      <tr>
           <td> </td>
           <td> </td>
           <td> </td>
           <td> </td>
      </tr>

      <tr>
           <td> </td>
           <td> </td>
           <td> </td>
           <td> </td>
      </tr>
</table>
</html>
```

마지막으로 표의 셀(표 안의 공간)에 데이터를 넣으면 되는데 데이터는 <td>와 </td> 사이에 넣으면 됩니다.

```
<html>
<table border=1>
      <tr>
           <td> 항목 </td>
           <td> 2013 </td>
           <td> 2014 </td>
           <td> 2015 </td>
      </tr>

      <tr>
           <td> 매출액 </td>
           <td> 100 </td>
           <td> 200 </td>
           <td> 300 </td>
      </tr>
</table>
</html>
```

이로써 간단히 HTML 태그에 대해 살펴봤습니다. 배운 내용을 정리해 보면 웹 페이지를 만들기 위해서는 HTML 태그를 사용해서 코드를 작성해야 합니다. 물론 HTML 태그를 사용해 직접 코드를 작성하지 않고 어도비 드림위버(Adobe Dreamweaver) 같은 위지위그 기반의 프로그램을 사용해 손쉽게 웹 페이지를 만들 수도 있습니다. 이는 여러분이 PyQt로 GUI를 구성할 때 직접 코드를 입력해서 만들 수도 있었고 Qt Designer를 이용해 UI 구성을 쉽게 수행했던 것과 같습니다.

어쨌거나 위지위그 기반의 프로그램을 사용하든 직접 HTML을 사용하든 결과물은 그림 19.10과 같이 HTML 태그로 구성된 *.html 파일입니다. 이 파일을 인터넷 익스플로러, 파이어폭스, 크롬 같은 웹 브라우저가 읽으면 웹 브라우저가 HTML 태그를 해석한 후 이를 웹 페이지로 보여주는 것입니다.

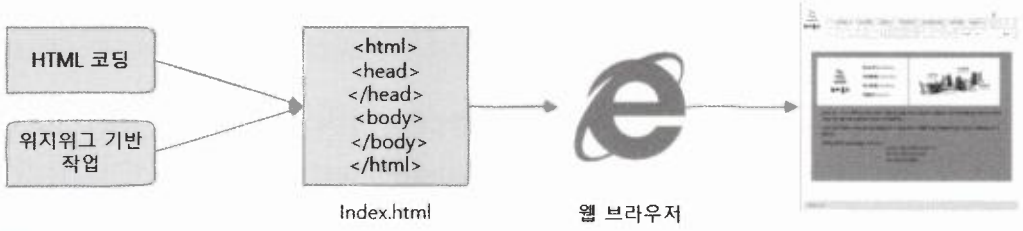


그림 19.10 웹 페이지 제작 및 출력 과정

웹 페이지 크롤링

이번 절에서는 웹 페이지에서 데이터를 추출하는 크롤링에 대해 알아보겠습니다. 파이썬에는 크롤링을 위한 다양한 라이브러리가 있습니다. 대표적으로 Beautiful Soup과 Scrapy를 많이 사용합니다. 이 밖에도 앞서 배운 pandas를 이용하면 간단한 형태의 HTML에서 데이터를 추출할 수도 있습니다.

pandas를 이용해 앞서 생성한 table.html 파일로부터 데이터를 추출해 보겠습니다. 이를 위해 먼저 아나콘다 프롬프트를 실행합니다. pandas로 웹 페이지 데이터를 파싱하려면 html5lib 모듈이 필요합니다. 그림 19.11과 같이 아나콘다 프롬프트에 'conda install html5lib'을 입력합니다. html5lib 설치에 필요한 추가 패키지가 출력되고 설치를 묻는 화면이 나오면 'y'를 눌러 설치를 진행합니다.



그림 19.11 html5lib 설치

html5lib 설치가 완료되면 Jupyter QtConsole을 실행합니다. 앞서 바탕 화면에 생성했던 table.html을 파싱하기 위해 다음과 같은 코드를 작성합니다. pandas의 read_html 메서드는 인자로 웹 페이지 경로 또는 HTML 파일의 경로를 입력받아 해당 웹 페이지에 있는 표 내용을 pandas의 DataFrame 객체로 만든 후 반환합니다. 다음 코드에서 df 객체가 바인딩하는 값을 확인해보면 표 내용이 정상적으로 파싱됐음을 확인할 수 있습니다.

```
In [1]: import pandas as pd

In [2]: df = pd.read_html("C:\\Users\\Jason\\Desktop\\table.html")[0]

In [3]: df
Out[3]:
0 1 2 3
0 항목 2013 2014 2015
1 매출액 100 200 300

In [4]:
```

표 안의 데이터가 파싱된 후 DataFrame 객체로 저장됐으므로 다음과 같이 DataFrame 객체의 행과 열에 접근할 수 있습니다.

```
In [4]: df.ix[0]
Out[4]:
0 항목
1 2013
```

```
2 2014
3 2015
Name: 0, dtype: object

In [5]: df.ix[1]
Out[5]:
0 매출액
1 100
2 200
3 300
Name: 1, dtype: object

In [6]: df[0]
Out[6]:
0 항목
1 매출액
Name: 0, dtype: object

In [7]:
```

이번에는 앞서 설명한 네이버 금융 사이트로부터 재무제표 데이터를 파싱해 보겠습니다. 웹 페이지로부터 재무제표를 얻어오는 함수는 webreader 모듈 내에 구현하겠습니다. PyCharm을 실행한 후 webreader.py 파일을 생성하고 다음과 같이 기본 코드를 구현합니다.

```
def get_financial_statements(code):
    url = "http://companyinfo.stock.naver.com/v1/company/ajax/cF1001.aspx?cmp_cd=%s&fin_typ=0&-
freq_typ=Y" % (code)
    print(url)

if __name__ == "__main__":
    get_financial_statements('035720')
```

get_financial_statements 함수는 함수 인자로 종목코드를 받습니다. 함수 인자로 전달된 종목코드는 URL의 중간에 삽입되어 최종 URL로 완성됩니다. 코드를 실행하면 그림 19.12와 같이 완성된 URL이 출력되는 것을 확인할 수 있습니다.



그림 19.12 완성된 URL 출력

데이터 요청을 위한 URL이 완성됐으니 해당 주소로부터 HTML 코드를 받아오면 됩니다. 웹 브라우저에 URL을 입력하면 웹 브라우저는 해당 주소로부터 HTML 코드를 받아 이를 파싱해서 여러분에게 보여줍니다.

데이터 파싱을 위해서는 여러분도 웹 브라우저와 비슷하게 URL로부터 먼저 HTML 코드를 다운로드해야 합니다. 파이썬에서는 requests 모듈을 이용하면 이러한 기능을 수행할 수 있습니다. 먼저 requests 모듈을 임포트한 후 get 함수를 통해 Response 클래스의 객체를 얻은 후 text 속성에 접근하면 HTML 코드를 얻을 수 있습니다.

```
import requests

def get_financial_statements(code):
    url = "http://companyinfo.stock.naver.com/v1/company/ajax/cF1001.aspx?cmp_cd=%s&fin_typ=0&-
freq_typ=Y" % (code)
    html = requests.get(url).text
    print(html)
    #print(type(requests.get(url)))
    #print(type(requests.get(url).text))

if __name__ == "__main__":
    get_financial_statements('035720')
```

위 코드를 실행하면 파이썬 문자열 타입의 HTML 코드를 html이라는 변수가 바인딩하고, print(html) 구문이 실행되어 그림 19.13과 같이 받아온 HTML 코드가 출력되는 모습을 확인할 수 있습니다.



그림 19.13 받아온 HTML 코드 출력

참고로 requests.get 메서드의 반환값이 궁금하신 분들은 다음 코드의 주석을 해제한 후 프로그램을 실행해보기 바랍니다.

```
#print(type(requests.get(url)))
#print(type(requests.get(url).text))
```

HTML 코드를 웹으로부터 받아왔으므로 pandas의 read_html 함수를 이용해 표의 내용을 파싱합니다. 웹 페이지에 여러 개의 표가 있을 수 있으므로 read_html 함수에서는 DataFrame 객체의 리스트를 반환합니다. 따라서 웹 페이지에 표가 하나만 존재하는 경우 파이썬 리스트에서 [0]을 이용해 인덱싱하면 됩니다.

```
import requests
import pandas as pd

def get_financial_statements(code):
    url = "http://companyinfo.stock.naver.com/v1/company/ajax/cF1001.aspx?cmp_cd=%s&fin_typ=0&-
freq_typ=Y" % (code)
    html = requests.get(url).text

    df_list = pd.read_html(html, index_col='주요재무정보')
    df = df_list[0]
    print(df)

if __name__ == "__main__":
    get_financial_statements('035720')
```

위 코드를 실행하면 그림 19.14와 같이 DataFrame 객체는 가져왔지만 DataFrame 객체의 칼럼명에 필요 없는 문자열이 존재하는 것을 확인할 수 있습니다.

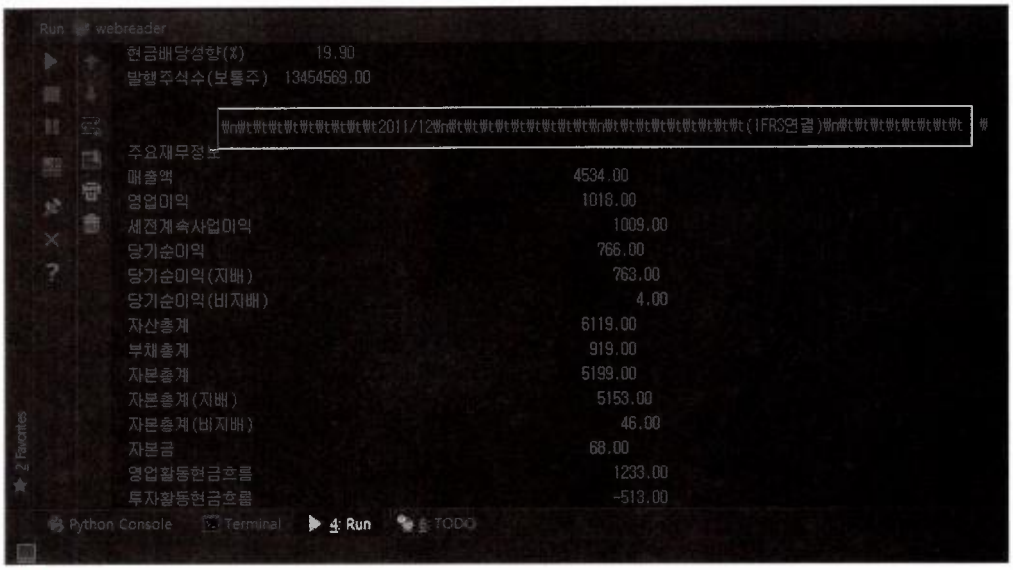


그림 19.14 read_html 함수를 이용한 재무제표 파싱 결과

이러한 문제는 표의 셀에 들어있는 데이터에 공백이 존재하기 때문입니다. 또한 그림 19.6을 참조하면 표에 '연간'이라는 셀이 존재하는데, 사실 이 부분은 테이블을 DataFrame 객체로 만들 때 필요 없는 부분입니다.

DataFrame 객체의 칼럼들의 이름과 인덱스 값을 좀 더 정확히 확인하기 위해 columns와 index 속성을 출력하도록 코드를 변경합니다.

```
import requests
import pandas as pd

def get_financial_statements(code):
    url = "http://companyinfo.stock.naver.com/v1/company/ajax/cF1001.aspx?cmp_cd=%s&fin_typ=0&-
freq_typ=Y" % (code)
    html = requests.get(url).text

    df_list = pd.read_html(html, index_col='주요재무정보')
    df = df_list[0]

    print(df.columns)
    print(df.index)

if __name__ == "__main__":
    get_financial_statements('035720')
```

그림 19.15를 참조하면 재무제표 페이지를 파싱해서 생성한 DataFrame 객체의 인덱스는 정상적으로 파싱됐지만 칼럼의 이름에는 '연간'이라는 칼럼이 들어가 있고 또한 각 연도에 대한 텍스트의 앞쪽과 뒤쪽으로 불필요한 데이터가 존재하는 것을 좀 더 정확히 확인할 수 있습니다.

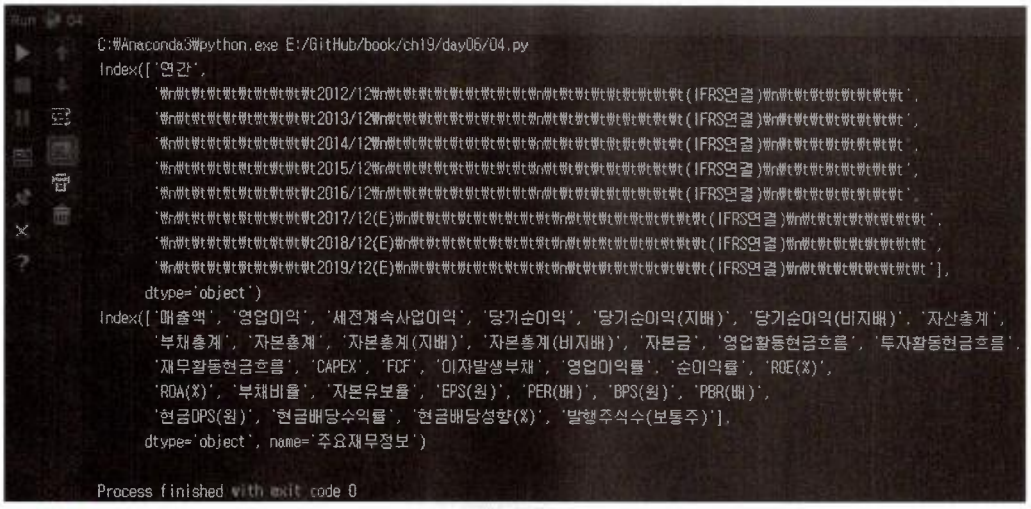


그림 19.15 DataFrame 객체의 칼럼 이름과 인덱스

여러분이 pandas의 read_html 함수를 변경할 수는 없기 때문에 HTML 코드를 적당히 변경한 후 read_html 함수의 인자로 전달해야 이러한 문제를 해결할 수 있습니다. HTML 코드는 파이썬 문자열 타입이므로 replace 메서드를 사용해 기존의 문자열 중 일부를 다른 문자열로 변경할 수 있습니다. 다음과 같이 replace 메서드를 사용해 불필요한 HTML 태그 및 공백 문자열을 제거합니다.

```
import requests
import pandas as pd

def get_financial_statements(code):
    url = "http://companyinfo.stock.naver.com/v1/company/ajax/cF1001.aspx?cmp_cd=%s&fin_typ=0&-
freq_typ=Y" % (code)
    html = requests.get(url).text

    html = html.replace('<th class="bg r01c02 endLine line-bottom"colspan="8">연간</th>', "")
    html = html.replace("<span class='span-sub'>(IFRS연결)</span>", "")
    html = html.replace("<span class='span-sub'>(IFRS별도)</span>", "")
    html = html.replace("<span class='span-sub'>(GAAP개별)</span>", "")
    html = html.replace('\t', '')
    html = html.replace('\n', '')
    html = html.replace('\r', '')
```

```
    df_list = pd.read_html(html, index_col='주요재무정보')
    df = df_list[0]

    print(df.columns)
    print(df.index)

if __name__ == "__main__":
    get_financial_statements('035720')
```

다시 코드를 실행해 보면 이번에는 DataFrame 객체의 인덱스에 '연간'이라는 데이터가 빠졌고 연도 값만 남아 있음을 확인할 수 있습니다.

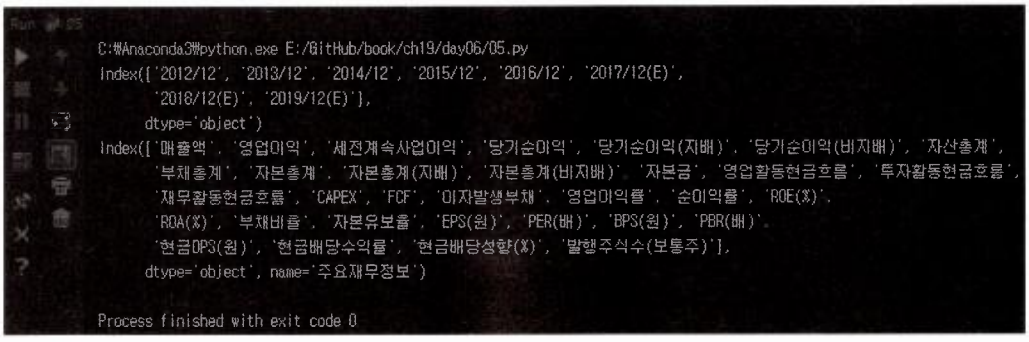


그림 19.16 DataFrame 객체의 칼럼 이름과 인덱스(2)

칼럼의 인덱스에는 각 연도별 재무제표 데이터가 존재합니다. 재무제표 데이터를 처리하다 보면 회사마다 결산월이 다른 것을 확인할 수 있습니다. 이 책에서는 데이터를 쉽게 처리하기 위해 5월까지 공시된 경우 이전 연도로 간주하고 6월부터는 당해 연도로 간주했습니다. 이를 위해 다음과 같은 코드를 추가합니다.

```
def get_financial_statements(code):
    # 코드 생략
    for year in range(2009, 2018):
        for month in range(6, 13):
            month = "/%02d" % month
            html = html.replace(str(year) + month, str(year))

        for month in range(1, 6):
            month = "/%02d" % month
            html = html.replace(str(year+1) + month, str(year))

        html = html.replace(str(year) + '(E)', str(year))
```

지금까지 구현한 get_financial_statements 함수의 최종 버전은 예제 19.4와 같습니다. 해당 함수는 웹 페이지로부터 재무제표 데이터를 추출한 후 DataFrame 객체로 반환합니다.

예제 19.4 네이버 금융으로부터 재무제표를 파싱하는 함수(소스코드: book/ch19/day06/05.py)

```
import requests
import pandas as pd

def get_financial_statements(code):
    url = "http://companyinfo.stock.naver.com/v1/company/ajax/cF1001.aspx?cmp_cd=%s&fin_typ=0&-
freq_typ=Y" % (code)
    html = requests.get(url).text

    html = html.replace('<th class="bg r01c02 endLine line-bottom"colspan="8">연간</th>', "")
    html = html.replace("<span class='span-sub'>(IFRS연결)</span>", "")
    html = html.replace("<span class='span-sub'>(IFRS별도)</span>", "")
    html = html.replace("<span class='span-sub'>(GAAP개별)</span>", "")
    html = html.replace('\t', '')
    html = html.replace('\n', '')
    html = html.replace('\r', '')

    for year in range(2009, 2018):
        for month in range(6, 13):
            month = "/%02d" % month
            html = html.replace(str(year) + month, str(year))

        for month in range(1, 6):
            month = "/%02d" % month
            html = html.replace(str(year+1) + month, str(year))

        html = html.replace(str(year) + '(E)', str(year))

    df_list = pd.read_html(html, index_col='주요재무정보')
    df = df_list[0]
    return df

if __name__ == "__main__":
    df = get_financial_statements('035720')
    print(df)
```

배당률 기반 투자 알고리즘

최근 저자는 『치과의사 피트씨의 똑똑한 배당주 투자』(피트 황/스마트북스)라는 책을 읽었습니다. 이 책에서는 국채시가배당률이라는 투자 지표를 소개합니다. 간단히 말해서 국채시가배당률은 '현금배당수익률'을 '3년 만기 국채 수익률'로 나눈 값을 의미합니다.

$$\text{국채시가배당률(배)} = \frac{\text{현금 배당 수익률}}{\text{3년만기 국채 수익률}}$$

3년 만기 국채 수익률은 시장 금리와 같은 역할을 하는 대표적인 지표인데, 국채시가배당률이 1.0보다 크다는 것은 현금배당수익률이 3년 만기 국채 수익률보다 높은 경우입니다. 즉, 돈을 은행에 저금해서 얻는 이자 수익보다 주식을 사서 배당금을 받는 수익이 더 높음을 의미합니다. 한마디로 국채시가배당률이 높은 종목은 좋은 배당주라고 해석할 수 있습니다. 종목을 선정할 때 또는 같은 종목이라도 매수/매도 시점을 파악할 때 국채시가배당률 값이 상대적으로 높으면 좋은 매수 시기이고 상대적으로 낮다면 좋은 매도 시기로 해석하는 것입니다.

이번 절에서는 앞서 파싱한 재무제표 데이터를 바탕으로 국채시가배당률이 높은 종목을 선정하는 프로그램을 구현하겠습니다. 국채시가배당률을 계산하려면 3년 만기 국채 수익률이 필요합니다. 3년 만기 국채 수익률은 국가지표체계 웹 사이트에서 다음 URL을 통해 얻을 수 있습니다(페이지 하단의 단축 URL을 사용하면 더욱 쉽게 접속할 수 있습니다).

- http://www.index.go.kr/strata/jsp/showStblGams3.jsp?stts_cd=288401&idx_cd=2884&freq=Y&period=1998:2016[2]

웹 브라우저에서 위 URL 또는 단축 URL로 들어가보면 그림 19.17과 같이 1998년 이후로 연 단위로 국고채 3년(평균) 값이 제공되는 것을 확인할 수 있습니다.

	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
국고채금리	6.95	9.03	6.70	5.91	5.11	4.82	3.28	5.08	4.92	5.74	3.41	4.41	3.38	3.34	2.82	2.86

그림 19.17 국가지표체계 웹 사이트(www.index.go.kr)

그림 19.17의 표에 나온 국고채금리를 파싱하는 함수를 webreader.py 파일에 get_3year_treasury라는 이름으로 구현하겠습니다. 해당 주소로부터 HTML 코드를 얻어온 후 이를 화면에 출력하는 기능까지 먼저 구현합니다.

2 단축 URL: https://goo.gl/sGBJ2h

```
def get_3year_treasury():
    url = "http://www.index.go.kr/strata/jsp/showStblGams3.jsp?stts_cd=288401&amp;idx_
cd=2884&amp;freq=Y&amp;period=1998:2016"
    html = requests.get(url).text
    print(html)
```

webreader.py 파일의 하단부를 수정해 get_3year_treasury 함수를 호출해서 결과를 확인해 봅시다.

```
if __name__ == "__main__":
    get_3year_treasury()
```

그림 19.18과 같이 가져온 HTML 코드가 출력되는 것을 확인할 수 있습니다.



그림 19.18 국고채금리 페이지에 대한 HTML 코드

그림 19.17의 표에서 국고채금리에 해당하는 행 데이터를 파싱해 보겠습니다. 이번에는 파이썬에서 HTML로부터 데이터를 파싱할 때 많이 사용하는 Beautiful Soup 라이브러리를 사용하겠습니다. 먼저 webreader.py 파일 상단에 모듈을 임포트합니다.

```
from bs4 import BeautifulSoup
```

HTML 문서를 파싱하려면 HTML 코드를 BeautifulSoup 객체로 변환해야 합니다.

```
def get_3year_treasury():
    url = "http://www.index.go.kr/strata/jsp/showStblGams3.jsp?stts_cd=288401&amp;idx_
cd=2884&amp;freq=Y&amp;period=1998:2016"
    html = requests.get(url).text
    soup = BeautifulSoup(html, 'lxml')
```

BeautifulSoup 클래스의 인스턴스를 생성했으므로 해당 클래스가 제공하는 속성이나 메서드를 사용할 수 있습니다. 대표적으로 find_all 메서드는 특정 HTML 태그가 사용된 부분을 가져올 때 많이 사용합니다.

find_all 메서드를 사용하려면 먼저 HTML 소스의 구조를 잘 알아야 합니다. 크롬 브라우저에서 마우스 오른쪽 버튼을 클릭한 후 그림 19.19와 같이 [검사]를 선택합니다.

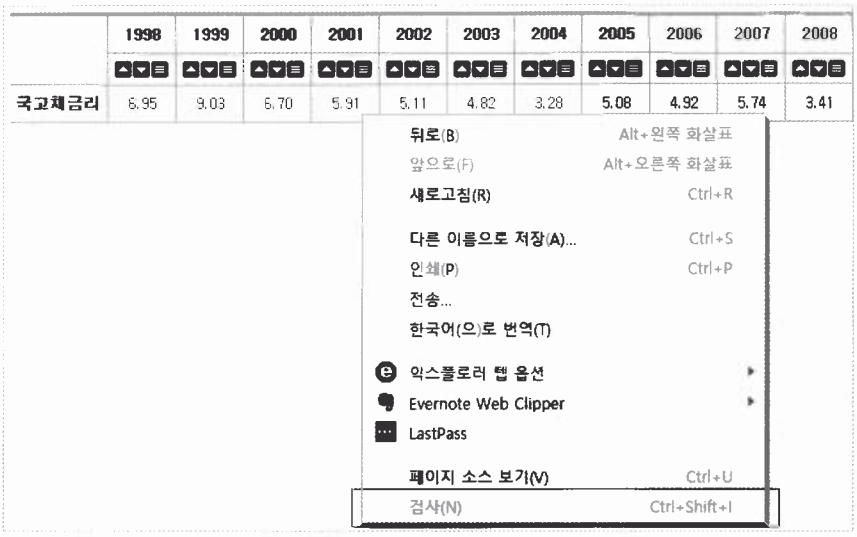


그림 19.19 크롬 웹 브라우저의 검사 기능

HTML에서 <tr> … </tr>은 행을 의미한다고 배웠습니다. 그림 19.20을 보면 오른쪽 편의 HTML 코드에서 <tr id="tr_288401_1">을 마우스로 클릭하면 왼쪽 표에서 '국고채금리' 행이 선택되는 것을 확인할 수 있습니다. 이처럼 크롬 웹 브라우저의 검사 기능을 사용하면 쉽게 특정 부분에 해당하는 HTML 코드를 확인할 수 있습니다.

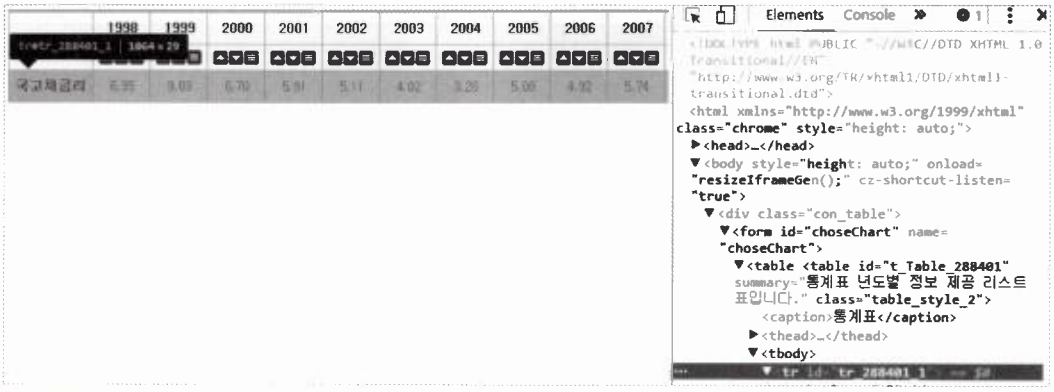


그림 19.20 표 구조 분석

BeautifulSoup 클래스의 find_all 메서드를 이용해 해당 행의 HTML 코드만 가져옵니다. find_all 메서드의 첫 번째 인자로 검색하고자 하는 HTML 태그를 입력하면 되고 id 값을 지정해 해당 id 값을 갖는 행만 추출할 수 있습니다.

```
def get_3year_treasury():
    url = "http://www.index.go.kr/strata/jsp/showStblGams3.jsp?stts_cd=288401&amp;idx_
cd=2884&amp;freq=Y&amp;period=1998:2016"
    html = requests.get(url).text
    soup = BeautifulSoup(html, 'lxml')
    tr_data = soup.find_all('tr', id='tr_288401_1')
```

행에는 열을 구성하는 <td> … </td> 태그가 존재합니다. 역시 find_all 메서드를 호출해 <td> 태그 데이터만 추출합니다.

```
def get_3year_treasury():
    url = "http://www.index.go.kr/strata/jsp/showStblGams3.jsp?stts_cd=288401&amp;idx_
cd=2884&amp;freq=Y&amp;period=1998:2016"
    html = requests.get(url).text
    soup = BeautifulSoup(html, 'lxml')
    tr_data = soup.find_all('tr', id='tr_288401_1')

    td_data = tr_data[0].find_all('td')
    print(td_data)
```

얻어온 td_data를 화면에 출력해보면 그림 19.21과 여러 개의 <td> … </td> 쌍이 존재하는 것을 확인할 수 있습니다.



그림 19.21 TD 태그 목록

td_data는 ResultSet이라는 데이터 타입인데, 파이썬 리스트와 같이 인덱싱을 통해 각 원소에 접근할 수 있습니다. 0번, 1번, 2번 원소에 접근한 후 text 속성값을 화면에 출력해 봅시다.

```
def get_3year_treasury():
    url = "http://www.index.go.kr/strata/jsp/showStblGams3.jsp?stts_cd=288401&amp;idx_
cd=2884&amp;freq=Y&amp;period=1998:2016"
    html = requests.get(url).text
    soup = BeautifulSoup(html, 'lxml')
```

```
    tr_data = soup.find_all('tr', id='tr_288401_1')

    td_data = tr_data[0].find_all('td')
    print(type(tr_data))
    print(td_data[0].text)
    print(td_data[1].text)
    print(td_data[2].text)
```

위 코드를 실행하면 다음과 같이 1998년, 1999년, 2000년도의 국고채금리 값이 출력되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/07.py
<class 'bs4.element.ResultSet'>
6.95
9.03
6.70

Process finished with exit code 0
```

연도별 국고채금리(3년 만기 국채 수익률)를 쉽게 얻을 수 있도록 파이썬 딕셔너리 형태로 데이터를 저장해 보겠습니다. 다음과 같이 treasury_3year라는 빈 딕셔너리 객체를 생성한 후 td_data를 순회하면서 데이터를 하나씩 딕셔너리 객체에 추가하면 됩니다. 이때 연도는 1998년도부터 시작해서 1씩 증가시키면서 이 값을 딕셔너리의 키 값으로 사용했습니다.

```
def get_3year_treasury():
    url = "http://www.index.go.kr/strata/jsp/showStblGams3.jsp?stts_cd=288401&amp;idx_
cd=2884&amp;freq=Y&amp;period=1998:2016"
    html = requests.get(url).text
    soup = BeautifulSoup(html, 'lxml')
    tr_data = soup.find_all('tr', id='tr_288401_1')
    td_data = tr_data[0].find_all('td')

    treasury_3year = {}
    start_year = 1998

    for x in td_data:
        treasury_3year[start_year] = x.text
        start_year += 1

    print(treasury_3year)
    return treasury_3year
```

위 코드를 실행하면 다음과 같이 각 연도별 3년 만기 국채 수익률이 저장된 딕셔너리 객체가 생성되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/07.py
{2016: '1.64', 1998: '6.95', 1999: '9.03', 2000: '6.70', 2001: '5.91', 2002: '5.11', 2003:
'4.82', 2004: '3.28', 2005: '5.08', 2006: '4.92', 2007: '5.74', 2008: '3.41', 2009: '4.41',
2010: '3.38', 2011: '3.34', 2012: '2.82', 2013: '2.86', 2014: '2.10', 2015: '1.66'}

Process finished with exit code 0
```

배당률 기반 투자 알고리즘 구현(1)

이번 절에서는 국채시가배당률 기반의 매매 알고리즘을 구현해 보겠습니다. 앞서 설명한 것처럼 국채 시가배당률을 계산하려면 현금배당수익률과 3년 만기 국채 수익률이 필요합니다.

$$\text{국채시가배당률(배)} = \frac{\text{현금 배당 수익률}}{\text{3년만기 국채 수익률}}$$

현금 배당 수익률은 주당배당금(DPS; Dividend per Share)을 주가로 나눈 후 100을 곱한 값입니다.

$$\text{현금배당수익률(\%)} = \frac{\text{주당배당금}}{\text{주가}} \times 100$$

어떤 기업의 현금 배당이 연말결산 후 4월에 한 번 지급된다고 가정하면 주당배당금은 다음 배당 전까지는 변하지 않는 값입니다. 이와 달리 주가는 거래일마다 변동되는 값이므로 어떤 시점의 주가를 사용했는가에 따라 현금배당수익률은 달라질 수 있습니다.

$$\text{주당배당금} = \frac{\text{배당금 총액}}{\text{발행주식수}}$$

국채시가배당률 기반의 투자는 나름 논리적이지만 이를 당장 시스템 트레이딩으로 구현하기에는 몇 가지 어려움이 있습니다. 첫 번째는 주당배당금이라는 것이 과거의 값이라는 점입니다. 기업이 일년 동안 영업하고 난 결과에 대해 배당 여부를 결정하기 때문에 현 시점에서는 이번 영업 연도에 배당이 진행될지, 그리고 얼마만큼 현금 배당이 나올지 알 수가 없습니다.

두 번째는 웹 사이트를 통해 제공되는 현금배당수익률은 분모에 사용되는 주가가 현재 시점의 주가가 아니라 과거 시점(배당기준일)의 일정 기간 동안의 주가일 수 있다는 점입니다. 만약 웹 사이트가 현금배당수익률의 계산에 사용됐던 시점의 주가보다 현재 주가가 큰 폭으로 올랐다면 웹 사이트 등을 통해 제공된 현금배당수익률을 그대로 사용할 수는 없습니다.

세 번째는 3년 만기 국채 수익률도 1년에 한 번 제공되는 값이라는 것입니다. 따라서 일 단위로 알고리즘을 동작시키는 경우 데이터가 변동되지 않기 때문에 동일한 결과가 나올 것입니다. 쉽게 말하면 일년에 딱 한번 매매 신호가 발생하고 나머지 기간 동안에는 동일한 신호가 발생합니다.

먼저 첫 번째 문제의 해결책부터 생각해보겠습니다. 미래의 주당 배당금을 알 수 있는 방법은 없습니다. 다만 기업도 순이익이 있어야 배당을 할 수 있기 때문에 증권사에서 제공하는 추정 실적을 통해 주당배당금을 어느 정도 예측할 수는 있습니다.

두 번째 문제는 웹 사이트를 통해 제공되는 현금배당수익률이 어떻게 계산됐는지 확인하면 됩니다. 만일 현금배당수익률이 배당 기준일의 주가를 토대로 계산됐다면 웹 사이트를 통해 현금배당수익률 데이터를 얻는 것이 아니라 직접 추정 실적을 바탕으로 거래일마다 예상 주당배당금과 종가를 사용해 예상 현금배당수익률을 계산하면 될 것입니다.

세 번째 문제의 해결책은 간단합니다. 사실 3년 만기 국채 수익률은 일별 시세가 존재합니다. 앞서 연도별 값은 일별 시세의 산술평균 값입니다. 따라서 매일 3년 만기 국채 수익률의 일별 시세를 조회한다면 거래일별로 최신 데이터를 확보할 수 있습니다.

국채시가 배당률이라는 간단한 아이디어도 이를 구현하려다 보면 많은 곳에서 구체화가 필요하다는 점을 알 수 있습니다. 말로는 간단할지 몰라도 이를 프로그램으로 구현하려면 모든 것이 구체적이고 정확해야 합니다.

앞서 네이버를 통해 얻어온 재무제표 데이터는 WISEfn이라는 회사를 통해 제공됩니다. 다음 URL에 접속해 보면 종목코드가 058470인 리노공업에 대한 기업 현황을 확인할 수 있습니다. 다른 기업을 조회하려면 종목코드만 변경하면 됩니다.

- http://companyinfo.stock.naver.com/company/c1010001.aspx?cmp_cd=058470[3]

해당 페이지를 잘 보면 그림 19.22와 같이 웹 페이지 상단부의 현금배당수익률은 최근 결산 수정 DPS(현금)를 전일자 보통주 수정주가로 나눠서 제공하고 있음을 확인할 수 있습니다. 앞서 언급한 현금배당수익률 계산식에서 주가 부분에 최근 거래일 기준의 종가가 이미 반영돼 있습니다.

3 단축 URL: https://goo.gl/iZxLS2

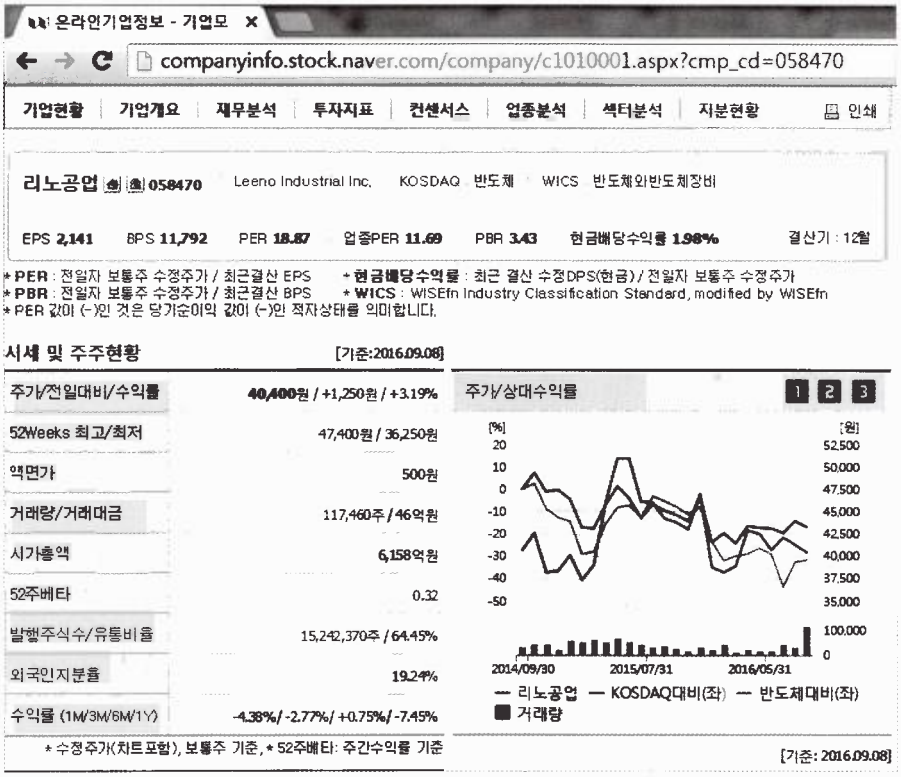


그림 19.22 기업현황 확인

또한 그림 19.23에서 알 수 있듯이 연간 재무제표에서도 연간 예상 실적을 기준으로 이미 예상 현금배당수익률을 제공하고 있음을 확인할 수 있습니다. 앞서 그림 19.22에서 제공된 현금배당수익률은 최근 결산 수정 DPS를 사용해서 계산한 것이고, 그림 19.23에서 제공된 현금배당수익률은 예상 주당배당금을 사용해서 계산한 것입니다.

그림 19.23에서 2016년에 대한 현금배당수익률은 예측치인데, 이는 추정 실적이 제공된 기업에 한해 제공됩니다. 따라서 추정 실적이 있는 경우에는 그림 19.23의 예상 현금배당수익률을 사용하고, 추정 실적이 제공되지 않는 경우에는 그림 19.22에서 제공되는 현금배당수익률을 사용하는 것이 좋을 것 같습니다.

주요재무정보	연간							
	2011/12 (IFRS별도)	2012/12 (IFRS별도)	2013/12 (IFRS별도)	2014/12 (IFRS별도)	2015/12 (IFRS별도)	2016/12(E) (IFRS별도)	2017/12(E) (IFRS별도)	2018/12(E) (IFRS별도)
매출액	660	752	806	934	995	1,165	1,296	1,438
영업이익	234	277	288	327	360	424	474	529
세전계속사업이익	241	308	323	384	407	477	525	568
당기순이익	193	249	262	309	326	382	422	461
EPS(원)	1,264	1,631	1,718	2,025	2,141	2,507	2,771	3,026
PER(배)	8.92	11.43	13.57	18.91	21.91	16.11	14.58	13.35
BPS(원)	6,850	7,978	9,148	10,340	11,792	13,093	14,812	16,637
PBR(배)	1.65	2.34	2.55	3.70	3.98	3.09	2.73	2.43
현금DPS(원)	474	526	550	700	800	875	938	988
현금배당수익률	4.20	2.82	2.36	1.83	1.71	2.17	2.32	2.44
현금배당성향(%)	37.48	32.27	32.02	34.27	37.04	0.35	0.34	0.33
발행주식수(보통주)	15,242,370	15,242,370	15,242,370	15,242,370	15,242,370			

* 재무실적 업데이트는 검수 및 주석반영으로 공시 이후 약 일주일 정도 소요되며, 기업별로 다르게 적용됨
* 분기 ROE/ROA/PER/PBR은 직전 4개 분기 데이터로 연환산 하여 산출

그림 19.23 기업의 주요 재무 정보

3년 만기 국채 수익률의 일별 시세는 다음 URL을 통해 확인할 수 있습니다. 웹 브라우저를 통해 들어가보면 그림 19.24와 같이 일별 시세가 제공되는 것을 확인할 수 있습니다.

- http://info.finance.naver.com/marketindex/interestDailyQuote.nhn?marketindexCd=IRR_GOVT03Y&page=1[4]

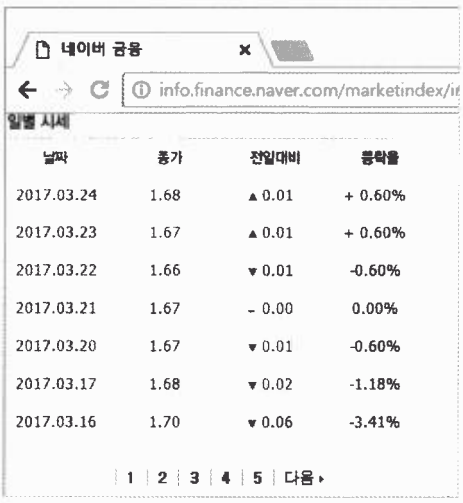


그림 19.24 3년 만기 국채 수익율(일별 시세)

4 단축 URL: https://goo.gl/oF6cqm

먼저 전일자 보통주 수정주가를 사용한 현금배당수익율을 파싱하는 함수를 구현해 보겠습니다. 크롬 웹 브라우저에서 그림 19.22와 같이 기업 현황 페이지로 이동한 후 마우스 오른쪽 버튼을 클릭해 [페이지 소스 보기]를 선택합니다.

그림 19.25와 같이 HTML 코드에서 현금배당수익률이라는 단어를 검색하면 관련 HTML 코드가 표시됩니다. 앞서 <td> … </td> 태그가 표의 열을 의미한다고 했는데, 현금배당수익률을 포함하는 <td> 태그는 <td class="cmp-table-cell td0301">임을 확인할 수 있습니다.

```
<tr class="cmp-table-row">
    <td class="cmp-table-cell td0301">
        <dl>
            <dt>EPS <b class="num">2,141</b></dt>
            <dt class="line-left">BPS <b class="num">11,792</b></dt>
            <dt class="line-left">PER <b class="num">19.15</b></dt>
            <dt class="line-left">업종PER <b class="num">11.31</b></dt>
            <dt class="line-left">PBR <b class="num">3.48</b></dt>
            <dt class="line-left">현금배당수익률 <b class="num">1.95%</b></dt>
            <dt class="right">결산기 : 12월</dt>
        </dl>
    </td>
</tr>
```

그림 19.25 현금배당수익률 관련 HTML 코드

먼저 HTML 코드를 가져온 후 BeautifulSoup의 find_all 메서드를 통해 원하는 태그를 가져옵니다. 참고로 함수의 이름은 get_dividend_yield이고 webreader.py 파일에 구현하겠습니다.

```
def get_dividend_yield(code):
    url = "http://companyinfo.stock.naver.com/company/c1010001.aspx?cmp_cd=" + code
    html = requests.get(url).text

    soup = BeautifulSoup(html, 'lxml')
    td_data = soup.find_all('td', {'class': 'cmp-table-cell td0301'})
    dt_data = td_data[0].find_all('dt')
```

그림 19.25를 참조하면 <td> … </td> 안에는 7개의 <dt> … </dt> 태그가 있습니다. 현금배당수익률은 6번째에 위치하므로 [5]로 인덱싱해서 데이터에 접근할 수 있습니다. 그런 다음 가져온 텍스트에서 현금배당수익률의 수치만 추출하도록 구현합니다. 완성된 함수는 다음과 같습니다

```
def get_dividend_yield(code):
    url = "http://companyinfo.stock.naver.com/company/c1010001.aspx?cmp_cd=" + code
    html = requests.get(url).text

    soup = BeautifulSoup(html, 'lxml')
    td_data = soup.find_all('td', {'class': 'cmp-table-cell td0301'})
    dt_data = td_data[0].find_all('dt')
```

```
    dividend_yield = dt_data[5].text
    dividend_yield = dividend_yield.split(' ')[1]
    dividend_yield = dividend_yield[:-1]

    return dividend_yield
```

구현된 함수의 동작을 테스트하기 위해 webreader.py 파일의 끝부분을 다음과 같이 수정한 후 Ctrl + F9를 눌러 프로그램을 실행합니다.

```
if __name__ == "__main__":
    dividend_yield = get_dividend_yield('058470')
    print(dividend_yield)
```

그림 19.22에서 현금배당수익률은 1.95%이고, 정상적으로 1.95를 가져온 것을 확인할 수 있습니다. 참고로 현금배당수익률은 거래일마다 변경되므로 여러분이 실행했을 때는 다른 값이 출력될 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/webreader.py
1.95

Process finished with exit code 0
```

이번에는 19.2.2절에서 파싱한 주요 재무 정보에서 당해 연도의 현금배당수익률을 가져오는 함수를 구현하겠습니다. webreader.py 파일에 get_estimated_dividend_yield라는 이름의 함수를 추가합니다. 앞서 구현한 get_financial_statement 함수를 호출해 재무제표를 DataFrame 객체 형태로 가져옵니다.

```
def get_estimated_dividend_yield(code):
    df = get_financial_statements(code)
```

그림 19.26을 참조하면 '현금배당수익률'은 ix 메서드를 통해 DataFrame 객체의 로우에 접근해서 구할 수 있습니다. 로우 데이터는 pandas의 Series 객체이므로 인덱스를 통해 각 연도별 현금배당수익을 얻을 수 있습니다. 프로그램이 동작할 때 당해 연도를 자동으로 알아내기 위해 datetime 모듈을 사용합니다. 만약 당해 연도에 현금배당수익률이 존재하지 않는 경우에는 이전 연도의 현금배당수익률을 반환하도록 구현합니다. 참고로 데이터가 존재하지 않을 때는 NaN을 돌려주는데, 이를 확인하기 위해 Numpy 모듈의 isnan 메서드를 사용했습니다.

```
def get_estimated_dividend_yield(code):
    df = get_financial_statements(code)
    dividend_yield = df.ix["현금배당수익률"]
```

```
now = datetime.datetime.now()
cur_year = now.year

if str(cur_year) in dividend_yield.index:
    cur_year_dividend_yield = dividend_yield[str(cur_year)]
    if np.isnan(cur_year_dividend_yield):
        return dividend_yield[str(cur_year-1)]
    else:
        return cur_year_dividend_yield
else:
    return dividend_yield[str(cur_year-1)]
```

주요재무정보

주요재무정보	2012/12 (IFRS별도)	2013/12 (IFRS별도)	2014/12 (IFRS별도)	2015/12 (IFRS별도)	2016/12 (IFRS별도)	2017/12(E) (IFRS별도)	2018/12(E) (IFRS별도)	2019/12(E) (IFRS별도)
매출액	752	806	934	995	1,128	1,282	1,435	
영업이익	277	288	327	360	393	463	517	
세전계속사업이익	308	323	384	407	459	513	567	
당기순이익	249	262	309	326	354	402	444	
당기순이익(지배)	249	262	309	326	354	402	444	
당기순이익(비지배)								
자산총계	1,303	1,479	1,666	1,883	2,167	2,376	2,624	
부채총계	87	84	103	102	158	171	175	
자본총계	1,216	1,394	1,563	1,782	2,009	2,208	2,449	
자본총계(지배)	1,216	1,394	1,563	1,782	2,009	2,208	2,449	
자본총계(비지배)								
자본금	40	76	76	76	76	79	79	
영업활동현금흐름	270	254	387	293	416	374	433	
투자활동현금흐름	-190	-185	-250	-191	-26	-199	-158	
재무활동현금흐름	-72	-80	-132	-108	-121	-92	-137	
CAPEX	134	155	160	133	116	124	126	
FCF	136	98	227	160	299	390	470	
이자발생부채								
영업이익률	36.88	35.75	35.02	36.20	34.86	36.09	36.00	
순이익률	33.05	32.47	33.04	32.80	31.38	31.36	30.96	
ROE(%)	22.00	20.06	20.88	19.52	18.68	19.11	19.08	
ROA(%)	20.50	18.82	19.63	18.39	17.48	17.79	17.77	
부채비율	7.18	6.05	6.60	5.72	7.86	7.76	7.15	
자본유보율	2,932.28	1,732.64	2,020.50	2,303.21	2,602.00			
EPS(원)	1,631	1,718	2,025	2,141	2,322	2,637	2,915	
PER(배)	11.43	13.57	18.91	21.91	18.67	15.75	14.25	
BPS(원)	7,978	9,148	10,340	11,792	13,299	14,617	16,208	
PBR(배)	2.34	2.55	3.70	3.98	3.26	2.84	2.56	
현금DPS(원)	526	550	700	800	900	900	933	
현금배당수익률	2.82	2.36	1.83	1.71	2.07	2.12	2.25	
현금배당성향(%)	32.27	32.02	34.27	37.04	38.41	0.34	0.32	
발행주식수(보통주)	15,242,370	15,242,370	15,242,370	15,242,370	15,242,370			

그림 19.26 재무제표의 표 구조(종목코드: 058470)

datetime과 numpy 모듈을 임포트합니다.

```
import datetime
import numpy as np
```

구현된 함수가 제대로 동작하는지 테스트하기 위해 webreader.py 파일의 끝부분을 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    estimated_dividend_yield = get_estimated_dividend_yield('058470')
    print(estimated_dividend_yield)
```

webreader.py 파일을 실행하면 다음과 같이 2.17이라는 데이터가 출력되는데, 그림 19.26에 나온 2017년의 현금배당수익률 값과 일치하는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/webreader.py
2.17

Process finished with exit code 0
```

이번에는 3년 만기 국채 수익률의 일별 시세를 파싱하는 함수를 구현하겠습니다. 그림 19.24에 나온 웹 페이지의 소스코드를 참조하면 최근 날짜의 시세는 표에서 첫 번째 행의 두 번째 열에서 구할 수 있음을 확인할 수 있습니다. webreader.py 파일에 get_current_3year_treasury라는 함수를 구현한 후 BeautifulSoup를 사용해 데이터를 파싱합니다.

```
def get_current_3year_treasury():
    url = "http://info.finance.naver.com/marketindex/interestDailyQuote.nhn?marketindexCd=IRR_
GOVT03Y&page=1"
    html = requests.get(url).text

    soup = BeautifulSoup(html, 'lxml')
    tbody_data = soup.find_all('tbody')
    tr_data = tbody_data[0].find_all('tr')
    td_data = tr_data[0].find_all('td')
    return td_data[1].text
```

구현한 함수를 테스트하기 위해 역시 webreader.py 파일의 끝부분을 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    print(get_current_3year_treasury())
```

webreader.py 파일을 실행하면 다음과 같이 1.68이라는 값이 출력됩니다. 이 값이 그림 19.24에서 첫 번째 행에 위치하는 종가와 일치하는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/webreader.py
1.68

Process finished with exit code 0
```

이번에는 그림 19.26의 재무제표에서 과거 5년치에 대한 현금배당수익률을 가져오는 함수를 구현해 보겠습니다. 함수의 이름은 get_previous_dividend_yield이고 이번에도 webreader.py 파일에 구현합니다. 먼저 get_financial_statements 함수를 호출해 code에 해당하는 종목의 재무제표를 DataFrame 객체로 가져옵니다. 그리고 ix 메서드를 이용해 '현금배당수익률'에 해당하는 로우 데이터를 가져옵니다.

```
def get_previous_dividend_yield(code):
    df = get_financial_statements(code)
    dividend_yields = df.ix['현금배당수익률']
```

현재 네이버는 과거 5년 치 재무제표 정보를 제공하므로 다음과 같이 현재 연도를 기준으로 5년 치의 현금배당수익률을 파이썬 딕셔너리 형태로 가공합니다. 이때 최근에 상장된 기업들은 최근 5년 치 데이터가 모두 존재하지는 않을 수 있으므로 다음과 같이 특정 연도가 인덱스에 존재할 때만 데이터를 가져오도록 코드를 작성합니다.

```
def get_previous_dividend_yield(code):
    df = get_financial_statements(code)
    dividend_yield = df.ix['현금배당수익률']

    now = datetime.datetime.now()
    cur_year = now.year

    previous_dividend_yield = {}

    for year in range(cur_year-5, cur_year):
        if str(year) in dividend_yield.index:
            previous_dividend_yield[year] = dividend_yield[str(year)]

    return previous_dividend_yield
```

구현한 함수를 테스트하기 위해 다음과 같이 코드를 변경합니다.

```
if __name__ == "__main__":
    print(get_previous_dividend_yield('058470'))
```

webreader.py 파일을 실행해보면 다음과 같이 2011~2015년도의 현금배당수익률을 잘 가져왔음을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/webreader.py
{2016: 2.0699999999999998, 2012: 2.8199999999999998, 2013: 2.3599999999999999, 2014:
1.8300000000000001, 2015: 1.71}

Process finished with exit code 0
```

배당률 기반 투자 알고리즘 구현(2)

국채시가배당률 기반 매매 알고리즘을 구현하는 데 필요한 데이터를 가져오는 기능은 앞서 webreader.py 파일에 구현했습니다. 이번에는 pymon.py 파일에서 webreader 모듈을 사용해 알고리즘을 구현해 보겠습니다.

먼저 현재 거래일 기준으로 예상 현금배당수익률(또는 현금배당수익률)과 3년 만기 국채수익률의 일별 시세를 가져와 국채시가배당률을 계산하는 메서드를 pymon.py 파일에 구현하겠습니다. 예상 현금배당수익률은 webreader 모듈의 get_estimated_dividend_yield 함수를 호출해서 구할 수 있고 현금배당수익률은 get_dividend_yield 함수를 사용하면 됩니다. 마지막으로 3년 만기 국채수익률은 get_current_3year_treasury 함수를 통해 구하면 됩니다.

예상 현금배당수익률이 존재하는 종목은 해당 값을 사용하고, 그렇지 않은 종목은 현금배당수익률을 사용합니다. 국채시가배당률은 현금배당수익률을 3년 만기 국채수익률로 나눈 값인데 함수를 호출해 가져온 값이 문자열이므로 실수형으로 형을 변환한 후 나누면 됩니다.

```
def calculate_estimated_dividend_to_treasury(self, code):
    estimated_dividend_yield = webreader.get_estimated_dividend_yield(code)
    if np.isnan(estimated_dividend_yield):
        estimated_dividend_yield = webreader.get_dividend_yield(code)

    current_3year_treasury = webreader.get_current_3year_treasury()
    estimated_dividend_to_treasury = float(estimated_dividend_yield) / float(current_3year_trea-
sury)
    return estimated_dividend_to_treasury
```

pymon.py에서 다음과 같이 모듈을 임포트합니다.

```
import datetime
import webreader
import numpy as np
```

구현한 메서드의 동작 여부를 확인하기 위해 pymon.py 파일의 끝부분을 다음과 같이 변경합니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    pymon = PyMon()
    #pymon.run()
    print(pymon.calculate_estimated_dividend_to_treasury('058470'))
```

pymon.py 파일을 실행해 보면 다음과 같이 1.29166이라는 값이 출력되는 것을 확인할 수 있습니다. 그림 19.26에 나온 2017년의 예상 현금배당수익률 2.17을 그림 19.24에 나온 최근 일별 시세인 1.68로 나누면 1.29166입니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/pymon.py
connected
1.2916666666666667

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

Process finished with exit code 0
```

이번에는 최근 5년에 대한 국채시가배당률 중 최댓값과 최솟값을 반환하는 함수를 구현하겠습니다. 리노공업의 최근 5년의 시가배당률과 해당 연도의 3년 국채 수익률은 표 19.1과 같습니다. 국채시가배당률의 최댓값은 1.26이고 최솟값은 0.83입니다.

표 19.1 리노공업 국채시가배당률표

	2012년	2013년	2014년	2015년	2016년
시가배당률	2.82%	2.36%	1.83%	1.71%	2.07%
국고채 3년(평균)	2.82%	2.86%	2.10%	1.66%	1.64%
국채시가배당률	1.00	0.83	0.87	1.03	1.26

먼저 webreader 모듈 내의 함수를 사용해 최대 5년 치에 대한 시가배당률과 연도별 3년 만기 국채 수익률 데이터(1998년~2016년)를 얻어옵니다.

```
def get_min_max_dividend_to_treasury(self, code):
    previous_dividend_yield = webreader.get_previous_dividend_yield(code)
    three_years_treasury = webreader.get_3year_treasury()
```

각 연도별 국채시가배당률을 저장할 파이썬 딕셔너리 객체를 생성합니다. 각 연도에 대해 국채시가배당률을 계산한 후 딕셔너리에 추가합니다. 이때 딕셔너리의 키는 연도이고 값은 해당 연도의 국채시가배당률입니다. 일부 종목은 과거 연도의 데이터가 존재하지 않을 수 있기 때문에 if 문을 사용해 먼저 해당 연도가 딕셔너리의 키 값에 존재하는지 확인합니다.

```
def get_min_max_dividend_to_treasury(self, code):
    # 코드 생략
    now = datetime.datetime.now()
    cur_year = now.year
    previous_dividend_to_treasury = {}

    for year in range(cur_year-5, cur_year):
        if year in previous_dividend_yield.keys() and year in three_years_treasury.keys():
            ratio = float(previous_dividend_yield[year]) / float(three_years_treasury[year])
            previous_dividend_to_treasury[year] = ratio
```

계산된 국채시가배당률을 임시로 출력하는 print 문을 추가하고 5년 기간의 국채시가배당률의 최댓값과 최솟값을 계산한 후 이를 반환하도록 구현합니다.

```
def get_min_max_dividend_to_treasury(self, code):
    # 코드 생략
    print(previous_dividend_to_treasury)
    min_ratio = min(previous_dividend_to_treasury.values())
    max_ratio = max(previous_dividend_to_treasury.values())

    return (min_ratio, max_ratio)
```

구현한 함수를 테스트하기 위해 pymon.py 파일의 끝부분에 있는 코드를 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    pymon = PyMon()
    print(pymon.get_min_max_dividend_to_treasury('058470'))
```

pymon.py 파일을 실행해 보면 다음과 같이 2012~2016년도 동안 리노공업 종목에 대한 국채시가배당률 값이 출력되는 것을 확인할 수 있습니다. 출력된 값을 표 19.1의 값과 비교해보면 국채시가배당률 값이 정확히 계산됐음을 알 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/pymon.py
connected
{2016: 1.2621951219512195, 2012: 1.0, 2013: 0.8251748251748252, 2014: 0.8714285714285714, 2015:
1.0301204819277108}
(0.8251748251748252, 1.2621951219512195)

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

Process finished with exit code 0
```

이번에는 현재 시점을 기준으로 계산한 국채시가배당률이 과거 5년 치 국채시가배당률의 최댓값보다 큰 경우 해당 종목을 매수하는 알고리즘을 구현해 보겠습니다.

PyMon 클래스에 buy_check_by_dividend_algorithm 메서드를 추가합니다. 해당 메서드에서는 현시점에서의 예상 국채시가배당률을 calculate_estimated_dividend_to_treasury 메서드를 호출해서 가져오고, 과거 5년에 대한 최대, 최소 국채시가배당률은 get_min_max_dividend_to_treasury 메서드를 호출해서 가져옵니다.

if 문을 사용해 현시점의 예상 국채시가배당률이 과거 5년에 대한 국채시가배당률의 최댓값보다 크거나 같은 경우 '(1, 예상 국채시가배당률)'의 튜플을 리턴합니다.

```
def buy_check_by_dividend_algorithm(self, code):
    estimated_dividend_to_treasury = self.calculate_estimated_dividend_to_treasury(code)
    (min_ratio, max_ratio) = self.get_min_max_dividend_to_treasury(code)

    if estimated_dividend_to_treasury >= max_ratio:
        return (1, estimated_dividend_to_treasury)
    else:
        return (0, estimated_dividend_to_treasury)
```

구현한 메서드를 테스트하기 위해 pymon.py 파일의 끝부분을 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    pymon = PyMon()
    print(pymon.buy_check_by_dividend_algorithm('058470'))
```

pymon.py 파일을 실행하면 다음과 같이 (1, 1.292)가 출력되는 것을 확인할 수 있습니다. 반환된 튜플의 첫 번째 원소가 1이므로 이는 매수를 의미합니다. 예상 국채시가배당률(1.29166)이 5년 동안의 국

채시가배당률의 최댓값(1.262)보다 크기 때문에 정상적으로 매수 신호가 발생한 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/pymon.py
connected
{2016: 1.2621951219512195, 2012: 1.0, 2013: 0.8251748251748252, 2014: 0.8714285714285714, 2015:
1.0301204819277108}
(1, 1.2916666666666667)

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

Process finished with exit code 0
```

배당률 기반 투자 알고리즘 구현(3)

19.2.5절을 통해 국채시가배당률 알고리즘을 기반으로 종목의 매수 여부를 결정하는 메서드를 구현했습니다. 이번에는 유가증권시장의 전 종목에 대해 매수 여부를 결정하는 메서드를 구현하겠습니다.

유가증권시장의 전 종목에 대해 국채시가배당률 기반으로 매수 여부를 체크하는 메서드는 run_dividend입니다. PyMon 클래스는 생성자에서 유가증권시장과 코스닥시장에 대한 종목코드를 가져옵니다. 따라서 반복문에서 self.kospi_codes의 종목을 하나씩 가져온 후 buy_check_by_dividend_algorithm 메서드를 호출하면 됩니다. 이때 반환값인 튜플의 첫 번째 원소가 1이면 buy_list에 추가합니다.

```
def run_dividend(self):
    buy_list = []

    for code in self.kospi_codes:
        print('Check: ', code)
        ret = self.buy_check_by_dividend_algorithm(code)

        if ret[0] == 1:
            print("Pass", ret)
            buy_list.append((code, ret[1]))
        else:
            print("Fail", ret)
```

화면에 출력되는 메시지를 최소화하기 위해 앞서 get_min_max_dividend_to_treasury 메서드에 넣었던 print 구문을 삭제합니다. 구현한 run_dividend 메서드를 테스트하기 위해 pymon.py 파일의 끝부분을 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    pymon = PyMon()
    pymon.run_dividend()
```

pymon.py 파일을 실행하면 유가증권시장의 전 종목에 대해 국채시가배당률 알고리즘을 사용해 매수 여부를 체크하게 되는데, 일부 종목에 대해서는 다음과 같이 에러가 발생합니다.

```
Check:  000300
Traceback (most recent call last):
  File "E:/GitHub/book/ch19/day06/pymon.py", line 129, in <module>
    pymon.run_dividend()
  File "E:/GitHub/book/ch19/day06/pymon.py", line 119, in run_dividend
    ret = self.buy_check_by_dividend_algorithm(code)
  File "E:/GitHub/book/ch19/day06/pymon.py", line 105, in buy_check_by_dividend_algorithm
    estimated_dividend_to_treasury = self.calculate_estimated_dividend_to_treasury(code)
  File "E:/GitHub/book/ch19/day06/pymon.py", line 82, in calculate_estimated_dividend_to_trea-
sury
    estimated_dividend_to_treasury = float(estimated_dividend_yield) / float(current_3year_trea-
sury)
ValueError: could not convert string to float:

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12
```

종목코드 000300은 '대유플러스'로 그림 19.27과 같이 현금배당수익률이 존재하지 않는 것을 확인할 수 있습니다.

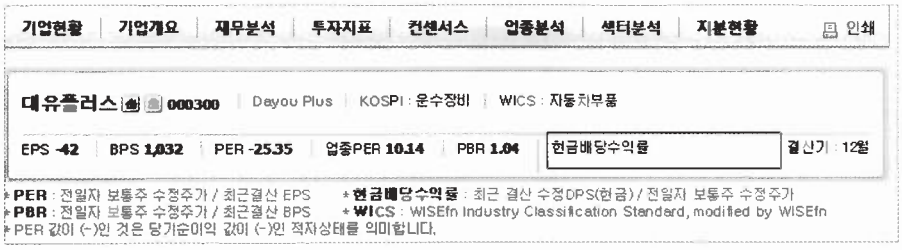


그림 19.27 기업현황(대유플러스/출처: 네이버증권)

현금배당수익률이 존재하지 않는 경우 빈 문자열이 반환되는데 이 경우 현금배당수익률을 0으로 할당합니다.

```
def calculate_estimated_dividend_to_treasury(self, code):
    estimated_dividend_yield = webreader.get_estimated_dividend_yield(code)

    if np.isnan(estimated_dividend_yield):
        estimated_dividend_yield = webreader.get_dividend_yield(code)

        if estimated_dividend_yield == "":
            estimated_dividend_yield = 0
```

pymon.py 파일을 다시 실행하면 이번에는 069500 종목에서 에러가 발생합니다. 069500은 KODEX200 ETF로서 그림 19.28과 같이 재무제표 데이터가 존재하지 않기 때문에 webreader.py 파일의 get_estimated_dividend_yield 함수에서 에러가 발생한 것입니다.

주요재무정보

주요재무정보 연간

매출액
영업이익
세전계속사업이익
당기순이익
당기순이익(지배)
당기순이익(비지배)
자산총계

그림 19.28 주요재무정보(종목코드: 069500)

먼저 get_estimated_dividend_yield 함수를 다음과 같이 수정합니다. 해당 연도 및 이전 연도에 데이터가 존재하는지 확인하기 위해 np.isnan 메서드를 사용했습니다. 예상 현금배당수익률이 존재하지 않는 경우에는 np.NaN을 반환합니다.

```
def get_estimated_dividend_yield(code):
    # 코드 생략
    if str(cur_year) in dividend_yield.index and not np.isnan(dividend_yield[str(cur_year)]):
        return dividend_yield[str(cur_year)]
    elif str(cur_year-1) in dividend_yield.index and not np.isnan(dividend_yield[str(cur_year-
1)]):
        return dividend_yield[str(cur_year-1)]
    else:
        return np.NaN
```

이번에는 현금배당수익률을 파싱하는 get_dividend_yield 함수를 수정하겠습니다. 앞서 현금배당수익률을 그림 19.22와 같이 기업현황 웹 페이지를 통해 얻어왔습니다. 그러나 종목코드 069500은 그림 19.29와 같이 현금배당수익률이 존재하지 않는 것을 알 수 있습니다.

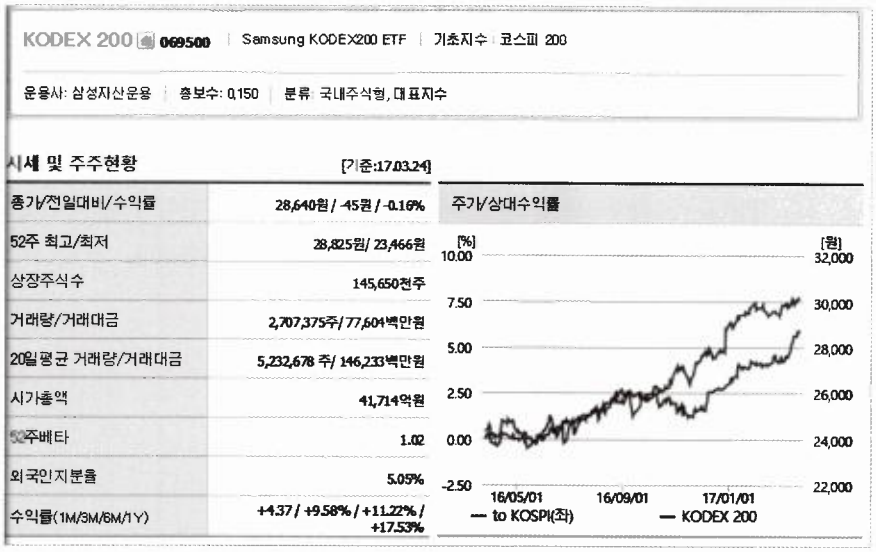


그림 19.29 기업현황(종목코드: 069500)

이러한 문제를 해결하기 위해 get_dividend_yield 코드의 일부를 다음과 같이 수정합니다. HTML 코드에서 특정 td 태그가 존재하지 않는 경우 빈 문자열을 반환하게 했습니다.

```
def get_dividend_yield(code):
    url = "http://companyinfo.stock.naver.com/company/c1010001.aspx?cmp_cd=" + code
    html = requests.get(url).text

    soup = BeautifulSoup(html, 'lxml')
    td_data = soup.find_all('td', {'class': 'cmp-table-cell td0301'})

    if not td_data:
        return ""

    # 코드 생략
```

이번에는 pymon.py 파일에서 buy_check_by_dividend_algorithm 메서드를 수정하겠습니다. 데이터가 존재하지 않는 경우 estimated_dividend_to_treasury와 max_ratio가 0이 됩니다. 이 경우 if 문을 만족해서 매수 신호가 발생하기 때문에 max_ratio !=0 구문을 추가합니다.

```
def buy_check_by_dividend_algorithm(self, code):
    # 코드 생략
    if estimated_dividend_to_treasury >= max_ratio and max_ratio != 0:
        return (1, estimated_dividend_to_treasury)
```

```
        else:
            return (0, estimated_dividend_to_treasury)
```

국채시가배당률 기반 알고리즘은 모든 데이터를 네이버 증권을 통해 얻어옵니다. 이때 너무 짧은 시간에 웹 페이지를 여러 번 읽어오면 네이버 서버에서 연결을 해제합니다. 따라서 run_dividend 메서드의 반복문 내에서 0.5초 정도 쉬었다가 조회하도록 time.sleep(0.5)를 추가합니다. 간단히 구현한 메서드를 테스트하기 위해 유가증권시장의 50개 종목에 대해서만 체크하도록 코드를 수정합니다.

```
def run_dividend(self):
    buy_list = []

    for code in self.kospi_codes[0:50]:
        time.sleep(0.5)
        ret = self.buy_check_by_dividend_algorithm(code)
        if ret[0] == 1:
            buy_list.append((code, ret[1]))
```

국채시가배당률 알고리즘 기반으로 매수 신호가 발생한 경우 종목과 해당 종목에 대한 국채시가배당률이 buy_list에 저장됩니다. 매수 종목이 여러 개인 경우 그중 국채시가배당률이 높은 종목이 더 매수에 적합하므로 다음과 같이 국채시가배당률이 높은 종목을 기준으로 정렬합니다.

```
def run_dividend(self):
    # 코드 생략
    sorted_list = sorted(buy_list, key=lambda t:t[1], reverse=True)
    print(sorted_list)
```

pymon.py 파일을 실행하면 다음과 같이 국채시가배당률이 높은 매수 종목부터 출력되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/ch19/day06/pymon.py
connected
[('000080', 2.761904761904762), ('000087', 2.761904761904762), ('000270', 1.898809523809524),
('000370', 0.9166666666666667), ('000670', 0.6607142857142858)]

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

[GetPCIdentity] VER 3.2.0.0  build 2015.8.12

Process finished with exit code 0
```

이번에는 국채시가배당률이 높은 상위 5개 종목을 buy_list.txt 파일에 추가하겠습니다. 매수 종목 리스트를 buy_list.txt 파일에 출력하는 기능은 update_buy_list 메서드에 이미 구현돼 있습니다. 따라서 sorted_list에서 상위 5개 종목에 대한 종목 코드로 새로운 리스트로 만든 후 해당 리스트를 update_buy_list 메서드의 인자로 전달하면 됩니다.

```
def run_dividend(self):
    # 코드 생략
    buy_list = []
    for i in range(0, 5):
        code = sorted_list[i][0]
        buy_list.append(code)

    self.update_buy_list(buy_list)
```

pymon.py 파일을 실행하면 그림 19.30과 같이 매수 종목 리스트 파일이 생성되는 것을 확인할 수 있습니다.

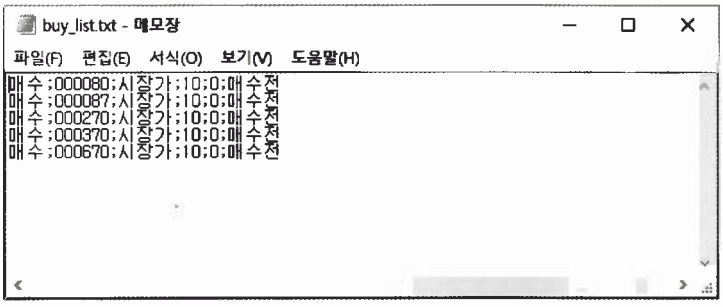


그림 19.30 국채시가배당률 기반 매수 종목

지금까지 국채시가배당률이라는 알고리즘을 파이썬 코드로 구현해 봤습니다. 아이디어는 간단하지만 이를 파이썬으로 구현하는 과정에서 많은 노력이 필요하다는 사실을 알 수 있습니다. 현재는 단순히 국채시가배당률이 높은 종목 위주로 매수 리스트를 구성했지만, 당기순이익이나 다른 재무제표 항목도 참고하면 더욱 좋은 알고리즘을 만들 수 있을 것입니다.

매도 역시 매수와 같은 방식으로 구현할 수 있습니다. 다만 매도는 유가증권시장과 코스닥시장의 전 종목에 대해 매도 조건을 만족하는지 확인하는 것이 아니라 현재 보유하고 있는 종목에 대해서만 체크하면 됩니다. 따라서 증권사 API를 통해 먼저 보유종목 리스트를 구성한 후 매수 종목 체크와 비슷하게 구현하면 됩니다.

부록

A

소스코드 다운로드 및 실행

본문에서 사용된 소스코드는 깃허브(GitHub) 사이트를 통해 제공되고 있습니다. 이번 절에서는 깃허브 사이트에서 소스코드를 다운로드하는 방법을 알아보겠습니다. 먼저 다음 웹사이트로 이동합니다.

- https://github.com/pystockhub/book

그림 A.1과 같이 [Clone or Download] 버튼을 누른 후 [Download ZIP] 메뉴를 선택하면 전체 소스코드를 압축 파일로 받을 수 있습니다.

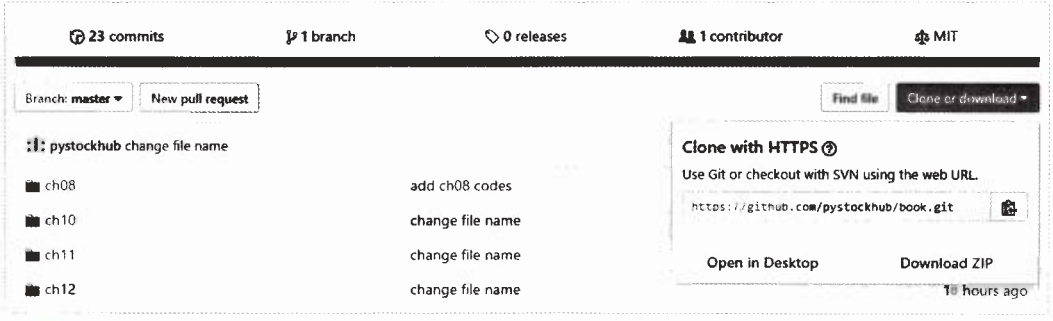


그림 A.1 깃허브 저장소

압축 프로그램으로 앞에서 내려받은 book-master.zip 파일의 압축을 풀면 그림 A.2와 같이 각 장별로 구성된 소스코드를 확인할 수 있습니다.

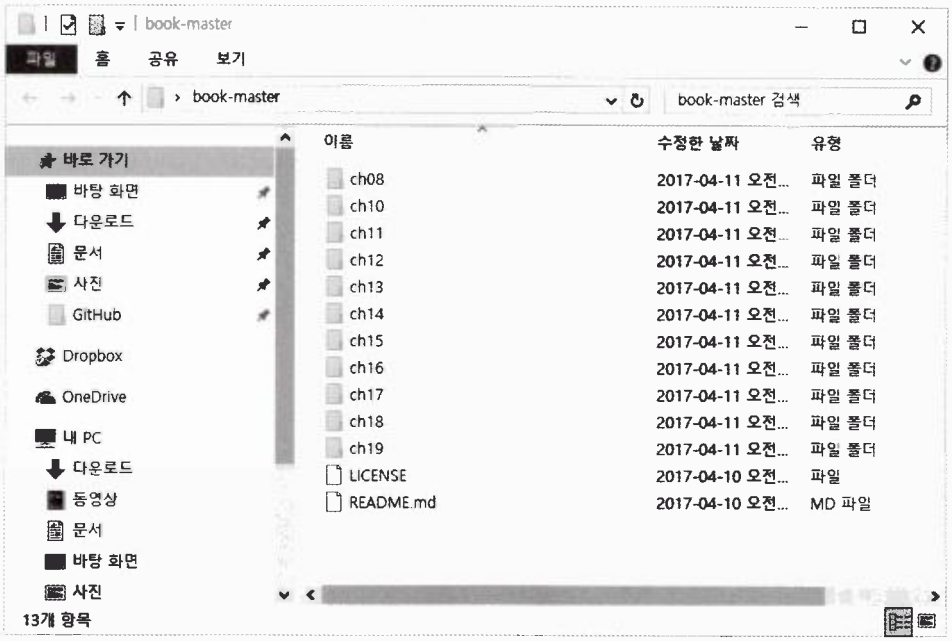


그림 A.2 본문 소스코드

소스코드를 실행해보기 위해 PyCharm을 실행합니다. 그림 A.3에서 [Open] 메뉴를 클릭합니다.

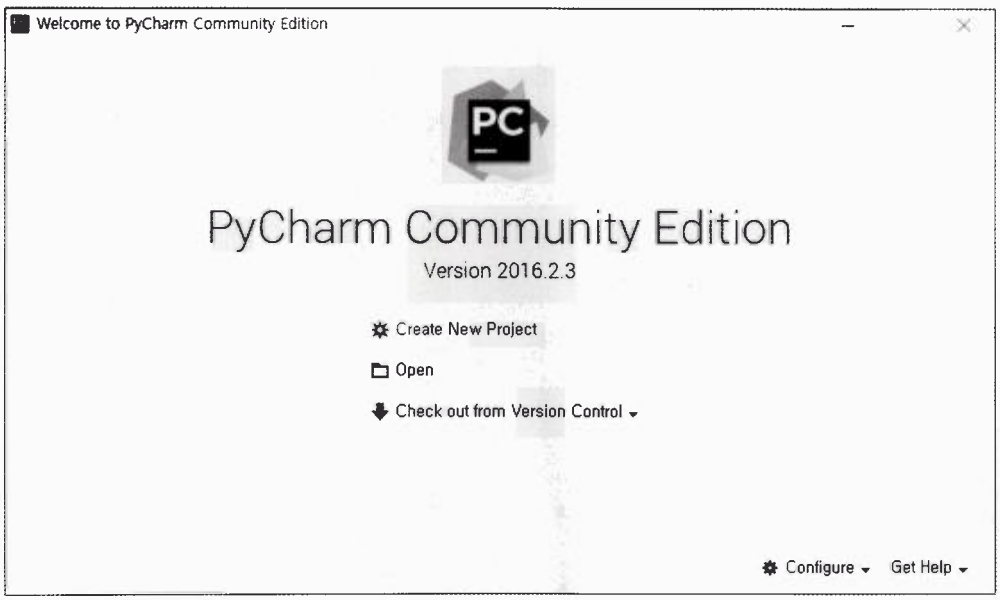


그림 A.3 PyCharm 시작 화면

그림 A.4와 같이 [Open File or Project] 창에서 앞서 압축을 해제한 'book-master' 디렉터리를 선택한 후 [OK] 버튼을 클릭합니다.

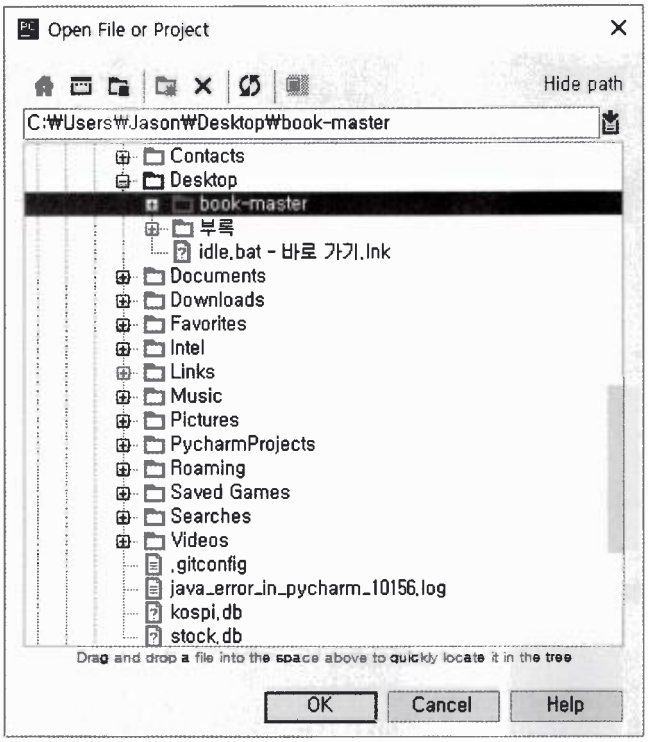


그림 A.4 book-master 디렉터리 열기

그림 A.5와 같이 PyCharm 프로젝트 윈도우를 통해 각 장별로 구성된 파이썬 파일을 확인할 수 있습니다. 실행하려는 파이썬 파일에 마우스 오른쪽 버튼을 클릭한 후 [Run] 메뉴를 선택하면 해당 파일이 실행됩니다.

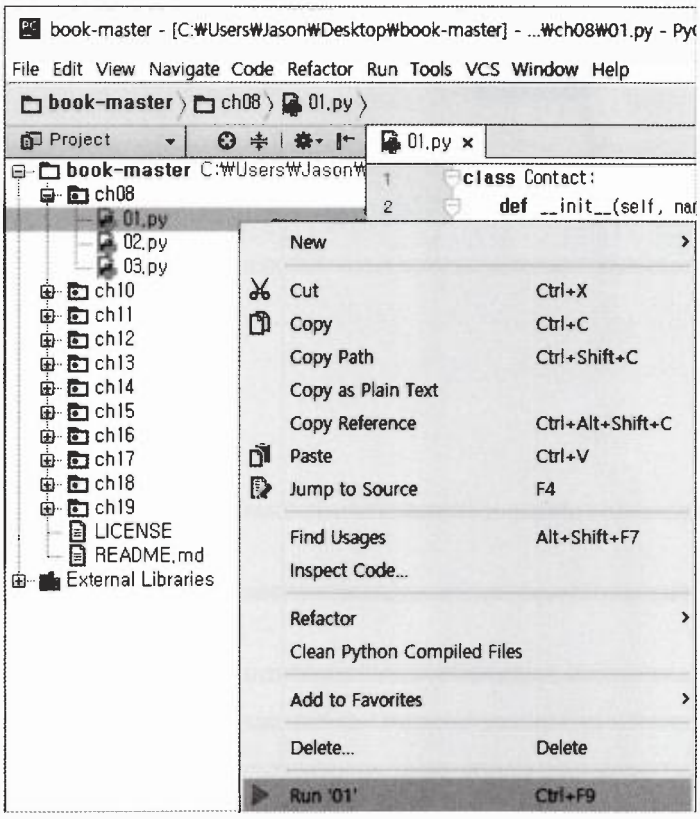


그림 A.5 파이썬 파일 실행

부록

B

PyCharm을 이용한 디버깅

컴퓨터 프로그램이 예상대로 동작하지 않고 잘못된 결과를 내는 것을 버그(bug)라고 합니다. 프로그래머는 프로그램을 작성한 후 프로그램이 올바르게 동작하는지 확인해야 하고 만약 프로그램에 버그가 있다면 이를 찾아내서 없애야 합니다. 이처럼 버그를 찾고 해결하는 작업을 디버깅(debugging)이라 합니다. 프로그램을 구현하는 것도 중요하지만 버그가 있을 때 이를 빨리 찾아서 해결하는 것도 매우 중요합니다. 이번 절에서는 PyCharm을 이용해 프로그램을 디버깅하는 방법을 알아보겠습니다.

01 브레이크포인트 설정

파이썬 코드는 기본적으로 위에서부터 아래로 순차적으로 실행됩니다. 브레이크포인트(breakpoint)는 코드의 실행을 멈추는 데 사용됩니다. 버그가 있을 것 같은 코드에 브레이크포인트를 설정한 후 디버거를 실행하면 브레이크포인트가 설정되기 이전까지의 코드가 실행되고 브레이크포인트가 설정된 코드에서 프로그램 실행을 멈추게 됩니다. 프로그래머는 코드 실행이 멈춰진 상태에서 변수가 바인딩하는 값을 보고 버그를 쉽게 찾을 수 있습니다.

본격적인 디버깅에 앞서 PyCharm의 Keymap 설정을 확인해 보겠습니다. PyCharm을 실행한 후 [File] → [Settings] 메뉴를 차례로 선택합니다. 그림 B.1의 왼쪽 메뉴에서 [Keymap]을 클릭한 후 오른쪽의 [Keymaps] 부분을 'Visual Studio'로 설정하면 됩니다.

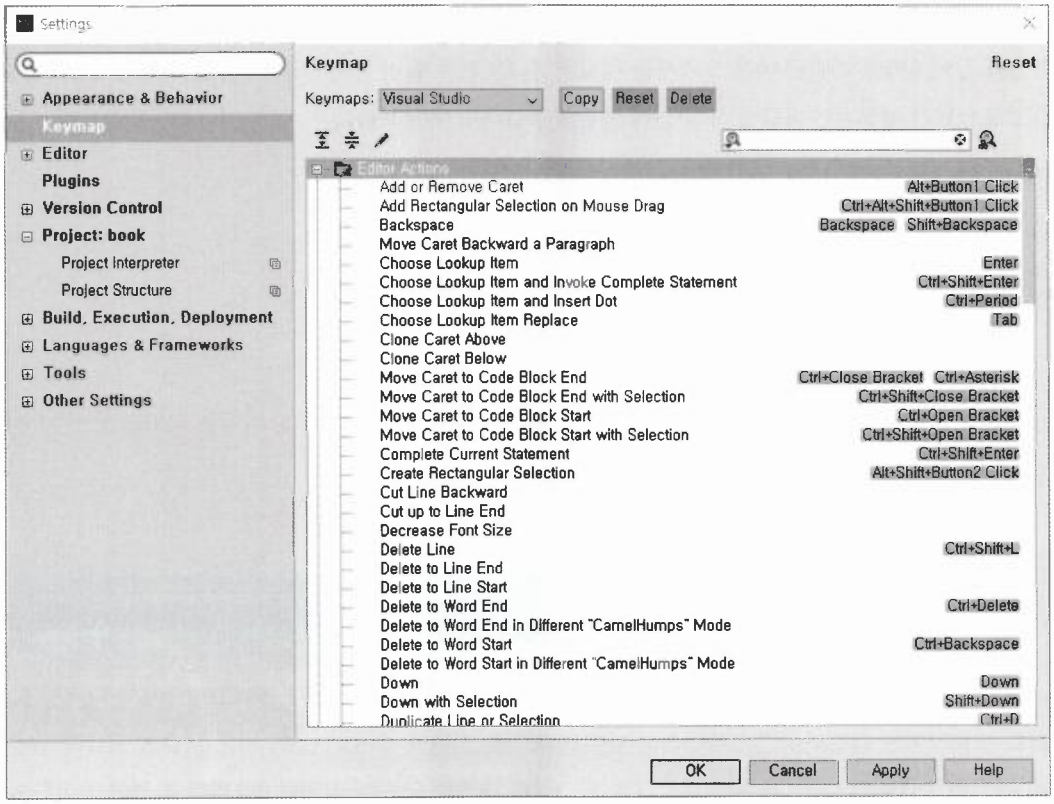


그림 B.1 Keymap 설정

지금부터는 간단한 코드를 사용해 브레이크포인트를 설정해보겠습니다. 디버깅에 사용할 코드는 다음과 같습니다.

```
a = 3
b = 4
c = a + b + 1

print("3+4=", c)
```

PyCharm에서 프로젝트를 생성한 후 파이썬 파일을 추가하고 위 코드를 작성합니다. 위 코드를 실행하면 다음과 같이 3+4의 결과가 7이 아니고 8로 출력되는 것을 확인할 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/appendix/01.py
3+4= 8

Process finished with exit code 0
```

프로그램을 디버깅하려면 먼저 에러가 있을 듯한 코드에 브레이크포인트를 설정해야 합니다. 그림 B.2와 같이 3번째 라인에 마우스 커서를 위치한 후 F9 키를 눌러 브레이크포인트를 설정합니다. 브레이크포인트가 설정되면 해당 코드 라인에 빨간색 동그라미가 표시됩니다.

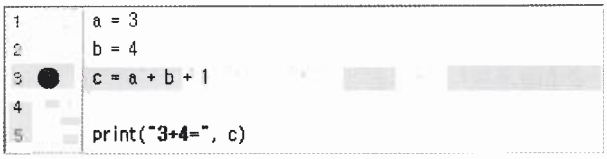


그림 B.2 브레이크포인트 설정

브레이크포인트를 제거하려면 브레이크포인트가 설정된 코드 라인에 마우스 커서를 위치한 후 다시 한 번 F9 키를 누르면 됩니다.

02 디버깅 모드로 프로그램 실행

브레이크포인트를 설정했다면 디버깅 모드로 프로그램을 실행하면 됩니다. PyCharm에서 디버깅 모드로 프로그램을 실행하려면 실행하려는 파이썬 파일을 선택한 후 Alt + F5 키를 누르면 됩니다. 3번째 라인에 브레이크포인트를 설정했으므로 1, 2번째 라인의 코드는 실행되고 3번째 라인에서 코드의 실행이 멈추게 됩니다.

그림 B.3을 참조하면 3번째 라인이 음영으로 표시된 것과 1, 2번째 라인의 코드 옆에 a와 b 변수가 바인딩하는 값이 표시된 것을 확인할 수 있습니다. 만일 디버거를 실행한 후의 레이아웃이 그림 B.3과 같지 않다면 PyCharm의 왼쪽 하단부에 있는 [Restore Layout] 버튼을 클릭하면 됩니다.

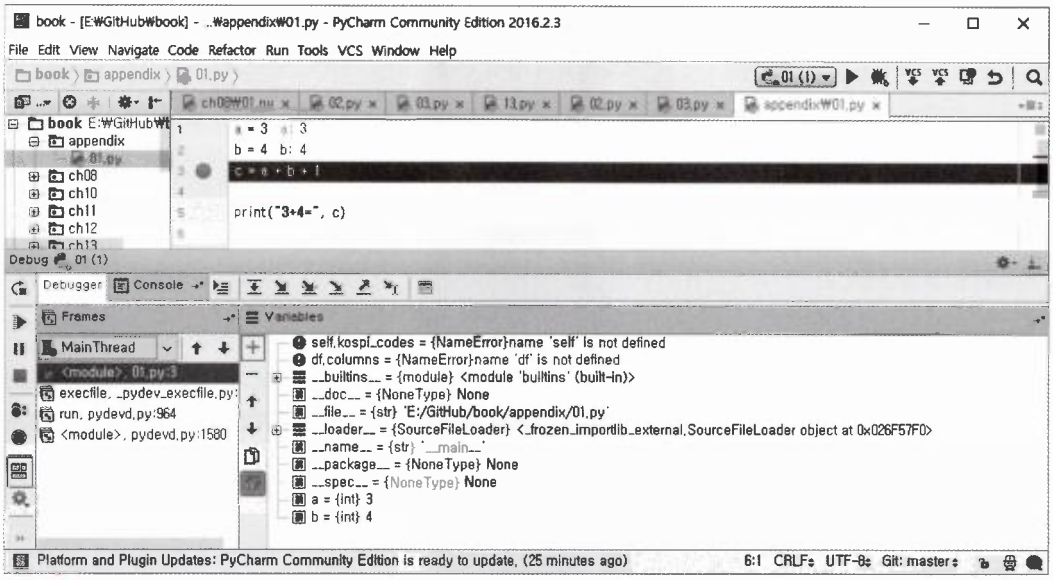


그림 B.3 PyCharm 디버거

그림 B.3을 참조하면 위쪽에는 파이썬 코드가 출력되고, 왼쪽 아래의 Frames 오른쪽 아래에 Variables 창이 출력되는 것을 확인할 수 있습니다. Variables 창에서 '+'를 클릭한 후 c를 입력해 봅시다. 그림 B.4와 같이 'c is not defined'라고 출력되는 것을 확인할 수 있습니다. 이는 현재 소스코드가 3번째 라인에서 멈춰 있어서 아직 c라는 변수가 값을 바인딩하고 있지 않기 때문입니다.



그림 B.4 Variables 창에 변수 추가

이번에는 브레이크포인트로 인해 멈춤 상태에 있는 소스코드를 라인 단위로 실행해보겠습니다. F10 키를 누르면 소스코드가 한 줄 실행됩니다. 그림 B.5를 참조하면 코드에 있던 음영 부분이 5번째 라인으로 옮겨진 것을 확인할 수 있습니다(4번째 라인은 공백). 또한 Variables 창에서 c라는 변수가 8을 바

인딩하고 있음을 확인할 수 있습니다. Variables 창이나 코드 옆에 표시된 값을 참조하면 a, b는 현재 3, 4를 각각 바인딩하고 있습니다. 그런데 a와 b의 합인 c에는 8이 저장돼 있으므로 두 값을 더하는 부분에서 문제가 있음을 쉽게 알 수 있습니다. 코드를 잘 살펴보니 c = a + b + 1에서 +1이라는 코드가 더 해져서 오류가 발생했음을 알 수 있습니다.

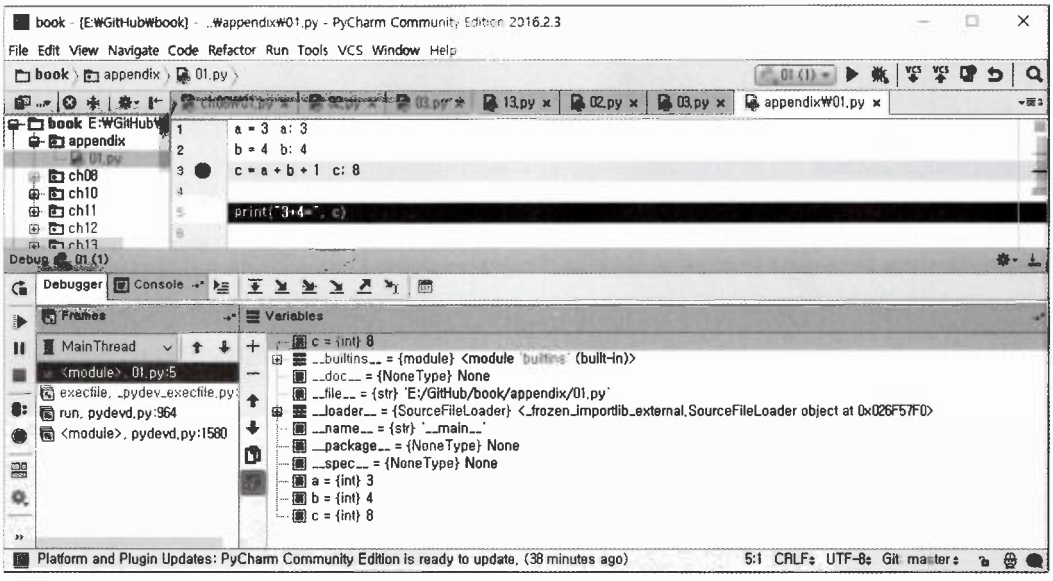


그림 B.5 F10 키를 이용한 코드 실행

이번 예제는 코드도 짧고 간단해서 사실 PyCharm의 디버깅 기능을 사용할 필요도 없이 코드를 리뷰하면서 버그를 찾을 수도 있습니다. 하지만 PyCharm의 디버깅 기능을 이용하면 파이썬 코드가 각 라인 단위로 어떻게 수행되는지 볼 수 있으므로 코드를 더욱더 잘 이해하는 데 도움될뿐더러 버그도 빨리 찾을 수 있습니다.

버그를 찾았다면 현재 디버거 모드에서 빠져나오면 됩니다. 디버거 모드에서 빠져나오려면 Shift + F5 키를 누릅니다. 참고로 브레이크포인트가 있더라도 Ctrl + F5나 Ctrl + F9 키를 사용해 프로그램을 실행하면 브레이크포인트와 상관없이 프로그램이 실행됩니다. 디버깅 모드로 실행하려면 꼭 Alt + F5 키를 사용해야 한다는 점을 기억하기 바랍니다.

03 함수 디버깅하기

이번에는 다음 코드를 통해 함수를 디버깅하는 방법을 배우겠습니다. 일반적으로 작성하는 프로그램의 규모가 커지면 코드의 재사용성을 높이기 위해 함수를 사용합니다.

```
a = 3
b = 4

def hap(a, b):
    return a+b+1

result = hap(a, b)
print(result)
```

PyCharm으로 프로젝트를 생성한 후 파이썬 파일을 하나 추가합니다. 해당 파일에 위의 코드를 입력한 후 Ctrl + F9 키를 누르면 다음과 같이 실행 결과가 출력됩니다. 두 수를 더하는 함수인 hap에 3, 4라는 인자를 전달했는데 결괏값으로 7이 아니라 8이 출력됐으니 프로그램에 버그가 있음을 알 수 있습니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/appendix/02.py
8

Process finished with exit code 0
```

먼저 코드에 브레이크포인트를 설정하겠습니다. 에러가 어디에서 발생했는지 모르겠다면 첫 번째 라인에 브레이크포인트를 설정해서 한 라인씩 디버깅하면 됩니다. 그림 B.6와 같이 첫 번째 라인에 마우스 커서를 위치한 후 F9 키를 눌러 브레이크포인트를 설정합니다.

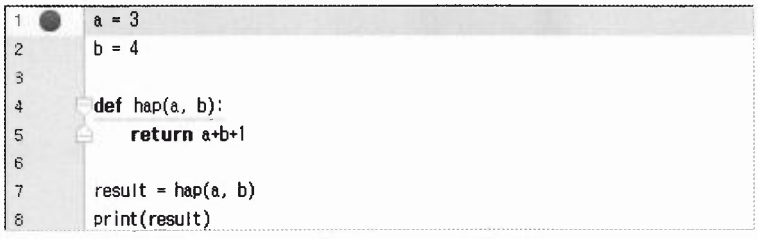


그림 B.6 브레이크포인트 설정

디버깅 모드로 프로그램을 실행하기 위해 Alt + F5 키를 누릅니다. 그림 B.7과 같이 첫 번째 라인에 브레이크포인트가 설정돼 있으므로 파이썬 인터프리터가 파이썬 코드를 실행하자마자 멈춰있는 것을 확인할 수 있습니다.



그림 B.7 디버깅 모드로 프로그램 실행

F10 키를 눌러 라인 단위로 프로그램을 실행합니다. 첫 번째 라인과 두 번째 라인이 실행되면서 a와 b가 3과 4를 각각 바인딩함을 알 수 있습니다. 4~5번째 라인은 함수의 정의부로서 F10 키를 누르면 해당 부분은 건너뛰는 것을 확인할 수 있습니다. F10 키를 눌러 그림 B.8과 같이 7번째 라인의 hap 함수를 호출하는 부분에서 코드가 멈추게 합니다.

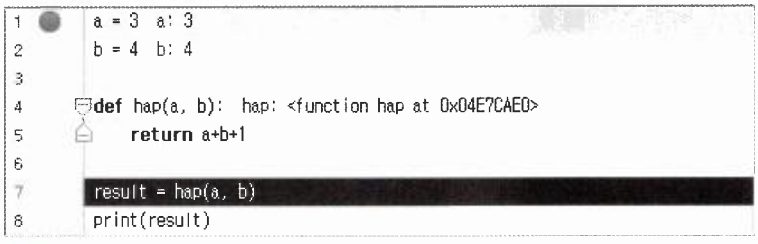


그림 B.8 함수 호출 전

그림 B.8을 참조하면 현재 코드의 실행은 7번째 라인에서 멈춰 있습니다. 여기서 F10 키를 누르면 함수 내부로 들어가지 않고 함수가 실행되고 8번째 라인으로 이동합니다. 이와 달리 F11 키를 누르면 함수 내부로 이동하게 됩니다. 함수 내부를 디버깅해야 하므로 F11 키를 누릅니다.

그림 B.9를 보면 함수의 인자인 a와 b가 함수 호출부에서 넘겨준 값인 3과 4를 각각 바인딩하고 있음을 확인할 수 있습니다. 그리고 코드의 실행은 5번째 라인에서 멈춰 있습니다.

```
1  a = 3
2  b = 4
3
4  def hap(a, b):  a: 3  b: 4
5      return a+b+1
6
7  result = hap(a, b)
8  print(result)
```

그림 B.9 함수 내 디버깅

hap 함수는 두 개의 정숫값을 더하는 기능을 하는데, 5번째 라인의 코드를 보면 +1이라는 코드가 입력된 것을 확인할 수 있습니다. 일단 F10 키를 눌러 5번째 라인을 실행하고 다음에 실행할 라인으로 이동합니다.

함수를 호출하면 함수로 이동했다가 함수가 끝나면 다시 함수의 호출부로 돌아옵니다. 그림 B.10을 보면 다시 7번째 라인에서 코드가 멈춰 있음을 알 수 있습니다. 여기서 중요한 점은 hap(a, b)는 이미 실행됐고 함수의 리턴값을 result가 바인딩하는 작업만 남아 있는 상태라는 점입니다. Variables 창을 보면 'Return Values' 항목에서 hap 함수의 리턴값이 8임을 확인할 수 있습니다.

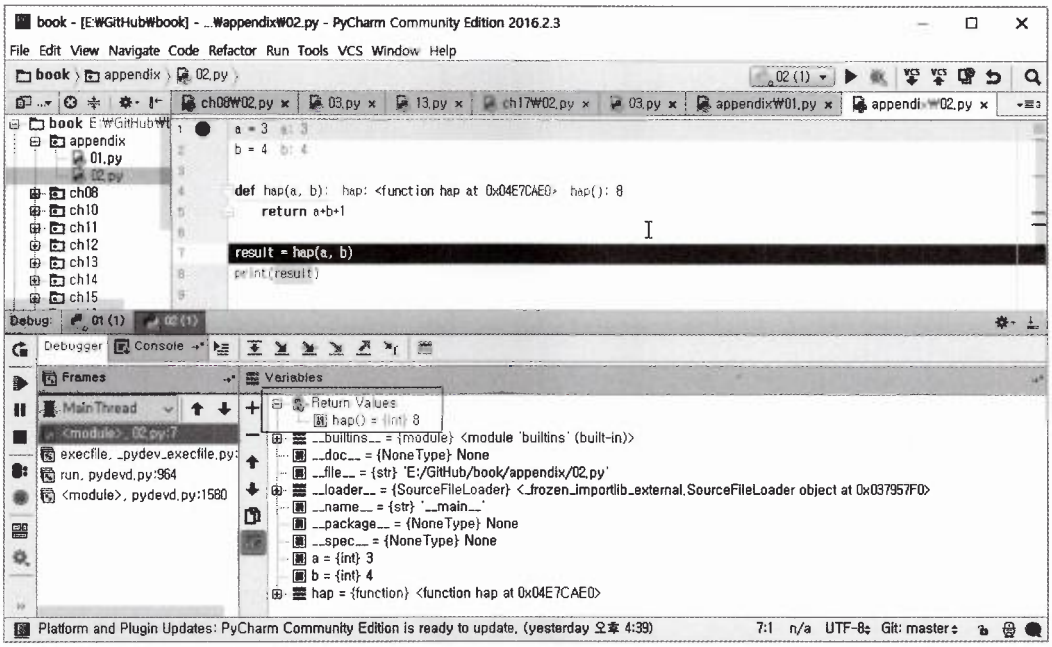


그림 B.10 함수의 리턴값 확인(1)

F10 키를 눌러 7번째 라인의 코드를 실행하면 result 변수가 8을 바인딩하는 것을 확인할 수 있습니다. 디버깅을 통해 함수의 리턴값이 잘못되어 result 변수가 7이 아닌 8을 바인딩했음을 쉽게 확인할 수 있습니다.

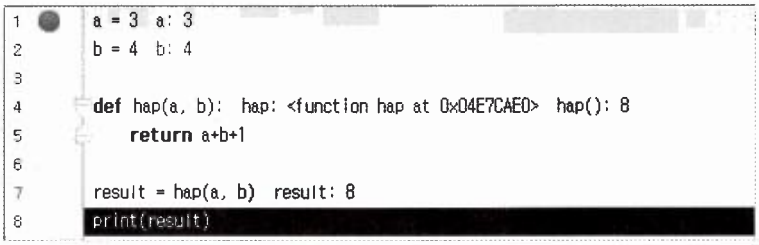


그림 B.11 함수의 리턴값 확인(2)

디버깅이 끝났다면 Shift + F5 키를 눌러 디버깅 모드를 종료합니다.

04 추가 기능 살펴보기

이번에는 다음 코드를 이용해 디버깅에 유용한 단축키를 몇 가지 더 배워보겠습니다. PyCharm에서 프로젝트를 만든 후 파일을 하나 추가합니다. 추가된 파일에 다음 코드를 입력합니다.

```
mylist1 = [0, 1, 2, 3]
mylist2 = ['a', 'b', 'c']

def debug_me(in_list):
    print(in_list[0])
    print(in_list[1])
    print(in_list[2])
    print(in_list[3])

debug_me(mylist1)
debug_me(mylist2)
```

Ctrl + F9를 눌러 코드를 실행하면 다음과 같이 결과가 출력됩니다. 다음 결과를 살펴보면 12번째 라인에서 debug_me 함수가 호출됐고 9번째 라인에서 print(in_list[3])이 수행돼서 'IndexError: list index out of range'가 출력된 것을 확인할 수 있습니다. 이는 리스트의 원소 개수보다 더 큰 수를 사용해서 인덱싱했기 때문입니다.

```
C:\Anaconda3\python.exe E:/GitHub/book/appendix/03.py
0
Traceback (most recent call last):
1
  File "E:/GitHub/book/appendix/03.py", line 12, in <module>
2
    debug_me(mylist2)
3
  File "E:/GitHub/book/appendix/03.py", line 9, in debug_me
a
    print(in_list[3])
b
IndexError: list index out of range
c

Process finished with exit code 1
```

먼저 그림 B.12와 같이 브레이크포인트를 1번째 라인에 설정해 봅시다.

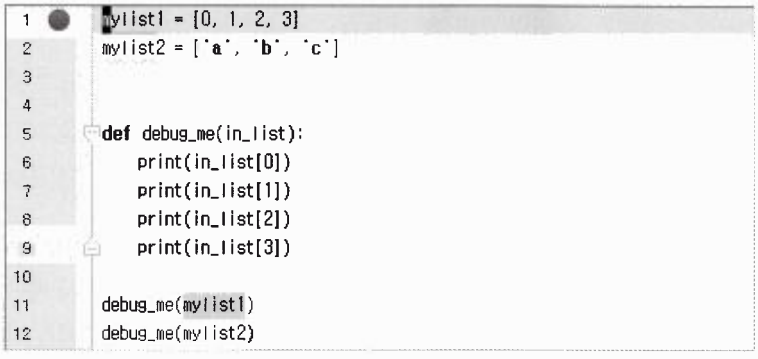


그림 B.12 브레이크포인트 설정

Alt + F5 키를 눌러 디버깅 모드로 프로그램을 실행합니다. 현재 그림 B.13과 같이 첫 번째 라인의 코드가 수행되기 전에 프로그램의 실행이 멈춘 것을 확인할 수 있습니다.

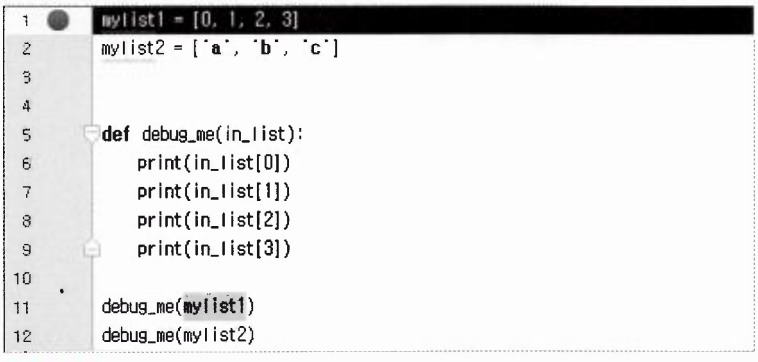


그림 B.13 디버깅 모드로 프로그램을 실행

F10 키를 누르면 1번째 라인의 코드를 실행한 후 2번째 라인의 코드가 수행되기 전에 멈춥니다. 앞서 코드의 실행 결과를 보면 12번째 라인에서 mylist2를 인자로 debug_me() 함수를 호출했을 때 문제가 있었습니다. 중간 코드는 건너뛰고 바로 12번째 라인으로 이동해 보겠습니다. 마우스 커서를 12번째 라인으로 위치한 후 Ctrl + F10 키를 누릅니다. 그림 B.14와 같이 음영으로 표시된 라인이 12번째 라인으로 이동한 것을 확인할 수 있습니다.

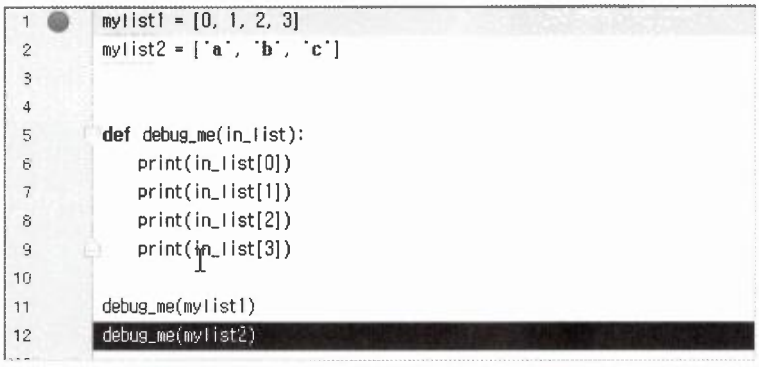


그림 B.14 실행 위치 이동

함수 내부로 이동하기 위해 F11 키를 눌러 debug_me 함수의 내부로 진입합니다. 진입 후에 F10 키를 눌러 라인 단위로 코드를 실행합니다. 이 예의 경우에는 프로그램을 실행했을 때 나오는 메시지를 보고 버그의 위치를 찾는 것이 효과적입니다. 9번째 라인에서 ['a', 'b', 'c']라는 리스트에 대해 [3]으로 인덱싱했기 때문에 에러가 발생한 것을 알 수 있습니다.

만약 길이가 긴 함수를 디버깅하는데 해당 함수에 에러가 없다고 생각이 될 때는 F10 키를 눌러서 계속 다음 라인으로 가지 않고 Shift + F11 키를 누르면 함수를 빠져나갈 수 있습니다. 꽤 자주 사용되는 기능이므로 잘 익혀두기 바랍니다.

지금까지 디버깅에 사용한 단축키는 표 B.1과 같습니다.

표 B.1 PyCharm 디버깅 단축키(Visual Studio 설정 기준)

단축키	역할
F9	브레이크포인트 설정/해제
Shift + F5	디버깅 모드로 프로그램 실행
F10	다음 코드 라인으로 이동(함수 호출인 경우 함수 내부로 이동하지 않음)
F11	다음 코드 라인으로 이동(함수 호출인 경우 함수 내부로 이동)
Shift + F11	함수를 끝내고 함수 호출부로 이동
Ctrl + F10	커서가 있는 곳으로 이동

부록

C

유용한 PyCharm 단축키

PyCharm 이번 절에서는 PyCharm의 유용한 단축키 몇 가지를 알아보겠습니다. 먼저 실습을 위해 PyCharm에서 프로젝트를 생성한 후 파일을 추가하고 다음과 같은 코드를 작성합니다.

```
print("hello world")
print("hello world")
print("hello world")
print("hello world")

def call_me():
    print("call_me")

call_me()
```

프로그래밍하다 보면 여러 줄의 코드를 주석으로 처리해야 할 때가 있습니다. 이 경우 주석으로 처리하고자 하는 코드 영역을 마우스로 선택한 후 Ctrl + / 키를 누르면 그림 C.1과 같이 해당 영역이 한 번에 주석으로 처리됩니다. 주석을 해제하려면 다시 영역을 선택한 후 Ctrl + / 키를 누르면 됩니다.

```
1   # print("hello world")
2   # print("hello world")
3   # print("hello world")
4   # print("hello world")
5
6
7   def call_me():
8       print("call_me")
9
10  call_me()
```

그림 C.1 코드 영역에 대한 주석 처리

이번에는 몇 가지 유용한 단축키를 설정해 보겠습니다. PyCharm에서 [File] → [Settings]를 차례로 선택한 후 그림 C.2와 같이 Keymap 항목에서 [Keymaps]를 'Visual Studio'로 설정합니다. 검색 창에서 "navigate declaration"을 입력하면 그림 C.2와 같은 결과가 나오는데 이때 [Declaration] 항목에 마우스 오른쪽 버튼을 클릭한 후 [Add Keyboard Shortcut]을 선택합니다.

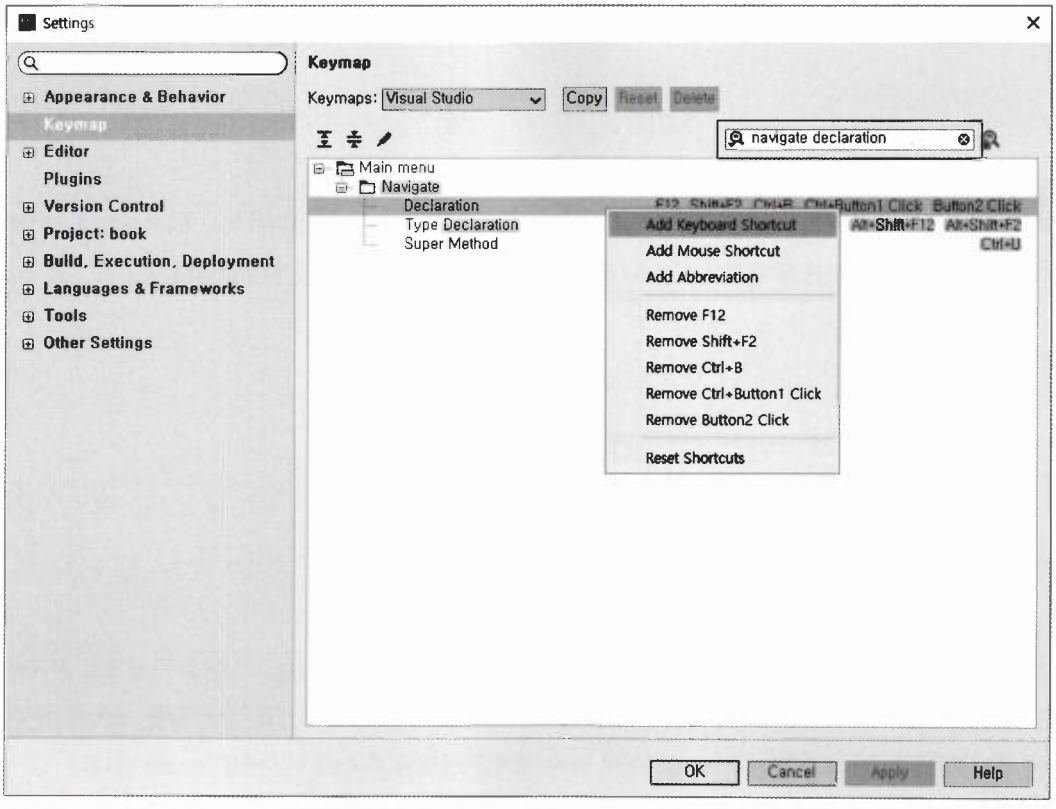


그림 C.2 Navigate Declaration 단축키 설정(1)

그런 다음 그림 C.3과 같이 Alt + G를 누르면 자동으로 단축키가 설정됩니다. 단축키를 설정하고 나면 [OK] 버튼을 클릭합니다.

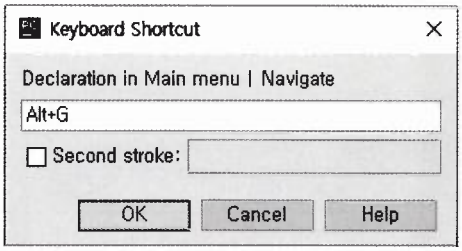


그림 C.3 Navigate Declaration 단축키 설정(2)

이번에는 Navigate Forward 단축키를 설정해 보겠습니다. 앞서 Navigate Declaration 단축키를 설정하면 [Keymaps] 항목이 'Visual Studio copy'로 자동으로 변경됩니다. 그림 C.4와 같이 검색창에 "navigate forward"를 검색한 후 앞서 설명한 방식대로 Alt + .으로 설정합니다. 단축키를 입력했다면 [OK] 버튼을 누릅니다.

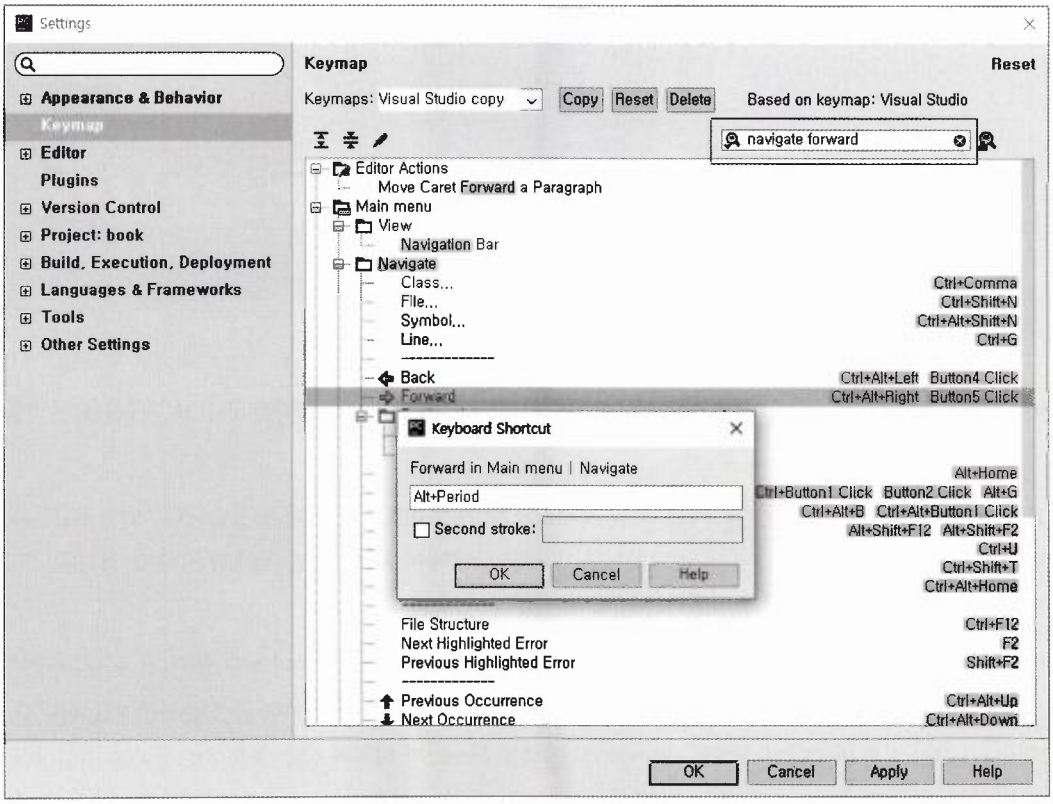


그림 C.4 Navigate Forward 단축키 설정

이번에는 Navigate Back 단축키를 설정하겠습니다. 그림 C.5와 같이 마찬가지로 검색창에 "navigate back"을 입력한 후 앞서 설명한 방식대로 Alt + ,로 설정합니다. 단축키를 입력하고 나면 [OK] 버튼을 누릅니다.

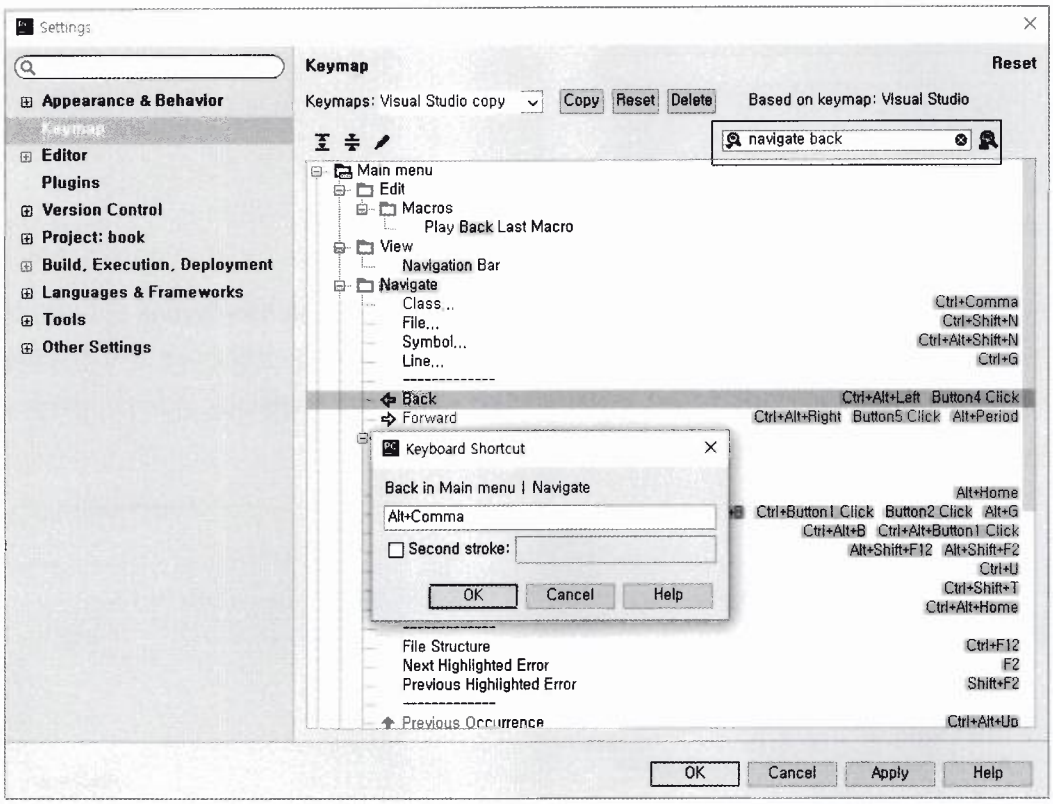


그림 C.5 Navigate Back 단축키 설정

이로써 단축키 설정을 완료했습니다. [Settings] 창의 [Apply] 버튼을 클릭한 후 [OK] 버튼을 클릭합니다.

그럼 이번에는 앞서 설정한 단축키를 통해 소스코드를 자유자재로 이동해 보겠습니다. 먼저 `call_me` 함수의 호출부(10번째 라인)에 마우스 커서를 위치한 상태에서 Alt + G를 누르면 함수가 정의된 곳(7번째 라인)으로 마우스 커서가 이동하는 것을 확인할 수 있습니다.

함수로 이동해서 함수를 살펴본 후에는 다시 함수를 호출하는 곳으로 돌아와야 합니다. 이때 사용하는 기능이 바로 Navigate Back입니다. Alt + ,를 입력하면 다시 커서가 함수 호출부로 이동합니다. Navigate Back은 마지막에 수행된 내비게이션 명령을 취소하기 때문에 함수 호출부로 돌아온 것입니다.

Navigate Forward는 취소했던 마지막 내비게이션 명령을 다시 실행하는 명령어입니다. Navigate Back을 했다가 다시 돌아가고 싶으면 Navigate Forward(단축키: Alt + .)를 사용하면 됩니다.

연습 문제 풀이

문제 1-1

1장의 설치 과정 참조

문제 1-2

```
>>> print('주식은 대박이다.')
```

주식은 대박이다.

문제 1-3

```
print
```

문제 1-4

1, 3번

2번은 "로 시작해서 '로 끝났으므로 문법 오류입니다. print 명령어로 문자를 출력할 때는 반드시 "로 시작하면 "로 끝내야 하고, '로 시작하면 '로 끝내야 합니다.

4, 5번은 ( )를 사용해야 합니다.

연습 문제 풀이

문제 2-1

```
>>> daum = 89000
>>> naver = 751000
>>> total = daum * 100 + naver * 20
>>> print(total)
23920000
```

문제 2-2

```
>>> daum = 89000
>>> naver = 751000
>>> loss = (daum * 0.05 * 100) + (naver * 0.1 * 20)
>>> print(loss)
1947000.0
```

문제 2-3

```
>>> f = 50
>>> c = (f-32) / 1.8
>>> print(c)
10.0
```

문제 2-4

```
>>> print("pizza\n" * 10)
pizza
pizza
pizza
pizza
pizza
pizza
pizza
pizza
```

```
pizza
pizza

>>>
```

문제 2-5

```
>>> monday_opening_price = 1000000
>>> monday_closing_price = monday_opening_price * 0.7
>>> tuesday_opening_price = monday_closing_price
>>> tuesday_closing_price = tuesday_opening_price * 0.7
>>> wednesday_opening_price = tuesday_closing_price
>>> wednesday_closing_price = wednesday_opening_price * 0.7
>>> print(wednesday_closing_price)
342999.99999999994
>>>
```

문제 2-6

```
>>> print("이름: 파이썬 생년월일: 2014년 12월 12일 주민등록번호: 20141212-1623210")
이름: 파이썬 생년월일: 2014년 12월 12일 주민등록번호: 20141212-1623210
>>>
```

문제 2-7

```
>>> s = 'Daum KaKao'
>>> s = s[5:] + ' ' + s[0:4]
>>> s
'KaKao Daum'
>>>
```

문제 2-8

```
>>> a = 'hi ' + a[6:]
>>> a
'hi world'
>>>
```

문제 2-9

```
>>> x = 'abcdef'
>>> x = x[1:]+x[0]
>>> x
'bcdefa'
>>>
```

문제 3-1

```
>>> naver_closing_price = [474500, 461500, 501000, 500500, 488500]
>>> naver_closing_price
[474500, 461500, 501000, 500500, 488500]
>>>
```

문제 3-2

```
>>> max_price = max(naver_closing_price)
>>> print(max_price)
501000
>>>
```

문제 3-3

```
>>> min_price = min(naver_closing_price)
>>> print(min_price)
461500
>>>
```

문제 3-4

```
>>> naver_closing_price = [474500, 461500, 501000, 500500, 488500]
>>> max_price = max(naver_closing_price)
>>> min_price = min(naver_closing_price)
>>> print("가격차: ", max_price - min_price)
가격차: 39500
>>>
```

문제 3-5

```
>>> print("수요일 종가: ", naver_closing_price[2])
수요일 종가:  501000
>>>
```

문제 3-6

```
>>> naver_closing_price2 = {'09/07':474500, '09/08':461500, '09/09':501000, '09/10': 500500,
'09/11':488500}
>>> naver_closing_price2
{'09/09': 501000, '09/11': 488500, '09/08': 461500, '09/10': 500500, '09/07': 474500}
>>>
```

문제 3-7

```
>>> print(naver_closing_price2['09/09'])
501000
>>>
```

문제 4-1

```
>>> for i in range(5):
        print('*', end='')

*****
>>>
```

문제 4-2

```
>>> for j in range(4):
        for i in range(5):
            print('*', end='')
        print("")

*****
*****
*****
*****
>>>
```

문제 4-3

```
>>> for j in range(5):
        for i in range(j+1):
            print('*', end='')
        print("")

*
**
***
****
*****
>>>
```

문제 4-4

```
>>> for j in range(5):
      for i in range(5-j):
          print('*', end='')
      print("")

*****
****
***
**
*
>>>
```

문제 4-5

```
>>> for j in range(5):
      for i in range(4-j):
          print(' ', end='')
      for i in range(j+1):
          print('*', end='')
      print("")

    *
   **
  ***
 ****
*****
>>>
```

문제 4-6

```
>>> for j in range(5):
      for i in range(j):
          print(' ', end='')
      for i in range(5-j):
          print('*', end='')
      print("")

*****
 ****
  ***
```

```
  **
   *
>>>
```

문제 4-7

```
>>> for j in range(5):
      for i in range(4-j):
           print(' ', end='')
      for i in range(2*(j+1)-1):
           print('*', end='')
      print("")

    *
   ***
  *****
 *******
*********
>>>
```

문제 4-8

```
>>> for j in range(5):
      for i in range(j):
           print(' ', end='')
      for i in range(2*(5-j)-1):
           print('*', end='')
      print("")

*********
 *******
  *****
   ***
    *
>>>
```

문제 4-9

```
>>> apart = [[101, 102, 103, 104],[201, 202, 203, 204],[301, 302, 302, 303], [401, 402, 403,
404]]
>>> arrears = [101, 203, 301, 404]
>>> for floor in apart:
      for house in floor:
           if house in arrears:
                continue
           else:
                print("Newspaper delivery: ", house)

Newspaper delivery:  102
Newspaper delivery:  103
Newspaper delivery:  104
Newspaper delivery:  201
Newspaper delivery:  202
Newspaper delivery:  204
Newspaper delivery:  302
Newspaper delivery:  302
Newspaper delivery:  303
Newspaper delivery:  401
Newspaper delivery:  402
Newspaper delivery:  403
>>>
```

문제 5-1

```
>>> def myaverage(a,b):
      return (a+b)/2

>>> myaverage(3,4)
3.5
>>>
```

문제 5-2

```
>>> def get_max_min(data_list):
      max_val = max(data_list)
      min_val = min(data_list)
      return (max_val, min_val)

>>> max_val, min_val = get_max_min([1, 2, 3, 4, 5])
>>> max_val
5
>>> min_val
1
>>>
```

문제 5-3

```
>>> import os
>>> def get_txt_list(path):
      org_list = os.listdir(path)
      ret_list = []
      for x in org_list:
          if x.endswith('txt'):
              ret_list.append(x)
      return ret_list
```

```
>>> get_txt_list('c:/')
['test.txt']
>>>
```

문제 5-4

```
>>> def cal_bmi(weight, height):
      height = height * 0.01
      bmi = weight / (height * height)
      if bmi < 18.5:
          print("마른체형")
      elif 18.5 <= bmi < 25.0:
          print("표준")
      elif 25.0 <= bmi < 30.0:
          print("비만")
      else:
          print("고도비만")

>>> cal_bmi(63, 170)
표준
>>> cal_bmi(79, 160)
고도비만
>>>
```

문제 5-5

```
>>> def cal_bmi2(height, weight):
      height = height * 0.01
      bmi = weight / (height * height)
      print("BMI: ", bmi)
      if bmi < 18.5:
          print("마른체형")
      elif 18.5 <= bmi < 25.0:
          print("표준")
      elif 25.0 <= bmi < 30.0:
          print("비만")
      else:
          print("고도비만")
>>> while 1:
      height = input("Height (cm): ")
      weight = input("Weight (kg): ")
```

```
    cal_bmi2(float(height), float(weight))

Height (cm): 170.0
Weight (kg): 63.0
BMI:  21.79930795847751
표준
```

문제 5-6

```
>>> def get_triangle_area(width, height):
    return width * height / 2

>>> get_triangle_area(10, 5)
25.0
>>>
```

문제 5-7

```
>>> def add_start_to_end(start, end):
    return sum(range(start, end+1))

>>> mysum2(1,10)
55
>>>
```

문제 5-8

```
>>> def get_abbr(data_list):
    result = []
    for x in data_list:
        result.append(x[:3])
    return result

>>> get_abbr(['Seoul', 'Daegu', 'Kwangju', 'Jeju'])
['Seo', 'Dae', 'Kwa', 'Jej']
>>>
```

연습 문제 풀이

문제 6-1

```
>>> class Point:
      def __init__(self, x, y):
           self.x = x
           self.y = y

      def setx(self, x):
           self.x = x

      def sety(self, y):
           self.y = y

      def get(self):
           return (self.x, self.y)

      def move(self, dx, dy):
           self.x += dx
           self.y += dy

>>>
```

문제 6-2

```
>>> a = Point(3,3)
>>> a.get()
(3, 3)
>>> a.setx(4)
>>> a.sety(2)
>>> a.get()
(4, 2)
>>> a.move(-4, -2)
>>> a.get()
(0, 0)
>>>
```

문제 6-3

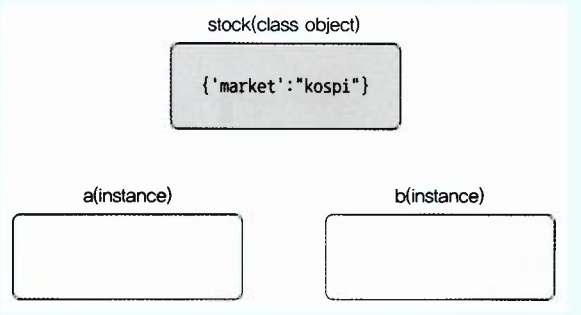


그림 6.16 Stock 클래스와 인스턴스의 네임스페이스

문제 6-4

```
>>> a.market
'kospi'
>>> b.market
'kospi'
>>> Stock.market
'kospi'
>>> a.market = "kosdaq"
>>> b.market = "nasdaq"
>>> a.market
'kosdaq'
>>> b.market
'nasdaq'
>>> Stock.market
'kospi'
>>>
```

문제 6-5

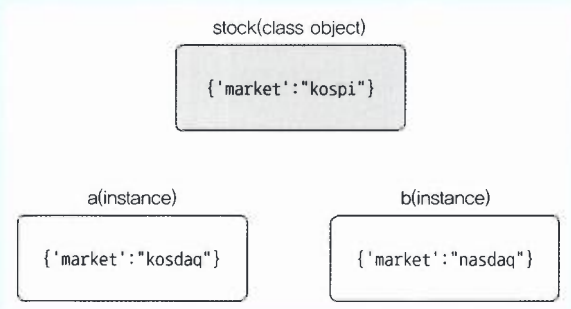


그림 6.17 Stock 클래스와 인스턴스의 네임스페이스(2)

문제 7-1

```
>>> f = open('c:\\Users\Jason\\Desktop\\number.txt', 'wt')
>>> for i in range(1, 11):
        f.write("%d\n" % i)
>>> f.close()
```

문제 7-2

```
>>> import os
>>> def print_flist(path):
        f = open('c:\\Users\\Jason\\Desktop\\flist.txt', 'wt')
        flist = os.listdir(path)
        for x in flist:
            f.write('%s\n' % x)
        f.close()

>>> print_flist('c:\\Anaconda3')
```

찾아보기

[A – D]

append() 46
abs() 115
ASCII 116
API 191
add_subplot() 406
AxesSubplot 415
binding 27
build-in function 48
boolean 58
break 80
Cpython 9
dictionary 51
dict 51
continue 80
ctime() 108
chr() 116
class 122
class variable 144
close() 157
COM 192
CYBOS Plus 208
CpStockCode 212
CpCodeMgr 217
CSV 217
CpTdUtil 245
CpTd0311 245
callback 256
CommConnect() 306
CommRqData() 318
CommGetData() 318, 320
Crawling 635
def 91
Dispatch() 210
DataFrame 338
DBMS 516

[E – K]

else 65
endswith() 112
enumerate 116
ETF 218
ETN 218
EPS 227
float 35
False 59
for 68
figure 405
getcwd() 111
GUI 292
GetConnectState() 307
GetLoginInfo() 322
GetCodeListByMarket() 324
GetMasterCodeName() 324
HTML 635
IDLE 16
indent 18
id() 28, 117
indexing 31
insert() 47
if 64
in 69
import 104
int() 118
instance 130
__init__() 133
instance variable 144
keys() 54
KOA Studio 289

[L – P]

len() 30,117
list 39
listdir() 110
module 100
max() 117
method 127
MarketEye 227
matplotlib 400
nested loop 81
__name__ 106
min() 118
os 111
open() 154
OnEventConnect 308
prompt 17
print 17
pass 126
PTVS 160

PyCharm 160
PER 227
PumpWaitingMessages() 260
PyQt 292
pandas 329
pandas-datareader 346
pip 369
pyplot 400

[Q - R]

QLabel 294, 468
Qapplication 296
QAxWidget 306
OnReceiveTrData 319
Qt Designer 451
QPushButton 466
QLineEdit 470
QStatusBar 470
QRadioButton 472
QGroupBox 472
QCheckBox 476
QSpinBox 479
QTableWidget 482
QVBoxLayout 489
QHBoxLayout 490
QGridLayout 492
QFileDialog 499
range 70
return 94
random 109
readlines() 155
Request/Reply 224

[S - Z]

string 29
slicing 30
split() 32
sleep() 108
sorted() 118
str() 118
self 135
StockChart 220
SetInputValue() 318
Series 329
subplot 405
SQLite 516
sqlite3 516
type() 35
tuple 49
True 59
time() 107
TUI 292
Transaction 311
variable 24
values() 54
while 75
win32com 194
xingAPI 248
XASession 255
widget 297
Zipline 363

[ㄱ – ㅁ]

기본 데이터 타입 34
객체지향 프로그래밍 124
객체 124
골든크로스 382
기술적 분석 515
기본적 분석 515
내장 함수 48, 115
네임스페이스 139
들여쓰기 18, 65
딕셔너리 50
딕셔너리 삽입 52
단일 데이터 조회 266
데드크로스 382
다이얼로그 498
리스트 39
리스트 생성 41
리스트 인덱싱 41
리스트 슬라이싱 43
리스트 삽입 46
리스트 삭제 49
문자열 29
모듈 100
메서드 127

[ㅂ – ㅇ]

배포판 9
변수 24
바인딩 27
부동 소수점 35
불리언 58
반환 94
반복 데이터 조회 271
백테스팅 362
봉 차트 423
슬라이싱 30
생성자 133
상속 146
시그널 446
알고리즘 트레이딩 6
아나콘다 9
인덱싱 31
이중 루프 83
임포트 104
아스키코드 116
인스턴스 127
인스턴스 변수 144
위젯 297
윈도우 298
이벤트 처리 301
이동평균선 352
일봉 데이터 저장 549
윈도우 작업 스케줄러 561

[ㅈ – ㅎ]

자료구조 38
제어문 58
중첩 루프 81
절차지향 프로그래밍 124
클래스 122
클래스 변수 144
콜백 함수 256
캔들 스틱 차트 422
크롤링 635
튜플 49
튜플 슬라이싱 50
타입 124
파이썬 7
프롬프트 17
파라미터 92
호가 61
함수 90
함수 인자 92