O'REILLY®

영화 평점, 이름 통계, 선거 데이터 등 실사례 사용

Python for Data Analysis

파이썬 라이브러리를 활용한 데이터 분석 2판

웨스 맥키니 지음
김영근 옮김

한빛미디어 Hanbit Media, Inc.

| 표지 설명 |

표지 그림은 붓꼬리나무두더지(학명: *Ptilocercus lowii*)다. 붓꼬리나무두더지는 깃털나무타기쥐 속의 유일한 종이며 다른 모든 나무두더지 종은 투파이아 속에 속한다. 나무두더지의 털은 부드러운 적갈색이고, 꼬리는 길다. 깃펜의 깃털을 닮은 꼬리 덕에 별명이 황금꼬리다. 잡식성으로 곤충이나 과일, 씨앗, 소형 척추동물을 주로 먹는다.

인도네시아, 말레이시아, 태국에서 주로 서식하는 붓꼬리나무두더지는 알코올을 굉장히 많이 섭취한다고 알려져 있다. 자연 발효된 야자즙을 몇 시간 동안 들이키는 말레이시아 나무두더지를 발견했는데, 야자즙은 대략 알코올 3.8도의 와인 10~12잔 분량이었다. 그럼에도 사람보다 훨씬 에탄올 분해 능력이 뛰어난 덕에 붓꼬리나무두더지는 취하는 법이 없다. 또한 특이하게도 사람을 포함한 다른 포유류와 비교했을 때 몸집에 비해 뇌가 크다.

이름은 두더지이지만 사실 붓꼬리나무두더지는 두더지가 아니고 영장류에 더 가깝다. 이런 점 때문에 근시, 스트레스, 간염치료제 등의 동물실험에 영장류 대신 붓꼬리나무두더지를 쓰기도 한다.

표지 그림은 『Cassel's Nature History』에서 가져왔다.

파이썬 라이브러리를 활용한 데이터 분석(2판)

영화 평점, 이름 통계, 선거 데이터 등 실사례 사용

초판 1쇄 발행 2013년 10월 1일
2판 1쇄 발행 2019년 5월 20일

지은이 웨스 맥키니 / **옮긴이** 김영근 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 서대문구 연희로2길 62 한빛미디어(주) IT출판사업부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제25100-2017-000058호 / **ISBN** 979-11-6224-190-5 93000

총괄 전태호 / **책임편집** 이상복 / **기획** 이미연 / **편집 · 조판** 김철수
디자인 표지 박정화 내지 김연정
영업 김형진, 김진불, 조유미 / **마케팅** 송경석, 김나예, 이행은 / **제작** 박성우, 김정우

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오. 잘못된 책은 구입하신 서점에서 교환해드립니다. 책값은 뒤표지에 표시되어 있습니다.
한빛미디어 홈페이지 www.hanbit.co.kr / 이메일 ask@hanbit.co.kr

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(writer@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

Python for Data Analysis

파이썬 라이브러리를 활용한 데이터 분석 2판

O'REILLY®

한빛미디어 Hanbit Media, Inc.

지은이 · 옮긴이 소개

지은이 **웨스 맥키니** Wes McKinney

뉴욕에서 활동하고 있는 소프트웨어 개발자이자 기업가다. 2007년 MIT 수학과 학부 과정을 마치고 코네티컷 주 그린위치에 있는 AQR 캐피탈 매니지먼트에서 금융 분석가로 근무했다. 복잡하고 느린 데이터 분석 툴에 실망하여 2008년 파이썬을 배우면서 pandas 프로젝트를 시작했다. 파이썬 데이터 커뮤니티의 활발한 일원이며 데이터 분석, 금융, 통계 계산 애플리케이션에서 파이썬 사용을 독려하고 있다.

웨스가 창업한 DataPad라는 회사가 2014년 클라우데라에 인수된 이후 빅데이터 기술에 집중하기 시작했으며 아파치 소프트웨어 재단의 프로젝트인 아파치 애로우와 아파치 파케이의 PMC Project Management Committee (프로젝트 관리 위원)로 합류했다. 2016년에는 뉴욕에 위치한 투시그마 투자회사로 옮겨 오픈소스를 통한 빠르고 쉬운 데이터 분석 환경을 만드는 데 노력을 기울이고 있다.

옮긴이 **김영근** iam@younggun.kim

애플 II에서 BASIC으로 처음 프로그래밍을 시작했고, 장래 희망은 항상 프로그래머라고 말하고 다니다 정신 차리고 보니 어느덧 20년 차 중년(?) 개발자가 되었다. 리눅스 커뮤니티에서 오랫동안 활동했으며 임베디드 환경에서부터 미들웨어, 웹, 스마트폰 애플리케이션에 이르기까지 다양한 분야에서 개발했다. 아시아 최초의 파이썬 소프트웨어 재단의 이사로 활동했으며 2014년 'PyCon 한국'을 처음 시작했다. 한빛미디어에서 『리눅스 시스템 프로그래밍(개정2판)』, 『고성능 파이썬』을 번역했다.

옮긴이의 말

1판을 번역하고 옮긴이의 말을 쓴 게 2013년이었으니 올해로 벌써 6년째다. 1판은 지금까지 9쇄를 찍었고 만 오천여 부가 팔렸다고 하니 어려운 국내 IT 출판 시장을 생각해볼 때 옮긴이로서는 큰 영광이 아닐 수 없다.

첫 출간 후 오랜 세월이 지났지만 이 책이 전하고자 하는 본질은 조금도 변하지 않았다. 파이썬을 이용해서 데이터 분석 작업을 하길 원하는 독자에게 다양한 분야의 사례를 통해 단계적으로 파이썬과 데이터 분석에 익숙해지도록 안내한다.

파이썬을 처음 접하는 독자를 위해 딱 필요한 만큼의 파이썬 언어의 기본을 포함하고 있으며 파이썬 데이터 분석 라이브러리인 pandas 외에도 주피터, NumPy, matplotlib 등 파이썬으로 데이터 분석 작업을 한다면 반드시 마주치게 되는 다양한 도구도 차근차근 안내하고 있다.

1판 옮긴이의 말에서는 파이썬을 프로그래밍 언어계의 아이돌이라고 소개했는데 지금의 파이썬은 아이돌이라기엔 너무 유명해진 듯하다. 파이썬이 지금 이렇게 다양한 분야에서 쓰이고 있는 이유는 바로 건강한 생태계 때문이다.

파이썬 커뮤니티는 파이썬이라는 공통 주제 하나로 구성원들 서로가 동료의식을 가지고 있다. 이런 동료의식에서 싹트는 존중과 배려는 구성원들이 커뮤니티 내에서 소속감과 참여감을 느끼게끔 만들고 이 소속감과 참여감은 생태계에 기여하는 형태로 발현된다. 이렇게 모인 자발적인 기여는 더 다양한 라이브러리, 더 풍부한 자료로 생태계를 발전시키고 여기에 유입되는 새로운 사람들이 점점 더 늘어난다.

이 책의 저자이자 pandas를 개발한 웨스 맥키니는 학교에서 수학을 전공하고 금융권에서 분석가로 근무하던, 프로그래밍과는 전혀 관련이 없는 사람이었지만 커뮤니티의 도움으로 pandas를 개발했고 이제는 pandas를 통해 더 많은 새로운 사용자들이 파이썬 커뮤니티로 유입되고 있으니 이것도 파이썬 커뮤니티에서 흔히 볼 수 있는 일종의 선순환인 셈이다.

저자가 그렇듯 저 또한 이 책이 오랫동안 사랑받을 수 있기를 바라며 1판에서 그랬던 것처럼 2판에서도 쇄를 거듭할 때마다 최신 버전에 맞춰 코드를 계속 개선할 예정이다. 좋은 책을 번역할 수 있는 기회를 주신 한빛미디어 IT출판사업부와 특히 마음고생이 컸을 이미연 님께 깊이 감사드린다. 책이 서점에 풀릴 시점이면 웨스를 만나고 있을 텐데 그전에 번역을 마무리해서 참 다행이라고 생각한다.

김영근

2판에 새로 추가된 내용

이 책의 초판은 2012년에 출간되었다. 그 시기는 pandas와 같은 파이썬을 위한 오픈소스 데이터 분석 라이브러리가 막 개발되기 시작했고 흔하지 않았던 시점이었다. 이번에 새로운 내용을 보완한『파이썬 라이브러리를 활용한 데이터 분석, 2판』에서는 pandas의 새로운 기능은 물론 5년간의 세월이 흐르는 동안 낡았거나 사용법이 바뀐 내용을 모두 반영하여 책 전반을 다시 다듬었다. 또한 2012년 당시에는 존재하지 않았거나 책에 싣기에는 불안했던 갓 나온 도구들을 소개하기 위해 새로운 내용을 추가했다. 마지막으로 1판에서 그랬던 것처럼 너무 새롭거나 아직 성숙하지 않은 오픈소스 프로젝트는 다루지 않으려 노력했다. 나는 2020년이나 2021년에도 독자들이 이 책에서 변함없이 유효한 내용을 찾을 수 있기 바란다.

2판의 주요 변경 사항은 다음과 같다.

- 파이썬 튜토리얼을 포함한 모든 코드를 파이썬 3.6 기반으로 수정했다.
- 파이썬 설치 과정을 아나콘다 파이썬 배포판과 몇몇 필수 파이썬 패키지 기준으로 설명했다.
- pandas 라이브러리 버전을 2017년 최신 버전으로 갱신했다.
- pandas 고급 사용법과 사용팁을 담은 새 장을 추가했다.
- statsmodels와 scikit-learn 라이브러리 사용에 대한 간략한 소개를 추가했다.

또한 새로운 독자들이 좀 더 쉽게 접근할 수 있도록 1판의 내용 중 많은 부분을 새롭게 재구성했다.

코드 예제

이 책에서 소개하는 코드 예제와 관련 데이터는 이 책의 깃허브 저장소에서 다운로드할 수 있다.

http://github.com/wesm/pydata-book

이 책의 내용은 전 세계 많은 사람의 수년간의 생산적인 논의, 협업 그리고 도움으로 작성되었다. 그중 일부에게 특별히 감사의 말을 전하고 싶다.

존 D. 헌터(1968-2012)를 기리며

우리의 절친한 친구이자 동료인 존 D. 헌터는 내가 이 책의 1판 최종 원고를 끝내고 얼마 지나지 않은 2012년 8월 28일 대장암으로 세상을 떠났다.

존이 파이썬 과학 계산 그리고 데이터 커뮤니티에 남긴 업적과 영향력은 말로 다할 수 없다. 2000년대 초반 그가 개발한 matplotlib은 오늘날 파이썬 생태계의 거목들이 자랄 수 있는 오픈소스 개발자 문화의 토양이 되었다.

내가 2010년 1월 pandas 0.1 버전을 릴리스하면서 오픈소스 커리어를 시작했을 때 그를 만날 수 있었던 건 큰 행운이었다. 그에게서 얻은 영감과 조언으로 인해 힘든 시기에도 pandas에 대한 비전과 최고의 데이터 분석 언어로서의 파이썬을 향해 앞으로 나아갈 수 있었다.

존은 IPython, 주피터 프로젝트를 시작했던 페르난도 페레즈, 브라이언 그레인저와 무척 가까운 사이였고 파이썬 커뮤니티 내 다른 많은 단체와도 가까이 지냈다. 우리 넷이서 함께 책을 쓰기 바랐으나 결국은 여유 시간이 가장 많았던 나 혼자 작업을 하게 되었다. 나는 존이 한 개인이자 커뮤니티로서 지난 5년간 이룩해낸 성과를 자랑스러워할 것이라 믿는다.

2판

이 책의 1판 최종 원고를 2012년 7월에 마무리했으니 거의 7년이 지났다. 그동안 많은 것이 변했다. 파이썬 커뮤니티는 엄청나게 성장했고 이를 둘러싸고 있는 오픈소스 소프트웨어 생태계 역시 크게 번창했다.

이번 2판은 지칠 줄 모르는 노력으로 pandas 프로젝트와 사용자 커뮤니티를 파이썬 데이터 과학 생태계의 주춧돌로 성장시킨 핵심 기여자들이 없었다면 결코 존재하지 않았을 것이다. 톰

옥스퍼거, 조리스 판 덴 보쉬, 크리스 발탁, 필립 클라우드, gfyoung, 앤디 헤이든, 마사키 호리코시, 스테판 호이어, 애덤 클라인, 워터 오버마이어, 제프 리백, 챙 쉬, 스키퍼 시볼드, 제프 트라트너, y-p 그리고 지면 사정상 언급하지 못하는 모든 pandas 기여자에게 감사의 말을 전한다.

2판을 집필하는 과정에서 인내를 가지고 도와준 오라일리의 마리 보로고, 벤 로리카, 콜린 토포렉에게도 감사의 인사를 전하고 싶다. 또한 큰 도움이 되어준 기술 감수자 톰 옥스퍼거, 폴 베리, 휴 브라운, 조나단 코우, 안드레아스 뮐러에게도 감사를 전한다.

이 책의 초판은 중국어, 프랑스어, 독일어, 일본어, 한국어 그리고 러시아어로 번역되었다. 책에 담긴 모든 내용을 번역하고 더 많은 독자에게 전할 수 있도록 하는 작업은 많은 노력이 필요한 일이다. 전 세계의 더 많은 사람이 프로그래밍하는 방법과 데이터 분석 도구 사용법을 배우는 데 도움을 주어서 감사하다.

나는 지난 몇 년간 클라우데라와 투시그마 인베스트먼트로부터 오픈소스 개발을 지속할 수 있도록 도움을 받았다. 사용자 규모에 비해 상대적으로 열약한 자원만으로 개발해야 하는 오픈소스 소프트웨어 프로젝트에서 이런 지원은 사업적인 측면에서도 날이 갈수록 중요해지고 있으며 올바른 방향이라고 생각한다.

1판(2012)

많은 사람의 도움이 없었다면 이 책을 쓰는 일은 무척 힘들었을 것이다.

오라일리 편집자인 메건 블란쳇과 출판 과정을 도와준 오라일리의 줄리 스틸에게 특히 감사의 말을 전한다. 마이크 로우키디스 역시 기획 단계에서 함께 일했고 이 책의 출간에 도움을 주었다.

여러 명에게 충분한 기술 검수를 받았는데, 특히 마튼 블레이스와 휴 화이트는 이 책의 예제와 정확성, 책의 시작부터 끝까지의 구성을 개선하는 데 많은 도움을 주었다. 제임스 롱, 드류 콘

웨이, 페르난도 페레즈, 브라이언 그레인저, 토마스 클루이베르, 애덤 클레인, 조쉬 클레인, 챙 쉐 그리고 스테판 반 데 발트는 각각 한 장 이상 검수해줬고 다양한 관점의 피드백을 주었다.

친구들과 데이터 커뮤니티의 동료인 마이크 듀어, 제프 해머바커, 제임스 존드로우, 크리스챤 럼, 애덤 클레인, 힐러리 메이슨, 챙 쉐 그리고 애슐리 윌리암스로부터 예제와 데이터셋에 대한 많은 훌륭한 아이디어를 얻었다.

내가 매일 사용하는 도구를 개발하고 이 책을 쓰는 동안 격려를 아끼지 않은 오픈소스 과학 기술 커뮤니티의 수많은 리더, IPython core 팀(페르난도 페레즈, 브라이언 그레인저, 민 래건-켈리, 토마스 클루이베르 그리고 다른 모든 사람), 존 헌터, 스키퍼 시볼드, 트래비스 올리펀트, 피터 웽, 에릭 존스, 로버트 컨, 조세프 퍼크롤드, 프란체스 알테드, 크리스 폰네스벡 그리고 언급하지 못한 많은 사람에게 큰 신세를 졌다. 드류 콘웨이, 션 테일러, 귀셉 팔레올로고, 재러드 랜더, 데이비드 엡스타인, 존 크로바스, 조슈아 블룸, 덴 필즈워스, 존 마일스-화이트 그리고 기억하지 못하는 다른 많은 사람이 많은 지원과 아이디어와 격려를 보내주었다.

또한 내가 성장하는 데 도움을 줬던 사람들에게도 감사의 마음을 전한다. 먼저 나의 AQR 동료로 오랫동안 pandas 개발을 응원해준 알렉스 레이프만, 마이클 웅, 팀 사르겐, 옥테이 쿠르바노프, 매튜 챤츠, 로니 이즈라엘로프, 마이클 캇츠, 크리스 우가, 프라사드 라마난, 테드 스퀘어, 김훈, 마지막으로 지도교수인 하이네스 밀러(MIT)와 마이크 웨스트(Duke)에게도 감사한다.

2014년에 pandas 라이브러리의 변화에 맞춰 잘못된 부분을 수정하기 위해 코드 예제를 갱신하면서 필립 클라우드와 조리스 판 덴 보쉬로부터 많은 도움을 받았다.

개인적으로는 책을 쓰는 내내 도와주고 너무 늦어진 일정으로 짜증내며 마지막 원고와 씨름하고 있을 때 견뎌준 캐시 딘킨에게 감사한다.

웨스 맥키니

CONTENTS

CONTENTS

CHAPTER 3 내장 자료구조, 함수, 파일

CHAPTER 4 NumPy 기본: 배열과 벡터 연산

CONTENTS

CHAPTER 6 데이터 로딩과 저장, 파일 형식

CONTENTS

CHAPTER 7 데이터 정제 및 준비

CHAPTER 8 데이터 준비하기: 조인, 병합, 변형

CHAPTER 9 그래프와 시각화

CONTENTS

CHAPTER 10 데이터 집계와 그룹 연산

CHAPTER 11 시계열

CONTENTS

CHAPTER 12 고급 pandas

CHAPTER 13 파이썬 모델링 라이브러리

CHAPTER 14 데이터 분석 예제

APPENDIX A 고급 NumPy

CONTENTS

APPENDIX B IPython 시스템 더 알아보기

CHAPTER 1

시작하기 전에

1.1 이 책에서 다루는 내용

이 책은 파이썬으로 데이터를 다루는 다양하고 기본적인 방법을 소개한다. 그러기 위해 파이썬 프로그래밍 언어의 일부와 데이터 분석 문제를 효율적으로 해결하는 데 도움이 되는 몇 가지 라이브러리를 다룬다. '데이터 분석'이 이 책의 제목이긴 하지만 데이터 분석 방법론이 아니라 파이썬 프로그래밍, 라이브러리 그리고 사용하는 도구에 집중한다. 데이터 분석을 위해 여러분에게 반드시 필요한 것은 파이썬 프로그래밍이기 때문이다.

1.1.1 어떤 데이터를 사용하나

여기서 '데이터'는 정확히 무슨 뜻일까? 주된 의미는 **구조화된 데이터**다. 일부러 구조화된 데이터라는 모호한 표현을 썼는데, 다음과 같은 여러 가지 형태의 데이터를 포함한다.

- 각 컬럼의 형식이 문자열, 숫자, 날짜 등으로 서로 다른 표 혹은 스프레드시트와 비슷한 데이터. 이는 관계형 데이터베이스 혹은 탭이나 쉼표로 구분되는 텍스트 파일 형식으로 저장되는 대부분의 데이터를 포함한다.
- 다차원 배열(행렬)
- SQL에서 기본키나 외래키 같은 키 컬럼에 의해 서로 연관되는 여러 가지 표
- 일정하거나 일정하지 않은 간격의 시계열

이 목록에 있는 형식이 전부는 아니다. 항상 명백하지는 않겠지만 대부분의 데이터는 모델링이나 분석을 위해 좀 더 쉬운 구조로 형태를 바꿀 수 있다. 또는 데이터 안에서 어떤 특성을 추출해

서 구조화된 형태로 만들 수 있다. 예를 들어 뉴스 기사 모음은 사용 단어 빈도표를 만들어 감성 분석에 사용할 수도 있다.

아마도 전 세계적으로 가장 널리 사용되고 있는 데이터 분석 툴인 마이크로소프트 엑셀 같은 스프레드시트 프로그램 사용자는 이런 종류의 데이터가 낯설지 않을 것이다.

1.2 왜 데이터 분석에 파이썬을 사용하나

파이썬은 매력적인 언어다. 1991년 처음 발표된 이래 펄, 루비 같은 인기 있는 언어가 되었다. 특히 최근 몇 년 사이에 레일즈(루비), 장고(파이썬) 같은 다양한 웹 프레임워크로 웹사이트를 만들면서 파이썬과 루비는 큰 인기를 얻었다. 이런 언어는 **스크립트** 언어라고 불리는데, 작은 프로그램이나 업무 자동화 스크립트를 빠르고 간단하게 만들 수 있다. 개인적으로 '스크립트 언어'라는 용어를 좋아하지 않는데, 이름 자체에 제대로 된 소프트웨어를 만드는 데는 사용하지 못한다는 의미를 담고 있기 때문이다. 인터프리터 언어 사이에서도 파이썬은 다양한 역사적, 문화적 이유로 인해 방대하고 활동적인 과학 계산 컴퓨팅 커뮤니티에서 사용되고 있다. 지난 10년간 파이썬은 '대안 언어' 위치에서 데이터 과학, 머신러닝 그리고 범용 소프트웨어 개발에 이르기까지 학계와 업계 모두에서 가장 중요한 프로그래밍 언어 중 하나로 성장했다.

파이썬은 데이터 분석과 대화형(인터랙티브) 컴퓨팅, 데이터 시각화에서 자주 사용하는 R, 매트랩MATLAB, SAS, Stata 같은 오픈소스나 상용 언어, 도구와 비교해도 뒤지지 않는다. 최근에는 (pandas나 scikit-learn 같은) 파이썬 라이브러리 지원이 개선되어 데이터 처리 업무에 두각을 나타내고 있다. 파이썬은 범용적인 프로그래밍 언어일 뿐만 아니라 과학 계산용으로도 손색이 없기에 데이터 애플리케이션 개발을 위한 최고의 언어라고 할 수 있다.

1.2.1 접착제처럼 쓰는 파이썬

파이썬이 과학 기술 컴퓨팅계에서 성공을 하게 된 이유로 C, C++, 포트란FORTRAN 코드와 통합이 쉽다는 점을 들 수 있다. 대부분의 최신 컴퓨팅 환경에서는 선형대수, 최적화, 통합, 고속 푸리에 변환 같은 알고리즘을 위해 C 라이브러리나 포트란 레거시를 공유한다. 그런 연유로 많은 회사나 국가 연구소에서 수십 년이 지난 소프트웨어를 파이썬과 함께 사용하고 있다.

프로그램은 실행 시간의 대부분을 차지하는 작은 부분의 코드와 실행 시간을 얼마 차지하지 않는 많은 양의 '글루 코드glue code(접착제 코드, 바인딩 코드라고도 함)'로 이루어져 있다. 대부분의 경우 글루 코드는 실행 시간에 영향을 주지 않을 만큼 비중이 낮다. 연산 병목을 최적화하기 위해 해당 부분을 C언어 같은 저수준 언어로 옮겨 쓴다면 유익한 결과를 얻을 수 있다.

1.2.2 한 가지 언어만 사용하자

보통 많은 기관에서 R이나 SAS 같은 좀 더 특화된 언어로 새로운 아이디어를 검증하고 프로토타입을 만들어 연구한 후 그 아이디어를 자바, C#, C++ 같은 언어를 이용하여 상용 시스템으로 포팅한다. 파이썬은 연구를 하거나 프로토타입을 만드는 데 적합한 언어인데다 실제 시스템을 개발하는 데도 적합하기에 갈수록 더 인기를 끌고 있다. 하나의 언어로 충분한데 별도의 다른 개발 환경을 유지할 필요는 없지 않은가. 연구자와 기술자가 같은 프로그래밍 도구를 사용함으로써 얻을 수 있는 장점이 많으므로 앞으로 더 많은 기관에서 파이썬을 사용하게 될 것이라 믿는다.

1.2.3 파이썬을 사용하면 안 되는 경우

파이썬이 분석 애플리케이션이나 범용 시스템을 개발하는 데 훌륭한 환경이긴 하지만 특수한 경우에는 파이썬이 아닌 다른 언어가 해답인 경우도 있다.

파이썬은 인터프리터 언어이므로 자바나 C++ 같은 컴파일 언어보다 많이 느리다. 하지만 **개발자의 시간 비용**은 **CPU의 시간 비용**보다 비싸므로 대개는 이런 등가 교환에 만족해한다. 그러나 실시간 거래 시스템처럼 매우 짧은 응답 시간을 필요로 하는 애플리케이션에서는 가능한 한 최고의 성능을 내고자 생산성은 떨어지지만 C++ 같은 저수준 언어로 개발을 한다.

파이썬은 동시다발적인 멀티스레드를 처리하거나 CPU에 집중된 많은 스레드를 처리하는 애플리케이션에 적합한 언어가 아니다. 바로 **GIL**global interpreter lock(전역 인터프리터 잠금) 때문인데, 이 메커니즘은 인터프리터가 한 번에 하나의 파이썬 명령만 실행하도록 한다. 왜 GIL이 존재하는지에 대한 기술적인 이유는 이 책에서 다루는 내용을 벗어난다. 대체로 빅데이터 처리 애플리케이션에서는 단일 클러스터가 적절한 시간 안에 데이터를 처리해야 하기에 단일 프로세스, 멀티스레드 시스템을 선호하는 경우도 있다.

그렇다고 엄밀히 말해서 파이썬이 멀티스레드나 병렬 코드를 실행하지 못한다는 뜻은 아니다. 네이티브 수준(C 또는 C++)에서 멀티스레드를 활용하는 파이썬 C 확장을 통해 GIL에 구애받지 않고 병렬 코드를 실행할 수 있다.

1.3 필수 파이썬 라이브러리

이 책에서 사용하는 파이썬 데이터 환경과 라이브러리에 익숙하지 않은 독자를 위해 그중 일부를 간단히 소개한다.

1.3.1 NumPy

NumPy(넘파이)는 Numerical Python의 줄임말로, 파이썬 산술 계산의 주춧돌 같은 라이브러리다. 자료구조, 알고리즘 산술 데이터를 다루는 대부분의 과학 계산 애플리케이션에서 필요한 라이브러리를 제공한다. NumPy가 제공하는 기능은 다음과 같다.

- 빠르고 효율적인 다차원 배열 객체 ndarray
- 배열 원소를 다루거나 배열 간의 수학 계산을 수행하는 함수
- 디스크로부터 배열 기반의 데이터를 읽거나 쓸 수 있는 도구
- 선형대수 계산, 푸리에 변환, 난수 생성기
- 파이썬 확장과 C, C++ 코드에서 NumPy의 자료구조에 접근하고 계산 기능을 사용할 수 있도록 해주는 C API

고속 배열 처리 외에도 NumPy는 데이터 분석 알고리즘에 사용할 데이터 컨테이너의 역할을 한다. 수치 데이터라면 NumPy 배열은 파이썬 내장 자료구조보다 훨씬 효율적인 방법으로 데이터를 저장하고 다룰 수 있다. 또한 C나 포트란 같은 저수준 언어로 작성한 라이브러리는 NumPy 배열에 저장된 데이터를 복사하지 않고 바로 사용할 수도 있다. 따라서 파이썬을 위한 많은 산술 계산 도구는 NumPy 배열을 기본 자료구조로 가정하고 있거나 NumPy와 쉽게 연동할 수 있는 기능을 제공한다.

1.3.2 pandas

pandas(팬더스)는 구조화된 데이터나 표 형식의 데이터를 빠르고 쉽고 표현적으로 다루도록 설계된 고수준의 자료구조와 함수를 제공한다. 2010년 처음 개발되어 파이썬으로 생산적이고 강력한 데이터 분석 환경을 구성하는 데 도움을 주고 있다. pandas의 주된 자료구조는 표 형태의 로우와 컬럼 이름을 가지는 `DataFrame`(데이터프레임)과 1차원 배열 객체인 `Series`(시리즈)다.

pandas는 'NumPy의 고성능, 배열 연산 아이디어'에 스프레드시트와 관계형 데이터베이스(SQL 같은)의 유연한 데이터 처리 기능을 결합한 것이다. 세련된 색인 기능을 제공하여 데이터 변형, 자르기, 취합 그리고 데이터의 부분집합을 선택할 수 있도록 해준다. 데이터를 처리하고 준비하고 다듬는 과정은 데이터 분석에서 가장 중요한 부분이므로 pandas는 이 책에서 우선적으로 집중하는 라이브러리다.

pandas 라이브러리 개발 배경을 간단히 알아보자. 나는 2008년 AQR 캐피탈 매니지먼트에서 퀀트quant로 근무하는 동안 pandas 개발을 시작했는데, 그 당시 다음과 같은 요구 사항을 만족하는 도구를 찾을 수 없었기 때문이었다.

- 자동적으로 혹은 명시적으로 축의 이름에 따라 데이터를 정렬할 수 있는 자료구조. 이러한 자료구조는 잘못 정렬된 데이터에 의한 일반적인 오류를 예방하고 다양한 소스에서 가져온 다양한 방식으로 색인되어 있는 데이터를 다룰 수 있다.
- 통합된 시계열 기능
- 시계열 데이터와 비시계열 데이터를 함께 다룰 수 있는 통합 자료구조
- 메타데이터를 보존하는 산술 연산과 축약 연산
- 누락된 데이터를 유연하게 처리할 수 있는 기능
- 일반 데이터베이스(예를 들면 SQL)처럼 데이터를 합치고 관계 연산을 수행하는 기능

나는 이 모든 것을 하나로 처리할 수 있기 바랐으며 범용 소프트웨어 개발에도 사용할 수 있는 언어를 원했다. 파이썬은 이 목적에 부합하는 좋은 후보였으나 당시에는 이러한 기능을 모두 제공하는 통합 자료구조와 도구가 존재하지 않았다. pandas는 금융 문제와 사업 분석 문제를 해결할 목적으로 개발되었기에 사업 진행에 따라 생성된 데이터를 다룰 수 있는 시계열 기능과 도구가 핵심 특성이 되었다.

통계 계산에 R 언어를 사용하는 사용자에게 R의 `data.frame` 객체에서 따온 DataFrame이란 이름은 익숙할 것이다. 파이썬과는 다르게 `data.frame`은 R 프로그래밍 언어의 표준 라이브러리

에 포함되어 있다. 결과적으로 pandas의 많은 기능은 R 핵심 구현의 일부 또는 애드온 패키지에서 따왔다.

pandas라는 이름은 다차원으로 구조화된 데이터를 뜻하는 경제학 용어인 **패널 데이터**panel data와 **파이썬 데이터 분석**Python data analysis에서 따온 이름이다.

1.3.3 matplotlib

matplotlib(맷플롯립)은 그래프나 2차원 데이터 시각화를 생성하는 유명한 파이썬 라이브러리다. 존 D. 헌터가 만들었고 지금은 많은 개발 팀이 유지하고 있다. 출판물에 필요한 그래프를 만드는 데 맞춰 설계되었다. 현재 파이썬에서 사용할 수 있는 다양한 시각화 라이브러리가 존재하지만 matplotlib은 생태계 내 다른 라이브러리들과 잘 연동되어 있기에 여전히 가장 많이 사용되고 있다. 기본 시각화 도구로 가장 안전한 선택이라고 생각한다.

1.3.4 IPython과 Jupyter

IPython(아이파이썬, 인터랙티브 파이썬Interactive Python)은 더 나은 대화형(인터랙티브) 파이썬 인터프리터를 만들 목적으로 2001년 페르난도 페레즈가 취미 프로젝트로 시작했다. 그 후 16년 동안 최신 파이썬 데이터 기술 스택에서 빠질 수 없는 가장 중요한 프로젝트로 성장했다. IPython 자체는 계산이나 데이터 분석 도구로서의 기능을 제공하지는 않지만 대화형 컴퓨팅과 소프트웨어 개발 양쪽 모두에서 생산성을 극대화할 수 있도록 설계되었다. IPython은 많은 프로그래밍 언어들의 특징인 전통적인 편집-컴파일-실행 방식 대신에 실행-탐색 방식을 장려하며 파일시스템과 운영체제 셸에도 쉽게 접근할 수 있다. 대부분의 데이터 분석 코드를 작성하는 일은 탐색적이며 반복적인 실행을 동반하므로 IPython을 이용하면 더 수월한 작업이 가능해진다.

2014년 페르난도와 IPython 팀은 언어에 상관없이 대화형 컴퓨팅 도구를 설계할 수 있는 주피터(Jupyter) 프로젝트를 발표했다. IPython 웹 노트북은 주피터 노트북으로 이름을 바꾸었고 현재 40개가 넘는 프로그래밍 언어를 지원한다. IPython 시스템은 이제 주피터에서 파이썬을 사용할 수 있게 해주는 **커널**(프로그래밍 언어 모드)로 역할을 변경했다.

IPython 자체는 이제 좀 더 큰 범위에 생산적인 대화형/탐색형 컴퓨팅 환경을 지원하는 주피터 오픈소스 프로젝트의 컴포넌트 중 하나가 되었다. 최초의 IPython은 파이썬 셸 기능을 확장하여 더 편리하게 파이썬 코드를 작성하고 테스트하고 디버깅할 수 있도록 설계되었다. 주피터 노트북에서도 IPython 시스템을 여전히 사용할 수 있는데, '노트북'이라고 하는 웹 기반의 대화형 코드 작성 환경은 다양한 프로그래밍 언어를 지원한다. IPython 셸과 주피터 노트북은 데이터를 탐색하고 시각화하는 데 특히 유용하다.

주피터 노트북 시스템은 노트북 내용을 마크다운이나 HTML로 저장할 수 있게 한다. 이를 통해 코드와 텍스트를 포함하는 문서를 생성할 수 있다. 다른 프로그래밍 언어도 주피터 환경을 위한 커널이 구현되어 있다면 파이썬 대신 주피터 환경에서 사용할 수 있다.

개인적으로 파이썬 코드를 실행하거나 디버깅, 테스트 작업을 할 때는 거의 항상 IPython을 사용한다.

이 책에서 사용된 모든 코드 예제는 주피터 노트북 파일로 찾을 수 있다(https://github.com/wesm/pydata-book).

1.3.5 SciPy

SciPy(사이파이)는 과학 계산 컴퓨팅 영역의 여러 기본 문제를 다루는 패키지 모음이다. 다음은 SciPy에 포함된 패키지 중 일부다.

- **`scipy.integrate`**
  수치적분 루틴과 미분방정식 풀이법
- **`scipy.linalg`**
  `numpy.linalg`에서 제공하는 것보다 더 확장된 선형대수 루틴과 매트릭스 분해
- **`scipy.optimize`**
  함수 최적화기와 방정식의 근을 구하는 알고리즘
- **`scipy.signal`**
  시그널 프로세싱 도구
- **`scipy.sparse`**
  희소 행렬과 희소 선형 시스템 풀이법
- **`scipy.special`**
  감마 함수처럼 흔히 사용되는 수학 함수를 구현한 포트란 라이브러리인 SPECFUN 래퍼

- `scipy.stats`
  표준 연속/이산 확률 분포(밀도 함수, 샘플러, 연속 분포 함수)와 다양한 통계 테스트 그리고 좀 더 기술적인 통계도구

NumPy와 SciPy를 함께 사용하면 전통적인 과학 계산 애플리케이션에서 제공하는 거의 모든 기능을 대체할 수 있다.

1.3.6 scikit-learn

scikit-learn(사이킷런)은 처음 개발되기 시작한 2010년부터 파이썬 개발자를 위한 범용 머신러닝 도구로 자리 잡기 시작했다. 단 7년 만에 전 세계에서 1,500명이 넘는 사람이 프로젝트에 기여하고 있다. 다음과 같은 모델의 하위모듈을 포함한다.

- **분류** : SVM, 최근접 이웃, 랜덤 포레스트, 로지스틱 회귀 등
- **회귀** : 라소, 리지 회귀 등
- **클러스터링** : k-평균, 스펙트럴 클러스터링 등
- **차원 축소** : PCA, 특징 선택, 행렬 인수분해 등
- **모델 선택** : 격자탐색, 교차검증, 행렬
- **전처리** : 특징 추출, 정규화

pandas, statsmodels 그리고 IPython과 함께 scikit-learn은 파이썬이 생산적인 데이터 과학 언어로 자리매김하는 데 일등공신 역할을 했다. 이 책에 scikit-learn의 자세한 내용을 모두 담을 수는 없지만 몇몇 모델과 이 책에서 소개한 다른 도구를 어떻게 함께 이용할 수 있는지 간략하게 소개하겠다.

1.3.7 statsmodels

statsmodels은 다양한 R 언어용 회귀분석 모델을 구현한 스탠퍼드 대학의 통계학 교수인 조나단 테일러Jonathan Taylor의 작업을 기반으로 만들어진 통계분석 패키지다. 스키퍼 시볼드Skipper Seabold와 죠세프 퍼크톨드Josef Perktold가 2010년에 새로운 statsmodels 프로젝트를 시작한 이후 수많은 사용자와 오픈소스 기여자에게 빼놓을 수 없는 프로젝트로 성장했다. 나다니엘 스미스Nathaniel Smith는 R 언어의 포뮬러 시스템에서 착안하여 statsmodels용 포뮬러 또는 모델 명세 프레임워크를 제공하는 Patsy(팻시) 프로젝트를 개발했다.

scikit-learn과 비교하여 statsmodels는 전통적인 통계(주로 빈도주의적 접근)와 계량경제학 알고리즘을 포함하고 있다. 다음과 같은 하위모듈을 포함한다.

- **회귀 모델**: 선형회귀, 일반화 선형 모델, 로버스트 선형 모델, 선형 혼합효과 모델 등
- **분산분석**(ANOVA: analysis of variance)
- **시계열분석**: AR, ARMA, ARIMA, VAR 및 기타 모델
- **비모수 기법**: 커널밀도추정, 커널회귀
- **통계 모델 결과의 시각화**

statsmodels는 통계추론에 좀 더 초점을 맞추고 있다. 인자를 위한 불확실성 예측치와 p 값을 제공한다. 반면 scikit-learn은 좀 더 예측에 초점을 맞추고 있다.

scikit-learn과 함께 statsmodels을 간략히 소개하고 어떻게 이를 NumPy 및 pandas와 함께 사용하는지 알아볼 것이다.

1.4 설치 및 설정

저마다 다른 애플리케이션에 파이썬을 사용하고 있으므로 필요한 추가 패키지나 파이썬 설정에 유일한 방법은 존재하지는 않는다. 많은 독자가 이 책을 따라 하는 데 완벽한 파이썬 개발 환경을 갖추고 있지 못할 것이므로 각각의 운영체제에 맞는 환경 설정 방법을 소개하겠다. 무료로 배포되는 아나콘다Anaconda 배포판을 이용할 것을 추천한다. 이 책을 쓰는 시점에 아나콘다는 파이썬 2.7과 3.6을 모두 지원하고 있다. 이 책에서는 파이썬 3.6 버전을 사용하고 있으므로 여러분도 파이썬 3.6 또는 그 이상의 버전을 사용할 것을 추천한다.

1.4.1 윈도우

먼저 아나콘다 인스톨러를 내려받는다. 이 책을 쓴 시점과 이를 읽는 시점에 따라 다소간의 차이가 있을 수 있지만 아나콘다 다운로드 페이지에 있는 안내대로 설치를 마무리한다.

설치가 완료되면 설정이 제대로 되어 있는지 확인해보자. 명령 프롬프트(cmd.exe라고도 함)를 실행하려면 '시작' 메뉴를 오른 클릭하고 '명령' 프롬프트를 선택하면 된다. `python`이라고 입

력해서 파이썬 인터프리터를 실행하자. 방금 설치한 아나콘다 버전이 표시된 메시지를 확인할 수 있다.

```
C:\Users\wesm>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, 11:41:13)
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

파이썬 인터프리터를 끝내려면 리눅스나 맥OS에서는 Ctrl-D를, 윈도우에서는 Ctrl-Z를 누르거나 **`exit()`**를 입력하고 엔터를 누르면 된다.

1.4.2 애플(OS X, macOS)

macOS용 아나콘다 인스톨러를 내려받자. Anaconda3-4.1.0-MacOSX-x86_64.pkg와 비슷한 이름의 파일이다. .pkg 파일을 더블 클릭해서 인스톨러를 실행하자. 인스톨러가 실행되면 .bash_profile 파일에 아나콘다 실행 경로가 추가된다. 이 파일은 /Users/$USER/.bash_profile에 위치한다.

잘 설치되었는지 확인하려면 터미널 애플리케이션을 실행하고 IPython을 실행해본다.

```
$ ipython
```

파이썬 인터프리터를 끝내려면 Ctrl-D나 **`exit()`**를 입력하고 엔터를 누르면 된다.

1.4.3 GNU/리눅스

리눅스 환경은 사용자의 취향에 따라 설치 방법이 조금씩 다를 수 있지만 여기서는 데비안, 우분투, CentOS, 페도라 같은 배포판을 기준으로 설명하겠다. 설치 방법은 macOS와 유사하지만 아나콘다가 설치되는 방법은 조금 다르다. 인스톨러는 터미널에서 실행해야 하는 셸 스크립트로 제공된다. 32비트 또는 64비트 환경에 맞게 x86(32비트) 또는 x86_64(64비트) 인스톨러를 내려받아야 한다. 인스톨러는 Anaconda3-4.1.0-Linux-x86_64.sh 같은 이름의 파일이다. 인스톨러를 실행하려면 bash(배시)를 이용해서 다음과 같이 실행한다.

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```

NOTE_ 어떤 리눅스 배포판은 이미 필요한 파이썬 패키지를 가지고 있으면 apt 같은 도구를 이용해서 설치할 수 있다. 여기서는 아나콘다를 이용한 방법을 설명하고 있는데 아나콘다를 이용하면 배포판이 달라도 쉽게 재현이 가능하며 간단하게 최신 버전으로 패키지를 업그레이드할 수 있다.

라이선스에 동의하고 나면 아나콘다 파일을 어디에 복사할 것인지 물어보는데, 기본 위치인 홈 디렉터리(예를 들면 /home/$USER/Anaconda)에 설치할 것을 추천한다.

$PATH 환경변수에 아나콘다의 실행 파일 디렉터리인 bin/을 추가할 것인지 물어볼 수 있다. 만일 설치 후 어떤 문제가 발생한다면 직접 .bashrc(zsh을 사용한다면 .zshrc) 파일에 다음 줄을 추가할 수도 있다.

```
export PATH=/home/$USER/anaconda/bin:$PATH
```

이 과정이 끝난 후 터미널을 새로 실행하거나 source ~/.bashrc 명령으로 .bashrc 파일을 다시 실행한다.

1.4.4 파이썬 패키지 설치하고 업데이트하기

이 책을 읽다가 아나콘다 배포판에 포함되어 있지 않은 추가 파이썬 패키지를 설치해야 할 경우 다음 명령을 이용해서 설치할 수 있다.

```
conda install package_name
```

만일 위 명령으로 설치할 수 없을 경우 pip 패키지 관리 도구를 이용해서 설치할 수 있다.

```
pip install package_name
```

conda update 명령을 이용해서 패키지를 업데이트할 수 있다.

```
conda update package_name
```

pip 역시 --upgrade 플래그를 이용해서 패키지를 업데이트할 수 있다.

```
pip install --upgrade package_name
```

위에서 소개한 명령들은 이 책을 읽는 동안 사용하게 될 기회가 있을 것이다.

CAUTION_ 패키지를 설치하기 위해 conda와 pip를 함께 사용할 수 있지만 conda 패키지를 pip 명령으로 업데이트하면 환경 설정에 문제가 발생할 수 있다. 아나콘다나 미니콘다(Miniconda)를 사용하고 있다면 conda를 이용한 업데이트를 먼저 시도하는 것이 안전하다.

1.4.5 파이썬 2와 파이썬 3

파이썬 3의 최초 버전은 2008년 말에 릴리스되었다. 파이썬 3에는 파이썬 2 코드와의 호환성을 유지할 수 없는 몇 가지 변경 사항이 포함되어 있었는데, 1991년 파이썬이 최초로 릴리스된 이후 17년이 지났고 이 호환성을 '포기하는' 릴리스는 언어를 발전시키기 위한 필수불가결한 결정으로 여겨졌다.

2012년까지만 하더라도 과학 계산과 데이터 분석 커뮤니티에서는 많은 패키지가 파이썬 3를 완벽하게 지원하지 않는다는 이유로 파이썬 2를 주로 사용하고 있었다. 이 책의 초판도 파이썬 2.7을 사용했지만 이제는 파이썬 2 혹은 파이썬 3를 선택하더라도 라이브러리 지원이 문제가 되는 경우는 없을 것이다.

파이썬 2는 2020년에 완전히 개발이 중단되고 치명적인 보안 패치조차 제공되지 않을 예정이므로 새로운 프로젝트를 파이썬 2.7로 시작하는 것은 현명한 선택이 아니다. 따라서 이 책에서도 널리 사용되고 있으며 잘 지원되며 안정적인 버전인 파이썬 3.6을 사용한다. 파이썬 2.x는 이제 레거시 파이썬이라고 부르며 파이썬 3.x를 그냥 파이썬이라고 부르기 시작했다. 독자들도 이제 파이썬 3에 동참하기 바란다.

이 책의 예제는 파이썬 3.6을 기반으로 하여 작성했다. 그러므로 더 높은 버전의 파이썬을 사용하면 예제들이 문제없이 동작할 테지만, 2.7 버전에서는 다르게 동작하거나 아예 동작하지 않을 수 있다.

1.4.6 통합 개발 환경과 텍스트 편집기

내게 기본 개발 환경에 대해서 물어오면 거의 항상 'IPython과 텍스트 편집기'라고 대답한다. 나는 보통 프로그램을 작성하고 나서 습관처럼 IPython에서 각 코드를 테스트하고 디버깅한다. 이 방법은 데이터를 쌍방향으로 다룰 수 있으며 특정한 데이터 묶음에 대한 조작이 제대로 되고 있는지 시각적으로 확인이 가능해서 상당히 유용하다. pandas와 NumPy 같은 라이브러리는 셸에서 사용하기 쉽도록 설계되었다.

하지만 어떤 사람은 소프트웨어를 개발할 때 이맥스Emacs나 빔Vim: Vi IMproved 같은 텍스트 편집기 대신 IDE를 선호하기도 한다. 몇 가지 IDE를 소개한다.

- **파이데브**(PyDev): 이클립스(Eclipse) 플랫폼 기반의 IDE. 무료
- 젯브레인스의 **파이참**(PyCharm): 상용 버전은 구독 모델이고 오픈소스 개발자들은 무료
- **PTVS**: 비주얼 스튜디오의 파이썬 도구. 윈도우 사용자용
- **스파이더**(Spyder): 아나콘다와 함께 배포되는 IDE. 무료
- **코모도**(Komodo): 유료

파이썬의 인기 덕분에 아톰Atom이나 서브라임 텍스트 2 같은 대부분의 텍스트 편집기는 파이썬을 아주 잘 지원하고 있다.

1.5 커뮤니티와 컨퍼런스

인터넷 검색 외에 과학 계산과 데이터 관련 파이썬 메일링 리스트는 일반적으로 도움을 얻을 수 있는 곳이며 질문에 대한 답도 얻을 수 있다. 살펴보면 좋은 커뮤니티를 소개한다.

- **pydata**: pandas와 파이썬 데이터 분석 관련 질문을 위한 구글 그룹
- **pystatsmodels**: 통계 모델이나 pandas 관련 질문을 올리는 곳
- scikit-learn과 일반적인 파이썬 머신러닝의 **메일링 리스트**
- **numpy-discussion**: NumPy 관련 질문을 올리는 곳
- **scipy-user**: 일반적인 SciPy나 과학 계산 파이썬 관련 질문을 올리는 곳

URL은 바뀔 수 있어서 일부러 적지 않았다. 인터넷을 검색하면 쉽게 찾을 수 있다.

해마다 전 세계 파이썬 개발자들을 대상으로 많은 컨퍼런스가 열리고 있다. 다른 파이썬 개발

자들과 관심사를 공유하고 싶다면 꼭 참석하기 바란다. 대부분의 컨퍼런스는 티켓이나 여행경비를 부담할 수 없는 사람들을 위한 재정 지원 프로그램을 운영하고 있다.

- **파이콘**(PyCon)과 **유로파이썬**(EuroPython): 각각 북미 지역과 유럽에서 열리는 주요 파이썬 컨퍼런스
- **사이파이**(SciPy)와 **유로사이파이**(EuroSciPy): 과학 계산 파이썬 컨퍼런스. 각각 북미와 유럽 지역에서 열린다.
- **파이데이터**(PyData): 데이터 과학과 데이터 분석 사례에 초점을 맞춘 컨퍼런스. 전 세계 각지에서 열린다.
- **각 국가의 파이콘**. 전체 목록은 http://pycon.org에서 확인할 수 있다.

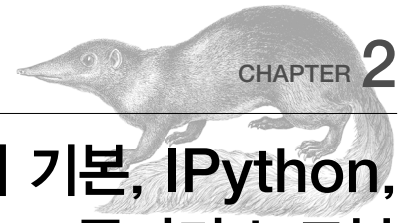
1.6 이 책을 살펴보는 방법

파이썬 프로그래밍 경험이 없다면 파이썬에서 제공하는 기능과 IPython 셸 및 주피터 노트북에 대한 요약 튜토리얼인 2장과 3장을 먼저 살펴보기 바란다. 이들 내용은 이 책을 공부하기 위해 필요한 사전지식이다. 이미 파이썬을 사용해본 경험이 있다면 가볍게 훑어보거나 건너뛰어도 좋다.

다음에는 NumPy의 핵심 기능을 간단히 소개한다(자세한 사용법은 부록 A에 따로 담았다). 또한 pandas를 소개하고 pandas와 NumPy 그리고 시각화를 위해 matplotlib을 사용하여 데이터 분석 주제를 다뤄볼 것이다. 가능한 한 점진적으로 진행되도록 책을 구성했으나 가끔 가볍게 겹치는 내용이 있을 수 있다.

독자마다 최종적으로 원하는 목표가 다를 수 있지만 크게 다음과 같은 업무를 필요로 하게 된다.

- **외부와 연동하기**
  다양한 파일 포맷과 데이터 저장소로부터 데이터를 읽고 쓰기
- **데이터 준비**
  데이터 분석을 위해 데이터를 정제, 조합, 정규화, 변형, 다듬는 작업
- **변환**
  수학이나 통계 작업을 통해 새로운 데이터셋을 도출(그룹 변수를 이용하여 큰 테이블 데이터를 집계)
- **모델링과 연산**
  통계 모델, 머신러닝 알고리즘 또는 다른 연산 도구를 데이터와 연동하기
- **프레젠테이션**
  대화형, 정적 시각화 또는 텍스트 요약 생성



CHAPTER 2

파이썬 언어의 기본, IPython, 주피터 노트북

2011년과 2012년에 걸쳐 처음 이 책의 초판을 집필할 때 데이터 분석을 위한 파이썬을 공부할 수 있는 자료가 충분치 않았다. 이는 닭과 달걀 문제와 비슷한데, 지금은 충분히 성숙한 pandas, scikit-learn 그리고 statsmodels 같은 라이브러리가 그 당시에는 상대적으로 불완전했었다. 2017년 기준으로 데이터 과학, 데이터 분석 그리고 머신러닝에 관한 문서가 많아졌으며 계산 과학자, 물리학자를 포함하여 다른 연구 분야의 전문가들을 위한 범용 과학 계산 작업용 보충 자료도 많아졌다. 파이썬 프로그래밍 언어를 배우고 유효한 소프트웨어 엔지니어가 되기 위한 훌륭한 책도 많이 출간되었다.

이 책은 파이썬을 활용한 데이터 업무의 입문서 역할을 하려는 목적으로 집필되었기에 나는 파이썬의 내장 자료구조와 라이브러리를 데이터를 다루는 관점에서 소개하는 것이 더 의미가 있을 거라고 생각한다. 그러므로 이 장과 다음 장에서는 이 책을 계속 읽어나가는 데 필요한 최소한의 정보만 소개할 것이다.

개인적으로는 데이터 분석을 생산적으로 하기 위해 파이썬으로 훌륭한 소프트웨어를 개발할 수 있을 정도로 파이썬 고수가 되어야 할 필요는 없다고 생각한다. 코드 예제를 실행해보고 파이썬의 다양한 자료형, 함수, 메서드에 대한 문서를 찾아보는 데 IPython 셸과 주피터 노트북을 사용할 것을 적극 권장한다. 비록 책에서 소개하는 내용들을 점진적인 형태로 만들기 위해 노력을 기울이긴 했지만 여기서 소개하는 코드 예제의 일부는 완전하게 설명하지 않는 것도 있음을 기억하자.

이 책의 대부분은 대용량 데이터를 다루기 위한 테이블 기반의 분석과 데이터 준비 도구에 초

점을 맞추고 있다. 이 도구들을 사용하기 위해서는 제멋대로인 데이터를 처리하기 쉽도록 깔끔하게 **구조화된** 형태로 다듬어야 한다. 다행히도 파이썬은 그런 데이터를 원하는 모양으로 쉽게 다듬을 수 있는 이상적인 언어다. 파이썬을 사용하는 데 익숙해지면 분석을 위해 데이터를 준비하는 과정이 좀 더 수월해진다.

이 책에서 소개하는 일부 도구는 IPython 또는 주피터로 살펴보는 것이 가장 효과적이다. IPython과 주피터를 어떻게 실행시키는지 배우고 난 다음에는 예제를 따라 해보고 다른 것들을 시도해보기 추천한다. 키보드를 주로 사용하게 되는 콘솔 같은 환경에서는 일반적인 명령어에 완전히 익숙해지는 것도 학습 과정에 포함된다.

> **NOTE_** 파이썬을 이용한 데이터 분석 여행에서 유용하다고 생각할 수 있는 클래스나 객체지향 프로그래밍 같은 파이썬의 기본 개념은 이 책에서 다루지 않는다.
>
> 파이썬 언어에 대한 지식을 더 알고자 하는 독자는 공식 파이썬 튜토리얼과 범용 파이썬 프로그래밍을 소개하는 훌륭한 다른 책을 더 보길 추천한다. 아래는 추천하는 도서 목록이다.
>
> - 『파이썬 쿡북 3판』(데이비드 비즐리, 브라이언 K 존스, 인피니티북스)
> - 『전문가를 위한 파이썬』(루시아누 하말류, 한빛미디어)
> - 『이펙티브 파이썬』(브렛 슬래킨, 길벗)

2.1 파이썬 인터프리터

파이썬은 **인터프리터** 언어다. 파이썬 인터프리터는 한 번에 하나의 명령어만 실행한다. 파이썬 표준 인터프리터는 명령행에서 python을 입력해서 실행한다.

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

>>>는 파이썬 인터프리터의 **프롬프트**인데, 여기에 코드를 입력한다. 파이썬 인터프리터를 종

료하고 명령행 프롬프트로 돌아가려면 **exit()**를 입력하거나 Ctrl-D를 누른다.

파이썬 프로그램을 실행할 때는 첫 번째 인자로 .py 파일을 넘긴다. 예를 들어 다음 내용으로 hello_world.py 파일을 작성했다고 하자.

```
print('Hello world')
```

터미널에서 다음과 같이 실행할 수 있다(hello_world.py 파일은 반드시 현재 작업하고 있는 터미널의 디렉터리에 존재해야 한다).

```
$ python hello_world.py
Hello world
```

일부 파이썬 개발자는 이런 식으로 파이썬 코드를 실행하며, 데이터 분석이나 과학 계산 파이썬 개발자들은 향상된 대화형 파이썬 인터프리터인 IPython 또는 IPython 프로젝트에서 만든 웹 기반의 주피터 노트북을 사용한다. 여기서는 IPython과 주피터를 간단히 소개하고 부록 A에서 IPython을 더 자세히 살펴본다. `%run` 명령어를 사용하면 IPython은 지정된 파일의 코드를 같은 프로세스 안에서 실행하여 실행이 끝났을 때 그 결과를 인터랙티브하게 탐색할 수 있게 해준다.

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world

In [2]:
```

표준 프롬프트는 >>>인 반면 IPython의 프롬프트는 `In [2]:`와 같은 형식으로 번호가 붙는다.

2.2 IPython 기초

이 절에서는 IPython 셸과 주피터 노트북을 실행하는 방법, 주요 콘셉트를 소개한다.

2.2.1 IPython 셸 실행하기

IPython은 일반 파이썬 인터프리터를 실행시키듯이 ipython 명령어를 입력해서 실행시킬 수 있다.

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: a = 5

In [2]: a
Out[2]: 5
```

파이썬 코드를 입력하고 Enter 키를 눌러서 실행시킬 수 있다. 파이썬에 그냥 변수 이름만 입력하면 그 객체의 문자열 표현을 출력한다.

```
In [5]: import numpy as np

In [6]: data = {i : np.random.randn() for i in range(7)}

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
 6: 0.09290787674371767}
```

첫 두 줄은 파이썬 코드인데, 두 번째 줄에서 data라는 이름의 변수를 생성하고 새로 생성한 파이썬 사전형(딕셔너리)을 참조하도록 했다. 마지막 줄은 data 변수의 값을 출력한다.

대부분의 파이썬 객체는 print를 이용한 보통의 출력 결과와는 달리 좀 더 읽기 편하거나 **보기 좋은 형태로 출력**된다. 위 예제를 표준 파이썬 인터프리터에서 보면 다소 읽기 불편한 형태로 출력된다.

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print(data)
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,
6: 0.3308507317325902}
```

또한 IPython은 일부 코드(약간 개선된 복사/붙여넣기 기능을 통해) 아니면 전체 파이썬 스크립트를 쉽게 실행할 수 있는 기능을 제공한다. 또한 주피터 노트북을 이용해서 많은 분량의 코드를 다룰 수 있는데 나중에 더 살펴보기로 하자.

2.2.2 주피터 노트북 실행하기

주피터 프로젝트의 주요 구성요소 중 하나인 **노트북**은 코드, 텍스트, 데이터 시각화를 비롯한 다른 출력을 대화형으로 구성할 수 있는 대화형 문서 형식이다. 주피터 노트북은 어떤 프로그래밍 언어로도 작성 가능한 주피터 대화형 컴퓨팅 프로토콜의 구현체인 **커널**과 상호작용한다. 파이썬 주피터 커널은 IPython 시스템을 이용하여 동작한다.

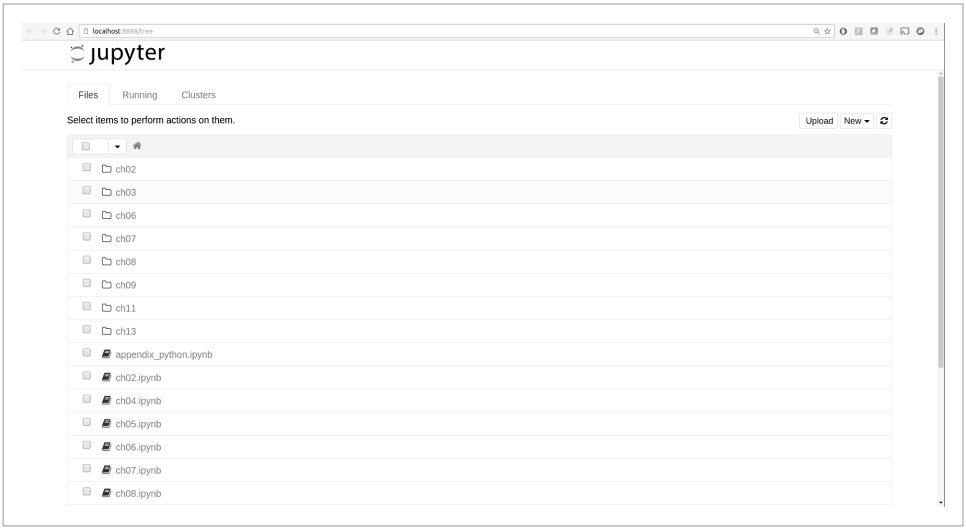
주피터를 실행하려면 터미널에 jupyter notebook 명령을 입력하면 된다.

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

대부분의 플랫폼에서는 `--no-browser` 옵션을 지정하지 않으면 자동으로 기본 웹 브라우저를 실행한다. 그렇지 않다면 노트북을 실행했을 때 출력되는 웹 주소(여기서는 `http://localhost:8888/`)로 접속하면 된다. 구글 크롬 브라우저에서의 실행 화면을 [그림 2-1]에서 확인할 수 있다.

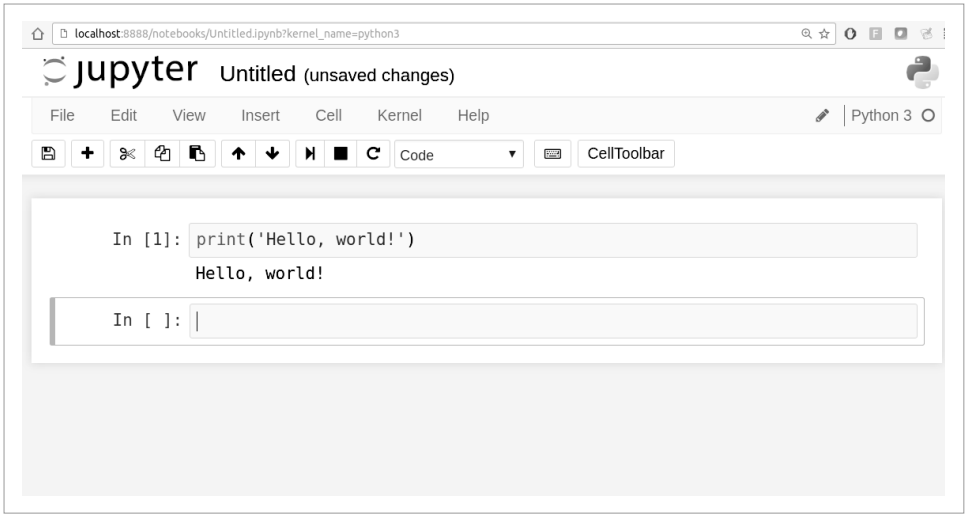
> **NOTE_** 많은 사람이 주피터를 로컬 개발 환경으로 사용하고 있다. 하지만 서버에 설치해두고 원격으로 접속해서 사용하는 것도 가능하다. 이 책에서는 원격 접속 설정은 다루지 않는다. 필요하다면 인터넷에서 관련 주제를 검색해보기 추천한다.

그림 2-1 주피터 노트북 랜딩 페이지



새로운 노트북을 생성하려면 New 버튼을 클릭하고 'Python 3'나 'conda[default]' 옵션을 선택한다. 그러면 [그림 2-2]와 유사한 화면을 보게 될 것이다. 처음 실행한다면 비어 있는 코드 '셀'을 클릭하고 파이썬 코드를 입력해보자. Shift-Enter를 누르면 코드를 실행할 수 있다.

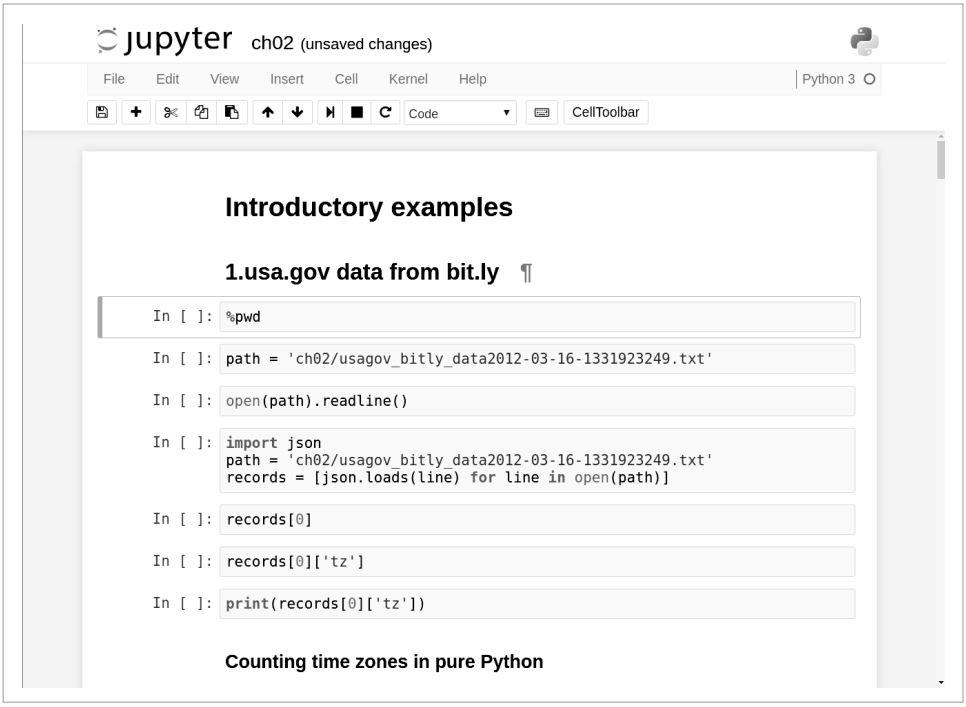
그림 2-2 새 주피터 노트북 화면



노트북 파일을 저장하면(노트북의 File 메뉴에서 Save and Checkpoint 클릭) 확장자가 .ipynb인 파일이 만들어진다. 이 파일에는 현재 노트북 화면의 모든 내용이 담겨 있다. 다른 주피터 사용자가 불러와서 편집할 수 있는 파일이다. 기존 노트북 파일을 불러오려면 주피터 노트북을 실행한 경로 혹은 그 하위 폴더에 파일을 옮겨두고 랜딩 페이지에서 해당 파일을 더블 클릭하면 된다. 깃허브 저장소 wesm/pydata-book에 올려둔 노트북 파일로 실습해보기 바란다. [그림 2-3]을 보자.

주피터 노트북은 IPython과는 완전히 다른 경험을 주기는 하지만 이 장에서 설명하는 거의 모든 명령과 도구는 양쪽 환경 모두에서 사용할 수 있다.

그림 2-3 기존 노트북 파일을 불러온 화면



2.2.3 탭 자동완성

겉으로 보기에 IPython은 표준 파이썬 인터프리터와는 조금 다르게 생겼다. 표준 파이썬 셸에 비해 가장 두드러진 개선은 **탭을 통한 자동완성 기능**으로, 대부분의 통합 개발 환경이나 대화형 데이터 분석 환경에 구현되어 있는 기능이다. 셸에서 입력을 하는 동안 탭을 누르면 네임스페이스에서 그 시점까지 입력한 내용과 맞아떨어지는 변수(객체, 함수 등)를 자동으로 찾아준다.

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and        an_example  any
```

이 예제에서 IPython은 앞서 정의한 두 변수는 물론이고 파이썬 예약어인 and와 내장 함수인 any를 함께 보여주는 것을 확인할 수 있다. 물론 어떤 객체의 메서드나 속성 뒤에 마침표를 입력한 후 자동완성 기능을 활용할 수도 있다.

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append  b.count   b.insert  b.reverse
b.clear   b.extend  b.pop     b.sort
b.copy    b.index   b.remove
```

모듈도 똑같이 동작한다.

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date          datetime.MAXYEAR       datetime.timedelta
datetime.datetime      datetime.MINYEAR       datetime.timezone
datetime.datetime_CAPI datetime.time          datetime.tzinfo
```

주피터 노트북과 IPython 5.0 이상 버전에서는 자동완성 목록이 일반 텍스트 출력이 아니라 드롭다운 형식으로 나타난다.

NOTE_ 탭을 눌렀을 때 화면에 출력 결과가 너무 많으면 초보자는 헷갈릴 수 있는데, IPython은 아예 _로 시작하는 내부 메서드와 속성을 제외한 결과를 보여준다. 물론 먼저 _를 입력하면 해당 메서드와 속성을 선택할 수 있다. 이런 메서드를 탭 자동완성 목록에 기본으로 넣고 싶다면 IPython 환경 설정에서 설정할 수 있다. 자세한 내용은 IPython 문서를 참고하자.

탭 자동완성은 대화형 네임스페이스 검색과 객체 및 모듈 속성의 자동완성뿐만 아니라 파일 경로를 입력(파이썬 문자열 안에서도)한 후 탭을 누르면 입력한 문자열에 맞는 파일 경로를 컴퓨터의 파일 시스템 안에서 찾아서 보여준다.

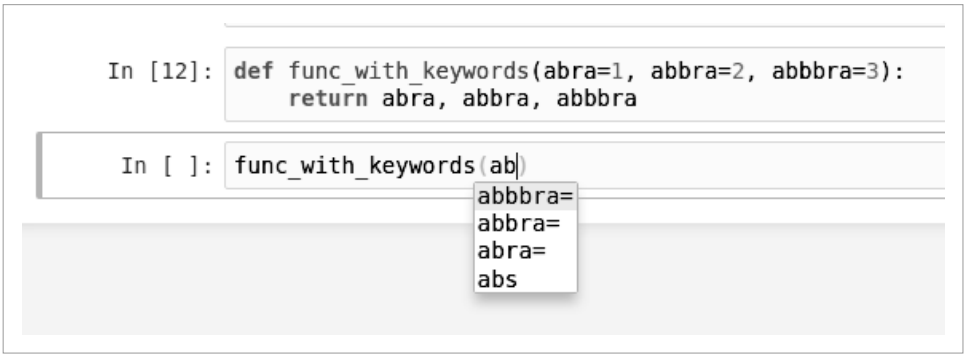
```
In [7]: datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/README
datasets/movielens/ratings.dat   datasets/movielens/users.dat
```

```
In [7]: path = 'datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/README
datasets/movielens/ratings.dat   datasets/movielens/users.dat
```

나중에 살펴볼 %run 명령어(2.2.5절 참조)와 조합하면 키 입력을 줄일 수 있다.

또한 자동완성 기능을 사용하면 함수에서 이름을 가진 인자(= 기호까지 포함해서)도 보여준다. [그림 2-4]를 참조하자.

그림 2-4 주피터 노트북에서 함수 키워드 자동완성



이런 함수들은 나중에 좀 더 살펴보기로 하자.

2.2.4 자기관찰

변수 이름 앞이나 뒤에 ? 기호를 붙이면 그 객체에 대한 일반 정보를 출력한다.

```
In [8]: b = [1, 2, 3]

In [9]: b?
Type:       list
String Form:[1, 2, 3]
Length:     3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

```
In [10]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

이 기능은 객체의 **자기관찰**(인트로스펙션introspection)이라고 하는데, 만약 객체가 함수이거나 인스턴스 메서드라면 정의되어 있는 문서docstring를 출력해준다. IPython이나 주피터에서 사용 가능한 아래와 같은 함수를 작성했다고 하자.

```
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

? 기호를 사용해서 문서를 출력해보자.

```
In [11]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Add two numbers together

Returns
-------
the_sum : type of arguments
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

??를 사용하면 가능한 경우 함수의 소스 코드도 보여준다.

```
In [12]: add_numbers??
Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

또한 ?는 표준 유닉스나 윈도우 명령행에서와 마찬가지로 IPython의 네임스페이스를 검색하는 데 사용할 수도 있다. 와일드카드(*) 기호와 함께 사용한 문자와 일치하는 모든 이름을 보여준다. 예를 들어 NumPy의 최상단 네임스페이스 안에서 load를 포함하는 모든 함수 목록을 가져올 수 있다.

```
In [13]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

2.2.5 %run 명령어

%run 명령어를 사용하면 IPython 세션 안에서 파이썬 프로그램 파일을 불러와서 실행할 수 있다. 다음과 같은 ipython_script_test.py 스크립트 파일이 있다고 하자.

```
def f(x, y, z):
    return (x + y) / z
```

```
a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

이 스크립트 파일은 %run 명령을 사용해서 다음처럼 실행할 수 있다.

```
In [14]: %run ipython_script_test.py
```

스크립트 파일은 **빈 네임스페이스**(다른 변수가 선언되지 않았거나 아무것도 임포트되지 않은 상태)에서 실행되므로 명령행에서 python script.py 명령을 실행한 것과 동일하게 동작한다. 스크립트 파일에 정의된 모든 변수(임포트, 함수, 전역 변수)는 실행된 후에 IPython에서 접근이 가능하다.

```
In [15]: c
Out[15]: 7.5

In [16]: result
Out[16]: 1.4666666666666666
```

만약 파이썬 스크립트에 명령행 인자(sys.argv에 저장되는)를 넘겨야 한다면 명령행에서 실행하는 것처럼 파일 경로 다음에 필요한 인자를 넘겨주면 된다.

NOTE_ 실행하려는 스크립트 파일에서 대화형 IPython 네임스페이스에 미리 선언된 변수에 접근해야 한다면 %run 대신 %run -i 명령을 사용한다.

주피터 노트북에서는 스크립트 파일을 코드 셀로 불러오는 %load 매직함수를 사용할 수도 있다.

```
>>> %load ipython_script_test.py

    def f(x, y, z):
        return (x + y) / z

    a = 5
```

```
b = 6
c = 7.5

result = f(a, b, c)
```

실행 중인 코드 중지하기

%run을 통해 스크립트가 실행되고 있거나 오랜 실행 시간이 필요한 코드가 실행되고 있는 중간에 Ctrl-C를 누르면 KeyboardInterrupt 예외를 발생시킨다. 이 예외는 몇몇 특수한 경우를 제외한 거의 모든 파이썬 프로그램을 즉시 멈추도록 한다.

> **CAUTION_** 실행 중인 파이썬 코드가 컴파일된 확장 모듈에서 호출된 경우에는 Ctrl-C를 눌러도 프로그램이 즉각 멈추지 않는데, 이런 경우에는 프로그램의 제어권이 파이썬 인터프리터로 되돌아올 때까지 기다리거나 심각한 경우에는 운영체제의 작업 관리자 메뉴를 이용해 파이썬 프로세스를 강제로 종료해야 한다.

2.2.6 클립보드에 있는 코드 실행하기

주피터 노트북을 사용하고 있다면 아무 코드 셀에나 코드를 붙여넣기해서 실행할 수 있다. IPython 셸에서도 클립보드의 내용을 실행하는 것이 가능하다. 다른 애플리케이션에서 사용 중인 다음 코드가 있다고 하자.

```
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
```

이 경우 %paste나 %cpaste 매직함수를 사용할 수 있다. %paste는 클립보드에 있는 텍스트를 단일 블록으로 실행한다.

```
In [17]: %paste
x = 5
```

```
y = 7
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```

%cpaste는 %paste와 유사하지만 코드를 붙여 넣을 때 특수한 프롬프트를 제공한다.

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:    y = 8
:--
```

%cpaste 블록을 사용하면 실행 전에 코드를 마음껏 붙여 넣을 수 있다. 붙여 넣은 코드를 실행하기 전에 한 번 살펴보기 위해 %cpaste 기능을 사용할 수 있다. 실수로 잘못된 코드를 붙여 넣었다면 Ctrl-C를 눌러 %cpaste 프롬프트를 빠져나올 수 있다.

2.2.7 키보드 단축키

IPython은 이맥스 편집기나 유닉스 bash shell(베시 셸) 사용자에게 친숙한 프롬프트 이동 단축키를 제공하며 이전에 입력한 셸의 명령어 히스토리를 사용할 수 있다. [표 2-1]에 주로 사용되는 단축키를 요약해두었다. [그림 2-5]는 커서 이동 같은 몇 가지 단축키의 사용법이다.

그림 2-5 IPython 키보드 단축키

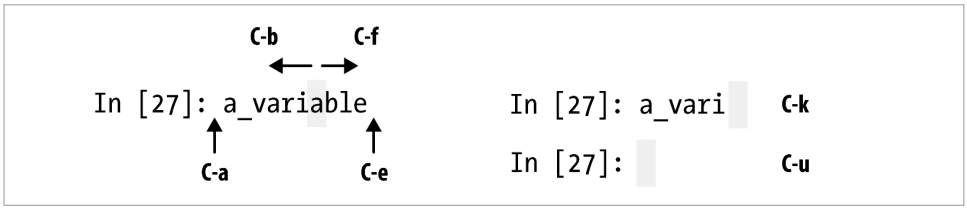


표 2-1 표준 IPython 키보드 단축키

키보드 단축키	설명
Ctrl-P 또는 위 화살표 키	명령어 히스토리를 역순으로 검색하기
Ctrl-N 또는 아래 화살표 키	명령어 히스토리에서 최근 순으로 검색하기
Ctrl-R	readline 형식의 히스토리 검색(부분 매칭)하기
Ctrl-Shift-V	클립보드에서 텍스트 붙여넣기
Ctrl-C	현재 실행 중인 코드 중단하기
Ctrl-A	커서를 줄의 처음으로 이동하기
Ctrl-E	커서를 줄의 끝으로 이동하기
Ctrl-K	커서가 놓인 곳부터 줄의 끝까지 텍스트 삭제하기
Ctrl-U	현재 입력된 모든 텍스트 지우기
Ctrl-F	커서를 앞으로 한 글자씩 이동하기
Ctrl-B	커서를 뒤로 한 글자씩 이동하기
Ctrl-L	화면 지우기

주피터 노트북은 편집과 이동에 관련된 단축키를 분리하여 제공한다. 단축키가 IPython보다 더 빠르게 진화하고 있으므로 주피터 노트북 메뉴에서 통합된 도움말을 살펴보기 권장한다.

2.2.8 매직 명령어

IPython은 파이썬 자체에는 존재하지 않는 '매직' 명령어라고 하는 여러 가지 특수한 명령어를 포함하고 있다. 이 매직 명령어는 일반적인 작업이나 IPython 시스템의 동작을 쉽게 제어할 수 있도록 설계된 특수한 명령어다. 매직 명령어는 앞에 % 기호를 붙이는 형식인데, 예를 들어 IPython에서 행렬 곱셈 같은 코드가 실행된 시간을 측정하고 싶을 때는 `%timeit` 매직함수를 이용해서 값을 얻어올 수 있다. `%timeit`은 나중에 좀 더 자세히 설명하겠다.

```
In [20]: a = np.random.randn(100, 100)

In [20]: %timeit np.dot(a, a)
10000 loops, best of 3: 20.9 µs per loop
```

매직 명령어는 IPython 시스템 안에서 실행되는 명령행 프로그램으로 볼 수 있는데, 많은 매직 명령어는 추가적인 옵션을 필요로 하며 ?를 이용해서 전체 옵션을 살펴볼 수 있다.

CHAPTER 3

내장 자료구조, 함수, 파일

이 장에서는 이 책 전반에 걸쳐 사용하게 될 파이썬 언어에 내장되어 있는 기능을 알아보자. pandas나 NumPy 같은 애드온 라이브러리는 대규모 데이터 계산을 위한 진보된 기능을 제공하지만, 내장되어 있는 기능은 파이썬 내장 자료 처리 도구와 함께 사용해야 한다.

먼저 파이썬의 기본 자료구조인 튜플, 리스트, 사전 그리고 집합부터 알아보고 재사용 가능한 파이썬 함수를 작성하는 방법을 살펴본다. 마지막으로 파이썬의 file 객체의 원리를 살펴보고 하드디스크에 직접 파일을 읽고 쓰는 방식을 알아보자.

3.1 자료구조와 순차 자료형

파이썬의 자료구조는 단순하지만 강력하다. 자료구조의 사용법을 숙지하는 것이 파이썬의 고수가 되는 지름길이다.

3.1.1 튜플

튜플은 1차원의 고정된 크기를 가지는 변경 불가능한 순차 자료형이다. 튜플을 생성하는 가장 쉬운 방법은 쉼표로 구분된 값을 대입하는 것이다.

```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)
```

괄호를 사용해서 값을 묶어줌으로써 중첩된 튜플을 정의할 수 있다. 아래 예제는 튜플의 튜플을 생성한다.

```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

모든 순차 자료형이나 이터레이터는 tuple 메서드를 호출해 튜플로 변환할 수 있다.

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

튜플의 각 원소는 대괄호 []를 이용해서 다른 순차 자료형처럼 접근할 수 있다. C, C++, 자바 그리고 다른 많은 언어처럼 순차 자료형의 색인은 0부터 시작한다.

```
In [8]: tup[0]
Out[8]: 's'
```

튜플에 저장된 객체 자체는 변경이 가능하지만 한 번 생성되면 각 슬롯에 저장된 객체를 변경하는 것은 불가능하다. 다음 예제를 보자.

```
In [9]: tup = tuple(['foo', [1, 2], True])
In [10]: tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-10-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

튜플 내에 저장된 객체는 그 위치에서 바로 변경이 가능하다.

```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

+ 연산자를 이용해서 튜플을 이어붙일 수 있다.

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

튜플에 정수를 곱하면 리스트와 마찬가지로 튜플의 복사본이 반복되어 늘어난다.

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

튜플 안에 있는 객체는 복사되지 않고 그 객체에 대한 참조만 복사된다는 점을 기억하자.

튜플에서 값 분리하기

만일 튜플과 같은 표현의 변수에 튜플을 **대입**하면 파이썬은 등호(=) 오른쪽에 있는 변수에서 값을 **분리**한다.

```
In [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

중첩된 튜플을 포함하는 순차 자료형에서도 값을 분리해낼 수 있다.

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```

이 기능을 사용하면 변수의 이름을 바꿀 때 다른 언어에서는 아래처럼 처리하는 것을 쉽게 해결할 수 있다.

```
tmp = a
a = b
b = tmp
```

즉, 파이썬에서는 다음과 같이 하여 두 변수의 값을 쉽게 바꿀 수 있다.

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

튜플이나 리스트를 순회할 때도 흔히 이 기능을 활용한다.

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in seq:
   ....:     print('a={0}, b={1}, c={2}'.format(a, b, c))
```

```
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

이 기능은 함수에서 여러 개의 값을 반환할 때도 자주 사용하는데 이는 나중에 다시 살펴보겠다.

파이썬은 최근에 튜플의 처음 몇몇 값을 '끄집어내야' 하는 상황을 위해 튜플에서 값을 분리하는 좀 더 진보된 방법을 수용했다. 이 경우 특수한 문법인 *rest를 사용하는데 함수의 시그니처에서 길이를 알 수 없는 긴 인자를 담기 위한 방법으로도 사용된다.

```
In [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

rest는 필요 없는 값을 무시하기 위해 사용하기도 한다. rest라는 이름 자체에는 특별한 의미가 없다. 불필요한 변수라는 것을 나타내기 위해 _를 사용하는 관습도 있다.

```
In [33]: a, b, *_ = values
```

튜플 메서드

튜플의 크기와 내용은 변경 불가능하므로 인스턴스 메서드가 많지 않다. 유용하게 사용되는 메서드 중 하나는 주어진 값과 같은 값이 몇 개 있는지 반환하는 count 메서드다(리스트에서도 사용 가능하다).

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 4
```

3.1.2 리스트

튜플과는 대조적으로 리스트는 크기나 내용의 변경이 가능하다. 리스트는 대괄호 []나 list 함수를 사용해서 생성할 수 있다.

```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'

In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

리스트와 튜플은 의미적으로 비슷한(비록 튜플을 수정할 수 없지만) 객체의 1차원 순차 자료형이며 많은 함수에서 교차적으로 사용할 수 있다.

list 함수는 이터레이터나 제너레이터 표현에서 실제 값을 모두 담기 위한 용도로도 자주 사용된다.

```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

원소 추가하고 삭제하기

append 메서드를 사용해서 리스트의 끝에 새로운 값을 추가할 수 있다.

```
In [45]: b_list.append('dwarf')
```

```
In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

insert 메서드를 사용해서 리스트의 특정 위치에 값을 추가할 수 있다.

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

값을 추가하려는 위치는 0부터 리스트 길이까지의 값이어야 한다.

CAUTION_ insert는 append에 비해 연산비용이 많이 든다. insert로 값을 추가하면 추가된 위치 이후의 원소들은 새로 추가될 원소를 위해 내부적으로 모두 자리를 옮겨야 하기 때문이다. 순차 자료형의 시작과 끝 지점에 원소를 추가하고 싶다면 이런 용도로 사용할 수 있는 양방향 큐인 collections.deque를 사용하자.

insert 메서드와 반대 개념으로 pop 메서드가 있다. pop 메서드는 특정 위치의 값을 반환하고 해당 값을 리스트에서 삭제한다.

```
In [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

remove 메서드를 이용해서 원소를 삭제할 수 있는데, 삭제는 리스트에서 제일 앞에 위치한 값부터 이루어진다.

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

성능이 큰 문제가 되지 않는다면 append와 remove 메서드를 사용해서 파이썬의 리스트를 여러 종류의 데이터를 담을 수 있는 자료구조로 사용할 수 있다.

in 예약어를 사용해서 리스트에 어떤 값이 있는지 검사할 수 있다.

```
In [55]: 'dwarf' in b_list
Out[55]: True
```

not 예약어를 사용해서 in 예약어를 반대 의미로 사용할 수도 있다.

```
In [56]: 'dwarf' not in b_list
Out[56]: False
```

리스트에 어떤 값이 있는지 검사하는 것은 리스트의 모든 값을 일일이 검사해야 하므로 해시 테이블을 사용한 파이썬의 사전이나 집합 자료구조처럼 즉각적으로 반환하지 않고 많이 느리다는 점을 기억하자.

리스트 이어붙이기

튜플과 마찬가지로 + 연산자를 이용하면 두 개의 리스트를 합칠 수 있다.

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]
```

만일 리스트를 미리 정의해두었다면 extend 메서드를 사용해서 여러 개의 값을 추가할 수 있다.

```
In [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```

리스트를 이어붙이면 새로운 리스트를 생성하고 값을 복사하게 되므로 상대적으로 연산비용이 높다는 점을 기억하자. 큰 리스트일수록 extend 메서드를 사용해서 기존의 리스트에 값을 추가하는 것이 일반적으로 더 나은 선택이다.

CHAPTER 4

NumPy 기본: 배열과 벡터 연산

NumPy는 Numerical Python의 줄임말로, 파이썬에서 산술 계산을 위한 가장 중요한 필수 패키지 중 하나다. 과학 계산을 위한 대부분의 패키지는 NumPy의 배열 객체를 데이터 교환을 위한 공통 언어처럼 사용한다.

NumPy에서 제공하는 것들은 다음과 같다.

- 효율적인 다차원 배열인 ndarray는 빠른 배열 계산과 유연한 **브로드캐스팅** 기능을 제공한다.
- 반복문을 작성할 필요 없이 전체 데이터 배열을 빠르게 계산할 수 있는 표준 수학 함수
- 배열 데이터를 디스크에 쓰거나 읽을 수 있는 도구와 메모리에 적재된 파일을 다루는 도구
- 선형대수, 난수 생성기, 푸리에 변환 기능
- C, C++, 포트란으로 작성한 코드를 연결할 수 있는 C API

NumPy의 C API는 사용하기 쉬우므로 저수준 언어로 작성된 외부 라이브러리에 데이터를 전달하거나 반대로 외부 라이브러리에서 NumPy 배열 형태로 파이썬에 데이터를 전달하기 용이하다. 이 기능은 파이썬으로 레거시 C, C++, 포트란 코드를 감싸서 동적이며 쉽게 사용할 수 있는 인터페이스를 만들 수 있도록 해준다.

NumPy 자체는 모델링이나 과학 계산을 위한 기능을 제공하지 않으므로 먼저 NumPy 배열과 배열 기반 연산에 대한 이해를 한 다음 pandas 같은 배열 기반 도구를 사용하면 훨씬 더 효율적이다. NumPy만으로도 방대한 주제이므로 브로드캐스팅 같은 NumPy의 고급 기능은 부록 A에서 따로 다루도록 하겠다.

대부분의 데이터 분석 애플리케이션에서 내가 중요하게 생각하는 기능은 다음과 같다.

- 벡터 배열 상에서 데이터 가공(데이터 먼징 또는 데이터 랭글링), 정제, 부분집합, 필터링, 변형 그리고 다른 여러 종류의 연산을 빠르게 수행
- 정렬, 유일 원소 찾기, 집합 연산 같은 일반적인 배열 처리 알고리즘
- 통계의 효과적인 표현과 데이터를 수집 요약하기
- 다양한 종류의 데이터를 병합하고 엮기 위한 데이터 정렬과 데이터 간의 관계 조작
- 내부에서 `if-elif-else`를 사용하는 반복문 대신 사용할 수 있는 조건절 표현을 허용하는 배열 처리
- 데이터 묶음 전체에 적용할 수 있는 수집, 변형, 함수 적용 같은 데이터 처리

NumPy는 일반적인 산술 데이터 처리를 위한 기반 라이브러리를 제공하기 때문에 많은 독자가 통계나 분석, 특히 표 형식의 데이터를 처리하기 위해 pandas를 사용하기 원할 것이다. 또한 pandas는 NumPy에는 없는 시계열 처리 같은 다양한 도메인 특화 기능을 제공한다.

> **NOTE_** 파이썬에서 배열 기반 연산을 시도했던 기록은 짐 헝구닌(Jim Hungunin)이 Numeric 라이브러리를 작성했던 1995년까지 거슬러 올라간다. 이후로 10년이 지나고 많은 과학 계산 커뮤니티는 배열 프로그래밍에 파이썬을 사용하기 시작했으나 라이브러리 생태계는 2000년대 초에 갈라지게 된다. 2005년 트라비스 올리펀트(Travis Oliphant)가 당시의 Numeric 라이브러리와 Numarray 프로젝트로부터 NumPy 프로젝트를 시작하여 커뮤니티에 단일 배열 컴퓨팅 프레임워크를 소개하게 되었다.

NumPy가 파이썬 산술 계산 영역에서 중요한 위치를 차지하는 이유 중 하나는 대용량 데이터 배열을 효율적으로 다룰 수 있도록 설계되었다는 점이다. 이에 대해 좀 더 알아보자.

- NumPy는 내부적으로 데이터를 다른 내장 파이썬 객체와 구분된 연속된 메모리 블록에 저장한다. NumPy의 각종 알고리즘은 모두 C로 작성되어 타입 검사나 다른 오버헤드 없이 메모리를 직접 조작할 수 있다. NumPy 배열은 또한 내장 파이썬의 연속된 자료형들보다 훨씬 더 적은 메모리를 사용한다.
- NumPy 연산은 파이썬 반복문을 사용하지 않고 전체 배열에 대한 복잡한 계산을 수행할 수 있다.

성능 차이를 확인하기 위해 백만 개의 정수를 저장하는 NumPy 배열과 파이썬 리스트를 비교해 보자.

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

이제 각각의 배열과 리스트 원소에 2를 곱해보자.

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

NumPy를 사용한 코드가 순수 파이썬으로 작성한 코드보다 열 배에서 백 배 이상 빠르고 메모리도 더 적게 사용하는 것을 확인할 수 있다.

4.1 NumPy ndarray: 다차원 배열 객체

NumPy의 핵심 기능 중 하나는 ndarray라고 하는 *N*차원의 배열 객체인데 파이썬에서 사용할 수 있는 대규모 데이터 집합을 담을 수 있는 빠르고 유연한 자료구조다. 배열은 스칼라 원소 간의 연산에 사용하는 문법과 비슷한 방식을 사용해서 전체 데이터 블록에 수학적인 연산을 수행할 수 있도록 해준다.

파이썬 내장 객체의 스칼라값을 다루는 것과 유사한 방법으로 에서 배치 계산을 처리하는 방법을 알아보기 위해 우선 NumPy 패키지를 임포트하고 임의의 값이 들어 있는 작은 배열을 만들어보겠다.

```
In [12]: import numpy as np

# 임의의 값을 생성
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047, 0.4789, -0.5194],
       [-0.5557, 1.9658, 1.3934]])
```

그리고 그 값에 산술 연산을 해보자.

```
In [15]: data * 10
Out[15]:
array([[ -2.0471, 4.7894, -5.1944],
       [ -5.5573, 19.6578, 13.9341]])

In [16]: data + data
Out[16]:
array([[-0.4094, 0.9579, -1.0389],
       [-1.1115, 3.9316, 2.7868]])
```

첫 번째 예제는 모든 원소의 값에 10을 곱했다. 두 번째 예제는 data 배열에서 같은 위치의 값끼리 서로 더했다.

> **NOTE_** 이 장과 책 전체에서 NumPy를 임포트할 경우 `import numpy as np` 컨벤션을 사용한다. 그냥 `from numpy import *`라고 해서 np를 입력하지 않아도 되지만 이런 습관은 지양하기 바란다. numpy 네임스페이스는 방대하고 파이썬 내장 함수와 같은 이름을 사용하는 경우(min과 max처럼)도 있기 때문이다.

ndarray는 같은 종류의 데이터를 담을 수 있는 포괄적인 다차원 배열이다. ndarray의 모든 원소는 같은 자료형이어야 한다. 모든 배열은 각 차원의 크기를 알려주는 shape라는 튜플과 배열에 저장된 **자료형**을 알려주는 dtype이라는 객체를 가지고 있다.

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

이 장에서는 NumPy의 배열을 사용하는 기초 방법을 소개하여 앞으로 책을 읽어나가는 데 충분한 도움을 줄 것이다. 대부분의 데이터 분석 애플리케이션을 작성하는 데 NumPy의 깊은 이해가 필수 사항은 아니며, 배열 위주의 프로그래밍과 생각하는 방법에 능숙해지는 것이 파이썬을 이용한 과학 계산의 고수가 되는 지름길이다.

NOTE_ 배열, NumPy 배열, ndarray는 아주 극소수의 예외를 제외하면 모두 ndarray 객체를 이르는 말이다.

4.1.1 ndarray 생성하기

배열을 생성하는 가장 쉬운 방법은 array 함수를 이용하는 것이다. 순차적인 객체(다른 배열도 포함하여)를 넘겨받고, 넘겨받은 데이터가 들어 있는 새로운 NumPy 배열을 생성한다. 예를 들어 파이썬의 리스트는 변환하기 좋은 예다.

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. ,7.5 , 8. , 0. , 1. ])
```

같은 길이를 가지는 리스트를 내포하고 있는 순차 데이터는 다차원 배열로 변환 가능하다.

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
    [5, 6, 7, 8]])
```

data2는 리스트를 담고 있는 리스트이므로 NumPy 배열인 arr2는 해당 데이터로부터 형태를 추론하여 2차원 형태로 생성된다. ndim과 shape 속성을 검사해서 이를 확인할 수 있다.

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

명시적으로 지정(자세한 내용은 잠시 후에 살펴보겠다)하지 않는 한 np.array는 생성될 때 적절한 자료형을 추론한다. 그렇게 추론된 자료형은 dtype 객체에 저장되는데 앞서 살펴본 예제에서 확인해보면 다음과 같다.

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

또한 np.array는 새로운 배열을 생성하기 위한 여러 함수를 가지고 있는데, 예를 들어 zeros와 ones는 주어진 길이나 모양에 각각 0과 1이 들어 있는 배열을 생성한다. empty 함수는 초기화되지 않은 배열을 생성한다. 이런 메서드를 사용해서 다차원 배열을 생성하려면 원하는 형태의 튜플을 넘기면 된다.

```
In [29]: np.zeros(10)
Out[29]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]],
       [[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]]])
```

CAUTION_ np.empty는 0으로 초기화된 배열을 반환하지 않는다. 앞서 살펴본 바와 같이 대부분의 경우 np.empty는 초기화되지 않은 '가비지' 값으로 채워진 배열을 반환한다.

arange는 파이썬의 range 함수의 배열 버전이다.

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

[표 4-1]은 표준 배열 생성 함수의 목록이다. NumPy는 산술 연산에 초점이 맞춰져 있기 때문에 만약 자료형을 명시하지 않으면 `float64`(부동소수점)가 될 것이다.

표 4-1 배열 생성 함수

함수	설명
`array`	입력 데이터(리스트, 튜플, 배열 또는 다른 순차형 데이터)를 ndarray로 변환하며 dtype을 명시하지 않은 경우 자료형을 추론하여 저장한다. 기본적으로 입력 데이터는 복사된다.
`asarray`	입력 데이터를 ndarray로 변환하지만 입력 데이터가 이미 ndarray일 경우 복사가 일어나지 않는다.
`arange`	내장 range 함수와 유사하지만 리스트대신 ndarray를 반환한다.
`ones, ones_like`	주어진 dtype과 모양을 가지는 배열을 생성하고 내용을 모두 1로 초기화한다. `ones_like`는 주어진 배열과 동일한 모양과 dtype을 가지는 배열을 새로 생성하여 내용을 모두 1로 초기화한다.
`zeros, zeros_like`	ones, ones_like와 동일하지만 내용을 0으로 채운다.
`empty, empty_like`	메모리를 할당하여 새로운 배열을 생성하지만 ones나 zeros처럼 값을 초기화하지 않는다.
`full, full_like`	인자로 받은 dtype과 배열의 모양을 가지는 배열을 생성하고 인자로 받은 값으로 배열을 채운다.
`eye, identity`	N x N 크기의 단위행렬을 생성한다(좌상단에서 우하단을 잇는 대각선은 1로 채워지고 나머지는 0으로 채워진다).

4.1.2 ndarray의 dtype

dtype은 ndarray가 메모리에 있는 특정 데이터를 해석하기 위해 필요한 정보(또는 **메타데이터**)를 담고 있는 특수한 객체다.

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')
```

```
In [36]: arr2.dtype
Out[36]: dtype('int32')
```

dtype이 있기에 NumPy가 강력하면서도 유연한 도구가 될 수 있었는데, 대부분의 경우 데이터는 디스크에서 데이터를 읽고 쓰기 편하도록 하위 레벨의 표현에 직접적으로 맞춰져 있어서 C나 포트란 같은 저수준 언어로 작성된 코드와 쉽게 연동이 가능하다. 산술 데이터의 dtype은 `float`나 `int` 같은 자료형의 이름과 하나의 원소가 차지하는 비트 수로 이루어진다. 파이썬의 `float` 객체에서 사용되는 표준 배정밀도 부동소수점double-precision floating point 값은 8바이트 혹은 64비트로 이루어지는데 이 자료형은 NumPy에서 `float64`로 표현된다. [표 4-2]는 NumPy가 지원하는 모든 자료형의 목록이다.

NOTE_ NumPy의 모든 dtype을 외울 필요는 없다. 주로 사용하게 될 자료형의 일반적인 종류(부동소수점, 복소수, 정수, 불리언, 문자열, 일반 파이썬 객체)만 신경 쓰면 된다. 주로 대용량 데이터가 메모리나 디스크에 저장되는 방식을 제어해야 할 필요가 있을 때 알아두면 좋다.

표 4-2 NumPy 자료형

자료형	자료형 코드	설명
`int8, uint8`	`i1, u1`	부호가 있는 8비트(1바이트) 정수형과 부호가 없는 8비트 정수형
`int16, uint16`	`i2, u2`	부호가 있는 16비트 정수형과 부호가 없는 16비트 정수형
`int32, uint32`	`i4, u4`	부호가 있는 32비트 정수형과 부호가 없는 32비트 정수형
`int64, uint64`	`i8, u8`	부호가 있는 64비트 정수형과 부호가 없는 64비트 정수형
`float16`	`f2`	반정밀도 부동소수점
`float32`	`f4` 또는 `f`	단정밀도 부동소수점. C언어의 `float`형과 호환
`float64`	`f8` 또는 `d`	배정밀도 부동소수점. C언어의 `double`형과 파이썬의 `float` 객체와 호환
`float128`	`f16` 또는 `g`	확장정밀도 부동소수점
`complex64, complex128, complex256`	`c8, c16, c32`	각각 2개의 32, 64, 128비트 부동소수점형을 가지는 복소수
`bool`	`?`	`True`와 `False` 값을 저장하는 불리언형
`object`	`O`	파이썬 객체형
`string_`	`S`	고정 길이 아스키 문자열형(각 문자는 1바이트). 길이가 10인 문자열 dtype은 `S10`이 된다.
`unicode_`	`U`	고정 길이 유니코드형(플랫폼에 따라 문자별 바이트 수가 다르다). `string_`형과 같은 형식을 쓴다(예: `U10`).

CHAPTER 5

pandas 시작하기

pandas는 앞으로 가장 자주 살펴볼 라이브러리다. 고수준의 자료구조와 파이썬에서 빠르고 쉽게 사용할 수 있는 데이터 분석 도구를 포함하고 있다. pandas는 다른 산술 계산 도구인 NumPy와 SciPy, 분석 라이브러리인 statsmodels와 scikit-learn, 시각화 도구인 matplotlib과 함께 사용하는 경우가 흔하다. pandas는 for 문을 사용하지 않고 데이터를 처리한다거나 배열 기반의 함수를 제공하는 등 NumPy의 배열 기반 계산 스타일을 많이 차용했다.

pandas가 NumPy의 스타일을 많이 차용했지만 가장 큰 차이점은 pandas는 표 형식의 데이터나 다양한 형태의 데이터를 다루는 데 초점을 맞춰 설계했다는 것이다. NumPy는 단일 산술 배열 데이터를 다루는 데 특화되어 있다.

2010년에 오픈소스로 공개한 이후 pandas는 다양한 현실의 요구 사항을 모두 수용할 수 있는 매우 큰 라이브러리가 되었다. 800명이 넘는 기여자가 자신이 매일 풀고 있는 데이터 문제를 더 편리하게 처리하기 위해 직접 pandas 프로젝트에 참여하고 있다.

이 책에서는 앞으로 pandas의 import 컨벤션을 다음과 같이 사용하겠다.

```
In [1]: import pandas as pd
```

앞으로 코드에서 pd.를 보면 pandas를 지칭하는 것으로 이해하자. Series와 DataFrame은 로컬 네임스페이스로 임포트하는 것이 훨씬 편하므로 그렇게 사용하도록 하겠다.

```
In [2]: from pandas import Series, DataFrame
```

5.1 pandas 자료구조 소개

pandas에 대해 알아보려면 Series와 DataFrame, 이 두 가지 자료구조에 익숙해질 필요가 있다. 이 두 가지 자료구조로 모든 문제를 해결할 순 없지만 대부분의 애플리케이션에서 사용하기 쉬우며 탄탄한 기반을 제공한다.

5.1.1 Series

Series는 일련의 객체를 담을 수 있는 1차원 배열 같은 자료구조다(어떤 NumPy 자료형이라도 담을 수 있다). 그리고 **색인**index이라고 하는 배열의 데이터와 연관된 이름을 가지고 있다. 가장 간단한 Series 객체는 배열 데이터로부터 생성할 수 있다.

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series 객체의 문자열 표현은 왼쪽에 색인을 보여주고 오른쪽에 해당 색인의 값을 보여준다. 위 예제에서는 데이터의 색인을 지정하지 않았으니 기본 색인인 정수 0에서 N - 1(N은 데이터의 길이)까지의 숫자가 표시된다. Series의 배열과 색인 객체는 각각 values와 index 속성을 통해 얻을 수 있다.

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index # range(4)와 같다.
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

각각의 데이터를 지칭하는 색인을 지정하여 Series 객체를 생성해야 할 때는 다음처럼 한다.

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d 4
b 7
a -5
c 3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

NumPy 배열과 비교하자면, 단일 값을 선택하거나 여러 값을 선택할 때 색인으로 라벨을 사용할 수 있다.

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

여기서 `['c', 'a', 'd']`는 (정수가 아니라 문자열이 포함되어 있지만) 색인의 배열로 해석된다.

불리언 배열을 사용해서 값을 걸러 내거나 산술 곱셈을 수행하거나 또는 수학 함수를 적용하는 등 NumPy 배열 연산을 수행해도 색인-값 연결이 유지된다.

```
In [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64
```

```
In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
c     6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Series를 이해하는 다른 방법은 고정 길이의 정렬된 사전형이라고 생각하는 것이다. Series는 색인값에 데이터값을 매핑하고 있으므로 파이썬의 사전형과 비슷하다. Series 객체는 파이썬의 사전형을 인자로 받아야 하는 많은 함수에서 사전형을 대체하여 사용할 수 있다.

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False
```

파이썬 사전형에 데이터를 저장해야 한다면 파이썬 사전 객체로부터 Series 객체를 생성할 수도 있다.

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

사전 객체만 가지고 Series 객체를 생성하면 생성된 Series 객체의 색인에는 사전의 키값이 순서대로 들어간다. 색인을 직접 지정하고 싶다면 원하는 순서대로 색인을 직접 넘겨줄 수도 있다.

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

이 예제를 보면 sdata에 있는 값 중 3개만 확인할 수 있는데, 'California'에 대한 값은 찾을 수 없기 때문이다. 이 값은 NaN[not a number]으로 표시되고 pandas에서는 누락된 값, 혹은 NA 값으로 취급된다. 'Utah'는 states에 포함되어 있지 않으므로 실행 결과에서 빠지게 된다.

나는 앞으로 '누락된' 또는 'NA'를 누락된 데이터를 지칭하는 데 사용하도록 하겠다. pandas의 isnull과 notnull 함수는 누락된 데이터를 찾을 때 사용된다.

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

이 메서드는 Series의 인스턴스 메서드로도 존재한다.

```
In [34]: obj4.isnull()
Out[34]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

누락된 데이터를 처리하는 방법은 7장에서 좀 더 자세히 살펴보기로 하자.

Series의 유용한 기능은 산술 연산에서 색인과 라벨로 자동 정렬하는 것이다.

```
In [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64

In [37]: obj3 + obj4
Out[37]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

데이터 정렬에 대한 내용은 나중에 좀 더 살펴보자. 데이터베이스를 사용해본 경험이 있다면 join 연산과 비슷하다고 여겨질 것이다.

Series 객체와 Series의 색인은 모두 name 속성이 있는데 이 속성은 pandas의 핵심 기능과 밀접한 관련이 있다.

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Series의 색인은 대입하여 변경할 수 있다.

```
In [41]: obj
Out[41]:
0    4
1    7
2   -5
3    3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [43]: obj
Out[43]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

5.1.2 DataFrame

DataFrame은 표 같은 스프레드시트 형식의 자료구조이고 여러 개의 컬럼이 있는데 각 컬럼

은 서로 다른 종류의 값(숫자, 문자열, 불리언 등)을 담을 수 있다. DataFrame은 로우와 컬럼에 대한 색인을 가지고 있는데, 색인의 모양이 같은 Series 객체를 담고 있는 파이썬 사전으로 생각하면 편하다. 내부적으로 데이터는 리스트나 사전 또는 1차원 배열을 담고 있는 다른 컬렉션이 아니라 하나 이상의 2차원 배열에 저장된다. 구체적인 DataFrame의 내부 구조는 이 책에서 다루는 내용에서 한참 벗어나므로 생략하겠다.

NOTE_ 물리적으로 DataFrame은 2차원이지만 계층적 색인을 이용해서 좀 더 고차원의 데이터를 표현할 수 있으며 이를 포함하여 pandas에서 데이터를 다루는 고급 기법은 8장에서 더 설명하겠다.

DataFrame 객체는 다양한 방법으로 생성할 수 있지만 가장 흔하게 사용되는 방법은 같은 길이의 리스트에 담긴 사전을 이용하거나 NumPy 배열을 이용하는 것이다.

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

만들어진 DataFrame의 색인은 Series와 같은 방식으로 자동으로 대입되며 컬럼은 정렬되어 저장된다.

```
In [45]: frame
Out[45]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
5  3.2  Nevada  2003
```

주피터 노트북을 사용한다면 DataFrame 객체는 브라우저에서 좀 더 보기 편하도록 HTML 표 형식으로 출력될 것이다.

큰 DataFrame을 다룰 때는 head 메서드를 이용해서 처음 5개의 로우만 출력할 수도 있다.

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

원하는 순서대로 columns를 지정하면 원하는 순서를 가진 DataFrame 객체가 생성된다.

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

Series와 마찬가지로 사전에 없는 값을 넘기면 결측치로 저장된다.

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])

In [49]: frame2
Out[49]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame의 컬럼은 Series처럼 사전 형식의 표기법으로 접근하거나 속성 형식으로 접근할 수 있다.

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]:
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

NOTE_ 편의를 위해 IPython에서는 `frame2.year`처럼 속성에 접근하듯 사용하거나 탭을 이용한 자동완성 기능을 제공한다.

`frame2[column]` 형태로 사용하는 것은 어떤 컬럼이든 가능하지만 `frame2.column` 형태로 사용하는 것은 파이썬에서 사용 가능한 변수 이름 형식일 때만 작동한다.

반환된 Series 객체가 DataFrame과 같은 색인을 가지면 알맞은 값으로 `name` 속성이 채워진다.

로우는 위치나 `loc` 속성을 이용해서 이름을 통해 접근할 수 있다.

```
In [53]: frame2.loc['three']
Out[53]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

CHAPTER 6

데이터 로딩과 저장, 파일 형식

이 책에서 다루는 대부분의 도구를 사용하는 첫 관문은 데이터에 접근하는 것이다. 다양한 형식의 데이터를 읽고 쓸 수 있는 많은 라이브러리가 있지만 이 책에서는 pandas에 초점을 맞춰 설명한다.

일반적으로 입출력은 몇 가지 작은 범주로 나뉘는데, 텍스트 파일을 이용하는 방법, 데이터베이스를 이용하는 방법, 웹 API를 이용해서 네트워크를 통해 불러오는 방법이 있다.

6.1 텍스트 파일에서 데이터를 읽고 쓰는 법

pandas에는 표 형식의 자료를 DataFrame 객체로 읽어오는 몇 가지 기능을 제공하고 있다. 아마 `read_csv`와 `read_table`을 주로 사용하게 될 테지만 [표 6-1]에 다른 함수도 정리해두었다.

표 6-1 pandas 파일 파싱 함수

함수	설명
`read_csv`	파일, URL 또는 파일과 유사한 객체로부터 구분된 데이터를 읽어온다. 데이터 구분자는 쉼표(,)를 기본으로 한다.
`read_table`	파일, URL 또는 파일과 유사한 객체로부터 구분된 데이터를 읽어온다. 데이터 구분자는 탭('\t')을 기본으로 한다.
`read_fwf`	고정폭 컬럼 형식에서 데이터를 읽어온다(구분자가 없는 데이터).
`read_clipboard`	클립보드에 있는 데이터를 읽어오는 `read_table` 함수. 웹페이지에서 표를 읽어올 때 유용하다.

함수	설명
read_excel	엑셀 파일(XLS, XLSX)에서 표 형식의 데이터를 읽어온다.
read_hdf	pandas에서 저장한 HDFS 파일에서 데이터를 읽어온다.
read_html	HTML 문서 내의 모든 테이블의 데이터를 읽어온다.
read_json	JSON 문자열에서 데이터를 읽어온다.
read_msgpack	메시지팩 바이너리 포맷으로 인코딩된 pandas 데이터를 읽어온다.
read_pickle	파이썬 피클 포맷으로 저장된 객체를 읽어온다.
read_sas	SAS 시스템의 사용자 정의 저장 포맷으로 저장된 데이터를 읽어온다.
read_sql	SQL 쿼리 결과를 pandas의 DataFrame 형식으로 읽어온다.
read_stata	Stata 파일에서 데이터를 읽어온다.
read_feather	Feather 바이너리 파일 포맷으로부터 데이터를 읽어온다.

위 함수들은 텍스트 데이터를 DataFrame으로 읽어오기 위한 함수인데, 아래와 같은 몇 가지 옵션을 취한다.

- **색인**
  반환하는 DataFrame에서 하나 이상의 컬럼을 색인으로 지정할 수 있다. 파일이나 사용자로부터 컬럼 이름을 받거나 아무것도 받지 않을 수 있다.
- **자료형 추론과 데이터 변환**
  사용자 정의 값 변환과 비어 있는 값을 위한 사용자 리스트를 포함한다.
- **날짜 분석**
  여러 컬럼에 걸쳐 있는 날짜와 시간 정보를 하나의 컬럼에 조합해서 결과에 반영한다.
- **반복**
  여러 개의 파일에 걸쳐 있는 자료를 반복적으로 읽어올 수 있다.
- **정제되지 않은 데이터 처리**
  로우나 꼬리말, 주석 건너뛰기 또는 천 단위마다 쉼표로 구분된 숫자 같은 사소한 것들의 처리를 해준다.

실제 데이터는 엉망진창인 상태이므로 데이터를 불러오는 함수(특히 read_csv 같은)는 개발이 계속됨에 따라 복잡도가 급속도로 증가한다. 넘쳐나는 함수 인자들(read_csv의 함수 인자는 현재 50개가 넘는다)을 보고 있으면 두통이 오는 것만 같다. pandas 온라인 문서에는 각 인자들이 어떻게 동작하는지 다양한 예제와 함께 설명하고 있으므로 특정 파일을 읽는 데 어려움을 느낀다면 필요한 인자와 그에 맞는 예제가 도움이 될 것이다.

이런 함수들 중 일부, 예를 들어 pandas.read_csv 같은 함수들은 데이터 형식에 자료형이 포

함되어 있지 않은 관계로 **타입 추론**을 수행한다. HDF5나 Feather, msgpack의 경우에는 데이터 형식에 자료형이 포함되어 있다.

날짜나 다른 몇 가지 사용자 정의 자료형을 처리하려면 수고가 조금 필요하다. 쉼표로 구분된 작은 CSV 파일을 한 번 살펴보자.

```
In [8]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

NOTE_ 여기서는 유닉스의 `cat` 명령어를 사용해서 파일의 내용을 확인했다. 윈도우 사용자라면 `cat` 대신 `type` 명령어를 사용해서 내용을 확인할 수 있다.

이 파일은 쉼표로 구분되어 있기 때문에 `read_csv`를 사용해서 DataFrame으로 읽어올 수 있다.

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

`read_table`에 구분자를 쉼표로 지정해서 읽어올 수도 있다.

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

모든 파일에 컬럼 이름이 있는 건 아니다. 다음 파일을 보자.

```
In [12]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

이 파일을 읽어오는 몇 가지 옵션이 있는데, pandas가 자동으로 컬럼 이름을 생성하도록 하거나 우리가 직접 컬럼 이름을 지정한다.

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

`message` 컬럼을 색인으로 하는 DataFrame을 반환하려면 `index_col` 인자에 4번째 컬럼 또는 `'message'` 이름을 가진 컬럼을 지정해서 색인으로 만들 수 있다.

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

계층적 색인을 지정하고 싶다면 컬럼 번호나 이름의 리스트를 넘기면 된다.

```
In [17]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
```

```
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
   ....:                      index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
           value1  value2
key1 key2
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

가끔 고정된 구분자 없이 공백이나 다른 패턴으로 필드를 구분해놓은 경우가 있다. 다음과 같은 파일이 있다고 하자.

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

직접 파일을 고쳐도 되지만, 이 파일은 필드가 여러 개의 공백 문자로 구분되어 있으므로 이를 표현할 수 있는 정규 표현식 \s+를 사용해서 처리할 수도 있다.

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')
```

```
In [22]: result
Out[22]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

이 경우 첫 번째 로우는 다른 로우보다 컬럼이 하나 적기 때문에 read_table은 첫 번째 컬럼이 DataFrame의 색인이 되어야 한다고 추론한다.

파서 함수는 파일 형식에서 발생할 수 있는 매우 다양한 예외(표 6-2)를 잘 처리할 수 있도록 많은 추가 인자를 가지고 있는데, 예를 들면 skiprows를 이용해서 첫 번째, 세 번째, 네 번째 로우를 건너뛸 수 있다.

```
In [23]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo

In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

누락된 값을 잘 처리하는 것은 파일을 읽는 과정에서 자주 발생하는 일이고 중요한 문제다. 보통 텍스트 파일에서 누락된 값은 표기되지 않거나(비어 있는 문자열) 구분하기 쉬운 특수한 문자로 표기된다. 기본적으로 pandas는 NA나 NULL처럼 흔히 통용되는 문자를 비어 있는 값으로 사용한다.

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
```

```
two,5,6,,8,world
three,9,10,11,12,foo

In [26]: result = pd.read_csv('examples/ex5.csv')

In [27]: result
Out[27]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [28]: pd.isnull(result)
Out[28]:
  something      a      b      c      d message
0     False  False  False  False  False    True
1     False  False  False   True  False   False
2     False  False  False  False  False   False
```

na_values 옵션은 리스트나 문자열 집합을 받아서 누락된 값을 처리한다.

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [30]: result
Out[30]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

컬럼마다 다른 NA 문자를 사전값으로 넘겨서 처리할 수도 있다.

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

[표 6-2]에 pandas_read_csv와 pandas_read_table에서 자주 사용하는 인자들을 모아두었다.

표 6-2 read_csv와 read_table 함수 인자

인자	설명
path	파일시스템에서의 위치, URL, 파일 객체를 나타내는 문자열
sep 또는 delimiter	필드를 구분하기 위해 사용할 연속된 문자나 정규 표현식
header	컬럼 이름으로 사용할 로우 번호. 기본값은 0(첫 번째 로우)이며 헤더가 없을 경우에는 None으로 지정할 수 있다.
index_col	색인으로 사용할 컬럼 번호나 이름. 계층적 색인을 지정할 경우 리스트를 넘길 수 있다.
names	컬럼 이름으로 사용할 리스트. header=None과 함께 사용한다.
skiprows	파일의 시작부터 무시할 행 수 또는 무시할 로우 번호가 담긴 리스트
na_values	NA 값으로 처리할 값들의 목록
comment	주석으로 분류되어 파싱하지 않을 문자 혹은 문자열
parse_dates	날짜를 datetime으로 변환할지 여부. 기본값은 False이며, True일 경우 모든 컬럼에 적용된다. 컬럼의 번호나 이름을 포함한 리스트를 넘겨서 변환할 컬럼을 지정할 수 있는데, [1, 2, 3]을 넘기면 각각의 컬럼을 datetime으로 변환하며, [[1,3]]을 넘기면 1, 3번 컬럼을 조합해서 하나의 datetime으로 변환한다.
keep_date_col	여러 컬럼을 datetime으로 변환했을 경우 원래 컬럼을 남겨둘지 여부. 기본값은 True
converters	변환 시 컬럼에 적용할 함수를 지정한다. 예를 들어 {'foo': f}는 'foo' 컬럼에 f 함수를 적용시킨다. 전달하는 사전의 키값은 컬럼 이름이나 번호가 될 수 있다.
dayfirst	모호한 날짜 형식일 경우 국제 형식으로 간주한다(7/6/2012는 2012년 6월 7일로 간주한다). 기본값은 False
date_parser	날짜 변환 시 사용할 함수
nrows	파일의 첫 일부만 읽어올 때 처음 몇 줄을 읽을 것인지 지정
iterator	파일을 조금씩 읽을 때 사용하도록 TextParser 객체를 반환하도록 한다. 기본값은 False
chunksize	TextParser 객체에서 사용할 한 번에 읽을 파일의 크기
skip_footer	파일의 끝에서 무시할 라인 수
verbose	파싱 결과에 대한 정보를 출력한다. 숫자가 아닌 값이 들어 있는 컬럼에 누락된 값이 있다면 줄 번호를 출력해준다. 기본값은 False
encoding	유니코드 인코딩 종류를 지정한다. UTF-8로 인코딩된 텍스트일 경우 'utf-8'로 지정한다.
squeeze	만일 컬럼이 하나뿐이라면 Series 객체를 반환한다. 기본값은 False
thousands	숫자를 천 단위로 끊을 때 사용할 ',' 나 '.' 같은 구분자

6.1.1 텍스트 파일 조금씩 읽어오기

매우 큰 파일을 처리할 때 인자를 제대로 주었는지 알아보기 위해 파일의 일부분만 읽어보거나 여러 파일 중에서 몇 개의 파일만 읽어서 확인해보고 싶을 것이다.

큰 파일을 다루기 전에 pandas의 출력 설정을 조금 손보자.

```
In [33]: pd.options.display.max_rows = 10
```

이제 최대 10개의 데이터만 출력한다.

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

파일 전체를 읽는 대신 처음 몇 줄만 읽어보고 싶다면 nrows 옵션을 주면 된다.

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

파일을 여러 조각으로 나누어서 읽고 싶다면 chunksize 옵션으로 로우의 개수를 주면 된다.

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)
```

```
In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

read_csv에서 반환된 TextParser 객체를 이용해서 chunksize에 따라 분리된 파일들을 순회할 수 있다. 예를 들어 ex6.csv 파일을 순회하면서 'key' 로우에 있는 값을 세어보려면 다음처럼 하면 된다.

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

그러면 다음과 같은 결과를 얻을 수 있다.

```
In [40]: tot[:10]
Out[40]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

TextParser는 임의 크기의 조각을 읽을 수 있는 get_chunk 메서드도 포함하고 있다.

6.1.2 데이터를 텍스트 형식으로 기록하기

읽어오기와 마찬가지로 데이터를 구분자로 구분한 형식으로 내보내는 것도 가능하다. 위에서 읽었던 CSV 파일 중 하나를 다시 보자.

CHAPTER 7

데이터 정제 및 준비

데이터 분석과 모델링 작업에서는 데이터를 불러오고, 정제하고, 변형하고, 재정렬하는 데이터 준비 과정에 많은 시간을 들이게 된다. 이런 작업들은 분석 시간의 80%를 잡아먹기도 한다. 가끔은 파일이나 데이터베이스에 저장된 데이터가 애플리케이션에서 사용하기 쉽지 않은 방식으로 저장되어 있기도 하다. 대부분의 사람은 파일이나 데이터베이스에 저장된 데이터를 다른 형태로 바꾸기 위해 파이썬이나 Perl, R, Java 혹은 awk나 sed 같은 유닉스의 텍스트 처리 유틸리티를 사용하기도 하는데, 파이썬 표준 라이브러리를 pandas와 함께 사용하면 큰 수고 없이 데이터를 원하는 형태로 가공할 수 있다. pandas는 이런 작업을 위한 유연하고 빠른 고수준의 알고리즘과 처리 기능을 제공한다.

혹시 이 책이나 pandas 라이브러리에서 찾을 수 없는 새로운 형태의 데이터 처리 방식을 발견하게 된다며 파이썬 메일링 리스트나 pandas 깃허브에 올려놓기 바란다. 실제로 pandas는 대부분의 설계와 구현에 실제 애플리케이션 개발 과정 중에 발생한 요구 사항을 고려했다.

이 장에서는 결측치, 중복 데이터, 문자열 처리 그리고 다른 분석적 데이터 변환에 대한 도구들을 다룬다. 다음 장에서는 데이터를 합치고 재배열하는 다양한 방법을 알아보겠다.

7.1 누락된 데이터 처리하기

누락된 데이터를 처리하는 일은 데이터 분석 애플리케이션에서 흔히 발생하는 일이다. pandas

의 설계 목표 중 하나는 누락 데이터를 가능한 한 쉽게 처리할 수 있도록 하는 것이다. 예를 들어 pandas 객체의 모든 기술 통계는 누락된 데이터를 배제하고 처리한다.

pandas 객체에서 누락된 값을 표현하는 방식은 완벽하다고 할 수 없다. 산술 데이터에 한해 pandas는 누락된 데이터를 실숫값인 `NaN`으로 취급한다. 이는 누락된 값을 쉽게 찾을 수 있도록 하는 파수병 역할을 한다.

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

pandas에서는 R 프로그래밍 언어에서 결측치를 NA[Not Available]로 취급하는 개념을 차용했다. 분석 애플리케이션에서 NA 데이터는 데이터가 존재하지 않거나, 존재하더라도 데이터를 수집하는 과정 등에서 검출되지 않았음을 의미한다. 분석을 위해 데이터를 정제하는 과정에서 결측치 자체를 데이터 수집 과정에서의 실수나 결측치로 인한 잠재적인 편향을 찾아내는 수단으로 인식하는 것은 중요하다.

파이썬의 내장 `None` 값 또한 NA 값으로 취급된다.

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0     True
```

```
1    False
2     True
3    False
dtype: bool
```

pandas 프로젝트에서는 결측치를 처리하는 방법을 개선하는 작업이 진행 중이지만 pandas.isnull 같은 사용자 API 함수에서는 성가신 부분을 추상화로 제거했다. [표 7-1]에 결측치 처리와 관련된 함수를 정리해두었다.

표 7-1 NA 처리 메서드

인자	설명
dropna	누락된 데이터가 있는 축(로우, 컬럼)을 제외시킨다. 어느 정도의 누락 데이터까지 용인할 것인지 지정할 수 있다.
fillna	누락된 데이터를 대신할 값을 채우거나 'ffill'이나 'bfill' 같은 보간 메서드를 적용한다.
isnull	누락되거나 NA인 값을 알려주는 불리언값이 저장된 같은 형의 객체를 반환한다.
notnull	isnull과 반대되는 메서드

7.1.1 누락된 데이터 골라내기

누락된 데이터를 골라내는 몇 가지 방법이 있는데, pandas.isnull이나 불리언 색인을 사용해 직접 손으로 제거하는 것도 한 가지 방법이지만, dropna를 매우 유용하게 사용할 수 있다. Series에 dropna 메서드를 적용하면 널이 아닌[non-null] 데이터와 색인값만 들어 있는 Series를 반환한다.

```
In [15]: from numpy import nan as NA


In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

위 코드는 다음과 동일하다.

```
In [18]: data[data.notnull()]
Out[18]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

DataFrame 객체의 경우에는 조금 복잡한데, 모두 NA 값인 로우나 컬럼을 제외시키거나 NA 값을 하나라도 포함하고 있는 로우나 컬럼을 제외시킬 수 있다. dropna는 기본적으로 NA 값을 하나라도 포함하고 있는 로우를 제외시킨다.

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

In [22]: cleaned
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

how='all' 옵션을 넘기면 모두 NA 값인 로우만 제외시킨다.

```
In [23]: data.dropna(how='all')
Out[23]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

컬럼을 제외시키는 방법도 동일하게 동작한다. 옵션으로 axis=1을 넘겨주면 된다.

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
     0    1    2    4
0  1.0  6.5  3.0  NaN
1  1.0  NaN  NaN  NaN
2  NaN  NaN  NaN  NaN
3  NaN  6.5  3.0  NaN

In [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

DataFrame의 로우를 제외시키는 방법은 시계열 데이터에 주로 사용되는 경향이 있다. 몇 개 이상의 값이 들어 있는 로우만 살펴보고 싶다면 thresh 인자에 원하는 값을 넘기면 된다.

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA

In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
```

```
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

7.1.2 결측치 채우기

누락된 값을 제외시키지 않고(잠재적으로 다른 데이터도 함께 버려질 가능성이 있다) 데이터 상의 '구멍'을 어떻게든 메우고 싶은 경우가 있다. 이 경우 fillna 메서드를 활용하면 되는데, fillna 메서드에 채워 넣고 싶은 값을 넘겨주면 된다.

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

fillna에 사전값을 넘겨서 각 컬럼마다 다른 값을 채울 수도 있다.

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
```

```
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

fillna는 새로운 객체를 반환하지만 다음처럼 기존 객체를 변경할 수도 있다.

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

재색인에서 사용 가능한 보간 메서드는 fillna 메서드에서도 사용 가능하다.

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN

In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
```

```
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232

In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

조금만 창의적으로 생각하면 fillna를 이용해서 매우 다양한 일을 할 수 있는데 예를 들어 Series의 평균값이나 중간값을 전달할 수도 있다.

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

fillna에 대한 설명은 [표 7-2]를 참조하자.

표 7-2 fillna 함수 인자

인자	설명
value	비어 있는 값을 채울 스칼라값이나 사전 형식의 객체
method	보간 방식. 기본적으로 'ffill'을 사용한다.
axis	값을 채워 넣을 축. 기본값은 axis=0이다.
inplace	복사본을 생성하지 않고 호출한 객체를 변경한다. 기본값은 False다.
limit	값을 앞 혹은 뒤에서부터 몇 개까지 채울지 지정한다.

7.2 데이터 변형

지금까지 데이터를 재배치하는 방법을 알아봤다. 필터링, 정제 및 다른 변형 역시 중요한 연산이다.

7.2.1 중복 제거하기

여러 가지 이유로 DataFrame에서 중복된 로우를 발견할 수 있다. 예제를 보자.

```
In [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
   ....:                      'k2': [1, 1, 2, 3, 3, 4, 4]})

In [46]: data
Out[46]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

DataFrame의 duplicated 메서드는 각 로우가 중복인지 아닌지 알려주는 불리언 Series를 반환한다.

```
In [47]: data.duplicated()
Out[47]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

drop_duplicates는 duplicated 배열이 False인 DataFrame을 반환한다.

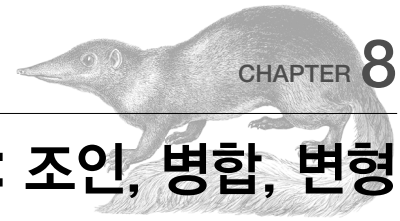
```
In [48]: data.drop_duplicates()
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

이 두 메서드는 기본적으로 모든 컬럼에 적용되며 중복을 찾아내기 위한 부분합을 따로 지정해 줄 수도 있다. 새로운 컬럼을 하나 추가하고 'k1' 컬럼에 기반해서 중복을 걸러내려면 다음과 같이 한다.

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
    k1  k2  v1
0  one   1   0
1  two   1   1
```

duplicated와 drop_duplicates는 기본적으로 처음 발견된 값을 유지한다. keep='last' 옵션을 넘기면 마지막으로 발견된 값을 반환한다.

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6
```



CHAPTER 8

데이터 준비하기: 조인, 병합, 변형

대부분의 경우 데이터는 여러 파일이나 데이터베이스 혹은 분석하지 쉽지 않은 형태로 기록되어 있다. 이 장에서는 데이터를 합치고, 재배열할 수 있는 도구들을 살펴보자.

먼저 데이터를 병합하거나 변환하는 과정에서 사용되는 pandas의 계층적 색인의 개념을 알아보고 이를 활용하여 데이터를 다듬는 과정을 심도 있게 살펴볼 것이다. 14장에서도 이런 도구를 사용하는 다양한 예시를 볼 수 있다.

8.1 계층적 색인

계층적 색인은 pandas의 중요한 기능인데 축에 대해 다중(둘 이상) 색인 **단계**를 지정할 수 있도록 해준다. 약간 추상적으로 말하면, 높은 차원의 데이터를 낮은 차원의 형식으로 다룰 수 있게 해주는 기능이다. 간단한 예제를 하나 살펴보자. 우선 리스트의 리스트(또는 배열)를 색인으로 하는 Series를 하나 생성하자.

```
In [9]: data = pd.Series(np.random.randn(9),
   ...:                  index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [10]: data
Out[10]:
```

```
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

지금 생성한 객체가 MultiIndex를 색인으로 하는 Series인데, 색인의 계층을 보여주고 있다. 바로 위 단계의 색인을 이용해서 하위 계층을 직접 접근할 수 있다.

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

계층적으로 색인된 객체는 데이터의 부분집합을 **부분적 색인으로 접근**partial indexing하는 것이 가능하다.

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64

In [13]: data['b':'c']
Out[13]:
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
dtype: float64

In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
   3    1.965781
```

```
d  2    0.281746
   3    0.769023
dtype: float64
```

하위 계층의 객체를 선택하는 것도 가능하다.

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

계층적인 색인은 데이터를 재형성하고 피벗테이블 생성 같은 그룹 기반의 작업을 할 때 중요하게 사용된다. 예를 들어 위에서 만든 DataFrame 객체에 unstack 메서드를 사용해서 데이터를 새롭게 배열할 수도 있다.

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

unstack의 반대 작업은 stack 메서드로 수행한다.

```
In [17]: data.unstack().stack()
Out[17]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

stack과 unstack 메서드는 이 장 후반부에서 더 자세히 알아보기로 하자.

DataFrame에서는 두 축 모두 계층적 색인을 가질 수 있다.

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
   ....:                      index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
   ....:                      columns=[['Ohio', 'Ohio', 'Colorado'],
   ....:                               ['Green', 'Red', 'Green']])

In [19]: frame
Out[19]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

계층적 색인의 각 단계는 이름(문자열이나 어떤 파이썬 객체라도 가능하다)을 가질 수 있고, 만약 이름을 가지고 있다면 콘솔 출력 시 함께 나타난다.

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

CAUTION_ 색인 이름인 'state'와 'color'를 로우 라벨과 혼동하지 말자.

컬럼의 부분집합을 부분적인 색인으로 접근하는 것도 컬럼에 대한 부분적 색인과 비슷하게 사용하면 된다.

```
In [23]: frame['Ohio']
Out[23]:
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

MultiIndex는 따로 생성한 다음에 재사용 가능하다. 위에서 살펴본 DataFrame의 컬럼 계층 이름은 다음처럼 생성할 수 있다.

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                       names=['state', 'color'])
```

8.1.1 계층의 순서를 바꾸고 정렬하기

계층적 색인에서 계층의 순서를 바꾸거나 지정된 계층에 따라 데이터를 정렬해야 하는 경우가 있을 수 있다. swaplevel은 넘겨받은 두 개의 계층 번호나 이름이 뒤바뀐 새로운 객체를 반환한다(하지만 데이터는 변경되지 않는다).

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
2    a        3   4        5
1    b        6   7        8
2    b        9  10       11
```

반면 sort_index 메서드는 단일 계층에 속한 데이터를 정렬한다. swaplevel을 이용해서 계층을 바꿀 때 sort_index를 사용해서 결과가 사전적으로 정렬되도록 만드는 것도 드물지 않은 일이다.

```
In [25]: frame.sort_index(level=1)
Out[25]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
b    1        6   7        8
a    2        3   4        5
b    2        9  10       11

In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
     b        6   7        8
2    a        3   4        5
     b        9  10       11
```

NOTE_ 객체가 계층적 색인으로 상위 계층부터 사전식으로 정렬되어 있다면(`sort_index(level=0)`이나 `sort_index( )`의 결과처럼) 데이터를 선택하는 성능이 훨씬 좋아진다.

8.1.2 계층별 요약 통계

DataFrame과 Series의 많은 기술 통계와 요약 통계는 `level` 옵션을 가지고 있는데, 어떤 한 축에 대해 합을 구하고 싶은 단계를 지정할 수 있는 옵션이다. 앞에서 살펴본 DataFrame에서 로우나 컬럼을 아래처럼 계층별로 합할 수 있다.

```
In [27]: frame.sum(level='key2')
Out[27]:
state  Ohio     Colorado
color Green Red    Green
key2
1         6   8       10
2        12  14       16
```

```
In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

이는 내부적으로는 pandas의 groupby 기능을 이용해서 구현되었는데, 자세한 내용은 앞으로 더 살펴보겠다.

8.1.3 DataFrame의 컬럼 사용하기

DataFrame에서 로우를 선택하기 위한 색인으로 하나 이상의 컬럼을 사용하는 것은 드물지 않은 일이다. 아니면 로우의 색인을 DataFrame의 컬럼으로 옮기고 싶을 것이다. 다음은 예제 DataFrame이다.

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
   ....:                       'c': ['one', 'one', 'one', 'two', 'two',
   ....:                             'two', 'two'],
   ....:                       'd': [0, 1, 2, 0, 1, 2, 3]})

In [30]: frame
Out[30]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

DataFrame의 set_index 함수는 하나 이상의 컬럼을 색인으로 하는 새로운 DataFrame을 생성한다.

```
In [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
       a  b
c   d
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

다음처럼 컬럼을 명시적으로 남겨두지 않으면 DataFrame에서 삭제된다.

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
       a  b    c  d
c   d
one 0  0  7  one  0
    1  1  6  one  1
    2  2  5  one  2
two 0  3  4  two  0
    1  4  3  two  1
    2  5  2  two  2
    3  6  1  two  3
```

반면 reset_index 함수는 set_index와 반대되는 개념인데 계층적 색인 단계가 컬럼으로 이동한다.

```
In [34]: frame2.reset_index()
Out[34]:
     c  d  a  b
0  one  0  0  7
1  one  1  1  6
2  one  2  2  5
3  two  0  3  4
4  two  1  4  3
5  two  2  5  2
6  two  3  6  1
```

8.2 데이터 합치기

pandas 객체에 저장된 데이터는 여러 가지 방법으로 합칠 수 있다.

- `pandas.merge`는 하나 이상의 키를 기준으로 DataFrame의 로우를 합친다. SQL이나 다른 관계형 데이터베이스의 join 연산과 유사하다.
- `pandas.concat`은 하나의 축을 따라 객체를 이어붙인다.
- `combile_first` 인스턴스 메서드는 두 객체를 포개서 한 객체에서 누락된 데이터를 다른 객체에 있는 값으로 채울 수 있도록 한다.

각각의 데이터를 합치는 방법에 대해 다양한 예제와 함께 살펴보게 될 것이다. 이 기법은 책 전체에서 계속 활용된다.

8.2.1 데이터베이스 스타일로 DataFrame 합치기

병합(머지merge)이나 **조인**join 연산은 관계형 데이터베이스의 핵심적인 연산인데, 하나 이상의 **키**를 사용해서 데이터 집합의 로우를 합친다. pandas의 `merge` 함수를 이용해서 이런 알고리즘을 데이터에 적용할 수 있다.

예제를 살펴보자.

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [37]: df1
Out[37]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b
```

```
In [38]: df2
Out[38]:
   data2 key
0      0   a
1      1   b
2      2   d
```

위 예제는 **다대일**의 경우다. df1의 데이터는 key 컬럼에 여러 개의 a, b 로우를 가지고 있고 df2의 데이터는 key 컬럼에 유일한 로우를 가지고 있다. 이 객체에 대해 merge 함수를 호출하면 다음과 같은 결과를 얻는다.

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

위에서 나는 어떤 컬럼을 병합할 것인지 명시하지 않았는데, merge 함수는 중복된 컬럼 이름을 키로 사용한다(위 예에서는 key 컬럼). 하지만 명시적으로 지정해주는 습관을 들이는 게 좋다.

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

만약 두 객체에 중복된 컬럼 이름이 하나도 없다면 따로 지정해주면 된다.

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})
```

```
In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   data1 lkey  data2 rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

결과를 잘 살펴보면 'c'와 'd'에 해당하는 값이 빠진 것을 알 수 있다. merge 함수는 기본적으로 내부 조인inner join을 수행하여 교집합인 결과를 반환한다. how 인자로 'left', 'right', 'outer'를 넘겨서 각각 왼쪽 조인, 오른쪽 조인, 외부 조인을 수행할 수도 있다. 외부 조인은 합집합인 결과를 반환하고 왼쪽 조인과 오른쪽 조인은 각각 왼쪽 또는 오른쪽의 모든 로우를 포함하는 결과를 반환한다.

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1 key  data2
0    0.0   b    1.0
1    1.0   b    1.0
2    6.0   b    1.0
3    2.0   a    0.0
4    4.0   a    0.0
5    5.0   a    0.0
6    3.0   c    NaN
7    NaN   d    2.0
```

[표 8-1]에 how 옵션에 따라 조인 연산이 어떻게 동작하는지 요약해두었다.

표 8-1 how 옵션에 따른 다양한 조인 연산

옵션	동작
'inner'	양쪽 테이블 모두에 존재하는 키 조합을 사용한다.
'left'	왼쪽 테이블에 존재하는 모든 키 조합을 사용한다.
'right'	오른쪽 테이블에 존재하는 모든 키 조합을 사용한다.
'output'	양쪽 테이블에 존재하는 모든 키 조합을 사용한다.

다대다 병합은 잘 정의되어 있긴 하지만 직관적이지는 않다.

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                     'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                     'data2': range(5)})

In [47]: df1
Out[47]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   b

In [48]: df2
Out[48]:
   data2 key
0      0   a
1      1   b
2      2   a
3      3   b
4      4   d

In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
    data1 key  data2
0       0   b    1.0
1       0   b    3.0
2       1   b    1.0
3       1   b    3.0
4       2   a    0.0
5       2   a    2.0
6       3   c    NaN
7       4   a    0.0
8       4   a    2.0
9       5   b    1.0
10      5   b    3.0
```

CHAPTER 9

그래프와 시각화

정보 시각화는 데이터 분석에서 무척 중요한 일 중 하나다. 시각화는 특잇값을 찾아내거나, 데이터 변형이 필요한지 알아보거나, 모델에 대한 아이디어를 찾기 위한 과정의 일부이기도 하다. 혹자에게는 웹상에서 구현되는 시각화가 최종 목표일 수도 있다. 파이썬은 다양한 시각화 도구를 구비하고 있지만, 이 책에서는 matplotlib과 matplotlib 기반의 도구들을 우선적으로 살펴보겠다.

matplotlib은 주로 2D 그래프를 위한 데스크톱 패키지로, 출판물 수준의 그래프를 만들어내도록 설계되었다. matplotlib 프로젝트는 파이썬에서 매트랩과 유사한 인터페이스를 지원하기 위해 2002년 존 헌터John Hunter가 시작했다. 그 후 IPython과 matplotlib 커뮤니티의 협력을 통해 IPython 셸(지금은 주피터 노트북)에서 대화형 시각화를 구현해냈다. matplotlib은 모든 운영체제의 다양한 GUI 백엔드를 지원하고 있으며 PDF, SVG, JPG, PNG, BMP, GIF 등 일반적으로 널리 사용되는 벡터 포맷과 래스터 포맷으로 그래프를 저장할 수 있다. 이 책에 수록된 대부분의 그래프는 matplotlib을 이용해서 만들었다.

시간이 흐름에 따라 내부적으로 matplotlib을 사용하는 새로운 데이터 시각화 도구들이 생겨났는데 그중 하나가 이 장 후반부에서 살펴볼 seaborn 라이브러리다.

이 장에 포함된 코드 예제를 실행시키는 가장 손쉬운 방법은 주피터 노트북의 대화형 시각화 기능을 사용하는 것이다. 이 기능을 활성화하려면 주피터 노트북을 실행시킨 후 다음 명령을 입력한다.

```
%matplotlib notebook
```

9.1 matplotlib API 간략하게 살펴보기

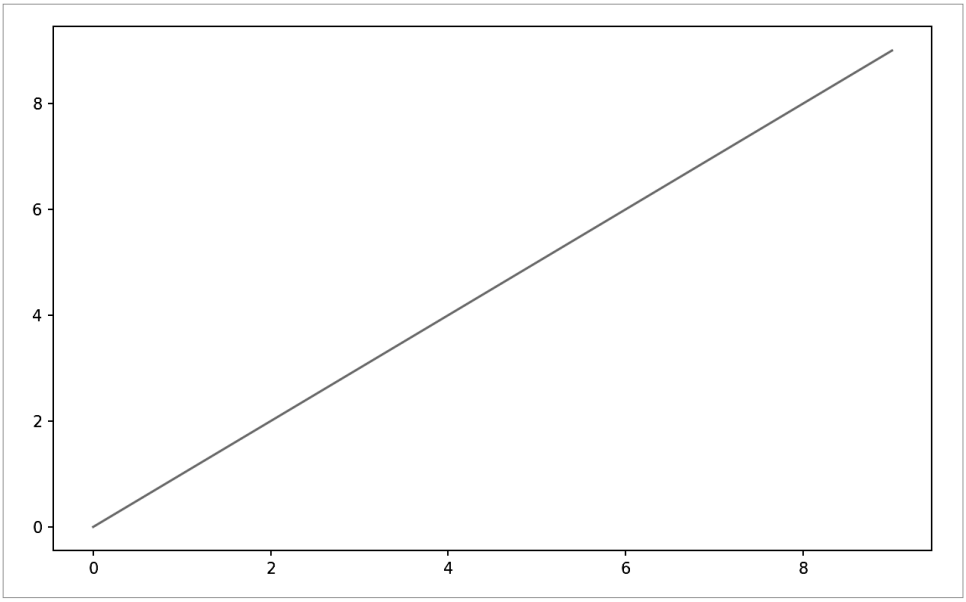
이 책에서는 matplotlib을 아래와 같은 네이밍 컨벤션으로 임포트하겠다.

```
In [11]: import matplotlib.pyplot as plt
```

주피터 노트북 환경에서 `%matplotlib notebook`을 실행한 다음(IPython인 경우 그냥 `%matplotlib`) 간단한 그래프를 그려보자. 모든 것이 제대로 설정되었다면 [그림 9-1]과 같은 선그래프가 그려진다.

```
In [12]: import numpy as np

In [13]: data = np.arange(10)

In [14]: data
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: plt.plot(data)
```

그림 9-1 간단한 선그래프



나중에 seaborn 라이브러리나 pandas로 그래프를 그리는 방법과 그래프가 만들어지는 세부 사항에 관한 재미없는 내용을 다루게 되는데, 함수에서 제공하는 옵션만 사용하는 데 그치지 않고 그 이상의 최적화를 하고 싶다면 matplotlib API도 어느 정도 알고 있어야 한다.

> **NOTE_** 이 책에서 matplotlib에서 제공하는 기능에 대한 폭넓고 심도 있는 내용이나 포괄적인 내용을 다루기에는 무리가 있다. 예제와 함께 matplotlib의 간단한 사용법만 알려줄 것이다. matplotlib 갤러리와 문서에 있는 내용을 참고하면 고급 기능을 사용하고 그래프를 작성하는 데 고수가 될 수 있는 길잡이 역할을 해줄 것이다.

9.1.1 figure와 서브플롯

matplotlib에서 그래프는 `Figure` 객체 내에 존재한다. 그래프를 위한 새로운 figure(피겨)는 `plt.figure`를 사용해서 생성할 수 있다.

```
In [16]: fig = plt.figure()
```

IPython에서 실행했다면 빈 윈도우가 나타날 것이다. 반면 주피터에서는 몇 가지 명령을 더 입력하기 전에는 아무것도 나타나지 않을 것이다. `plt.figure`에는 다양한 옵션이 있는데 그중 `figsize`는 파일에 저장할 경우를 위해 만들려는 figure의 크기와 비율을 지정할 수 있다.

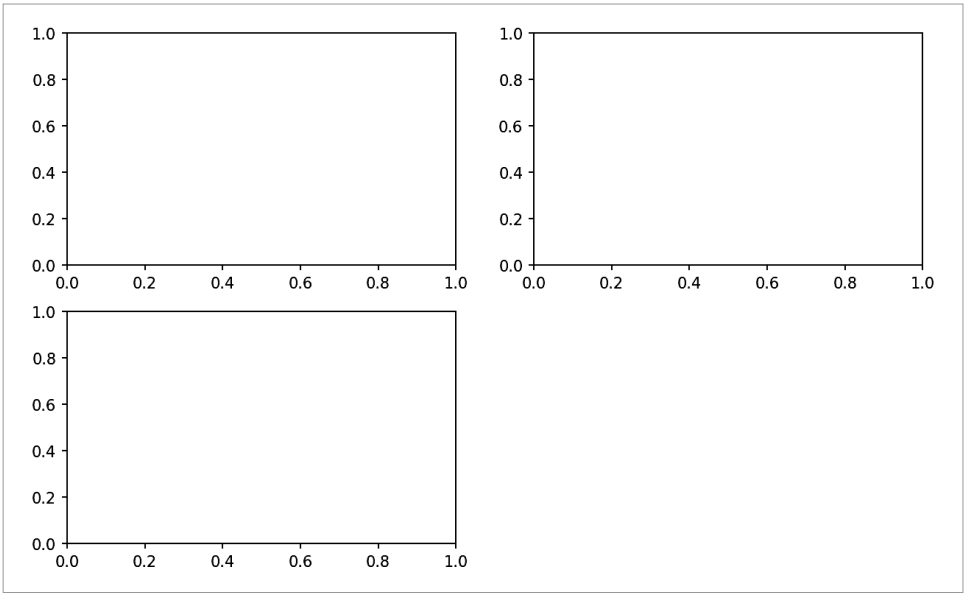
빈 figure로는 그래프를 그릴 수 없다. `add_subplot`을 사용해서 최소 하나 이상의 `subplots`를 생성해야 한다.

```
In [17]: ax1 = fig.add_subplot(2, 2, 1)
```

위 코드는 figure가 2×2 크기이고 4개의 서브플롯 중에서 첫 번째를 선택하겠다는 의미다(서브플롯은 1부터 숫자가 매겨진다). 다음처럼 2개의 서브플롯을 더 추가하면 [그림 9-2]와 같은 모양이 된다.

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)

In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

그림 9-2 3개의 서브플롯을 가지는 빈 matplotlib figure



TIP 주피터 노트북을 사용할 때는 실행되는 셀마다 그래프가 리셋된다. 따라서 복잡한 그래프를 그릴 때는 단일 노트북 셀에 그래프를 그리는 코드를 전부 입력해야 한다.

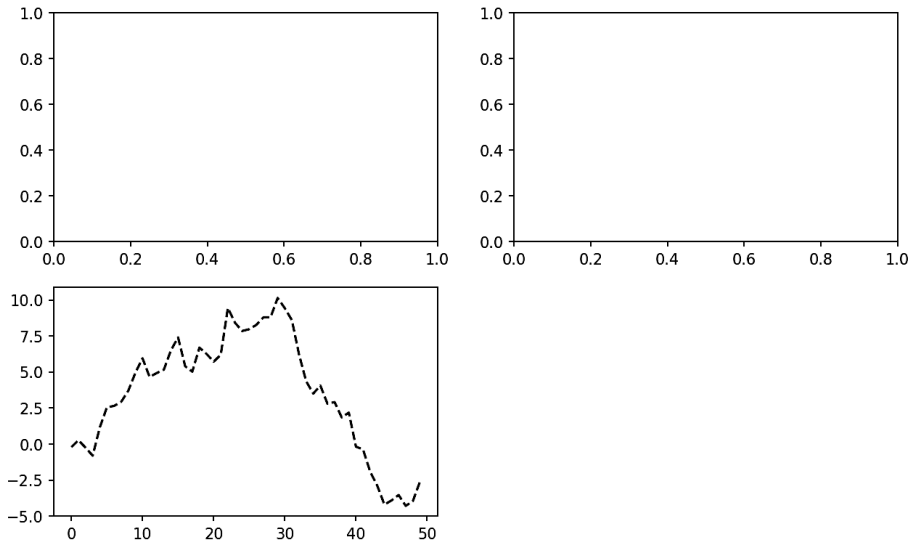
여기서는 아래 코드를 모두 같은 셀에서 실행했다.

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

`plt.plot([1.5, 3.5, -2, 1.6])` 명령으로 그래프를 띄우면 matplotlib은 가장 최근의 figure와 그 서브플롯을 그린다. 서브플롯이 없다면 서브플롯 하나를 생성한다. 이렇게 해서 figure와 서브플롯이 생성되는 과정을 숨겨준다. 따라서 다음 명령을 실행하면 [그림 9-3]과 같은 그래프를 얻을 수 있다.

```
In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--')
```

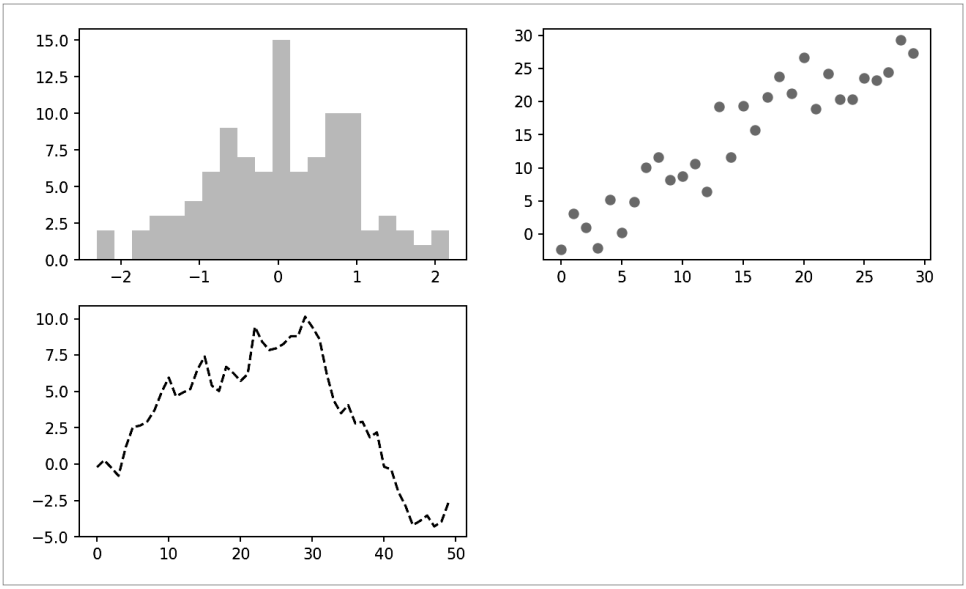
그림 9-3 하나의 그래프를 가지는 figure



'k--' 옵션은 검은 점선을 그리기 위한 **스타일** 옵션이다. fig.add_subplot에서 반환되는 객체는 AxesSubplot인데, 각각의 인스턴스 메서드를 호출해서 다른 빈 서브플롯에 직접 그래프를 그릴 수 있다. [그림 9-4]를 참조하자.

```
In [21]: _ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)

In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
```

그림 9-4 여러 그래프를 추가한 figure



matplotlib 문서에서 여러 가지 그래프 종류를 확인할 수 있다.

특정한 배치에 맞추어 여러 개의 서브플롯을 포함하는 figure를 생성하는 일은 흔히 접하게 되는 업무인데 이를 위한 `plt.subplots`라는 편리한 메서드가 있다. 이 메서드는 NumPy 배열과 서브플롯 객체를 새로 생성하여 반환한다.

```
In [24]: fig, axes = plt.subplots(2, 3)

In [25]: axes
Out[25]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fb626374048>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb62625db00>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6262f6c88>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6261a36a0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb626181860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6260fd4e0>]], dtype
=object)
```

`axes` 배열은 `axes[0, 1]`처럼 2차원 배열로 쉽게 색인될 수 있어서 편리하게 사용할 수 있다. 서브플롯이 같은 x축 혹은 y축을 가져야 한다면 각각 `sharex`와 `sharey`를 사용해서 지정할 수

있다. 같은 범위 내에서 데이터를 비교해야 할 경우 특히 유용하다. 그렇지 않으면 matplotlib은 각 그래프의 범위를 독립적으로 조정한다. 이 메서드에 대한 자세한 내용은 [표 9-1]을 참조하자.

표 9-1 pyplot.subplots 옵션

인자	설명
nrows	서브플롯의 로우 수
ncols	서브플롯의 컬럼 수
sharex	모든 서브플롯이 같은 x축 눈금을 사용하도록 한다(xlim 값을 조절하면 모든 서브플롯에 적용된다).
sharey	모든 서브플롯이 같은 y축 눈금을 사용하도록 한다(ylim 값을 조절하면 모든 서브플롯에 적용된다).
subplot_kw	add_subplot을 사용해서 각 서브플롯을 생성할 때 사용할 키워드를 담고 있는 사전
**fig_kw	figure를 생성할 때 사용할 추가적인 키워드 인자. 예를 들면 plt.subplots(2, 2, figsize=(8, 6))

서브플롯 간의 간격 조절하기

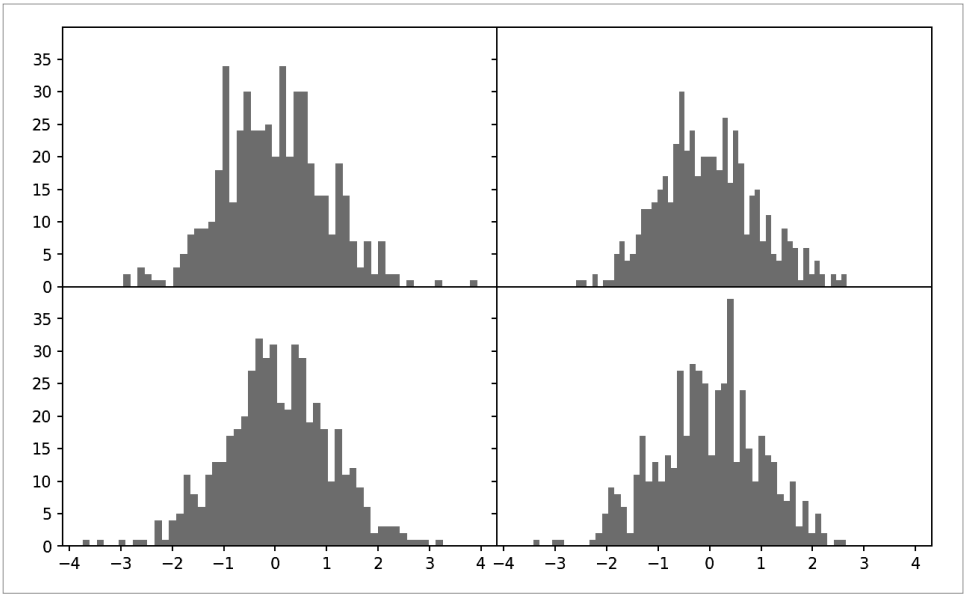
matplotlib은 서브플롯 간에 적당한 간격spacing과 여백padding을 추가해준다. 이 간격은 전체 그래프의 높이와 너비에 따라 상대적으로 결정된다. 그러므로 프로그램을 이용하든 아니면 직접 GUI 윈도우의 크기를 조정하든 그래프의 크기가 자동으로 조절된다. 서브플롯 간의 간격은 Figure 객체의 subplots_adjust 메서드를 사용해서 쉽게 바꿀 수 있다. subplots_adjust 메서드는 최상위 함수로도 존재한다.

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

wspace와 hspace는 서브플롯 간의 간격을 위해 각각 figure의 너비와 높이에 대한 비율을 조절한다. 다음 코드는 서브플롯 간의 간격을 주지 않은 그래프를 생성하는 코드다(그림 9-5).

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

그림 9-5 서브플롯 간의 간격을 주지 않은 그래프



그래프를 그렸을 때 축 이름이 겹치는 경우가 있다. matplotlib은 그래프에서 이름이 겹치는지 검사하지 않기 때문에 이와 같은 경우에는 눈금 위치와 눈금 이름을 명시적으로 직접 지정해야 한다. 다음 절에서 알아보자.

9.1.2 색상, 마커, 선 스타일

matplotlib에서 가장 중요한 `plot` 함수는 x와 y 좌푯값이 담긴 배열과 추가적으로 색상과 선 스타일을 나타내는 축약 문자열을 인자로 받는다. 예를 들어 녹색 점선으로 그려진 x 대 y 그래프는 아래처럼 나타낼 수 있다.

```
ax.plot(x, y, 'g--')
```

이와 같이 문자열로 색상과 선 스타일을 지정하는 방법은 편의를 위해 제공되고 있는데, 실무에서 프로그램으로 그래프를 생성할 때는 그래프를 원하는 형식으로 생성하기 위해 문자열을 지저분하게 섞어 쓰고 싶지 않을 것이다. 위에서 만든 그래프는 아래처럼 좀 더 명시적인 방법으로 표현 가능하다.

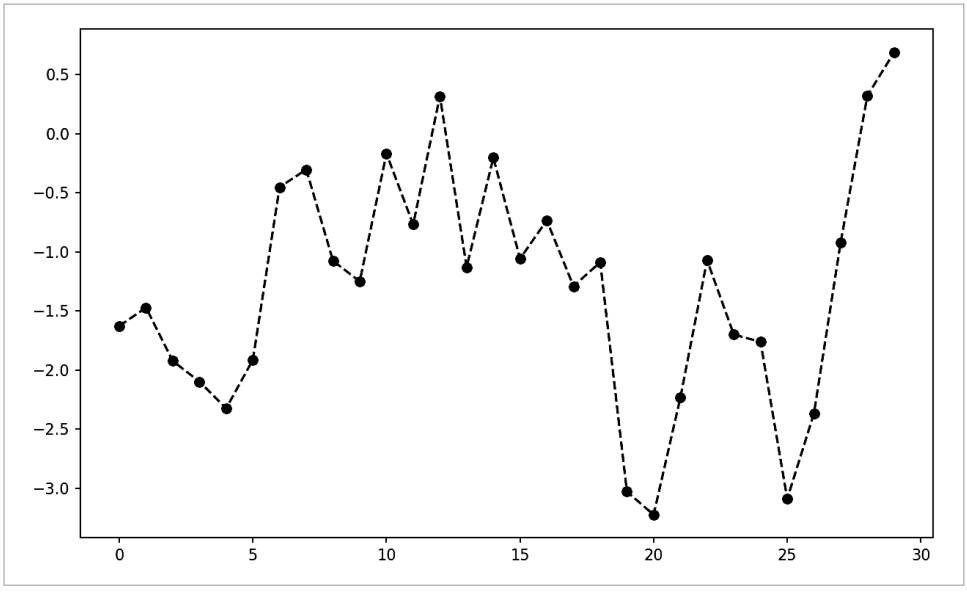
```
ax.plot(x, y, linestyle='--', color='g')
```

흔히 사용되는 색상을 위해 몇 가지 색상 문자열이 존재하지만 RGB 값(예: #CECECE)을 직접 지정해서 색상표에 있는 어떤 색상이라도 지정할 수 있다. 선 스타일에 대한 전체 목록은 plot 메서드의 도움말을 참고하자(IPython이나 주피터에서 plot?을 입력한다).

선그래프는 특정 지점의 실제 데이터를 돋보이게 하기 위해 **마커**를 추가하기도 한다. matplotlib은 점들을 잇는 연속된 선그래프를 생성하기 때문에 어떤 지점에 마커를 설정해야 하는지 확실치 않은 경우가 종종 있다. 마커도 스타일 문자열에 포함시킬 수 있는데 색상 다음에 마커 스타일이 오고 그 뒤에 선 스타일을 지정한다(그림 9-6).

```
In [30]: from numpy.random import randn

In [31]: plt.plot(randn(30).cumsum(), 'ko--')
```

그림 9-6 마커가 포함된 선그래프



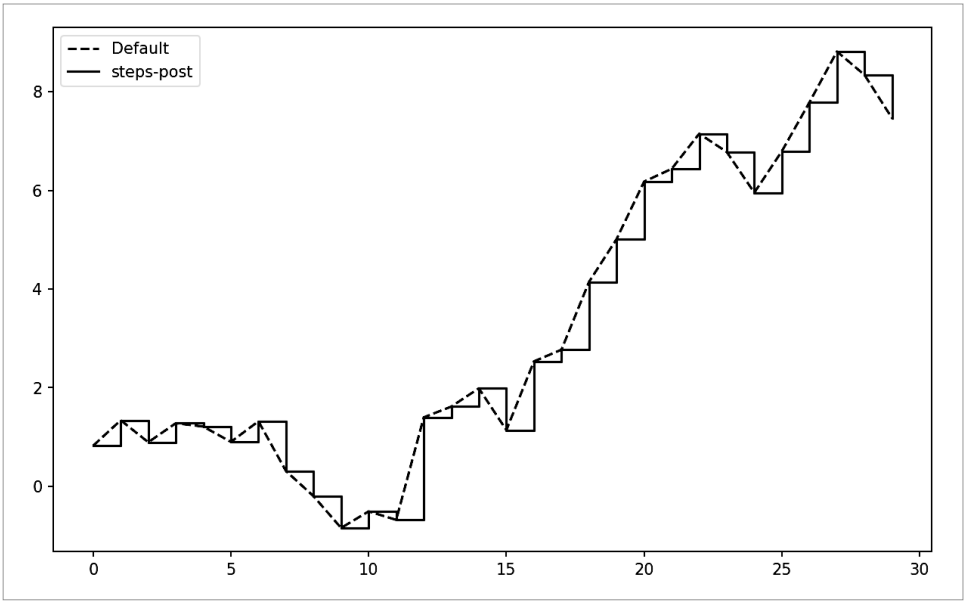
이 역시 좀 더 명시적인 방법으로 표현할 수 있다.

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

선그래프를 보면 일정한 간격으로 연속된 지점이 연결되어 있다. 이 역시 drawstyle 옵션을 이용해서 바꿀 수 있다(그림 9-7).

```
In [33]: data = np.random.randn(30).cumsum()

In [34]: plt.plot(data, 'k--', label='Default')
Out[34]: [<matplotlib.lines.Line2D at 0x7fb624d86160>]

In [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[35]: [<matplotlib.lines.Line2D at 0x7fb624d869e8>]

In [36]: plt.legend(loc='best')
```

그림 9-7 다양한 스타일이 적용된 선그래프



이 코드를 실행해보면 <matplotlib.lines.Line2D at ...>과 같은 결과를 확인할 수 있다. matplotlib은 방금 추가된 그래프의 하위 컴포넌트에 대한 레퍼런스 객체를 리턴한다. 이 결과

CHAPTER 10

데이터 집계와 그룹 연산

데이터셋을 분류하고 각 그룹에 집계나 변형 같은 함수를 적용하는 건 데이터 분석 과정에서 무척 중요한 일이다. 데이터를 불러오고 취합해서 하나의 데이터 집합을 준비하고 나면 그룹 통계를 구하거나 가능하다면 **피벗테이블**을 구해서 보고서를 만들거나 시각화하게 된다. pandas는 데이터 집합을 자연스럽게 나누고 요약할 수 있는 groupby라는 유연한 방법을 제공한다.

관계형 데이터베이스와 SQL[Structured Query Language]이 인기 있는 이유 중 하나는 데이터를 쉽게 합치고 걸러내고 변형하고 집계할 수 있기 때문이다. 하지만 SQL 같은 쿼리문은 그룹 연산에 제약이 있다. 앞으로 살펴보겠지만 파이썬과 pandas의 강력한 표현력을 잘 이용하면 아주 복잡한 그룹 연산도 pandas 객체나 NumPy 배열을 받는 함수의 조합으로 해결할 수 있다. 이 장에서는 다음 내용을 배우게 된다.

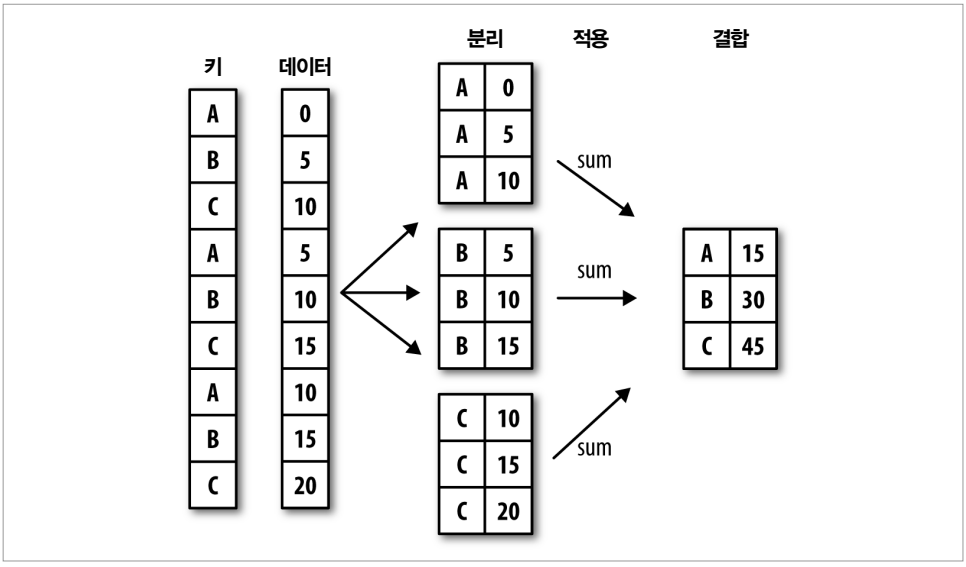
- 하나 이상의 키(함수, 배열, DataFrame의 컬럼 이름)를 이용해서 pandas 객체를 여러 조각으로 나누는 방법
- 합계, 평균, 표준편차, 사용자 정의 함수 같은 그룹 요약 통계를 계산하는 방법
- 정규화, 선형회귀, 등급 또는 부분집합 선택 같은 집단 내 변형이나 다른 조작을 적용하는 방법
- 피벗테이블과 교차일람표를 구하는 방법
- 변위치 분석과 다른 통계 집단 분석을 수행하는 방법

NOTE_ 시계열 데이터의 집계 같은 특수한 groupby 사용 방법을 **리샘플링**이라고 하는데, 이 내용은 11장에서 따로 다룬다.

10.1 GroupBy 메카닉

다수의 인기 있는 R 프로그래밍 패키지의 저자인 해들리 위캠Hadley Wickham은 **분리-적용-결합**split-apply-combine이라는 그룹 연산에 대한 새로운 용어를 만들었는데, 나는 이 말이 그룹 연산에 대한 훌륭한 설명이라고 생각한다. 그룹 연산의 첫 번째 단계에서는 Series, DataFrame 같은 pandas 객체나 아니면 다른 객체에 들어 있는 데이터를 하나 이상의 **키**를 기준으로 **분리**한다. 객체는 하나의 축을 기준으로 분리하는데, 예를 들어 DataFrame은 로우(`axis=0`)로 분리하거나 컬럼(`axis=1`)으로 분리할 수 있다. 분리하고 나서는 함수를 각 그룹에 **적용**시켜 새로운 값을 얻어낸다. 마지막으로 함수를 적용한 결과를 하나의 객체로 **결합**한다. 결과를 담는 객체는 보통 데이터에 어떤 연산을 했는지에 따라 결정된다. 간단한 그룹 연산의 예시를 살펴보자(그림 10-1).

그림 10-1 그룹 연산 예시



각 그룹의 색인은 다음과 같이 다양한 형태가 될 수 있으며, 모두 같은 타입일 필요도 없다.

- 그룹으로 묶을 축과 동일한 길이의 리스트나 배열
- DataFrame의 컬럼 이름을 지칭하는 값
- 그룹으로 묶을 값과 그룹 이름에 대응하는 사전이나 Series 객체
- 축 색인 혹은 색인 내의 개별 이름에 대해 실행되는 함수

앞 목록에서 마지막 세 방법은 객체를 나눌 때 사용할 배열을 생성하기 위한 방법이라는 것을 기억하자. 아직까지 확실한 개념이 잡히지 않는다고 너무 걱정하지 말자. 앞으로 차차 이 방법들을 사용하는 다양한 예제를 살펴보게 될 것이다. 먼저 다음과 같이 DataFrame으로 표현되는 간단한 표 형식의 데이터가 있다고 하자.

```
In [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
   ....:                    'key2' : ['one', 'two', 'one', 'two', 'one'],
   ....:                    'data1' : np.random.randn(5),
   ....:                    'data2' : np.random.randn(5)})

In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

이 데이터를 key1으로 묶고 각 그룹에서 data1의 평균을 구해보자. 여러 가지 방법이 있지만 그중 하나는 data1에 대해 groupby 메서드를 호출하고 key1 컬럼을 넘기는 것이다.

```
In [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa31537390>
```

이 grouped 변수는 GroupBy 객체다. df['key1']로 참조되는 중간값에 대한 것 외에는 아무것도 계산되지 않은 객체다. 이 객체는 그룹 연산을 위해 필요한 모든 정보를 가지고 있어서 각 그룹에 어떤 연산을 적용할 수 있게 해준다. 예를 들어 그룹별 평균을 구하려면 GroupBy 객체의 mean 메서드를 사용하면 된다.

```
In [14]: grouped.mean()
Out[14]:
key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

.mean() 메서드를 호출했을 때의 자세한 내용은 나중에 설명하기로 하고, 이 예제에서 중요한 점은 데이터(Series 객체)가 그룹 색인에 따라 수집되고 key1 컬럼에 있는 유일한 값으로 색인되는 새로운 Series 객체가 생성된다는 것이다. 새롭게 생성된 Series 객체의 색인은 'key1'인데, 그 이유는 DataFrame 컬럼인 df['key1'] 때문이다.

만약 여러 개의 배열을 리스트로 넘겼다면 조금 다른 결과를 얻었을 것이다.

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()

In [16]: means
Out[16]:
key1  key2
a     one     0.880536
      two     0.478943
b     one    -0.519439
      two    -0.555730
Name: data1, dtype: float64
```

여기서는 데이터를 두 개의 색인으로 묶었고, 그 결과 계층적 색인을 가지는 Series를 얻을 수 있었다.

```
In [17]: means.unstack()
Out[17]:
key2       one       two
key1
a     0.880536  0.478943
b    -0.519439 -0.555730
```

이 예제에서는 그룹의 색인 모두 Series 객체인데, 길이만 같다면 어떤 배열이라도 상관없다.

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California  2005    0.478943
            2006   -0.519439
Ohio        2005   -0.380219
```

```
            2006    1.965781
Name: data1, dtype: float64
```

한 그룹으로 묶을 정보는 주로 같은 DataFrame 안에서 찾게 되는데, 이 경우 컬럼 이름(문자열, 숫자 혹은 다른 파이썬 객체)을 넘겨서 그룹의 색인으로 사용할 수 있다.

```
In [21]: df.groupby('key1').mean()
Out[21]:
         data1     data2
key1
a     0.746672  0.910916
b    -0.537585  0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
               data1      data2
key1 key2
a    one    0.880536   1.319920
     two    0.478943   0.092908
b    one   -0.519439   0.281746
     two   -0.555730   0.769023
```

위에서 `df.groupby('key1').mean()` 코드를 보면 key2 컬럼이 결과에서 빠져 있는 것을 확인할 수 있다. 그 이유는 `df['key2']`는 숫자 데이터가 아니기 때문인데, 이런 컬럼은 **성가신 컬럼**nuisance column이라고 부르며 결과에서 제외시킨다. 기본적으로 모든 숫자 컬럼이 수집되지만 곧 살펴보듯이 원하는 부분만 따로 걸러내는 것도 가능하다.

groupby를 쓰는 목적과 별개로, 일반적으로 유용한 GroupBy 메서드는 그룹의 크기를 담고 있는 Series를 반환하는 size 메서드다.

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

그룹 색인에서 누락된 값은 결과에서 제외된다는 것을 기억하자.

10.1.1 그룹 간 순회하기

GroupBy 객체는 이터레이션을 지원하는데, 그룹 이름과 그에 따른 데이터 묶음을 튜플로 반환한다. 다음 예제를 살펴보자.

```
In [24]: for name, group in df.groupby('key1'):
   ....:     print(name)
   ....:     print(group)
   ....:
a
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
4  1.965781  1.246435    a  one
b
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

이처럼 색인이 여럿 존재하는 경우 튜플의 첫 번째 원소가 색인값이 된다.

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....:     print((k1, k2))
   ....:     print(group)
   ....:
('a', 'one')
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
('a', 'two')
      data1     data2 key1 key2
1  0.478943  0.092908    a  two
('b', 'one')
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
('b', 'two')
      data1    data2 key1 key2
3 -0.55573  0.769023    b  two
```

당연히 이 안에서 원하는 데이터만 골라낼 수 있다. 한 줄이면 그룹별 데이터를 사전형으로 쉽게 바꿔서 유용하게 사용할 수 있다.

```
In [26]: pieces = dict(list(df.groupby('key1')))

In [27]: pieces['b']
Out[27]:
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

groupby 메서드는 기본적으로 axis=0에 대해 그룹을 만드는데, 다른 축으로 그룹을 만드는 것도 가능하다. 예를 들어 예제로 살펴본 df의 컬럼을 dtype에 따라 그룹으로 묶을 수도 있다.

```
In [28]: df.dtypes
Out[28]:
data1    float64
data2    float64
key1      object
key2      object
dtype: object

In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

그룹을 아래처럼 출력해볼 수 있다.

```
In [30]: for dtype, group in grouped:
   ....:     print(dtype)
   ....:     print(group)
   ....:
float64
      data1     data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435
object
  key1 key2
```

```
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

10.1.2 컬럼이나 컬럼의 일부만 선택하기

DataFrame에서 만든 GroupBy 객체를 컬럼 이름이나 컬럼 이름이 담긴 배열로 색인하면 수집을 위해 해당 컬럼을 선택하게 된다.

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

위 코드는 아래 코드에 대한 신택틱 슈거로 같은 결과를 반환한다.

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

특히 대용량 데이터를 다룰 경우 소수의 컬럼만 집계하고 싶을 때가 종종 있는데, 예를 들어 위 데이터에서 data2 컬럼에 대해서만 평균을 구하고 결과를 DataFrame으로 받고 싶다면 아래와 같이 작성한다.

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
              data2
key1 key2
a    one   1.319920
     two   0.092908
b    one   0.281746
     two   0.769023
```

색인으로 얻은 객체는 groupby 메서드에 리스트나 배열을 넘겼을 경우 DataFrameGroupBy 객체가 되고, 단일 값으로 하나의 컬럼 이름만 넘겼을 경우 SeriesGroupBy 객체가 된다.

CHAPTER 11

시계열

시계열 데이터는 금융, 경제, 생태학, 신경과학, 물리학 등 여러 다양한 분야에서 사용되는 매우 중요한 구조화된 데이터다. 시간상의 여러 지점을 관측하거나 측정할 수 있는 모든 것이 시계열이다. 대부분의 시계열은 **고정 빈도**fixed frequency로 표현되는데 데이터가 존재하는 지점이 15초마다, 5분마다, 한 달에 한 번 같은 특정 규칙에 따라 고정 간격을 가지게 된다. 시계열은 또한 고정된 단위나 시간 혹은 단위들 간의 간격으로 존재하지 않고 **불규칙적인** 모습으로 표현될 수도 있다. 어떻게 시계열 데이터를 표시하고 참조할지는 애플리케이션에 의존적이며 다음 중 한 유형일 수 있다.

- 시간 내에서 특정 순간의 **타임스탬프**
- 2007년 1월이나 2010년 전체 같은 고정된 **기간**
- 시작과 끝 타임스탬프로 표시되는 시간 **간격**. 기간은 시간 간격의 특수한 경우로 생각할 수 있다.
- 실험 혹은 경과 시간. 각 타임스탬프는 특정 시작 시간에 상대적인 시간의 측정값이다(예: 쿠키를 오븐에 넣은 시점부터 매 초가 지날 때마다 쿠키 반죽의 지름).

실험의 시작 시점부터의 경과 시간이 정수나 부동소수점으로 표현되는 경우 실험 시계열에도 해당 기술들을 적용할 수 있지만 이 장에서는 위에서 소개한 시계열 데이터의 처음 3가지 종류에 대해 주로 알아볼 것이다. 가장 단순하고 널리 사용되는 시계열의 종류는 타임스탬프로 색인된 데이터다.

TIP pandas는 시간차에 기반한 색인을 지원하며, 이는 경과 시간을 나타낼 때 유용하다. 이 책에서는 시간차 색인을 알아보지 않겠지만 pandas 공식 문서에서 자세한 내용을 찾아볼 수 있다.

pandas는 표준 시계열 도구와 데이터 알고리즘을 제공한다. 이를 통해 대량의 시계열 데이터를 효과적으로 다룰 수 있으며 쉽게 나누고, 집계하고, 불규칙적이며 고정된 빈도를 갖는 시계열을 리샘플링할 수 있다. 눈치 챘겠지만 대부분의 도구는 금융이나 경제 관련 애플리케이션에서 특히 유용하다. 하지만 서버 로그 데이터를 분석하는 데도 사용할 수 있다.

11.1 날짜, 시간 자료형, 도구

파이썬 표준 라이브러리는 날짜와 시간을 위한 자료형과 달력 관련 기능을 제공하는 자료형이 존재한다. `datetime`, `time` 그리고 `calendar` 모듈은 처음 공부하기에 좋은 주제다. `datetime.datetime`형이나 단순한 `datetime`이 널리 사용되고 있다.

```
In [10]: from datetime import datetime

In [11]: now = datetime.now()

In [12]: now
Out[12]: datetime.datetime(2017, 9, 25, 14, 5, 52, 72973)

In [13]: now.year, now.month, now.day
Out[13]: (2017, 9, 25)
```

`datetime`은 날짜와 시간을 모두 저장하며 마이크로초까지 지원한다. `datetime.timedelta`는 두 `datetime` 객체 간의 시간적인 차이를 표현할 수 있다.

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [15]: delta
Out[15]: datetime.timedelta(926, 56700)

In [16]: delta.days
Out[16]: 926

In [17]: delta.seconds
Out[17]: 56700
```

timedelta를 더하거나 빼면 그만큼의 시간이 datetime 객체에 적용되어 새로운 객체를 만들 수 있다.

```
In [18]: from datetime import timedelta

In [19]: start = datetime(2011, 1, 7)

In [20]: start + timedelta(12)
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)

In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

[표 11-1]에 datetime 모듈의 자료형을 정리해두었다. 이 장에서 주로 다루는 내용은 pandas의 자료형과 고수준의 시계열을 다루는 방법이며, 실제 파이썬을 사용하면서 다양한 곳에서 datetime 기반의 자료형을 마주치게 되리라는 점은 의심할 여지가 없다.

표 11-1 datetime 모듈의 자료형

자료형	설명
date	그레고리안 달력을 사용해서 날짜(연, 월, 일)를 저장한다.
time	하루의 시간을 시, 분, 초, 마이크로초 단위로 저장한다.
datetime	날짜와 시간을 저장한다.
timedelta	두 datetime 값 간의 차이(일, 초, 마이크로초)를 표현한다.
tzinfo	지역시간대를 저장하기 위한 기본 자료형

11.1.1 문자열을 datetime으로 변환하기

datetime 객체와 나중에 소개할 pandas의 Timestamp 객체는 str 메서드나 strftime 메서드에 포맷 규칙을 넘겨서 문자열로 나타낼 수 있다.

```
In [22]: stamp = datetime(2011, 1, 3)

In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'

In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

[표 11-2]에 포맷 코드를 모두 정리해두었다(2장에서 소개했던 내용이다).

표 11-2 Datetime 포맷 규칙(ISO C89 호환)

포맷	설명
%Y	4자리 연도
%y	2자리 연도
%m	2자리 월 [01, 12]
%d	2자리 일 [01, 31]
%H	시간(24시간 형식) [00, 23]
%I	시간(12시간 형식) [01, 12]
%M	2자리 분 [00, 59]
%S	초 [00, 61] (60, 61은 윤초)
%w	정수로 나타낸 요일 [0(일요일), 6]
%U	연중 주차 [00, 53]. 일요일을 그 주의 첫 번째 날로 간주하며, 그 해에서 첫 번째 일요일 앞에 있는 날은 0주차가 된다.
%W	연중 주차 [00, 53]. 월요일을 그 주의 첫 번째 날로 간주하며, 그 해에서 첫 번째 월요일 앞에 있는 날은 0주차가 된다.
%z	UTC 시간대 오프셋을 +HHMM 또는 -HHMM으로 표현한다. 만약 시간대를 신경 쓰지 않는다면 비워둔다.
%F	%Y-%m-%d 형식에 대한 축약(예: 2012-4-18)
%D	%m/%d/%y 형식에 대한 축약(예: 04/18/12)

이 포맷 코드는 datetime.strptime을 사용해서 문자열을 날짜로 변환할 때 사용할 수 있다.

```
In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[28]:
[datetime.datetime(2011, 7, 6, 0, 0),
 datetime.datetime(2011, 8, 6, 0, 0)]
```

datetime.strptime은 알려진 형식의 날짜를 파싱하는 최적의 방법이다. 하지만 매번 포맷 규칙을 써야 하는 건 귀찮은 일이다. 특히 흔히 쓰는 날짜 형식에 대해서는 더 그렇다. 이 경우에는 서드파티 패키지인 dateutil에 포함된 parser.parse 메서드를 사용하면 된다(pandas를 설치할 때 자동으로 함께 설치된다).

```
In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

dateutil은 거의 대부분의 사람이 인지하는 날짜 표현 방식을 파싱할 수 있다.

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

국제 로케일의 경우 날짜가 월 앞에 오는 경우가 매우 흔하다. 이런 경우에는 dayfirst=True를 넘겨주면 된다.

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas는 일반적으로 DataFrame의 컬럼이나 축 색인으로 날짜가 담긴 배열을 사용한다. to_datetime 메서드는 많은 종류의 날짜 표현을 처리한다. ISO 8601 같은 표준 날짜 형식은 매우 빠르게 처리할 수 있다.

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'], dtype='dat
etime64[ns]', freq=None)
```

또한 누락된 값(None, 빈 문자열 등)으로 간주되어야 할 값도 처리해준다.

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'], dty
pe='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT
```

```
In [38]: pd.isnull(idx)
Out[38]: array([False, False, True], dtype=bool)
```

NaT[Not a Time]는 pandas에서 누락된 타임스탬프 데이터를 나타낸다.

> **CAUTION_** dateutil.parser는 매우 유용하지만 완벽한 도구는 아니다. 날짜로 인식하지 않길 바라는 문자열을 날짜로 인식하기도 하는데, '42'를 2042년으로 해석하기도 한다.

datetime 객체는 여러 나라 혹은 언어에서 사용하는 로케일에 맞는 다양한 포맷 옵션을 제공한다. 예를 들어 독일과 프랑스에서는 각 월의 단축명이 영문 시스템과 다르다. [표 11-3]에서 로케일별 날짜 포맷을 확인하자.

표 11-3 로케일별 날짜 포맷

포맷	설명
%a	축약된 요일 이름
%A	요일 이름
%b	축약된 월 이름
%B	월 이름
%c	전체 날짜와 시간(예: 'Tue 01 May 2012 04:20:57 PM')
%p	해당 로케일에서 AM, PM에 대응되는 이름(AM은 오전, PM은 오후)
%x	로케일에 맞는 날짜 형식(예: 미국이라면 2012년 5월 1일은 '05/01/2012')
%X	로케일에 맞는 시간 형식(예: '04:24:12 PM')

11.2 시계열 기초

pandas에서 찾아볼 수 있는 가장 기본적인 시계열 객체의 종류는 파이썬 문자열이나 datetime 객체로 표현되는 타임스탬프로 색인된 Series다.

```
In [39]: from datetime import datetime

In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
   ....:          datetime(2011, 1, 7), datetime(2011, 1, 8),
   ....:          datetime(2011, 1, 10), datetime(2011, 1, 12)]
```

```
In [41]: ts = pd.Series(np.random.randn(6), index=dates)

In [42]: ts
Out[42]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

내부적으로 보면 이들 datetime 객체는 DatetimeIndex에 들어 있으며 ts 변수의 타입은 TimeSeries다.

```
In [43]: ts.index
Out[43]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

다른 Series와 마찬가지로 서로 다르게 색인된 시계열 객체 간의 산술 연산은 자동으로 날짜에 맞춰진다.

```
In [44]: ts + ts[::2]
Out[44]:
2011-01-02   -0.409415
2011-01-05         NaN
2011-01-07   -1.038877
2011-01-08         NaN
2011-01-10    3.931561
2011-01-12         NaN
dtype: float64
```

ts[::2]는 ts에서 매 두 번째 항목을 선택한다.

pandas는 NumPy의 datetime64 자료형을 사용해서 나노초의 정밀도를 가지는 타임스탬프를 저장한다.

```
In [45]: ts.index.dtype
Out[45]: dtype('<M8[ns]')
```

DatetimeIndex의 스칼라값은 pandas의 Timestamp 객체다.

```
In [46]: stamp = ts.index[0]

In [47]: stamp
Out[47]: Timestamp('2011-01-02 00:00:00')
```

Timestamp는 datetime 객체를 사용하는 어떤 곳에도 대체 사용이 가능하다. 게다가 가능하다면 빈도에 관한 정보도 저장하며 시간대 변환을 하는 방법과 다른 종류의 조작을 하는 방법도 포함하고 있다. 자세한 내용은 차후에 다루도록 하겠다.

11.2.1 색인, 선택, 부분 선택

시계열은 라벨에 기반해서 데이터를 선택하고 인덱싱할 때 pandas.Series와 동일하게 동작한다.

```
In [48]: stamp = ts.index[2]

In [49]: ts[stamp]
Out[49]: -0.51943871505673811
```

해석할 수 있는 날짜를 문자열로 넘겨서 편리하게 사용할 수 있다.

```
In [50]: ts['1/10/2011']
Out[50]: 1.9657805725027142

In [51]: ts['20110110']
Out[51]: 1.9657805725027142
```

긴 시계열에서는 연을 넘기거나 연, 월만 넘겨서 데이터의 일부 구간만 선택할 수도 있다.

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),
   ....:                       index=pd.date_range('1/1/2000', periods=1000))

In [53]: longer_ts
Out[53]:
2000-01-01    0.092908
2000-01-02    0.281746
2000-01-03    0.769023
2000-01-04    1.246435
2000-01-05    1.007189
2000-01-06   -1.296221
2000-01-07    0.274992
2000-01-08    0.228913
2000-01-09    1.352917
2000-01-10    0.886429
                ...
2002-09-17   -0.139298
2002-09-18   -1.159926
2002-09-19    0.618965
2002-09-20    1.373890
2002-09-21   -0.983505
2002-09-22    0.930944
2002-09-23   -0.811676
2002-09-24   -1.830156
2002-09-25   -0.138730
2002-09-26    0.334088
Freq: D, Length: 1000, dtype: float64

In [54]: longer_ts['2001']
Out[54]:
2001-01-01    1.599534
2001-01-02    0.474071
2001-01-03    0.151326
2001-01-04   -0.542173
2001-01-05   -0.475496
2001-01-06    0.106403
2001-01-07   -1.308228
2001-01-08    2.173185
2001-01-09    0.564561
2001-01-10   -0.190481
                ...
2001-12-22    0.000369
2001-12-23    0.900885
```

```
2001-12-24   -0.454869
2001-12-25   -0.864547
2001-12-26    1.129120
2001-12-27    0.057874
2001-12-28   -0.433739
2001-12-29    0.092698
2001-12-30   -1.397820
2001-12-31    1.457823
Freq: D, Length: 365, dtype: float64
```

여기서 문자열 '2001'은 연도로 해석되어 해당 기간의 데이터를 선택한다. 월에 대해서도 마찬가지로 선택할 수 있다.

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01   -0.622547
2001-05-02    0.936289
2001-05-03    0.750018
2001-05-04   -0.056715
2001-05-05    2.300675
2001-05-06    0.569497
2001-05-07    1.489410
2001-05-08    1.264250
2001-05-09   -0.761837
2001-05-10   -0.331617
                ...
2001-05-22    0.503699
2001-05-23   -1.387874
2001-05-24    0.204851
2001-05-25    0.603705
2001-05-26    0.545680
2001-05-27    0.235477
2001-05-28    0.111835
2001-05-29   -1.251504
2001-05-30   -2.949343
2001-05-31    0.634634
Freq: D, Length: 31, dtype: float64
```

datetime 객체로 데이터를 잘라내는 작업은 일반적인 Series와 동일한 방식으로 할 수 있다.

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

대부분의 시계열 데이터는 연대순으로 정렬되기 때문에 범위를 지정하기 위해 시계열에 포함하지 않고 타임스탬프를 이용해서 Series를 나눌 수 있다.

```
In [57]: ts
Out[57]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [58]: ts['1/6/2011':'1/11/2011']
Out[58]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
dtype: float64
```

앞서와 같이 날짜 문자열이나 `datetime` 혹은 타임스탬프를 넘길 수 있다. 이런 방식으로 데이터를 나누면 NumPy 배열을 나누는 것처럼 원본 시계열에 대한 뷰를 생성한다는 사실을 기억하자. 즉, 데이터 복사가 발생하지 않고 슬라이스에 대한 변경이 원본 데이터에도 반영된다.

이와 동일한 인스턴스 메서드로 `truncate`가 있는데, 이 메서드는 TimeSeries를 두 개의 날짜로 나눈다.

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02   -0.204708
2011-01-05    0.478943
```

```
2011-01-07   -0.519439
2011-01-08   -0.555730
dtype: float64
```

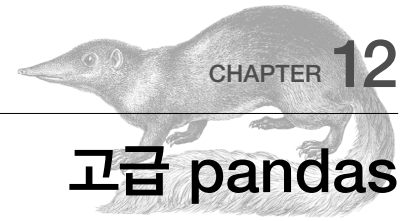
위 방식은 DataFrame에서도 동일하게 적용되며 로우에 인덱싱된다.

```
In [60]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')

In [61]: long_df = pd.DataFrame(np.random.randn(100, 4),
   ....:                        index=dates,
   ....:                        columns=['Colorado', 'Texas',
   ....:                                 'New York', 'Ohio'])

In [62]: long_df.loc['5-2001']
Out[62]:
            Colorado     Texas  New York      Ohio
2001-05-02 -0.006045  0.490094 -0.277186 -0.707213
2001-05-09 -0.560107  2.735527  0.927335  1.513906
2001-05-16  0.538600  1.273768  0.667876 -0.969206
2001-05-23  1.676091 -0.817649  0.050188  1.951312
2001-05-30  3.260383  0.963301  1.201206 -1.852001
```

11.2.2 중복된 색인을 갖는 시계열

어떤 애플리케이션에서는 여러 데이터가 특정 타임스탬프에 몰려 있는 것을 발견할 수 있다.

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000',
   ....:                           '1/2/2000', '1/3/2000'])

In [64]: dup_ts = pd.Series(np.arange(5), index=dates)

In [65]: dup_ts
Out[65]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```



CHAPTER 12

고급 pandas

지금까지는 다양한 종류의 데이터를 다듬는 과정과 NumPy, pandas 그리고 다른 라이브러리의 기능을 소개했다. 시간이 갈수록 pandas에는 고급 사용자를 위한 깊이 있는 기능들이 추가되고 있다. 이 장에서는 고급 pandas 사용자가 되기 위한 몇 가지 고급 기능을 소개하겠다.

12.1 Categorical 데이터

이 절에서는 pandas의 `Categorical`형을 활용하여 pandas 메모리 사용량을 줄이고 성능을 개선할 수 있는 방법을 소개한다. 통계와 머신러닝에서 범주형 데이터를 활용하기 위한 도구들도 함께 소개하겠다.

12.1.1 개발 배경과 동기

하나의 컬럼 내에 특정 값이 반복되어 존재하는 경우는 흔하다. 우리는 이미 배열 내에서 유일한 값을 추출하거나 특정 값이 얼마나 많이 존재하는지 확인할 수 있는 `unique`와 `value_counts` 같은 메서드를 공부했다.

```
In [10]: import numpy as np; import pandas as pd
```

```
In [11]: values = pd.Series(['apple', 'orange', 'apple',
   ....:                     'apple'] * 2)

In [12]: values
Out[12]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
dtype: object

In [13]: pd.unique(values)
Out[13]: array(['apple', 'orange'], dtype=object)

In [14]: pd.value_counts(values)
Out[14]:
apple     6
orange    2
dtype: int64
```

데이터웨어하우스, 분석 컴퓨팅 외 여러 다양한 데이터 시스템은 중복되는 데이터를 얼마나 효율적으로 저장하고 계산할 수 있는가를 중점으로 개발되었다. 데이터웨어하우스의 경우 구별되는 값을 담고 있는 **차원 테이블**과 그 테이블을 참조하는 정수키를 사용하는 것이 일반적이다.

```
In [15]: values = pd.Series([0, 1, 0, 0] * 2)

In [16]: dim = pd.Series(['apple', 'orange'])

In [17]: values
Out[17]:
0    0
1    1
2    0
3    0
4    0
5    1
6    0
```

```
7    0
dtype: int64

In [18]: dim
Out[18]:
0     apple
1    orange
dtype: object
```

take 메서드를 사용하면 Series 내에 저장된 원래 문자열을 구할 수 있다.

```
In [19]: dim.take(values)
Out[19]:
0     apple
1    orange
0     apple
0     apple
0     apple
1    orange
0     apple
0     apple
dtype: object
```

여기서 정수로 표현된 값은 **범주형** 또는 **사전형 표기법**이라고 한다. 별개의 값을 담고 있는 배열은 **범주**, **사전** 또는 **단계 데이터**라고 부른다. 이 책에서는 이런 종류의 데이터를 categorical 또는 범주형 데이터라고 부르겠다. 범주형 데이터를 가리키는 정숫값은 **범주 코드** 또는 그냥 단순히 **코드**라고 한다.

범주형 표기법을 사용하면 분석 작업에 있어서 엄청난 성능 향상을 얻을 수 있다. 범주 코드를 변경하지 않은 채로 범주형 데이터를 변형하는 것도 가능하다. 비교적 적은 연산으로 수행할 수 있는 변형의 예는 다음과 같다.

- 범주형 데이터의 이름 변경하기
- 기존 범주형 데이터의 순서를 바꾸지 않고 새로운 범주 추가하기

12.1.2 pandas의 Categorical

pandas에는 정수 기반의 범주형 데이터를 표현(또는 **인코딩**)할 수 있는 Categorical형이라고 하는 특수한 데이터형이 존재한다. 앞서 살펴본 Series를 다시 보자.

```
In [20]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2

In [21]: N = len(fruits)

In [22]: df = pd.DataFrame({'fruit': fruits,
   ....:                    'basket_id': np.arange(N),
   ....:                    'count': np.random.randint(3, 15, size=N),
   ....:                    'weight': np.random.uniform(0, 4, size=N)},
   ....:                   columns=['basket_id', 'fruit', 'count', 'weight'])

In [23]: df
Out[23]:
   basket_id   fruit  count    weight
0          0   apple      5  3.858058
1          1  orange      8  2.612708
2          2   apple      4  2.995627
3          3   apple      7  2.614279
4          4   apple     12  2.990859
5          5  orange      8  3.845227
6          6   apple      5  0.033553
7          7   apple      4  0.425778
```

아래 예제에서 df['fruit']는 파이썬 문자열 객체의 배열로, 아래 방법으로 쉽게 범주형 데이터로 변경할 수 있다.

```
In [24]: fruit_cat = df['fruit'].astype('category')

In [25]: fruit_cat
Out[25]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
```

```
7     apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

fruit_cat의 값은 NumPy 배열이 아니라 pandas.Categorical의 인스턴스다.

```
In [26]: c = fruit_cat.values

In [27]: type(c)
Out[27]: pandas.core.categorical.Categorical
```

Categorical 객체는 categories와 codes 속성을 가진다.

```
In [28]: c.categories
Out[28]: Index(['apple', 'orange'], dtype='object')

In [29]: c.codes
Out[29]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

변경 완료된 값을 대입함으로써 DataFrame의 컬럼을 범주형으로 변경할 수 있다.

```
In [30]: df['fruit'] = df['fruit'].astype('category')

In [31]: df.fruit
Out[31]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

파이썬 열거형에서 pandas.Categorical형을 직접 생성하는 것도 가능하다.

```
In [32]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])

In [33]: my_categories
Out[33]:
[foo, bar, baz, foo, bar]
Categories (3, object): [bar, baz, foo]
```

기존에 정의된 범주와 범주 코드가 있다면 from_codes 함수를 이용해서 범주형 데이터를 생성하는 것도 가능하다.

```
In [34]: categories = ['foo', 'bar', 'baz']

In [35]: codes = [0, 1, 2, 0, 0, 1]

In [36]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [37]: my_cats_2
Out[37]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

범주형으로 변경하는 경우 명시적으로 지정하지 않는 한 특정 순서를 보장하지 않는다. 따라서 categories 배열은 입력 데이터의 순서에 따라 다른 순서로 나타날 수 있다. from_codes를 사용하거나 다른 범주형 데이터 생성자를 이용하는 경우 순서를 지정할 수 있다.

```
In [38]: ordered_cat = pd.Categorical.from_codes(codes, categories,
   ....:                                         ordered=True)

In [39]: ordered_cat
Out[39]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

여기서 [foo < bar < baz]는 foo, bar, baz 순서를 가진다는 의미다. 순서가 없는 범주형 인스턴스는 as_ordered 메서드를 이용해 순서를 가지도록 만들 수 있다.

```
In [40]: my_cats_2.as_ordered()
```

```
Out[40]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

여기서는 문자열만 예로 들었지만 범주형 데이터는 꼭 문자열일 필요는 없다. 범주형 배열은 변경이 불가능한 값이라면 어떤 자료형이라도 포함할 수 있다.

12.1.3 Categorical 연산

pandas에서 Categorical은 문자열 배열처럼 인코딩되지 않은 자료형을 사용하는 방식과 거의 유사하게 사용할 수 있다. groupby 같은 일부 pandas 함수는 범주형 데이터에 사용할 때 더 나은 성능을 보여준다. ordered 플래그를 활용하는 함수들도 마찬가지다.

임의의 숫자 데이터를 pandas.qcut 함수로 구분해보자. 그렇게 하면 pandas.Categorical 객체를 반환한다. 책 초반부에서 pandas.cut 함수를 살펴봤지만 어떻게 범주형 데이터를 다루는지는 제대로 설명하지 않았다.

```
In [41]: np.random.seed(12345)

In [42]: draws = np.random.randn(1000)

In [43]: draws[:5]
Out[43]: array([-0.2047, 0.4789, -0.5194, -0.5557, 1.9658])
```

이 데이터를 사분위로 나누고 통계를 내보자.

```
In [44]: bins = pd.qcut(draws, 4)

In [45]: bins
Out[45]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0101], (0.63,
3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.95, -0.684], (-0.0101, 0.63
], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.684] < (-0.684, -0.0101] < (-0.010
1, 0.63] <
                                    (0.63, 3.928]]
```

사분위 이름을 실제 데이터로 지정하는 것은 별로 유용하지 않아 보인다. qcut 함수의 labels 인자로 직접 이름을 지정하자.

```
In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [47]: bins
Out[47]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]
Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

bins에 이름을 붙이고 나면 데이터의 시작값과 끝값에 대한 정보를 포함하지 않으므로 groupby를 이용해서 요약 통계를 내보자.

```
In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)
   ....:            .groupby(bins)
   ....:            .agg(['count', 'min', 'max'])
   ....:            .reset_index())

In [51]: results
Out[51]:
  quartile  count       min       max
0       Q1    250 -2.949343 -0.685484
1       Q2    250 -0.683066 -0.010115
2       Q3    250 -0.010032  0.628894
3       Q4    250  0.634238  3.927528
```

결과에서 quartile 컬럼은 bins의 순서를 포함한 원래 범주 정보를 유지하고 있다.

```
In [52]: results['quartile']
Out[52]:
0    Q1
1    Q2
2    Q3
```

```
3    Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

categorical을 이용한 성능 개선

특정 데이터셋에 대해 다양한 분석을 하는 경우 범주형categorical으로 변환하는 것만으로도 전체 성능을 개선할 수 있다. 범주형으로 변환한 DataFrame의 컬럼은 메모리도 훨씬 적게 사용한다. 소수의 독립적인 카테고리로 분류되는 천만 개의 값을 포함하는 Series를 살펴보자.

```
In [53]: N = 10000000

In [54]: draws = pd.Series(np.random.randn(N))

In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

labels를 categorical로 변환하자.

```
In [56]: categories = labels.astype('category')
```

categories가 labels에 비해 훨씬 더 적은 메모리를 사용하는 것을 확인할 수 있다.

```
In [57]: labels.memory_usage()
Out[57]: 80000080

In [58]: categories.memory_usage()
Out[58]: 10000272
```

범주형으로 변환하는 과정이 그냥 이루어지는 것은 아니지만 이는 한 번만 변환하면 되는 일회성 비용이다.

```
In [59]: %time _ = labels.astype('category')
CPU times: user 490 ms, sys: 240 ms, total: 730 ms
Wall time: 726 ms
```

범주형에 대한 그룹 연산은 문자열 배열을 사용하는 대신 정수 기반의 코드 배열을 사용하는 알고리즘으로 동작하므로 훨씬 빠르게 동작한다.

12.1.4 Categorical 메서드

범주형 데이터를 담고 있는 Series는 특화된 문자열 메서드인 Series.str과 유사한 몇 가지 특수 메서드를 제공한다. 이를 통해 categories와 codes에 쉽게 접근할 수 있다. 다음 Series를 살펴보자.

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [61]: cat_s = s.astype('category')

In [62]: cat_s
Out[62]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

특별한 속성인 cat을 통해 categorical 메서드에 접근할 수 있다.

```
In [63]: cat_s.cat.codes
Out[63]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8
```

```
In [64]: cat_s.cat.categories
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

이 데이터의 실제 카테고리가 데이터에서 관측되는 4종류를 넘는 것을 이미 알고 있다고 가정하자. 이 경우 set_categories 메서드를 이용해서 변경하는 것이 가능하다.

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e']

In [66]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [67]: cat_s2
Out[67]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

데이터는 변함이 없지만 위에서 변경한 대로 새로운 카테고리가 추가되었다. 예를 들어 value_counts를 호출해보면 변경된 카테고리를 반영하고 있다.

```
In [68]: cat_s.value_counts()
Out[68]:
d    2
c    2
b    2
a    2
dtype: int64

In [69]: cat_s2.value_counts()
Out[69]:
d    2
c    2
b    2
a    2
```

```
e    0
dtype: int64
```

큰 데이터셋을 다룰 경우 categorical을 이용하면 메모리를 아끼고 성능도 개선할 수 있다. 분석 과정에서 큰 DataFrame이나 Series를 한 번 걸러내고 나면 실제로 데이터에는 존재하지 않는 카테고리가 남아 있을 수 있다. 이 경우 `remove_unused_categories` 메서드를 이용해서 관측되지 않는 카테고리를 제거할 수 있다.

```
In [70]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [71]: cat_s3
Out[71]:
0    a
1    b
4    a
5    b
dtype: category
Categories (4, object): [a, b, c, d]

In [72]: cat_s3.cat.remove_unused_categories()
Out[72]:
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): [a, b]
```

[표 12-1]에 categorical 메서드의 종류를 나열해두었다.

표 12-1 categorical 메서드

메서드	설명
add_categories	기존 카테고리 끝에 새로운 카테고리를 추가한다.
as_ordered	카테고리가 순서를 가지도록 한다.
as_unordered	카테고리가 순서를 가지지 않도록 한다.
remove_categories	카테고리를 제거한다. 해당 카테고리에 속한 값들은 null로 설정한다.
remove_unused_categories	데이터에서 관측되지 않는 카테고리를 삭제한다.
rename_categories	카테고리 이름을 지정한 이름으로 변경한다. 카테고리 수는 변하지 않는다.

CHAPTER 13

파이썬 모델링 라이브러리

이 책은 파이썬을 활용한 데이터 분석에 필요한 기본 프로그래밍 실력을 키우는 데 초점을 맞추었다. 데이터 분석가와 과학자들은 데이터를 정제하고 준비하는 데 너무 많은 시간을 쓰고 있으며 이 책에서도 관련 기법을 습득하는 데 많은 지면을 할애했다.

모델을 개발하는 데 어떤 라이브러리를 사용할지는 어떤 애플리케이션에 적용하느냐에 따라 달라진다. 많은 통계 문제는 최소제곱회귀 같은 단순한 기법으로 해결할 수 있으며 어떤 문제는 고급 머신러닝 방식으로 해결할 수 있다. 다행히도 파이썬은 이런 분석 기법들을 구현할 수 있는 언어 중 하나가 되었고 이 책을 다 읽은 후에도 더 공부할 많은 도구가 존재한다.

이 장에서는 모델 피팅 및 스코어링과 pandas를 이용한 데이터 정제 작업 사이를 오가는 와중에 도움이 될 만한 pandas의 기능을 살펴보겠다. 그리고 유명한 모델링 도구인 statsmodels와 scikit-learn을 간략히 소개한다. 이 두 프로젝트는 그 자체로도 책 한권이 필요할 정도의 방대한 프로젝트이므로 전체를 다루기보다는 두 프로젝트의 온라인 문서와 데이터 과학, 통계, 그리고 머신러닝을 다루는 다른 파이썬 서적을 소개하는 것으로 대신하려 한다.

13.1 pandas와 모델 코드의 인터페이스

모델 개발의 일반적인 흐름은 데이터를 불러오고 정제하는 과정은 pandas를 이용하고 그 후 모델 개발을 위해 모델링 라이브러리로 넘어가게 된다. 모델을 개발하는 과정에서 중요한 단계

는 특징을 선택하고 추출하는 **피처 엔지니어링**인데 원시 데이터셋으로부터 모델링에서 유용할 수 있는 정보를 추출하는 변환이나 분석 과정을 일컫는다. 이 책에서 살펴본 데이터 요약이나 GroupBy 도구들이 피처 엔지니어링 과정에서 자주 사용된다.

'좋은' 피처 엔지니어링에 대한 자세한 내용은 이 책의 범위를 벗어나므로 pandas를 이용한 데이터 조작과 모델링 사이를 편리하게 오갈 수 있는 방법을 설명하겠다.

pandas와 다른 분석 라이브러리는 주로 NumPy 배열을 사용해서 연계할 수 있다. DataFrame을 NumPy 배열로 변환하려면 `.values` 속성을 이용한다.

```
In [10]: import pandas as pd

In [11]: import numpy as np

In [12]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [13]: data
Out[13]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0

In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

In [15]: data.values
Out[15]:
array([[ 1. ,  0.01, -1.5 ],
       [ 2. , -0.01,  0.  ],
       [ 3. ,  0.25,  3.6 ],
       [ 4. , -4.1 ,  1.3 ],
       [ 5. ,  0.  , -2.  ]])
```

다시 DataFrame으로 되돌리려면 앞서 공부했던 것처럼 2차원 ndarray와 필요하다면 컬럼 이름 리스트를 넘겨서 생성할 수 있다.

```
In [16]: df2 = pd.DataFrame(data.values, columns=['one', 'two', 'three'])

In [17]: df2
Out[17]:
   one   two  three
0  1.0  0.01   -1.5
1  2.0 -0.01    0.0
2  3.0  0.25    3.6
3  4.0 -4.10    1.3
4  5.0  0.00   -2.0
```

NOTE_ .values 속성은 데이터가 한 가지 타입(예를 들면 모두 숫자형)으로 이루어져 있다는 가정 하에 사용된다. 만약 데이터 속성이 한 가지가 아니라면 파이썬 객체의 ndarrary가 반환될 것이다.

```
In [18]: df3 = data.copy()

In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']

In [20]: df3
Out[20]:
   x0    x1    y strings
0   1  0.01 -1.5       a
1   2 -0.01  0.0       b
2   3  0.25  3.6       c
3   4 -4.10  1.3       d
4   5  0.00 -2.0       e

In [21]: df3.values
Out[21]:
array([[1,  0.01, -1.5, 'a'],
       [2, -0.01,  0.0, 'b'],
       [3,  0.25,  3.6, 'c'],
       [4, -4.1 ,  1.3, 'd'],
       [5,  0.0 , -2.0, 'e']], dtype=object)
```

어떤 모델은 전체 컬럼 중 일부만 사용하고 싶은 경우도 있을 것이다. 이 경우 loc을 이용해서 values 속성에 접근하기 바란다.

```
In [22]: model_cols = ['x0', 'x1']
```

```
In [23]: data.loc[:, model_cols].values
Out[23]:
array([[ 1. ,  0.01],
       [ 2. , -0.01],
       [ 3. ,  0.25],
       [ 4. , -4.1 ],
       [ 5. ,  0.  ]])
```

어떤 라이브러리는 pandas를 직접 지원하기도 하는데 위에서 설명한 과정을 자동으로 처리해 준다. DataFrame에서 NumPy 배열로 변환하고 모델 인자의 이름을 출력 테이블이나 Series의 컬럼으로 추가한다. 아니라면 이런 메타 데이터 관리를 수동으로 직접 해야 한다.

12장에서 pandas의 `Categorical`형과 `pandas.get_dummies` 함수를 살펴봤다. 예제 데이터셋에 숫자가 아닌 컬럼이 있다고 가정하자.

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
   ....:                                   categories=['a', 'b'])

In [25]: data
Out[25]:
   x0    x1    y category
0   1  0.01 -1.5        a
1   2 -0.01  0.0        b
2   3  0.25  3.6        a
3   4 -4.10  1.3        a
4   5  0.00 -2.0        b
```

만일 `'category'` 컬럼을 더미값으로 치환하고 싶다면 더미값을 생성하고 `'category'` 컬럼을 삭제한 다음 결과와 합쳐야 한다.

```
In [26]: dummies = pd.get_dummies(data.category, prefix='category')

In [27]: data_with_dummies = data.drop('category', axis=1).join(dummies)

In [28]: data_with_dummies
Out[28]:
   x0    x1    y  category_a  category_b
0   1  0.01 -1.5           1           0
1   2 -0.01  0.0           0           1
```

```
2   3  0.25  3.6          1          0
3   4 -4.10  1.3          1          0
4   5  0.00 -2.0          0          1
```

특정 통계 모델을 더미값으로 피팅하는 기법도 있다. 단순히 숫자형 컬럼만 가지고 있는 게 아니라면 다음 절에서 살펴볼 Patsy를 사용하는 편이 더 단순하고 에러를 일으킬 가능성도 줄여준다.

13.2 Patsy를 이용해서 모델 생성하기

Patsy(팻시)는 통계 모델(특히 선형 모델)을 위한 파이썬 라이브러리이며 R이나 S 통계 프로그래밍 언어에서 사용하는 수식 문법과 비슷한 형식의 문자열 기반 '수식 문법'을 제공한다.

Patsy는 통계 모델에서 선형 모델을 잘 지원하므로 이해를 돕기 위해 주요 기능 중 일부만 살펴보도록 하자. Patsy의 수식 문법은 다음과 같은 특수한 형태의 문자열이다.

```
y ~ x0 + x1
```

`a + b` 문법은 a와 b를 더하라는 의미가 아니라 모델을 위해 생성된 **배열을 설계**하는 용법이다. `patsy.dmatrices` 함수는 수식 문자열과 데이터셋(DataFrame 또는 배열의 사전)을 함께 받아 선형 모델을 위한 설계 배열을 만들어낸다.

```
In [29]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [30]: data
Out[30]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0
```

```
In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

dmatrices 함수를 실행하면 다음과 같은 결과를 얻을 수 있다.

```
In [33]: y
Out[33]:
DesignMatrix with shape (5, 1)
     y
  -1.5
   0.0
   3.6
   1.3
  -2.0
  Terms:
    'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept  x0     x1
          1   1   0.01
          1   2  -0.01
          1   3   0.25
          1   4  -4.10
          1   5   0.00
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'x1' (column 2)
```

Patsy의 DesignMatrix 인스턴스는 몇 가지 추가 데이터가 포함된 NumPy ndarray로 볼 수 있다.

```
In [35]: np.asarray(y)
Out[35]:
array([[-1.5],
       [ 0. ],
       [ 3.6],
       [ 1.3],
       [-2. ]])
```

```
In [36]: np.asarray(X)
Out[36]:
array([[ 1.  ,  1.  ,  0.01],
       [ 1.  ,  2.  , -0.01],
       [ 1.  ,  3.  ,  0.25],
       [ 1.  ,  4.  , -4.1 ],
       [ 1.  ,  5.  ,  0.  ]])
```

여기서 Intercept는 최소자승회귀와 같은 선형 모델을 위한 표현이다. 모델에 0을 더해서 intercept(절편)를 제거할 수 있다.

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
  x0     x1
   1   0.01
   2  -0.01
   3   0.25
   4  -4.10
   5   0.00
  Terms:
    'x0' (column 0)
    'x1' (column 1)
```

Pasty 객체는 최소자승회귀분석을 위해 numpy.linalg.lstsq 같은 알고리즘에 바로 넘길 수도 있다.

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

모델 메타데이터는 design_info 속성을 통해 얻을 수 있는데 예를 들면 모델의 컬럼명을 피팅된 항에 맞추어 Series를 만들어낼 수도 있다.

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])
```

```
In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)

In [41]: coef
Out[41]:
Intercept    0.312910
x0          -0.079106
x1          -0.265464
dtype: float64
```

13.2.1 Patsy 용법으로 데이터 변환하기

파이썬 코드를 Patsy 용법과 섞어서 사용할 수도 있는데, Patsy 문법을 해석하는 과정에서 해당 함수를 찾아 실행해준다.

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)

In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
  Intercept  x0  np.log(np.abs(x1) + 1)
          1   1                 0.00995
          1   2                 0.00995
          1   3                 0.22314
          1   4                 1.62924
          1   5                 0.00000
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'np.log(np.abs(x1) + 1)' (column 2)
```

자주 쓰이는 변수 변환으로는 표준화(평균 0, 분산 1)와 센터링(평균값을 뺌)이 있는데 Pasty에는 이런 목적을 위한 내장 함수가 존재한다.

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)

In [45]: X
Out[45]:
```

```
DesignMatrix with shape (5, 3)
  Intercept  standardize(x0)  center(x1)
          1         -1.41421        0.78
          1         -0.70711        0.76
          1          0.00000        1.02
          1          0.70711       -3.33
          1          1.41421        0.77
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)
```

모델링 과정에서 모델을 어떤 데이터셋에 피팅하고 그다음에 다른 모델에 기반하여 평가해야 하는 경우가 있다. 이는 홀드-아웃hold-out[1]이거나 신규 데이터가 나중에 관측되는 경우다. 센터링이나 표준화 같은 변환을 적용하는 경우 새로운 데이터에 기반하여 예측하기 위한 용도로 모델을 사용한다면 주의해야 한다. 이를 상태를 가지는stateful 변환이라고 하는데 새로운 데이터셋을 변경하기 위해 원본 데이터의 표준편차나 평균 같은 통계를 사용해야 하기 때문이다.

patsy.build_design_matrices 함수는 입력으로 사용되는 원본 데이터셋에서 저장한 정보를 사용해서 출력 데이터를 만들어내는 변환에 적용할 수 있는 함수다.

```
In [46]: new_data = pd.DataFrame({
   ....:     'x0': [6, 7, 8, 9],
   ....:     'x1': [3.1, -0.5, 0, 2.3],
   ....:     'y': [1, 2, 3, 4]})

In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
   Intercept  standardize(x0)  center(x1)
           1          2.12132        3.87
           1          2.82843        0.27
           1          3.53553        0.77
           1          4.24264        3.07
```

1 옮긴이_ 전체 데이터셋을 학습을 위한 데이터셋과 검증을 위한 데이터셋으로 나누어 모델을 검증하는 방법

```
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)]
```

Patsy 문법에 더하기 기호(+)는 덧셈이 아니므로 데이터셋에서 이름으로 컬럼을 추가하고 싶다면 I라는 특수한 함수로 둘러싸야 한다.

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

In [50]: X
Out[50]:
DesignMatrix with shape (5, 2)
  Intercept  I(x0 + x1)
          1        1.01
          1        1.99
          1        3.25
          1       -0.10
          1        5.00
  Terms:
    'Intercept' (column 0)
    'I(x0 + x1)' (column 1)
```

Patsy는 patsy.builtins 모듈 내에 여러 가지 변환을 위한 내장 함수들을 제공한다. 자세한 내용은 온라인 문서를 참고하자.

범주형 데이터의 변환은 좀 특별하다. 다음 절에서 살펴보자.

13.2.2 범주형 데이터와 Patsy

비산술 데이터는 여러 가지 형태의 모델 설계 배열로 변환될 수 있다. 이 주제에 대한 본격적인 논의는 이 책의 영역을 벗어나므로 따로 통계와 관련한 내용과 함께 살펴보는 것이 좋은 듯하다.

Patsy에서 비산술 용법을 사용하면 기본적으로 더미 변수로 변환된다. 만약 intercept가 존재한다면 공선성을 피하기 위해 레벨 중 하나는 남겨두게 된다.

CHAPTER 14

데이터 분석 예제

드디어 마지막 장이다. 여기서는 실제 데이터셋을 살펴본다. 지금까지 책에서 배운 기술을 이용해서 데이터에서 의미 있는 정보를 추출해 보도록 하자. 여기서 설명하는 기술은 여러분의 데이터셋을 포함하여 모든 데이터셋에 적용할 수 있을 것이다. 이 장에는 이 책에서 배웠던 도구들을 실습해볼 수 있는 예제 데이터셋들을 모아두었다.

책에서 사용한 예제 데이터셋은 이 책의 깃허브 저장소에서 다운로드할 수 있다.

- **깃허브 저장소**
  http://github.com/wesm/pydata-book

14.1 Bit.ly의 1.USA.gov 데이터

2011년 URL 축약 서비스인 Bit.ly는 미국 정부 웹사이트인 USA.gov와 제휴하여 .gov나 .mil로 끝나는 URL을 축약한 사용자들에 대한 익명 정보를 제공했었다. 2011년에는 실시간 피드뿐 아니라 매 시간마다 스냅샷을 텍스트 파일로 내려받을 수 있었다. 이 책을 쓰는 현재 해당 서비스는 더 이상 존재하지 않지만 그 데이터 파일 중 하나를 살펴보자.

매 시간별 스냅샷 파일의 각 로우는 웹 데이터 형식으로 흔히 사용되는 JSON[JavaScript Object Notation]이다. 스냅샷 파일의 첫 줄을 열어보면 다음과 비슷한 내용을 확인할 수 있다.

```
In [5]: path = 'datasets/bitly_usagov/example.txt'

In [6]: open(path).readline()
Out[6]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1,
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u":
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

파이썬에는 JSON 문자열을 파이썬 사전 객체로 바꿔주는 다양한 내장 모듈과 서드파티 모듈이 있다. 여기서는 json 모듈의 loads 함수를 이용해서 내려받은 샘플 파일을 한 줄씩 읽는다.

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

결과를 담고 있는 records 객체는 파이썬 사전의 리스트다.

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
 'c': 'US',
 'cy': 'Danvers',
 'g': 'A6qOVH',
 'gr': 'MA',
 'h': 'wfLQtf',
 'hc': 1331822918,
 'hh': '1.usa.gov',
 'l': 'orofrog',
 'll': [42.576698, -70.954903],
 'nk': 1,
 'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
 't': 1331923247,
 'tz': 'America/New_York',
 'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

14.1.1 순수 파이썬으로 표준시간대 세어보기

이 데이터에서 가장 빈도가 높은 표준시간대(tz 필드)를 구한다고 가정하자. 다양한 방법이 있지만 먼저 리스트 표기법을 사용해서 표준시간대의 목록을 가져오자.

```
In [12]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-db4fbd348da9> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-db4fbd348da9> in <listcomp>(.0)
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

하지만 records의 아이템이 모두 표준시간대 필드가 가지고 있는 건 아니라는 게 드러났다! 이 문제는 `if 'tz' in rec`을 리스트 표기법 뒤에 추가해서 tz 필드가 있는지 검사하면 쉽게 해결할 수 있다.

```
In [13]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [14]: time_zones[:10]
Out[14]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

상위 10개의 표준시간대를 보면 그중 몇 개는 비어 있어서 뭔지 알 수 없다. 비어 있는 필드를 제거할 수도 있지만 일단은 그냥 두고 표준시간대를 세어보자.

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

파이썬 표준 라이브러리에 익숙하다면 다음처럼 좀 더 간단하게 작성할 수도 있다.

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # 값이 0으로 초기화된다.
    for x in sequence:
        counts[x] += 1
    return counts
```

재사용이 쉽도록 이 로직을 함수로 만들고 이 함수에 time_zones 리스트를 넘겨서 사용하자.

```
In [17]: counts = get_counts(time_zones)

In [18]: counts['America/New_York']
Out[18]: 1251

In [19]: len(time_zones)
Out[19]: 3440
```

가장 많이 등장하는 상위 10개의 표준시간대를 알고 싶다면 좀 더 세련된 방법으로 사전을 사용하면 된다.

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

이제 상위 10개의 표준시간대를 구했다.

```
In [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
 (36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
 (74, 'Europe/London'),
 (191, 'America/Denver'),
 (382, 'America/Los_Angeles'),
 (400, 'America/Chicago'),
 (521, ''),
 (1251, 'America/New_York')]
```

파이썬 표준 라이브러리의 collections.Counter 클래스를 이용하면 지금까지 했던 작업을 훨씬 쉽게 할 수 있다.

```
In [22]: from collections import Counter

In [23]: counts = Counter(time_zones)

In [24]: counts.most_common(10)
Out[24]:
[('America/New_York', 1251),
 ('', 521),
 ('America/Chicago', 400),
 ('America/Los_Angeles', 382),
 ('America/Denver', 191),
 ('Europe/London', 74),
 ('Asia/Tokyo', 37),
 ('Pacific/Honolulu', 36),
 ('Europe/Madrid', 35),
 ('America/Sao_Paulo', 33)]
```

14.1.2 pandas로 표준시간대 세어보기

records를 가지고 DataFrame을 만드는 방법은 아주 쉽다. 그냥 레코드가 담긴 리스트를 pnadas.DataFrame으로 넘기면 된다.

```
In [25]: import pandas as pd

In [26]: frame = pd.DataFrame(records)

In [27]: frame.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3560 entries, 0 to 3559
Data columns (total 18 columns):
_heartbeat_    120 non-null float64
a              3440 non-null object
al             3094 non-null object
c              2919 non-null object
cy             2919 non-null object
g              3440 non-null object
gr             2919 non-null object
h              3440 non-null object
hc             3440 non-null float64
hh             3440 non-null object
kw             93 non-null object
l              3440 non-null object
ll             2919 non-null object
nk             3440 non-null float64
r              3440 non-null object
t              3440 non-null float64
tz             3440 non-null object
u              3440 non-null object
dtypes: float64(4), object(14)
memory usage: 500.7+ KB

In [28]: frame['tz'][:10]
Out[28]:
0     America/New_York
1       America/Denver
2     America/New_York
3    America/Sao_Paulo
4     America/New_York
5     America/New_York
6        Europe/Warsaw
7
8
9
Name: tz, dtype: object
```

frame의 출력 결과는 거대한 DataFrame 객체의 **요약 정보**다. frame['tz']에서 반환되는 Series 객체에는 value_counts 메서드를 이용해서 시간대를 세어볼 수 있다.

```
In [29]: tz_counts = frame['tz'].value_counts()

In [30]: tz_counts[:10]
Out[30]:
America/New_York       1251
                        521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
America/Sao_Paulo        33
Name: tz, dtype: int64
```

matplotlib 라이브러리로 이 데이터를 그래프로 그릴 수 있다. 그전에 records에서 비어 있는 표준시간대를 다른 이름으로 바꿔보자. fillna 함수로 빠진 값을 대체하고, 불리언 배열 색인을 이용해서 비어 있는 값을 대체할 수 있다.

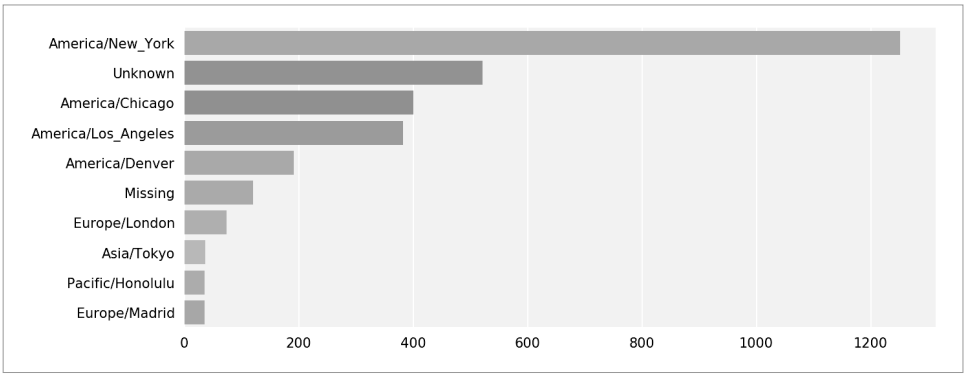
```
In [31]: clean_tz = frame['tz'].fillna('Missing')

In [32]: clean_tz[clean_tz == ''] = 'Unknown'

In [33]: tz_counts = clean_tz.value_counts()

In [34]: tz_counts[:10]
Out[34]:
America/New_York       1251
Unknown                 521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Missing                 120
Europe/London            74
Asia/Tokyo               37
```

```
Pacific/Honolulu         36
Europe/Madrid            35
Name: tz, dtype: int64
```

여기서는 seaborn 패키지를 이용해서 수평막대그래프를 그려보자(그림 14-1).

```
In [36]: import seaborn as sns

In [37]: subset = tz_counts[:10]

In [38]: sns.barplot(y=subset.index, x=subset.values)
```

그림 14-1 1.usa.gov 예제 데이터에서 가장 많이 나타난 시간대



a 필드에는 URL 단축을 실행하는 브라우저, 단말기, 애플리케이션에 대한 정보(User Agent 문자열)가 들어 있다.

```
In [39]: frame['a'][1]
Out[39]: 'GoogleMaps/RochesterNY'

In [40]: frame['a'][50]
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [41]: frame['a'][51][:50] # 긴 문자열
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

'agent'라고 하는 흥미로운 문자열 정보를 분석하는 일이 어려워 보일 수도 있다. 한 가지 가능한 전략은 문자열에서 첫 번째 토큰(브라우저의 종류를 어느 정도 알 수 있을 만큼)을 잘라내서 사용자 행동에 대한 또 다른 개요를 만드는 것이다.

```
In [42]: results = pd.Series([x.split()[0] for x in frame.a.dropna()])

In [43]: results[:5]
Out[43]:
0               Mozilla/5.0
1    GoogleMaps/RochesterNY
2               Mozilla/4.0
3               Mozilla/5.0
4               Mozilla/5.0
dtype: object

In [44]: results.value_counts()[:8]
Out[44]:
Mozilla/5.0                 2594
Mozilla/4.0                  601
GoogleMaps/RochesterNY       121
Opera/9.80                    34
TEST_INTERNET_AGENT           24
GoogleProducer                21
Mozilla/6.0                    5
BlackBerry8520/5.0.0.681       4
dtype: int64
```

이제 표준시간대 순위표를 윈도우 사용자와 비윈도우 사용자 그룹으로 나눠보자. 문제를 단순화해서 agent 문자열이 'Windows'를 포함하면 윈도우 사용자라고 가정하고 agent 값이 없는 데이터는 다음과 같이 제외한다.

```
In [45]: cframe = frame[frame.a.notnull()]
```

그리고 이제 각 로우가 윈도우인지 아닌지 검사한다.

```
In [47]: cframe['os'] = np.where(cframe['a'].str.contains('Windows'),
   ....:                         'Windows', 'Not Windows')
```

```
In [48]: cframe['os'][:5]
Out[48]:
0        Windows
1    Not Windows
2        Windows
3    Not Windows
4        Windows
Name: os, dtype: object
```

그런 다음 표준시간대와 운영체제를 기준으로 데이터를 그룹으로 묶는다.

```
In [49]: by_tz_os = cframe.groupby(['tz', 'os'])
```

앞에서 살펴본 value_count 함수처럼 그룹별 합계는 size 함수로 계산할 수 있다. 결과는 unstack 함수를 이용해서 표로 재배치한다.

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [51]: agg_counts[:10]
Out[51]:
os                               Not Windows  Windows
tz
                                       245.0    276.0
Africa/Cairo                             0.0      3.0
Africa/Casablanca                        0.0      1.0
Africa/Ceuta                             0.0      2.0
Africa/Johannesburg                      0.0      1.0
Africa/Lusaka                            0.0      1.0
America/Anchorage                        4.0      1.0
America/Argentina/Buenos_Aires           1.0      0.0
America/Argentina/Cordoba                0.0      1.0
America/Argentina/Mendoza                0.0      1.0
```

마지막으로 전체 표준시간대의 순위를 모아보자. 먼저 agg_counts를 보자.

```
# 오름차순으로 정렬
In [52]: indexer = agg_counts.sum(1).argsort()

In [53]: indexer[:10]
```

```
Out[53]:
tz
                                 24
Africa/Cairo                     20
Africa/Casablanca                21
Africa/Ceuta                     92
Africa/Johannesburg              87
Africa/Lusaka                    53
America/Anchorage                54
America/Argentina/Buenos_Aires   57
America/Argentina/Cordoba        26
America/Argentina/Mendoza        55
dtype: int64
```

agg_counts에 take를 사용해서 로우를 정렬된 순서 그대로 선택하고 마지막 10개 로우(가장 큰 값)만 잘라낸다.

```
In [54]: count_subset = agg_counts.take(indexer[-10:])

In [55]: count_subset
Out[55]:
os                  Not Windows  Windows
tz
America/Sao_Paulo           13.0     20.0
Europe/Madrid               16.0     19.0
Pacific/Honolulu             0.0     36.0
Asia/Tokyo                   2.0     35.0
Europe/London               43.0     31.0
America/Denver             132.0     59.0
America/Los_Angeles        130.0    252.0
America/Chicago            115.0    285.0
                           245.0    276.0
America/New_York           339.0    912.0
```

pandas에는 이와 똑같은 동작을 하는 nlargest라는 편리한 메서드가 존재한다.

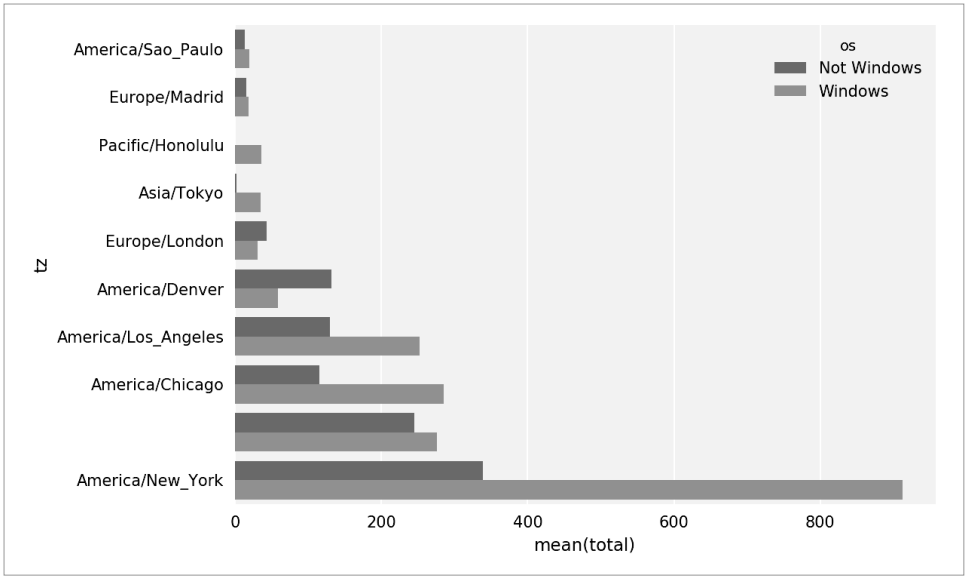
```
In [56]: agg_counts.sum(1).nlargest(10)
Out[56]:
tz
America/New_York       1251.0
                        521.0
```

```
America/Chicago         400.0
America/Los_Angeles     382.0
America/Denver          191.0
Europe/London            74.0
Asia/Tokyo               37.0
Pacific/Honolulu         36.0
Europe/Madrid            35.0
America/Sao_Paulo        33.0
dtype: float64
```

그런 다음 앞에서 해본 것처럼 plot 함수에 stacked=True를 넘겨주면 데이터를 중첩막대그래프로 만들 수 있다(그림 14-2).

```
# 시각화를 위해 데이터 재배치
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
                 tz           os  total
0  America/Sao_Paulo  Not Windows   13.0
1  America/Sao_Paulo      Windows   20.0
2      Europe/Madrid  Not Windows   16.0
3      Europe/Madrid      Windows   19.0
4   Pacific/Honolulu  Not Windows    0.0
5   Pacific/Honolulu      Windows   36.0
6         Asia/Tokyo  Not Windows    2.0
7         Asia/Tokyo      Windows   35.0
8      Europe/London  Not Windows   43.0
9      Europe/London      Windows   31.0

In [62]: sns.barplot(x='total', y='tz', hue='os', data=count_subset)
```

그림 14-2 윈도우 사용자와 비윈도우 사용자별 시간대



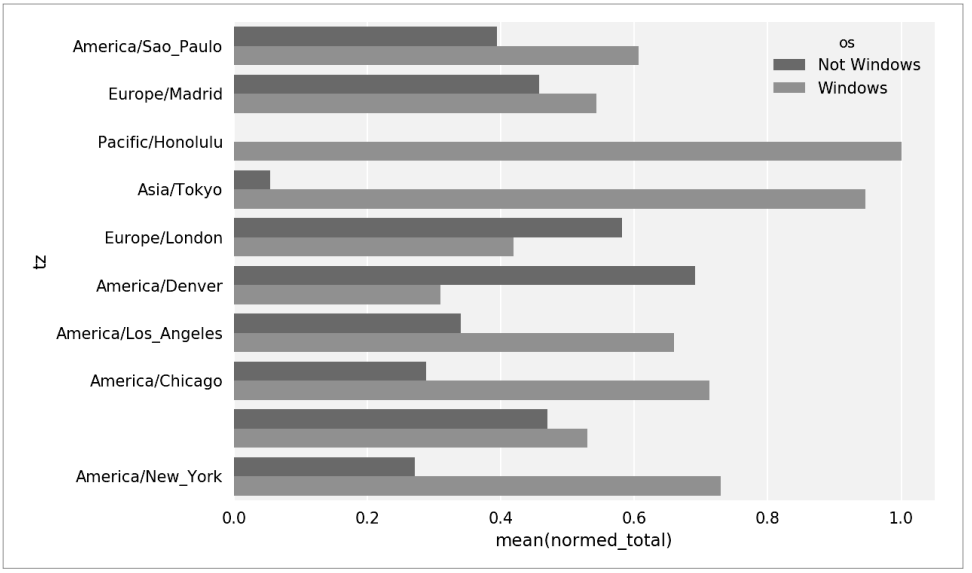
위 그래프로는 작은 그룹에서 윈도우 사용자의 상대 비율을 확인하기 어렵다. 하지만 각 로우에서 총합을 1로 정규화한 뒤 그래프를 그리면 쉽게 확인할 수 있다.

```
def norm_total(group):
    group['normed_total'] = group.total / group.total.sum()
    return group

results = count_subset.groupby('tz').apply(norm_total)
```

정규화한 데이터를 그래프로 그려보자(그림 14-3).

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os', data=results)
```

그림 14-3 윈도우 사용자와 비윈도우 사용자별 시간대 비율



groupby와 transform 메서드를 이용해서 정규합 계산을 더 효율적으로 할 수도 있다.

```
In [66]: g = count_subset.groupby('tz')

In [67]: results2 = count_subset.total / g.total.transform('sum')
```

14.2 MovieLens의 영화 평점 데이터

GroupLens 연구소는 1990년대 말부터 2000년대 초까지 MovieLens 사용자로부터 수집한 방대한 영화 평점 데이터를 제공하고 있다. 이 데이터에는 영화 평점과 영화에 대한 정보(장르, 개봉 년도) 그리고 사용자에 대한 정보(나이, 우편번호, 성별, 직업)가 포함되어 있다. 이런 종류의 데이터는 머신러닝 알고리즘 기반의 추천 시스템을 개발하는 데 주로 활용한다. 머신러닝 기법을 여기서 소개하기는 어렵고 이런 종류의 데이터를 요구 사항에 맞도록 잘 쪼개는 방법만 소개하겠다.

MovieLens 1M(백만 개) 데이터셋은 약 6,000여 명의 사용자로부터 수집한 4,000여 편의

영화에 대한 백만 개의 영화 평점을 담고 있다. 이 데이터셋은 평점, 사용자 정보, 영화 정보의 3가지 테이블로 나뉘어 있는데, zip 파일의 압축을 풀고 각 테이블을 pandas.read_table 함수를 사용하여 DataFrame 객체로 불러오자.

```
import pandas as pd

# 출력되는 내용을 줄인다.
pd.options.display.max_rows = 10

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('datasets/movielens/users.dat', sep='::',
                      header=None, names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('datasets/movielens/ratings.dat', sep='::',
                        header=None, names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
                       header=None, names=mnames)
```

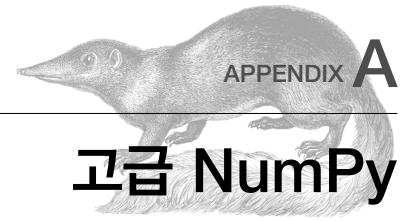
DataFrame 객체에 데이터가 제대로 들어갔는지 확인하기 위해 파이썬의 리스트 분할 문법을 사용해서 첫 5개 로우를 출력해보자.

```
In [69]: users[:5]
Out[69]:
   user_id gender  age  occupation     zip
0        1      F    1          10  48067
1        2      M   56          16  70072
2        3      M   25          15  55117
3        4      M   45           7  02460
4        5      M   25          20  55455

In [70]: ratings[:5]
Out[70]:
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
3        1      3408       4  978300275
4        1      2355       5  978824291
```

```
In [71]: movies[:5]
Out[71]:
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy

In [72]: ratings
Out[72]:
         user_id  movie_id  rating  timestamp
0              1    1193         5  978300760
1              1     661         3  978302109
2              1     914         3  978301968
3              1    3408         4  978300275
4              1    2355         5  978824291
...          ...     ...       ...        ...
1000204     6040    1091         1  956716541
1000205     6040    1094         5  956704887
1000206     6040     562         5  956704746
1000207     6040    1096         4  956715648
1000208     6040    1097         4  956715569
[1000209 rows x 4 columns]
```

나이와 직업은 실제 값이 아니라 그룹을 가리키는 코드 번호이며 데이터셋에 있는 README 파일에 해당 코드에 대한 설명이 들어 있다. 세 종류의 테이블에 걸쳐 있는 데이터를 분석하는 일은 단순한 작업이 아니다. 나이와 성별에 따른 어떤 영화의 평균 평점을 계산한다고 해보자. 아래에서 확인할 수 있지만, 모든 데이터를 하나의 테이블로 병합하여 계산하면 무척 쉽게 처리할 수 있다. pandas의 `merge` 함수를 이용해서 `ratings` 테이블과 `users` 테이블을 병합하고 그 결과를 다시 `movies` 테이블과 병합한다. pandas는 병합하려는 두 테이블에서 중복되는 컬럼의 이름을 키로 사용한다.

```
In [73]: data = pd.merge(pd.merge(ratings, users), movies)

In [74]: data
Out[74]:
         user_id  movie_id  rating  timestamp gender  age  occupation    zip  \
0              1      1193       5  978300760      F    1          10  48067
```



APPENDIX A

고급 NumPy

부록 A에서는 배열 계산을 위한 NumPy 라이브러리를 좀 더 자세히 살펴보도록 하자. ndarray 자료형의 내부 구조를 상세히 알아보고 고급 배열 조작 기법과 알고리즘을 살펴본다.

여기서는 여러 가지 주제를 다루고 있으며 꼭 순서대로 읽어야 할 필요는 없다.

A.1 ndarray 객체 구조

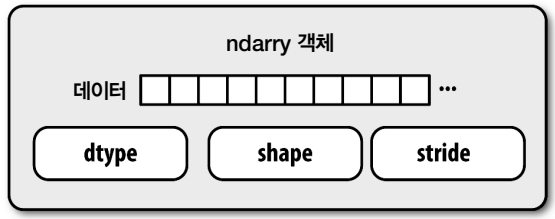
NumPy의 ndarray는 연속적이든 아니든 단일 형태의 데이터 블록을 다차원 배열 객체 형태로 해석할 수 있는 수단을 제공한다. dtype이라고 하는 자료형은 데이터가 실수, 정수, 불리언 혹은 다른 형인지 알려주는 역할을 한다.

ndarray가 유연한 까닭은 모든 배열 객체가 띄엄띄엄 떨어진 데이터 블록에 대한 뷰이기 때문이다. 예를 들어 `arr[::2, ::-1]` 배열은 어째서 데이터 복사가 일어나지 않는 것인지 의아할 수 있다. 그 이유는 ndarray는 단순한 메모리 덩어리와 dtype만을 가지는 것이 아니기 때문이다. ndarray에는 다양한 너비로 메모리 사이를 건너뛸 수 있는 보폭striding 정보를 포함하고 있다. 좀 더 설명하자면, ndarray는 내부적으로 다음과 같이 구성되어 있으며 [그림 A-1]에 간략한 ndarray의 내부 구조도를 나타냈다.

- **데이터 포인터** : RAM이나 메모리 맵 파일에서 데이터의 블록
- **dtype**은 배열 내에서 값을 담는 고정된 크기를 나타낸다.

- 배열의 **모양**(shape)을 알려주는 튜플
- 하나의 차원을 따라 다음 원소로 몇 바이트 이동해야 하는지를 나타내는 **stride**를 담고 있는 튜플

그림 A-1 NumPy ndarray 객체



예를 들어 10×5 크기의 배열의 모양은 `(10, 5)`로 표현된다.

```
In [10]: np.ones((10, 5)).shape
Out[10]: (10, 5)
```

C언어 형식의 3×4×5 크기의 `float64`(8바이트) 배열은 `(160, 40, 8)`의 stride 값을 가진다. stride 정보를 알고 있으면 편리한데, 일반적으로 stride 값이 클수록 해당 축을 따라 연산을 수행하는 비용이 많이 들기 때문이다.

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[11]: (160, 40, 8)
```

일반적인 NumPy 사용자들은 배열의 stride 값에 흥미를 가지는 경우가 드물지만, stride 값은 복사가 이루어지지 않는 배열의 뷰를 생성하는 데 중요한 역할을 한다. stride 값은 음수일 수도 있는데 이는 메모리상에서 뒤로 이동해야 한다는 의미다. 배열을 `obj[::-1]` 또는 `obj[:, ::-1]` 형태로 잘라내는 경우가 그렇다.

A.1.1 NumPy dtype 구조

종종 배열에 담긴 값이 정수, 실수, 문자열 혹은 파이썬 객체인지 확인하는 코드를 작성할 경우가 있다. 실수에도 다양한 형태(`float16`부터 `float128`까지)가 있고, 리스트를 따라 dtype을

확인하는 과정은 꽤나 번거롭기 때문이다. 다행히도 dtype은 np.issubdtype 함수와 결합하여 사용할 수 있는 np.integer나 np.floating 같은 부모 클래스를 가진다.

```
In [12]: ints = np.ones(10, dtype=np.uint16)

In [13]: floats = np.ones(10, dtype=np.float32)

In [14]: np.issubdtype(ints.dtype, np.integer)
Out[14]: True

In [15]: np.issubdtype(floats.dtype, np.floating)
Out[15]: True
```

특정 dtype의 모든 부모 클래스는 mro 메서드를 이용해서 확인할 수 있다.

```
In [16]: np.float64.mro()
Out[16]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

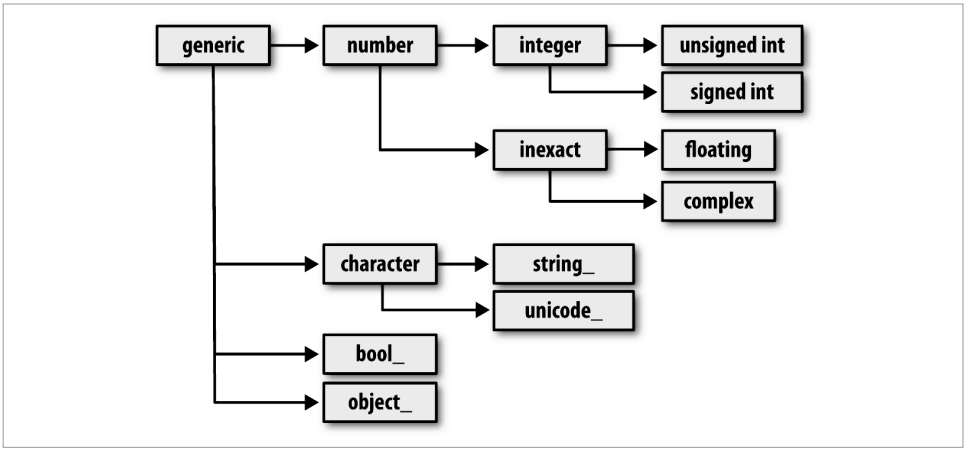
따라서 아래와 같이 ints가 np.number 형임을 확인할 수 있다.

```
In [17]: np.issubdtype(ints.dtype, np.number)
Out[17]: True
```

대부분의 NumPy 사용자들은 이런 내용을 알 필요가 없겠지만 가끔 도움이 되는 경우도 있다. [그림 A-2]에 dtype의 부모와 자식클래스 관계를 나타냈다.[1]

1 몇몇 dtype은 이름이 _로 끝나기도 하는데 이는 NumPy에서 사용하는 자료형과 파이썬 내장 자료형 간의 충돌을 피하기 위해서다.

그림 A-2 NumPy dtype 클래스 계층



A.2 고급 배열 조작 기법

배열을 세련된 방법으로 색인하고, 나누고, 불리언으로 값의 일부를 취하는 다양한 방법이 존재한다. 데이터 분석 애플리케이션에서 까다로운 대부분의 작업은 pandas의 상위레벨 함수에서 처리하지만 라이브러리에 존재하지 않는 데이터 알고리즘을 직접 작성해야 하는 경우도 있다.

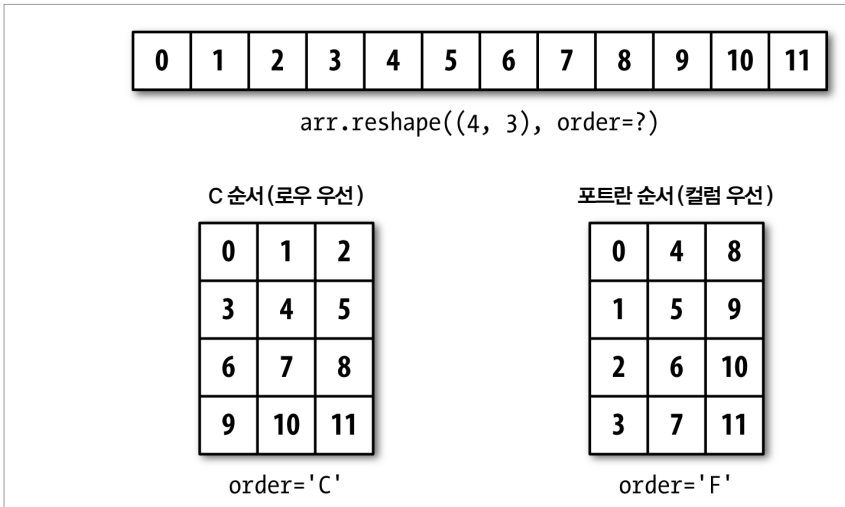
A.2.1 배열 재형성하기

NumPy 배열에 대해 지금까지 배운 내용으로 배열의 데이터를 복사하지 않고 다른 모양으로 변환할 수 있다는 것은 약간 놀라운 점이다. 배열의 모양을 변환하려면 배열의 인스턴스 메서드인 reshape 메서드에 새로운 모양을 나타내는 튜플을 넘기면 된다. 예를 들어 1차원 배열을 행렬로 바꾸려 한다고 가정해보자(결과는 [그림 A-3] 참조).

```
In [18]: arr = np.arange(8)

In [19]: arr
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [20]: arr.reshape((4, 2))
```

```
Out[20]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

그림 A-3 로우 우선, 컬럼 우선 재형성하기



다차원 배열 또한 재형성이 가능하다.

```
In [21]: arr.reshape((4, 2)).reshape((2, 4))
Out[21]:array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
```

reshape에 넘기는 값 중 하나가 −1이 될 수도 있는데 이 경우에는 원본 데이터를 참조해서 적절한 값을 추론하게 된다.

```
In [22]: arr = np.arange(15)

In [23]: arr.reshape((5, -1))
Out[23]:
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

배열의 shape 속성은 튜플이기 때문에 reshape 메서드에 이를 직접 넘기는 것도 가능하다.

```
In [24]: other_arr = np.ones((3, 5))

In [25]: other_arr.shape
Out[25]: (3, 5)

In [26]: arr.reshape(other_arr.shape)
Out[26]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

다차원 배열을 낮은 차원으로 변환하는 것은 **평탄화**flattening, raveling라고 한다.

```
In [27]: arr = np.arange(15).reshape((5, 3))

In [28]: arr
Out[28]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [29]: arr.ravel()
Out[29]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

ravel 메서드는 필요하지 않다면 원본 데이터의 복사본을 생성하지 않는다. flatten 메서드는 ravel 메서드와 유사하게 동작하지만 항상 데이터의 복사본을 반환한다.

```
In [30]: arr.flatten()
Out[30]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

APPENDIX B

IPython 시스템 더 알아보기

2장에서는 IPython 셸과 주피터 노트북의 기본 사용법을 살펴봤다. 이 장에서는 주피터나 콘솔에서 사용할 수 있는 IPython 시스템의 상세 기능을 좀 더 살펴보도록 하겠다.

B.1 명령어 히스토리 사용하기

IPython은 이전에 실행했던 명령어를 디스크에 작은 데이터베이스 형식으로 보관하며, 다음 목적으로 사용한다.

- 검색, 자동완성, 최소한의 입력으로 이전에 실행했던 명령 재실행
- 세션 간의 명령어 히스토리 유지
- 입출력 히스토리를 파일에 기록

이 기능은 노트북보다는 셸에서 더 유용한데, 노트북은 각각의 코드 셀에서 입력과 출력을 기록하도록 설계되었기 때문이다.

B.1.1 명령어 검색과 재사용

IPython 셸에서는 이전에 실행했던 코드나 명령을 검색하고 실행할 수 있다. 이 기능은 예를 들어 `%run` 명령어나 코드를 반복 사용하는 경우가 잦을 경우 무척 유용하다. 다음과 같이 실행했다고 하자.

```
In[7]: %run first/second/third/data_script.py
```

그리고 스크립트의 실행 결과를 보니 틀린 계산이 있었다고 하자. 문제의 원인을 밝혀낸 다음 data_script.py를 수정하고 %run 명령어의 첫 일부를 입력한 다음 Ctrl-P를 누르거나 위 화살표 키를 누르면 방금 입력한 명령어와 맞아떨어지는 가장 최근에 입력한 명령어 히스토리를 검색한다. Ctrl-P나 위 화살표 키를 여러 번 누르면 계속해서 더 이전의 히스토리를 검색한다. 만일 실행하려는 명령어를 지나쳤다면 Ctrl-N이나 아래 화살표 키를 눌러 다시 돌아올 수 있다. 이 기능을 몇 번 사용해보면 어느 순간 의도하지 않고도 이 기능을 사용하는 자신을 발견하게 될 것이다!

Ctrl-R을 사용하면 bash 같은 유닉스 셸의 readline 명령어처럼 부분 순차 검색을 할 수 있다. 윈도우에서는 IPython이 readline 기능을 흉내 내도록 한다. Ctrl-R을 누르고 입력 줄에 검색하려는 몇 글자를 입력하면 이 기능을 사용할 수 있다.

```
In [1]: a_command = foo(x, y, z)

(reverse-i-search)`com': a_command = foo(x, y, z)
```

Ctrl-R을 누르면 방금 입력한 글자와 맞아떨어지는 줄을 계속해서 반복적으로 찾아준다.

B.1.2 입출력 변수

함수의 실행 결과를 변수에 저장하는 것을 잊으면 매우 귀찮아진다. 다행히 IPython은 입력한 명령과 출력된 결과물인 반환된 객체를 특수한 변수에 저장한다. 마지막 2개의 결과를 각각 _ 변수와 __ 변수에 저장한다.

```
In [24]: 2 ** 27
Out[24]: 134217728

In [25]: _
Out[25]: 134217728
```

입력 변수는 _iX 변수(X는 입력 줄 번호)에 저장된다. 이와 유사하게 출력 변수는 _X 변수에 저장된다. 그래서 27번 줄을 입력한 후에는 출력 결과를 저장하는 출력 변수 _27과 입력 변수 _i27이 생겨난다.

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

입력 변수는 문자열이기 때문에 다음처럼 파이썬의 exec 예약어를 사용해서 실행한다.

```
In [30]: exec(_i27)
```

여기서 _i27은 In[27]에 입력한 코드를 나타낸다.

몇 가지 매직함수를 사용하면 입출력 히스토리를 사용한 작업을 할 수 있다. %hist는 전체 혹은 일부 입력 히스토리를 줄 번호와 함께 혹은 줄 번호 없이 출력한다. %reset은 인터랙티브 네임스페이스와 추가적으로 입출력 캐시를 비운다. %xdel은 IPython 내에서 **특정** 객체에 대한 참조를 삭제하여 그 객체가 사용한 메모리를 해제하는 함수다. 이 매직함수에 대한 자세한 내용은 문서를 참고하자.

CAUTION_ 매우 큰 데이터를 이용한 작업을 할 때는 del 예약어를 사용해서 인터랙티브 네임스페이스에서 어떤 변수를 지웠다고 해도 IPython의 입출력 히스토리가 가비지 컬렉트(메모리 정리)를 방해할 수 있다는 점을 기억하자. 그런 경우에는 %xdel과 %reset을 신중하게 사용해서 메모리 문제를 회피할 수 있다.

B.2 운영체제와 함께 사용하기

IPython의 또 다른 중요한 기능은 운영체제 셸과 강력하게 통합되어 있다는 것이다. 즉, IPython을 종료하지 않고도 윈도우나 유닉스(리눅스, macOS) 셸에서 일반적인 명령행에서 할 수 있는 작업이 가능하다는 뜻이다. 여기에는 셸 명령어를 실행하거나, 디렉터리를 옮기거나, 명령어의 결과를 파이썬 객체(리스트나 문자열)에 저장하는 기능이 포함된다. 또한 간단한 셸 명령어 앨리어싱과 디렉터리 북마크 기능도 제공한다.

[표 B-1]에 매직함수를 요약해두었다. 기능은 잠시 후에 간단히 살펴보겠다.

표 B-1 IPython의 시스템 관련 명령어

명령	설명
`!cmd`	시스템 셸에서 `cmd` 명령어를 실행한다.
`output = !cmd args`	`cmd` 명령어를 실행하고 표준 출력(stdout) 결과를 `output`에 저장한다.
`%alias` *`alias_name cmd`*	시스템(셸) 명령어의 별칭을 정의한다.
`%bookmark`	IPython의 디렉터리 북마크 시스템 활용한다.
`%cd` *`directory`*	시스템의 작업 디렉터리를 *`directory`*로 변경한다.
`%pwd`	현재 시스템의 작업 디렉터리를 반환한다.
`%pushd` *`directory`*	현재 디렉터리를 스택에 추가하고 새로운 디렉터리로 이동한다.
`%popd`	스택에 저장된 디렉터리를 꺼내서 그 디렉터리로 이동한다.
`%dirs`	현재 디렉터리 스택에 저장된 디렉터리 목록을 보여준다.
`%dhist`	접근했던 디렉터리 히스토리를 출력한다.
`%env`	시스템 환경 변수를 사전 타입으로 출력한다.
`%matplotlib`	matplotlib 통합 옵션을 설정한다.

B.2.1 셸 명령어와 별칭

IPython에서 `!`로 시작하는 줄은 느낌표 다음에 있는 내용을 시스템 셸에서 실행하라는 의미다. 이 말은 `rm`이나 `del` 명령어를 사용해서 파일을 지우거나, 디렉터리를 옮기거나, 다른 프로세스를 실행할 수 있다는 것이다.

셸 명령어의 콘솔 출력은 `!`를 이용해서 변수에 저장할 수 있다. 예를 들어 인터넷에 연결되어 있는 자신의 리눅스에서 IP 주소를 얻어서 파이썬 변수에 대입할 수도 있다

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11  Bcast:10.0.0.255  Mask:255.255.255.0'
```

반환된 파이썬 객체 ip_info는 콘솔 출력의 다양한 버전을 포함하고 있는 사용자 정의 리스트 타입이다.

IPython은 !를 사용해서 현재 환경에 정의되어 있는 파이썬 값을 대체하기도 한다. 이렇게 하려면 변수 이름 앞에 달러 기호 $를 붙이면 된다.

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py  test.py  test.xml
```

%alias 매직함수는 셸 명령어에 대한 사용자 단축키(별칭)를 정의한다. 다음은 간단한 예제다.

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x   2 root root  69632 2012-01-29 20:36 bin/
drwxr-xr-x   2 root root   4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root  20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x  44 root root  69632 2011-12-26 18:08 lib32/
lrwxrwxrwx   1 root root      3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x  15 root root   4096 2011-10-13 19:03 local/
drwxr-xr-x   2 root root  12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root  12288 2011-11-04 22:53 share/
drwxrwsr-x  24 root src    4096 2011-07-17 18:38 src/
```

여러 개의 명령어도 세미콜론으로 구분하여 한 번에 실행할 수 있다.

```
In [558]: %alias test_alias (cd examples ls cd ..)

In [559]: test_alias
macrodata.csv  spx.csv  tips.csv
```

IPython은 세션이 종료되면 즉시 정의해놓은 별칭을 잊어버린다. 고정 별칭을 만들고 싶다면 설정 시스템을 사용해야 한다.

B.2.2 디렉터리 북마크 시스템

IPython은 간단한 디렉터리 북마크 시스템을 가지고 있어서 자주 사용하는 디렉터리에 대한 별칭을 저장해두고 쉽게 이동할 수 있도록 한다. 예를 들어 이 책의 코드 예제를 모아둔 디렉터리로 쉽게 이동하기 위해 북마크를 정의할 수 있다.

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

북마크를 해두면 `%cd` 매직을 이용할 때 사용할 수 있다.

```
In [7]: cd py4da
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

만약에 북마크 이름이 현재 작업 디렉터리에 있는 이름과 충돌한다면 `-b` 플래그를 사용해서 오버라이드하고 북마크 위치를 사용한다. `%bookmak`에 `-l` 옵션을 주면 모든 북마크를 보여준다.

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

별칭과 다르게 북마크는 IPython이 종료되어도 유지된다.

B.3 소프트웨어 개발 도구

지금까지 살펴본 데이터 조회와 계산에 편리한 대화형 환경에 덧붙여서 IPython은 소프트웨어 개발 환경으로도 손색이 없다. 데이터 분석 애플리케이션에서는 **올바른** 코드를 작성하는 일이 가장 중요하다. 다행히도 IPython은 향상된 파이썬 `pdb` 디버거를 내장하고 있다. 또한 코

드 실행이 빨라야 하는데 IPython은 쉽게 사용할 수 있는 코드 타이밍과 프로파일링 도구를 포함하고 있다.

B.3.1 대화형 디버거

IPython의 디버거는 pdb에 탭 자동완성 기능, 문법 강조, 예외 트레이스백에서 각 줄에 해당하는 컨텍스트 등이 개선되었다. 코드를 디버깅하기 가장 최적인 시점은 에러가 발생한 직후다. 예외가 발생한 뒤 %debug 명령어를 사용하면 사후처리post-mortem 디버거가 실행되고 예외가 발생한 시점의 스택 프레임 정보를 보여준다.

```
In [2]: run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb>
```

디버거 안에서는 아무 파이썬 코드나 실행해볼 수 있고 각각의 스택 프레임에서 인터프리터 안에서 유지되고 있는 모든 객체와 데이터를 살펴볼 수 있다. 디폴트로 에러가 발생한 가장 아래 레벨에서부터 시작한다. `u`(up)와 `d`(down)를 눌러 스택 트레이스 사이를 이동할 수 있다.

```
ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
     12     works_fine()
---> 13     throws_an_exception()
     14
```

`%pdb` 명령어를 실행하면 예외가 발생했을 때 IPython이 자동적으로 디버거를 실행하는데 이 모드는 많은 사용자가 아주 유용하다고 생각할 것이다.

디버거는 개발하는 중에 스크립트나 함수를 실행하는 과정에서 각 단계를 하나씩 검증하거나 브레이크포인트를 설정하고 싶을 때 쉽게 사용할 수 있다. 디버거를 실행하는 몇 가지 방법이 있는데 `%run` 명령에 `-d` 옵션을 주면 스크립트를 실행하기 전에 디버거를 먼저 실행한다. 그리고 바로 `s`(step)를 누르면 스크립트로 진입한다.

```
In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

이제부터 스크립트 파일을 어떤 식으로 동작하게 할지는 독자의 몫이다. 예를 들어 위 예제에서 `works_fine` 메서드를 호출하기 바로 직전에 브레이크포인트를 걸고 `c`(continue)를 눌러 브레이크포인트에서 멈출 때까지 스크립트를 실행할 수 있다.

```
ipdb> b 12
ipdb> c
```

```
> /home/wesm/code/pydata-book/examples/ipython_bug.py(12)calling_things()
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

이제 s를 눌러 works_fine() 안으로 진입하거나, n(next)을 눌러 works_fine()을 실행하고 다음 줄로 진행할 수 있다.

```
ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
2    12     works_fine()
---> 13     throws_an_exception()
     14
```

이제 throws_an_exception 안으로 진입하고 에러가 발생한 다음 줄로 진행한 후 해당 범위 내에 있는 변수를 살펴보자. 디버거 명령어는 변수 이름보다 우선되므로 디버거 명령과 같은 이름의 변수가 있다면 !를 변수 이름 앞에 붙여서 내용을 확인할 수 있다.

```
ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(6)throws_an_exception()
      5
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(7)throws_an_exception()
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(8)throws_an_exception()
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
```

```
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb> !a
5

ipdb> !b
6
```

대화형 디버거는 많은 연습과 경험을 통해서만 익숙해질 수 있다. [표 B-2]에 디버거에서 사용할 수 있는 명령어를 모두 나열해두었다. IDE를 사용 중이라면 처음엔 터미널 기반의 디버거가 익숙하지 않겠지만 차츰 익숙해질 것이다. 대부분의 파이썬 IDE는 훌륭한 GUI 디버거를 내장하고 있으므로 자신에게 맞는 도구를 찾을 수 있을 것이다.

표 B-2 (I)Python 디버거 명령

명령	실행
h(elp)	명령어 목록을 보여준다.
help *command*	*command*에 대한 문서를 보여준다.
c(ontinue)	프로그램의 실행을 재개한다.
q(uit)	더 이상 코드를 실행하지 않고 디버거를 종료한다.
b(reak) *number*	현재 파일의 *number*번째 줄에 브레이크포인트를 설정한다.
b *path/to/file.py:number*	지정한 파일의 *number*번째 줄에 브레이크포인트를 설정한다.
s(tep)	함수 호출 안으로 진입한다.
n(ext)	현재 줄을 실행하고 같은 레벨의 다음 줄로 진행한다.
u(p) / d(own)	함수 콜 스택(호출 정보)의 위아래로 이동한다.
a(rgs)	현재 함수의 인자를 보여준다.
debug *statement*	*statement* 문장을 새로운 (재귀적) 디버거에서 실행한다.
l(ist)	현재 위치와 스택의 현재 레벨에 대한 문맥을 보여준다.
w(here)	현재 위치에 대한 문맥과 함께 전체 스택 정보를 출력한다.

디버거를 사용하는 다른 방법

디버거를 실행하는 몇 가지 다른 유용한 방법이 있다. set_trace 함수(pdb.set_trace에서 유래한 이름)를 이용하는 것이 그중 하나인데, '가난뱅이의 브레이크포인트'라고 불리는 것이다. 아래에 짧은 예제가 있는데 이 코드를 범용적으로 사용하기 위해 나는 IPython 프로파일에 추가해서 사용하고 있다.

빅데이터 분석에 관한 가장 완벽한 교재

Python for Data Analysis

이 책은 NumPy, pandas, matplotlib, IPython, Jupyter 등 다양한 파이썬 라이브러리를 사용해서 효과적으로 데이터를 분석할 수 있게 알려준다. pandas의 새로운 기능뿐만 아니라 메모리 사용량을 줄이고 성능을 개선하는 고급 사용법까지 다룬다. 또한 모델링 도구인 statsmodels와 scikit-learn 라이브러리도 소개한다. 연대별 이름 통계 자료, 미 대선 데이터베이스 자료 등 실사례로 따라 하다 보면 어느덧 여러분도 데이터에 알맞게 접근하고 효과적으로 분석하는 전문가가 될 것이다.

- IPython 셸, 주피터 노트북 사용하기
- NumPy 기본 및 고급 기능 알아보기
- pandas로 데이터 분석 시작하기
- 유연한 도구를 사용해 데이터 로딩, 정제, 조인, 병합, 변형하기
- matplotlib으로 유용한 시각화 만들기
- pandas groupby 기능을 적용해 데이터를 나누고 요약하기
- 시계열 데이터 분석 및 조작하기

"이미 필독서가 된 이 책이 업그레이드되었다. 2판에는 파이썬 3.6부터 pandas 최신 기능에 이르기까지 이 책의 가치를 더 향상시킬 내용이 담겼다. 왜 파이썬 라이브러리인지, 이 도구들을 어떻게 다뤄야 하는지 설명해 독자가 새롭고 창의적인 방식으로 효율적인 사용법을 익히도록 도와준다."

페르난도 페레즈 Fernando Pérez, IPython 창시자, UC 버클리 통계학과 조교수

예제 소스 http://github.com/wesm/pydata-book

관련 도서

파이썬 데이터 분석 입문	처음 배우는 데이터 과학	처음 배우는 머신러닝	우아한 사이파이

인공지능/데이터 분석

9 791162 241905 93000

ISBN 979-11-6224-190-5

정가 35,000원