

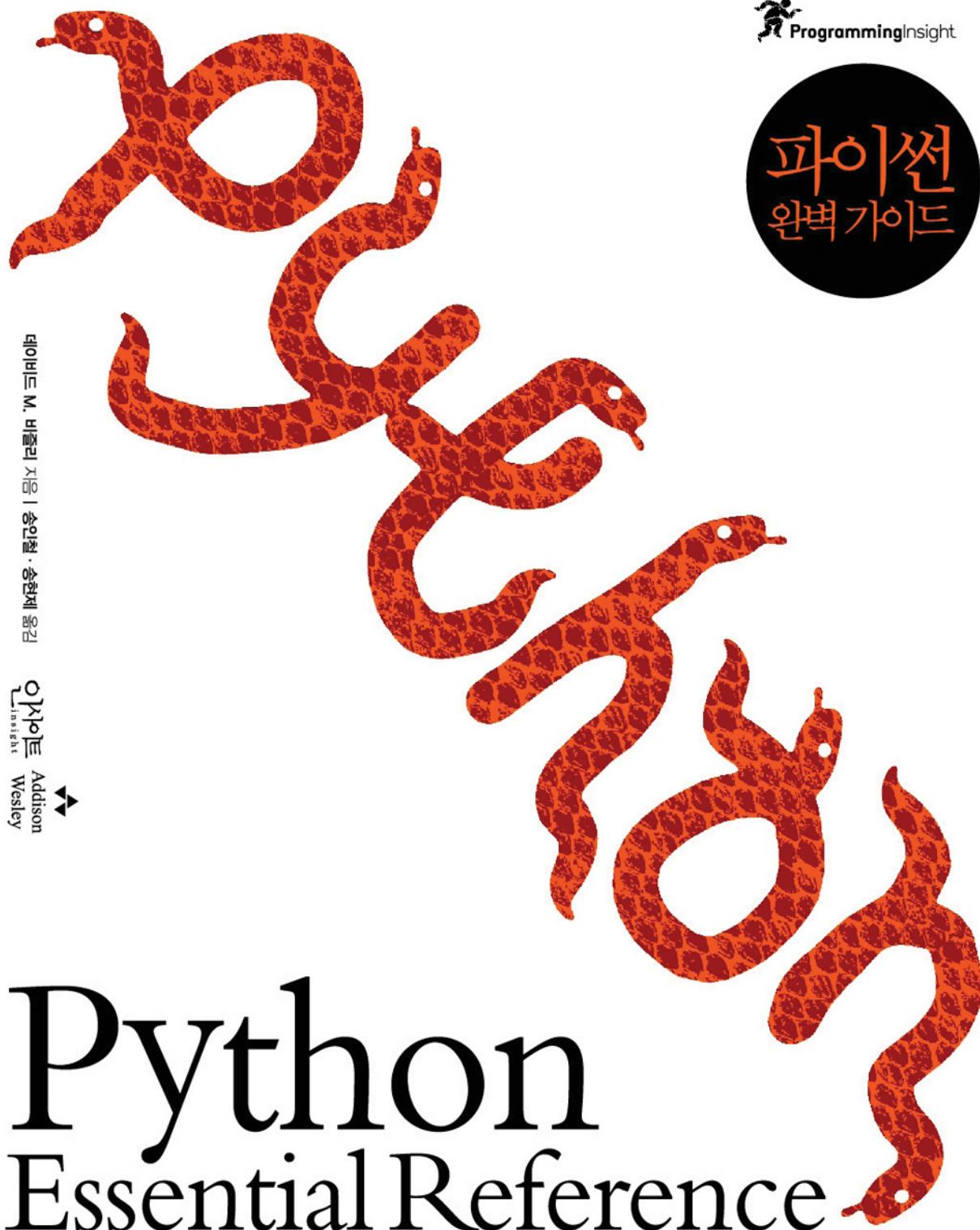


파이썬
완벽 가이드

데이비드 M. 버클리 지음 | 송인철 · 송현재 옮김

인사이트
Addison Wesley

Python Essential Reference



파이썬

완벽 가이드

Python Essential Reference

by David M. Beazley

Authorized translation from the English language edition, entitled PYTHON ESSENTIAL REFERENCE, 4th Edition by DAVID BEAZLEY, published by Pearson Education, Inc, publishing as Addison-Wesley Publishing, Copyright © 2010

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Electronic KOREAN language edition published by INSIGHT PRESS, Copyright © 2012

이 전자책의 한국어판 저작권은 에이전시 원을 통해 저자와의 독점 계약으로 인사이트 출판사에 있습니다. 신저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단전재와 무단복제를 금합니다.

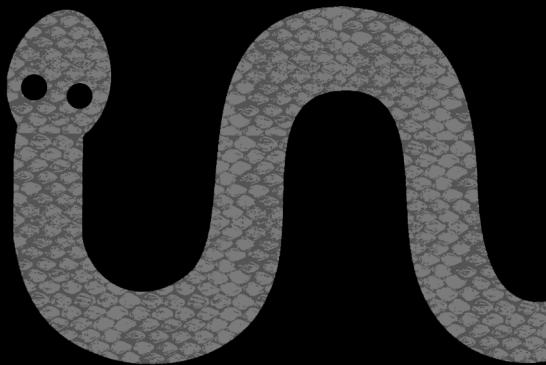
파이썬 완벽 가이드 Python Essential Reference 4th Edition

초판 PDF 1쇄 발행 2012년 4월 16일 **지은이** 데이비드 M. 비즐리 **옮긴이** 송인철, 송현재 **펴낸이** 한기성 **펴낸 곳** 인사이트 편집 이은순, 김승호 **등록번호** 제10-2313호 **등록일자** 2002년 2월 19일 **주소** 서울시 마포구 서교동 469-9번지 석우빌딩 3층 **전화** 02-322-5143 **팩스** 02-3143-5579 **블로그** <http://blog.insightbook.co.kr> **이메일** insight@insightbook.co.kr **ISBN** 978-89-6626-028-7



Python

Essential Reference



파이썬

완벽 가이드

데이비드 M. 비즐리 지음 | 송인철 · 송현제 옮김

인사이트
insight

차례

옮긴이의 글.....	xx
지은이의 글.....	xxii
감사의 글	xxiv

1부 파이썬 언어

1

1장 파이썬 맛보기	3
파이썬 실행하기.....	3
변수와 산술 표현식.....	5
조건문	8
파일 입력과 출력	9
문자열	10
리스트	12
튜플	14
집합	16
사전	16
반복과 루프.....	18
함수	19
생성기.....	20
코루틴	22
객체와 클래스.....	23
예외	25
모듈	26
도움 얻기	27

2장 어휘 규약과 구문	29
줄 구조와 들여쓰기	29
식별자와 예약어	30
숫자 상수	31
문자열 상수	32
컨테이너	35
연산자, 구분 문자, 특수 기호	35
문서화 문자열	36
장식자	36
소스 코드 인코딩	37
3장 타입과 객체	39
용어	39
객체 신원과 타입	40
참조 횟수와 쓰레기 수집	41
참조와 복사	42
1급 객체	44
데이터 표현을 위한 내장 타입	45
None 타입	46
숫자 타입	46
순서열 타입	47
매핑 타입	53
집합 타입	55
프로그램 구조를 나타내는 내장 타입	57
호출가능 타입	57
클래스, 타입, 인스턴스	60
모듈	61

인터프리터의 내부를 나타내는 내장 타입	62
코드 객체	62
프레임 객체	63
역추적 객체	64
생성기 객체	64
분할 객체	64
Ellipsis 객체	65
객체의 작동 방식과 특수 메서드	65
객체 생성 및 파괴	66
객체의 문자열 표현	67
객체 비교와 순서 매기기	68
타입 검사	69
속성 접근	69
속성 감싸기 및 기술자	70
순서열 및 매핑 메서드	70
반복	72
수학 연산	72
호출가능 인터페이스	75
컨텍스트 관리 프로토콜	75
객체 검사와 dir()	76
 4장 연산자와 표현식	77
숫자에 대한 연산	77
순서열에 대한 연산	80
문자열 포맷 지정	84
고급 문자열 포맷 지정	86
사전에 대한 연산	89
집합에 대한 연산	90
확장 대입	90
속성() 연산자	91
함수 호출() 연산자	91
변환 함수	92
불리언 표현식과 진리값	93
객체 동등 및 신원	94
평가 순서	94
조건 표현식	95

5장 프로그램 구조와 제어 흐름	97
프로그램 구조와 실행	97
조건부 실행	98
루프와 반복	98
예외	101
내장 예외	104
새로운 예외 정의	106
컨텍스트 관리자 with문	107
assert와 __debug__	109
6장 함수와 함수형 프로그래밍	111
함수	111
매개변수 전달과 반환 값	114
유효 범위 규칙	115
객체와 클로저로서 함수	117
장식자	121
생성기와 yield	123
코루틴과 yield 표현식	125
생성기와 코루틴의 사용	128
리스트 내포	130
생성기 표현식	132
선언형 프로그래밍	133
lambda 연산자	134
재귀	135
문서화 문자열	136
함수 속성	137
eval(), exec()과 compile()	138
7장 클래스와 객체지향 프로그래밍	141
class문	141
클래스 인스턴스	142
유효 범위 규칙	143
상속	144
다형성 동적 바인딩과 오리 타입화	148

정적 메서드와 클래스 메서드.....	149
프로퍼티.....	151
기술자	154
데이터 캡슐화와 개인 속성	155
객체 메모리 관리	156
객체 표현과 속성 바인딩.....	160
__slots__	162
연산자 오버로딩.....	163
타입과 클래스 멤버 검사.....	165
추상 기반 클래스.....	167
메타클래스.....	169
클래스 장식자	173
 8장 모듈, 패키지와 배포	175
모듈과 import문	175
모듈에서 선택된 기호 임포트.....	177
메인 프로그램으로 실행.....	179
모듈 검색 경로.....	180
모듈 로딩과 컴파일	181
모듈 재로딩과 내리기.....	182
패키지.....	183
파이썬 프로그램과 라이브러리 배포	186
써드파티 라이브러리 설치	189
 9장 입력과 출력	191
명령줄 옵션 읽기	191
환경 변수.....	193
파일과 파일 객체	193
표준 입력, 출력과 에러	197
print문	198
print() 함수	199
텍스트 출력에서 변수 보간	199
출력 생성	201
유니코드 문자열 처리	202

<u>유니코드 I/O</u>	204
<u>유니코드 데이터 인코딩</u>	206
<u>유니코드 문자 속성</u>	209
<u>객체 영속화와 pickle 모듈</u>	209
10장 실행 환경	213
<u>인터프리터 옵션과 환경</u>	213
<u>대화식 세션</u>	216
<u>파이썬 응용 프로그램 실행</u>	217
<u>사이트 설정 파일</u>	218
<u>사용자별 사이트 패키지</u>	219
<u>미래 기능 활성화</u>	220
<u>프로그램 종료</u>	221
11장 테스트, 디버깅, 프로파일링과 튜닝	223
<u>문서화 문자열과 doctest 모듈</u>	223
<u>단위 테스트와 unittest 모듈</u>	226
<u>파이썬 디버거와 pdb 모듈</u>	229
<u>디버거 명령</u>	230
<u>명령줄에서 디버깅</u>	234
<u>디버거 설정</u>	234
<u>프로그램 프로파일링</u>	234
<u>튜닝과 최적화</u>	236
<u>시간 측정</u>	236
<u>메모리 측정</u>	237
<u>분해</u>	238
<u>튜닝 전략</u>	239
2부 파이썬 라이브러리	245
12장 내장 함수와 예외	247
<u>내장 함수와 타입</u>	247
<u>내장 예외</u>	261

예외 기반 클래스	262
예외 인스턴스	262
미리 정의된 예외 클래스	263
내장 경고	266
future_builtins	268
13장 파일 런타임 서비스 —————	269
atexit	269
copy	269
gc	270
inspect	273
marshal	278
pickle	279
sys	283
변수	283
함수	287
traceback	290
types	292
warnings	293
weakref	296
예	298
14장 수학 —————	299
decimal	299
Decimal 객체	300
Context 객체	301
함수와 상수	304
예	306
fractions	307
math	309
numbers	310
random	312
씨 뿌리기와 초기화	312
무작위 정수	313

무작위 순서열.....	313
실수 무작위 분포.....	313
 15장 데이터 구조, 알고리즘과 코드 단순화 ——————	317
abc	317
array.....	319
bisect.....	322
collections	323
deque와 defaultdict.....	323
이름 있는 튜플.....	325
추상 기반 클래스.....	327
contextlib.....	330
functools.....	331
heapq.....	333
itertools	334
예	337
operator	338
 16장 문자열과 텍스트 처리—————	341
codecs	341
저수준 codecs 인터페이스	341
I/O 관련 함수	343
유용한 상수	344
표준 인코딩	345
re	345
패턴 문법	346
함수	348
정규 표현식 객체	349
Match Object.....	350
예	352
string	353
상수	353
Formatter 객체	354
Template 문자열	356
유틸리티 함수	356

struct	357
포장 함수와 풀기 함수	357
Struct 객체	358
포맷 코드	358
unicodedata	360
 17장 파일 데이터베이스 접근	365
관계 데이터베이스 API 명세서	365
연결	366
커서	366
질의 만들기	369
타입 객체	370
에러 처리	371
다중 스레드	372
결과를 사전에 매핑하기	372
데이터베이스 API 확장	373
sqlite3 모듈	373
모듈 수준 함수	374
Connection 객체	375
커서와 기본 연산	379
DBM 파일 데이터베이스 모듈	381
shelve 모듈	383
 18장 파일 및 디렉터리 다루기	385
bz2	385
filecmp	387
fnmatch	389
예	390
glob	390
예	390
gzip	390
shutil	391
tarfile	393
예외	397

예	397
tempfile.....	398
zipfile	400
zlib	404
19장 운영체제 서비스	407
commands	408
ConfigParser, configparser	408
ConfigParser 클래스	409
예	411
datetime	413
date 객체.....	414
time 객체	416
datetime 객체	417
timedelta 객체.....	419
날짜 관련 수학 연산.....	420
tzinfo 객체.....	421
날짜와 시간 파싱.....	423
errno	423
POSIX 에러 코드.....	423
윈도 에러 코드.....	425
fcntl	427
예	428
io	429
기반 I/O 인터페이스	429
무가공 I/O	430
버퍼 이진 I/O.....	431
텍스트 I/O.....	434
open() 함수.....	435
추상 기반 클래스.....	435
logging.....	436
로깅 수준	436
기본 설정	437
Logger 객체	438
처리기 객체.....	445

메시지 포맷 지정	449
기타 유ти리티 함수	451
로깅 설정	452
성능 고려	454
mmap	455
msvcrt	458
optparse	461
예	464
os	466
<u>프로세스 환경</u>	467
파일 생성과 파일 기술자	470
파일과 디렉터리	475
프로세스 관리	480
시스템 설정	487
예외	488
os.path	489
signal	492
예	495
subprocess	495
예	498
time	499
winreg	503
 20장 스레드와 동시성	507
기본 개념	508
동시 실행 프로그래밍과 파이썬	510
multiprocessing	511
<u>프로세스</u>	512
<u>프로세스 사이 통신</u>	514
<u>프로세스 풀</u>	522
공유 데이터와 동기화	525
관리 객체	528
연결	534
기타 유ти리티 함수	536
다중 프로세스 프로그래밍에 관한 일반적인 조언	536

threading	537
Thread 객체	538
Timer 객체	540
Lock 객체	540
RLock	541
세마포어와 경계 세마포어	542
이벤트	543
조건 변수	544
락 다루기	546
스레드 종료와 정지	547
유ти리티 함수	548
전역 인터프리터 락	548
스레드 프로그래밍	549
queue, Queue	549
스레드에서 큐 사용 예	550
코루틴과 마이크로스레딩	552
 21장 네트워크 프로그래밍과 소켓	553
네트워크 프로그래밍 기본	554
asynchat	557
asyncore	561
예	563
select	565
고급 모듈 가능	567
고급 비동기 I/O 예	567
언제 비동기 네트워킹을 사용할 것인가	575
socket	578
주소 체계	579
소켓 종류	579
주소	580
함수	583
예외	597
예	598
ssl	599
예	601

SocketServer	602
처리기.....	603
Servers	604
커스터마이즈된 서버 정의.....	606
응용 프로그램 서버의 커스터마이즈.....	608
22장 인터넷 응용 프로그래밍 —————	611
ftplib	611
예.....	615
http 패키지	615
http.client(httplib)	617
http.server(BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer).....	622
http.cookies(Cookie).....	628
http.cookiejar(cookielib).....	631
smtplib.....	632
예.....	633
urllib 패키지.....	633
urllib.request(urllib2).....	634
urllib.response	640
urllib.parse.....	640
urllib.error	645
urllib.robotparser(robotparser)	645
xmlrpc 패키지	646
xmlrpc.client(xmlrpclib)	646
xmlrpc.server(SimpleXMLRPCServer, DocXMLRPCServer).....	650
23장 웹 프로그래밍 —————	655
cgi	658
CGI 프로그래밍 조언	663
cgitb	665
wsgiref	666
WSGI 명세서.....	666
wsgiref 패키지	668
webbrowser	671

24장 인터넷 데이터 처리와 인코딩	673
base64	673
binascii.....	676
csv	677
방언	680
예	681
email 패키지	681
이메일 파싱.....	682
이메일 작성.....	686
hashlib	691
hmac	692
예	692
HTMLParser.....	693
예	695
json.....	696
mimetypes	700
quopri.....	701
xml 패키지	703
XML 문서 예	704
xml.dom.minidom	705
xml.etree.ElementTree.....	709
xml.sax	718
xml.sax.saxutils	722
25장 기타 라이브러리 모듈	723
파이썬 서비스.....	723
문자열 처리	724
운영체제 모듈	724
네트워크.....	725
인터넷 데이터 처리	725
국제화	726
멀티미디어 서비스.....	726
기타	726

3부 확장과 임베딩

727

26장 파이썬 확장과 임베딩	729
확장 모듈.....	730
확장 모듈 프로토타입.....	732
확장 모듈 이름.....	735
확장 기능 컴파일과 패키징.....	735
파이썬에서 C로 타입 변환.....	737
C에서 파이썬으로 타입 변환.....	743
모듈에 값 추가	745
에러 처리.....	746
참조 횟수 세기.....	748
스레드.....	749
파이썬 인터프리터 임베딩.....	750
임베딩 템플릿	750
컴파일과 링크.....	750
기본 인터프리터 연산과 설정.....	750
C에서 파이썬에 접근	752
파이썬 객체를 C로 변환	754
ctypes.....	755
공유 라이브러리 로드.....	755
외부 함수.....	756
데이터 타입.....	757
외부 함수 호출	758
기타 타입 생성 메서드들	760
유ти리티 함수.....	761
예	763
고급 확장과 임베딩	764
Jython과 IronPython	765
 부록 파이썬 3	767
누가 파이썬 3를 사용해야 하나?.....	767
새로운 언어 기능	769
소스 코드 인코딩과 식별자	769

집합 상수	769
집합과 사전 내포	769
확장된 반복 가능한 풀어헤치기	770
nonlocal 변수	771
함수 주석	771
키워드 전용 인수	773
표현식인 Ellipsis	774
연쇄 예외	774
향상된 super()	775
고급 메타클래스	775
흔한 위험	777
텍스트 대 바이트들	778
새로운 I/O 시스템	780
print()와 exec() 함수	781
반복자와 뷰의 사용	781
정수와 정수 나누기	782
비교	783
반복자와 생성기	783
파일 이름, 인수와 환경 변수	784
라이브러리 재구성	784
절대적인 임포트	785
코드 이주와 2to3	785
파이썬 2.6으로 코드 포팅	785
테스트 커버리지 제공	786
2 to 3 도구 사용	786
효과적 포팅 전략	789
파이썬 2와 3 동시 지원	790
참여하라	790
찾아보기	791

옮긴이의 글

처음에 이 책을 번역하자는 제의를 받았을 때 흔쾌히 수락하였다. 그 이유는 역자들이 이미 파이썬을 오랫동안 써오고 있었고 『파이썬 완벽 가이드』가 아마존에서 좋은 평점을 받았다고 들었기 때문이다.

최근 파이썬이 점점 널리 쓰이고 있다. 세계 최고의 IT 기업 중 하나인 구글에서도 파이썬을 활발하게 사용하고 있다. 클라우드 컴퓨팅의 일환으로 구글에서 제공하는 서비스인 구글 앱 엔진에서도 처음으로 지원했던 언어가 바로 파이썬이다. 요즘 역자 중 한 명이 다니고 있는 학교에서는 학부 전산학과 신입생이 처음으로 듣는 프로그래밍 수업에서 파이썬을 사용하기 시작했다. 그만큼 파이썬이라는 언어가 배우고 쓰기 쉽다는 사실을 말해주고 있다.

역자들의 주 언어는 원래 자바였다. 그러다가 파이썬을 접하게 되면서부터 코딩을 거의 파이썬으로 하고 있다. 파이썬은 문법이 깔끔하고 직관적이어서 생각을 코드로 옮기기가 쉽다. 그리고 다른 언어로 하면 오래 걸릴 일을 파이썬을 사용하면 순식간에 끝낼 수 있다. 다른 언어로 몇 줄에 걸쳐 작성해야 하는 코드도 파이썬으로 작성하면 한두 줄이면 끝나는 경우가 많다. 이렇게 파이썬을 점점 많이 쓰기 시작하면서 항상 옆에 두고 참고할만한 책이 있었으면 했다. 시중에 여러 책이 나와 있기는 하지만 다들 뭔가 부족한 것 같았다. 백과사전식으로 모든 함수를 나열해 놓은 것이 아니라 실제로 파이썬을 쓰면서 자주 쓰는 기능을 요약해놓은 책이 있으면 했다. 언제든 기억이 가물가물할 때 다시 찾아볼 수 있게 말이다. 그리고 파이썬에서 제공하는 방대한 라이브러리에는 뭐가 있는지, 어떻게 활용할 수 있는지를 잘 설명해 줄 책을 원했다. 『파이썬 완벽 가이드』는 이러한 역자들의 욕구를 100퍼센트 만족시켜 주었다.

책의 앞부분에서는 파이썬에서 가장 많이 쓰는 기능을 일목요연하게 정리한다. 특히 파이썬을 쓰면서 약간 까다롭다고 느낄 수 있는 부분들을 알기 쉽게 정리를 해놓은 점이 좋다. 아직까지 파이썬의 핵심 기능을 이 정도로 깔끔하게 정리해놓은 책을 발견하지 못했다. 그리고 책 뒷부분에는 파이썬에서 제공하는 라이브러리를 체계적으로 정리한다. 파이썬 공식 온라인 매뉴얼과 다른 점은 모든 기능을 나열하기보다는 저자의 오랜 경험을 바탕으로 파이썬에서 자주 쓰이는 기능을 집중적으

로 설명했다는 점이다. 따라서 뒷부분은 시간을 내어 한 번이라도 일독하기를 권한다. 파이썬에서 이렇게나 많은 기능을 제공하고 있다는 점에 놀랄 것이다. 지은이의 글에서는 앞으로 독자들이 파이썬으로 코딩할 때 항상 참고할 수 있는 책을 쓰고 싶었다고 하였는데, 역자들에게는 이미 필수 참고 서적이 되었다. 이 책을 읽게 될 독자들에게도 그런 책이 되었으면 하는 바람이다.

이 책이 나오기까지 많은 분들이 도움을 주셨다. 먼저, 항상 독자들을 위해 좋은 책을 펴내고자 노력하시는 인사이트 출판사의 한기성 사장님께 감사의 말씀을 드린다. 번역 일을 하면 할수록 느끼는 것이 편집자의 중요성인 것 같다. 이 책의 초안을 독자의 입장에서 검토하여 읽기 쉽게 다듬고, 의미가 정확하게 전달되도록 손봐 준 편집자 김승호 씨에게도 감사를 드린다. 이 책은 여러 리뷰어들의 검토를 거쳤다. 고민수, 목봉균, 박진영, 석우정, 임대림, 임진혁, 한성수 리뷰어들께 감사의 말을 전한다. 특히 많은 도움을 주신 김남형 리뷰어께 감사드린다. 마지막으로 곁에서 항상 힘이 되어주는 가족들에게 감사의 말을 전하고 싶다.

대전, 대구에서 봄을 맞이하며
송인철, 송현제

지은이의 글

나는 파이썬 프로그램 언어에 대한 간단하면서도 명료한 참고 자료를 제공하려고 이 책을 썼다. 숙련된 프로그래머라면 이 책으로 파이썬 프로그램 작성법을 배울 수도 있을 것이다. 하지만 기본적인 파이썬 프로그램 작성법을 설명하거나 파이썬 언어 자체에 대한 전문적인 정보를 제공하려고 이 책을 쓰지는 않았다. 나는 이 책을 쓰면서 파이썬 언어에서 가장 필수적인 부분을 담고 파이썬의 핵심 라이브러리를 간단하고 정확하게 기술하려고 노력했다. 이 책은 독자들이 이미 파이썬을 써보았거나 C나 자바 같은 다른 언어로 프로그래밍을 해본 경험이 있다고 가정한다. 시스템 프로그래밍 주제(운영체제의 기본 개념이나 네트워크 프로그래밍 등)에 관해 어느 정도 지식을 가지고 있으면 일부 라이브러리를 이해하는 데 도움이 될 것이다.

파이썬은 <http://www.python.org>에서 자유롭게 내려받을 수 있다. 유닉스, 윈도, 맥킨토시 등 거의 모든 운영체계에서 파이썬을 사용할 수 있다. 파이썬 웹 사이트에 가면 다양한 문서들, 하우투(how-to) 가이드, 써드 파티 소프트웨어 목록 등을 찾을 수 있다.

『파이썬 완벽 가이드』의 이번 4판은 파이썬이 진화하는 과정에서 중요한 시점에 나왔다. 즉, 파이썬 2.6과 파이썬 3.0이 거의 동시에 릴리스된 것이다. 파이썬 3은 이전 파이썬 버전과 역호환성을 지키지 않는 릴리스다. 저자이자 프로그래머로서 나는 파이썬 3으로 넘어갈지 아니면 대부분의 프로그래머들이 더 익숙하게 생각하는 파이썬 2.x 릴리스를 기반으로 할지 고민해야 했다.

나는 몇 년 전까지 C 프로그래밍을 했기 때문에 프로그램 언어에서 제공하는 기능을 사용할 때 특정 책을 절대적인 기준 자료로 사용했다. 예를 들어, 브라이언 커니핸(Brian Kernighan)과 데니스 리치(Dennis Ritchie)가 쓴 『C 프로그래밍 언어 (The C Programming Language)』에 나와 있지 않은 기능을 사용하면 코드 이식성이 떨어지니까 주의를 기울여야지 하는 식이었다. 프로그래머로서 이 접근법이 도움이 많이 되었고 이번 판에서 내가 취하기로 한 접근법이기도 하다. 즉, 파이썬 2의 기능 중에서 파이썬 3에서 제거된 기능은 설명하지 않기로 하였다. 비슷하게, 파이썬 3의 기능 중에서 파이썬 2.x 버전으로 다를 수 없는 기능에는 초점을 맞추지 않았다(부록에서 다루기는 할 것이다). 이렇게 함으로써 나는 사용 중인 파이썬 버전

에 상관없이 이 책이 모든 파이썬 프로그래머에게 유용한 지침서가 될 수 있으리라 생각한다.

10년도 더 전에 초판이 나온 아래, 이번 4판이야말로 아마 흥미로운 변화들을 가장 많이 담고 있을 것이다. 지난 몇 년간 파이썬 언어 개발은 새로운 언어 기능을 추가하는 것에 초점이 맞추어졌다. 특히 함수형 프로그래밍이나 메타 프로그래밍과 관련된 부분이 그렇다. 이번 판에서는 함수형 프로그래밍과 객체지향 프로그래밍에 관한 장을 크게 확장하여 생성기, 반복자, 코루틴, 장식자, 메타클래스 등의 주제를 자세히 다루었다. 라이브러리를 설명하는 부분에서도 최신 모듈의 내용을 담으려고 노력하였다. 책 전체에 걸쳐서 코드와 예를 새롭게 고쳤다. 아마 대부분의 프로그래머들이 바뀐 내용에 만족할 것이다.

마지막으로 파이썬과 관련해서 이미 수천 페이지에 달하는 유용한 문서들이 있다. 이 책의 내용 중 많은 부분이 이러한 문서에 도움을 받았다. 하지만 몇 가지 차이점이 있다. 먼저, 이 책에서는 관련 내용을 훨씬 압축된 형태로 제공하고 다양한 예를 담고 있으며 많은 주제를 나름대로의 방식으로 설명한다. 둘째, 라이브러리를 설명할 때는 상당 부분 외부 참고 자료를 포함하여 내용을 확장하였다. 특히 여러 매뉴얼이나 외부 참고 자료에 나와 있는 다양한 옵션을 고려해야 하는 저수준 시스템이나 네트워크 관련 모듈을 설명할 때 신경을 많이 썼다. 또한 좀더 간단 명료한 참고 자료를 제공하기 위해서 사용이 권장되지 않거나 비교적 사용 방법이 모호한 라이브러리 모듈은 과감히 제외했다.

나는 파이썬을 사용할 때 필요한 모든 내용과 파이썬의 대규모 라이브러리에 대한 설명을 이 책에 담으려고 애썼다. 파이썬이라는 언어를 친절히 소개하는 입문서와는 거리가 멀지도 모르겠다. 그래도 여러분이 앞으로 몇 년 동안 파이썬으로 프로그램을 작성할 때 늘 참고할 정도로 유용한 책이 되었으면 한다. 여러분의 의견을 언제든지 환영한다.

데이비드 비즐리

시카고, 일리노이

2009년 6월

감사의 글

많은 사람의 도움이 없었더라면 이 책은 나올 수 없었을 것이다. 가장 먼저 이 책의 4판 프로젝트에 뛰어들어서 유용한 피드백을 준 것에 대해서 Noah Gift에게 감사의 말을 전한다. Kurt Grandis도 여러 장에 대해서 조언을 아끼지 않았다. 또한 소중한 조언들로 이전 판들이 성공할 수 있게 도와준 점에 대해, 과거 기술 감수자였던 Timothy Boronczyk, Paul DuBois, Mats Wichmann, David Ascher, Tim Bell에게 감사의 말을 전한다. Guido van Rossum, Jeremy Hylton, Fred Drake, Roger Masse, Barry Warsaw도 지난 1999년 뜨거운 여름의 몇 주 동안 1판이 나올 수 있게 도와 주었다. 마지막으로 이 책은 독자들의 피드백이 없었더라면 나올 수 없었을 것이다. 이곳에 다 나열하기에는 너무 많은 수의 독자들이 도움을 주었고, 더 좋은 책이 될 수 있게 말해준 제안들을 모두 반영하기 위해서 최선을 다했다. 그리고 Addison-Wesley와 Pearson Education에 있는 모든 사람에게 그들이 보내준 이 프로젝트에 대한 끝없는 헌신과 도움에 감사의 말을 전하고 싶다. Mark Taber, Michael Thurston, Seth Kerney, Lisa Thibault 모두 이번 판이 멋진 모습으로 나올 수 있게 도왔다. 특히 이전 판들을 편집하는 데 큰 노력을 기울여 3판이 나올 수 있게 한 Robin Drake에게 감사의 말을 전한다. 마지막으로 멋진 아내이자 파트너인 Paula Kamen에게 격려와 유머, 사랑을 보내준 것에 대해 감사의 말을 전한다.

1부

파이썬 언어



P y t h o n E s s e n t i a l R e f e r e n c e

1장

P y t h o n E s s e n t i a l R e f e r e n c e

파이썬 맛보기

이 장에서는 파이썬을 간략히 소개한다. 특수한 규칙이나 세부 사항에 너무 얕매이지 않으면서 파이썬의 핵심적인 기능을 대부분 살펴볼 것이다. 이를 위해 이 장에서는 변수, 표현식, 제어 흐름, 함수, 생성기, 클래스, 입출력 같은 기본적인 개념들을 간략히 다룬다. 하지만, 이러한 개념들을 아주 자세하게 다루지는 않는다. 그래도 숙련된 프로그래머는 이 장에 나오는 내용을 바탕으로 더 고급 프로그램을 작성할 수 있을 것이다. 초보 프로그래머라면 파이썬에 대한 감을 잡기 위해 몇 가지 예를 직접 실행해볼 것을 권한다. 파이썬을 처음 접하는데 파이썬 3을 사용 중이라면 이 장은 파이썬 2를 사용해 따라가는 것이 좋을 것이다. 사실상 대부분의 개념이 두 버전 모두에 적용되지만, 파이썬 3에서는 몇 가지 구문상의 중대한 변화(대부분 출력 및 I/O와 관련된)가 있기 때문에 이 장에 나오는 많은 예들이 실행되지 않을 수 있다. 여기에 관한 더 자세한 내용은 부록 ‘파이썬 3’을 참고하기 바란다.

파이썬 실행하기

파이썬 프로그램은 인터프리터에 의해 실행된다. 인터프리터는 보통 명령 셸에서 python을 입력해 실행한다. 인터프리터에는 여러 가지 구현이 있으므로(Jython, IronPython, IDLE, ActivePython, Wing IDE, pydev 등) 인터프리터를 실행하는 방법은 관련 문서를 찾아보기 바란다. 인터프리터가 시작되면 단순히 읽고 평가하기를 반복하는 루프에 프로그램을 입력할 수 있는 프롬프트가 나타난다. 예를 들어, 다음 출력 결과는 인터프리터에서 저작권 메시지가 출력된 후 >>> 프롬프트가 나타나

고 여기에 친숙한 “Hello World” 명령을 입력한 모습을 보여준다.

```
Python 2.6rc2 (r26rc2:66504, Sep 19 2008, 08:50:24)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> print "Hello World"
Hello World
>>>
```

Note

앞의 예를 실행했는데 SyntaxError가 발생하면 아마 파이썬 3을 사용하고 있어서 그럴 것이다. 만약 실제로 그렇다면 이 장을 계속 읽어 나가도 된다. 다만 파이썬 3에서 print문이 함수로 변했다는 것을 염두에 두기 바란다. 따라서, 앞으로 나올 예에서 출력할 내용을 간단히 괄호로 둘러싸면 된다. 예를 들어, 다음과 같이 하면 된다.

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

출력할 내용을 괄호로 둘러싸는 방식은 항목이 하나인 경우에 한해 파이썬 2에서도 작동한다. 하지만 기존 파이썬 코드에서 그리 많이 볼 수 있지는 않다. 이 구문은 이후 장에서 주된 초점이 출력과 관련된 기능이 아니면서 파이썬 2와 3에서 모두 작동하도록 작성된 예에서 종종 사용된다.

파이썬의 대화식 모드는 정말 유용하다. 대화식 셸에서는 유효한 문장 또는 일련의 문장들을 입력하고 그 결과를 바로 확인할 수 있다. 필자를 포함한 많은 이들은 심지어 대화식 파이썬을 탁상용 계산기로 사용하기도 한다. 다음은 그 예를 보여 준다.

```
>>> 6000 + 4523.50 + 134.12
10657.620000000001
>>> _ + 8192.32
18849.940000000002
>>>
```

파이썬을 대화식 모드로 사용하고 있을 때 특수한 변수인 `_`는 최종 연산 결과를 담는다. 이 변수는 최종 연산 결과를 저장해 두었다가 나중에 사용하고자 할 때 유용하게 쓰인다. 한 가지 명심해야 할 점은 이 변수가 대화식 모드에서만 정의된다 는 점이다.

프로그램을 반복 실행하고 싶다면 다음과 같이 문장들을 파일에 저장하면 된다.

```
# helloworld.py
print "Hello World"
```

파이썬의 소스 파일은 평범한 텍스트 파일이며 파일 확장자는 .py이다. # 문자는 줄 끝까지 이어지는 주석을 나타낸다. helloworld.py 파일을 실행하려면 다음과 같이 인터프리터를 실행할 때 파일 이름을 인수로 넘겨주면 된다.

```
% python helloworld.py
Hello World
%
```

윈도에서는 .py 파일을 마우스로 더블 클릭하거나 윈도 시작 메뉴에서 실행을 클릭한 후 프로그램 이름을 입력하여 실행할 수 있다. 이렇게 하면 인터프리터가 구동되고 콘솔 윈도에서 프로그램이 실행된다. 주의할 점은 프로그램이 종료되는 순간 곧바로 콘솔 윈도가 사라진다는 점이다(종종 결과를 확인하기도 전에). 디버깅을 위해서라면 IDLE 같은 파이썬 개발 도구에서 프로그램을 실행하는 것이 좋다.

유닉스에서는 다음과 같이 프로그램의 첫째 줄에 #!를 추가할 수 있다.

```
#!/usr/bin/env python
print "Hello World"
```

인터프리터는 입력 파일의 끝에 도달할 때까지 문장을 실행한다. 대화식 모드라면 EOF(end of file, 파일 끝을 의미) 문자를 입력하거나 파이썬 IDE의 풀다운 메뉴에서 Exit를 선택하여 인터프리터를 종료할 수 있다. 유닉스에서 EOF는 Ctrl+D이고 윈도에서는 Ctrl+Z(또는 F6)이다. SystemExit 예외를 발생시켜서 프로그램을 종료할 수도 있다.

```
>>> raise SystemExit
```

변수와 산술 표현식

코드 1.1은 변수와 표현식을 사용해 간단한 복리 계산을 수행하는 프로그램이다.

코드 1.1 간단한 복리 계산

```
principal = 1000    # 초기 금액
rate = 0.05         # 이자율
numyears = 5        # 몇 년인지
year = 1
while year <= numyears:
    principal = principal * (1 + rate)
    print year, principal    # 파이썬 3에서는 print(year, principal)
```

```
year += 1
```

이 프로그램의 실행 결과는 다음과 같다.

```
1 1050.0
2 1102.5
3 1157.625
4 1215.50625
5 1276.2815625
```

파이썬은 프로그램이 실행되는 도중 변수 이름에 묶여 있는 값 또는 타입이 변할 수 있는 동적 타입 언어다. 대입 연산자는 변수 이름과 값을 연결하기만 한다. 각 값은 정수나 문자열 같은 연관 타입을 가지지만, 변수 이름에는 타입이 없어서 실행 도중 어떤 타입의 데이터라도 가리킬 수 있다. 이 점은 변수 이름이 타입, 크기, 값이 저장될 메모리 위치와 연결되는 C 같은 언어와 다르다. 파이썬의 동적인 측면은 코드 1.1에서 principle 변수와 관련된 부분에서 살펴볼 수 있다. 처음에 이 변수에는 정수 값이 할당된다. 하지만 이후에 다음과 같이 다른 값이 할당된다.

```
principal = principal * (1 + rate)
```

앞의 문장은 표현식을 평가하고 principal이라는 이름에 결과 값을 다시 연결한다. principal의 원래 값은 정수 1000이었지만, 새 값은 이제 부동 소수점 수가 되었다(rate가 실수로 정의되어 있어서 위의 표현식도 실수가 된다).^{*} 따라서, principal의 외관상으로 보이는 ‘타입’은 프로그램 실행 도중에 정수에서 실수로 동적으로 변했다. 하지만, 정확히 말하자면 변한 것은 principle의 타입이 아니라 principle이라는 이름이 가리키는 값의 타입이다.

줄바꿈 문자(newline) 각 문장을 끝낸다. 하지만, 다음과 같이 세미콜론을 사용해 동일한 줄에 여러 문장을 같이 적어줄 수도 있다.

```
principal = 1000; rate = 0.05; numyears = 5;
```

while문은 바로 이어 나오는 조건식을 평가한다. 평가 값이 참이면 while문의 몸체가 실행된다. 그리고 나서 조건이 다시 평가되고 조건이 거짓이 될 때까지 몸체가 반복 실행된다. 들여쓰기에 의해 루프의 몸체가 표시되기 때문에 코드 1.1에서 while에 바로 따라 나오는 세 문장이 각 반복마다 실행된다. 파이썬은 블록 안에서 일관

* (옮긴이) 부동 소수점(floating point)은 컴퓨터에서 실수를 표현하는 방식이다. 따라서, 부동 소수점 수와 실수를 같은 것으로 생각해도 문제가 없다.

성만 있으면 들여쓰기를 얼마나 해야 하는지를 규정하지 않는다. 하지만, 보통은 각 들여쓰기 수준마다 네 개의 공백을 사용한다(일반적으로 권장되는 방식이다).

코드 1.1에 나와 있는 프로그램에서 한 가지 문제점은 결과가 씩 보기 좋지 않다는 점이다. 열을 오른쪽으로 정렬하고 principle의 정밀도를 두 자리 숫자로 제한하는 것이 더 보기 좋을 것 같다. 이렇게 하기 위한 방법이 몇 가지 있다. 그중에 문자열 포맷 연산자(%)가 주로 쓰인다.

```
print "%3d %.2f" % (year, principal)
print("%3d %.2f" % (year, principal)) # 파이썬 3
```

이제 프로그램의 출력 결과는 다음과 같다.

```
1 1050.00
2 1102.50
3 1157.63
4 1215.51
5 1276.28
```

포맷 문자열은 평범한 텍스트와, “%d”, “%s”와 “%f” 같은 특수한 포맷 지정자들을 담는다. 포맷 지정자는 정수, 문자열, 부동 소수점 수 같은 데이터 타입의 포맷을 지정한다. 또한 폭과 정밀도를 지정하는 변경자를 포함할 수도 있다. 예를 들어, “%3d”는 폭이 3인 열에서 오른쪽으로 정렬된 정수를 의미하고 “%.2f”는 소수점 아래로 두 자리 숫자를 표시하는 부동 소수점 수를 의미한다. 포맷 문자열이 작동하는 방식은 C의 printf() 함수와 거의 동일하다. 자세한 내용은 4장을 참고하기 바란다.

요즘은 다음과 같이 format() 함수를 사용하여 각 부분에 대해 별개로 포맷을 지정하는 방법이 많이 사용된다.

```
print format(year,"3d"),format(principal,"0.2f")
print(format(year,"3d"),format(principal,"0.2f")) # 파이썬 3
```

format()은 기존 문자열 포맷 연산자(%)에서 사용되는 것과 비슷한 포맷 지정자를 사용한다. 예를 들어, “3d”는 폭이 3인 오른쪽으로 정렬된 정수를, “0.2f”는 두 자리 숫자의 정확도를 가지는 부동 소수점 수를 나타낸다. 문자열에도 format() 메서드가 존재하며, 이 메서드로 다음과 같이 한 번에 여러 값에 대한 포맷을 지정할 수 있다.

```
print "{0:3d} {1:0.2f}".format(year,principal)
print("{0:3d} {1:0.2f}".format(year,principal)) # 파이썬 3
```

앞의 예에서, “{0:3d}”와 “{1:0.2f}”의 콜론 앞에 있는 숫자는 `format()` 메서드에 전달된 인수를 가리키고 콜론 뒤에 있는 부분은 포맷 지정자이다.

조건문

`if`와 `else`문은 간단한 테스트를 위해 사용된다. 다음은 한 예다.

```
if a < b:
    print "Computer says Yes"
else:
    print "Computer says No"
```

`if`와 `else`절의 몸체는 들여쓰기로 표시한다. `else`절은 생략할 수 있다.

`pass`문은 다음에서 보듯이 빈 절을 나타낸다.

```
if a < b:
    pass # 아무 일도 안 함
else:
    Print "Computer says No"
```

키워드 `or`, `and`와 `not`으로 불리언 표현식을 만들 수 있다.

```
if product == "game" and type == "pirate memory" \
    and not (age < 4 or age > 8):
    print "I'll take it!"
```

Note

복잡한 조건을 작성하다 보면 줄이 성가실 정도로 길어지는 경우가 있다. 앞에 나온 예처럼, 줄 끝에 역슬래시를 넣어주면 다음 줄에서 계속 문장을 이어갈 수 있다. 이때 보통의 들여쓰기 규칙이 적용되지 않기 때문에 이어지는 줄에서는 원하는 대로 문장을 작성하면 된다.*

파이썬에는 `switch`나 `case`문이 따로 없다. 여러 조건 검사가 필요한 경우에는 다음과 같이 `elif`문을 사용하면 된다.

```
if suffix == ".htm":
    content = "text/html"
elif suffix == ".jpg":
    content = "image/jpeg"
elif suffix == ".png":
    content = "image/png"
else:
```

* (옮긴이) (), {}, []로 둘러싸인 코드에서는 역슬래시를 넣을 필요가 없다.

```
raise RuntimeError("Unknown content type")
```

진리값은 불리언 값인 True와 False로 표현한다. 다음은 한 예다.

```
if 'spam' in s:
    has_spam = True
else:
    has_spam = False
```

'X'나 'Y' 같은 모든 관계 연산자는 그 결과로 True나 False를 반환한다. 앞의 예에서 사용된 in 연산자는 주어진 값이 문자열, 리스트 혹은 사전 같은 객체에 들어 있는지를 검사하는 데 주로 사용된다. in 연산자도 True나 False를 반환하므로 앞의 예는 다음과 같이 줄여 쓸 수 있다.

```
has_spam = 'spam' in s
```

파일 입력과 출력

다음 프로그램은 파일을 열어 내용을 한 줄씩 읽는다.

```
f = open("foo.txt")          # 파일 객체를 반환
line = f.readline()          # 파일에 대고 readline( ) 메서드 호출
while line:
    print line,             # 끝에 붙어 있는 ','는 줄바꿈 문자를 생략
    # print(line,end="")
    line = f.readline()      # 파일 3에서는 이렇게 함
f.close()
```

open() 함수는 새 파일 객체를 반환한다. 이 객체에 대해 여러 메서드를 호출하여 파일과 관련된 다양한 연산을 수행할 수 있다. readline() 메서드는 입력으로부터 끝의 줄바꿈 문자를 포함한 한 줄을 읽는다. 파일의 끝에 도달하면 빈 문자열이 반환된다.

앞의 예에서는 간단히 foo.txt 파일에 있는 모든 줄에 대해 루프를 돈다. 이렇게 데이터 컬렉션(입력 줄, 숫자, 문자열 등)에 대해 루프를 도는 것을 흔히 반복(iteration)이라고 한다. 반복은 매우 자주 사용되는 연산이기 때문에 파일에는 여러 항목에 대해 반복 수행을 하는 전용 for문이 있다. 예를 들어, 앞의 예는 다음과 같이 훨씬 더 간결하게 작성할 수 있다.

```
for line in open("foo.txt"):
    print line,
```

프로그램의 출력을 파일에 쓰려면 다음과 같이 >>를 사용해 print문에 출력할 파일을 지정하면 된다.

```
f = open("out","w") # 쓰기용으로 파일을 연다.  
while year <= numyears:  
    principal = principal * (1 + rate)  
    print >>f,"%3d %0.2f" % (year,principal)  
    year += 1  
f.close()
```

〉) 구문은 파이썬 2에서만 작동한다. 파이썬 3에서는 print문을 다음과 같이 바꾸면 된다.

```
print("%3d %0.2f" % (year,principal),file=f)
```

파일 객체에는 미가공(raw) 데이터를 쓰는 데 사용되는 write() 메서드도 있다. 예를 들어, 바로 앞의 예는 다음과 같이 쓸 수도 있다.

```
f.write("%3d %0.2f\n" % (year,principal))
```

앞의 예들은 파일에 관한 것이었지만, 동일한 기법을 인터프리터의 표준 입력 스트림과 표준 출력 스트림에도 적용할 수 있다. 예를 들어, 대화식으로 사용자 입력을 읽어 들이려면 sys.stdin 파일을 읽으면 된다. 화면에 데이터를 출력하고 싶으면 sys.stdout에 쓰면 된다. sys.stdout은 print문에 의해 생성되는 데이터가 출력되는 파일이기도 하다. 다음은 그 예다.

```
import sys  
sys.stdout.write("Enter your name :")  
name = sys.stdin.readline()
```

파이썬 2에서는 앞의 코드를 다음과 같이 줄일 수 있다.

```
name = raw_input("Enter your name :")
```

파이썬 3에서는 raw_input()이 input()으로 바뀌었다. 작동 방식은 동일하다.

문자열

다음과 같이 작은따옴표나 큰따옴표 또는 삼중따옴표로 둘러싸서 문자열 상수를 만든다.

```
a = "Hello World"  
b = 'Python is groovy'  
c = """Computer says 'No'"""
```

문자열 시작 부분의 따옴표와 끝부분의 따옴표는 같은 종류여야 한다. 논리적으로 한 줄 안에 있어야 하는 작은따옴표나 큰따옴표와는 달리, 삼중따옴표는 종

료를 알리는 삼중따옴표가 나오기 전까지의 모든 텍스트를 담는다. 삼중따옴표는 다음과 같이 여러 줄에 걸친 내용을 담고 있는 문자열 상수를 생성할 때 유용하게 쓰인다.

```
print "Content-type: text/html

<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.
""
```

문자열은 문자들의 순서열로 저장되고 0부터 시작하는 정수로 색인된다. 문자 하나를 추출하려면 다음과 같이 색인 연산자 `s[i]`를 사용하면 된다.

```
a = "Hello World"
b = a[4]      # b = 'o'
```

부분문자열을 얻으려면 분할 연산자 `s[i:j]`를 사용한다. `s[i:j]`는 범위가 $i \leq k < j$ 인 색인 k 에 해당하는 모든 문자열을 추출한다. 생략된 색인은 문자열의 시작이나 끝으로 취급된다.

```
c = a[:5]      # c = "Hello"
d = a[6:]      # d = "World"
e = a[3:8]     # e = "lo Wo"
```

플러스 연산자(+)는 문자열을 연결한다.

```
g = a + " This is a test"
```

파이썬은 결코 암묵적으로 문자열의 내용을 숫자 데이터로 해석하지 않는다(Perl이나 PHP 같은 언어와는 달리). 예를 들어, +는 항상 문자열을 연결한다.

```
x = "37"
y = "42"
z = x + y      # z = "3742" (문자열 연결)
```

수리 계산을 수행하려면 `int()`나 `float()` 같은 함수로 문자열을 먼저 숫자 값으로 변환해야 한다. 다음은 한 예다.

```
z = int(x) + int(y)    # z = 79 (정수 +)
```

문자열이 아닌 값은 `str()`, `repr()`, `format()` 함수를 사용해 문자열 표현으로 변환 할 수 있다. 다음은 몇 가지 예다.

```
s = "The value of x is " + str(x)
s = "The value of x is " + repr(x)
s = "The value of x is " + format(x,"4d")
```

`str()`과 `repr()`는 둘 다 문자열을 생성하지만 보통 그 결과는 약간 다르다. `str()`는 `print`문을 사용할 때 얻는 결과를 생성하지만, `repr()`는 객체의 값을 정확히 입력하기 위해 여러분이 프로그램에 입력해야 하는 문자열을 생성한다. 다음은 한 예다.

```
>>> x = 3.4
>>> str(x)
'3.4'
>>> repr(x)
'3.399999999999999'
>>>
```

앞의 예에서 3.4가 부정확하게 표현되는 것은 파이썬의 버그가 아니다. 그것은 주어진 컴퓨터 하드웨어에서 기본수(base) 10인 십진수 소수들을 정확히 표현하지 못하는 배정밀도(double-precision) 부동 소수점 수 때문이다.

`format()` 함수는 주어진 값을 지정된 포맷을 적용해 문자열로 변환한다. 다음은 한 예다.

```
>>> format(x, "0.5f")
'3.40000'
>>>
```

리스트

리스트(list)는 임의의 객체들의 순서열이다. 다음과 같이 대괄호로 값을 둘러싸서 리스트를 만든다.

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

리스트는 0부터 시작하는 정수로 색인된다. 리스트의 개별 항목에 접근하거나 수정하려면 색인 연산자를 사용하면 된다.

```
a = names[2]          # 리스트의 세 번째 항목인 "Ann"을 반환
names[0] = "Jeff"     # 첫 번째 항목을 "Jeff"로 교체
```

리스트의 끝에 새로운 항목을 추가하려면 `append()` 메서드를 사용하면 된다.

```
names.append("Paula")
```

리스트의 가운데에 항목을 삽입하려면 `insert()` 메서드를 사용하면 된다.

```
names.insert(2, "Thomas")
```

리스트의 일부를 추출하거나 재할당하는 데에는 분할 연산자를 사용한다.

```
b = names[0:2]        # [ "Jeff", "Mark" ]를 반환
```

```
c = names[2:]          # [ "Thomas", "Ann", "Phil", "Paula" ]를 반환
names[1] = 'Jeff'      # 두 번째 항목을 'Jeff'로 교체
names[0:2] = ['Dave','Mark','Jeff']  # 리스트 앞쪽 두 항목을
                                    # 오른쪽에 있는 리스트로 교체
```

리스트를 연결하려면 플러스 연산자를 사용하면 된다.

```
a = [1,2,3] + [4,5]  # 결과는 [1,2,3,4,5]
```

다음 두 방법 중 하나로 빈 리스트를 생성할 수 있다.

```
names = []            # 빈 리스트
names = list( )       # 빈 리스트
```

리스트는 다른 리스트를 포함한, 아무 파이썬 객체나 담을 수 있다.

```
a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

중첩된 리스트에 들어 있는 항목에 접근하려면 색인 연산자를 여러 번 사용하면 된다.

```
a[1]           # "Dave"를 반환
a[3][2]         # 9를 반환
a[3][3][1]     # 101을 반환
```

코드 1.2는 리스트의 고급 기능을 보여주기 위해 명령줄에서 지정된 파일로부터 숫자 목록을 읽어와서 최솟값과 최댓값을 출력하는 코드이다.

코드 1.2 리스트의 고급 기능

```
import sys          # sys 모듈을 로드한다.
if len(sys.argv) != 2 :    # 명령줄 인수의 개수를 검사한다.
    print "Please supply a filename"
    raise SystemExit(1)
f = open(sys.argv[1])      # 명령줄에서 파일 이름을 얻어 온다.
lines = f.readlines()      # 모든 줄을 리스트로 읽어 들인다.
f.close()

# 모든 입력 값을 문자열에서 실수로 변환한다.
fvalues = [float(line) for line in lines]

# 최솟값과 최댓값을 출력한다.
print "The minimum value is ", min(fvalues)
print "The maximum value is ", max(fvalues)
```

앞의 코드의 첫 번째 줄에서 파이썬 라이브러리로부터 sys 모듈을 로드하기 위해 import문을 사용했다. 이 모듈은 명령줄 인수들을 얻는 데 사용된다.

open() 함수는 명령줄 옵션으로 지정된, sys.argv에 담겨 있는 파일 이름을 사용

한다. `readlines()` 메서드는 입력되는 줄 전체를 문자열 리스트로 읽어 들인다.

표현식 `[float(line) for line in lines]`은 리스트 `lines`에 있는 모든 문자열에 대해 루프를 돌면서 각 요소에 `float()` 함수를 적용해 새로운 리스트를 생성한다. 이런 식으로 리스트를 생성하는 특별하고도 강력한 방법을 리스트 내포(list comprehension)라고 한다. 파일 안에 있는 줄들은 for 루프로도 읽을 수 있기 때문에, 앞의 프로그램을 다음과 같이 한 문장으로 줄일 수 있다.

```
fvalues = [float(line) for line in open(sys.argv[1])]
```

입력되는 줄들이 부동 소수점 수들의 리스트로 변환되고 나면, 마지막으로 내장 함수인 `min()` 및 `max()`로 최솟값과 최댓값이 계산된다.

튜플

튜플(tuple)을 사용하면 값들을 단일 개체에 채워 넣음으로써 간단한 데이터 구조를 생성할 수 있다. 다음과 같이 값을 팔호로 둘러싸서 튜플을 생성한다.

```
stock = ('GOOG', 100, 490.10)
address = ('www.python.org', 80)
person = (first_name, last_name, phone)
```

종종 팔호가 생략된 경우에도 파이썬은 튜플 생성 의도를 알아차린다.

```
stock = 'GOOG', 100, 490.10
address = 'www.python.org', 80
person = first_name, last_name, phone
```

특수한 구문을 사용해 0개와 1개의 요소를 가지는 튜플도 정의할 수 있다.

```
a = ()          # 0개 요소 튜플(빈 튜플)
b = (item,)     # 1개 요소 튜플(끝에 있는 콤마 주목)
c = item,       # 1개 요소 튜플(끝에 있는 콤마 주목)
```

리스트처럼 튜플의 값도 숫자 색인으로 추출할 수 있다. 하지만 다음과 같이 튜플을 변수들로 풀어헤치는 방식이 더 흔히 쓰인다.

```
name, shares, price = stock
host, port = address
first_name, last_name, phone = person
```

리스트가 지원하는 대부분의 연산(색인, 분할, 연결 등)을 튜플도 지원하지만 한번 생성되고 나면 튜플의 내용은 변경할 수 없다(즉, 기존 튜플의 요소를 대체, 삭제하거나 새로운 요소를 추가할 수 없다). 따라서, 튜플은 항목을 추가하거나 삭제

할 수 있는 객체들의 컬렉션이 아니라, 여러 부분으로 이루어진 단일 객체로 보는 것이 더 적절하다.

튜플과 리스트는 겹치는 부분이 많고 리스트가 좀 더 유연성이 있기 때문에, 프로그래머들은 아예 튜플을 완전히 무시하고 리스트만 사용하려는 경향이 있다. 이것이 문제가 되지 않을 수도 있지만, 프로그램에서 작은 리스트를 많이 생성하는 경우(즉, 각 리스트가 10개도 안 되는 항목을 담는 경우)에는 메모리를 많이 낭비하게 된다. 그 이유는 새로운 항목을 추가하는 연산의 속도를 빠르게 하기 위해 리스트에서는 메모리를 필요한 것보다 약간 더 많이 할당하기 때문이다. 튜플은 변경이 불가능하기 때문에 추가 공간 없이 조밀하게 저장될 수 있다.

데이터를 저장하는 데 튜플과 리스트를 함께 사용하기도 한다. 예를 들어, 다음 프로그램은 콤마로 구분되는 여러 열을 갖는 데이터를 읽어 들이는 방법을 보여준다.

```
# "종목(name), 보유량(shares), 주가(price)" 형식으로 된 줄들이 있는 파일
filename = "portfolio.csv"
portfolio = []
for line in open(filename):
    fields = line.split(",")           # 줄을 분할해 리스트로 만든다.
    name = fields[0]                  # 각 필드를 추출하고 변환한다.
    shares = int(fields[1])
    price = float(fields[2])
    stock = (name, shares, price)     # 튜플을 만든다.
    portfolio.append(stock)          # 레코드 리스트에 추가한다.
```

문자열의 split() 메서드는 주어진 구분 문자를 기준으로 문자열을 분할해 리스트를 생성한다. 이 프로그램을 실행하면 생성되는 portfolio 데이터 구조는 행과 열로 구성되는 이차원 배열 같은 구조를 갖는다. 각 행은 튜플로 표현되며 다음과 같이 접근할 수 있다.

```
>>> portfolio[0]
('GOOG', 100, 490.10)
>>> portfolio[1]
('MSFT', 50, 54.23)
>>>
```

각 데이터 항목은 다음과 같이 접근한다.

```
>>> portfolio[1][1]
50
>>> portfolio[1][2]
54.23
>>>
```

다음은 모든 레코드를 순회하면서 필드들을 변수들로 풀어 헤치는 간단한 방법이다.

```
total = 0.0
for name, shares, price in portfolio:
    total += shares * price
```

집합

집합(set)은 객체들의 순서 없는 모음을 담는 데 사용된다. 집합은 다음과 같이 set() 함수에 일련의 항목들을 넘겨주어 생성한다.

```
s = set([3,5,9,10])      # 숫자들의 집합을 생성한다.
t = set("Hello")         # 고유한 문자들의 집합을 생성한다.
```

리스트나 튜플과는 달리, 집합은 순서가 없기 때문에 숫자로 색인될 수 없다. 게다가, 집합은 요소가 중복되는 일이 없다. 가령 앞의 코드에 나온 t의 값을 들여다 보면, 다음을 알게 된다.

```
>>> t
set(['H', 'e', 'l', 'o'])
```

'l'이 한번만 나타나는 것을 볼 수 있다.

집합은 합집합, 교집합, 차집합, 대칭 차집합 등 표준 연산들을 지원한다. 다음은 그 예를 보여준다.

```
a = t | s    # t와 s의 합집합
b = t & s    # t와 s의 교집합
c = t - s    # 차집합 (t에는 있지만 s에는 없는 항목들)
d = t ^ s    # 대칭 차집합 (t나 s 중 하나에만 들어 있는 항목들)
```

add()와 update()는 집합에 새로운 아이템을 추가하는 데 사용된다.

```
t.add('x')          # 아이템을 하나 추가한다.
s.update([10,37,42]) # 여러 아이템을 추가한다.
```

remove()로 아이템을 제거할 수도 있다.

```
t.remove('H')
```

사전

사전(dictionary)은 키로 색인되는 객체들을 담는 연관 배열 혹은 해시 테이블이다. 다음과 같이 값들을 중괄호({ })로 둘러싸서 사전을 생성한다.

```
stock = {
    "name"   : "GOOG",
    "shares" : 100,
    "price"  : 490.10
}
```

사전의 요소에 접근하려면 다음과 같이 키 색인 연산자를 사용한다.

```
name = stock["name"]
value = stock["shares"] * shares["price"]
```

객체 추가 및 수정은 다음과 같이 수행한다.

```
stock["shares"] = 75
stock["date"] = "June 7, 2007"
```

주로 문자열이 키로서 사용되지만, 숫자나 튜플 같은 다른 파이썬 객체도 키로 사용될 수 있다. 리스트나 사전 같이 그 내용이 바뀔 수 있는 객체는 키로 사용할 수 없다.

사전은 앞에서 본 것처럼 이름 있는 필드들로 구성되는 객체를 정의하는 데 유용하게 쓰인다. 하지만, 사전은 순서 구분 없는 데이터를 빠르게 검색하기 위한 용도의 컨테이너로도 사용된다. 예를 들어, 다음은 주가를 담는 사전이다.

```
prices = {
    "GOOG" : 490.10,
    "AAPL" : 123.50,
    "IBM" : 91.50,
    "MSFT" : 52.13
}
```

빈 사전은 다음 두 가지 방법으로 만들 수 있다.

```
prices = {}          # 빈 사전
prices = dict()      # 빈 사전
```

사전에 어떤 키가 들어 있는지는 다음과 같이 in 연산자로 검사한다.

```
if "SCOX" in prices:
    p = prices["SCOX"]
else:
    p = 0.0
```

앞에 나온 바로 이 일련의 단계는 다음과 같이 더욱 간결하게 표현할 수 있다.

```
p = prices.get("SCOX", 0.0)
```

사전의 키 목록을 얻고 싶으면 사전을 리스트로 변환하면 된다.

```
syms = list(prices)    # syms = ["AAPL", "MSFT", "IBM", "GOOG"]
```

사전에서 한 요소를 삭제하는 데는 del문을 사용한다.

```
del prices["MSFT"]
```

사전은 파이썬 인터프리터에서 가장 섬세하게 튜닝된 데이터 타입이다. 따라서, 만약 간단히 데이터를 저장하고 작업을 하려는 것이라면 직접 자신만의 데이터 구조를 만들기보다는 사전을 사용하는 것이 대부분의 경우 훨씬 낫다.

반복과 루프

가장 널리 사용되는 루프 관련 구조물은 `for`문이다. `for`문은 항목들의 모임에 대해 반복 수행하는 데 사용된다. 반복은 파이썬에서 가장 풍부한 기능을 제공하는 기능 중 하나다. 가장 널리 사용되는 반복의 형태는 간단히 문자열, 리스트, 튜플 같은 순서열의 모든 구성 요소에 대해 루프를 도는 것이다. 다음은 한 예다.

```
for n in [1,2,3,4,5,6,7,8,9]:
    print "2 to the %d power is %d" % (n, 2**n)
```

앞의 예에서, 변수 `n`에는 각 반복마다 리스트 `[1, 2, 3, 4, ..., 9]`에 있는 다음 항목이 할당된다. 정수 범위에 대해 루프를 도는 일은 상당히 잣으로 다음과 같은 간단한 방법이 제공된다.

```
for n in range(1,10):
    print "2 to the %d power is %d" % (n, 2**n)
```

`range(i, j [, 보폭])` 함수는 `i`에서 `j-1`까지의 값을 가지는 정수 범위를 표현하는 객체를 생성한다. 초기 값이 생략되면 초기 값으로 0이 사용된다. 추가로 세 번째 인수로 보폭을 지정할 수도 있다. 다음은 관련 예다.

```
a = range(5)          # a = 0,1,2,3,4
b = range(1,8)        # b = 1,2,3,4,5,6,7
c = range(0,14,3)     # c = 0,3,6,9,12
d = range(8,1,-1)     # d = 8,7,6,5,4,3,2
```

파이썬 2에서 `range()`와 관련해 한 가지 주의할 것은 이 함수가 모든 값으로 완전히 채워진 리스트를 생성한다는 점이다. 매우 큰 범위를 지정하면 의도하지 않게 메모리를 다 소진할 수도 있다. 따라서, 오래된 파이썬 코드에서는 다음과 같이 대안 함수인 `xrange()`가 사용되는 것을 볼 수 있다. 다음은 한 예다.

```
for i in xrange(100000000):  # i = 0,1,2,...,99999999
```

`xrange()`에 의해 생성되는 객체는 검색이 요청되는 시점에 값을 계산한다. 이러한 이유로 매우 큰 정수 범위를 표현할 때는 `xrange()`를 사용하는 것이 좋다. 파이썬 3에서는 `xrange()` 함수의 이름이 `range()` 함수로 변경되었으며 이전 `range()` 함수

는 제거되었다.

for문은 정수 순서열뿐 아니라 문자열, 리스트, 사전, 파일 등 많은 종류의 객체들에 대해 반복 수행하는 데 사용될 수 있다. 다음은 몇 가지 예다.

```
a = "Hello World"
# a에 있는 각 문자를 출력한다.
for c in a:
    print c

b = ["Dave", "Mark", "Ann", "Phil"]
# 리스트의 구성 요소를 출력한다.
for name in b:
    print name

c = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
# 사전의 모든 구성 요소를 출력한다.
for key in c:
    print key, c[key]

# 파일에 들어 있는 모든 줄을 출력한다.
f = open("foo.txt")
for line in f:
    print line,
```

for문에 값들의 순서열을 제공하는 반복자 객체나 생성기 함수를 직접 만들 수 있기 때문에 for 루프는 파이썬에서 제공하는 가장 강력한 언어 기능 중 하나다. 반복자와 생성기에 관해서는 이 장의 후반부와 6장에서 더 자세히 다룬다.

함수

다음 예처럼 함수는 def문을 사용해 생성한다.

```
def remainder(a,b):
    q = a // b    # //는 꼴수를 버리는 나누기
    r = a - q*b
    return r
```

함수를 호출하려면 result = remainder(37, 15)처럼 함수 이름을 쓰고 이어서 인수들을 괄호로 둘러싸주면 된다. 다음과 같이 튜플을 사용해 여러 값을 반환할 수도 있다.

```
def divide(a,b):
    q = a // b    # a와 b가 모두 정수이면 q는 정수
    r = a - q*b
    return (q,r)
```

튜플로 여러 값을 반환할 때 다음과 같이 그 결과를 쉽게 개별 변수에 담을 수 있다.

```
quotient, remainder = divide(1456,33)
```

함수 인수에 기본 값은 할당하려면 대입문을 사용한다.

```
def connect(hostname,port,timeout=300):  
    # 함수 몸체
```

함수 정의에 기본 값이 주어지면 이어지는 함수 호출에서 해당 값을 생략할 수 있다. 생략된 값은 기본 값을 가지게 된다. 다음은 한 예다.

```
connect('www.python.org', 80)
```

키워드 인수를 사용하면 임의의 순서로 인수들을 넘겨주면서 함수를 호출할 수 있다. 하지만, 이를 위해서 함수 정의에서 사용된 인수의 이름을 알고 있어야 한다. 다음은 한 예다.

```
connect(port=80,hostname="www.python.org")
```

함수 안에서 변수가 생성되거나 값이 대입되면 이 변수의 유효 범위는 지역 범위를 가진다. 즉, 변수는 함수 몸체 안에서만 정의되고 함수가 반환되면 변수는 사라진다. 함수 안에서 전역 변수의 값을 수정하려면 다음과 같이 global문을 사용하면 된다.

```
count = 0  
...  
def foo():  
    global count  
    count += 1 # 전역 변수인 count를 수정
```

생성기

함수는 단일 값을 반환하는 대신, yield문을 사용해 일련의 결과 값을 생성할 수도 있다. 다음은 한 예를 보여준다.

```
def countdown(n):  
    print "Counting down!"  
    while n > 0:  
        yield n      # 값(n)을 생성한다.  
        n -= 1
```

yield를 사용하는 함수를 생성기(generator)라고 부른다. 생성기 함수를 호출하면, next() 메서드(파이썬 3에서는 __next__())를 반복적으로 호출할 때 일련의 결과를 반환하는 객체가 생성된다. 다음은 한 예다.

```
>>> c = countdown(5)
```

```
>>> c.next( )
Counting down!
5
>>> c.next( )
4
>>> c.next( )
3
>>>
```

`next()`를 호출하면, 생성기 함수는 다음 `yield`문에 도달할 때까지 실행된다. 이때, `yield`에 전달된 값이 `next()`에 의해 반환되고 생성기 함수는 실행이 잠시 중지된다. `next()`가 다시 호출되면, 함수가 `yield`문 바로 다음 문장부터 다시 실행된다. 이 과정은 함수가 반환될 때까지 계속된다.

보통 앞에서 본 것처럼 `next()`를 직접 호출하는 일은 거의 없다. 대신 다음과 같이 `for` 루프에 연결시켜 사용한다.

```
>>> for i in countdown(5):
...     print i,
Counting down!
5 4 3 2 1
>>>
```

생성기는 처리 파이프라인, 스트림, 데이터 흐름에 기반한 프로그램을 작성할 때 강력한 위력을 발휘한다. 예를 들어, 다음 생성기 함수는 로그 파일을 검토하는 데 흔히 사용되는 유닉스의 `tail -f` 명령의 동작을 흉내낸다.

```
# 파일의 끝부분을 출력한다(tail -f처럼).
import time
def tail(f):
    f.seek(0,2)                      # 파일 끝으로 이동
    while True:
        line = f.readline()          # 다음 줄 읽기를 시도한다.
        if not line:                 # 아무 것도 없으면 잠시 쉬었다가 다시 시도한다.
            time.sleep(0.1)
            continue
        yield line
```

다음은 일련의 줄들로부터 특정 부분 문자열을 찾는 생성기다.

```
def grep(lines, searchtext):
    for line in lines:
        if searchtext in line: yield line
```

다음은 앞의 두 생성기를 연결시켜 간단한 처리 파이프라인 생성하는 예다.

```
# 유닉스 "tail -f | grep python"의 파이썬 구현
```

```
wwwlog = tail(open("access-log"))
pylines = grep(wwwlog,"python")
```

```
for line in pylines:
    print line,
```

생성기와 관련해서 알아두어야 할 것은 생성기는 보통 리스트나 파일 같은 다른 반복자 객체와 함께 섞일 수 있다는 점이다. 특히, for item in s 같은 문장에서 s는 아이템 리스트, 파일의 줄들, 생성기 함수 결과 또는 반복을 지원하는 어떤 객체든 될 수 있다. s에 어떤 객체든 올 수 있기 때문에 확장성 있는 프로그램 작성이 가능하다.

코루틴

보통 함수는 입력으로 주어진 인수에 대해서 한 번만 실행된다. 하지만, 일련의 입력을 처리하도록 함수를 작성할 수도 있다. 이런 종류의 함수를 코루틴(coroutine)이라고 하고, 다음 예처럼 yield문을 표현식 (yield) 형태로 사용해 생성할 수 있다.

```
def print_matches(matchtext):
    print "Looking for", matchtext
    while True:
        line = (yield)      # 한 줄을 가져온다.
        if matchtext in line:
            print line
```

이 함수를 사용하려면 먼저 호출을 한번 해서 첫 번째 (yield)까지 진행하고 그 다음부터 send()로 데이터를 보내기 시작하면 된다. 다음은 그 예다.

```
>>> matcher = print_matches("python")
>>> matcher.next()    # 첫 번째 (yield)로 진행
Looking for python
>>> matcher.send("Hello World")
>>> matcher.send("python is cool")
python is cool
>>> matcher.send("yow!")
>>> matcher.close()   # 함수 호출 종료
>>>
```

코루틴은 send()로 값이 도착할 때까지 멈춰 있다. send()가 호출되면, 코루틴 안에서 (yield) 표현식에 의해 값이 반환되고 바로 다음 문장에 의해 처리된다. 이러한 처리는 다음 (yield) 표현식을 만날 때까지 계속되고, 그 순간 함수가 다시 멈춘다. 앞의 예에서 보았듯이, 이 과정은 close()가 호출될 때까지 이어진다.

코루틴은 프로그램의 한 곳에서 생성된 데이터를 다른 곳에서 소비하는 생성자 소비자 모델에 기반한 병행 프로그램을 작성할 때 유용하게 쓰인다. 이 모델에서

코루틴은 데이터 소비자의 역할을 수행한다. 다음은 생성기와 코루틴을 결합하는 예다.

```
# 매치기(matcher) 코루틴 집합
matchers = [
    print_matches("python"),
    print_matches("guido"),
    print_matches("jython")
]

# next( )를 호출해 모든 매치기를 준비시킨다.
for m in matchers: m.next()

# 활성 로그 파일을 모든 매치기에 공급한다. 올바른 작동을 위해서는
# 웹 서버가 활발하게 데이터를 로그 파일에 기록해야 한다.
wwwlog = tail(open("access-log"))
for line in wwwlog:
    for m in matchers:
        m.send(line) # 데이터를 각 매치기 코루틴에 보낸다.
```

코루틴에 관해서는 6장에서 더 자세히 설명한다.

객체와 클래스

프로그램 안의 모든 값은 객체다. 객체(object)는 내부 데이터와 내부 데이터와 연관된 다양한 연산을 수행하는 메서드로 구성된다. 앞서 이미 문자열과 리스트 같은 내장 타입을 다루면서 객체와 메서드를 사용해보았다. 다음은 한 예다.

```
items = [37, 42] # 리스트 객체를 생성한다.
items.append(73) # append( ) 메서드를 호출한다.
```

`dir()` 함수는 객체가 제공하는 메서드들을 나열한다. 이 함수는 대화식으로 이것 저것 실험해보는 데 유용하게 쓰인다. 다음은 한 예다.

```
>>> items = [37, 42]
>>> dir(items)
['__add__', '__class__', '__contains__', '__delattr__',
 ...
 'append', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
>>>
```

객체를 살펴볼 때 `append()`나 `insert()` 같은 익숙한 메서드들을 보게 될 것이다. 이 밖에도 항상 이중 밑줄로 시작하고 끝나는 특수한 메서드들이 있다. 이 메서드들은 언어상에서 제공하는 다양한 연산자를 구현한다. 다음 예에서 `__add__()` 메서드는 + 연산자를 구현한다.

```
>>> items.__add__([73,101])
[37, 42, 73, 101]
>>>
```

class문은 새로운 객체 탑입을 정의하며 객체지향 프로그램을 작성하는 데 쓰인다. 예를 들어, 다음 클래스는 push(), pop(), length() 연산자를 갖는 간단한 스택을 정의한다.

```
class Stack(object):
    def __init__(self):      # 스택을 초기화한다.
        self.stack = [ ]
    def push(self,object):
        self.stack.append(object)
    def pop(self):
        return self.stack.pop( )
    def length(self):
        return len(self.stack)
```

클래스 정의의 첫 번째 줄에서 class Stack(object)는 Stack을 object로 선언한다. 파이썬에서는 괄호로 상속 관계를 기술한다. 앞에서는 Stack이 모든 파이썬 탑입의 루트인 object로부터 상속을 받는다. 클래스 정의에서 메서드는 def문으로 정의된다. 각 메서드의 첫 번째 인수는 항상 객체 자기 자신을 가리킨다. 관습적으로 이 인수의 이름으로 self를 사용한다. 객체의 속성과 관련된 연산에서는 명시적으로 self 변수를 참조해야 한다. 시작과 끝이 이중 밑줄인 메서드는 특수 메서드다. 예를 들어, __init__은 객체가 생성된 후 초기화를 위해 사용된다.

클래스는 다음과 같이 사용한다.

```
s = Stack( )          # 스택을 생성한다.
s.push("Dave")       # 스택에 뭔가를 집어넣는다.
s.push(42)
s.push([3,4,5])
x = s.pop( )         # x는 [3,4,5]
y = s.pop( )         # y는 42
del s               # s를 파괴한다.
```

앞의 예에서 스택을 구현하기 위해 완전히 새로운 객체를 생성했다. 하지만, 스택은 내장 리스트 객체와 거의 동일하다. 따라서, 다음과 같이 list에서 상속 받고 메서드를 하나 추가해서 스택을 구현할 수도 있다.

```
class Stack(list):
    # 스택 인터페이스에 push( ) 메서드를 추가한다.
    # 주의: 리스트에는 이미 pop( ) 메서드가 있다.
    def push(self,object):
        self.append(object)
```

보통 클래스 안에서 정의된 모든 메서드는 클래스의 인스턴스에만 적용된다(즉, 클래스로부터 생성된 객체에 적용된다). 하지만, C++나 자바 프로그래머에게 익숙한 정적 메서드 같은 다른 종류의 메서드도 정의할 수 있다. 다음은 한 예다.

```
class EventHandler(object):
    @staticmethod
    def dispatcherThread( ):
        while (1):
            # 요청을 기다린다.
            ...

EventHandler.dispatcherThread( )    # 메서드를 함수처럼 호출한다.
```

여기서, `@staticmethod`는 바로 다음에 나오는 메서드가 정적 메서드라는 것을 선언한다. `@staticmethod`는 6장에서 더 자세히 설명할 장식자(decorator)의 한 예다.

예외

프로그램에서 에러가 발생하면 예외가 발생하고 다음과 같이 역추적 메시지가 나타난다.

```
Traceback (most recent call last):
  File "foo.py", line 12, in <module>
    IOError: [Errno 2] No such file or directory: 'file.txt'
```

역추적 메시지는 발생된 에러의 종류와 위치를 표시한다. 보통 에러는 [프로그램](#)을 종료시킨다. 하지만, 다음과 같이 `try`와 `except`문으로 예외를 잡아서 처리할 수도 있다.

```
try:
    f = open("file.txt","r")
except IOError as e:
    print e
```

`IOError`가 발생하면, 에러의 원인에 대한 내용이 `e`에 담기고 `except` 블록으로 제어가 넘어간다. 또 다른 예외가 발생하면 에워싼 코드 블록(존재하는 경우)으로 제어가 넘어간다. 아무런 에러도 발생하지 않으면 `except` 블록에 있는 코드는 무시된다. 예외가 처리되면 최종 `except` 블록 바로 뒤이어 나오는 문장부터 [프로그램](#) 실행이 다시 시작된다. [프로그램](#)은 예외가 발생된 시점으로 되돌아가지 않는다.

`raise`문은 예외 발생을 알리는 데 사용된다. 예외를 발생시킬 때, 다음과 같이 내장 예외 중 하나를 사용할 수 있다.

```
raise RuntimeError("Computer says no")
```

또는 5장의 ‘새로운 예외 정의’ 절에서 설명된 것처럼 여러분만의 예외를 정의할 수도 있다.

락(lock), 파일, 네트워크 연결 같은 시스템 자원을 적절히 관리하는 일은 예외 처리와 맞물릴 경우 까다로운 일이 된다. 이러한 일을 수월하게 만들어주기 위해 특정한 객체에 대해서는 `with`문을 사용할 수 있다. 다음은 뮤텍스 락(mutex lock)을 사용하는 코드를 작성하는 예이다.

```
import threading
message_lock = threading.Lock( )
...
with message_lock:
    messages.add(newmessage)
```

이 예에서 `with`문이 실행될 때 `message_lock` 객체가 자동으로 획득된다. 실행이 `with` 블록의 컨텍스트를 벗어나는 순간 락이 자동으로 해제된다. 이러한 관리는 `with` 블록 안에서 무슨 일이 벌어지든지 이루어진다. 예를 들어, 예외가 발생하면 제어가 블록의 컨텍스트 밖으로 나가는 순간 락이 해제된다.

`with`문은 보통 파일, 연결, 락 같은 시스템 자원이나 실행 환경과 관련된 객체와 호환된다. 하지만, 사용자 정의 객체를 통해 자신만의 처리 방식을 정의할 수도 있다. 여기에 관해서는 3장 ‘컨텍스트 관리 프로토콜’ 절에서 더 자세히 설명한다.

모듈

프로그램 크기가 커지면 손쉬운 관리를 위해 프로그램을 여러 파일로 나누고 싶어질 것이다. 파이썬에서는 정의들을 파일에 넣어 다른 프로그램이나 스크립트에 임포트(import)할 수 있는 모듈의 형태로 사용할 수 있다. 모듈을 생성하려면 관련 문장과 정의들을 모듈과 동일한 이름을 가지는 파일에 넣으면 된다(파일 확장자가 .py이어야 한다). 다음은 한 예다.

```
# 파일 : div.py
def divide(a,b):
    q = a/b          # a와 b가 모두 정수면 q는 정수다.
    r = a - q*b
    return (q,r)
```

이 모듈을 다른 프로그램에서 사용하려면 `import`문을 사용하면 된다.

```
import div
```

```
a, b = div.divide(2305, 29)
```

import문은 새로운 네임스페이스를 생성하고 .py 파일 안에 있는 모든 문장을 이 네임스페이스 안에서 실행한다. 해당 네임스페이스 안에 있는 내용에 접근하려면 앞의 예의 div.divide()처럼 간단히 모듈의 이름을 앞에 붙여주면 된다.

모듈을 다른 이름으로 임포트하고 싶으면 다음과 같이 import문에 선택적인 as 한정어를 써주면 된다.

```
import div as foo
a,b = foo.divide(2305,29)
```

특정 정의를 현재 네임스페이스에 임포트하려면 from문을 사용하면 된다.

```
from div import divide
a,b = divide(2305,29)      # div 접두어가 더 이상 필요 없음
```

모듈의 모든 내용을 현재 네임스페이스로 가져오려면 다음과 같이 하면 된다.

```
from div import *
```

객체를 다룰 때와 동일하게 dir() 함수를 쓰면 모듈의 내용을 볼 수 있고 대화식으로 이것저것 실험해볼 수 있다.

```
>>> import string
>>> dir(string)

['__builtins__', '__doc__', '__file__', '__name__', '__idmap',
 '__idmapL', '__lower__', '__swapcase', '__upper__', 'atof', 'atof_error',
 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',
 ...
>>>
```

도움 얻기

파이썬으로 작업할 때 필요한 정보를 빠르게 얻을 수 있는 방법이 몇 가지 있다. 먼저, 파이썬을 대화식 모드로 실행하고 있을 때는 help() 명령을 사용해서 내장 모듈이나 기타 파이썬의 기능에 관한 정보를 얻을 수 있다. 일반적인 정보를 얻으려면 간단히 help()만 입력하고 특정 모듈에 대한 정보를 얻으려면 help('모듈이름')을 입력한다. help() 명령은 함수 이름을 받으면 함수에 대한 정보를 반환한다.

파이썬 함수는 대부분 사용법을 설명하는 문서화 문자열을 지닌다. 이를 출력하려면 간단히 __doc__ 속성을 출력하면 된다. 다음은 한 예다.

```
>>> print issubclass.__doc__  
issubclass(C, B) -> bool  
  
Return whether class C is a subclass (i.e., a derived class) of  
class B.  
When using a tuple as the second argument issubclass(X, (A, B, ...)),  
is a shortcut for issubclass(X, A) or issubclass(X, B) or ... (etc.).  
>>>
```

마지막으로 대부분의 파이썬 설치본은 파이썬 모듈에 대한 문서화를 얻는 데 사용할 수 있는 pydoc 명령을 제공한다. 간단히 시스템 명령 프롬프트에서 pydoc 주제를 입력하면 된다.

2장

P y t h o n E s s e n t i a l R e f e r e n c e

어휘 규약과 구문

이 장에서는 파이썬 프로그램의 구문과 어휘 규약을 설명한다. 줄 구조, 문장 그룹, 예약어, 상수, 연산자, 토큰, 소스 코드 인코딩 등의 주제를 다룬다.

줄 구조와 들여쓰기

프로그램에서 각 문장은 줄바꿈 문자(newline)로 끝난다. 긴 문장은 다음 예처럼 줄이음 문자(\)를 사용하여 여러 줄로 작성할 수 있다.

```
a = math.cos(3 * (x - n)) + \
    math.sin(3 * (y - n))
```

삼중따옴표로 둘러싸인 문자열, 리스트, 튜플, 사전 정의가 여러 줄에 걸쳐 있는 경우에는 줄이음 문자가 필요 없다. 더 일반화해서 말하면, 프로그램에서 팔호 (...), 대괄호 [...], 중괄호 (...)나 삼중따옴표로 둘러싸인 부분은 시작과 끝이 명확히 구분되므로 줄이음 문자 없이도 여러 줄에 걸쳐 있을 수 있다.

들여쓰기는 함수 몸체, 조건문, 루프, 클래스 등 다양한 코드 블록을 나타낸다. 한 블록 안에서 첫 번째 문장에 적용되는 들여쓰기에는 제한이 없지만, 블록 전체로 봤을 때는 일관성이 있어야 한다. 다음은 관련 예를 보여준다.

```
if a:
    statement1      # 일관된 들여쓰기
    statement2
else:
    statement3
    statement4    # 일관되지 않은 들여쓰기 (에러)
```

함수, 조건문, 루프, 클래스의 몸체가 오직 한 문장만 담는다면, 다음과 같이 같은 줄에 쓸 수 있다.

```
if a: statement1  
else: statement2
```

빈 몸체나 블록은 pass문으로 나타낸다. 다음은 한 예다.

```
if a:  
    pass  
else:  
    statements
```

들여쓰기로 탭을 사용할 수 있지만 권장하지 않는다. 스페이스를 사용하는 것이 파이썬 프로그램 커뮤니티에서 보편적으로 선호되는(그리고 권장되는) 방식이다. 탭 문자를 만나면, 8의 배수인 다음 열까지 이동하도록 스페이스가 추가된다(예를 들어, 열 11에 있는 탭은 열 16으로 이동할 정도의 스페이스가 추가된다). -t 옵션으로 파이썬을 실행하면, 동일한 프로그램 블록 안에서 탭과 스페이스가 일관성 없이 사용된 경우 경고 메시지를 출력한다. -tt 옵션은 이 경고 메시지를 TabError 예외로 변환한다.

한 줄에 둘 이상의 문장을 작성하려면, 문장들을 세미콜론(:)으로 구분하면 된다. 불필요하긴 하지만, 한 문장으로 된 줄도 세미콜론으로 끝낼 수 있다.

#는 줄의 끝까지 이어지는 주석을 표시한다. 따옴표로 둘러싸인 문자열 안에 있는 #는 주석으로 취급되지 않는다.

마지막으로, 인터프리터는 대화식 모드에서 수행될 때를 제외하고는 빈 줄을 모두 무시한다. 대화식 모드에서는 여러 줄에 걸친 문장을 입력할 때 빈 줄이 입력의 끝을 알리는 역할을 한다.

식별자와 예약어

식별자(identifier)는 변수, 함수, 클래스, 모듈 및 기타 객체를 식별하는 데 사용되는 이름이다. 식별자의 이름은 문자, 숫자, 밑줄(_)을 포함할 수 있지만, 항상 숫자 아닌 문자로 시작되어야 한다. 문자로는 현재 ISO-Latin 문자 집합에 속하는 A-Z와 a-z만 쓸 수 있다. 식별자에서는 대소문자를 구별하므로 FOO는 foo와 서로 다른 식별자이다. \$, %, @ 같은 특수한 기호는 식별자로 쓸 수 없다. 추가적으로, if, else, for 같은 단어는 예약어로 지정되어 있어 식별자 이름으로 쓰일 수 없다. 다음 목록

은 모든 예약어를 나열한 것이다.

```
and      del      from     nonlocal   try
as       elif     global    not        while
assert   else     if        or         with
break    except   import   pass       yield
class    exec    in        print
continue finally is        raise
def      for     lambda  return
```

이름이 밑줄로 시작하거나 끝나는 식별자는 보통 특수한 의미를 지닌다. 예를 들어, `_foo`처럼 단일 밑줄로 시작하는 식별자는 `from module import *` 문에 의해 임포트되지 않는다. `__init__` 같이 이중 밑줄로 시작하고 끝나는 식별자는 특수한 메서드용으로 예약되어 있으며, `__bar`처럼 이중 밑줄로 시작하는 식별자는 7장에 설명되어 있는 것처럼 private 클래스 멤버를 구현하는 데 사용된다. 이러한 식별자들은 일반적인 용도로 사용하지 말아야 한다.

숫자 상수

내장 숫자 상수(numeric literal)에는 네 종류가 있다.

- 불리언
- 정수
- 부동 소수점 수
- 복소수

식별자 `True`와 `False`는 각각 정수 값 1과 0을 가지는 불리언 값으로 해석된다. 1234 같은 수는 십진수로 해석된다. 정수를 8진법, 16진법, 2진법으로 표기 하려면, 각각 `0o`, `0x`, `0b`를 값 앞에 붙여주면 된다.(예를 들어, `0o644`, `0x100fea8`, `0b11101010`)

파이썬에서 정수는 임의의 개수의 숫자를 가질 수 있다. 매우 큰 정수를 표현하려면 12345678901234567890처럼 간단히 모든 숫자를 나열하면 된다. 하지만, 오래된 파이썬 코드에서 값을 살펴보면, 12345678901234567890L처럼 큰 수 뒤에 `l`(소문자 L)이나 `L` 문자가 붙어 있는 것을 종종 볼 수 있다. 여기서 `L`이 사용되는 이유는 파이썬이 내부적으로 값의 크기에 따라 해당 머신에서 사용되는 고정된 정밀도의 정수형 또는 임의 정밀도를 가지는 긴 정수형 중 하나로 표현하기 때문이다. 오래된 버전의 파이썬에서는 둘 중 어느 것을 사용할지 선택할 수 있었고 긴 정수형을

사용하기 위해 끝에 L을 붙일 수도 있었다. 현재는 이러한 구분이 불필요하며 더 이상 사용을 권장하지 않는다. 따라서, 큰 정수 값을 나타내려면 L을 붙이지 말고 쓰기 바란다.

123.34와 1.2334e+02 같은 수는 부동 소수점 수로 표현된다. 뒤에 J나 j가 붙은 정수나 부동 소수점 수는 허수를 나타낸다. 1.2 + 12.34j처럼 실수와 허수를 더해 실수부와 허수부를 가진 복소수를 생성할 수 있다.

문자열 상수

문자열 상수는 문자들의 순서열을 나타내는데 사용되며, 작은따옴표('), 큰따옴표 ("')나 삼중따옴표("""나 """)로 둘러싸서 정의한다. 시작과 끝에 동일한 종류의 따옴표를 사용해야 한다는 점을 제외하고는 서로 다른 따옴표 간에 의미 차이는 없

표 2.1 표준 문자 탈출 코드

문자	설명
\	줄바꿈 이음
\\\	역슬래시
\'	작은따옴표
\"	큰따옴표
\a	벨소리 문자
\b	백스페이스
\e	탈출(escape) 문자
\0	널(null)
\n	줄바꿈(line feed) 문자
\v	수직 탭
\t	수평 탭
\r	캐리지 리턴(carriage return)
\f	폼 피드(form feed)
\ooo	8진수 (\000에서 \377까지)
\xxxxx	유니코드 문자 (\u0000에서 \uffff까지)
\xxxxxxxx	유니코드 문자 (\U00000000에서 \Uffffffffff까지)
\N{문자이름}	유니코드 문자 이름
\xhh	16진수 (\x00에서 \xff까지)

다. 작은따옴표와 큰따옴표 문자는 반드시 한 줄 안에 있어야 하지만, 삼중따옴표 문자열은 여러 줄에 걸쳐서 존재할 수 있으며 내부의 포맷 문자(줄바꿈, 템, 스페이스 등)들을 그대로 담는다. “hello” ‘world’처럼 스페이스, 줄바꿈, 줄이음 문자로 구분되는 인접한 문자열은 연결되어서 “helloworld”처럼 단일 문자열이 된다.

문자열 상수 안에서 역슬래시(\) 문자는 줄바꿈, 역슬래시 자체, 따옴표, 비출력 문자 같은 특수한 문자를 탈출(escape)시키는 데 사용된다. 표 2.1은 허용되는 탈출 코드들을 보여준다. 식별이 안되는 탈출 순서열(escape sequence)은 문자열 안에서 수정되지 않은 채로 남겨지고 첫 역슬래시 문자도 같이 남겨진다.

탈출 코드인 \ooo와 \x는 쉽게 입력할 수 없는(즉, 제어 코드, 비출력 문자, 기호, 국제 문자 등) 문자를 상수 안에 집어넣는 데 사용된다. 이 두 탈출 코드에 대해서는 문자 값에 해당하는 정수를 지정해 주어야 한다. 예를 들어, 단어 “Jalapeño”에 대한 문자열 상수를 생성하려면, “Jalape\xf1o”라고 입력해야 한다. 여기서 xf1은 문자 ñ에 대한 문자 코드다.

파이썬 2에서 문자열 상수는 8비트 문자나 바이트 지향적인 데이터에 대응된다. 이 때문에 국제 문자 집합과 유니코드를 완벽하게 지원하지 못하는 심각한 제약이 있었다. 이러한 제약을 극복하기 위해 파이썬 2에서는 유니코드 데이터에 대해 별개의 문자열 타입을 사용한다. 유니코드 문자열 상수를 작성하려면 첫 번째 따옴표 앞에 문자 “u”를 써주면 된다. 다음은 한 예다.

```
s = u"Jalape\u00f1o"
```

파이썬 3에서는 이러한 접두어가 불필요하다(실제로 구문 오류를 발생시킨다). 파이썬 2에서도 인터프리터를 -U 옵션으로 실행하면 이를 흉내낸다(모든 문자열 상수를 유니코드로 취급하며 u를 생략할 수 있다).

사용 중인 파이썬 버전에 상관없이, 표 2.1에 나와 있는 \u, \U와 \N를 사용하면 임의의 문자를 유니코드 상수에 넣을 수 있다. 모든 유니코드 문자에는 코드 포인트(code point)가 할당된다. 코드 포인트는 유니코드 차트에서 보통 U+XXXX로 표기되는데, XXXX는 네 개 혹은 그 이상의 16진수 숫자들의 순서열을 의미한다.(이 표기법은 파이썬 문법에 속하는 것이 아니라, 유니코드 문자를 설명할 때 관련 서적의 저자들이 주로 사용하는 표기법이다.) 예를 들어, ñ는 코드 포인트 U+00F1을 지닌다. \u 탈출 코드는 U+0000에서 U+FFFF까지 코드 포인트를 갖는 유니코드 문자를 입력하는 데 사용된다(예를 들어, \u00f1). \U 탈출 코드는 U+10000과 그 위

쪽에 속하는 범위의 문자를 입력하는 데 사용된다(예를 들어, \U00012345). \U 탈출 코드와 관련해서 주의해야 할 점은 U+10000를 넘어서는 코드 포인트를 지닌 유니코드 문자는 보통 대리자 쌍(surrogate pair)이라고 불리는 문자 쌍으로 나누어진다는 점이다. 이것은 유니코드 문자열의 내부 표현 방식과 관련이 있으며 3장 ‘타입과 객체’에서 더 자세하게 다룬다.

유니코드 문자는 서술적 이름(descriptive name)도 갖는다. 이 이름을 알면, \N{문자이름} 탈출 순서열을 사용할 수 있다. 다음은 한 예다.

```
s = u"Jalape\N{LATIN SMALL LETTER N WITH TILDE}o"
```

<http://www.unicode.org/charts>에서 코드 포인트와 문자 이름에 관한 정식 자료를 얻을 수 있다.

선택적으로, r'\d'처럼 문자열 상수 앞에 r이나 R을 붙일 수 있다. 이러한 문자열은 문자열 안에 있는 역슬래시가 그대로 남겨지기 때문에 미가공 문자열(raw string)이라고 부른다. 즉, 문자열이 안에 들어 있는 역슬래시를 포함한 문자 그대로의 텍스트를 담게 된다. 미가공 문자열은 주로 역슬래시 문자가 중요한 의미를 가지는 문자열 상수를 생성할 때 사용된다. 그러한 예에는 re 모듈로 정규 표현식 패턴을 기술하거나 윈도에서 파일 이름을 지정할 때(예를 들어, r'c:\newdata\tests') 등이 있다.

미가공 문자열은 r\"처럼 단일 역슬래시 문자로 끝날 수 없다. 미가공 문자열 안에서 \uXXXX 탈출 순서열은 앞에 붙는 \ 문자가 홀수 개인 경우에 한해서 유니코드로 문자로 해석된다. 예를 들어, ur"\u1234"는 단일 문자 U+1234를 담는 미가공 유니코드 문자열을 나타내지만, ur"\\"u1234"는 첫 두 문자가 역슬래시이고 나머지 다섯 글자가 상수 "u1234"인 일곱 글자로 구성되는 문자열을 나타낸다. 또한 파이썬 2.2에서는 앞에서 보았듯이, 미가공 유니코드 문자열에는 u 다음에 r을 반드시 써주어야 한다. 파이썬 3.0에서는 u를 앞에 붙일 필요가 없다.

UTF-8이나 UTF-16 같은 데이터 인코딩 방식에 대응되는 미가공 바이트 순서열로 문자열을 표기하면 안 된다. 예를 들어, 'Jalape\xc3\xb1o'처럼 미가공 UTF-8을 직접 지정하면, 아마도 의도하지 않았을 아홉 글자 문자열인 U+004A, U+0061, U+006C, U+0061, U+0070, U+0065, U+00C3, U+00B1, U+006F이 만들어진다. UTF-8에서 다중 바이트 순서열인 \xc3\xb1은 단일 문자 U+00F1를 나타내야 함에도 U+00C3, U+00B1인 두 문자로 취급되었다. 인코딩된 바이트 문자열을 상수로 만들

기 위해서는 b“Jalape\xc3\xb1o”처럼 첫 따옴표 앞에 “b”를 붙여주면 된다. 이렇게 정의하면 문자 그대로 단일 바이트들로 구성되는 문자열이 만들어진다. 바이트 상수의 값을 바이트 상수의 decode() 메서드로 디코딩하면 보통 문자열을 만들 수 있다. 여기에 관해서는 3장과 4장 ‘연산자와 표현식’에서 더 자세히 다룬다.

파이썬 2.6이 되어서야 이러한 문법이 생겼기 때문에 바이트 상수는 프로그램에서 거의 사용되지 않으며 하지만 파이썬 2.6에서는 바이트 상수와 보통의 문자열 사이에 구분이 없다. 하지만, 파이썬 3에서는 바이트 상수는 보통의 문자열과는 다르게 작동하는 새로운 bytes 데이터 타입에 대응된다(부록 ‘파이썬 3’ 참조).

컨테이너

다음 예에서 볼 수 있듯이, 중괄호 [...], 괄호 (...), 대괄호 (...)로 둘러싸인 값들은 각각 리스트, 튜플, 사전에 들어 있는 객체들의 모임을 나타낸다.

```
a = [ 1, 3.4, 'hello' ]      # 리스트
b = ( 10, 20, 30 )          # 튜플
c = { 'a': 3, 'b': 42 } # 사전
```

리스트, 튜플, 사전 상수은 줄이음 문자(\) 없이도 여러 줄에 걸쳐 있을 수 있다. 또한, 마지막 항목에 콤마를 붙여도 된다.

```
a = [
    1,
    3.4,
    'hello',
]
```

연산자, 구분 문자, 특수 기호

파이썬은 다음과 같은 연산자들을 사용한다.

+	-	*	**	/	//	%	<<	>>	&	
^	~	<	>	<=	>=	==	!=	<>	+=	
-=	*=	/=	//=	%=	**=	&=	=	^=	>>=	<<=

다음 토큰들은 표현식, 리스트, 사전 및 기타 문장들에 대한 구분자(delimiter)의 역할을 한다.

```
( ) [ ] { } , : . ` = ;
```

예를 들어, 등호(=) 문자는 대입문에서 이름과 값을 구분하는 역할을 하고 콤마(,) 문자는 함수의 인수들, 리스트와 튜플의 요소 등을 구분짓는 데 사용된다. 마침

표(.)는 부동 소수점 수나 확장 분할 연산에 사용되는 생략 부호(...)에 사용된다.

마지막으로 다음의 특수한 기호들도 사용된다.

' " # \ @

\$와 ? 문자는 파이썬에서 특별한 의미를 갖지 않으며 따옴표로 둘러싸인 문자열 상수 안을 제외하고는 프로그램에서 사용하면 안 된다.

문서화 문자열

다음 예에서 보듯이 모듈, 클래스, 함수 정의의 첫 번째 문장이 문자열이면, 이 문자열은 연관된 객체의 문서화 문자열이 된다.

```
def fact(n):
    "This function computes a factorial"
    if (n <= 1): return 1
    else: return n * fact(n - 1)
```

문서화 문자열은 코드 브라우저나 문서화 생성 도구에서 흔히 사용된다. 문서화 문자열에는 객체의 `__doc__` 속성으로 접근할 수 있다.

```
>>> print fact.__doc__
This function computes a factorial
>>>
```

어떤 정의 안에서 문서화 문자열의 들여쓰기는 다른 문장들과 일관성이 있어야 한다. 또한, 문서화 문자열은 변수로부터 표현식의 형태로 계산되거나 할당될 수 없다. 문서화 문자열은 항상 따옴표로 둘러싸인 문자열 상수의 형태여야 한다.

장식자

함수, 메서드, 클래스 정의 앞에 특수한 문자가 올 수 있는데 이를 장식자(decorator)라고 한다. 장식자는 바로 따라 나오는 정의의 작동 방식을 변경하는 데 쓰인다. 기호 `@`로 장식자를 표시하며 장식자는 별개의 줄로서 관련 함수, 메서드, 클래스 바로 앞에 나와야 한다.

```
class Foo(object):
    @staticmethod
    def bar( ):
        pass
```

하나 이상의 장식자를 사용할 수 있으며, 장식자들은 각각 별개의 줄에 있어야

한다.

```
@foo
@bar
def spam( ):
    pass
```

장식자에 대해서는 6장과 7장에서 더 자세하게 설명한다.

소스 코드 인코딩

파이썬의 소스 프로그램은 보통 표준 7비트 ASCII로 작성된다. 하지만, 사용자가 유니코드 환경에서 작업하고 있으면 이 점이 약간 성가실 수 있다. 특히 국제 문자를 포함하는 문자열 상수를 많이 입력해야 하는 경우에 더욱 그렇다.

파이썬 프로그램의 첫 두 줄에 특수한 인코딩 주석을 입력하면 다른 인코딩으로 파이썬 소스 코드를 작성할 수 있다.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

s = "Jalapeño"      # 따옴표 안의 문자열이 그대로 UTF-8로 인코딩됨.
```

coding: 주석이 있으면, 유니코드를 인식하는 편집기를 사용해 문자열 상수를 바로 입력할 수 있다. 하지만, 식별자 이름과 예약어 같은 파이썬의 다른 요소들은 여전히 ASCII 문자로만 입력해야 한다.

3장

Python Essential Reference

타입과 객체

파이썬 프로그램에서 모든 데이터는 객체(object)라는 개념을 사용하여 저장된다. 가장 기본이 되는 데이터 타입인 숫자, 문자열, 리스트, 사전은 다 객체이다. 클래스를 사용해 사용자 정의 객체를 생성할 수도 있다. 또한, 프로그램의 구조와 인터프리터의 내부 동작과 관련된 객체들도 있다. 이 장에서는 파이썬 객체 모델의 내부적인 작동 방식을 알아보고 내장 데이터 타입들을 개략적으로 살펴본다. 연산자와 표현식에 관해서는 4장에서 더 자세히 다룬다. 7장에서는 사용자 정의 객체를 어떻게 생성하는지 설명한다.

용어

프로그램에서 저장되는 모든 데이터는 객체이다. 각 객체는 신원(identity), 타입(클래스라고도 함)과 값을 가진다. 예를 들어, `a = 42`라고 쓰면 42라는 값을 갖는 정수 객체가 생성된다. 객체의 신원은 객체가 메모리에 저장된 위치를 가리키는 포인터라고 생각할 수 있다. `a`는 그 위치를 가리키는 이름이다.

객체의 타입(클래스라고도 부른다)은 객체의 내부적인 표현 형태와 객체가 지원하는 메서드 및 연산들을 설명한다. 특정 타입의 객체가 생성되면 그 객체를 그 타입의 인스턴스(instance)라고 부르기도 한다. 인스턴스가 일단 생성되면 그 인스턴스의 신원과 타입을 변경할 수 없다. 객체의 값을 변경할 수 있으면 그 객체는 변경 가능(mutable)하다라고 한다. 값을 변경할 수 없으면 변경 불가능(immutable)하다라고 한다. 다른 객체에 대한 참조들을 담는 객체를 컨테이너(container) 혹은 컬렉

션(collection)이라고 부른다.

객체는 대부분 몇 개의 속성과 메서드로 특징지어진다. 속성(attribute)은 객체에 연결된 값이다. 메서드(method)는 호출될 때 객체에 대해 특정 연산을 수행하는 함수이다. 다음 예에서 보듯이, 속성과 메서드에는 점(.) 연산자를 사용해서 접근할 수 있다.

```
a = 3 + 4j    # 복소수를 생성한다.
r = a.real    # 실수부(속성)를 얻는다.

b = [1, 2, 3]    # 리스트를 생성한다.
b.append(7)      # append 메서드로 새로운 원소를 추가한다.
```

객체 신원과 타입

내장 함수인 `id()`는 객체의 신원을 나타내는 정수를 반환한다. 보통 이 정수는 객체의 메모리 상의 위치를 나타낸다. 그러나, 이 부분은 파이썬 구현에 따라 달라질 수 있기 때문에 신원을 이런 식으로 해석하지 않는 것이 좋다. `is` 연산자는 두 객체의 신원을 비교한다. 다음은 두 객체를 비교하는 여러 가지 방법을 보여준다.

```
# 두 객체를 비교한다
def compare(a,b):
    if a is b:
        # a와 b는 동일한 객체이다.
        문장들
    if a == b:
        # a와 b는 동일한 값을 갖는다.
        문장들
    if type(a) is type(b):
        # a와 b는 동일한 타입이다.
        문장들
```

객체의 타입은 객체의 클래스라고 하고 그 자체도 객체이다. 이 객체는 고유하게 정의되기 때문에 주어진 타입의 모든 인스턴스에 대해서 항상 동일하다. 따라서, 타입은 `is` 연산자로 비교할 수 있다. 모든 타입 객체는 타입 검사에 쓰일 수 있는 이름을 갖는다. 이 이름들은 `list`, `dict`, `file`처럼 대부분 내장된 이름이다. 다음은 관련 예를 보여준다.

```
if type(s) is list:
    s.append(item)

if type(d) is dict:
    d.update(t)
```

타입은 클래스를 통해 특수화(specialization)가 가능하기 때문에 내장 함수인

`isinstance(object, type)`를 사용하여 타입을 검사하는 것이 더 낫다. 다음은 한 예다.

```
if isinstance(s, list):
    s.append(item)

if isinstance(d, dict):
    d.update(t)
```

`isinstance()` 함수는 상속 관계를 인식하기 때문에 어떤 파이썬 객체의 타입을 검사하든지 이 함수를 사용하는 것이 좋다.

프로그램에서 타입 검사를 수행할 수 있지만 종종 타입 검사는 여러분이 생각하는 것만큼 유용하지는 않다. 일단 과도한 타입 검사는 성능을 크게 저하시킨다. 그리고 프로그램에서는 항상 상속 계층에 꼭 들어맞도록 객체를 정의하지 않는다. 예를 들어, 앞서 나왔던 문장 `isinstance(s, list)`를 s가 ‘리스트와 유사한지’를 검사하기 위해 사용했다면 리스트와 동일한 인터페이스를 지니지만 내장 타입 `list`에서 직접 상속받지 않은 타입에 대해서는 제대로 작동하지 않을 것이다. 추상 기반 클래스를 정의함으로써 타입 검사를 수행하는 방법도 있다. 여기에 관해서는 7장에서 자세하게 다룬다.

참조 횟수와 쓰레기 수집

모든 객체에는 참조 횟수(reference count)가 유지된다. 다음과 같이 객체가 새로운 이름에 대입되거나 리스트, 튜플, 사전 같은 컨테이너에 추가될 때 참조 횟수가 하나 증가한다.

```
a = 37      # 값 37을 가지는 객체를 생성한다.
b = a      # 37에 대한 참조 횟수를 증가시킨다.
c = []
c.append(b)  # 37에 대한 참조 횟수를 증가시킨다.
```

앞의 예에서 값 37을 담는 객체가 하나 만들어졌다. `a`는 단순히 새로 생성된 객체를 가리키는 이름일 뿐이다. `b`에 `a`가 대입되면 `b`는 동일한 객체에 대한 새로운 이름이 되고 객체의 참조 횟수가 하나 증가한다. 비슷하게 `b`를 리스트에 넣으면 객체의 참조 횟수가 다시 하나 증가한다. 앞의 예에서 오직 하나의 객체만이 37을 담고 있다. 다른 모든 연산은 단순히 그 객체에 대한 새로운 참조를 생성할 뿐이다.

객체의 참조 횟수는 `del`문이 사용되거나 참조가 유효 범위를 벗어날 경우(혹은 재할당될 경우) 하나 감소한다. 다음은 앞의 예에서 이어지는 예이다.

```
del a      # 37에 대한 참조 횟수가 하나 감소한다.
```

```
b = 42      # 37에 대한 참조 횟수가 하나 감소한다.
c[0] = 2.0  # 37에 대한 참조 횟수가 하나 감소한다.
```

객체의 현재 참조 횟수는 `sys.getrefcount()` 함수로 얻을 수 있다.

```
>>> a = 37
>>> import sys
>>> sys.getrefcount(a)
7
>>>
```

참조 횟수 값은 예상보다 훨씬 큰 경우가 많다. 인터프리터는 숫자나 문자열 같은 변경 불가능한 객체에 대해서 사용되는 메모리를 절약하기 위해서 이들을 프로그램의 여러 곳에서 최대한 공유한다.

객체의 참조 횟수가 0이 되면 쓰레기 수집(garbage collection)이 수행된다. 종종 더 이상 사용되지 않는 객체들 간에 순환 의존성(circular dependency)이 존재하는 경우가 있다. 다음은 한 예를 보여준다.

```
a = { }
b = { }
a['b'] = b    # a는 b에 대한 참조를 담고 있다
b['a'] = a    # b는 a에 대한 참조를 담고 있다
del a
del b
```

이 예에서 `del`문은 `a`와 `b`의 참조 횟수를 하나씩 줄이고 내부 객체들을 가리키는 이름들을 파괴한다. 그러나 둘 다 서로에 대한 참조를 갖고 있기 때문에 참조 횟수가 0이 되지 못하고 이들은 할당된 채로 그대로 남겨지게 된다(메모리 누수를 일으킨다). 이러한 문제를 해결하기 위해서 인터프리터는 주기적으로 접근 불가능한 객체들의 순환 참조를 감지하는 순환 참조 감지기(cycle detector)를 구동한다. 인터프리터가 실행 중에 메모리를 더 많이 쓰게 됨에 따라 순환 참조 감지 알고리즘이 주기적으로 실행된다. 쓰레기 수집이 작동하는 방식은 `gc` 모듈에 있는 함수들을 사용하여 세세하게 조정, 제어할 수 있다.(13장 참고)

참조와 복사

프로그램에서 `a = b` 같은 대입이 이루어지면 `b`에 대한 새로운 참조가 생성된다. 앞의 대입이 숫자나 문자열 같은 변경 불가능한 객체에 대해서 수행되면 `b`에 대한 복사본이 생성되는 것처럼 작동한다. 리스트나 사전 같은 변경 가능한 객체에 대해서 대입이 이루어질 때는 변경 불가능한 객체의 경우와는 다소 다른 결과가 나타난다.

다음은 한 예다.

```
>>> a = [1,2,3,4]
>>> b = a          # b는 a에 대한 참조이다.
>>> b is a
True
>>> b[2] = -100   # b에 있는 한 원소를 변경
>>> a            # a도 변경된 것에 주목
[1, 2, -100, 4]
>>>
```

앞의 예에서 a와 b가 동일한 객체를 참조하기 때문에 두 변수 중 하나에 가해진 변화가 다른 변수에서도 보인다. 이를 방지하려면 객체에 대한 참조가 아니라 복사본을 생성해야 한다.

리스트나 사전 같은 컨테이너 객체에 적용되는 복사 연산에는 얕은 복사 두 가지가 있다. 얕은 복사(shallow copy)는 새로운 객체를 생성하지만 그 안은 원래 객체에 들어 있던 참조로 채워진다. 다음은 한 예다.

```
>>> a = [ 1, 2, [3,4] ]
>>> b = list(a)      # a에 대한 얕은 복사본을 생성
>>> b is a
False
>>> b.append(100)    # b에 새로운 원소를 하나 추가
>>> b
[1, 2, [3, 4], 100]
>>> a            # a가 변하지 않은 것에 주목
[1, 2, [3, 4]]
>>> b[2][0] = -100  # b에 들어 있는 원소를 변경
>>> b
[1, 2, [-100, 4], 100]
>>> a            # a도 변한 것에 주목
[1, 2, [-100, 4]]
>>>
```

앞의 예에서 a와 b는 별개의 리스트 객체이지만 그 안의 원소는 공유된다. 따라서, 앞에서 보았듯이 b의 한 원소에 가한 변화는 a의 원소에도 적용된다.

깊은 복사(deep copy)는 새로운 객체를 생성하고 원래 객체가 담고 있던 모든 객체를 재귀적으로 복사한다. 어떤 객체의 깊은 복사본을 만들기 위한 내장 연산은 없다. 깊은 복사를 수행하기 위해서는 다음 예처럼 표준 라이브러리에 들어 있는 `copy.deepcopy()`를 사용하면 된다.

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
>>> b
[1, 2, [-100, 4]]
>>> a      # a가 변경되지 않은 것에 주목
```

```
[1, 2, [3, 4]]
>>>
```

1급 객체

파이썬에서 모든 객체는 ‘1급(first class)’이다. 이 말은 식별자로 명명될 수 있는 모든 객체는 동일한 지위를 지닌다는 것을 의미한다. 또한 이름을 가질 수 있는 모든 객체는 데이터로서 취급될 수 있다는 것을 의미하기도 한다. 다음 예는 두 개의 값을 담는 간단한 사전을 보여준다.

```
items = {
    'number' : 42
    'text' : 'Hello World'
}
```

모든 객체가 1급이라는 말의 의미는 앞의 사전에 일반적이지 않은 항목을 더 추가해보면 더 확실하게 이해할 수 있다. 다음은 몇 가지 예다.

```
items["func"] = abs                      # abs( ) 함수를 추가
import math
items["mod"] = math                       # 모듈을 추가
items["error"] = ValueError              # 예외 타입을 추가
nums = [1,2,3,4]
items["append"] = nums.append             # 다른 객체의 메서드를 추가
```

앞의 예에서 사전 items는 함수, 모듈, 예외 및 다른 객체의 메서드를 담는다. 원할 경우 원래는 이름이 나타나야 할 자리에 items에 대한 사전 검색을 대신 사용할 수도 있으며 그래도 코드는 문제 없이 실행된다. 다음은 한 예다.

```
>>> items["func"](-45)                  # abs(-45)를 실행
45
>>> items["mod"].sqrt(4)                # math.sqrt(4)를 실행
2.0
>>> try:
...     x = int("a lot")
... except items["error"] as e:          # except ValueError as e와 동일
...     print("Couldn't convert")
...
Couldn't convert
>>> items["append"](100)                # nums.append(100) 실행
>>> nums
[1, 2, 3, 4, 100]
>>>
```

프로그래밍을 처음 시작한 사람들은 종종 파이썬에서 모든 것이 1급이라는 사실을 올바르게 인식하지 못한다. 파이썬에서는 모든 것이 1급이기 때문에 매우 간결하고 유연한 코드를 작성할 수 있다. 예를 들어, “GOOG,100,490.10”라는 텍스트가

주어졌을 때 이것을 타입이 적절히 변환된 필드들의 리스트로 변환하고 싶다고 하자. 다음은 타입(1급 객체다)들의 리스트를 만들고 몇 가지 간단한 리스트 처리 연산을 수행함으로써 이를 영리하게 수행하는 방법을 보여준다.

```
>>> line = "GOOG,100,490.10"
>>> field_types = [str, int, float]
>>> raw_fields = line.split(',')
>>> fields = [ty(val) for ty,val in zip(field_types,raw_fields)]
>>> fields
['GOOG', 100, 490.1000000000002]
>>>
```

데이터 표현을 위한 내장 타입

파이썬에는 프로그램에서 사용되는 대부분의 데이터를 표현하는 데 사용할 수 있는 14 개의 내장 데이터 타입이 제공된다. 이 타입들은 표 3.1에서 볼 수 있듯이 몇 개의 주요 범주로 나누어진다. 표에서 타입 이름 옆은 `isinstance()`나 다른 타입 관련 함수로 해당 타입을 검사할 때 사용할 수 있는 이름이나 표현식을 보여준다. 몇몇 타입은 파이썬 2에만 있으며 그런 타입은 따로 표시를 해두었다.(파이썬 3에서 그런 타입은 사용하는 것이 권장되지 않거나(deprecated) 다른 타입과 합쳐졌다.)

표 3.1 데이터 표현을 위한 내장 타입

타입 범주	타입 이름	설명
없음(None)	<code>type(None)</code>	널 객체인 <code>None</code>
숫자(number)	<code>int</code>	정수
	<code>long</code>	임의 정밀도 정수(파이썬 2에만 있음)
	<code>float</code>	부동 소수점
	<code>complex</code>	복소수
	<code>bool</code>	불리언(<code>True</code> 나 <code>False</code>)
순서열(sequence)	<code>str</code>	문자열
	<code>unicode</code>	유니코드 문자열(파이썬 2에만 있음)
	<code>list</code>	리스트
	<code>tuple</code>	튜플
	<code>xrange</code>	<code>xrange()</code> 로 생성되는 정수 범위(파이썬 3에서는 <code>range</code>)
매핑(mapping)	<code>dict</code>	사전
집합(set)	<code>set</code>	변경 가능한 집합
	<code>frozenset</code>	변경 불가능한 집합

None 타입

None 타입은 널 객체(아무 값이 없는 객체)를 나타낸다. 파이썬에서는 정확히 하나의 널 객체가 제공되며 이를 프로그램 코드에서는 None이라고 쓴다. 이 객체는 값을 반환하지 않는 함수에 의해서 반환된다. None은 생략해도 되는 인수의 기본 값으로 흔히 사용되어 함수에서 호출자 쪽에서 실제로 해당 인수에 대한 값을 지정하였는지의 여부를 검사할 수 있게 한다. None은 아무런 속성도 지니지 않으며 불리언 표현식에서 False로 평가된다.

숫자 타입

파이썬은 불리언, 정수, 긴 정수, 부동 소수점 수, 복소수, 이렇게 다섯 가지의 숫자 타입을 제공한다. 불리언 타입을 제외한 모든 숫자 객체는 부호를 지닌다. 모든 숫자 타입은 변경이 불가능하다.

불리언 타입은 True와 False, 두 값으로 표현된다. True와 False라는 이름은 각각 숫자 값인 1과 0에 대응된다.

정수는 -2147483648에서 2147483647까지의 범위(기계에 따라 이 범위 값이 더 넓을 수도 있다)에 있는 전체 숫자를 나타낸다. 긴 정수는 무제한 범위의 전체 숫자를 나타낸다(그 범위가 사용 가능한 메모리에 의해서만 제한된다). 파이썬에서는 두 가지 정수 타입이 있지만 이 둘은 차이가 거의 없게 만들어져 있다(실제로 파이썬 3에서는 두 타입이 단일 정수 타입으로 합쳐졌다). 기존 파이썬 코드에서 긴 정수를 사용하는 것을 볼 때도 있을 테지만, 대부분 구현상의 문제일 뿐이니 무시해도 좋다. 여러분은 정수 연산을 수행할 때 그냥 정수 타입을 사용하면 된다. 한 가지 예외로는 정수 값에 대해 명시적인 타입 검사를 수행해야 하는 경우가 있다. 파이썬 2에서는 x가 정수에서 긴 정수로 타입 변환이 된 경우 isinstance(x, int) 표현식이 False를 반환한다.

부동 소수점 수는 해당 기계에 특화된(native) 배정밀도(64비트)로 표현된다. 보통은 대략 17자리 정밀도와 -308에서 308 사이의 범위의 지수를 사용하는 IEEE 754가 사용된다. 이것은 C에서 double 타입과 동일하다. 파이썬은 32비트 단일 정밀도 부동 소수점 수는 지원하지 않는다. 프로그램에서 숫자가 차지하는 메모리 공간이나 정밀도를 정확하게 제어할 필요가 있는 경우라면 numpy 확장 기능의 사용을 고려해보라.(<http://numpy.sourceforge.net>에서 관련 정보를 찾을 수 있다.)

복소수는 부동 소수점 수의 쌍으로 표현된다. 복소수 z의 실수부와 허수부는 z.real과 z.imag로 접근한다. z.conjugate() 메서드는 z의 복소수를 계산한다 (a+bj의 복소수는 a-bj다).

숫자 타입은 간단한 혼합 연산을 위해서 몇 가지 속성과 메서드를 갖는다. 정수는 유리수(fractions 모듈 참고)와 호환성 있는 연산을 위해 속성 x.numerator와 x.denominator를 갖는다. 정수 또는 부동 소수점 수 y는 복소수와의 호환성을 위해 y.real과 y.imag 속성 및 y.conjugate() 메서드를 갖는다. 부동 소수점 수 y는 y.as_integer_ratio()를 사용해 분수 정수 쌍으로 변환할 수 있다. y.is_integer() 메서드는 부동 소수점 수가 정수를 나타내는지를 검사한다. y.hex()과 y.fromhex() 메서드는 부동 소수점 수를 저수준 이진 표현법을 통해 다룰 필요가 있을 때 쓰인다.

라이브러리 모듈을 통해 몇 가지 추가적인 숫자 타입이 제공된다. decimal 모듈은 기본수 10인 십진수의 일반적인 연산을 지원한다. fractions 모듈은 유리수를 표현한다. 이 모듈들은 14장에서 자세하게 설명한다.

순서열 타입

순서열(sequence)은 음수가 아닌 정수로 색인되는 순서 있는 객체들의 모음을 표현한다. 문자열, 리스트, 튜플 등이 순서열에 포함된다. 문자열은 문자들의 순서열이고 리스트와 튜플은 임의의 파이썬 객체들의 순서열이다. 문자열과 튜플은 변경이 불가능하다. 리스트는 원소의 추가, 삭제, 치환이 가능하다. 모든 순서열은 반복을 지원한다.

모든 순서열에 공통적인 연산

표 3.2는 모든 순서열 타입에 적용할 수 있는 연산자와 메서드를 보여준다. 순서열 s의 요소 i는 색인 연산자 s[i]로 선택할 수 있고 부분순서열은 분할 연산자 s[i:j]나 확장 분할 연산자 s[i:j:stride]로 선택할 수 있다(이 연산자들은 4장에서 설명한다). 순서열의 길이는 내장 함수인 len(s)로 얻을 수 있다. 순서열의 최댓값과 최솟값은 내장 함수인 min(s)와 max(s)로 얻을 수 있다. 이 두 함수는 순서열 원소들 사이에 순서가 있는 경우에만 사용할 수 있다(보통은 숫자와 문자열에 대해서 사용). sum(s)는 s에 있는 항목들을 더하며, 오직 숫자 데이터에만 사용할 수 있다.

표 3.2 모든 순서열에 적용 가능한 연산과 메서드

항목	설명
<code>s[i]</code>	순서열의 원소 <code>i</code> 를 반환
<code>s[i:]</code>	조각을 반환
<code>s[i:j:stride]</code>	확장 분할에 의한 조각을 반환
<code>len(s)</code>	<code>s</code> 에 있는 원소 개수
<code>min(s)</code>	<code>s</code> 의 최솟값
<code>max(s)</code>	<code>s</code> 의 최댓값
<code>sum(s, [initial])</code>	<code>s</code> 의 항목들의 합
<code>all(s)</code>	<code>s</code> 에 있는 모든 항목이 <code>True</code> 인지 검사
<code>any(s)</code>	<code>s</code> 에 있는 아무 항목이나 <code>True</code> 인지를 검사

표 3.3은 리스트 같은 변경 가능한 순서열에 적용할 수 있는 추가 연산을 보여 준다.

표 3.3 변경 가능한 순서열에 적용할 수 있는 연산

항목	설명
<code>s[i] = v</code>	항목 대입
<code>s[i:] = t</code>	조각 대입
<code>s[i:j:stride] = t</code>	확장 분할에 의한 조각 대입
<code>del s[i]</code>	항목 삭제
<code>del s[i:]</code>	조각 삭제
<code>del s[i:j:stride]</code>	확장 분할에 의한 조각 삭제

리스트

리스트는 표 3.4에 나와 있는 메서드를 지원한다. 내장 함수인 `list(s)`는 반복 가능한 타입을 리스트로 변환한다. `s`가 이미 리스트이면 이 함수는 `s`의 얇은 복사본인 새로운 리스트를 생성한다. `s.append(x)` 메서드는 리스트의 끝에 새로운 원소 `x`를 추가 한다. `s.index(x)` 메서드는 리스트에서 처음으로 나타나는 `x`를 검색한다. `x`가 없으면 `ValueError` 예외를 던진다. 비슷하게, `s.remove(x)` 메서드는 리스트에서 처음으로 나타나는 `x`를 제거하고 만약 `x`가 없으면 `ValueError` 예외를 던진다. `s.extend(t)`는 `t`에 있는 원소들을 추가함으로써 리스트 `s`를 확장한다.

표 3.4 리스트 메서드

메서드	설명
s.list(s)	s를 리스트로 변환
s.append(x)	s의 끝에 새로운 원소 x를 추가한다
s.extend(t)	s의 끝에 새로운 리스트 t를 추가한다
s.count(x)	s에서 x가 출현한 횟수를 센다
s.index(x, [start [,stop]])	s[i] == x인 가장 작은 i를 반환한다. start와 end는 추가적으로 검색의 시작 및 종료 지점의 색인을 지정한다.
s.insert(i,x)	색인 i의 위치에 x를 삽입한다
s.pop([i])	리스트에서 원소 i를 제거하면서 i를 반환한다. i를 생략하면 마지막 원소가 제거되면서 반환된다.
s.remove(x)	x를 찾아서 s에서 제거한다.
s.reverse()	s의 항목들을 그 자리에서 뒤집는다.
s.sort([key [, reverse]])	s의 항목들을 그 자리에서 정렬한다. key는 키 함수이다. reverse는 리스트를 거꾸로 정렬하라는 플래그다. key와 reverse는 항상 키워드 인수로 지정되어야 한다. s에 있는 항목은 모두 동일한 타입이어야 한다.

s.sort() 메서드는 리스트의 원소들을 정렬하며(단, 모든 원소가 동일한 타입이어야 함) 추가로 키워드 인수이어야 하는 키 함수와 반전 플래그를 받아들인다. 키 함수는 정렬하기 전에 각 원소에 적용되는 함수다. 키 함수로 주어지는 함수는 입력으로 단일 항목을 넘겨 받아서 정렬 때 비교를 수행하는 데 사용할 값을 반환하는 함수이어야 한다. 키 함수는 문자열들을 정렬하되 대소문자를 구별하지 않는다면 하는 것처럼 특별한 종류의 정렬을 수행할 때 유용하게 쓰인다. s.reverse() 메서드는 리스트 항목들의 순서를 뒤집는다. sort()와 reverse() 메서드 모두 리스트의 원소들을 그 자리에서(in-place) 수정하고 None을 반환한다.

문자열

파이썬 2에는 두 종류의 문자열 객체 타입이 있다. 바이트 문자열은 8비트 데이터를 담는 바이트들의 순서열이다. 바이트 문자열은 이진 데이터와 널 바이트를 담는다. 유니코드 문자열은 인코딩되지 않은 유니코드 문자들의 순서열이며 각 유니코드 문자는 내부적으로 16비트 정수로 표현된다. 16비트 정수는 65,536개의 고유한 문자 값을 나타낼 수 있다. 유니코드 표준에서는 최대 100만 개의 고유한 문자 값

을 지원하지만, 파이썬에서는 추가 문자들을 기본으로 지원하지 않는다. 대신 추가 문자들은 대리자 쌍(surrogate pair)이라고 부르는 특별한 두 개의 문자(4바이트)로 인코딩된다. 대리자 쌍을 어떻게 해석할 것인지는 응용 프로그램에서 결정한다. 선택적인 기능으로 파이썬이 유니코드 문자를 32비트 정수로 저장하게 할 수도 있다. 이 기능이 활성화되면 파이썬에서 U+000000에서 U+110000까지 유니코드 값의 전체 범위를 표현할 수 있게 된다. 모든 유니코드와 관련된 함수도 그에 따라 적절히 조정된다.

문자열은 표 3.5에 나와 있는 메서드를 제공한다. 이 메서드들이 모두 문자열 인스턴스에 적용되는 것이긴 하지만, 어느 메서드도 문자열 데이터 자체를 변경하지는 않는다. `s.capitalize()`, `s.center()`, `s.expandtabs()` 같은 메서드들은 `s`를 변경하는 대신 항상 새로운 문자열을 반환한다. `s.isalnum()`과 `s.isupper()` 같은 문자 검사 메서드들은 문자열 `s` 안에 있는 모든 문자가 검사를 통과하면 `True`를 반환하고 아니면 `False`를 반환한다. 또한 이 메서드들은 문자열의 길이가 0이면 항상 `False`를 반환한다.

`s.find()`, `s.index()`, `s.rfind()`, `s.rindex()` 메서드는 `s`에서 특정 부분문자열을 찾는데 사용된다. 모두 `s`에서 부분문자열의 정수 색인을 반환한다. `find()` 메서드는 부분문자열을 찾지 못하면 -1을 반환하는 반면, `index()` 메서드는 `ValueError` 예외를 발생시킨다. `s.replace()` 메서드는 부분 문자열을 대체 텍스트로 바꾼다. 앞의 모든 메서드는 간단한 부분문자열만 다룰 수 있다는 점을 염두에 두기 바란다. 정규 표현식 패턴 매칭과 검색을 수행하려면 `re` 라이브러리 모듈에 들어 있는 함수들을 사용해야 한다.

`s.split()`과 `s.rsplit()` 메서드는 문자열을 구분자에 의해 분리되는 필드들의 리스트로 분할한다. `s.partition()`과 `s.rpartition()` 메서드는 분리자(separator) 부분 문자열을 검색한 다음에 `s`를 분리자 앞쪽의 텍스트, 분리자, 그리고 분리자 뒤쪽의 텍스트로 분할한다.

많은 문자열 메서드가 선택적으로 `s`에서 시작과 끝 색인을 지정하는 정수 값인 `start`와 `end` 매개변수를 받아들인다. 이 매개변수들은 대부분 음수 값을 취할 수 있으며, 그런 경우 문자열의 끝부터 색인이 적용된다.

`s.translate()` 메서드는 문자열에서 제어 문자를 모두 재빨리 제거하는 등 고급 문자 치환을 수행하는 데 사용된다. 이 메서드는 원본 문자열의 문자들과 결과 문자열의 문자들 간의 일대일 매핑 정보를 담은 변환 표(translation table)를 인수로 받

아들인다. 8비트 문자열에 대해서 변환 표는 256개의 문자로 된 문자열이어야 한다. 유니코드에 대해서 변환 표는 `s[n]`이 정수 값 n의 유니코드 문자에 대응하는 정수 문자 코드나 유니코드 문자를 반환하는 순서열 객체 `s`이어야 한다.

`s.encode()`와 `s.decode()` 메서드는 지정된 문자 인코딩에 따라서 문자열 데이터를 변환한다. 이 두 메서드는 입력으로 ‘ascii’, ‘utf-8’, ‘utf-16’ 같은 인코딩 이름을 받아들인다. 이 메서드들은 유니코드 문자열을 I/O 연산을 수행하기에 적합한 데이터 인코딩으로 변환하는 데 주로 사용된다. 여기에 관해서는 9장에서 더 자세하게 설명한다. 파이썬 3에서는 `encode()` 메서드가 문자열에만 있고 `decode` 메서드는 바이트 데이터 타입에만 있다.

`s.format()` 메서드는 문자열의 포맷을 지정하는 데 사용된다. 인수로서 위치 인수와 키워드 인수의 조합을 받아들인다. `{item}`으로 표현되는 `s`의 자리 표시자 (placeholder)는 적절한 인수에 의해 대체된다. 위치 인수는 `{0}`이나 `{1}` 같은 자리 표시자로 참조할 수 있다. 키워드 인수는 `{name}` 같은 이름을 갖는 자리 표시자로 참조할 수 있다. 다음은 한 예다.

```
>>> a = "Your name is {0} and your age is {age}"
>>> a.format("Mike", age=40)
'Your name is Mike and your age is 40'
>>>
```

포맷 문자열 안에서 자리 표시자 `{item}`은 간단한 색인 검색이나 속성 검색을 포함할 수도 있다. `n`이 숫자인 자리 표시자 `{item[n]}`은 `item`에 대해 순서열 검색을 수행한다. 숫자 아닌 `key`를 포함하는 자리 표시자 `{item[key]}`는 사전 검색 `item["key"]`를 수행한다. 자리 표시자 `{item.attr}`는 `item`의 속성 `attr`를 참조한다. `format()` 메서드에 대한 더 자세한 내용은 4장 ‘문자열 포맷 지정’ 절에서 확인할 수 있다.

표 3.5 문자열 메서드

메서드	설명
<code>s.capitalize()</code>	첫 번째 문자를 대문자로 만든다.
<code>s.center(width [,pad])</code>	width 길이를 가지는 필드에 문자열을 가운데 정렬한다. pad는 남는 공간을 채울 문자를 지정한다.
<code>s.count(sub [,start [,end]])</code>	지정된 부분문자열이 나타나는 횟수를 센다.
<code>s.decode([encoding [,errors]])</code>	문자열을 디코딩해서 유니코드 문자열을 반환한다(바이트 문자열에만 적용).

메서드	설명
s.encode([encoding [,errors]])	인코딩된 버전의 문자열을 반환한다(유니코드 문자열에만 적용).
s.endswith(suffix [,start [,end]])	문자열이 suffix로 끝나는지를 검사한다.
s.expandtabs([tabsize])	탭을 스페이스로 대체한다.
s.find(sub [, start [,end]])	부분문자열 sub이 처음으로 나타나는 위치를 찾는다. 못 찾으면 -1을 반환한다.
s.format(*args, **kwargs)	s의 포맷을 지정한다.
s.index(sub [, start [,end]])	부분문자열 sub이 처음으로 나타나는 위치를 찾는다. 못 찾으면 예외를 발생시킨다.
s.isalnum()	모든 문자가 알파벳이나 숫자인지를 검사한다.
s.isalpha()	모든 문자가 알파벳인지를 검사한다.
s.isdigit()	모든 문자가 숫자인지를 검사한다.
s.islower()	모든 문자가 소문자인지를 검사한다.
s.isspace()	모든 문자가 공백 문자인지를 검사한다.
s.istitle()	문자열이 제목 대소문자 형태(각 단어의 첫 글자가 대문자)인지를 검사한다.
s.isupper()	모든 문자가 대문자인지를 검사한다.
s.join(t)	s를 분리자로 사용해서 순서열 t에 들어 있는 문자열들을 이어 붙인다.
s.ljust(width [, fill])	길이 width인 문자열에서 s를 왼쪽 정렬한다.
s.lower()	소문자로 변경한다.
s.lstrip([chrs])	앞쪽에 있는 공백이나 chrs로 지정된 문자들을 제거한다.
s.partition(sep)	문자열 sep에 기반해서 문자열을 분할한다. 튜플 (머리,sep,꼬리)를 반환하거나, sep를 찾을 수 없는 경우 (s, " ", " ")를 반환한다.
s.replace(old, new [,maxreplace])	부분문자열을 대체한다.
s.rfind(sub [,start [,end]])	부분문자열이 최종적으로 나타난 위치를 찾는다.
s.rindex(sub [,start [,end]])	부분문자열이 최종적으로 나타난 위치를 찾거나 예외를 발생시킨다.
s.rjust(width [, fill])	길이 width인 문자열에서 s를 오른쪽 정렬한다.
s.rpartition(sep)	s를 분리자 sep에 기반해서 분할하지만 문자열의 오른쪽 끝에서부터 검색한다.
s.rsplit([sep [,maxsplit]])	sep를 구분자로 사용해서 문자열을 끝에서부터 분할한다. maxsplit은 최대 분할 횟수를 지정한다. maxsplit이 생략되면 split() 메서드와 결과가 동일하다.
s.rstrip([chrs])	끝에 나오는 공백이나 chrs로 지정된 문자들을 제거한다.

메서드	설명
s.split([sep [,maxsplit]])	sep를 구분자로 사용해서 문자열을 분할한다. maxsplit은 최대 분할 횟수를 지정한다.
s.splitlines([keepends])	문자열을 줄들의 리스트로 분할한다. keepends가 1이면 끝에 있는 줄바꿈 문자들이 유지된다.
s.startswith(prefix [,start [,end]])	문자열이 prefix로 시작하는지를 검사한다.
s.strip([chrs])	앞이나 뒤에 나오는 공백이나 chrs로 지정된 문자들을 제거한다.
s.swapcase()	대문자를 소문자로, 소문자를 대문자로 바꾼다.
s.title()	제목 대소문자 형태로 된 문자열을 반환한다.
s.translate(table [,deletechars])	문자 변환 표 table을 사용해서 문자열을 치환한다. deletechars에 있는 문자들은 삭제된다.
s.upper()	대문자로 변경한다.
s.zfill(width)	문자열을 왼쪽에서부터 width만큼 0으로 채운다.

xrange() 객체

내장 함수인 xrange([i,j [,stride]])는 범위가 k인($i \leq k < j$) 정수들을 나타내는 객체를 생성한다. 첫 번째 색인인 i와 stride는 생략할 수 있고 각각 기본 값 0과 1을 갖는다. xrange 객체는 접근될 때마다 값을 계산한다. xrange 객체는 순서열 같지만 실제로는 몇 가지 제약이 있다. 예를 들어, 분할 연산자들은 지원되지 않는다. 이러한 제약 사항 때문에 xrange는 간단히 루프를 돌면서 반복 수행을 하는 것과 같은 몇 가지 용도로만 사용된다.

파이썬 3에서는 xrange()의 이름이 range()로 변경되었다. 작동 방식은 이전과 동일하다.

매핑 타입

매핑 객체(mapping object)는 거의 임의적인 키 값으로 색인되는 임의의 객체들을 나타낸다. 순서열과는 달리 매핑 객체는 순서를 지니지 않으며, 숫자, 문자열 및 기타 다른 객체로 색인될 수 있다. 매핑 객체는 변경이 가능하다.

사전은 유일한 내장 매핑 타입이며 해시 테이블이나 연관 배열의 파이썬 버전이다. 변경 불가능한 객체라면 어떤 객체든 사전의 키 값으로 사용할 수 있다(문자열, 숫자, 튜플 등). 리스트, 사전, 변경 가능한 객체를 담은 튜플은 키로 사용할 수 없

다(사전 타입은 키 값이 변하지 않아야 하는 것을 요구한다).

매핑 객체에서 항목을 선택하려면 키 색인 연산자 `m[k]`를 사용하면 된다(`k`는 키 값). 키를 못 찾으면 `KeyError` 예외가 발생한다. `len(m)` 함수는 매핑 객체에 들어 있는 항목의 개수를 반환한다. 표 3.6은 지원되는 메서드와 연산들을 보여준다.

표 3.6에 나와 있는 메서드는 대부분 사전의 내용을 수정하거나 추출하는 데 사용된다. `m.clear()` 메서드는 모든 항목을 제거한다. `m.update(b)` 메서드는 매핑 객체 `b`에 있는 모든 (`키`, `값`) 쌍을 추가함으로써 현재 매핑 객체를 갱신한다. `m.get(k, v)` 메서드는 객체를 추출하지만, 부가적으로 기본 값 `v`를 받아들여서 키가 존재하

표 3.6 사전의 메서드와 연산

항목	설명
<code>len(m)</code>	<code>m</code> 에 있는 항목 개수를 반환한다.
<code>m[k]</code>	키 <code>k</code> 로 <code>m</code> 의 항목을 반환한다.
<code>m[k] = x</code>	<code>m[k]</code> 를 <code>x</code> 로 설정한다.
<code>del m[k]</code>	<code>m</code> 에서 <code>m[k]</code> 를 제거한다.
<code>k in m</code>	키 <code>k</code> 가 <code>m</code> 에 있으면 <code>True</code> 를 반환한다.
<code>m.clear()</code>	<code>m</code> 에서 모든 항목을 제거한다.
<code>m.copy()</code>	<code>m</code> 의 복사본을 생성한다.
<code>m.fromkeys(s [,value])</code>	순서열 <code>s</code> 에서 키를 가져오고 모든 값을 <code>value</code> 로 설정한 새로운 사전을 생성한다.
<code>m.get(k [,v])</code>	<code>m[k]</code> 가 있으면 <code>m[k]</code> 를 반환하고 아니면 <code>v</code> 를 반환한다.
<code>m.has_key(k)</code>	<code>m</code> 에 키 <code>k</code> 가 있으면 <code>True</code> 를 반환한다. 아니면 <code>False</code> 를 반환한다. 사용이 권장되지 않는다. 대신 <code>in</code> 연산자를 사용하도록 한다. 파이썬 2에서만 사용 가능.
<code>m.items()</code>	<code>m</code> 의 모든 (<code>키</code> , <code>값</code>) 쌍들로 구성되는 순서열을 반환한다.
<code>m.keys()</code>	<code>m</code> 의 모든 키들의 순서열을 반환한다.
<code>m.pop(k [,default])</code>	<code>m[k]</code> 가 있으면 <code>m[k]</code> 를 반환하고 <code>m</code> 에서 제거한다. <code>m[k]</code> 가 없을 경우 <code>default</code> 가 제공되면 <code>default</code> 를 반환하고, 아니면 <code>KeyError</code> 예외를 발생시킨다.
<code>m.popitem()</code>	<code>m</code> 에서 임의의 (<code>키</code> , <code>값</code>) 쌍을 제거하고 이를 튜플로서 반환한다.
<code>m.setdefault(k [, v])</code>	<code>m[k]</code> 가 있으면 <code>m[k]</code> 를 반환한다. 없으면, <code>v</code> 를 반환하고 <code>m[k]</code> 를 <code>v</code> 로 설정한다.
<code>m.update(b)</code>	<code>b</code> 에 있는 모든 객체를 <code>m</code> 에 추가한다.
<code>m.values()</code>	<code>m</code> 에 있는 모든 값으로 구성되는 순서열을 반환한다.

지 않는 경우 기본 값을 반환한다. `m.setdefault(k, v)` 메서드는 객체가 존재하지 않는 경우 `v`를 반환할 뿐만 아니라 `m[k]`를 `v`로 설정하다는 점을 제외하고는 `m.get()`과 비슷하다. `v`가 생략되면 기본 값으로 `None`이 사용된다. `m.pop()` 메서드는 사전에서 항목을 반환하는 동시에 제거한다. `m.popitem()` 메서드는 반복적으로 사전의 내용을 파괴하는 데 쓰인다.

`m.copy()` 메서드는 매핑 객체에 들어 있는 항목들에 대한 얇은 복사본들을 생성하고 이들을 새로운 매핑 객체로 반환한다. `m.fromkeys(s, value)` 메서드는 모든 키를 순서열 `s`에서 가져와서 새로운 매핑 객체를 생성한다. 옵션인 `value` 값이 주어지지 않으면, 모든 키에 대한 값은 `None`으로 설정된다. `fromkeys()` 메서드는 클래스 메서드로 정의되어 있기 때문에 `dict.fromkeys()`처럼 클래스 이름으로도 호출할 수 있다.

`m.items()` 메서드는 (`키, 값`) 쌍들을 담은 순서열을 반환한다. `m.keys()` 메서드는 키들로 구성되는 순서열을 반환하고 `m.values()` 메서드는 값들로 구성되는 순서열을 반환한다. 이 메서드들의 결과에 대해서 수행할 수 있는 안전한 연산은 오직 반복밖에 없다고 가정하는 것이 좋다. 파이썬 2에서는 리스트가 반환되지만, 파이썬 3에서는 매핑 객체의 현재 내용에 대해 반복을 수행하는 반복자가 반환된다. 반환되는 것이 반복자라고 생각하고 코드를 작성하면 두 버전의 파이썬에서 모두 호환된다. 이 메서드들의 반환 값을 데이터로 저장하고 싶으면 리스트로 저장함으로써 복사본을 생성하면 된다. 예를 들어, `items = list(m.items())`처럼 할 수 있다. 간단히 모든 키의 리스트를 원하면 `keys = list(m)`처럼 하면 된다.

집합 타입

집합(set)은 고유한 항목들의 순서 없는 모음이다. 순서열과는 달리 집합은 색인 및 분할 연산을 지원하지 않는다. 객체에 연결된 키가 없다는 점에서 사전과도 다르다. 집합에 들어 있는 항목들은 변경이 불가능해야 한다. 집합에는 두 종류가 있다. `set`은 변경이 가능한 집합이고 `frozenset`은 변경이 불가능한 집합이다. 다음과 같이 둘 다 내장 함수를 사용해서 생성한다.

```
s = set([1, 5, 10, 15])
f = frozenset(['a', 37, 'hello'])
```

`set()`과 `frozenset()` 모두 주어진 인수에 대해 반복을 수행함으로써 집합을 채운

표 3.7 집합 타입의 메서드와 연산들

항목	설명
len(s)	s에 있는 항목의 개수
s.copy()	s의 복사본을 생성한다.
s.difference(t)	차집합. s에는 있지만 t에는 없는 모든 항목을 반환한다.
s.intersection(t)	교집합. s와 t에 둘 다 들어 있는 모든 항목을 반환한다.
s.isdisjoint(t)	s와 t에 공통으로 들어 있는 항목이 없을 경우 True를 반환한다.
s.issubset(t)	s가 t의 부분집합인 경우 True를 반환한다.
s.issuperset(t)	s가 t의 포함집합인 경우 True를 반환한다.
s.symmetric_difference(t)	대칭 차집합. s나 t에 들어 있지만 둘 모두에는 들어 있지 않은 모든 항목을 반환한다.
s.union(t)	합집합. s나 t에 있는 모든 항목을 반환한다.

표 3.8 변경 가능한 집합의 메서드들

항목	설명
s.add(item)	item을 s에 추가한다. 이미 item이 s에 있으면 아무런 효과가 없다.
s.clear()	s에서 모든 항목을 제거한다.
s.difference_update(t)	s에서 t에 있는 모든 항목을 제거한다.
s.discard(item)	s에서 item을 제거한다. s에 item이 없는 경우 아무런 일도 일어나지 않는다.
s.intersection_update(t)	s와 t의 교집합을 구하고 결과를 s에 남겨둔다.
s.pop()	s에서 아무 원소나 반환하는 동시에 s에서 제거한다.
s.remove(item)	s에서 item을 제거한다. item이 없으면 KeyError 예외가 발생한다.
s.symmetric_difference_update(t)	s와 t의 대칭 차집합을 구하고 결과를 s에 남겨둔다.
s.update(t)	t의 모든 항목을 s에 추가한다. t는 집합, 순서열 또는 반복을 지원하는 어떤 객체든 될 수 있다.

다. 두 집합 모두 표 3.7에 나와 있는 메서드를 지원한다.

메서드 s.difference(t), s.intersection(t), s.symmetric_difference(t), s.union(t)는 집합에 대한 표준 수학 연산을 수행하는 데 사용된다. 반환되는 값은 s와 동일한 타입을 갖게 된다(set나 frozenset 중 하나). 매개변수 t는 반복을 지원하는 어떤 객체든 될 수 있다. 여기에는 집합, 리스트, 튜플, 문자열 등이 포함된다. 이 집합 연산들

은 4장에서 설명되듯이 수학 연산자의 형태로도 제공된다.

변경 가능한 집합은 표 3.8에 나와 있는 메서드를 추가로 지원한다.

이 모든 연산은 `s`를 직접 수정한다. 매개변수 `t`는 반복을 지원하는 어떤 객체든 될 수 있다.

프로그램 구조를 나타내는 내장 타입

파이썬에서 함수, 클래스, 모듈은 모두 데이터로서 다루어질 수 있는 객체이다. 표 3.9는 프로그램 자체의 다양한 요소들을 나타내는 데 사용되는 타입들을 보여준다.

표 3.9 프로그램 구조를 나타내기 위한 내장 파이썬 타입들

타입 분류	타입 이름	설명
호출가능(callable)	<code>types.BuiltinFunctionType</code>	내장 함수나 메서드
	<code>type</code>	내장 타입과 클래스의 타입
	<code>object</code>	모든 타입과 클래스의 조상
	<code>types.FunctionType</code>	사용자 정의 함수
	<code>types.MethodType</code>	클래스 메서드
모듈	<code>types.ModuleType</code>	모듈
클래스	<code>object</code>	모든 타입과 클래스의 조상
타입	<code>type</code>	내장 타입과 클래스의 타입

표 3.9에서 `object`와 `type`은 두 번씩 나와 있다. 그 이유는 클래스와 타입의 경우 모두 함수로서 호출될 수 있기 때문이다.

호출가능 타입

호출가능(callable) 타입은 함수 호출 연산을 지원하는 객체를 나타낸다. 이러한 객체에는 몇 종류가 있는데, 여기에는 사용자 정의 함수, 내장 함수, 인스턴스 메서드, 클래스가 포함된다.

사용자 정의 함수

사용자 정의 함수(user-defined function)은 모듈 수준에서 `def`문이나 `lambda` 연산자로 생성되는 호출가능 객체이다. 다음은 한 예이다.

```
def foo(x,y):
    return x + y

bar = lambda x,y: x + y
```

사용자 정의 함수 f는 다음 속성들을 갖는다.

속성	설명
f.__doc__	문서화 문자열
f.__name__	함수 이름
f.__dict__	함수 속성을 담은 사전
f.__code__	바이트 컴파일된 코드
f.__defaults__	기본 인수들을 담는 튜플
f.__globals__	전역 네임스페이스를 나타내는 사전
f.__closure__	중첩된 유효범위와 관련된 데이터를 담은 튜플

파이썬 2의 오래전 버전에서는 앞에 나와 있는 속성 중 많은 것들이 func_code, func_defaults 등의 형태를 가지고 있었다. 여기에 있는 속성들은 파이썬 2.6과 파이썬 3에서 사용할 수 있다.

메서드

메서드(method)는 클래스 정의 안에서 정의되는 함수이다. 메서드에는 인스턴스 메서드, 클래스 메서드, 정적 메서드 세 종류가 있다.

```
class Foo(object):
    def instance_method(self,arg):
        문장들
    @classmethod
    def class_method(cls,arg):
        문장들
    @staticmethod
    def static_method(arg):
        문장들
```

인스턴스 메서드(instance method)는 주어진 클래스에 속하는 인스턴스에 대해 수행되는 메서드다. 인스턴스 메서드에는 첫 번째 인수로 인스턴스가 전달되고, 관례에 따라 이 인스턴스를 self라고 부른다. 클래스 메서드(class method)는 클래스 자체를 객체로 보고 여기에 대해 수행되는 메서드다. 클래스 메서드에는 첫 번째 인수 cls로 클래스 객체가 전달된다. 정적 메서드(static method)는 클래스 안에 함께 묶이게 된 함수이다. 정적 메서드는 첫 번째 인수로 인스턴스나 클래스 객체를 받지 않는다.

인스턴스 메서드와 클래스 메서드는 모두 타입이 types.MethodType인 특수한

객체로 표현된다. 이 특수한 타입을 이해하기 위해서는 객체 속성 검색()이 어떻게 작동하는지 정확히 이해하고 있어야 한다. 객체에서 무언가를 찾는 연산은 항상 함수를 호출하는 연산과는 별개의 연산으로서 수행된다. 즉, 메서드를 호출할 때 이 두 연산 모두 발생하지만, 별개의 단계로서 수행된다는 의미이다. 다음 예는 앞의 코드에 나온 Foo의 인스턴스에 대해 f.instance_method(arg)를 호출하는 과정을 보여준다.

```
f = Foo()
meth = f.instance_method      # 인스턴스를 생성한다.
meth(37)                      # 메서드를 검색한다. ()이 없는 것에 주목
                             # 이제 메서드를 호출한다.
```

앞의 예에서 meth를 묶인 메서드(bound method)라고 부른다. 묶인 메서드는 함수(메서드)와 함수와 연관된 인스턴스 둘 다를 감싸는 호출가능 객체이다. 묶인 메서드를 호출하면 인스턴스가 메서드의 첫 번째 인수(self)로서 전달된다. 앞의 예에서 meth를 실행할 준비가 되었지만, 아직 함수 호출 연산자 ()를 사용하여 호출되지 않은 메서드 호출로 볼 수 있다.

클래스 자체에 대해서 메서드 검색을 수행할 수도 있다. 다음은 한 예다.

```
umeth = Foo.instance_method    # Foo에 대고 instance_method를 검색한다.
umeth(f, 37)                   # self를 직접 전달하면서 호출한다.
```

앞의 예에서 umeth를 안 묶인 메서드(unbound method)라고 부른다. 안 묶인 메서드는 메서드 함수를 감싸지만, 적절한 타입의 인스턴스가 첫 번째 인수로 넘어올 것을 기대하는 호출가능 객체이다. 앞의 예에서 Foo의 인스턴스인 f를 첫 번째 인자로서 사용하였다. 엉뚱한 타입의 객체를 전달하면 TypeError 예외가 발생한다. 다음은 그 예이다.

```
>>> umeth("hello", 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'instance_method' requires a 'Foo' object but
received a
'str'
>>>
```

사용자 정의 클래스에 대해서 묶인 메서드와 안 묶인 메서드는 모두 타입이 types.MethodType인 객체로 표현되며, 이 타입은 단순히 보통의 함수 객체를 감싸는 얇은 래퍼에 불과하다. 메서드 객체에는 다음 속성이 정의되어 있다.

속성	설명
m.__doc__	문서화 문자열

<code>m.__name__</code>	메서드 이름
<code>m.__class__</code>	이 메서드가 정의되어 있는 클래스
<code>m.__func__</code>	메서드를 구현하는 함수 객체
<code>m.__self__</code>	메서드와 연결되어 있는 인스턴스(안 뮐인 메서드에 대해서는 None)

파이썬 3에서는 안 뮐인 메서드가 더 이상 `types.MethodType` 객체에 의해 감싸지지 않는다는 점이 약간 다르다. 앞의 예에서처럼 `Foo.instance_method`에 접근하면 메서드를 구현하는 보통의 함수 객체를 얻게 된다. 또한, 매개변수 `self`에 대한 타입 검사도 수행되지 않는다.

내장 함수와 메서드

`types.BuiltinFunctionType` 객체는 C와 C++로 구현된 내장 함수와 메서드를 나타낸다. 내장 함수에는 다음 속성이 존재한다.

속성	설명
<code>b.__doc__</code>	문서화 문자열
<code>b.__name__</code>	함수나 메서드 이름
<code>b.__self__</code>	메서드와 연결된 인스턴스(뮤인 메서드일 경우)

`len()` 같은 내장 함수에 대해서 `__self__`는 `None`으로 설정되며 어떤 객체에도 뮐이지 않게 된다. `x`가 리스트일 때 `x.append` 같은 내장 메서드에 대해서 `__self__`는 `x`로 설정된다.

호출 가능한 클래스와 인스턴스

클래스 객체와 인스턴스는 호출가능 객체로도 작동한다. 클래스 객체는 `class`문에 의해서 생성되며, 새로운 인스턴스를 생성하기 위해서 호출된다. 클래스 객체가 호출될 때 인수들은 클래스의 `__init__()` 메서드에 전달되어서 새로이 생성되는 인스턴스를 초기화하는 데 사용된다. 인스턴스는 특수한 메서드인 `__call__()`을 정의함으로써 함수를 흉내 낼 수 있다. 인스턴스 `x`에 대해서 이 메서드가 정의되어 있으면 `x(args)`를 호출할 경우 `x.__call__(args)`가 호출된다.

클래스, 타입, 인스턴스

클래스를 정의하면 일반적으로 타입이 `type`인 객체가 생성된다. 다음은 한 예다.

```
>>> class Foo(object):
...     pass
...
>>> type(Foo)
<type 'type'>
```

다음은 타입 객체 t에 대해서 주로 사용되는 속성을 보여준다.

속성	설명
t.__doc__	문서화 문자열
t.__name__	클래스 이름
t.__bases__	기반 클래스들로 구성되는 튜플
t.__dict__	클래스 메서드와 변수들을 담은 사전
t.__module__	클래스가 정의되어 있는 모듈의 이름
t.__abstractmethods__	추상 메서드 이름들의 집합(추상 메서드가 없는 경우 정의가 안될 수도 있음)

어떤 객체의 인스턴스가 생성되면 이 인스턴스의 타입은 인스턴스를 정의하는 클래스가 된다. 다음은 한 예다.

```
>>> f = Foo()
>>> type(f)
<class '__main__.Foo'>
```

다음은 인스턴스 i의 특수한 속성을 보여준다.

속성	설명
i.__class__	인스턴스가 속하는 클래스
i.__dict__	인스턴스 데이터를 담은 사전

__dict__ 속성에는 일반적으로 인스턴스와 관련된 모든 데이터가 저장된다. i.attr = value 같은 대입문이 실행되면 값이 __dict__에 저장된다. 사용자 정의 클래스에서 __slots__를 사용하면 더 효율적인 내부적인 표현 방식이 사용되고 __dict__ 속성은 사용되지 않는다. 객체와 파이썬의 객체 시스템의 구성에 관해서는 7장에서 더 자세하게 설명한다.

모듈

모듈(module) 타입은 import문으로 로드되는 객체들을 담는 컨테이너이다. 예를 들어, import foo문이 프로그램에 들어 있으면 대응되는 모듈 객체에 이름 foo가 할당된다. 모듈은 속성 __dict__로 접근할 수 있는 사전을 통해서 구현되는 네임스페이스를 정의한다. 모듈의 속성이 참조되면(점 연산자를 사용해서) 사전 검색으로

해석된다. 예를 들어, m.x는 m.__dict__["x"]와 같다. 비슷하게, m.x = y처럼 속성에 값을 할당하는 것은 m.__dict__["x"] = y와 같다. 모듈에는 다음 속성이 존재한다.

속성	설명
m.__dict__	모듈과 연결된 사전
m.__doc__	모듈의 문서화 문자열
m.__name__	모듈의 이름
m.__file__	모듈이 로드된 파일
m.__path__	완전히 한정된(fully qualified) 패키지 이름. 모듈 객체가 패키지를 참조할 경우에만 정의됨.

인터프리터의 내부를 나타내는 내장 타입

인터프리터의 내부에서 사용되는 몇몇 객체들은 사용자에게 공개되어 있다. 여기에는 표 3.10에 나와 있듯이 역추적(traceback) 객체, 코드 객체, 프레임(frame) 객체, 생성기 객체, 분할(slice) 객체, Ellipsis 등이 있다. 프로그램에서 이 객체들을 직접 다루는 일은 드물지만 이 객체들은 도구를 제작하거나 프레임워크를 설계할 때 유용하게 쓸 수 있다.

표 3.10 인터프리터 내부를 나타내는 내장 파이썬 타입

타입 이름	설명
types.CodeType	바이트 컴파일된 코드
types.FrameType	실행 프레임
types.GeneratorType	생성기 객체
types.TracebackType	예외의 스택 역추적
slice	확장 분할 연산에 의해서 생성됨
Ellipsis	확장 분할 연산에서 사용됨

코드 객체

코드 객체는 바이트 컴파일된 미가공 실행 코드, 즉 바이트코드(bytecode)를 나타내며, 주로 내장 함수인 compile() 함수에 의해 반환된다. 코드 객체는 함수와 비슷하지만, 코드가 정의된 네임스페이스와 관련된 어떤 컨텍스트도 포함하지 않으며 기본 인수 값에 대한 정보도 저장하지 않는다는 점이 다르다. 코드 객체 c는 다음에 나오는 읽기 전용 속성들을 갖는다.

속성	설명
c.co_name	함수 이름
c.co_argcount	(기본 값들도 포함한) 위치 인수(positional argument)의 개수
c.co_nlocals	함수에 의해 사용되는 지역 변수의 개수
c.co_varnames	지역 변수들의 이름을 담은 튜플
c.co_cellvars	중첩된 함수에 의해 참조되는 변수들의 이름을 담은 튜플
c.co_freevars	중첩된 함수에 의해서 사용되는 자유 변수(free variable)들의 이름을 담은 튜플
c.co_code	미가공 바이트코드를 나타내는 문자열
c.co_consts	바이트코드에 의해서 사용되는 상수들을 담은 튜플
c.co_names	바이트코드에 의해서 사용되는 이름들을 담은 튜플
c.co_filename	코드가 컴파일된 파일의 이름
c.co_firstlineno	함수의 첫 번째 줄 번호
c.co_lnotab	바이트코드 오프셋(offset)을 줄 번호로 인코딩하는 문자열
c.co_stacksize	요구되는 스택 크기(지역 변수를 포함해서)
c.co_flags	인터프리터의 플래그들을 담은 정수. 함수가 “*args”로 가변 개수의 위치 인수를 사용하면 두 번째 비트가 켜진다. 함수가 “**kwargs”로 임의의 키워드 인수를 받아들이면 세 번째 비트가 켜진다. 나머지 비트들은 예약되어 있다.

프레임 객체

프레임 객체는 실행 프레임을 나타내며 역추적 객체에서 주로 사용된다. 프레임 객체 f는 다음의 읽기 전용 속성을 갖는다.

속성	설명
f.f_back	이전 스택 프레임(호출자 쪽)
f.f_code	실행 중인 코드 객체
f.f_locals	지역 변수를 위한 사전
f.f_globals	전역 변수를 위한 사전
f.f_builtins	내장 이름들을 위한 사전
f.f_lineno	줄 번호
f.f_lasti	현재 명령어. f_code에 들어 있는 바이트코드 문자열에 대한 색인.

다음 속성들은 변경이 가능하다(디버거나 기타 도구에 의해서 사용됨).

속성	설명
f.f_trace	소스 코드의 각 줄의 시작 시점에서 호출된 함수
f.f_exc_type	가장 최근 예외 타입(파이썬 2에만 있음)
f.f_exc_value	가장 최근 예외 값(파이썬 2에만 있음)
f.f_exc_traceback	가장 최근 예외 역추적 정보(파이썬 2에만 있음)

역추적 객체

역추적 객체는 예외가 발생할 때 생성되며 스택 추적 정보를 담는다. 예외 처리기에 sys.exc_info() 함수를 사용해서 스택 추적 정보를 얻을 수 있다. 역추적 객체에는 다음 읽기 전용 속성이 있다.

속성	설명
t.tb_next	스택 추적에서 다음 수준(예외가 발생한 실행 프레임 쪽)
t.tb_frame	현재 수준의 실행 프레임 객체
t.tb_lineno	예외가 발생된 줄 번호
t.tb_lasti	현재 수준에서 실행 중인 명령어

생성기 객체

생성기 객체는 생성기 함수가 호출되면 생성된다(6장 참고). 함수에서 특수한 키워드인 yield가 사용되면 생성기 함수가 정의된다. 생성기 객체는 반복자의 역할과 생성기 함수 자체에 대한 정보를 담는 역할 둘 다를 수행한다. 생성기 객체에는 다음 속성과 메서드가 있다.

속성	설명
g.gi_code	생성기 함수에 대한 코드 객체
g.gi_frame	생성기 함수에 대한 실행 프레임
g.gi_running	생성기 함수가 현재 실행 중인지를 나타내는 정수
g.next()	다음 yield문까지 실행하고 값을 반환한다.(파이썬 3에서는 __next__)
g.send(value)	생성기에 값을 보낸다. 전달된 값은 yield 표현식에 의해 반환되고 다음 yield 표현식에 도달할 때까지 실행이 이어진다. send()는 생성기의 다음 값을 반환한다.
g.close()	생성기 함수에서 GeneratorExit 예외를 발생시켜서 생성기를 닫는다. 이 메서드는 생성기 객체가 쓰레기 수집될 때 자동으로 호출된다.
g.throw(exc[,exc_value[,exc_tb]])	현재 yield문의 위치에서 예외를 발생시킨다. exc는 예외의 타입을, exc_value는 예외의 값을, exc_tb는 역추적 정보를 지정한다. 예외가 적절히 잡혀서 처리되면 생성기의 다음 값을 반환한다.

분할 객체

분할 객체는 a[i:j:stride], a[i:j, n:m] 또는 a[... , i:j] 같은 확장 분할 문법을 통해 주어지는 분할을 나타낸다. 분할 객체는 내장 함수 slice([i,] j [,stride])에 의해서도 생성된다. 다음은 분할 객체의 읽기 전용 속성이다.

속성	설명
s.start	분할의 하한을 나타낸다. 생략된 경우에는 None.
s.stop	분할의 상한을 나타낸다. 생략된 경우에는 None.
s.step	분할의 보폭을 나타낸다. 생략된 경우에는 None.

분할 객체에는 s.indices(length)라는 메서드 하나만 있다. 이 메서드는 길이를 받아서 그 길이의 순서열에 분할이 어떻게 적용되는지를 알려주는 튜플 (start, stop, stride)을 반환한다. 다음은 한 예다.

```
s = slice(10,20)    # [10:20]를 나타내는 분할 객체
s.indices(100)      # (10,20,1)를 반환 -> [10:20]
s.indices(15)       # (10,15,1)를 반환 -> [10:15]
```

Ellipsis 객체

Ellipsis 객체는 색인 검색 []에서 생략 부호(...)가 있다는 것을 나타내는 데 사용된다. 이 타입의 객체는 하나가 있고 내장 이름 Ellipsis를 통해 접근이 가능하다. 파이썬의 내장 타입에서는 Ellipsis를 사용하는 곳이 없지만, 사용자 정의 객체에서 색인 연산자 []에 대한 고급 기능을 제공하고자 할 때 유용하게 쓸 수 있다. 다음 코드는 Ellipsis가 생성되어서 색인 연산자에 전달되는 모습을 보여준다.

```
class Example(object):
    def __getitem__(self, index):
        print(index)
e = Example()
e[3, ..., 4]    # e.__getitem__((3, Ellipsis, 4))를 호출한다.
```

객체의 작동 방식과 특수 메서드

일반적으로 파이썬에 있는 객체들은 구현 기능에 따라 분류된다. 예를 들어, 문자열, 리스트, 튜플 같은 모든 순서열 타입은 s[n], len(s) 같은 순서열 연산의 공통된 집합을 구현하고 있다는 이유만으로 함께 묶인다. 모든 기본적인 인터프리터 연산은 특수한 객체 메서드에 의해 구현된다. 이 메서드들의 이름은 항상 이중 밑줄(__)로 시작해서 이중 밑줄로 끝난다. 이 메서드들은 프로그램이 실행됨에 따라 인터프리터에 의해 자동으로 호출된다. 예를 들어, 연산 x + y는 내부 메서드인 x.__add__(y)에 대응되고 색인 연산인 x[k]는 x.__getitem__(k)에 대응된다. 각 데이터 타입의 작동 방식은 온전히 그 타입이 구현하고 있는 특수한 메서드들의 집합에 따라 결정된다.

사용자 정의 클래스도 이 장에서 설명된 특수한 메서드들의 적절한 부분 집합을 제공함으로써 내장 타입처럼 작동할 수 있다. 또한, 리스트나 사전 같은 내장 타입들은 상속 후 특수 메서드 중 일부를 재정의함으로써 특수화할 수 있다.

이어지는 절에서는 여러 가지 인터프리터의 기능에 대응되는 특수 메서드들을 설명한다.

객체 생성 및 파괴

표 3.11에 나와 있는 메서드들은 인스턴스를 생성하고, 초기화하며, 파괴하는 데 사용된다. `__new__()`는 인스턴스를 생성하기 위해 호출되는 클래스 메서드이다. `__init__()` 메서드는 객체의 속성을 초기화하며, 객체가 생성된 직후 호출된다. `__del__()` 메서드는 객체가 파괴되기 직전에 호출된다. 이 메서드는 객체가 더 이상 사용되지 않는 경우에만 호출된다. `del x`문은 객체의 참조 횟수를 감소시킬 뿐, 반드시 `__del__()` 함수의 호출로 이어지지는 않는다. 이 메서드들은 7장에서 더 자세하게 다룬다.

`__new__()`와 `__init__()` 메서드는 새 인스턴스를 생성하고 초기화하는 데 함께 사용된다. `A(args)`를 호출해서 객체를 생성하면 다음 단계가 수행된다.

표 3.11 객체 생성과 파괴를 위한 특수 메서드

메서드	설명
<code>__new__(cls [, *args [, **kwargs]])</code>	새 인스턴스를 생성하기 위해 호출되는 클래스 메서드
<code>__init__(self [, *args [, **kwargs]])</code>	새 인스턴스를 초기화하기 위해 호출된다.
<code>__del__(self)</code>	인스턴스가 파괴될 때 호출된다.

```
x = A.__new__(A, args)
if isinstance(x, A): x.__init__(args)
```

사용자 정의 객체에서 `__new__()`나 `__del__()`을 정의하는 일은 드물다. `__new__()`는 보통 변경 불가능한 타입(정수, 문자열, 튜플 등) 중 하나로부터 상속을 받은 사용자 정의 객체나 메타클래스에서만 사용된다. `__del__()`은 락을 해제하거나 연결을 끊는 등 중요한 자원 관리 문제를 다루어야 하는 상황에서만 정의된다.

객체의 문자열 표현

표 3.12의 메서드들은 객체의 다양한 문자열 표현을 생성하는 데 사용된다.

표 3.12 객체 표현을 위한 특수한 메서드

메서드	설명
<code>__format__(self, format_spec)</code>	포맷이 적용된 표현을 생성한다.
<code>__repr__(self)</code>	객체의 문자열 표현을 생성한다.
<code>__str__(self)</code>	객체의 간단한 문자열 표현을 생성한다.

`__repr__()`와 `__str__()` 메서드는 객체의 간단한 문자열 표현을 생성한다. 일반적으로 `__repr__()` 메서드는 평가될 경우 객체를 재생성하는 표현식 문자열을 반환한다. 대화식 인터프리터에서 변수를 살펴볼 때 여러분이 보게 되는 출력 값을 생성하는 메서드이기도 하다. 이 메서드는 내장 `repr()` 함수에 의해 호출된다. 다음은 `repr()`와 `eval()`을 함께 사용하는 예이다.

```
a = [2,3,4,5]    # 리스트를 생성한다.
s = repr(a)      # s = '[2, 3, 4, 5]'
b = eval(s)       # s를 다시 리스트로 되돌린다.
```

문자열 표현식을 생성할 수 없는 경우에는 다음과 같이 `__repr__()`에서 `<...메시지...>` 형태의 문자열을 반환하는 것이 관례이다.

```
f = open("foo")
a = repr(f)      # a = "<open file 'foo', mode 'r' at dc030>"
```

`__str__()` 메서드는 내장 함수 `str()` 또는 출력과 관련된 함수에 의해서 호출된다. `__str__()`은 `__repr__()`에 비해서 더 간결하며 사용자에게 정보를 제공하는 문자열을 반환한다. `__str__()`이 정의되어 있지 않으면 `__repr__()`가 호출된다.

`__format__()` 메서드는 `format()` 함수 또는 문자열의 `format()` 메서드에 의해 호출된다. `format_spec` 인수는 포맷 지정자를 담는 문자열이다. 형식은 `format()`의 인수와 동일하다. 다음은 한 예이다.

```
format(x,"spec")          # x.__format__("spec")를 호출한다.
"x is {0:spec}".format(x)  # x.__format__("spec")를 호출한다.
```

포맷을 지정하는 문법은 고정된 것이 아니며 객체마다 다르게 정의할 수 있다. 표준 문법은 4장에서 설명한다.

객체 비교와 순서 매기기

표 3.13은 객체에 대해 간단한 검사를 수행하는 메서드를 보여준다. `__bool__()` 메서드는 진리값 검사를 수행하는 데 사용되며 `True` 또는 `False`를 반환해야 한다. 이 메서드가 정의되지 않을 경우에는 진리값을 결정하기 위해서 `__len__()` 메서드가 대신 사용된다. `__hash__()` 메서드는 사전에서 키로서 쓰일 수 있는 객체에서 정의해야 하는 메서드이다. 반환되는 값은 동일한 두 객체에 대해서 동일한 값을 갖는 정수여야 한다. 또한, 변경 가능한 객체에서는 이 메서드를 정의하면 안 된다. 객체에 변경이 가해지면 해시 값이 변경되어 이후의 사전 검색에서 객체를 찾을 수 없기 때문이다.

표 3.13 객체 검사 및 해싱을 위한 특수 메서드

메서드	설명
<code>__bool__(self)</code>	진리값 검사를 위해 <code>False</code> 나 <code>True</code> 를 반환
<code>__hash__(self)</code>	정수 해시 색인을 계산

객체는 하나 이상의 관계 연산자(`<`, `>`, `<=`, `>=`, `==`, `!=`)를 구현할 수 있다. 각 메서드는 두 개의 인수를 받으며, 불리언 값, 리스트, 또는 다른 아무 파이썬 타입 등 어떤 종류의 객체이든 반환할 수 있다. 예를 들어, 수치 관련 패키지의 경우 두 행렬의 원소별 비교를 수행할 때 그 결과로서 행렬을 반환할 수도 있다. 비교가 수행될 수 없는 경우에는 예외가 발생되기도 한다. 표 3.14는 비교 연산자를 위한 특수 메서드를 보여준다.

표 3.14 비교를 위한 메서드

메서드	결과
<code>__lt__(self,other)</code>	<code>self < other</code>
<code>__le__(self,other)</code>	<code>self <= other</code>
<code>__gt__(self,other)</code>	<code>self > other</code>
<code>__ge__(self,other)</code>	<code>self >= other</code>
<code>__eq__(self,other)</code>	<code>self == other</code>
<code>__ne__(self,other)</code>	<code>self != other</code>

표 3.14에 나와 있는 모든 연산을 구현할 필요는 없다. ==로 객체를 비교하거나 사전의 키로서 객체를 사용하려면 `__eq__()` 메서드를 정의해야 한다. 객체를 정렬하거나 `min()`이나 `max()` 같은 함수를 사용하려면 적어도 `__lt__()`는 정의해야 한다.

타입 검사

표 3.15에 나와 있는 메서드는 `isinstance()`과 `issubclass()` 함수의 타입 검사의 작동 방식을 재정의하는 데 사용된다. 이 메서드들은 7장에 설명되어 있는 것처럼 추상 기반 클래스와 인터페이스를 정의하는 데 주로 사용된다.

표 3.15 타입 검사를 위한 메서드

메서드	결과
<code>__instancecheck__(cls, object)</code>	<code>isinstance(object, cls)</code>
<code>__subclasscheck__(cls, sub)</code>	<code>issubclass(sub, cls)</code>

속성 접근

표 3.16에 나와 있는 메서드들은 점(.) 연산자나 `del` 연산자를 사용해서 객체의 속성을 읽거나 쓰거나 삭제하는 데 사용한다.

표 3.16 속성 접근을 위한 특수 메서드

메서드	설명
<code>__getattribute__(self, name)</code>	<code>self.name</code> 속성을 반환한다.
<code>__getattr__(self, name)</code>	보통의 속성 검색으로 찾을 수 없는 경우에 <code>self.name</code> 속성을 반환하거나 <code>AttributeError</code> 예외를 발생시킨다.
<code>__setattr__(self, name, value)</code>	<code>self.name = value</code> 속성을 설정한다. 보통의 방식대로 속성을 설정하는 대신 이 메서드가 호출된다.
<code>__delattr__(self, name)</code>	<code>self.name</code> 속성을 삭제한다.

속성이 접근될 때마다 `__getattribute__()` 메서드가 항상 호출된다. 속성이 발견되면 반환된다. 그렇지 않은 경우 `__getattr__()` 메서드가 호출된다. `__setattr__()` 메서드는 속성을 설정할 때마다 항상 호출되고 `__delattr__()` 메서드는 속성을 삭제할 때마다 항상 호출된다.

속성 감싸기 및 기술자

속성 조작과 관련해서 다소 까다로운 측면은 종종 객체의 속성들이 이전 절에서 설명한 get, set, delete 연산과 상호 작용하는 추가 로직 계층으로 감싸지는 경우가 있다는 점이다. 이런 식의 계층은 표 3.17에 나와 있는 메서드 중 하나 이상을 구현하는 기술자(descriptor) 객체를 생성함으로써 만들어진다. 기술자를 사용할지의 여부는 선택적이며 보통 이를 직접 정의하는 일은 드물다.

표 3.17 기술자 객체를 위한 특수 메서드

메서드	설명
<code>__get__(self, instance, cls)</code>	instance의 속성 값을 반환한다. 만약 instance가 Homeo라면 self를 반환한다.
<code>__set__(self, instance, value)</code>	value를 속성에 설정한다
<code>__delete__(self, instance)</code>	속성을 삭제한다

기술자 `__get__()`, `__set__()` 및 `__delete__()` 메서드는 클래스와 타입에 대한 `__getattribute__()`, `__setattr__()` 및 `__delattr__()` 메서드의 기본 구현과 상호 작용한다. 이러한 상호 작용은 사용자 정의 클래스의 몸체에 기술자 객체의 인스턴스를 둘 경우 일어난다. 이 경우 기술자의 속성에 접근하면 항상 기술자 객체의 대응되는 메서드가 암묵적으로 호출된다. 보통, 기술자는 뮤인 메서드, 안 뮤인 메서드, 클래스 메서드, 정적 메서드, 프로퍼티 등 객체 시스템의 하위 수준의 기능을 구현하는 데 사용된다. 더 많은 예는 7장에서 찾아볼 수 있다.

순서열 및 매팅 메서드

표 3.18 순서열과 매팅을 위한 메서드

메서드	설명
<code>__len__(self)</code>	self의 길이를 반환한다.
<code>__getitem__(self, key)</code>	self[key]를 반환한다.
<code>__setitem__(self, key, value)</code>	self[key] = value를 반환한다.
<code>__delitem__(self, key)</code>	self[key]를 삭제한다.
<code>__contains__(self, obj)</code>	obj가 self에 있으면 True를 반환한다. 아닐 경우 False를 반환한다.

표 3.18에 나와 있는 메서드는 순서열이나 매핑 객체를 흉내 내기를 원하는 객체에 의해서 사용된다.

다음은 한 예이다.

```
a = [1,2,3,4,5,6]
len(a)      # a.__len__( )
x = a[2]    # x = a.__getitem__(2)
a[1] = 7    # a.__setitem__(1,7)
del a[2]    # a.__delitem__(2)
5 in a     # a.__contains__(5)
```

`__len__()` 메서드는 내장 `len()` 함수에 의해 호출되며 음수가 아닌 길이를 반환한다. 이 함수는 `__bool__()` 메서드가 정의되지 않은 경우 진리값을 결정하는 데에도 사용된다.

개별 항목 조작과 관련해서 `__getitem__()` 메서드는 키로 항목을 찾아서 반환한다. 키는 아무 파이썬 객체나 될 수 있지만 순서열에 대해서는 보통 정수가 쓰인다. `__setitem__()` 메서드는 원소에 값을 할당한다. `__delitem__()` 메서드는 단일 원소에 `del` 연산이 적용될 때마다 호출된다. `__contains__()` 메서드는 `in` 연산자를 구현하는 데 사용된다.

`x = s[i:j]` 같은 분할 연산도 `__getitem__(), __setitem__()` 및 `__delitem__()`으로 구현된다. 하지만, 분할 연산에 대해서는 특수한 slice 객체가 키로서 전달된다. 이 객체는 요청된 분할의 범위를 설명하는 속성을 갖는다. 다음은 한 예이다.

```
a = [1,2,3,4,5,6]
x = a[1:5]    # x = a.__getitem__(slice(1,5,None))
a[1:3] = [10,11,12]  # a.__setitem__(slice(1,3,None), [10,11,12])
del a[1:4]    # a.__delitem__(slice(1,4,None))
```

파이썬의 분할 기능은 실제로는 많은 프로그래머들이 생각하는 것보다 더 강력하다. 예를 들어, 다음에 나오는 다양한 종류의 확장 분할들을 모두 지원하며 이들은 NumPy와 같은 서드파티 확장 모듈(third-party extension)에서 행렬이나 배열 같은 다차원 데이터 구조를 다루는 데 유용하게 쓰인다.

```
a = m[0:100:10]          # 보폭 분할 (보폭=10)
b = m[1:10, 3:20]        # 다차원 분할
c = m[0:100:10, 50:75:5] # 보폭 다차원 분할
m[0:5, 5:10] = n         # 확장 분할 대입
del m[:10, 15:]          # 확장 분할 삭제
```

확장 분할에서 각 차원의 일반적인 형식은 `i:j:stride`이며 `stride`는 생략이 가능하다. 보통의 분할에 대해서는 분할의 각 부분에 대해 시작이나 끝 값을 생략할 수 있다. 또한, 생략 부호(...로 씀)는 확장 분할에서 임의의 개수의 앞쪽 혹은 뒤쪽 차원

을 표시하는 데 사용된다.

```
a = m[..., 10:20]    # Ellipsis를 사용한 확장 분할
m[10:20, ...] = n
```

확장 분할을 사용할 때 `__getitem__()`, `__setitem__()` 및 `__delitem__()` 메서드가 각각 접근, 수정, 삭제를 구현하는 데 사용된다. 이 경우 전달되는 값은 정수가 아니라 slice나 Ellipsis 객체의 조합을 담은 튜플이다. 다음은 한 예이다.

```
a = m[0:10, 0:100:5, ...]
```

앞의 문장을 실행하면 다음과 같이 `__getitem__()`이 호출된다.

```
a = m.__getitem__((slice(0,10,None), slice(0,100,5), Ellipsis))
```

4장에서 설명되듯이 파이썬의 문자열, 튜플, 리스트는 확장 분할을 어느 정도 지원한다. 과학적인 용도와 같이 특수한 목적으로 파이썬을 확장하려고 하는 경우 고급 확장 분할 연산 기능을 제공하는 타입이나 객체를 정의할 수도 있다.

반복

객체 `obj`가 반복을 지원하면 `obj`는 반복자 객체를 반환하는 `obj.__iter__()` 메서드를 제공해야 한다. 반복자 객체 `iter`는 다음 번 객체를 반환하거나 반복의 끝을 알리는 `StopIteration` 예외를 발생시키는 단일 메서드 `iter.next()`를 구현해야 한다(파이썬 3에서는 `iter.__next__()`). 두 메서드 모두 `for`문과 기타 암묵적으로 반복을 수행하는 연산을 구현하는 데 사용된다. 예를 들어, `for x in s`문은 다음에 나오는 것과 동일한 단계들을 수행하는 것과 같다.

```
_iter = s.__iter__( )
while 1:
    try:
        x = _iter.next()      # 파이썬 3에서는 _iter.__next__( )
    except StopIteration:
        break
    # for 루프의 몸체에 있는 문장들을 실행한다.
    ...
```

수학 연산

표 3.19는 어떤 객체가 숫자를 흉내 내기 위해 반드시 구현해야 하는 특수 메서드들을 나열한 것이다. 수학 연산은 4장에서 설명되는 우선순위 규칙에 따라 항상 왼쪽에서 오른쪽으로 평가된다. $x + y$ 같은 표현식을 만나면 인터프리터는 `x.__add__(y)`

(y) 메서드를 호출하려고 시도한다. r로 시작하는 특수 메서드는 피연산자의 순서를 반대로 한 연산을 지원한다. 이러한 메서드는 왼쪽 피연산자가 해당 연산을 지원하지 않는 경우에만 호출된다. 예를 들어, $x + y$ 에서 x 가 `__add__()` 메서드를 지원하지 않으면 인터프리터는 `y.__radd__(x)` 메서드를 호출하려고 시도한다.

표 3.19 수학 연산을 위한 메서드

메서드	결과
<code>__add__(self, other)</code>	<code>self + other</code>
<code>__sub__(self, other)</code>	<code>self - other</code>
<code>__mul__(self, other)</code>	<code>self * other</code>
<code>__div__(self, other)</code>	<code>self / other</code> (파이썬 2에만 있음)
<code>__truediv__(self, other)</code>	<code>self / other</code> (파이썬 3)
<code>__floordiv__(self, other)</code>	<code>self // other</code>
<code>__mod__(self, other)</code>	<code>self % other</code>
<code>__divmod__(self, other)</code>	<code>divmod(self,other)</code>
<code>__pow__(self, other [, modulo])</code>	<code>self ** other, pow(self, other, modulo)</code>
<code>__lshift__(self, other)</code>	<code>self << other</code>
<code>__rshift__(self, other)</code>	<code>self >> other</code>
<code>__and__(self, other)</code>	<code>self & other</code>
<code>__or__(self, other)</code>	<code>self other</code>
<code>__xor__(self, other)</code>	<code>self ^ other</code>
<code>__radd__(self, other)</code>	<code>other + self</code>
<code>__rsub__(self, other)</code>	<code>other - self</code>
<code>__rmul__(self, other)</code>	<code>other * self</code>
<code>__rdiv__(self, other)</code>	<code>other / self</code> (파이썬 2에만 있음)
<code>__rtruediv__(self, other)</code>	<code>other / self</code> (파이썬 3)
<code>__rfloordiv__(self, other)</code>	<code>other // self</code>
<code>__rmod__(self, other)</code>	<code>other % self</code>
<code>__rdivmod__(self, other)</code>	<code>divmod(other,other)</code>
<code>__rpow__(self, other)</code>	<code>other ** self</code>
<code>__rlshift__(self, other)</code>	<code>other << self</code>
<code>__rrshift__(self, other)</code>	<code>other >> self</code>

__rand__(self, other)	other & self
__ror__(self, other)	other self
__rxor__(self, other)	other ^ self
__iadd__(self, other)	self += other
__isub__(self, other)	self -= other
__imul__(self, other)	self *= other
__idiv__(self, other)	self /= other (파이썬 2에만 있음)
__itruediv__(self, other)	self /= other (파이썬 3)
__ifloordiv__(self, other)	self // other
__imod__(self, other)	self %= other
__ipow__(self, other)	self **= other
__iand__(self, other)	self &= other
__ior__(self, other)	self = other
__ixor__(self, other)	self ^= other
__ilshift__(self, other)	self <<= other
__irshift__(self, other)	self >>= other
__neg__(self)	-self
__pos__(self)	+self
__abs__(self)	abs(self)
__invert__(self)	~self
__int__(self)	int(self)
__long__(self)	long(self) (파이썬 2에만 있음)
__float__(self)	float(self)
__complex__(self)	complex(self)

메서드 __iadd__(), __isub__() 등은 $a+=b$ 나 $a=b$ 같은 제자리(in-place) 산술 연산자(확장 대입(augmented assignment)라고도 부른다)를 지원하는 데 사용된다. 이러한 연산자는 성능 최적화 등을 위해 나름대로 그 구현을 달리할 수 있도록 하기 위해서 보통의 산술 연산자와는 구별되어 있다. 예를 들어, self 매개변수가 공유되지 않는다면 결과 값을 저장하기 위해 새로운 객체를 만들 필요 없이 객체의 값을 직접 수정하는 것을 생각해 볼 수 있다.

세 종류의 나누기 연산자인 __div__(), __truediv__(), __floordiv__()는 순수 나누기(true division)(/)와 끝수를 버리는 나누기(truncating division)(//)를 구현하

는 데 사용된다. 이렇게 세 가지 연산이 존재하는 이유는 파이썬 2.2에서 시작되었고 파이썬 3에서 기본이 된 정수 나누기의 기본 작동 방식의 변화와 관련이 있다. 파이썬 2에서는 파이썬의 기본 작동 방식이 / 연산자를 `__div__()`로 대응시키는 것이다. 정수에 대해서는 이 연산이 결과의 끝수를 버려 정수로 만든다. 파이썬 3에서는 나누기가 `__truediv__()`로 대응되고 정수에 대해서 실수가 반환된다. 나중의 방식은 파이썬 2에서도 프로그램에 `from __future__ import문`을 포함시킴으로써 선택적으로 활성화시킬 수 있다.

변환 메서드인 `__int__()`, `__long__()`, `__float__()`와 `__complex__()`는 객체를 네 가지 내장 타입 중 하나로 변환한다. 이 메서드들은 `int()`나 `float()` 같은 명시적 타입 변환에 의해 호출된다. 이들은 수학 연산에서 암묵적으로 강제 타입 변환을 수행하는 데 사용되지는 않는다. 예를 들어, 표현식 $3 + x$ 는 x 가 정수 변환을 위해 `__int__()`를 정의하고 있는 사용자 정의 객체라 할지라도 `TypeError` 예외를 생성한다.

호출가능 인터페이스

객체는 `__call__(self [*args, **kwargs])` 메서드를 제공함으로써 함수를 흉내 낼 수 있다. 객체 x 가 이 함수를 제공하면 함수처럼 호출이 가능하다. 즉, $x(arg1, arg2, \dots)$ 는 $x.__call__(self, arg1, arg2, \dots)$ 를 호출한다. 함수를 흉내 내는 객체는 함수기(function)나 대리자(proxy)를 생성하는 데 유용하다. 다음은 한 예이다.

```
class DistanceFrom(object):
    def __init__(self, origin):
        self.origin = origin
    def __call__(self, x):
        return abs(x - self.origin)

nums = [1, 37, 42, 101, 13, 9, -20]
nums.sort(key=DistanceFrom(10))      # 10에서의 거리로 정렬
```

이 예에서 `DistanceFrom` 클래스는 단일 인수를 받는 함수를 흉내 내는 인스턴스를 생성한다. 이 인스턴스는 보통의 함수 자리에 대신 쓰일 수 있다. 이 예에서는 `sort()` 함수를 호출할 때 사용되었다.

컨텍스트 관리 프로토콜

`with`문은 컨텍스트 관리자(context manager)라고 불리는 객체의 제어 하에서 일련

의 문장들을 실행하는 데 사용된다. 일반적인 문법은 다음과 같다.

```
with context [ as var]:  
    문장들
```

여기서 context 객체는 표 3.20에 나와 있는 메서드를 구현해야 한다. `__enter__()` 메서드는 `with`문이 실행될 때 호출된다. 이 메서드에 의해 반환되는 값은 옵션인 `as var` 지정자로 지정된 변수에 담기게 된다. `__exit__()` 메서드는 제어 흐름이 `with`문과 연관된 문장들의 블록으로부터 벗어나자마자 호출된다. `__exit__()`는 예외가 발생한 경우 예외 타입, 값 및 역추적 정보를 인수로서 넘겨 받게 된다. 만약 아무런 에러도 처리되지 않은 경우라면 세 값 모두 `None`으로 설정된다.

표 3.20 컨텍스트 관리자를 위한 특수 메서드

메서드	설명
<code>__enter__(self)</code>	새로운 컨텍스트에 들어설 때 호출된다. 반환되는 값은 <code>with</code> 문에서 <code>as</code> 지정자로 지정된 변수에 담기게 된다.
<code>__exit__(self, type, value, tb)</code>	컨텍스트를 벗어날 때 호출된다. 예외가 발생되었으면, <code>type</code> , <code>value</code> 및 <code>tb</code> 는 예외 타입, 값, 역추적 정보를 담게 된다. 컨텍스트 관리 인터페이스를 사용하는 주된 이유는 열린 파일, 네트워크 연결이나 락 같은 시스템의 상태와 관련된 객체에 대해 손쉬운 자원 관리를 수행하기 위해서이다. 이 인터페이스를 구현함으로써 객체는 객체가 사용 중인 컨텍스트를 벗어날 경우 안전하게 자원들을 정리할 수 있게 된다. 더 자세한 내용은 5장에서 찾아볼 수 있다.

객체 검사와 `dir()`

`dir()` 함수는 객체를 검사하는 데 주로 사용된다. 객체는 `__dir__(self)`를 구현함으로써 `dir()`에 의해서 반환되는 이름 목록을 제공할 수 있다. 이를 정의하면 사용자에 의해서 직접적으로 접근되기를 원하지 않는 객체의 내부 사항을 감추기가 쉽다. 그래도 사용자는 여전히 인스턴스나 객체의 `__dict__` 속성을 검사함으로써 정의된 모든 것을 볼 수 있다.

4장

P y t h o n E s s e n t i a l R e f e r e n c e

연산자와 표현식

이 장에서는 파이썬의 내장 연산자, 표현식, 평가 규칙에 대해 살펴본다. 이 장에서는 주로 이들을 파이썬의 내장 타입과 관련지어서 설명한다. 하지만, 사용자 정의 객체에서도 나름대로의 기능을 제공하기 위해 이 장에 나오는 연산자들 중 어느 것이라도 재정의할 수 있다.

숫자에 대한 연산

다음의 연산들은 모든 숫자 타입에 적용될 수 있다.

연산	설명
$x + y$	더하기
$x - y$	빼기
$x * y$	곱하기
x / y	나누기
$x // y$	끝수를 버리는 나누기
$x ** y$	제곱(x^y)
$x \% y$	나머지($x \bmod y$)
$-x$	단항 마이너스
$+x$	단항 플러스

끝수를 버리는 나누기 연산자 ($//$, 바닥 나누기)(floor division)라고도 부른다)는 결과의 끝수를 버림으로써 정수를 만들며, 부동 소수점 수와 정수 모두에 적용될 수 있다. 파이썬 2에서는 나누기 연산자($/$)도 피연산자가 정수일 경우 결과를 잘라낸다. 따라서, $7/4$ 는 1.75가 아닌 1이 된다. 하지만, 파이썬 3에서는 파이썬 2와 달리

이러한 경우 부동 소수점 수를 생성한다. 나머지 연산자를 사용하면 $x // y$ 연산 결과의 나머지를 구할 수 있다. 예를 들어, $7 \% 4$ 는 3이 된다. 부동 소수점 수일 경우, 나머지 연산자는 $x // y$ 의 계산 결과에서 부동 소수점 수인 나머지, 즉 $x - (x // y) * y$ 를 반환한다. 복소수에 대해서 나머지 연산자와 끝수를 버리는 나누기 연산자는 유효하지 않다.

아래의 이동(shift) 연산자와 비트(bitwise) 논리 연산자는 정수에만 적용될 수 있다.

연산	설명
$x << y$	왼쪽 이동
$x >> y$	오른쪽 이동
$x & y$	비트 and
$x y$	비트 or
$x ^ y$	비트 xor(exclusive or)
$\sim x$	비트 negation

비트 연산자는 정수가 2의 보수 이진 표현(complement binary representation)으로 나타내고 부호 비트는 왼쪽으로 무한히 확장된다고 가정한다. 하드웨어에 특화된 정수로 대응시키기 위해 무가공 비트 패턴을 다루는 경우라면 약간의 주의가 요구된다. 파이썬은 비트를 자르지 않으며 값의 오버플로우를 허용하지 않는다. 즉, 파이썬에서 결과 값은 얼마든지 커질 수 있다.

다음의 내장 함수들은 모든 숫자 타입에 대해 사용할 수 있다.

함수	설명
<code>abs(x)</code>	절대값
<code>divmod(x,y)</code>	$(x // y, x \% y)$ 를 반환한다.
<code>pow(x,y [,modulo])</code>	$(x^{**}y) \% \text{modulo}$ 를 반환한다.
<code>round(x,[n])</code>	10^{-n} 의 가장 가까운 수로 반올림(부동 소수점만 가능)

`abs()` 함수는 주어진 수의 절대값을 반환한다. `divmod()` 함수는 x를 y로 나눈 몫과 나머지를 반환하며, 복소수 이외의 수에 대해서만 사용할 수 있다. `pow()` 함수는 ** 연산자 대신 사용할 수 있고 3항 제곱 모듈로(ternary power-modulo) 함수(암호 알고리즘에서 종종 사용된다)도 지원한다. `round()` 함수는 부동 소수점 수 x를 10의 $-n$ 제곱에 가장 가까운 배수로 반올림한다. n을 입력하지 않으면 n은 0으로 설정된다. x가 두 배수에 동일하게 가까우면, 파이썬 2에서는 x가 0에서 더 멀리 떨어진 배수로 반올림된다. 예를 들어, 0.5는 1.0으로 반올림되며, -0.5는 -1.0으로 반

올림된다. 한 가지 주의할 점은 파이썬 3에서는 어떤 수가 두 배수에 동일하게 가까울 경우 가장 가까운 짹수인 배수로 반올림된다는 점이다. 예를 들어, 0.5의 경우 0.0으로 반올림되며, 1.5는 2.0으로 반올림된다. 이 때문에 수학 관련 프로그램을 파이썬 3로 포팅할 때는 주의가 요구된다.

다음의 비교 연산자들은 보통의 수학적인 의미를 지니며, 값이 참이면 불리언 값 True를, 거짓이면 불리언 값 False를 반환한다.

연산	설명
$x < y$	~보다 작은
$x > y$	~보다 큰
$x == y$	~와 같은
$x != y$	~와 다른
$x >= y$	~보다 크거나 같은
$x <= y$	~보다 작거나 같은

$w < x < y < z$ 와 같이, 비교 연산은 연쇄적으로 연결될 수 있다. 이 표현식은 $w < x$ and $x < y$ and $y < z$ 로 평가된다. $x < y > z$ 와 같은 표현식을 사용할 수 있지만, 코드를 읽는 사람에게 혼란을 주기 쉽다(x와 z 간의 비교는 수행되지 않는다는 점을 주목하기 바란다). 복소수를 포함하는 비교 연산의 결과는 정의되어 있지 않으며, 비교를 수행할 경우 TypeError 예외가 발생한다.

수와 관련된 연산자는 피연산자들이 같은 타입일 경우에만 유효하다. 내장 숫자 타입에 대해서는 아래와 같이 한 타입을 다른 타입으로 변환하는 강제 타입 변환 (coercion) 연산이 수행된다.

- 어느 한쪽의 피연산자가 복소수인 경우, 나머지 피연산자가 복소수로 변환된다.
- 어느 한쪽의 피연산자가 부동 소수점 수인 경우, 나머지 피연산자가 실수로 변환된다.
- 그렇지 않을 경우, 모든 피연산자는 반드시 정수여야 하며, 변환은 수행되지 않는다.

사용자 정의 객체의 경우, 혼합 피연산자와 관련된 표현식의 결과는 객체의 구현에 따라 달라진다. 일반적으로 인터프리터는 암묵적인 타입 변환을 시도하지 않는다.

순서열에 대한 연산

다음의 연산자들은 문자열, 리스트, 튜플 등의 순서열에 대해 사용할 수 있다.

연산	설명
<code>s + r</code>	연결
<code>s * n, n * s</code>	<code>s</code> 에 대한 <code>n</code> 개 복사본을 만든다. <code>n</code> 은 정수.
<code>v1, v2, ..., vn = s</code>	변수 풀어헤치기(unpacking)
<code>s[i]</code>	색인
<code>s[i:j]</code>	분할
<code>s[i:j:stride]</code>	확장 분할
<code>x in s, x not in s</code>	멤버 검사
<code>for x in s:</code>	반복
<code>all(s)</code>	<code>s</code> 에 속한 모든 항목이 참이면 <code>True</code> 를 반환
<code>any(s)</code>	<code>s</code> 에 속한 항목 중 하나라도 참이면 <code>True</code> 를 반환
<code>len(s)</code>	길이
<code>min(s)</code>	<code>s</code> 에 있는 가장 작은 항목
<code>max(s)</code>	<code>s</code> 에 있는 가장 큰 항목
<code>sum(s [, initial])</code>	옵션인 초기 값과 함께 항목들을 합한 값

+ 연산자는 같은 타입의 두 순서열을 연결한다. `s * n` 연산자는 순서열의 `n`개 복사본을 생성한다. 생성된 복사본은 참조로 요소들을 복사하는 얇은 복사본(shallow copy)이다. 아래의 코드를 살펴보자.

```
>>> a = [3, 4, 5]
>>> b = [a]
>>> c = 4 * b
>>> c
[[3,4,5], [3,4,5], [3,4,5], [3,4,5]]
>>> a[0] = -7
>>> c
[[-7,4,5], [-7,4,5], [-7,4,5], [-7,4,5]]
```

리스트 `c`에 있는 모든 원소에 변경이 가해진 것을 주목하기 바란다. 앞의 코드에서는 먼저 리스트 `a`에 대한 참조가 리스트 `b`에 추가되었다. 다음으로 `b`가 복제될 때 `a`에 대한 참조가 추가적으로 4개 생성되었다. 마지막으로 `a`가 변경되었고 이것 이 `a`의 다른 모든 복사본으로 전파되었다. 이러한 순서열 곱하기의 결과는 종종 예상과 다르며, 이는 프로그래머의 의도와 다를 수 있다. 이 문제를 해결하기 위한 방법으로 `a`의 내용을 복사해서 순서열을 직접 생성하는 방법이 있다. 다음은 한 예다.

```
a = [3, 4, 5]
c = [list(a) for j in range(4)] # list( )는 list의 복사본을 생성한다.
```

표준 라이브러리의 copy 모듈을 사용하여 객체의 복사본을 생성할 수도 있다.

모든 순서열은 일련의 변수로 풀어헤칠 수 있다. 다음은 몇 가지 예다.

```
item = [3,4,5]
x,y,z = items # x = 3, y = 4, z = 5

letters = "abc"
x,y,z = letters # x = 'a', y = 'b', z = 'c'

datetime = ((5,19,2008), (10,30,"am"))
(month,day,year), (hour,minute,am_pm) = datetime
```

값을 변수로 풀어헤칠 경우, 변수의 개수는 반드시 순서열 항목의 개수와 일치해야 한다. 또한, 변수의 구조도 순서열의 구조와 동일해야 한다. 예를 들어, 앞의 코드의 마지막 줄에서는 값을 6 개의 변수로 나누고 이를 세 항목씩 묶어서 튜플 2개로 구성하였다. 이 구조는 오른쪽 순서열의 구조와 정확히 일치한다. 순서열을 변수로 풀어헤치는 것은 반복이나 생성기를 통해 만들어진 순서열을 포함한, 어떤 종류의 순서열에든 적용할 수 있다.

색인 연산자 `s[n]`은 순서열의 n번째 객체를 반환한다. 여기서 `s[0]`은 순서열의 첫 번째 객체를 나타낸다. 음수 인덱스로 순서열의 끝에서부터 문자들을 추출할 수 있다. 예를 들어, `s[-1]`은 가장 끝에 있는 항목을 반환한다. 범위 밖의 원소에 접근할 경우 `IndexError` 예외가 발생한다.

분할 연산자 `s[i:j]`는 순서열 `s`에서 색인 `k`($i \leq k < j$)에 있는 요소들로 구성되는 부분 순서열을 반환한다. `i`와 `j`는 정수이거나 긴 정수여야 한다. 시작 또는 끝 지점 색인이 생략되면 순서열의 시작 또는 끝 지점 색인을 지정한 것으로 간주된다. 음수 색인을 사용할 수도 있으며, 음수 색인은 순서열의 끝에서부터 계산된다. `i` 또는 `j`가 범위를 벗어난 값이면 첫 번째 항목 이전을 가리키는지 마지막 항목 이후를 가리키는지에 따라, 순서열의 시작이나 끝을 가리키는 것으로 간주된다.

분할 연산자에는 `s[i:j:stride]`와 같이 추가적으로 보폭(stride)을 지정할 수 있다. 이렇게 하면 원소들을 건너뛸 수 있다. 원소들을 건너뛰는 방식을 이해하는 데에는 약간의 주의가 필요하다. 보폭이 주어지면, `i`는 시작 인덱스가 되고, `j`는 끝 인덱스가 되며, `s[i], s[i+stride], s[i+2*stride]` 등에 해당하는 원소들로 구성되는 순서열이 생성된다(색인 `j`에 도달할 때까지이며, 색인 `j`에 해당하는 요소는 포함되지 않는다). 보폭으로 음수 값을 사용할 수도 있다. 시작 인덱스 `i`가 생략되었을 때, `stride`가 양수이면 `i`는 순서열의 시작 지점으로 설정되고 `stride`가 음수이면 순서열의 마지막 지점

으로 설정된다. 끝 인덱스 j 가 생략되었을 때, $stride$ 가 양수이면 순서열의 끝 지점으로 설정되고 $stride$ 가 음수이면 순서열의 시작 지점으로 설정된다. 다음 예를 살펴보자.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

b = a[::2]          # b = [0, 2, 4, 6, 8 ]
c = a[::-2]         # c = [9, 7, 5, 3, 1 ]
d = a[0:5:2]        # d = [0, 2, 4]
e = a[5:0:-2]       # e = [5, 3, 1]
f = a[:5:1]          # f = [0, 1, 2, 3, 4]
g = a[:5:-1]         # g = [9, 8, 7, 6]
h = a[5::-1]         # h = [5, 6, 7, 8, 9]
i = a[5::-1]         # i = [5, 4, 3, 2, 1, 0]
j = a[5:0:-1]        # j = [5, 4, 3, 2, 1]
```

`x in s` 연산자는 순서열 s 에 객체 x 가 포함되어 있는지를 검사하여 `True`나 `False`를 반환한다. 비슷하게, `x not in s` 연산자는 순서열 s 에 객체 x 가 포함되어 있지 않은지를 검사한다. 문자열의 경우, `in`과 `not in` 연산자는 부분 문자열에 적용할 수 있다. 예를 들어, ‘hello’ in ‘hello world’는 `True`가 된다. `in` 연산자는 와일드카드 또는 패턴 매칭을 지원하지 않는다. 이를 위해서는 정규 표현식 패턴 매칭을 지원하는 `re` 모듈 같은 라이브러리 모듈을 사용해야 한다.

연산자 `for x in s`는 순서열의 모든 원소에 대해 반복을 수행한다. 이 연산자에 관해서는 5장에서 더 자세하게 설명한다. `len(s)`는 순서열에 들어 있는 원소의 개수를 반환한다. `min(s)`와 `max(s)`는 각각 순서열에서 가장 작은 값과 큰 값을 반환한다. 이 두 연산자는 순서열에 들어 있는 원소들이 (연산자에 대해서 순서가 결정되는 경우에만 유효하다(예를 들어, 파일 객체들을 담는 리스트에서 가장 큰 값을 찾는다는 것은 말이 안 된다)). `sum(s)`는 s 에 있는 모든 원소의 합을 반환하며, 항목들이 숫자인 경우에만 작동한다. 추가적으로 초기 값을 `sum()`에 지정할 수 있다. 초기 값의 타입에 따라 `sum()`의 결과는 달라질 수 있다. 예를 들어, `sum(items, decimal.Decimal(0))`은 `Decimal` 객체를 반환한다(`decimal` 모듈에 관해서는 14장을 참고하기 바란다).

문자열과 튜플은 변경이 불가능하며 한번 생성되면 수정할 수 없다. 리스트는 다음의 연산자들을 사용하여 변경할 수 있다.

연산자	설명
<code>s[i] = x</code>	색인 대입
<code>s[i:j] = r</code>	분할 대입

<code>s[i:j:stride] = r</code>	확장된 분할 대입
<code>del s[i]</code>	원소 삭제
<code>del s[i:j]</code>	분할 삭제
<code>del s[i:j:stride]</code>	확장 분할 삭제

`s[i] = x` 연산자는 리스트의 i 번째 원소가 객체 x를 가리키도록 변경하고 객체 x의 참조 회수를 증가시킨다. 음수 색인은 리스트의 끝에 대해서 상대적으로 해석되고 범위를 벗어난 색인 위치에 값을 할당하려고 시도하면 IndexError 예외가 발생한다. 분할 대입 연산자 `s[i:j] = r`은 색인 $k(i \leq k < j)$ 의 위치에 존재하는 원소들을 순서 열 r의 원소들로 대치한다. 사용되는 색인들은 분할 연산자에서와 같은 값을 가져야 하며, 범위를 벗어난 경우 리스트의 시작 또는 끝 값으로 적절히 조정된다. 순서 열 s는 r에 속하는 모든 원소들을 수용하기 위해서 필요에 따라 확장 또는 축소된다. 다음 예를 살펴보자.

```
a = [1, 2, 3, 4, 5]
a[1] = 6          # a = [1, 6, 3, 4, 5]
a[2:4] = [10,11] # a = [1, 6, 10, 11, 5]
a[3:4] = [-1,-2,-3] # a = [1, 6, 10, -1, -2, -3, 5]
a[:] = [7,8]      # a = [7, 8] id(a)는 변경되지 않음
```

분할 대입은 추가적인 보폭 인수를 지원한다. 이때, 분할 대입의 사용 방식에는 약간의 제약이 있다. 오른쪽 편에 있는 인수의 원소 개수가 대치하려는 원소의 개수와 동일해야 한다. 다음의 예를 살펴보자.

```
a = [1, 2, 3, 4, 5]
a[1::2] = [10, 11]      # a = [1, 10, 3, 11, 5]
a[1::2] = [30, 40, 50]  # ValueError 예외가 발생한다. 왼쪽 편에 원소가 두 개만 있다.
```

`del s[i]` 연산자는 리스트의 i번째 원소를 삭제하고 참조 횟수를 하나 감소시킨다. `del s[i:j]` 연산자는 해당 분할 조각에 속하는 모든 원소를 삭제한다. `del s[i:j:stride]`와 같이 보폭을 지정할 수도 있다.

순서열은 `<`, `<=`, `=`, `==`, `!=` 연산자를 사용하여 비교할 수 있다. 두 개의 순서열을 비교할 때는 먼저 각 순서열의 첫 번째 원소들이 비교된다. 이들이 서로 다를 경우 비교 결과가 결정된다. 이들이 같으면, 각 순서열의 두 번째 원소들을 비교하게 되며, 이와 같은 일련의 과정은 서로 다른 두 원소가 발견되거나, 어느 한 순서열에 원소가 존재하지 않을 때까지 반복된다. 두 순서열 모두의 끝에 도달하면, 두 순서열은 동일한 것으로 간주된다. 순서열 a가 순서열의 b의 부분 순서열이면, $a < b$ 가 된다.

문자열은 사전 순서대로 비교된다. 각각의 문자에는 문자 집합(ASCII 또는 Unicode 등)에 따라 고유한 숫자 색인이 할당된다. 한 문자의 색인이 다른 문자의 색인보다 작으면, 한 문자가 다른 문자보다 작다. 문자의 순서를 정하는 규칙과 관련하여 한 가지 주의할 점은 앞서 살펴본 간단한 비교 연산자들이 로케일(locale)이나 언어 설정과는 관련이 없다는 점이다. 따라서, 특정 언어의 표준적인 관례에 따라 문자열의 순서를 정하고자 할 때는 이 연산자들을 사용하지 않는 것이 좋다.(더 자세한 정보는 `unicodedata`와 `locale` 모듈을 참고하기 바란다.)

문자열과 관련해서 유의할 점이 또 있다. 파이썬에는 바이트 문자열과 유니코드 문자열, 두 종류가 있다. 바이트 문자열은 보통 인코딩되어 있는 것으로 간주되는 반면 유니코드 문자열은 인코딩이 되어 있지 않은 무가공 문자 값들을 표현하는 것으로 간주된다. 이러한 이유로, 표현식에서나 비교를 수행할 때(바이트 문자열과 유니코드 문자열을 연결하기 위해서 + 연산자를 사용한다든지 문자열 비교를 위해서 == 연산자를 사용하는 등) 바이트 문자열과 유니코드 문자열을 절대 혼합해서 사용하면 안 된다. 파이썬 3에서는 두 타입을 혼합해서 사용할 경우 `TypeError` 예외가 발생한다. 하지만, 파이썬 2에서는 암묵적으로 바이트 문자열을 유니코드 문자열로 변환하려고 시도한다. 파이썬 2에서 이 부분은 많은 사람들이 설계상의 실수라고 생각하고 있으며, 종종 예기치 않은 예외를 발생시키거나 프로그램이 불가사의하게 작동하는 원인을 제공하기도 한다. 따라서, 끌치 아픈 일을 피하려면 순서별 연산에서 문자열 타입을 혼합해서 사용하지 않도록 한다.

문자열 포맷 지정

나머지 연산자 (`s % d`)는 포맷 문자열 `s`와 튜플 또는 매핑 객체(사전) `d`에 들어 있는 객체들이 주어졌을 때 포맷이 적용된 문자열을 생성한다. 이 연산자는 C 언어의 `sprintf()` 함수와 유사하게 작동한다. 포맷 문자열은 보통의 문자들과(변경되지 않은 채로 남겨짐) 변환 지정자들, 이 두 가지 객체들을 담는다. 각 변환 지정자는 튜플이나 매핑의 관련 원소를 나타내는 포맷이 적용된 문자열로 대체된다. `d`가 튜플이면, 변환 지정자의 개수는 `d`에 있는 객체의 개수와 반드시 동일하여야 한다. `d`가 매핑이면, 각 변환 지정자는 매핑의 유효한 키 이름과 연결되어야 한다(괄호를 사용하여 지정한다). 각 변환 지정자는 `%` 문자로 시작하고 표4.1에 나와 있는 변환 문자 중 하나로 끝난다.

표 4.1 문자열 포맷 변환 문자

문자	출력 포맷
d, i	십진수 정수 또는 긴 정수
u	부호 없는 정수 또는 긴 정수
o	8진수 또는 긴 정수
x	16진수 정수 또는 긴 정수
X	16진수 정수 (대문자)
f	부동 소수점 [-]m.ddddd
e	부동 소수점 [-]m.ddddde±xx
E	부동 소수점 [-]m.dddddeE±xx
g, G	지수가 -4보다 작거나 주어진 정밀도보다 크면 %e 또는 %E를 사용. 아니면 %f를 사용.
s	문자열 또는 아무 객체. 문자열을 생성하는 데는 str()가 사용됨.
r	repr()에 의해 생성되는 것과 동일한 문자열을 생성
c	단일 문자
%	상수 %

%와 변환 지정자 사이에는 다음의 변경자들이 아래에 나온 순서대로 나타날 수 있다.

- 필호로 둘러싸인 키 이름. 매핑 객체에서 특정 항목을 선택한다. 해당 원소가 존재하지 않으면 KeyError 예외가 발생한다.
- 다음 중 하나 또는 그 이상.
 - 부호: 왼쪽 정렬을 의미한다. 기본적으로 값은 오른쪽으로 정렬된다.
 - + 부호: 숫자 부호의 포함을 의미한다(양수일지라도).
 - 0: 0으로 채우는 것을 의미한다.
- 필드의 폭을 나타내는 숫자. 변환된 값은 적어도 지정된 폭 이상의 크기를 갖는 필드 안에 출력된다. 폭을 맞추기 위해 필드는 왼쪽부터(- 플래그가 주어지면 오른쪽부터) 채워진다.
- 정밀도와 필드의 폭을 구분하는 점.
- 문자열인 경우 출력할 최대 문자의 개수를, 부동 소수점 수일 경우 소수점 뒤에 나오는 숫자의 개수를, 정수일 경우 숫자의 최소 개수를 지정하는 숫자.

추가적으로 별표(*) 문자를 필드의 폭을 지정하는 숫자 대신 사용할 수 있다. 별표가 나타나면, 튜플에 있는 다음 항목의 값이 필드의 폭 값으로 사용된다.

다음 코드는 몇몇 예를 보여준다.

```
a = 42
b = 13.142783
c = "hello"
d = {'x':13, 'y':1.54321, 'z':'world'}
e = 5628398123741234

r = "a is %d" % a          # r = "a is 42"
r = "%10d %f" % (a,b)      # r = "           42 13.142783"
r = "%+010d %E" % (a,b)    # r = "+000000042 1.314278E+01"
r = "%(x)-10d %(y)0.3g" % d # r = "13           1.54"
r = "%0.4s %s" % (c, d['z']) # r = "hell world"
r = "%.*f" % (5,3,b)        # r = "13.143"
r = "e = %d" % e            # r = "e = 5628398123741234"
```

사전과 함께 사용될 때, 문자열 포맷 연산자 %는 스크립트 언어에서 종종 볼 수 있는 문자열 보간(string interpolation) 기능을 흉내 내는 데 사용될 수 있다(예를 들어, 문자열에서 \$var 기호를 확장한다든지). 예를 들어, 값들을 담은 사전이 있을 때, 다음처럼 포맷 문자열의 필드들을 해당 값들로 확장할 수 있다.

```
stock = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10 }

r = "%(shares)d of %(name)s at %(price)0.2f" % stock
# r = "100 shares of GOOG at 490.10"
```

다음 코드는 현재 정의되어 있는 변수 값들을 문자열 안에서 어떻게 확장하는지를 보여준다. vars() 함수는 vars()이 호출된 시점에서 정의되어 있는 모든 변수를 담는 사전을 반환한다.

```
name = "Elwood"
age = 41
r = "%(name)s is %(age)d years old" % vars()
```

고급 문자열 포맷 지정

문자열의 s.format (*args, **kwargs) 메서드를 사용하여 더 고급의 문자열 포맷 지정을 수행할 수 있다. 이 메서드는 임의의 개수의 위치 인수 및 키워드 인수를 받아들여서 s에 들어 있는 자리 표시자들을 교체한다. n이 숫자인 '{n}' 형태의 자리 표시

자는 format()에 지정된 위치 인수 n으로 대체된다. '{name}' 형태의 자리 표시자는 format()에 지정된 키워드 인수 name으로 대체된다. '{{'로 단일 '{'를, '}}'로 단일 '}'를 나타낸다. 다음은 한 예를 보여준다.

```
r = "{0} {1} {2}".format('GOOG', 100, 490.10)
r = "{name} {shares} {price}".format(name='GOOG', shares=100, price=490.10)
r = "Hello {0}, your age is {age}".format("Elwood", age=47)
r = "Use {{ and }} to output single curly braces".format()
```

자리 표시자에는 추가적으로 색인과 속성 검색을 지정할 수 있다. 예를 들어, 정수 n에 대해서 '{name[n]}'은 순서열 검색을 수행하고 숫자 아닌 문자열 key에 대해서 '{name[key]}'는 name['key'] 형태의 사전 검색을 수행한다. '{name.attr}'에 대해서는 속성 검색이 수행된다. 다음은 몇 가지 예다.

```
stock = {'name' : 'GOOG',
         'shares' : 100,
         'price' : 490.10 }
r = "{0[name]} {0[shares]} {0[price]}".format(stock)

x = 3 + 4j
r = "{0.real} {0.imag}".format(x)
```

색인 및 속성 검색에는 이름만 지정할 수 있다. 입의의 표현식, 메서드 호출, 다른 연산은 올 수 없다.

출력 결과를 더 세밀하게 제어하기 위해서 추가로 포맷 지정자를 사용할 수 있다. 이를 위해서 '{place:format_spec}'와 같이 각 자리 표시자에 콜론(:)을 사용하여 포맷 지정자를 추가로 지정하면 된다. 포맷 지정자를 사용해서 열의 폭, 소수점의 위치, 정렬 방식 등을 지정할 수 있다. 다음은 한 예다.

```
r = "{name:8} {shares:8d} {price:8.2f}".format
{name="GOOG", shares=100, price=490.10}
```

포맷 지정자의 일반적인 형식은 [[fill][align][sign][0][width][.precision][type]]으로, []로 둘러싸인 부분은 생략할 수 있다. width는 필드의 최소 폭을 지정하고, align은 '<', '>', '^' 중 하나가 될 수 있으며, 각각 왼쪽, 오른쪽, 가운데 정렬을 나타낸다. fill은 빈 공간을 채우기 위해 사용되는 문자를 지정한다. 다음은 몇 가지 예를 보여 준다.

```
name = "Elwood"
r = "{0:<10}".format(name)      # r = 'Elwood      '
r = "{0:>10}".format(name)      # r = '      Elwood'
r = "{0:^10}".format(name)       # r = '  Elwood  '
r = "{0:={}10)".format(name)     # r = '==Elwood=='
```

`type`은 데이터 타입을 지정한다. 표 4.2는 파이썬에서 지원하는 포맷 코드를 보여준다. `type`이 지정되지 않으면, 기본 포맷 코드로 문자열에 대해서는 ‘s’가, 정수에 대해서는 ‘d’가, 실수에 대해서는 ‘f’가 사용된다.

표 4.2 고급 문자열 포맷 지정에서 지원하는 타입 코드

문자	출력 포맷
d	십진수 정수 또는 긴 정수
b	이진 정수 또는 긴 정수
o	8진수 또는 긴 정수
x	16진수 정수 또는 긴 정수
X	16진수 정수 (대문자)
f, F	부동 소수점 [-]m.ddddddd
e	부동 소수점 [-]m.dddddddE±xx
E	부동 소수점 [-]m.dddddddE±xx
g, G	지수가 -4보다 작거나 주어진 정밀도보다 크면 e 또는 E를 사용. 아니면 f를 사용.
n	현재 로케일 설정에 따라 소수점을 나타내는 문자가 결정된다는 것을 제외하고는 g와 동일
%	숫자에 100을 곱한 후 그 값을 f 포맷으로 출력하고 끝에 %를 붙인다.
s	문자열 또는 아무 객체. 문자열을 생성하는 데는 <code>str()</code> 가 사용됨.
c	단일 문자

`sign`은 ‘+’, ‘-’, 또는 ‘ ’ 중 하나가 될 수 있다. ‘+’는 모든 수에 대해 앞쪽에 항상 부호 표시가 나타나야 한다는 것을 가리킨다. ‘-’는 기본 값으로서 음수에 대해서만 부호를 붙인다. ‘ ’는 양수에 대해 앞에 스페이스를 하나 집어 넣는다. `precision`은 십진수에 대한 정확도를 숫자의 개수로서 지정한다. 숫자의 경우, 필드의 폭을 지정하는 부분의 제일 앞에 ‘0’을 써주면, 앞쪽의 빈 공간이 0으로 채워진다. 다음은 숫자의 포맷을 지정하는 여러 가지 예를 보여준다.

```

x = 42
r = '{0:10d}'.format(x)      # r = '        42'
r = '{0:10x}'.format(x)      # r = '        2a'
r = '{0:10b}'.format(x)      # r = '        101010'
r = '{0:010b}'.format(x)     # r = '0000101010'

y = 3.1415926
r = '{0:10.2f}'.format(y)    # r = '        3.14'
r = '{0:10.2e}'.format(y)    # r = ' 3.14e+00'

```

```
r = '{0:+10.2f}'.format(y)      # r = '      +3.14'
r = '{0:+010.2f}'.format(y)    # r = '+000003.14'
r = '{0:+10.2%}'.format(y)     # r = ' +314.16%
```

선택적으로, 포맷 지정자의 일부를 format 함수에 지정된 인수를 이용해서 지정할 수 있다. 보통의 필드에 접근할 때 사용하는 문법을 그대로 쓰면 된다. 다음은 한 예다.

```
y = 3.1415926
r = '{0:{width}.{precision}f}'.format(y,width=10,precision=3)
r = '{0:{1}.{2}f}'.format(y,10,3)
```

이렇게 필드들을 중첩시키는 것은 오직 한 단계 깊이까지만 가능하고 포맷 지정자 부분에서만 지원된다. 또한, 중첩된 값들은 다시 내부적으로 추가적인 포맷 지정자를 지닐 수 없다.

객체는 자신만의 포맷 지정자 집합을 정의할 수 있다. 고급 문자열 포맷 지정이 수행될 때는 내부적으로 각 필드에 대해서 특수한 메서드인 `__format__(self, format_spec)`가 호출된다. 이처럼 `format()`의 기능은 변경이 가능하고 적용되는 객체에 따라 달라질 수 있다. 한 예로 날짜나 시간을 표현하는 객체에 나름대로의 포맷 코드를 정의하는 것을 생각해볼 수 있다.

어떤 경우에는 `__format__()` 메서드에 의해 구현된 부분을 사용하지 않고, 단순히 `str()` 또는 `repr()`에 의해 출력되는 결과를 사용하고 싶을 때가 있다. 이를 위해서는 포맷 지정자 앞에 `!s` 또는 `!r` 변경자를 추가하면 된다. 다음은 한 예다.

```
name = "Guido"
r = '{0!r:^20}'.format(name) # r = "        'Guido'        "
```

사전에 대한 연산

사전은 이름과 객체 사이의 매핑 기능을 제공한다. 사전에는 다음 연산들을 사용할 수 있다.

연산	설명
<code>x = d[k]</code>	키를 이용한 색인
<code>d[k] = x</code>	키를 이용한 할당
<code>del d[k]</code>	키를 이용한 항목 삭제
<code>k in d</code>	키의 존재 여부 검사
<code>len(d)</code>	사전에 들어 있는 항목의 개수

문자열, 숫자, 튜플 등과 같은 변경이 불가능한 객체는 무엇이든지 키 값으로 사용될 수 있다. 또한, 사전 키를 다음과 같이 콤마로 구분된 값들의 리스트로서 지정할 수도 있다.

```
d = { }
d[1,2,3] = "foo"
d[1,0,3] = "bar"
```

이 경우 키 값들은 튜플을 표현하게 되고, 이에 따라 앞의 할당문은 다음과 동일하게 된다.

```
d[(1,2,3)] = "foo"
d[(1,0,3)] = "bar"
```

집합에 대한 연산

`set`과 `frozenset` 타입은 다음과 같이 공통적으로 여러 집합 연산을 지원한다.

연산	설명
<code>s t</code>	<code>s</code> 와 <code>t</code> 의 합집합
<code>s & t</code>	<code>s</code> 와 <code>t</code> 의 교집합
<code>s - t</code>	차집합
<code>s ^ t</code>	대칭 차집합
<code>len(s)</code>	집합의 항목 개수
<code>max(s)</code>	가장 큰 값
<code>min(s)</code>	가장 작은 값

합집합, 교집합, 차집합 연산의 결과는 가장 왼쪽에 위치한 피연산자의 타입과 동일한 타입을 갖게 된다. 예를 들어, `s`가 `frozenset`일 경우 `t`가 `set`일지라도 연산 결과는 `frozenset`이 된다.

확장 대입

파이썬은 다음과 같은 확장 대입 연산자들을 지원한다.

연산	설명
<code>x += y</code>	$x = x + y$
<code>x -= y</code>	$x = x - y$
<code>x *= y</code>	$x = x * y$
<code>x /= y</code>	$x = x / y$
<code>x //= y</code>	$x = x // y$
<code>x **= y</code>	$x = x ** y$

x %≡ y	x = x % y
x &≡ y	x = x & y
x ≡ y	x = x y
x ^≡ y	x = x ^ y
x >>≡ y	x = x >> y
x <<≡ y	x = x << y

위 연산자들은 보통의 대입문이 사용되는 어느 곳에서든 사용할 수 있다. 다음은 몇 가지 예를 보여준다.

```
a = 3
b = [1,2]
c = "Hello %s %s"
a += 1           # a = 4
b[1] += 10      # b = [1, 12]
c %= ("Monty", "Python")    # c = "Hello Monty Python"
```

확장 대입은 그 구현 방식에 따라 객체의 내용을 직접(inplace) 변경하거나 별도의 객체를 생성할 수도 있다. 즉, $x += y$ 는 변경 가능한 객체 x 의 값을 직접 바꿀 수도 있고 또는 $x + y$ 의 값을 갖는 새로운 객체를 생성한 후 x 가 이 객체를 가리키게 할 수도 있다. 사용자 정의 클래스에서는 3장에 설명된 특수한 메서드를 사용해서 확장 대입 연산자를 재정의할 수 있다.

속성(.) 연산자

점(.) 연산자는 객체의 속성에 접근하는 데 사용된다. 다음은 한 예를 보여준다.

```
foo.x = 3
print foo.y
a = foo.bar(3,4,5)
```

단일 표현식 안에서 $foo.y.a.b$ 와 같이 둘 이상의 점 연산자가 쓰일 수 있다. 또한, 점 연산자는 $a = foo.bar(3,4,5).spam$ 과 같이 함수의 중간 결과에 적용될 수도 있다. 사용자 정의 클래스에서는 점 연산자를 재정의하거나 커스터마이즈할 수 있다. 자세한 내용은 3장과 7장을 참고하기 바란다.

함수 호출 () 연산자

$f(args)$ 연산자는 f 에 대해 함수 호출을 수행한다. 함수의 각 인수는 표현식이다. 함수가 호출되기 전에 모든 인수 표현식이 왼쪽에서 오른쪽으로 평가된다. 이를 적용

순 평가(applicative order evaluation)라고도 한다.

functools 모듈의 partial() 함수를 사용하여 함수 인수들을 일부만 평가하는 것이 가능하다. 다음은 한 예다.

```
def foo(x,y,z):
    return x + y + z

from functools import partial
f = partial(foo,1,2) # foo의 인수 x와 y에 값을 제공
result = f(3)       # foo(1,2,3)을 호출. result는 6.
```

partial() 함수는 함수의 인수들 중 일부분을 평가하고, 나중에 나머지 인수들을 제공해서 호출할 수 있는 객체를 반환한다. 앞의 예에서, 변수 f는 함수의 첫 번째, 두 번째 인수가 미리 계산된 부분적으로 평가된 함수를 나타낸다. 함수 실행을 위해서는 마지막 남은 인수만 제공하면 된다. 함수 인자들을 부분적으로 평가하는 것은 커링(currying)이라고 알려진 프로세스와 밀접한 관련이 있다. 커링은 여러 인수를 입력받는 함수를 하나의 인수만을 입력받는 일련의 함수로 분해하는 메커니즘이다(예를 들어, $f(x,y)$ 가 있을 때 x 를 고정시킴으로써 부분적으로 f 를 평가하고, 다시 여기에 y 의 값을 주어서 최종 결과를 생성하는 식으로).

변환 함수

내장 타입 간의 변환을 수행해야 할 때가 있다. 타입 간의 변환을 위해서는 간단히 타입 이름을 함수처럼 사용하면 된다. 몇몇 내장 함수들은 특수한 종류의 변환을 수행한다. 모든 변환 함수는 변환된 값을 표현하는 새로운 객체를 반환한다.

함수	설명
<code>int(x [,base])</code>	<code>x</code> 를 정수로 변환한다. <code>x</code> 가 문자열인 경우, <code>base</code> 는 기본수를 나타낸다.
<code>float(x)</code>	<code>x</code> 를 부동 소수점 수로 변환한다.
<code>complex(real [,imag])</code>	복소수를 생성한다.
<code>str(x)</code>	객체 <code>x</code> 를 문자열로 변환한다.
<code>repr(x)</code>	객체 <code>x</code> 를 표현식 문자열로 변환한다.
<code>format(x [,format_spec])</code>	객체 <code>x</code> 를 포맷이 적용된 문자열로 변환한다.
<code>eval(str)</code>	문자열을 평가하고 객체를 반환한다.
<code>tuple(s)</code>	<code>s</code> 를 튜플로 변환한다.
<code>list(s)</code>	<code>s</code> 를 리스트로 변환한다.
<code>set(s)</code>	<code>s</code> 를 집합으로 변환한다.
<code>dict(d)</code>	사전을 생성한다. <code>d</code> 는 (키, 값) 튜플들의 순서열이어야 한다.

<code>frozenset(s)</code>	s를 frozenset으로 변환한다.
<code>chr(x)</code>	정수를 문자로 변환한다.
<code>unichr(x)</code>	정수를 유니코드 문자로 변환한다.(파이썬 2에서만 사용 가능)
<code>ord(x)</code>	단일 문자를 정수값으로 변환한다.
<code>hex(x)</code>	정수를 16진수 문자열로 변환한다.
<code>bin(x)</code>	정수를 2진수 문자열로 변환한다.
<code>oct(x)</code>	정수를 8진수 문자열로 변환한다.

`str()`과 `repr()` 함수는 서로 다른 결과를 반환할 수 있다. `repr()`은 `eval()` 함수를 통해 평가될 경우 동일한 객체를 다시 생성해낼 수 있는 표현식 문자열을 반환한다. 반면에 `str()`은 객체를 표현하는 간결하고 포맷이 잘 적용된 문자열을 생성한다 (이 함수는 `print`문에 의해서 사용된다). `format(x, [format_spec])` 함수는 고급 문자열 포맷 지정 연산을 수행한 것과 동일한 결과를 생성하지만, 단일 객체 `x`에 대해 적용된다. 이 함수는 포맷 코드를 포함하는 문자열 `format_spce`을 부가적으로 받아들인다. `ord()` 함수는 문자의 정수 서열 값(ordinal value)을 반환한다. 유니코드의 경우 이 값은 정수 코드 포인트가 된다. 함수 `chr()`와 `unichr()`는 정수를 다시 문자로 변환한다.

함수 `int()`, `float()`, `complex()`는 문자열을 다시 숫자로 변환한다. `eval()` 함수는 유효한 표현식을 담은 문자열을 객체로 변환한다. 아래의 예를 살펴보자.

```
a = int("34")          # a = 34
b = long("0xfe76214", 16)    # b = 266822164L (0xfe76214L)
b = float("3.1415926")      # b = 3.1415926
c = eval("3, 5, 6")        # c = (3,5,6)
```

`list()`, `tuple()`, `set()` 같은 컨테이너를 생성하는 함수의 인수로는 반복을 지원하는 객체라면 어떤 객체든지 사용될 수 있다. 반복을 지원하는 객체는 컨테이너를 채울 항목들을 생성하는 데 사용된다.

불리언 표현식과 진리값

키워드 `and`, `or`, `not`은 불리언 표현식을 생성하는 데 사용된다. 이 연산자들의 의미는 다음과 같다.

연산자	설명
<code>x or y</code>	x가 거짓이면, y를 반환한다. 그렇지 않으면 x를 반환한다.
<code>x and y</code>	x가 거짓이면, x를 반환한다. 그렇지 않으면 y를 반환한다.
<code>not x</code>	x가 거짓이면 True를 반환한다. 그렇지 않으면 False를 반환한다.

참 또는 거짓을 결정하는 표현식에서 True, 영이 아닌 수, 비어 있지 않은 문자열, 리스트, 튜플 또는 사전은 참으로 간주된다. False, 0, None, 빈 리스트, 튜플, 사전은 거짓으로 평가된다. 불리언 표현식은 왼쪽에서 오른쪽으로 평가되며, 오른쪽 편의 피연산자는 최종 값을 평가하는 데 필요한 경우에만 사용된다. 예를 들어, a and b는 a가 참일 경우에만 b로 평가된다. 이러한 평가 방식을 단축 평가(short-circuit evaluation)라고 부르기도 한다.

객체 동등 및 신원

동등 연산자 ($x == y$)는 x의 값과 y의 값이 같은지를 평가한다. 리스트와 튜플의 경우, 모든 원소들이 비교되고 모두 같은 값을 가질 경우 참으로 평가된다. 사전의 경우, x와 y가 동일한 키 집합을 갖고 동일한 키를 가진 객체들이 모두 같을 경우에만 참으로 평가된다. 집합의 경우, 동등 연산자($==$)로 원소들을 비교했을 때 두 집합이 동일한 원소들을 가질 경우 동등하다고 평가된다.

신원 연산자($x \text{ is } y$ 와 $x \text{ is not } y$)의 경우, 두 객체가 메모리에 있는 동일한 객체를 가리키고 있는지를 검사한다. 일반적으로, $x == y$ 이지만 $x \text{ is not } y$ 인 경우도 있다.

파일과 부동소수점 수를 비교하는 것과 같이 호환성 없는 타입의 객체들을 비교할 수도 있다. 하지만, 어떤 결과가 나올지 모르며 말이 안 되는 경우도 있다. 타입에 따라서는 예외가 발생하기도 한다.

평가 순서

표 4.3은 연산들의 평가 순서(우선순위 규칙)를 보여준다. 제곱 연산자 ($**$)를 제외한 나머지 모든 연산자는 왼쪽에서 오른쪽으로 평가된다. 표 4.3은 연산자들을 우선순위가 높은 순에서 낮은 순으로 보여주고 있다. 즉, 표에서 먼저 나온 연산자가 나중에 나온 연산자보다 먼저 평가된다($x * y$, x / y , $x \% y$ 처럼 같은 항목에 속하는 연산자들은 동일한 우선순위를 갖는다).

표 4.3에 나와 있는 연산 순서는 x와 y의 타입에 따라 결정되는 것이 아니다. 따라서, 사용자 정의 객체에서 개별 연산자를 재정의하더라도 기본적인 연산 순서, 우선 순위, 결합 법칙을 변경할 수 없다.

표 4.3 평가 순서(내림차순)

연산자	이름
(...), [...], {...}	튜플, 리스트, 사전 생성
s[i], s[i:]	색인과 분할
s.attr	속성
f(...)	함수 호출
+x, -x, ~x	단항 연산자
x ** y	제곱. 우측 결합(right associative).
x * y, x / y, x // y, x % y	곱하기, 나누기, 끝수를 버리는 나누기, 나머지
x + y, x - y	더하기, 빼기
x << y, x >> y	비트 이동
x & y	비트 and
x ^ y	비트 xor(exclusive or)
x y	비트 or
x < y, x <= y, x > y, x >= y	비교, identity, 순서열 멤버 검사
x == y, x != y	
x is y, x is not y	
x in s, x not in s	
not x	논리 부정
x and y	논리 곱
x or y	논리 합
lambda args: expr	익명 함수

조건 표현식

프로그램을 작성할 때 표현식의 결과에 기반해서 값을 변수에 할당하는 경우가 종종 있다. 다음은 한 예다.

```
if a <= b:
    minvalue = a
else:
    minvalue = b
```

앞의 코드는 조건 표현식(conditional expression)을 사용하여 다음과 같이 줄여 쓸 수 있다.

```
minvalue = a if a <= b else b
```

조건 표현식에서 가운데에 있는 조건이 가장 먼저 평가된다. 평가 결과가 참이면

if 왼쪽에 있는 표현식이 평가되고 그렇지 않으면 else 다음의 표현식이 평가된다.

조건 표현식은 이해하기 어려울 수 있기 때문에 가급적 사용하지 않는 것이 좋다 (특히 다른 복잡한 표현식과 중첩되거나 섞여 있을 경우). 하지만, 다음과 같이 리스트 내포나 생성기 표현식 같은 곳에서 유용하게 쓰이기도 한다.

```
values = [1, 100, 45, 23, 73, 37, 69]
clamped = [x if x < 50 else 50 for x in values]
print(clamped) # [1, 50, 45, 23, 50, 37, 50]
```

5장

P y t h o n E s s e n t i a l R e f e r e n c e

프로그램 구조와 제어 흐름

이 장에서는 프로그램 구조와 제어 흐름에 대해 자세히 살펴본다. 조건문, 반복문, 예외, 그리고 컨텍스트 관리자(context manager)에 관한 내용을 다룰 것이다.

프로그램 구조와 실행

파이썬 프로그램은 일련의 문장들로 구성된다. 변수, 대입, 함수 정의, 클래스, 모듈 임포트(import) 등 언어의 구성 요소는 모두 문장이고 서로 동등하게 취급된다. 사실, 파이썬에는 ‘특수한’ 문장이란 것은 존재하지 않기 때문에 문장은 프로그램의 어느 곳이나 올 수 있다. 다음 코드에서는 한 함수의 두 가지 버전을 정의하고 있다.

```
if debug:
    def square(x):
        if not isinstance(x, float):
            raise TypeError("Expected a float")
        return x * x
else:
    def square(x):
        return x * x
```

소스 파일이 로드될 때 인터프리터는 문장이 더 없을 때까지 문장들을 순차적으로 실행한다. 간단히 메인 프로그램으로서 실행되는 파일이나 import를 통해 로드된 라이브러리 파일 모두 이러한 식으로 실행된다.

조건부 실행

if, else, elif 구문은 조건부 코드 실행을 제어한다. 조건문의 일반적인 형식은 다음과 같다.

```
if 표현식:  
    문장들  
elif 표현식:  
    문장들  
elif 표현식:  
    문장들  
...  
else:  
    문장들
```

실행할 것이 없는 경우에는 조건문의 else와 elif절을 모두 생략할 수 있다. 특정 절에 대해서 실행할 구문이 없는 경우에는 pass문을 사용한다.

```
if 표현식:  
    pass      # 아무 일도 하지 않는다.  
else:  
    문장들
```

루프와 반복

루프는 for와 while문을 사용해 구현한다. 다음 예를 살펴보자.

```
while 표현식:  
    문장들  
  
for i in s:  
    문장들
```

while문은 표현식이 거짓으로 평가될 때까지 문장들을 실행한다. for문은 원소가 더 이상 남아 있지 않을 때까지 s의 모든 원소에 대해 반복 실행을 수행한다. for문은 반복을 지원하는 객체라면 어떤 객체에 대해서든 사용할 수 있다. 이러한 객체에는 리스트, 튜플, 문자열 등의 기본 순서열 타입뿐만 아니라 반복자 프로토콜(iterator protocol)을 구현한 객체도 포함된다.

객체 s가 반복을 지원한다는 말은 for문의 구현을 흉내 내는 다음 코드에서처럼 사용될 수 있다는 것을 의미한다.

```
it = s.__iter__()      # s에 대한 반복자를 가져온다.  
while 1:  
    try:  
        i = it.next() # 다음 항목을 가져온다.(파이썬 3에서는 __next__)
```

```

except StopIteration: # 더 이상 항목이 없다.
    break
    # i에 대해 필요한 작업을 수행
...

```

for i in s 구문에서 변수 i를 반복 변수(iteration variable)라고 부른다. 각 반복마다 변수 i는 s로부터 새로운 값을 할당받는다. 반복 변수의 유효 범위는 for문 내부에만 한정되지 않는다. 즉, for문을 수행하기 이전에 반복 변수와 동일한 이름을 가진 변수가 정의되어 있었다면 그 변수의 값은 for문에 의해서 덮어 써진다. 또한, 반복 변수는 루프가 종료된 후에도 최종 값을 유지한다.

반복에서 사용되는 원소들이 동일한 크기를 가지는 순서열일 경우 다음 코드와 같이 개별적인 반복 변수로 풀어헤칠 수 있다.

```

for x,y,z in s:
    문장들

```

위 예에서 s는 반드시 각각 3개의 원소를 가지는 순서열을 담고 있거나 생성해야 한다. 반복할 때마다 변수 x, y, z에는 순서열의 요소들이 할당된다. 이 방식은 s가 튜플로 구성되는 순서열일 때 주로 사용되지만 s에 들어 있는 것이 리스트, 생성기, 문자열 등일 때도 적용할 수 있다.

루프를 돌 때 데이터의 값뿐만 아니라 숫자 인덱스를 알고 있으면 유용할 때가 있다. 다음은 한 예이다.

```

i = 0
for x in s:
    문장들
    i += 1

```

파이썬의 내부 함수인 enumerate()를 사용하면 앞의 코드를 다음과 같이 간단하게 표현할 수 있다.

```

for i, x in enumerate(s):
    문장들

```

enumerate(s)는 간단히 (0, s[0]), (1, s[1]), (2, s[2]) 등과 같은 일련의 튜플들을 반환하는 반복자를 생성한다.

루프를 돌 때 2개 이상의 순서열에 대해 동시에 반복을 수행해야 하는 경우가 종종 있다. 예를 들어, 각 반복마다 여러 순서열로부터 항목을 하나씩 가져오는 코드는 다음과 같이 작성할 수 있다.

```
# s와 t는 순서열
i = 0
while i < len(s) and i < len(t):
    x = s[i]      # s로부터 항목을 가져온다.
    y = t[i]      # t로부터 항목을 가져온다.
    문장들
    i += 1
```

위 코드는 `zip()` 함수를 사용함으로써 아래처럼 간단히 표현할 수 있다.

```
# s와 t는 순서열
for x,y in zip(s,t):
    문장들
```

`zip(s, t)` 함수는 순서열 `s`와 순서열 `t`를 `(s[0], t[0]), (s[1], t[1]), (s[2], t[2])`와 같이 하나의 순서열로 합친다. 이때, 하나로 합친 순서열의 길이는 `s`와 `t`가 길이가 서로 다르면 `s`와 `t` 중 길이가 작은 값으로 설정된다. 파이썬 2에서 `zip()`을 사용할 때 주의할 점은 이 함수가 `s`와 `t`를 한 번에 다 소진하면서 큰 튜플 리스트를 생성한다는 점이다. 여러분은 아마 많은 양의 데이터를 담는 생성기와 순서열에 대해서 `zip()` 함수가 이러한 식으로 작동하는 것을 원하지 않을 것이다. `itertools.izip()` 함수는 `zip()`과 동일한 효과를 내지만 커다란 튜플 리스트를 생성하는 대신에 값을 한 번에 하나씩 생성한다. 파이썬 3의 `zip()` 함수도 이러한 식으로 값을 생성한다.

루프를 빠져나오려면 `break`문을 사용하면 된다. 예를 들어, 다음 코드는 빈 줄을 만날 때까지 파일에서 텍스트를 한 줄씩 읽는다.

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        break      # 빈 줄이다. 읽기를 중단한다.
    # 이번 줄을 처리한다.
    ...
```

루프의 다음 반복으로 건너뛰려면(현재 반복의 나머지 부분을 생략) `continue`문을 사용하면 된다. `continue`문은 덜 사용되는 경향이 있지만, 검사를 거꾸로 수행하거나 코드를 한 수준 더 들여쓰기 했을 때 코드가 너무 길어 중첩되거나 필요 없이 복잡해지는 경우 유용하게 사용할 수 있다. 예를 들어, 다음은 파일에서 빈 줄을 모두 건너뛰는 루프를 보여준다.

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        continue      # 빈 줄을 건너뛴다.
    # 이번 줄을 처리한다.
    ...
```

break문과 continue문은 실행되고 있는 루프 중 가장 안쪽 루프에만 적용된다. 깊게 중첩된 루프 구조에서 벗어나야 할 경우에는 예외를 사용해야 한다. 파이썬에서는 goto문을 지원하지 않는다.

다음 예와 같이 반복문에 else문을 추가할 수도 있다.

```
# for-else
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        break
    # 이번 줄을 처리한다.
    ...
else:
    raise RuntimeError("Missing section separator")
```

루프에서 else절은 루프가 종료되었을 때만 실행된다. 루프는(루프가 아예 실행이 안 된 경우) 바로 종료되거나 마지막 반복 후에 종료된다. 루프가 break문에 의해 일찍 중단되면 else절은 수행되지 않는다.

루프에서 else절은 데이터에 대해 반복을 수행하되, 루프가 이른 시점에 종료될 경우 플래그나 조건을 설정하거나 검사할 필요가 있을 때 주로 사용된다. 예를 들어 else문을 사용하지 않으면, 앞의 예제의 코드는 플래그 변수를 사용해서 다음과 같이 다시 작성할 수 있다.

```
found_separator = False
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        found_separator = True
        break
    # 이번 줄을 처리한다.
    ...
if not found_separator:
    raise RuntimeError("Missing section separator")
```

예외

예외(exception)는 오류를 나타나며 프로그램이 일반적인 제어 흐름에서 벗어나게 만든다. raise문으로 예외를 발생시킬 수 있다. raise문의 일반적인 형식은 raise Exception([value])이며, 여기서 Exception은 예외 타입을 나타내고 value는 추가 값으로서 예외와 관련된 구체적인 설명을 담는다. 다음은 한 예이다.

```
raise RuntimeError("Unrecoverable Error");
```

`raise`문을 홀로 사용하면 최종적으로 생성된 예외를 다시 발생시킨다(단, 이전에 발생된 예외를 처리하고 있는 도중에만 가능하다).

예외를 잡으려면 다음과 같이 `try`와 `except`를 사용하면 된다.

```
try:
    f = open('foo')
except IOError as e:
    문장들
```

예외가 발생하면 인터프리터는 `try` 블록 안의 문장들을 수행하는 것을 중단하고, 발생한 예외와 부합하는 `except`절을 찾는다. 해당하는 `except`절을 찾았다면 제어 흐름이 `except`절의 첫 문장으로 넘어간다. `except`절을 실행한 후 제어 흐름은 `try-except` 블록 다음에 나타나는 첫 문장으로 넘어간다. 만약 해당하는 `except`절을 찾지 못했다면 예외는 `try`문을 담고 있는 코드 블록 쪽으로 전파된다. 이 코드 블록 자체는 다시 `try-except` 블록으로 감싸져 있을 수도 있다. 예외가 잡히지 않은 채로 프로그램의 최상위 수준까지 도달하면 인터프리터는 에러 메시지를 출력하면서 실행을 중단한다. 13장에서 설명하겠지만, 잡히지 않은 예외는 사용자 정의 함수인 `sys.excepthook()`으로 전달될 수도 있다.

`except`문에서 선택적으로 지정할 수 있는 `as var` 변경자는 예외가 발생하였을 때 `raise`문에 의해서 발생된 예외 인스턴스가 저장될 변수의 이름을 지정한다. 예외 처리기는 이 변수를 통해 예외가 발생한 원인에 관한 정보를 얻을 수 있다. 예를 들어, `isinstance()`를 사용해서 예외의 타입을 검사해볼 수 있다. `except`문의 문법과 관련해서 주의할 점이 있다. 파이썬의 이전 버전에서 `except`문을 작성할 때는 `except ExcType, var`와 같이 예외 타입과 변수를 콤마로 구분했다. 파이썬 2.6에서도 이 문법을 여전히 사용할 수 있지만, 사용을 권장하지 않는다. 앞으로는 `as var` 문법을 사용하도록 한다. 파이썬 3에서는 오직 이 문법만을 지원하기 때문이다.

다중 예외 처리 블록은 다음의 예와 같이 여러 개의 `except`절을 사용해 작성한다.

```
try:
    무언가를 수행한다.
except IOError as e:
    # I/O 에러를 처리한다.
    ...
except TypeError as e:
    # Type 에러를 처리한다.
    ...
except NameError as e:
    # Name 에러를 처리한다.
    ...
```

하나의 예외 처리기가 다음과 같이 여러 가지 예외 타입을 처리할 수 있다.

```
try:
    무언가를 수행한다.
except (IOError, TypeError, NameError) as e:
    # I/O, Type, Name 에러를 처리한다.
    ...
    ...
```

다음과 같이 pass문을 사용해서 예외를 무시할 수 있다.

```
try:
    무언가를 수행한다.
except IOError:
    pass    # 아무것도 하지 않는다.
```

프로그램 종료와 관련된 것을 제외한 모든 예외를 잡고자 한다면 다음과 같이 Exception을 사용하면 된다.

```
try:
    do something
except Exception as e:
    error_log.write('An error occurred : %s\n' % e)
```

모든 예외를 잡는 경우에는 사용자에게 에러에 관한 정확한 정보를 알려주도록 신경 써야 한다. 예를 들어, 앞의 코드에서는 에러 메시지와 관련 예외에 관한 정보를 로그에 기록하였다. 예외에 관한 정보를 기록해 놓지 않으면 예상치 못한 원인으로 인해 에러가 발생된 코드를 디버깅하기가 매우 힘들어진다.

다음과 같이 예외 타입을 지정하지 않은 except를 사용해서 모든 예외를 잡을 수도 있다.

```
try:
    do something
except:
    error_log.write('An error occurred\n')
```

앞의 형식의 except는 올바르게 사용하기가 보기보다 어렵기 때문에 가급적 사용을 피해야 한다. 예를 들어, 앞의 코드는 아마 여러분이 잡기를 원하지 않았을 키보드 인터럽트나 프로그램 종료 요청까지 잡아버릴지도 모른다.

try문 또한 else절을 지원한다. else절은 반드시 마지막 except절 바로 다음에 나와야 한다. else문은 try 블록에서 예외가 발생하지 않을 경우 실행된다. 다음은 한 예이다.

```

try:
    f = open('foo', 'r')
except IOError as e:
    error_log.write('Unable to open foo : %s\n' % e)
else:
    data = f.read()
    f.close()

```

finally문은 try 블록 안에 있는 코드를 정리하는 작업을 수행한다. 다음은 한 예이다.

```

f = open('foo','r')
try:
    # 무언가를 수행한다.
    ...
finally:
    f.close()
    # 결과에 상관없이 파일을 닫는다.

```

finally절은 예외를 처리하기 위해 사용되지 않는다. 대신에, 예외의 발생 유무와 상관없이 반드시 실행해야 할 코드를 담는 데 사용된다. 예외가 발생하지 않으면, finally절에 있는 코드는 try 블록에 존재하는 코드를 다 실행한 후에 실행된다. 예외가 발생하면 제어 흐름은 먼저 finally절의 첫 문장으로 넘어간다. finally절의 코드가 수행되고 나면, 해당 예외는 다시 발생되어 다른 예외 처리기에 의해 처리된다.

내장 예외

파이썬에는 표 5.1에 나와 있는 내장 예외들이 있다.

표 5.1 내장 예외

예외	설명
BaseException	모든 예외의 조상
GeneratorExit	생성기의 .close() 메서드에 의해 발생
KeyboardInterrupt	인터럽트 키(보통 Ctrl+C)에 의해 발생
SystemExit	프로그램 종료
Exception	존재하지 않는 모든 예외를 위한 기반 클래스
StopIteration	반복을 중단하기 위해 발생됨
StandardError	모든 내장 예외에 대한 기반 클래스(파이썬 2에서만). 파이썬 3에서는 모든 예외가 Exception 아래에 묶여 있음
ArithmeticError	산술 예외와 관련된 기반 클래스

FloatingPointError	부동 소수점 연산의 실패
OverflowError	정수 값이 너무 큼
ZeroDivisionError	0으로 나누기 또는 나머지 연산
AssertionError	assert문에 의해 발생됨
AttributeError	속성 이름이 유효하지 않을 경우 발생됨
EnvironmentError	파이썬의 외부에서 발생된 에러
IOError	I/O 또는 파일 관련 에러
OSError	운영체제 에러
WindowsError	윈도에 국한된 에러
EOFError	파일의 끝에 도착하였을 때 발생됨
ImportError	import문의 실패
LookupError	색인과 키 에러
IndexError	순서열 색인이 범위를 벗어남
KeyError	사전의 키가 존재하지 않음
MemoryError	메모리 부족
NameError	지역 또는 전역 이름 찾기 실패
UnboundLocalError	묶이지 않은 지역 변수
ReferenceError	참조 대상이 쓰레기 수집된 후에 약한 참조가 이루어짐
RuntimeError	다목적용 범용 에러
NotImplementedError	구현 안 된 기능
SyntaxError	구문 에러
IndentationError	들여쓰기 에러
TabError	일관성 없는 탭 사용(-tt 옵션으로 실행한 경우 발생)
SystemError	인터프리터에서 치명적이지 않은 시스템 에러
TypeError	연산에 적절하지 않는 타입을 넘김
ValueError	유효하지 않는 타입
UnicodeError	유니코드 에러
UnicodeDecodeError	유니코드 디코딩 에러
UnicodeEncodeError	유니코드 인코딩 에러
UnicodeTranslateError	유니코드 변환 에러

표에서 볼 수 있듯이 예외는 계층적으로 조직화되어 있다. 특정한 그룹에 있는 예외들은 except절에 그룹 이름을 지정해주어 잡을 수 있다. 다음은 한 예이다.

```

try:
    문장들
except LookupError:      # IndexError 또는 KeyError를 잡음
    문장들
또는

try:
    문장들
except Exception:        # 프로그램과 관련된 모든 에러를 잡음
    문장들

```

예외 계층의 최상층에서 예외들은 프로그램의 종료와 관련된 예외인지 아닌지에 따라 묶여 있다. 예를 들어, SystemExit와 KeyboardInterrupt 예외는 Exception 아래에 있지 않다. 그 이유는 프로그램과 관련된 모든 에러를 잡고자 할 때 보통은 갑작스런 프로그램 종료까지 처리하려는 것이 아니기 때문이다.

새로운 예외 정의

모든 내장 예외는 클래스로서 정의된다. 새로운 예외를 생성하기 위해서는 다음과 같이 Exception으로부터 상속받아 새로운 클래스를 정의하면 된다.

```

class NetworkError (Exception):
    pass

새로 정의된 예외를 사용하기 위해서는 아래와 같이 raise문을 사용하면 된다.

raise NetworkError("Cannot find host.")

```

예외를 발생시킬 때 raise문에 제공된 추가 값은 예외 클래스 생성자의 인수로서 사용된다. 이 인수는 대부분의 경우 간단히 에러 메시지를 담은 문자열이다. 사용자 정의 예외는 다음과 같이 하나 이상의 예외 값을 받도록 작성할 수 있다.

```

class DeviceError(Exception):
    def __init__(self,errno,msg):
        self.args = (errno, msg)
        self(errno = errno
        self.errmsg = msg

# 예외를 발생시킨다. (다수의 인수)
raise DeviceError(1, 'Not Responding')

```

`__init__()__`를 재정의하는 새로운 예외를 생성할 때 앞의 예에서 보듯이 인수들을 담는 튜플을 `self.args` 속성에 저장하는 일이 중요하다. 이 속성은 예외 역추적 메시지를 출력할 때 사용된다. 이 속성을 그대로 두면 사용자는 에러가 발생했을 때 예외와 관련된 유용한 정보를 볼 수 있게 된다.

예외는 상속을 통해 계층적으로 조직화할 수 있다. 예를 들어, 앞에서 정의한 NetworkError 예외는 더 구체적인 다양한 에러들을 위한 기반 클래스로 쓰일 수 있다. 다음 예를 살펴보자.

```
class HostnameError(NetworkError): pass
class TimeoutError(NetworkError): pass

def error1( ):
    raise HostnameError("Unknown host")

def error2( ):
    raise TimeoutError("Timed out")

try:
    error1( )
except NetworkError as e:
    if type(e) is HostnameError:
        # 이 에러에 대해서 특별한 작업을 수행한다.
    ...

```

앞의 코드에서 except NetworkError문은 NetworkError로부터 파생된 모든 예외를 잡는다. 발생된 예외의 구체적인 타입을 알아내려면 type()으로 예외 값의 타입을 검사하면 된다. 또는 sys.exc_info() 함수를 사용하여 가장 마지막에 발생된 예외와 관련된 정보를 얻을 수도 있다.

컨텍스트 관리자 with문

예외가 발생한 경우 파일, 락, 연결 등의 시스템 자원을 적절히 관리하는 일은 쉽지 않다. 예를 들어, 예외 발생 때문에 락과 같은 중요한 자원을 해제하는 부분이 실행되지 않는 경우가 있을 수 있다.

with문은 컨텍스트 관리자의 역할을 하는 객체에 의해 제어되는 런타임 컨텍스트 안에서 일련의 문장들을 실행한다. 다음 예를 살펴보자.

```
with open("debuglog", "a") as f:
    f.write("Debugging\n")
    문장들
    f.write("Done\n")

import threading
lock = threading.Lock( )
with lock:
    # 임계 구역(critical section)
    문장들
    # 임계 구역 종료
```

첫 번째 예에서 with문은 제어 흐름이 블록을 벗어날 때 열린 파일을 자동으로 닫는다. 두 번째 예에서 with문은 이어서 나오는 블록에 진입할 때와 빠져나올 때 자동적으로 락을 획득하고 품다.

with obj문은 제어 흐름이 이어서 나오는 블록에 진입하고 빠져나올 때 일어나는 일을 객체 obj에서 관리할 수 있게 해준다. with obj문이 실행되면 새로운 컨텍스트에 진입한다는 신호로서 메서드 obj.__enter__()가 호출된다. 제어 흐름이 컨텍스트를 벗어날 때는 메서드 obj.__exit__(type, value, traceback)가 호출된다. 발생된 예외가 없는 경우 __exit__()의 3개의 인수는 모두 None으로 설정된다. 그렇지 않을 경우 이 3개의 인수는 제어 흐름을 컨텍스트에서 벗어나게 한 예외의 타입, 예외 값, 역추적 정보를 담게 된다. __exit__() 메서드는 발생된 예외가 처리되었는지의 여부에 따라 True 또는 False를 반환한다(False가 반환되면 발생된 예외가 컨텍스트 밖으로 전달된다).

with obj문은 추가적으로 var 지정자를 받아들인다. var 지정자가 있으면 obj.__enter__()에서 반환된 값이 var에 저장된다. 항상 obj가 var에 할당되는 것은 아니다.

with문은 컨텍스트 관리 프로토콜(__enter__()과 __exit__() 메서드)을 지원하는 객체에 대해서만 사용할 수 있다. 사용자 정의 클래스에서는 이 메서스들을 정의하여 자신만의 컨텍스트 관리를 수행할 수 있다. 다음은 간단한 예를 보여준다.

```
Class ListTransaction(object):
    def __init__(self, thelist):
        self.thelist = thelist
    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy
    def __exit__(self, exctype, value, tb):
        if exctype is None:
            self.thelist[:] = self.workingcopy
        return False
```

이 클래스는 리스트에 일련의 변경을 가할 수 있는 기능을 제공한다. 하지만, 변경된 내용은 예외가 발생되지 않는 경우에만 리스트에 적용된다. 예외가 발생되면 원래의 리스트는 변경되지 않는다. 다음은 이 클래스를 사용하는 예이다.

```
Items = [1,2,3]
with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items)    # [1,2,3,4,5]를 생성
```

```

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("We're hosed!")
except RuntimeError:
    pass
print(items)    # [1,2,3,4,5]를 생성

```

contextlib 모듈을 사용하면 생성기 함수를 래퍼로 감쌈으로서 컨텍스트 관리기 를 더욱 쉽게 구현할 수 있다. 다음은 한 예이다.

```

from contextlib import contextmanager
@contextmanager
def ListTransaction(thelist):
    workingcopy = list(thelist)
    yield workingcopy
    # 예외가 발생하지 않는 경우에만 원래의 리스트를 변경한다.
    thelist[:] = workingcopy

```

앞의 예에서 yield로 전달된 값은 `__enter__()`의 반환값으로 사용된다. `__exit__()` 메서드가 호출되면 yield 다음부터 실행이 재개된다. 컨텍스트 내부에서 예외가 발생되면 생성기 함수로 전달된다. 원할 경우 예외를 잡을 수도 있지만, 앞의 예에서는 다른 곳에서 예외가 처리되도록 생성기 밖으로 예외가 전달되도록 놔두었다.

assert와 `__debug__`

assert문으로 프로그램에 디버깅 코드를 넣을 수 있다. assert문의 일반적인 형식은 다음과 같다.

```
assert test [, msg]
```

여기서 test는 True와 False로 평가되는 표현식이다. 만약 test가 False로 평가되면 assert문에 지정한 메시지인 msg와 함께 AssertionError 예외가 발생한다. 다음은 한 예이다.

```

def write_date(file,data):
    assert file, "write_data: file not defined!"
    ...

```

프로그램이 올바르게 작동하기 위해서 반드시 수행되어야 하는 코드에는 assert 문을 사용하지 않도록 한다. 왜냐하면 파이썬이 최적화 모드(인터프리터에 -O 옵션)로 실행될 때 assert문이 실행되지 않기 때문이다. 특히 사용자의 입력을 확인하는 데 assert문을 사용하면 안 된다. assert문은 항상 참이어야 하는 것들을 검사하

는 데 사용한다. 참이 아닌 것이 있으면 사용자의 오류가 아니라 프로그램의 버그 때문인 것이다.

예를 들어, 앞의 예에서 `write_data()`가 최종 사용자에 의해서 사용되는 함수라면 `assert`문은 `if`문과 예리 처리 코드로 대체되어야 한다.

`assert`와 더불어 파이썬에서는 읽기 전용 내장 변수인 `__debug__`가 제공된다. 이 변수는 인터프리터가 최적 모드(인터프리터에 `-O 옵션`)로 수행되지 않는 한 `True`로 설정된다. 필요에 따라 프로그램에서 이 변수를 확인할 수 있고, 이 변수가 설정된 경우 추가적인 예리 검사 과정을 수행할 수도 있다. 인터프리터에서 `__debug__` 변수가 구현되는 방식은 최적화되어 있어서 추가적인 `if`문의 제어 흐름 로직은 코드에 실제로 포함되지 않는다. 파이썬이 일반 모드로 실행되는 경우 `if __debug__` 문 아래에 있는 문장들은 `if`문 없이 프로그램 코드에 직접 나타난다. 최적화 모드에서 `if __debug__` 문과 관련된 모든 문장들이 프로그램에서 완전히 제거된다.

`assert`문과 `__debug__`를 사용하면 프로그램을 두 가지 모드를 통해 효율적으로 개발할 수 있다. 예를 들어, 디버그 모드에서는 코드가 올바르게 작동하는지를 검증하기 위한 단언 및 버그 검사 코드를 자유롭게 사용할 수 있다. 최적화 모드에서는 이러한 모든 추가적인 검사들이 자동으로 제거되기 때문에 성능에 영향이 없다.

6장

P y t h o n E s s e n t i a l R e f e r e n c e

함수와 함수형 프로그래밍

많은 프로그램이 모듈화와 유지 보수의 용이성을 위해 여러 함수로 구성된다.

파이썬에서는 함수를 쉽게 정의할 수 있을 뿐만 아니라 함수형 프로그래밍 언어의 놀라울 정도로 많은 기능을 사용할 수 있다. 이 장에서는 함수, 유효 범위 규칙, 클로저, 장식자, 생성기, 코루틴과 기타 함수형 프로그래밍 언어의 특징에 대해서 다룬다. 또한, 선언형 스타일 프로그래밍을 하거나 데이터 처리를 수행할 때 강력한 힘을 발휘하는 도구인 리스트 내포와 생성기 표현식에 대해서도 다룬다.

함수

함수는 def문으로 정의한다.

```
def add(x,y) :
    return x + Y
```

함수의 몸체는 함수가 호출될 때 실행되는 문장들로 이루어져 있다. 함수는 `a = add(3, 4)` 같이 함수의 이름을 먼저 적고 바로 이어서 함수 인수들의 튜플을 쓰는 방식으로 호출한다. 여기서 인수의 순서와 개수는 함수에서 정의한 것과 반드시 일치하여야 한다. 일치하지 않을 경우, `TypeError` 예외가 발생한다.

다음의 예와 같이 함수 정의에서 값을 할당하여 함수 매개변수에 기본 인수를 지정할 수 있다.

```
def split(line,delimiter=',')
문장들
```

함수에 기본 값을 가지는 매개변수가 있으면 그 매개변수와 뒤에 따라 나오는 모든 매개변수는 생략할 수 있다. 함수 정의에서 생략 가능한 모든 매개변수에는 값을 할당해야 하고 그렇지 않을 경우 SyntaxError 예외가 발생한다.

기본 매개변수 값은 항상 함수를 정의할 때 지정한 객체로 설정된다. 다음 예를 보자.

```
a = 10
def foo(x=a):
    return x

a = 5          # 'a'를 재할당한다.
foo( )         # 10을 반환한다(기본 값은 바뀌지 않는다).
```

변경 가능한 객체를 기본 값으로 지정할 경우 의도하지 않은 결과를 얻을 수 있다.

```
def foo(x, items=[]):
    items.append(x)
    return items

foo(1)      # [1]을 반환한다.
foo(2)      # [1,2]를 반환한다.
foo(3)      # [1,2,3]을 반환한다.
```

이 예에서 기본 인수가 이전 호출 때의 변경 사항을 담고 있다. 이를 피하려면 다음과 같이 기본 값으로 None을 지정하고 추가적인 검사를 해주는 것이 좋다.

```
def foo(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

마지막 매개변수 이름 앞에 별표(*)를 추가하면 함수는 여러 개의 매개변수를 받을 수 있다.

```
def fprintf(file, fmt, *args) :
    file.write(fmt % args)

# fprintf 사용. args는 (42, "hello world", 3.45)가 된다.
fprintf(out, "%d %s %f", 42, "hello world", 3.45)
```

이렇게 할 경우 남아 있는 모든 인수가 튜플로서 args 변수에 저장된다. 튜플 args를 매개변수인 것처럼 함수에 전달하려면 다음과 같이 함수를 호출할 때 *args 문법을 사용하면 된다.

```
def printf(fmt, *args) :
    # 다른 함수를 호출하고 args를 넘겨준다.
    fprintf(sys.stdout, fmt, *args)
```

함수 인수를 전달할 때 각 매개변수의 이름과 값을 직접 지정할 수도 있다. 이러한 인수를 키워드 인수(keyword argument)라고 한다. 다음 예를 살펴보자.

```
def foo(w, x, y, z):
    문장들

    # 키워드 인수 호출
    foo(x=3, y=22, w='hello', z=[1,2])
```

키워드 인수를 사용할 때는 매개변수의 순서가 중요하지 않다. 하지만, 기본 값이 지정된 것을 제외하고는 함수의 모든 매개변수를 직접 지정해주어야 한다. 필요한 매개변수를 누락하거나 키워드의 이름이 함수에서 정의된 매개변수의 이름 중 어느 것과도 일치하지 않으면 `TypeError` 예외가 발생한다. 파이썬에서는 모든 함수를 키워드 호출 방법으로 호출할 수 있기 때문에 함수를 정의할 때는 인수 이름을 이해하기 쉽고 기억하기 쉽도록 정하는 것이 좋다.

위치 인수(positional argument)와 키워드 인수는 함수를 호출할 때 같이 나타날 수 있다. 이런 경우 모든 위치 인수를 먼저 지정해야 하고 모든 생략 가능한 인수의 값을 지정해야 하며 어떤 인수 값을 한 번 이상 설정하면 안 된다. 다음 예를 보자.

```
foo('hello', 3, z=[1,2], y=22)
foo(3, 22, w='hello', z=[1,2]) # TypeError. w에 여러 값을 지정하였음.
```

함수의 마지막 인수의 이름이 `**`로 시작할 경우 (어떤 매개변수의 이름과도 일치하지 않는) 모든 추가 키워드 인수들이 사전으로 구성되어 함수에 전달된다. 이 방법은 구성 옵션(configuration option)이 몇 개가 될지 알 수 없는 상황에서 많은 수의 옵션을 받아들여야 하지만 매개변수들을 직접 나열하기는 어려울 때 유용하게 쓰인다. 다음 예를 살펴보자.

```
def make_table(data, **parms):
    # parms(dict 타입)로부터 구성 옵션을 얻어온다.
    fgcolor = parms.pop("fgcolor", "black")
    bgcolor = parms.pop("bgcolor", "white")
    width = parms.pop("width", None)

    ...
    # 옵션이 더는 없다.
    if parms:
        raise TypeError("Unsupported configuration options %s",
                        % list(parms))
```

```
make_table(items, fgcolor="black", bgcolor="white", border=1,
           borderstyle="grooved", cellpadding=10, width=400)
```

** 매개변수가 가장 나중에 나오는 경우에 한 해서 추가 키워드 인수들이 가변 길이 인수 목록과 함께 나타날 수 있다.

```
# 다수의 위치 또는 키워드 인수를 받음
def spam(*args, **kwargs):
    # args는 위치 인수들의 튜플
    # kwargs는 키워드 인수들의 사전
    ...
```

키워드 인수는 **kwargs 문법을 사용하여 다른 함수로 전달할 수 있다.

```
def callfunc(*args, **kwargs):
    func(*args, **kwargs)
```

*args와 **kwargs는 다른 함수에 대한 래퍼(wrapper)나 대리자(proxy)를 작성하는 데 주로 사용된다. 예를 들어, callfunc()는 임의의 인수 조합을 받아서 단순히 func()으로 전달하기만 한다.

매개변수 전달과 반환 값

함수를 호출할 때 함수의 매개변수는 단순히 전달된 입력 객체를 참조하는 이름일 뿐이다. 매개변수 전달이 내부적으로 작동하는 방식은 다른 프로그래밍 언어와 관련해서 들었을지도 모르는 “값에 의한 전달(pass by value)” 또는 “참조에 의한 전달(pass by reference)” 같은 방식 중에 어느 하나의 스타일과 정확히 일치하지는 않는다. 예를 들어, 변경 불가능한 값을 전달할 때는 실질적으로 인수가 값으로 전달되는 것처럼 보인다. 하지만, 변경 가능한 객체(리스트나 사전 같은)가 함수에 전달되어 그 값이 변경되면 원래의 객체 값에도 반영된다. 다음 예를 보자.

```
a = [1, 2, 3, 4, 5]
def square(items):
    for i, x in enumerate(items):
        items[i] = x * x # 항목을 제자리에서 수정한다.

square(a)      # a를 [1, 4, 9, 16, 25]로 변경한다.
```

입력으로 넘어온 값을 변경하거나 알게 모르게 프로그램의 다른 부분을 변경하는 함수는 부작용(side effect)을 가진다고 한다. 일반적으로 프로그램이 커지거나 복잡해질수록 까다로운 문제를 발생시키는 원인이 될 수 있기 때문에 이러한 함수

를 작성하지 않는 것이 좋다(예를 들어, 함수 호출 모습만 보고서는 함수가 부작용을 가지는지 알기 어렵다). 보통 부작용이 있는 경우 락이 필요하기 때문에 스레드나 병행 프로그래밍이 필요한 곳에서 이러한 함수를 사용하는 일은 까다롭다.

함수에서 return문은 값을 반환한다. 반환할 값을 지정하지 않거나 return문을 생략하면 None 객체가 반환된다. 여러 값을 반환하려면 다음과 같이 튜플에 넣어서 반환하면 된다.

```
def factor(a):
    d = 2
    while ( d <= (a / 2)):
        if ((a / d) * d == a ):
            return ((a / d), d)
        d = d + 1
    return (a, 1)
```

튜플로 반환되는 여러 반환 값은 다음과 같이 개별 변수에 저장할 수 있다.

```
x, y = factor(1243)      # x와 y에 반환 값이 저장된다.
```

또는

```
(x,y) = factor(1243)      # 같은 일을 하는 다른 버전.
```

유효 범위 규칙

함수가 실행될 때마다 새로운 지역 네임스페이스가 생성된다. 이 네임스페이스는 매개변수의 이름과 함수 몸체 안에서 할당된 변수의 이름을 담은 내부적인 실행 환경을 나타낸다. 변수 이름을 해석할 때 인터프리터는 먼저 지역 네임스페이스를 검색한다. 일치하는 것이 없으면 다음으로 전역 네임스페이스를 검색한다. 함수의 전역 네임스페이스는 항상 함수가 정의된 모듈이다. 인터프리터가 전역 네임스페이스에서도 이름을 찾지 못하면 마지막으로 내장 네임스페이스를 확인한다. 이것마저 실패하면 NameError 예외가 발생한다

함수 안에서 전역 변수를 다룰 때 네임스페이스와 관련해서 독특한 점이 하나 있다. 예를 들어, 다음의 코드를 살펴보자.

```
a = 42
def foo( ):
    a = 13
foo( )
# a는 여전히 42다.
```

이 코드를 실행하면 foo함수 안에서 a를 변경하는 것처럼 보이지만 a는 여전히 42가 된다. 함수 안에서 변수에 값을 할당하면 변수는 항상 함수의 지역 네임스페이스에 뮤인다. 그 결과 함수 몸체에 있는 변수 a는 밖에 있는 변수 a가 아니라 13이라는 값을 담은 완전히 새로운 객체를 가리키게 된다. 이러한 작동 방식을 변경하려면 global문을 사용하면 된다. global은 어떤 이름이 전역 네임스페이스에 속한다는 것을 선언한다. global문은 전역 변수를 수정하고자 할 때만 필요하다. global문은 함수 몸체 안 어느 곳에나 나타날 수 있고 반복적으로 사용해도 된다. 다음 예를 살펴보자.

```
a = 42
b = 37
def foo( ):
    global a      # 'a'는 전역 네임스페이스의 'a'를 의미한다.
    a = 13
    b = 0
foo()
# a는 13이되고 b는 여전히 37이다.
```

파이썬은 중첩 함수 선언을 지원한다. 다음은 그 예이다.

```
def countdown(start):
    n = start
    def display( ):          # 중첩 함수 선언
        print('T-minus %d' % n)
    while n > 0:
        display( )
        n -= 1
```

중첩 함수 안의 변수의 이름은 어휘 유효 범위(lexical scoping)에 따라 찾아진다. 즉, 먼저 지역 유효 범위에서 찾고 다음으로 가장 안쪽 유효 범위부터 가장 바깥쪽 유효 범위까지 모든 둘러싸는 바깥 함수의 유효 범위에서 찾는다. 아무것도 찾지 못하면 이전처럼 전역 네임스페이스와 내장 네임스페이스를 검사한다. 둘러싸는 유효 범위에 있는 이름에 접근할 수는 있지만 파이썬 2에서는 가장 안쪽의 유효 범위(지역 변수)와 전역 네임스페이스(global을 사용해서)에 있는 변수에만 값을 재할당할 수 있다. 따라서, 바깥쪽 함수에서 정의된 지역 변수의 값을 안쪽 함수에서 다시 할당할 수 없다. 예를 들어, 다음 코드는 제대로 작동하지 않는다.

```
def countdown(start):
    n = start
    def display( ):
        print('T-minus %d' % n)
    def decrement( ):
        n -= 1      # 파이썬 2에서 작동하지 않는다.
    while n > 0:
        display( )
        decrement( )
```

파이썬 2에서 이 문제를 해결하려면 변경하고자 하는 값을 리스트나 사전에 넣으면 된다. 파이썬 3에서는 다음과 같이 n을 nonlocal로 선언하면 된다.

```
def countdown(start):
    n = start
    def display( ):
        print(T-minus %d' % n)
    def decrement( ):
        nonlocal n      # 바깥쪽 n에 묶인다(파이썬 3에서만).
        n -= 1
    while n > 0:
        display( )
        decrement( )
```

nonlocal 선언문은 어떤 이름을 현재 호출 스택보다 아래에 있는 임의의 함수 안에서 정의된 지역 변수에 묶지는 않는다(즉, 동적 유효 범위(dynamic scope)를 사용하지 않는다). 따라서 펄(Perl)을 사용한 적이 있다면 파이썬의 nonlocal이 펄에서 local 변수와는 다르다는 것에 유의해야 한다.

지역 변수에 값을 할당하기 전에 사용하면 UnboundLocalError 예외가 발생한다. 다음 예는 이러한 일이 일어날 수 있는 한 시나리오를 보여준다.

```
i = 0
def foo( ):
    i = i + 1      # UnboundLocalError 예외가 발생한다.
    print(i)
```

이 함수에서 변수 i는 지역 변수로 정의되었다(왜냐하면 함수 안에서 값이 할당되며 global문이 없기 때문이다). 하지만 대입문 i = i + 1은 먼저 i에 값이 할당되기 전에 i의 값을 읽으려고 한다. 이 예에서 전역 변수 i가 있어도 이 값이 사용되지 않았다. 변수는 함수를 정의하는 순간에 지역 또는 전역인지 여부가 결정되지 않은 중간에 유효 범위가 갑자기 변경되는 일은 없다. 예를 들어, 앞의 코드에서 표현식 i + 1의 i는 전역 변수를 가리키는 반면에 print(i)의 i는 이전 문장에서 생성된 지역 변수 i를 가리키는 일은 일어나지 않는다.

객체와 클로저로서 함수

파이썬에서 함수는 1급 객체이다. 이 말은 함수가 다른 함수의 인수로 전달될 수 있고 자료 구조 안에 들어갈 수도 있으며 함수의 결과로서 반환될 수도 있음을 뜻한다. 다음은 다른 함수를 받아서 호출하는 예를 보여준다.

```
# foo.py
def callf(func):
    return func( )
```

다음은 앞에 나온 함수를 사용하는 예이다.

```
>>> import foo
>>> def helloworld( ):
...     return 'Hello World'
...
>>> foo.callf(helloworld)      # 함수를 인수로서 전달한다.
'Hello World'
>>>
```

함수가 데이터로 취급될 때는 알게 모르게 함수가 정의된 곳의 주변 환경 정보가 함께 저장된다. 이 사실은 함수 안에 있는 자유 변수(free variable)가 어떤 값을 가지는지에 영향을 준다. 한 예로서 이제 변수 정의를 포함하고 있는 다음에 나와 있는 foo.py의 수정된 버전을 보자.

```
# foo.py
x = 42
def callf(func):
    return func( )
```

이제 이 예가 어떻게 작동하는지 보자.

```
>>> import foo
>>> x = 37
>>> def helloworld( ):
...     return "Hello World. x is %d" % x
...
>>> foo.callf(helloworld)          # 함수를 인수로서 전달한다.
'Hello World. x is 37'
>>>
```

앞의 예에서 함수 helloworld()가 정의되었던 환경에서 정의된 x의 값을 어떻게 사용하는지 살펴보자. 비록 x가 foo.py에서도 정의되어 있고 실제로 helloworld()가 호출된 곳도 foo.py이지만 출력된 값은 이 x 값이 아니었다.

함수를 구성하는 문장과 실행 환경을 함께 묶은 것을 클로저(closure)라고 부른다. 앞에 나온 예제의 결과는 모든 함수는 자신이 정의된 전역 네임스페이스를 가리키는 `__globals__`라는 속성을 가진다는 사실로 설명할 수 있다. 전역 네임스페이스는 항상 함수가 정의된 모듈이다. 앞의 예에서 `__globals__`를 검사해보면 다음과 같다.

```
>>> helloworld.__globals__
{'__builtins__': <module '__builtin__' (built-in)>,
```

```
'helloworld': <function helloworld at 0x7bb30>,
'__x': 37, '__name__': '__main__', '__doc__': None
'foo': <module 'foo' from 'foo.py'>
>>>
```

중첩 함수가 사용될 때 클로저는 내부 함수 실행을 위한 전체 환경을 담는다. 다음 예를 살펴보자.

```
import foo
def bar( ):
    x = 13
    def helloworld( ):
        return "Hello World. x is %d" % x
    foo.callf(helloworld)      # 'Hello World, x is 13'을 반환한다.
```

클로저와 중첩 함수는 게으른 평가(lazy evaluation) 또는 지연 평가(delayed evaluation)라는 개념에 기초하여 코드를 작성하고자 할 때 특히 유용하게 쓰인다. 또 다른 예를 살펴보자.

```
from urllib import urlopen
# from urllib.request import urlopen (파이썬 3)
def page(url):
    def get( ):
        return urlopen(url).read( )
    return get
```

이 예에서 page()함수는 실질적으로 아무런 흥미로운 일도 수행하지 않는다. 대신에, 단순히 호출될 때 웹 페이지의 내용을 가져오는 get() 함수를 생성하고 반환한다. get()의 실행은 프로그램에서 나중에 get()이 평가되는 순간까지 미루어진다. 다음 예를 살펴보자.

```
>>> python = page("http://www.python.org")
>>> jython = page("http://www.jython.org")
>>> python
<function get at 0x95d5f0>
>>> jython
<function get at 0x9735f0>
>>> pydata = python( )      # http://www.python.org를 얻어온다.
>>> jydata = jython( )     # http://www.jython.org를 얻어온다.
>>>
```

앞의 예에서 두 변수 python과 jython은 실제로는 get() 함수의 서로 다른 버전이다. 이 두 변수를 생성한 page() 함수는 더 이상 실행되지 않지만 get() 함수는 자신이 생성될 때 정의된 바깥쪽 변수의 값을 암묵적으로 가져오게 된다. 따라서, get() 함수가 실행되면 원래 page() 함수에 제공되었던 url 값으로 urlopen(url)을 호출하게 된다. 클로저를 잠깐 들여다보면 어떤 변수가 함께 딸려 왔는지를 볼 수 있다.

다음 예를 보자.

```
>>> python.__closure__
(<cell at 0x67f50: str object at 0x69230>,)
>>> python.__closure__[0].cell_contents
'http://www.python.org'
>>> jython.__closure__[0].cell_contents
'http://www.jython.org'
>>>
```

클로저는 일련의 함수 호출 사이에 상태 정보를 보존하는 데 아주 효율적이다.
예를 들어, 다음에 나온 간단한 카운터를 살펴보자.

```
def countdown(n):
    def next( ):
        nonlocal n
        r = n
        n -= 1
        return r
    return next

# 사용 예
next = countdown(10)
while True:
    v = next( )      # 다음 값을 가져온다.
    if not v: break
```

이 코드에서 내부 카운터 값 n을 저장하기 위해서 클로저를 사용하였다. 내부 함수 next()는 호출될 때마다 카운터 변수의 이전 값을 갱신하고 반환한다. 클로저에 익숙하지 않은 프로그래머라면 다음과 같이 클래스를 사용해서 비슷한 기능을 구현하려고 할 것이다.

```
class Countdown(object):
    def __init__(self,n):
        self.n = n
    def next(self):
        r = self.n
        self.n -= 1
        return r

# 사용 예
c = Countdown(10)
while True:
    v = c.next( )      # 다음 값을 가져온다.
    if not v: break
```

하지만, 카운트다운을 시작하는 값을 크게 잡고 벤치마킹을 수행해보면 클로저를 사용한 경우가 훨씬 빠르다는 것을 볼 수 있을 것이다. (저자의 컴퓨터에서 테스트하였을 경우 50% 정도 속도가 향상되었다.)

클로저는 내부 함수 환경을 기록하기 때문에 기존 함수에 추가 기능을 넣기 위해서 함수를 포장하려고 할 때 유용하게 쓰인다. 여기에 관해서는 다음 절에서 설명한다.

장식자

장식자(decorator)는 다른 함수나 클래스를 포장하는 것이 주 목적인 함수를 말한다. 장식자를 사용해서 포장하려는 객체의 작동 방식을 변경하거나 향상시키는 일을 투명하게 수행할 수 있다. 문법적으로 장식자는 다음과 같이 특수 문자 @를 사용하여 표시한다.

```
@trace
def square(x):
    return x*x
```

앞의 코드는 다음 코드를 간단히 작성한 것과 같다.

```
def square(x):
    return x*x
square = trace(square)
```

앞의 예에서 함수 square()가 정의된다. 그리고 바로 이어서 함수 객체 자체가 trace() 함수에 전달되고 trace() 함수는 원본 square를 교체할 객체를 반환한다. 그럼, 이렇게 하는 것이 왜 유용한지를 알아보기 위해서 다음에 나와 있는 trace의 구현을 보자.

```
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log", "w")

def trace(func):
    if enable_tracing:
        def callf(*args, **kwargs):
            debug_log.write("Calling %s: %s, %s\n" %
                            (func.__name__, args, kwargs))
            r = func(*args, **kwargs)
            debug_log.write("%s returned %s\n" % (func.__name__, r))
            return r
        return callf
    else:
        return func
```

이 코드에서 trace()는 디버깅 메시지를 출력한 다음 원본 함수 객체를 호출하는 래퍼 함수를 생성한다. square()를 호출하면 래퍼에서 write() 메서드를 통해 디버

깅 메시지가 출력되는 것을 보게 된다. trace() 함수에서 반환된 callf 함수는 원본 함수를 대체하는 역할을 하는 클로저이다. 앞에 나온 trace() 구현에서 마지막으로 언급할 점은 추적 기능 자체가 전역 변수인 enable_tracing을 통해 활성화된다는 것이다. 이 변수가 False로 설정되면 trace() 장식자는 간단히 원본 함수를 있는 그대로 반환한다. 따라서, 추적 기능을 비활성화하면 장식자를 사용함으로써 발생하는 성능 저하는 없다.

장식자는 반드시 함수나 클래스 정의 바로 앞에 별개의 줄로 써주어야 한다. 하나 이상의 장식자를 사용할 수도 있다. 다음 예를 보자.

```
@foo
@bar
@spam
def grok(x):
    pass
```

이 경우 장식자는 나열된 순서대로 적용된다. 앞의 예는 다음과 동일하다.

```
def grok(x):
    pass
grok = foo(bar(spam(grok)))
```

장식자는 인수를 받을 수 있다. 다음 예를 보자.

```
@eventhandler('BUTTON')
def handle_button(msg):
    ...
@eventhandler('RESET')
def handle_reset(msg):
    ...
```

인수가 있으면 장식자는 다음과 같이 작동한다.

```
def handle_button(msg):
    ...
temp = eventhandler('BUTTON')          # 제공된 인수로 장식자를 호출한다.
handle_button = temp(handle_button)    # 장식자에 의해서 반환된 함수를 호출한다.
```

이 경우 장식자 함수는 @ 지정자로 제공한 인수만을 받는다. 그리고 장식자 함수는 주어진 함수를 인수로 하여 호출될 새로운 함수를 반환한다. 다음 예를 보자.

```
# 이벤트 처리기 장식자
event_handlers = { }
def eventhandler(event):
    def register_function(f):
        event_handlers[event] = f
        return f
    return register_function
```

장식자는 다음과 같이 클래스 정의에 적용할 수도 있다.

```
@foo
class Bar(object):
    def __init__(self,x):
        self.x = x
    def spam(self):
        문장들
```

클래스에 대해서 장식자 함수는 항상 클래스 객체를 결과로서 반환해야 한다. 원본 클래스를 다루는 코드에서 Bar.spam 같이 클래스 멤버를 직접 참조하는 일이 있기 때문이다. 만약 장식자 함수 foo에서 함수를 반환하면 코드가 제대로 작동하지 않는다.

장식자는 재귀, 문서화 문자열, 함수 속성 등 함수와 관련된 다른 부분과 서로 기묘하게 작용한다. 여기에 관해서는 이 장에서 나중에 설명한다.

생성기와 yield

함수에서 yield 키워드를 사용하면 생성기(generator)라고 불리는 객체를 정의하게 된다. 생성기는 반복에서 사용할 값을 생성하는 함수이다. 다음 예를 보자.

```
def countdown(n):
    print("Counting down from %d" % n)
    while n > 0:
        yield n
        n -= 1
    return          # 주의: 생성기는 오직 Home만을 반환할 수 있다.
```

이 함수를 호출하면 아무 코드도 실행되지 않는다. 다음 예를 보자.

```
>>> c = countdown(10)
>>>
```

그 대신, 생성기 객체가 반환된다. 이 생성기 객체는 next()가 호출될 때마다 함수를 실행한다(파이썬 3에서는 __next__()__). 다음 예를 보자.

```
>>> c.next()          # 파이썬 3에서는 c.__next__()__를 사용한다.
Counting down from 10
10
>>> c.next()
9
```

next()가 호출되면 생성기 함수는 yield문까지 문장을 실행한다. yield문은 결과를 생성하고 next()가 다시 호출되기 전까지 함수 실행이 중단된다. 함수 실행은

yield 바로 다음에 나오는 문장에서 재개된다.

보통 생성기의 next()를 직접 호출하는 일은 없고 순서열을 소비하는 for문, sum() 또는 기타 연산에서 next()가 사용된다. 다음 예를 살펴보자.

```
for n in countdown(10):
    문장들
    a = sum(countdown(10))
```

생성기 함수는 None을 반환하거나 StopIteration 예외를 발생시킴으로써 종료를 알리고 그 순간 반복은 멈춘다. 종료 순간에 None 말고 다른 값을 반환하면 안 된다.

생성기 함수는 부분적으로만 실행되는 경우가 있다. 예를 들어, 다음 코드를 살펴보자.

```
for n in countdown(10):
    if n == 2: break
    문장들
```

이 예에서 break가 호출되면 for 루프가 중단되고 연관된 생성자는 끝까지 실행되지 않는다. 이 경우를 처리하기 위해서 생성기 객체에는 종료를 알리는 데 사용되는 메서드 close()가 있다. 생성기가 더 이상 사용되지 않거나 삭제될 경우 close()가 호출된다. 보통은 close()를 호출할 필요가 없지만 다음과 같이 직접 호출할 수도 있다.

```
>>> c = countdown(10)
>>> c.next( )
Counting down from 10
10
>>> c.next( )
9
>>> c.close( )
>>> c.next( )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

생성기 함수 안에서는 yield문에서 GeneratorExit 예외가 발생되는 것으로 close()가 호출된 것을 알 수 있다. 이 예외를 잡아서 추가적으로 정리 작업을 수행할 수 있다.

```
def countdown(n):
    print("Counting down from %d" % n)
    try:
        while n > 0:
```

```

yield n
n = n - 1
except GeneratorExit:
    print("Only made it to %d" % n)

```

GeneratorExit 예외를 잡는 것이 허용되지만 생성기 함수 안에서 예외를 처리한 다음 yield로 또 다른 값을 생성해서는 안 된다. 또한 프로그램에서 현재 생성기에 대해서 반복을 수행하고 있다면 별개의 실행 스레드나 신호 처리기(signal hanlder)에서 비동기적으로 close()를 호출해서는 안 된다.

코루틴과 yield 표현식

함수 안에서 yield문은 대입 연산자의 오른쪽에 나타나는 표현식으로 사용될 수 있다. 다음 예를 살펴보자.

```

def receiver( ):
    print("Ready to receive")
    while True:
        n = (yield)
        print("Got %s" % n)

```

이렇게 yield를 사용하는 함수를 코루틴(coroutine)이라고 하며 이 함수는 보내오는 값에 답하여 실행되는 함수이다. 이 함수의 작동 방식은 생성기와 매우 유사하다. 다음 예를 살펴보자.

```

>>> r = receiver( )
>>> r.next( ) # 첫번째 yield까지 진행한다(파이썬 3에서는 r.__next__( ))
Ready to receive
>>> r.send(1)
Got 1
>>> r.send(2)
Got 2
>>> r.send("Hello")
Got Hello
>>>

```

이 예에서 맨 처음 next() 함수 호출은 코루틴이 첫 번째 yield 표현식으로 이끄는 문장들을 실행하게 하는 데 필요하다. 이 시점에서 코루틴은 실행을 중단하고 연관된 생성기 객체 r의 send() 메서드를 통해 값이 보내지기를 기다린다. send()를 통해 전달된 값은 코루틴 안의 (yield) 표현식에 의해 반환된다. 값을 받으면 코루틴은 다음 yield문까지 문장을 실행한다.

코루틴에 대해서 next()를 먼저 호출해야 한다는 점을 종종 잊어버릴 수 있기 때문에 실수를 저지르기 쉽다. 따라서, 다음과 같이 이 단계를 자동으로 수행하는 장

식자로 코루틴을 감싸는 방법을 추천한다.

```
def coroutine(func):
    def start(*args,**kwargs):
        g = func(*args,**kwargs)
        g.next( )
        return g
    return start
```

이 장식자를 사용해서 다음과 같이 코루틴을 작성하여 사용할 수 있다.

```
@coroutine
def receiver( ):
    print("Ready to receive")
    while True:
        n = (yield)
        print("Got %s" % n)

# 사용 예
r = receiver( )
r.send("Hello World")      # 주의: 최초 .next( ) 호출이 필요 없다.
```

코루틴은 직접 종료시키거나 스스로 종료하는 경우가 아니라면 보통 계속해서 실행된다. 입력 값 스트림을 닫으려면 다음과 같이 close() 메서드를 호출하면 된다.

```
>>> r.close( )
>>> r.send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

일단 코루틴이 닫히면 추가 값을 코루틴에 보낼 경우 StopIteration 예외가 발생한다. close()를 호출하면 생성기에 관한 앞 절에서 설명한 것처럼 GeneratorExit 예외가 발생한다. 다음 예를 살펴보자.

```
def receiver( ):
    print("Ready to receive")
    try:
        while True:
            n = (yield)
            print("Got %s" % n)
    except GeneratorExit:
        print("Receiver done")
```

코루틴 안에서 throw(exctype [, value [, tb]]) 메서드를 사용해서 예외를 발생시킬 수도 있다. 여기서 exctype은 예외 타입을, value는 예외 값을, tb는 역추적 객체를 나타낸다. 다음 예를 보자.

```
>>> r.throw(RuntimeError,"You're hosed!")
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "<stdin>", line 4, in receiver
RuntimeError: You're hosed!
```

이렇게 던져진 예외는 코루틴에서 현재 실행 중인 yield문에서 발생한다. 코루틴에서는 이 예외를 잡아서 적절히 처리할 수도 있다. 비동기적으로 코루틴에 신호를 보내기 위해서 throw()를 사용하는 것은 안전하지 않다. 별개의 실행 스레드나 신호 처리기에서 throw()를 호출하면 안 된다.

yield 표현식에 값을 줄 경우 코루틴은 값을 받는 동시에 반환할 수 있다. 다음 예를 보자.

```
def line_splitter(delimiter=None):
    print("Ready to split")
    result = None
    while True:
        line = (yield result)
        result = line.split(delimiter)
```

이 경우 이전과 동일한 방법으로 코루틴을 사용하면 된다. 하지만, 이제 send()를 호출하면 값이 생성된다. 다음 예를 보자.

```
>>> s = line_splitter(",")
>>> s.next()
Ready to split
>>> s.send("A,B,C")
['A', 'B', 'C']
>>> s.send("100,200,300")
['100', '200', '300']
>>>
```

이 예의 실행 과정을 이해하는 것이 매우 중요하다. 첫 번째로 next() 호출을 하면 코루틴이 (yield result)까지 진행하고 result의 기본 값인 None을 반환한다. 이어서는 send()의 호출에서는 받은 값을 line에 저장하고 분할해서 result에 넣는다. send()는 다음 yield문을 만났을 때 전달된 값을 반환한다. 즉, send()의 반환 값은 send()에 전달된 값을 받는 역할을 수행한 yield 표현식의 값이 아니라 그 다음 yield 표현식의 값이다.

코루틴이 값을 반환한다면 throw()에 의해서 발생된 예외를 처리할 때 주의를 기울여야 한다. 코루틴에서 throw()로 예외를 발생시키면 다음 yield에 전달된 값이 throw()의 결과로 반환된다. 이 값은 저장해놓지 않으면 사라져버린다.

생성기와 코루틴의 사용

얼핏 보아서는 어떻게 생성기와 코루틴을 실용적인 문제 해결에 사용할 수 있을지 잘 떠오르지 않을 것이다. 하지만, 생성기와 코루틴은 시스템, 네트워크, 분산 처리 등 특정한 종류의 프로그래밍을 수행할 때 대단히 효과적으로 사용할 수 있다. 예를 들어, 생성기 함수는 유닉스 셸에서 파이프를 사용하는 것과 유사한 처리 파이프라인을 만들고자 할 때 유용하게 쓰인다. 1장에서 이와 관련된 한 예를 살펴보았다. 다음은 파일을 찾고 열고 읽고 처리하는 데 생성기 함수를 사용하는 또 다른 예를 보여준다.

```
import os
import fnmatch
def find_files(topdir, pattern):
    for path, dirname, filelist in os.walk(topdir):
        for name in filelist:
            if fnmatch.fnmatch(name, pattern):
                yield os.path.join(path, name)

import gzip, bz2
def opener(filenames):
    for name in filenames:
        if name.endswith(".gz"):      f = gzip.open(name)
        elif name.endswith(".bz2"):    f = bz2.BZ2File(name)
        else: f = open(name)
        yield f

def cat(filelist):
    for f in filelist:
        for line in f:
            yield line

def grep(pattern, lines):
    for line in lines:
        if pattern in line:
            yield line
```

다음은 이 함수들을 사용하여 처리 파이프라인을 만드는 예이다.

```
wwwlogs = find_files("www", "access-log*")
files = opener(wwwlogs)
lines = cat(files)
pylines = grep("python", lines)
for line in pylines:
    sys.stdout.write(line)
```

이 예에서, 프로그램은 최상위 디렉터리 “www”에 속한 모든 하위 디렉터리에서 발견할 수 있는 모든 “access-log*” 파일에 대해 전체 줄을 처리한다. 각 “access-log” 파일은 압축 방식에 따라 적절한 파일 열기 객체를 사용하여 열린다. 모든 줄

은 서로 연결되고 부분 문자열 “python”을 찾는 필터를 통해 처리된다. 마지막에 있는 for문에 의해서 전체 과정이 진행된다. for 루프는 각 반복마다 파이프라인을 통해 새로운 값을 가져와서 소비한다. 이 구현은 임시 리스트나 기타 대규모 자료 구조를 전혀 생성하지 않으므로 메모리 사용 측면에서 매우 효율적이다.

코루틴은 데이터 흐름 처리에 기반한 프로그램을 작성할 때 사용할 수 있다. 이렇게 작성된 프로그램은 파이프라인을 거꾸로 뒤집은 것처럼 보인다. for 루프를 사용하여 생성기 함수를 통해서 값을 가져오는 대신에 서로 연결된 코루틴에 값을 보낸다. 다음은 앞에 나온 생성기 함수를 흉내 내도록 코루틴을 작성한 예이다.

```
import os
import fnmatch

@coroutine
def find_files(target):
    while True:
        topdir, pattern = (yield)
        for path, dirname, filelist in os.walk(topdir):
            for name in filelist:
                if fnmatch.fnmatch(name, pattern):
                    target.send(os.path.join(path, name))

import gzip, bz2
@coroutine
def opener(target):
    while True:
        name = (yield)
        if name.endswith(".gz"): f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        target.send(f)

@coroutine
def cat(target):
    while True:
        f = (yield)
        for line in f:
            target.send(line)

@coroutine
def grep(pattern, target):
    while True:
        line = (yield)
        if pattern in line:
            target.send(line)

@coroutine
def printer():
    while True:
        line = (yield)
        sys.stdout.write(line)
```

다음은 어떻게 코루틴들을 연결시켜서 데이터 흐름 처리 파이프라인을 만드는지를 보여준다.

```
finder = find_files(opener(cat(grep("python", printer( )))))

# 값을 보낸다.
finder.send(("www","access-log*"))
finder.send(("otherwww","access-log*"))
```

이 예에서 각 코루틴은 인수 target으로 지정된 다른 코루틴으로 데이터를 보낸다. 앞에 나온 생성기를 사용하는 예와 달리, 실행 과정이 첫 번째 코루틴 find_files()에 데이터를 밀어 넣음으로써 진행된다. 이 코루틴은 다시 다음 단계로 데이터를 밀어 넣는다. 이 예에서 주목할 점은 코루틴 파이프라인이 close()를 직접 호출하기 전까지 계속 활성 상태로 있다는 점이다. 따라서, 앞의 예에서 send()를 두 번 호출한 것처럼 필요할 때마다 코루틴에 데이터를 계속 밀어 넣을 수 있다.

코루틴은 병행 프로그램을 작성하는 데 사용할 수 있다. 예를 들어, 중앙집중식 작업 관리자 또는 이벤트 루프에서 스케줄링을 담당하면서 데이터를 다양한 작업을 처리하는 수백 또는 수천 개의 코루틴에 보내는 예를 생각해볼 수 있다. 데이터가 코루틴에 ‘전달’되기 때문에 코루틴을 사용하면 프로그램의 구성 요소 사이에 통신을 하기 위해서 필요한 메시지 큐라든지 메시지 전달 기능을 구현하기가 쉽다. 여기에 관해서는 20장에서 자세하게 설명한다.

리스트 내포

함수와 관련해서, 리스트의 각 아이템에 함수를 적용하고 그 결과를 가지고 새로운 리스트를 생성하는 연산을 자주 사용하게 된다. 다음 예를 살펴보자.

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    squares.append(n * n)
```

이러한 형태의 연산은 아주 흔하기 때문에 리스트 내포(list comprehension)라고 부르는 연산자가 만들어지게 되었다. 간단한 예를 살펴보자.

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

리스트 내포의 일반적인 문법은 다음과 같다.

```
[표현식 for 항목1 in 반복가능객체1 if 조건1
  for 항목2 in 반복가능객체2 if 조건2
  ...
  for 항목N in 반복가능객체N if 조건N ]
```

이 문법은 대략 다음 코드와 같다.

```
s = []
for 항목1 in 반복가능객체1:
    if 조건1:
        for 항목2 in 반복가능객체2:
            if 조건2:
                ...
                for 항목N in 반복가능객체N:
                    if 조건N: s.append(표현식)
```

더 많은 예를 살펴보자.

```
a = [-3, 5, 2, -10, 7, 8]
b = 'abc'

c = [2*s for s in a]          # c = [-6, 10, 4, -20, 14, 16]
d = [s for s in a if s>=0]    # d = [5, 2, 7, 8]
e = [(x,y) for x in a         # e = [(5,'a'), (5,'b'), (5,'c'),
      for y in b               #   (2,'a'), (2,'b'), (2,'c'),
      if x > 0 ]              #   (7,'a'), (7,'b'), (7,'c'),
                            #   (8,'a'), (8,'b'), (8,'c')]

f = [(1,2), (3,4), (5,6)]
g = [math.sqrt(x*x+y*y)       # g = [2.23606, 5.0, 7.81024]
      for x,y in f]
```

리스트 내포에서 사용되는 순서열은 길이가 서로 같을 필요가 없다. 그 이유는 앞에서 보았듯이 중첩된 for 루프를 사용하여 순서열에 대해 반복을 수행하기 때문이다. 결과 리스트에는 일련의 표현식 값이 저장된다. if 절은 생략 가능하다. if 절이 사용되면, 조건이 참일 때만 표현식이 평가되어 결과에 추가된다.

리스트 내포를 사용하여 튜플 리스트를 구축할 때는 튜플 값을 반드시 괄호로 묶어주어야 한다. 예를 들어, [(x,y) for x in a for y in b]는 문법적으로 맞지만, [x,y for x in a for y in b]는 문법적으로 틀리다.

마지막으로, 파이썬 2에서는 리스트 내포 안에서 정의한 반복 변수가 현재 유효 범위 안에서 평가되고 리스트 내포의 수행이 끝난 후에도 남겨지게 된다. 예를 들어, [x for x in a]에서 반복 변수 x는 이전에 이미 존재하는 x 값이 있으면 그것을 덮어쓰고 결과 리스트가 생성된 후에 a의 마지막 항목 값으로 설정된다. 다행히 이 부분은 파이썬 3에는 해당하지 않으며 반복 변수는 내부(private) 변수로만 사용된다.

생성기 표현식

생성기 표현식(generator expression)은 리스트 내포와 동일한 계산을 수행하지만 결과를 반복적으로 생성하는 객체이다. 문법은 대괄호 대신 괄호를 사용한다는 점 만 제외하면 리스트 내포를 사용할 때의 문법과 동일하다. 다음은 생성기 표현식의 일반적인 문법을 보여준다.

```
(표현식 for 항목1 in 반복가능객체1 if 조건1
      for 항목2 in 반복가능객체2 if 조건2
      ...
      for 항목N in 반복가능객체N if 조건N)
```

리스트 내포와 다르게 생성기 표현식은 실제로 리스트를 생성하거나 괄호 안에서 표현식을 즉시 평가하지 않는다. 대신에 반복을 통해 필요할 때 값을 생성하는 생성기 객체를 반환한다. 다음 예를 살펴보자.

```
>>> a = [1, 2, 3, 4]
>>> b = (10*i for i in a)
>>> b
<generator object at 0x590a8>
>>> b.next( )
10
>>> b.next( )
20
...
...
```

리스트와 생성기 표현식에는 차이가 있지만 그 차이는 미세하다. 리스트 내포에 대해서 파이썬은 실제로 결과 데이터를 담은 리스트를 생성한다. 생성기 표현식에 대해서 파이썬은 단순히 필요할 때 데이터를 어떻게 생성하는지를 알고 있는 생성기를 반환한다. 특정 응용 프로그램에서 이 차이는 성능과 메모리 사용 효율을 크게 향상시킬 수 있다. 다음 예를 살펴보자.

```
# 파일을 읽음
f = open("data.txt")
lines = (t.strip( ) for t in f) # 파일을 연다.
# 줄을 읽어서 앞뒤 공백을
# 벗겨낸다.
comments = (t for t in lines if t[0] == '#') # 모든 주석
for c in comments:
    print(c)
```

이 예에서 줄을 추출하여 공백을 벗겨내는 생성기 표현식은 실제로 전체 파일을 메모리에 읽어들이지 않는다. 주석을 가져오는 부분도 마찬가지이다. 파일 안에 있는 줄들은 뒤에 나오는 for 루프에서 반복을 시작할 때 읽어들이기 시작한다. 반복을 수행하는 동안 줄들이 필요에 따라 생성되어 적절히 걸러진다. 실제로 이러한 동

안 전체 파일이 메모리에 로드되는 일은 없다. 따라서, 이 코드는 기가바이트 크기의 파이썬 소스 파일로부터 주석을 추출한다고 할 때 매우 효율적으로 작동하게 된다.

리스트 내포와 다르게 생성기 표현식은 순서열처럼 작동하는 객체를 생성하지 않는다. 즉, 색인될 수 없고 보통의 리스트 연산은 작동하지 않는다(예를 들면, `append()`). 생성기 표현식은 다음과 같이 내장 `list()` 함수를 사용하여 리스트로 변환할 수 있다.

```
clist = list(comments)
```

선언형 프로그래밍

리스트 내포와 생성기 표현식은 선언형 프로그래밍에서 볼 수 있는 연산과 깊은 관련이 있다. 사실 이것은 어느 정도 수리 집합 이론의 아이디어에서 유래된 것이다. 예를 들어, $\{x^2 \mid x \in a, x > 0\}$ 같은 집합을 기술하는 것과 어느 정도 유사하다.

데이터에 대해 직접 반복을 수행하는 프로그램을 작성하는 대신에, 간단히 모든 데이터에 한 번에 적용되는 일련의 계산으로 프로그램을 구조화하기 위해서 선언형 프로그램을 작성할 수 있다. 예를 들어, 다음과 같이 주식 포트폴리오를 담은 “`portfolio.txt`” 파일이 있다고 하자.

```
AA 100 32.20
IBM 50 91.10
CAT 150 83.44
MSFT 200 51.23
GE 95 40.37
MSFT 50 65.10
IBM 100 70.44
```

다음은 두 번째 열에 세 번째 열을 곱하고 전부 더해서 총 비용을 계산하는 선언형 스타일의 프로그램을 보여준다.

```
lines = open("portfolio.txt")
fields = (line.split() for line in lines)
print(sum(float(f[1]) * float(f[2]) for f in fields))
```

이 프로그램에서 파일에 대해 한 줄씩 반복을 수행하는 매커니즘에 관해서는 신경 쓸 필요가 없다. 대신에 모든 데이터에 대해 수행될 계산들을 선언하기만 하면 된다. 이 접근법을 취하면 코드가 매우 간단해질 뿐만 아니라 다음에 나오는 전형

적인 버전보다 더 빠른 경우가 많다.

```
total = 0
for line in open("portfolio.txt"):
    fields = line.split( )
    total += float(fields[1]) * float(fields[2])
print(total)
```

선언형 프로그래밍 스타일은 유닉스 셸에서 수행하는 일과 다소 관련이 있다. 예를 들어, 앞에 나온 생성기 표현식을 사용한 예는 다음에 나오는 한 줄짜리 awk 명령어와 유사하다.

```
% awk '{ total += $2 * $3} END { print total }' portfolio.txt
44671.2
%
```

리스트 내포와 생성기 표현식의 선언형 스타일은 데이터베이스를 처리할 때 흔히 사용하는 SQL select문을 작동 방식을 흉내 내는 데 사용할 수도 있다. 예를 들어, 사전들의 리스트로 데이터를 읽어오는 다음 예를 보자.

```
fields = (line.split( ) for line in open("portfolio.txt"))
portfolio = [ {'name' : f[0],
              'shares' : int(f[1]),
              'price' : float(f[2]) }
              for f in fields]

# 몇몇 질의
msft = [s for s in portfolio if s['name'] == 'MSFT']
large_holdings = [s for s in portfolio
                   if s['shares']*s['price'] >= 10000]
```

사실 데이터베이스 접근(17장을 보라)에 관련된 모듈을 사용할 때 데이터베이스 질의와 리스트 내포를 함께 사용하는 경우가 종종 있다. 다음 예를 살펴보자.

```
sum(shares*cost for shares,cost in
     cursor.execute("select shares, cost from portfolio"
                    " if shares*cost >= 10000))
```

lambda 연산자

다음과 같은 lambda문을 사용하여 표현식 형태로 된 익명 함수를 작성할 수 있다.

```
lambda args : expression
```

args는 콤마로 분리된 인수 목록이고 expression은 인수와 관련된 표현식이다.

다음 예를 보자.

```
a = lambda x,y : x+y
r = a(2,3)      # r은 5가 된다.
```

lambda와 함께 사용되는 코드는 반드시 유효한 표현식이어야 한다. 여러 문장 또는 기타 for와 while 같은 표현식이 아닌 문장은 lambda문에서 쓸 수 없다. lambda 표현식에는 함수와 동일한 유효 범위 규칙이 적용된다.

lambda는 간단한 콜백 함수를 구현하는 데 주로 사용된다. 예를 들어, 대소문자를 구분하지 않고 이름 목록을 정렬하는 코드를 다음과 같이 작성할 수 있다.

```
names.sort(key=lambda n: n.lower( ))
```

재귀

재귀 함수는 다음 예처럼 쉽게 정의할 수 있다.

```
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1)
```

하지만, 재귀 함수 호출의 깊이에 제한이 있다는 점에 유의하여야 한다. 함수 sys.getrecursionlimit()는 현재 최대 재귀 깊이를 반환하고 함수 sys.setrecursionlimit()는 이 값을 변경하는 데 사용된다. 기본 값은 1000이다. 이 값을 증가시킬 수 있지만 여전히 재귀 깊이는 운영 체제에서 설정된 스택 크기 제한을 넘을 수 없다. 재귀 깊이를 초과하면 RuntimeError 예외가 발생한다. 파이썬은 스케미(Scheme) 같은 함수형 언어에서 종종 볼 수 있는 꼬리 재귀(tail-recursion) 최적화를 수행하지 않는다.

여러분의 기대와는 다르게 재귀는 생성기 함수나 코루틴 안에서 제대로 작동하지 않는다. 예를 들어, 다음은 중첩된 리스트에 있는 모든 항목을 출력하는 코드이다.

```
def flatten(lists):
    for s in lists:
        if isinstance(s,list):
            flatten(s)
        else:
            print(s)

items = [[1,2,3],[4,5,[5,6]],[7,8,9]]
flatten(items)      # 1 2 3 4 5 6 7 8 9를 출력한다.
```

여기서 print를 yield로 바꾸면 코드가 더 이상 제대로 작동하지 않는다. 그 이유는 flatten()을 재귀적으로 호출해봐야 실제로 반복을 수행하는 것이 아니라 단순히 새로운 생성기 객체만 생성하기 때문이다. 다음은 제대로 작동하는 재귀적인 생

성기의 버전이다.

```
def genflatten(lists):
    for s in lists:
        if isinstance(s, list):
            for item in genflatten(s):
                yield item
        else:
            yield item
```

재귀 함수와 장식자를 같이 사용할 때도 주의가 필요하다. 재귀 함수에 장식자를 사용하면 모든 내부 재귀 함수 호출이 이제 장식이 적용된 버전을 거치게 된다. 다음 예를 보자.

```
@locked
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1) # factorial의 래퍼를 호출한다.
```

장식자를 사용하는 목적이 동기화나 락 같은 시스템 관리와 연관된 것이라면 재귀는 가급적 사용하지 않는 것이 가장 좋다.

문서화 문자열

함수의 첫 번째 문장은 주로 함수의 사용법을 설명하는 문서화 문자열인 경우가 많다. 다음 예를 보자.

```
def factorial(n):
    """Computes n factorial. For example:

    >>> factorial(6)
    120
    >>>
    """
    if n <= 1: return 1
    else: return n*factorial(n-1)
```

문서화 문자열은 함수의 `__doc__` 속성에 저장되고 주로 IDE에서 대화식 도움말을 제공하는 데 사용된다.

장식자를 사용할 때 장식자에 의해서 생성되는 래퍼로 인해 문서화 문자열과 관련된 도움말 기능이 제대로 작동하지 않을 수도 있다. 다음 코드를 살펴보자.

```
def wrap(func):
    def call(*args, **kwargs):
        return func(*args, **kwargs)
    return call
```

```
@wrap
def factorial(n):
    """Computes n factorial."""
    ...
```

사용자가 이 버전의 `factorial()`에 대해 도움말을 요청하면 알 수 없는 말이 출력된다.

```
>>> help(factorial)
Help on function call in module __main__:

def call(*args, **kwargs)
(END)
>>>
```

이 문제를 해결하려면 다음과 같이 함수 이름과 문서화 문자열을 가져오도록 장식자 함수를 작성해야 한다.

```
def wrap(func):
    def call(*args,**kwargs):
        return func(*args,**kwargs)
    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    return call
```

이 문제가 자주 발생하다보니 파이썬에는 `functools` 모듈이 있어서 이 속성들을 자동으로 복사하는 `wraps`라는 함수가 제공된다. 예상대로 이 함수도 장식자이다.

```
from functools import wraps
def wrap(func):
    @wraps(func)
    def call(*args,**kwargs):
        return func(*args,**kwargs)
    return call
```

`functools`에 있는 `@wraps(func)` 장식자는 `func`의 속성을 정의될 래퍼 함수로 복사한다.

함수 속성

함수는 그 자체가 임의의 속성을 가질 수 있다. 다음 예를 보자.

```
def foo( ):
    문장들

foo.secure = 1
foo.private = 1
```

함수 속성은 함수의 `__dict__` 속성에 저장된 사전에 담긴다.

함수 속성은 파서 생성기라든지 함수 객체에 추가 정보를 붙이려는 응용 프레임워크 같은 아주 특수한 곳에서 주로 사용된다.

문서화 문자열처럼 장식자와 함수 속성을 함께 사용할 경우 주의하여야 한다. 함수에 장식자가 적용된 경우 속성에 접근하면 원래 함수가 아니라 장식자 함수의 속성이 접근하게 된다. 이것은 경우에 따라 의도된 것일 수도 또는 그렇지 않을 수도 있다. 이미 정의된 함수 속성을 장식자 함수에 전달하려면 다음에 나온 템플릿을 사용하거나 앞 절에서 살펴본 `functools.wraps()` 장식자를 사용하도록 한다.

```
def wrap(func):
    def call(*args, **kwargs):
        return func(*args, **kwargs)
    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    call.__dict__.update(func.__dict__)
    return call
```

`eval()`, `exec()`과 `compile()`

`eval(str [,globals [,locals]])` 함수는 표현식을 담은 문자열을 실행하고 그 결과를 반환한다. 다음 예를 보자.

```
a = eval('3*math.sin(3.5+x) + 7.2')
```

비슷하게 `exec(str [, globals [, locals]])` 함수는 임의의 파이썬 코드를 담은 문자열을 실행한다. `exec()`에 의해서 실행되는 코드는 마치 코드가 `exec` 자리에 대신 있는 것처럼 실행된다. 다음 예를 보자.

```
a = [3, 5, 10, 13]
exec("for i in a: print(i)")
```

`exec()`과 관련해서 한 가지 주의할 점은 파이썬 2에서 `exec`가 실제로는 문장으로 정의되어 있다는 점이다. 이에 따라, 레거시 코드에서 `exec "for i in a: print i"` 같이 둘러싸는 괄호 없이 `exec`를 호출하는 문장을 볼 수 있을 것이다. 이러한 문장은 파이썬 2.6에서는 여전히 작동하지만 파이썬 3에서는 제대로 작동하지 않는다. 최신 프로그램에서는 `exec()`를 함수처럼 사용해야 한다.

이 두 함수 모두 호출자의 네임스페이스 안에서 코드를 실행한다(문자열이나 파일 안에 나타나는 기호를 해석하는 데 이 네임스페이스가 사용된다). 선택적으로 `eval()`과 `exec()`에 실행할 코드를 위한 전역 네임스페이스와 지역 네임스페이스 역할을 하는 하나 또는 두 개의 매핑 객체를 넘겨줄 수도 있다. 다음 예를 보자.

```

globals = {'x': 7,
           'y': 10,
           'birds': ['Parrot', 'Swallow', 'Albatross']
       }
locals = { }

# 위의 사전을 전역과 지역 네임스페이스로 사용하여 실행한다.
a = eval("3 * x + 4 * y", globals, locals)
exec("for b in birds: print(b)", globals, locals)

```

네임스페이스를 지정하지 않으면 현재의 전역 네임스페이스와 지역 네임스페이스가 사용된다. 또한, 중첩 유효 범위와 관련된 문제로 인해 어떤 함수가 중첩 함수 정의를 포함하거나 lambda 연산자를 사용하고 있을 경우 그 함수 안에서 exec()를 사용하면 SyntaxError 예외가 발생할 수 있다.

문자열이 exec() 또는 eval()로 전달되면 파서는 먼저 문자열을 바이트코드로 컴파일한다. 이 작업은 오래 걸리기 때문에 같은 코드가 여러 번 실행될 경우에는 코드를 미리 컴파일하여 이어지는 호출에서 바이트코드를 재사용하는 것이 좋다.

compile(str, filename, kind) 함수는 문자열을 바이트코드로 컴파일한다. 여기서 str은 컴파일될 코드를 포함하는 문자열이고 filename은 문자열이 정의된 파일(역 추적 정보를 생성할 때 사용)이다. kind 인수는 컴파일될 코드의 종류를 지정한다(단일 문장인 경우 single, 여러 문장에 대해서는 exec, 표현식은 eval). compile() 함수에 의해 반환되는 코드 객체는 eval() 함수와 exec() 문에 전달할 수 있다. 다음 예를 보자.

```

s = "for i in range(0,10): print(i)"
c = compile(s, 'exec')          # 코드 객체로 컴파일한다.
exec(c)                        # 코드 객체를 실행한다.

s2 = "3 * x + 4 * y"
c2 = compile(s2, 'eval')        # 표현식으로 컴파일한다.
result = eval(c2)              # 표현식을 평가한다.

```


7장

P y t h o n E s s e n t i a l R e f e r e n c e

클래스와 객체지향 프로그래밍

클래스는 새로운 객체를 생성하는 데 사용되는 메커니즘이다. 이 장에서는 클래스에 관해서 자세하게 다룬다. 하지만, 객체지향 프로그래밍이나 설계에 관한 심층적인 참고 자료를 제공하는 것이 이 장의 목적은 아니다. 또한 이 장은 독자가 C나 자바 같은 언어로 데이터 구조와 객체지향 프로그래밍에 대해서 어느 정도 경험을 해보았다고 가정하고 썼다(기타 객체와 관련된 용어나 객체의 내부 구현에 대해서는 3장에 설명되어 있다).

class문

클래스(class)는 인스턴스(instance)라고 부르는 객체에 연결되고 또 이를 객체 사이에 공유되는 속성을 정의한다. 일반적으로 클래스는 함수(메서드method라고 한다), 변수(클래스 변수class variable라고 한다), 그리고 계산된 속성(프로퍼티property라고 한다)을 모아 놓은 것이다.

클래스는 class문으로 정의한다. 클래스의 몸체는 클래스가 정의되는 도중에 실행되는 문장을 담고 있다. 다음 예를 보자.

```
class Account(object):
    num_accounts = 0
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1
    def __del__(self):
        Account.num_accounts -= 1
```

```

def deposit(self,amt):
    self.balance = self.balance + amt
def withdraw(self,amt):
    self.balance = self.balance - amt
def inquiry(self):
    return self.balance

```

클래스 몸체가 실행되는 도중에 생성되는 값은 모듈처럼 네임스페이스의 역할을 하는 클래스 객체에 저장된다. 예를 들어, Account 클래스의 멤버에는 다음과 같이 접근한다.

```

Account.num_accounts
Account.__init__
Account.__del__
Account.deposit
Account.withdraw
Account.inquiry

```

class문 자체는 클래스의 인스턴스를 생성하지 않는다(예를 들어, 앞의 예에서 Account 클래스의 인스턴스는 생성되지 않는다). 클래스는 단지 나중에 생성될 모든 인스턴스에 공통적인 속성을 설정하는 일을 할 뿐이다. 어떻게 보면 클래스는 청사진과도 같다.

클래스 안에서 정의되는 함수는 인스턴스 메서드(instance method)라고 한다. 인스턴스 메서드는 첫 번째 인수로 넘어오는 클래스의 인스턴스에 대해서 작동하는 함수이다. 적법한 식별자 이름이라면 아무 것이나 이 첫 번째 인수를 가리키는 데 쓸 수 있지만, 관례적으로 self를 쓴다. 앞의 예에서 deposit(), withdraw(), inquiry()는 모두 인스턴스 메서드이다.

num_accounts 같은 클래스 변수는 클래스의 모든 인스턴스에서 공유되는 값을 담는다. 앞의 예에서 num_accounts 변수는 현재 존재하는 Account 인스턴스의 개수를 기록한다.

클래스 인스턴스

클래스의 인스턴스는 클래스 객체를 함수로서 호출함으로써 생성한다. 그러면 새로운 인스턴스가 생성되고 클래스의 __init__() 메서드에 이 인스턴스가 전달된다. __init__()는 새롭게 생성된 인스턴스 self와 클래스 객체를 함수로서 호출할 때 제공한 인수를 받는다. 다음 예를 보자.

```
# 계좌를 몇 개 생성한다.
```

```
a = Account("Guido", 1000.00) # Account.__init__(a,"Guido",1000.00)
를 호출한다.
b = Account("Bill", 10.00)
```

`__init__()` 안에서 `self`에 무언가를 대입하면 인스턴스에 속성이 추가된다. 예를 들어, `self.name = name`은 `name` 속성을 인스턴스에 추가한다. 일단 새 인스턴스를 얻었으면 점(.) 연산자를 사용해서 인스턴스나 클래스의 속성에 접근할 수 있다.

```
a.deposit(100.00) # Account.deposit(a,100.00)를 호출한다.
b.withdraw(50.00) # Account.withdraw(b,50.00)를 호출한다.
name = a.name      # 계좌의 이름을 가져온다.
```

점 연산자는 속성 바인딩(attribute binding)을 수행한다. 속성에 접근할 때 속성의 값은 여러 곳에서 올 수 있다. 예를 들어, 앞의 예에서 `a.name`은 인스턴스 `a`의 속성 `name`을 반환한다. 반면에 `a.deposit`은 `Account` 클래스의 `deposit` 속성(메서드다)를 반환한다. 속성에 접근하면 먼저 인스턴스를 검사하고 해당 속성이 없으면 인스턴스의 클래스를 계속해서 검사한다. 이러한 내부적인 메커니즘을 통해 클래스의 속성이 모든 인스턴스에서 공유된다.

유효 범위 규칙

클래스는 새로운 네임스페이스를 정의하지만 메서드 안에서 참조하는 이름에 대한 유효 범위를 생성해주지는 않는다. 따라서 클래스의 메서드를 작성할 때 속성이나 메서드에 대한 참조는 항상 완전히 한정되어야(fully qualified) 한다. 이에 따라, 앞의 예에서 `balance`라고 쓰지 않고 `self.balance`라고 썼다. 다음 예에서 볼 수 있듯이, 어느 한 메서드에서 다른 메서드를 호출할 때에도 마찬가지로 적용된다.

```
class Foo(object):
    def bar(self):
        print("bar!")
    def spam(self):
        bar(self)      # 틀림! 'bar'는 NameError 예외를 발생시킨다.
        self.bar()     # 제대로 작동한다.
        Foo.bar(self) # 역시 제대로 작동한다.
```

클래스에서 유효 범위가 생성되지 않는 점은 파이썬이 C++나 자바와 다른 점 중 하나이다. C++나 자바를 써본 적이 있으면 파이썬의 `self` 매개변수를 `this` 포인터와 같다고 생각하면 된다. 파이썬에서는 변수를 명시적으로 선언할 수 있는 방법(즉, C에서 `int x`나 `float y` 같은 선언)이 없기 때문에 `self`를 써주어야 한다. 그렇지 않으면 메서드에서 변수에 값을 대입할 때 이 값이 지역 변수에 대입되어야 하는지 인스

턴스 속성에 저장되어야 하는지 알 수 있는 방법이 없다. `self`를 직접 써줌으로써 이 문제를 해결한다. `self`에 저장되는 값은 모두 인스턴스의 일부가 되고 나머지는 모두 지역 변수에 저장된다.

상속

상속(inheritance)은 기존 클래스의 작동 방식을 특수화하거나 변경함으로써 새 클래스를 만드는 메커니즘이다. 원본 클래스는 기반 클래스(base class) 또는 상위 클래스(superclass)라고 부른다. 새 클래스는 파생 클래스(derived class) 또는 하위 클래스(subclass)라고 부른다. 클래스가 상속을 통해 생성되면 이 클래스는 기반 클래스에서 정의된 속성을 상속받는다. 파생 클래스는 상속받은 속성을 재정의할 수 있고 자신만의 새로운 속성을 가질 수도 있다.

상속 관계를 지정하려면 `class`문에서 기반 클래스의 이름을 콤마로 구분하여 써주면 된다. 특별히 기반 클래스가 없는 경우에는 앞의 예에서 보았듯이 `object`로부터 상속받는다. `object`는 모든 파이썬 객체의 조상 클래스이며 출력에 쓰일 문자열을 생성하는 `__str__()` 메서드 같은 몇몇 공통되는 메서드의 기본 구현을 제공한다.

상속은 종종 기존 메서드의 작동 방식을 재정의하는 데 사용된다. 한 가지 예로서, 다음 코드는 자신의 현재 상황에 주의를 기울이지 않고 있는 사람에게, 서브 프라임 모기지를 상환할 때 큰 불이익이 발생하도록 주기적으로 현재 잔고를 부풀려서 보고함으로써 계좌에서 돈을 과도하게 인출하게 만들도록 `inquiry()` 메서드를 재정의한 `Account`의 특수화된 버전을 보여준다.

```
import random
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return self.balance * 1.10 # 주의: 특히 출원 중인 아이디어
        else:
            return self.balance

c = EvilAccount("George", 1000.00)
c.deposit(10.0)           # Account.deposit(c,10.0)를 호출한다.
available = c.inquiry()   # EvilAccount.inquiry(c)를 호출한다.
```

앞의 예에서 `EvilAccount`의 인스턴스는 `inquiry()` 메서드를 재정의한 것 말고는 `Account`의 인스턴스와 같다.

점(.) 연산자의 작동 방식을 약간 개선함으로써 상속이 구현된다. 구체적으로 말하면, 속성 검색을 수행할 때 인스턴스나 인스턴스의 클래스에서 부합하는 속성을 찾지 못할 경우 기반 클래스에서 속성을 계속해서 찾는다. 이 과정은 더 찾아볼 기반 클래스가 없을 때까지 계속된다. 앞의 예에서 `c.deposit()` 호출 때 `Account` 클래스에 정의된 `deposit()`가 호출된 것도 이 때문이다.

하위 클래스에서 자신만의 `__init__()` 버전을 정의함으로써 인스턴스에 새로운 속성을 추가할 수 있다. 예를 들어, 다음은 `EvilAccount`에 새 속성 `evilfactor`를 추가한 버전을 보여준다.

```
class EvilAccount(Account):
    def __init__(self, name, balance, evilfactor):
        Account.__init__(self, name, balance) # Account를 초기화한다
        self.evilfactor = evilfactor
    def inquiry(self):
        if random.randint(0, 4) == 1:
            return self.balance * self.evilfactor
        else:
            return self.balance
```

파생 클래스에서 `__init__()`을 정의하면 기반 클래스의 `__init__()`은 자동으로 호출되지 않는다. `__init__()` 메서드를 호출해서 기반 클래스를 적절하게 초기화하는 일은 파생 클래스의 몫이다. 앞의 예에도 `Account.__init__()`을 호출하는 부분이 있다. 기반 클래스에서 `__init__()`을 정의하지 않았다면 `__init__()` 호출을 생략해도 된다. 기반 클래스에서 `__init__()`을 정의하였는지 잘 모를 경우에는 단순히 인수 없이 `__init__()`을 호출해도 된다. 항상 아무 일도 하지 않는 기본 구현이 제공되기 때문이다.

파생 클래스에서 어떤 메서드를 다시 구현하였지만 원래 메서드의 구현을 호출하고 싶은 경우가 종종 있다. 이를 위해서는 다음과 같이 기반 클래스의 원래 메서드를 직접 호출하면서 첫 번째 매개변수로 인스턴스 `self`를 넘겨주면 된다.

```
class MoreEvilAccount(EvilAccount):
    def deposit(self, amount):
        self.withdraw(5.00) # 수수료를 제한다.
        EvilAccount.deposit(self, amount) # 이제 예금한다.
```

이 예에서 `EvilAccount`는 실제로 `deposit()` 메서드를 구현하고 있지 않다. `deposit()` 메서드는 `Account` 클래스에서 구현된 것이다. 앞의 코드는 문제 없이 작동한다. 하지만 다른 사람이 이 코드를 보고 혼동할 수 있다. (`deposit()`가 `EvilAccount`에서 구현된 것인가?) 다음과 같이 `super()` 함수를 사용하는 방법도 있다.

```
class MoreEvilAccount(EvilAccount):
    def deposit(self, amount):
        self.withdraw(5.00)           # 수수료를 제한다.
        super(MoreEvilAccount, self).deposit(amount)  # 이제 예금한다.
```

super(cls, instance)는 기반 클래스에 대해 속성 검색을 수행하는 특수한 객체를 반환한다. 속성 검색은 보통의 검색 규칙에 따라 기반 클래스에 대해 수행된다. 이제 메서드의 정확한 위치를 써줄 필요가 없기 때문에 메서드 호출 의도를 더욱 명확하게 표현할 수 있다(즉, 어느 기반 클래스에서 정의된 것인지에 상관 없이 기존 구현을 호출하고 싶다). 불행히도 super()의 문법에는 개선의 여지가 많다. 파이썬 3에서는 더욱 간략화된 super().deposit(amount) 문을 사용하여 앞의 예와 동일한 일을 수행할 수 있다. 파이썬 2에서는 더 장황한 문법을 사용할 수밖에 없다.

파이썬은 다중 상속(multiple inheritance)을 지원한다. 간단히 클래스에 여러 기반 클래스를 지정하면 된다. 다음 예를 보자.

```
class DepositCharge(object):
    fee = 5.00
    def deposit_fee(self):
        self.withdraw(self.fee)

class WithdrawCharge(object):
    fee = 2.50
    def withdraw_fee(self):
        self.withdraw(self.fee)

# 다중 상속을 사용하는 클래스
class MostEvilAccount(EvilAccount, DepositCharge, WithdrawCharge):
    def deposit(self, amt):
        self.deposit_fee()
        super(MostEvilAccount, self).deposit(amt)
    def withdraw(self, amt):
        self.withdraw_fee()
        super(MostEvilAccount, self).withdraw(amt)
```

다중 상속에서는 속성 바인딩을 수행할 때 많은 검색 경로가 존재할 수 있기 때문에 속성을 분석하는 일이 훨씬 복잡해진다. 다음 예를 통해서 이 과정의 복잡성에 관해 감을 잡아보자.

```
d = MostEvilAccount("Dave", 500.00, 1.10)
d.deposit_fee() # DepositCharge.deposit_fee()를 호출한다. 수수료는 5.00
d.withdraw_fee() # WithdrawCharge.withdraw_fee()를 호출한다. 수수료는 5.00?
```

앞의 예에서 deposit_fee()나 withdraw_fee() 같은 메서드는 고유한 이름을 갖기 때문에 관련 기반 클래스에서 적절히 찾을 수 있다. 그러나, withdraw_fee() 함수는 제대로 작동하지 않는 것 같다. 왜냐하면 이 메서드는 자신의 클래스에서 초

기화된 fee 값을 사용하지 않기 때문이다. 앞의 코드를 보면 속성 fee가 두 기반 클래스에서 클래스 변수로 정의되어 있다. 두 메서드 모두에서 이 중 하나만 사용되었다. 그렇다면 다른 값은 어떻게 된 것인가? (힌트: 사용된 것은 DepositCharge.fee이다.)

다중 상속에서 속성을 찾을 때는 먼저 모든 기반 클래스를 가장 특수화된 것부터 가장 덜 특수화된 것 순서대로 나열한다. 다음으로 찾고자 하는 속성의 첫 번째 정의가 발견될 때까지 이 목록을 차례대로 검색한다. 앞의 예에서 EvilAccount는 Account에서 상속을 받았기 때문에 Account보다 더 특수화된 것이다. 비슷하게 MostEvilAccount에서 DepositCharge는 기반 클래스 목록에서 먼저 나타났기 때문에 WithdrawCharge보다 더 특수화된 것으로 간주된다. 어떤 클래스의 기반 클래스 순서는 `__mro__` 속성*을 출력해보면 알 수 있다. 다음 예를 보자.

```
>>> MostEvilAccount.__mro__
(<class '__main__.MostEvilAccount'>,
 <class '__main__.EvilAccount'>,
 <class '__main__.Account'>,
 <class '__main__.DepositCharge'>,
 <class '__main__.WithdrawCharge'>,
 <type 'object'>)
>>>
```

일반적으로 기반 클래스 순서는 이치에 맞는 규칙에 따라 정해진다. 즉, 파생 클래스는 항상 기반 클래스보다 먼저 검사된다. 그리고 클래스가 하나 이상의 부모를 가지고 있을 때는 항상 클래스 정의에 나와 있는 순서대로 검사된다. 기반 클래스의 정확한 순서는 상당히 복잡한 과정을 거쳐서 결정되며, 깊이 우선 검색(depth-first search)이나 너비 우선 검색(breadth-first search) 같은 간단한 알고리즘에 의해 결정되지 않는다. 이 순서는 ‘Dylan을 위한 단조 상위 클래스 선형화(A Monotonic Superclass Linearization for Dylan)’라는 제목의 논문(바렛 등이 OOPSLA’96 학회에서 발표함)에서 설명된 C3 선형화 알고리즘에 의해서 결정된다. 이 알고리즘에 따르면 파이썬에서는 특정 종류의 클래스 계층을 생성할 수 없고 그러한 계층을 생성하려고 시도할 경우 `TypeError` 예외가 발생한다. 다음 예를 보자.

```
class X(object): pass
class Y(X): pass
class Z(X,Y): pass    # TypeError 예외가 발생한다.
                      # 일관된 메서드 분석 순서를 정하지 못함.
```

* (옮긴이) mro는 method resolution order의 줄임말이다.

앞의 예에서 메서드 분석 알고리즘은 기반 클래스의 순서를 제대로 정할 수 없기 때문에 클래스 Z의 생성을 거부한다. 또 클래스 X는 상속 목록에서 클래스 Y보다 먼저 나타나기 때문에 먼저 검사된다. 그러나 클래스 Y는 X에서 상속을 받았기 때문에 더 특수화된 클래스이다. 따라서, X가 먼저 검사된다면 Y에서 특수화된 메서드를 찾는 일이 불가능해진다. 실전에서는 이 문제가 거의 발생하지 않는다. 이 문제가 발생한다는 것은 보통 프로그램의 설계에 심각한 문제가 있다는 것을 의미한다.

보통의 경우 프로그램에서 다중 상속을 사용하지 않는 것이 좋다. 하지만, 다중 상속은 혼합 클래스(mixin class)를 정의할 때 종종 사용된다. 혼합 클래스는 추가 기능을 제공하기 위한 목적으로 다른 클래스에 혼합될 메서드를 정의한다(거의 매크로처럼 작동한다). 일반적으로 혼합 클래스에서 정의되는 메서드는 또 다른 메서드가 있다고 가정하여 그러한 메서드들을 이용하여 작성된다. 앞의 예에서 DepositCharge와 WithdrawCharge 클래스는 혼합 클래스이다. 이 두 클래스에서는 이들을 기반 클래스로 사용할 클래스를 위해 deposit_fee() 같은 메서드를 정의한다. DepositCharge 클래스의 인스턴스를 직접 생성할 이유는 없다. 인스턴스를 생성하더라도 생성된 인스턴스는 아무런 유용한 일도 하지 못한다(즉, 메서드가 제대로 실행조차 되지 않을 것이다).

마지막으로, 앞의 예에서 fee 속성과 관련된 문제를 해결하려면 deposit_fee()와 withdraw_fee()에서 self를 사용하는 대신 클래스 이름을 직접 써주면 된다(예를 들어, DepositCharge.fee).

다형성 동적 바인딩과 오리 타입화

동적 바인딩(dynamic binding; 속과 관련해서 쓰일 때는 다형성polymorphism이라고도 한다)은 타입을 신경 쓰지 않고 인스턴스를 사용할 수 있게 해주는 메커니즘이다. 동적 바인딩은 전적으로 이전 절에서 상속과 관련된 속성 검색 과정을 통해서 이루어진다. obj.attr를 통해 속성에 접근하면 먼저 인스턴스 자체에서 attr를 검색하고, 다음으로 인스턴스의 클래스 정의에서, 그리고 나서 기반 클래스에서 검색한다. 가장 먼저 검색된 속성이 반환된다.

이렇게 동적으로 속성을 바인딩하는 과정은 객체 obj가 실제로 어떤 타입인지와는 아무런 상관이 없다. obj.name 같은 속성 검색은 name 속성을 갖고 있는 객체라면 어떤 객체에 대해서든지 제대로 작동한다. 이러한 작동 방식은 “오리처럼 생겼

고, 오리처럼 꽉꽉 울고, 오리처럼 걸으면, 그것은 오리이다”라는 격언에서 유래한 용어인 오리 타입화(duck typing)라고도 부른다.

파이썬에서는 종종 이러한 작동 방식에 기초해서 코드를 작성한다. 예를 들어, 기존 객체의 커스터마이즈된 버전을 원할 경우 기존 객체에서 상속을 할 수도 있고, 아니면 기존 객체와 관련이 없지만 기존 객체처럼 보이고 작동하는 완전히 새로운 객체를 생성할 수도 있다. 이 둘 중 후자의 접근법은 프로그램의 구성 요소들 간에 느슨한 결합이 이루어지기를 원할 때 사용한다. 예를 들어, 특정 메서드를 가지고 있는 객체라면 어떤 객체에든지 작동하는 코드를 작성하는 예를 생각해볼 수 있다. 대표적인 예로 표준 라이브러리에 있는 파일처럼 작동하는 객체들이 있다. 이 객체들은 파일처럼 작동하지만 내장 파일 객체로부터 상속을 받지 않았다.

정적 메서드와 클래스 메서드

클래스 정의에서 모든 함수는 인스턴스에 대해서 작동하는 것으로 가정한다. 이 때문에 첫 번째 매개변수로서 항상 self가 전달된다. 이 밖에도 클래스 정의에 볼 수 있는 흔히 사용되는 두 종류의 메서드가 더 있다.

정적 메서드(static method)는 클래스에 의해 정의되는 네임스페이스에 들어 있는 보통의 함수다. 정적 메서드는 인스턴스에 대해서 작동하지 않는다. 정적 메서드를 정의하려면 다음과 같이 @staticmethod 장식자를 사용하면 된다.

```
class Foo(object):
    @staticmethod
    def add(x,y):
        return x + y
```

정적 메서드를 호출하려면 그저 클래스 이름을 앞에 붙이기만 하면 된다. 추가 정보를 넘겨줄 필요는 없다. 다음 예를 보자.

```
x = Foo.add(3,4) # x = 7
```

정적 메서드는 다양한 방식으로 인스턴스를 생성하는 클래스를 작성할 때 흔히 사용된다. 클래스에서 __init__() 함수는 단 하나만 존재할 수 있기 때문에 다음에서 볼 수 있듯이 다른 생성 함수를 정적 메서드로서 정의하는 경우가 종종 있다.

```
import time
class Date(object):
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
```

```

        self.day = day
    @staticmethod
    def now( ):
        t = time.localtime( )
        return Date(t.tm_year, t.tm_mon, t.tm_day)
    @staticmethod
    def tomorrow( ):
        t = time.localtime(time.time( )+86400)
        return Date(t.tm_year, t.tm_mon, t.tm_day)

# 날짜 객체들을 몇 개 생성하는 예
a = Date(1967, 4, 9)
b = Date.now( )           # 정적 메서드 now( )를 호출한다.
c = Date.tomorrow( )     # 정적 메서드 tomorrow( )를 호출한다.

```

클래스 메서드(class method)는 클래스 자체를 객체로 보고 클래스 객체에 대해 작동하는 메서드를 말한다. 클래스 메서드는 @classmethod 장식자를 사용해서 정의한다. 인스턴스 메서드와 다른 점은 첫 번째 인수로서 관례적으로 cls라는 이름을 가진 클래스가 전달된다는 점이다. 다음은 한 예이다.

```

class Times(object):
    factor = 1
    @classmethod
    def mul(cls,x):
        return cls.factor*x

class TwoTimes(Times):
    factor = 2

x = TwoTimes.mul(4)      # Times.mul(TwoTimes, 4)를 호출한다. 결과는 8이다.

```

앞의 예에서 mul()에 클래스 TwoTimes가 객체로서 전달되었다. 앞의 예가 약간 난해하게 보일 수도 있을 것 같다. 클래스 메서드는 약간 복잡해 보이지만 실용적인 용도로 쓰인다. 예를 들어, 앞에서 정의한 Date 클래스로부터 상속을 받아서 다음과 같이 약간 커스터마이즈를 하였다고 하자.

```

class EuroDate(Date):
    # 유럽의 날짜 표현 방식대로 문자열 변환을 수행하도록 수정
    def __str__(self):
        return "%02d/%02d/%4d" % (self.day, self.month, self.year)

```

이 클래스는 Date에서 상속받았기 때문에 Date의 모든 기능을 갖게 된다. 그러나 now()와 tomorrow() 메서드는 의도한 대로 작동하지 않는다. 예를 들어, 누군가가 EuroDate.now()를 호출하면 EuroDate 객체가 아닌 Date 객체가 반환된다. 다음과 같이 클래스 메서드를 사용함으로써 이 문제를 해결할 수 있다.

```
class Date(object):
```

```

...
@classmethod
def now(cls):
    t = time.localtime( )
    # 적절한 타입의 객체를 생성한다.
    return cls(t.tm_year, t.tm_mon, t.tm_day)

class EuroDate(Date):
    ...

a = Date.now( )          # Date.now(Date)가 호출되어서 Date 객체가 반환된다.

b = EuroDate.now( )     # Date.now(EuroDate)가 호출되어서 EuroDate 객체가 반
환된다.

```

정적 메서드와 클래스 메서드를 사용할 때 파이썬에서는 이 메서드들을 인스턴스 메서드와 별개의 네임스페이스로 관리하지 않는다. 이 때문에 다음과 같이 정적 메서드와 클래스 메서드를 인스턴스에 대해서 호출할 수 있다.

```

d = Date(1967,4,9)
b = d.now( )          # Date.now(Date)를 호출한다.

```

이것이 가능하다는 점은 d.now()가 인스턴스 d와는 아무런 관련이 없다는 점 때문에 혼동을 가져올 수 있다. 이 부분은 파이썬의 객체 시스템이 스몰톡(Smalltalk)이나 루비(Ruby) 같은 다른 객체지향 언어와 다른 점 중 하나이다. 다른 객체지향 언어에서는 클래스 메서드와 인스턴스 메서드가 엄격하게 분리되어 있다.

프로퍼티

일반적으로 인스턴스나 클래스의 속성에 접근하면 저장된 값이 반환된다. 프로퍼티(property)는 접근되는 순간 값이 계산되는 특수한 속성이다. 다음 예를 보자.

```

class Circle(object):
    def __init__(self, radius):
        self.radius = radius
    # 원의 몇 가지 프로퍼티들
    @property
    def area(self):
        return math.pi * self.radius**2
    @property
    def perimeter(self):
        return 2 * math.pi * self.radius

```

Circle 객체는 다음과 같이 작동한다.

```

>>> c = Circle(4.0)
>>> c.radius

```

```

4.0
>>> c.area
50.26548245743669
>>> c.perimeter
25.132741228718345
>>> c.area = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>

```

앞의 예에서 Circle의 인스턴스는 인스턴스 변수 c.radius를 갖는다. c.area와 c.perimeter는 c.radius의 값으로 간단히 계산된다. 장식자 @property는 바로 다음에 나오는 메서드를 보통 메서드를 호출할 때 붙여야 하는 괄호 () 없이 단순 속성인 것처럼 접근할 수 있게 한다. 객체를 사용하는 입장에서는 속성을 재정의하려고 시도할 경우 에러 메시지가 발생한다(앞의 예에서는 AttributeError 예외가 발생)는 것 말고는 속성 값이 계산되는지 여부를 알 수 없다.

이렇게 프로퍼티를 사용하는 것을 균일 접근 원칙(uniform access principle)이라고도 한다. 기본적으로 클래스를 정의할 때는 프로그래밍 인터페이스가 가능하면 균일할수록 좋다. 프로퍼티를 사용하지 않는다면 객체의 어떤 속성은 c.radius처럼 단순 속성으로서 접근해야 하고 다른 속성은 c.area()처럼 메서드로서 접근해야 한다. 언제 추가로 ()를 붙여주어야 하는지를 기억하는 일은 번거롭다. 프로퍼티를 사용하면 이런 문제를 해결할 수 있다.

파이썬 프로그래머들은 메서드도 프로퍼티의 한 종류로서 다루어진다는 것을 모르는 경우가 종종 있다. 다음 클래스를 한번 보자.

```

class Foo(object):
    def __init__(self, name):
        self.name = name
    def spam(self, x):
        print("%s, %s" % (self.name, x))

```

사용자가 f = Foo("Guido")와 같이 인스턴스를 하나 생성하고 f.spam에 접근하면 원래의 함수 객체인 spam이 반환되지 않는다. 대신 () 연산자로 호출할 경우 실행되는 메서드 호출을 나타내는 뮤인 메서드(bound method)라고 불리는 객체를 얻게 된다. 뮤인 메서드는 매개변수 self가 이미 채워졌고 ()를 사용해서 호출될 때 추가 인수가 제공되기를 기대하는 부분적으로 평가된 함수이다. 이 뮤인 메서드를 생성하는 작업은 배후에서 실행되는 프로퍼티 함수를 통해 조용히 처리된다. @staticmethod나 @classmethod로 정적 메서드나 클래스 메서드를 정의할 때에

도 실제로는 이들 메서드에 대한 접근을 처리하는 프로퍼티 함수를 정의하는 것과 마찬가지이다. 예를 들어, `@staticmethod`는 간단히 메서드 함수를 특별히 다른 것으로 감싸거나 추가로 처리하지 않고 있는 그대로 반환한다.

프로퍼티는 속성을 설정하거나 삭제하는 연산을 가로챌 수도 있다. 이를 위해서는 프로퍼티에 설정(setter) 메서드나 삭제(deleter) 메서드를 추가하면 된다. 다음 예를 보자.

```
class Foo(object):
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError("Must be a string!")
        self.__name = value
    @name.deleter
    def name(self):
        raise TypeError("Can't delete name")

f = Foo("Guido")
n = f.name          # f.name( )를 호출한다. - 초기(get) 함수
f.name = "Monty"   # 설정 함수인 name(f, "Monty")을 호출한다.
f.name = 45         # 설정 함수인 name(f, 45)을 호출한다. -> TypeError
del f.name         # 삭제 함수인 name(f)을 호출한다. -> TypeError
```

앞의 예에서 속성 `name`은 `@property` 장식자와 관련 메서드를 사용해서 먼저 읽기 전용 프로퍼티로 정의된다. 이어서 나오는 `@name.setter`와 `@name.deleter` 장식자는 설정 및 삭제 연산을 위한 추가 메서드를 프로퍼티에 연결시킨다. 추가 메서드는 원본 프로퍼티 메서드의 이름과 동일해야 한다. 앞의 예에서 `name`의 실제 값은 `__name` 속성에 저장된다. 이 속성의 이름을 정하는 데에는 특별한 규칙은 없지만 프로퍼티와 구별하기 위해 프로퍼티의 이름과는 달라야 한다.

오래된 코드를 보면, `property(getf=None, setf=None, delf=None, doc=None)` 함수와 각 연산에 대응되는 고유한 이름을 갖는 메서드들로 프로퍼티를 정의하는 것을 종종 볼 수 있다. 다음 예를 보자.

```
class Foo(object):
    def getname(self):
        return self.__name
    def setname(self, value):
        if not isinstance(value, str):
            raise TypeError("Must be a string!")
```

```

        self.__name = value
    def delname(self):
        raise TypeError("Can't delete name")
    name = property(getname, setname, delname)

```

이 오래된 방식도 여전히 지원되기는 하지만, 장식자를 사용하는 방식이 훨씬 더 깔끔하다. 예를 들어, 장식자를 사용하면 get, set, delete 메서드가 외부에 노출되지 않는다.

기술자

프로퍼티를 사용하면 속성에 대한 접근을 사용자 정의 함수 get, set, delete로 제어 할 수 있다. 이러한 식의 속성 접근 제어는 기술자 객체(descriptor object)를 사용해 서 더욱 일반화할 수 있다. 기술자는 간단히 속성의 값을 나타내는 객체이다. 기술자에서 특수 메서드인 __get__(), __set__(), __delete__() 중 하나 이상의 메서드를 구현함으로써 속성 접근 메커니즘을 가로채 관련 연산들을 커스터마이즈할 수 있다. 다음 예를 보자.

```

class TypedProperty(object):
    def __init__(self, name, type, default=None):
        self.name = "_" + name
        self.type = type
        self.default = type() if default is None else default
    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default) if instance else self.default
    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance, self.name, value)
    def __delete__(self, instance):
        raise AttributeError("Can't delete attribute")

class Foo(object):
    name = TypedProperty("name", str)
    num = TypedProperty("num", int, 42)

```

앞의 예에서 TypedProperty 클래스는 속성에 값이 할당될 때 타입 검사를 수행하고 속성을 삭제하려는 시도가 있을 경우 에러를 발생시키는 기술자를 정의한다. 다음 예를 보자.

```

f = Foo()
a = f.name          # 암묵적으로 Foo.name.__get__(f, Foo)를 호출한다.
f.name = "Guido"   # Foo.name.__set__(f, "Guido")를 호출한다.
del f.name         # Foo.name.__delete__(f)를 호출한다.

```

클래스 수준에서만 기술자의 인스턴스를 생성할 수 있다. `__init__()`나 기타 메서드 안에서 인스턴스별 기술자를 생성하는 일은 적법하지 않다. 또한 인스턴스에 있는 속성보다 기술자를 담는 클래스 속성 이름이 먼저 사용된다. 이러한 이유 때문에 앞의 예에서 기술자 객체에서 `name` 매개변수를 받은 후 이를 앞에 밑줄을 하거나 추가하였다. 기술자 객체에서 인스턴스에 값을 저장하려면 기술자 자체에서 사용되는 것과는 다른 이름을 사용해야 한다.

데이터 캡슐화와 개인 속성

파이썬에서 기본적으로 클래스의 모든 속성과 메서드는 공개되어 있다. 즉, 모두 제한 없이 접근할 수 있다. 또한 일반 클래스에서 정의된 모든 것이 상속되며 파생 클래스에서 접근할 수 있다. 객체의 내부 구현이 밖으로 다 드러나고 파생 클래스와 일반 클래스 사이에 네임스페이스 충돌이 발생할 수 있기 때문에 객체지향 응용 프로그램을 작성할 때에는 이 점이 달갑지 않을 때가 종종 있다.

이 문제를 해결하기 위해서 파이썬에서는 클래스에서 `__Foo` 같은 이중 밑줄로 시작하는 모든 이름을 자동으로 `_클래스이름__Foo`의 형태를 가지는 이름으로 변환한다(이것을 이름 변형(name mangling)이라고 한다). 이렇게 하면 파생 클래스에서 사용하는 개인(private) 이름과 일반 클래스에서 사용하는 개인 이름이 충돌하지 않기 때문에 사실상 클래스에서 개인 속성과 메서드를 갖게 된다. 다음 예를 보자.

```
class A(object):
    def __init__(self):
        self.__X = 3    # self._A__X로 변형된다.
    def __spam(self):  # _A__spam()로 변형된다.
        pass
    def bar(self):
        self.__spam()  # 오직 A.__spam()만을 호출한다.

class B(A):
    def __init__(self):
        A.__init__(self)
        self.__X = 37  # self._B__X로 변형된다.
    def __spam(self):  # _B__spam()로 변형된다.
        pass
```

이러한 방식을 통해 데이터가 제대로 감추어진 것처럼 보이지만, 파이썬에서 클래스의 개인 속성에 접근하는 것을 근본적으로 막을 방법은 없다. 특히, 클래스의 이름과 해당 개인 속성을 알고 있으면 변형된 이름을 사용해서 여전히 접근할 수 있다.

개인 속성을 외부에 덜 노출하는 방법으로 `__dir__()` 메서드를 재정의하여 `dir()` 함수로 객체를 검사할 때 반환되는 이름 목록을 바꾸어주는 방법도 있다.

이렇게 이름을 변형할 때 추가적인 작업이 수행되는 것처럼 보이지만 실제로는 클래스가 정의될 때 한 번의 처리만 거치게 된다. 메서드를 실행하는 도중에 추가적인 작업이 필요한 것이 아니기 때문에 프로그램을 실행할 때 추가로 부하가 발생하지는 않는다. 또한 `getattr()`, `hasattr()`, `setattr()`, `delattr()` 같이 속성 이름을 문자열로 받는 함수에서는 이름 변형이 수행되지 않는다. 즉, 이 함수들에 대해서는 해당 속성에 접근하려면 '`_클래스이름__이름`'처럼 변형된 이름을 직접 직접 써주어야 한다.

변경 가능한 속성을 프로퍼티를 통해서 정의하였을 때는 해당 속성을 개인 속성으로 만드는 것이 좋다. 이렇게 하면 사용자가 내부 인스턴스에 직접 접근하는 것을 막고 프로퍼티를 사용하게 유도할 수 있다(애초에 프로퍼티를 사용한다는 자체가 속성에 직접 접근하는 것을 원하지 않는다는 의미일 것이다). 앞 절에서 이와 관련된 예를 살펴보았다.

상위 클래스에서 어떤 메서드에 개인 이름을 주면 파생 클래스에서 그 메서드를 재정의하는 것 때문에 구현이 변경되는 일을 막을 수 있다. 예를 들어, 앞의 예에서 `A.bar()` 메서드는 `self`의 타입에 상관없이, 그리고 파생 클래스에 다른 `__spam()` 메서드가 있는지 여부와 상관없이 항상 `A.__spam()`을 호출한다.

클래스에서 개인 속성의 이름을 짓는 방식과 모듈에서 개인 정의에 이름을 부여하는 방식을 혼동하지 않기 바란다. 흔히 하는 실수로 클래스에서 속성 값을 감추기 위해 이름 앞에 단일 밑줄을 붙이는 경우가 있다(예를 들어, `_name`). 모듈에서는 이름을 이렇게 지으면 `from module import *` 문에 의해서 해당 이름이 임포트되는 것이 방지된다. 클래스에서는 이름을 이렇게 짓는다고 해서 속성이 감추어지는 것도 아니고 누군가가 클래스를 상속받아서 동일한 이름을 가지는 속성이나 메서드를 정의할 경우 발생하는 이름 충돌을 막을 수도 없다.

객체 메모리 관리

클래스가 하나 정의되면, 그 클래스는 새로운 인스턴스를 생성하는 공장 역할을 수행한다.

```
class Circle(object):
```

```

def __init__(self, radius):
    self.radius = radius

# Circle 인스턴스를 몇 개 생성한다.
c = Circle(4.0)
d = Circle(5.0)

```

인스턴스의 생성은 새로운 인스턴스를 생성하는 특수 메서드인 `__new__()`와 생성된 인스턴스를 초기화하는 `__init__()` 메서드를 통해 두 단계를 거쳐서 수행된다. 예를 들어, 연산 `c = Circle(4.0)`은 다음 단계를 수행하는 것과 같다.

```

c = Circle.__new__(Circle, 4.0)
if isinstance(c, Circle):
    Circle.__init__(c, 4.0)

```

클래스의 `__new__()` 메서드를 사용자 코드에서 정의하는 일은 드물다. 이 메서드를 정의할 일이 있을 때는 일반적으로 `__new__(cls, *args, **kwargs)`의 형태로 정의한다. 여기서 `args`와 `kwargs`는 `__init__()`에 전달되는 인수와 동일하다. `__new__()`는 언제나 첫 번째 인수로 클래스 객체를 받는 클래스 메서드로 정의해야 한다. `__new__()`가 인스턴스를 생성하지만 알아서 `__init__()`를 호출하지는 않는다.

클래스에서 `__new__()`가 정의되어 있다는 것은 그 클래스가 둘 중 하나를 하고 있는 것이다. 첫째로 변경이 불가능한 기반 클래스로부터 상속을 받는 경우가 있다. 이와 관련된 예로 정수, 문자열, 튜플 같은 변경이 불가능한 내장 타입으로부터 상속을 받는 객체를 정의하는 경우가 있다. 그 이유는 인스턴스가 생성되기 전에 실행되는 메서드는 `__new__()`가 유일하고, 따라서 인스턴스의 값을 변경할 수 있는 유일한 곳이기 때문이다(`__init__()` 안이라면 이미 늦은 것이다). 다음 예를 보자.

```

class Upperstr(str):
    def __new__(cls, value=""):
        return str.__new__(cls, value.upper())

```

```
u = Upperstr("hello") # u = "HELLO"
```

`__new__()`의 다른 주된 용도는 메타클래스를 정의하기 위함이다. 여기에 관해서는 이 장 마지막에 설명한다.

일단 생성되면 인스턴스는 참조 횟수 세기(reference counting)를 통해 관리된다. 참조 횟수가 영이 되면 인스턴스는 즉시 파괴된다. 인스턴스가 파괴되려고 할 때 인터프리터는 먼저 인스턴스에 `__del__()` 메서드가 있는지를 살펴보고 있으면 호

출한다. 실제로 `__del__()`를 정의할 필요가 있는 경우는 드물다. 한 가지 예외로는 객체를 파괴할 때 파일을 닫는다거나 네트워크 연결을 끊는다거나 기타 시스템 자원을 해제하는 등 청소 작업을 해야 할 때이다. 이런 경우라도 깔끔한 종료 과정을 수행하기 위해서 `__del__()`에 의존하는 일은 위험하다. 인터프리터가 종료될 때 이 메서드가 호출되리라는 보장이 없기 때문이다. `close()` 같은 메서드를 두어서 프로그램에서 직접 종료 작업을 수행할 수 있도록 하는 것이 더 낫다.

종종 프로그램에서는 `del`문을 사용해서 객체에 대한 참조를 삭제한다. 이로 인해 객체의 참조 횟수가 영에 도달하면 `__del__()` 메서드가 호출된다. 그러나 일반적으로 `del`문이 `__del__()`을 직접 호출하지는 않는다.

객체 파괴와 관련해서 한 가지 조심해야 할 점은 `__del__()`을 정의할 경우 파이썬의 순환 참조 쓰래기 수집기에 의해서 객체가 처리되지 않을 수도 있다는 점이다 (불필요하게 `__del__()`을 정의하지 않는 것이 좋은 이유이기도 하다). 자동 쓰래기 수집 기능을 제공하지 않는 언어(C++ 같은)를 사용해온 프로그래머들은 불필요하게 `__del__()`을 정의하는 프로그래밍 스타일을 파이썬에서도 사용하지 않도록 주의해야 한다. `__del__()`을 정의한다고 해서 쓰래기 수집 기능이 망가지는 경우는 드물지만, 부모 자식 관계나 그래프가 관여하는 특정한 종류의 프로그래밍 패턴에서는 이것이 문제가 될 수 있다. 예를 들어, 다음과 같이 ‘관찰자 패턴(Observer Pattern)’의 일종을 구현하고 있는 객체가 있다고 하자.

```
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        self.observers = set()
    def __del__(self):
        for ob in self.observers:
            ob.close()
        del self.observers
    def register(self, observer):
        self.observers.add(observer)
    def unregister(self, observer):
        self.observers.remove(observer)
    def notify(self):
        for ob in self.observers:
            ob.update()
    def withdraw(self, amt):
        self.balance -= amt
        self.notify()

class AccountObserver(object):
    def __init__(self, theaccount):
```

```

        self.theaccount = theaccount
        theaccount.register(self)
    def __del__(self):
        self.theaccount.unregister(self)
        del self.theaccount
    def update(self):
        print("Balance is %.2f" % self.theaccount.balance)
    def close(self):
        print("Account no longer in use")

# 예를 위한 설정
a = Account('Dave', 1000.00)
a_ob = AccountObserver(a)

```

이 코드에서 Account 클래스는 잔고에 변화가 있을 때마다 알려서 Account Observer 객체들이 Account 인스턴스를 감시할 수 있게 하고 있다. 이를 위해서 각 Account는 자신의 관찰자를 추적하고 각 AccountObserver는 계좌로의 역참조를 유지한다. 각 클래스는 청소 기능(등록 해제 등)을 제공하기 위해서 `__del__()`을 정의한다. 그러나 청소 기능은 제대로 작동하지 않는다. 이 클래스들은 서로 순환 참조를 생성하여 참조 횟수가 절대로 0으로 떨어지지 않는다. 이 때문에 아무런 청소 작업이 수행되지 않는다. 그뿐만 아니라 쓰레기 수집기(gc 모듈)도 이들을 청소하지 않으므로 영구적인 메모리 누수가 발생한다.

앞의 예에서 발생한 문제를 해결하는 방법 중 하나로 `weakref` 모듈을 사용해서 한 클래스가 다른 클래스에 대한 약한 참조를 가지도록 만드는 방법이 있다. 약한 참조(weak reference)는 참조 횟수를 증가시키지 않고 객체에 대한 참조를 생성한다. 약한 참조를 사용하려면 참조되는 객체가 아직 존재하는지 여부를 검사하는 부분을 추가해야 한다. 다음은 약한 참조를 사용하도록 수정한 관찰자 클래스이다.

```

import weakref
class AccountObserver(object):
    def __init__(self, theaccount):
        self.accountref = weakref.ref(theaccount) # 약한 참조 생성한다.
        theaccount.register(self)
    def __del__(self):
        acc = self.accountref() # 계좌를 가져온다.
        if acc:                 # 여전히 존재하면 등록을 해제한다.
            acc.unregister(self)
    def update(self):
        print("Balance is %.2f" % self.accountref().balance)
    def close(self):
        print("Account no longer in use")

# 예를 위한 설정

```

```
a = Account('Dave', 1000.00)
a.ob = AccountObserver(a)
```

앞의 예에서 약한 참조인 accountref가 생성되었다. 약한 참조를 함수처럼 호출하면 그 아래에 있는 Account에 접근할 수 있다. 이 호출은 Account를 반환하거나 계좌가 더 존재하지 않으면 None을 반환한다. 이제 더 이상 순환 참조가 일어나지 않는다. Account 객체가 파괴될 때 `__del__()` 메서드가 실행되고 관찰자들은 통지를 받는다. gc 모듈도 제대로 작동한다. weakref 모듈에 대한 더 자세한 정보는 13장에서 찾을 수 있다.

객체 표현과 속성 바인딩

내부적으로 인스턴스는 `__dict__` 속성으로 접근할 수 있는 사전을 사용해서 구현된다. 이 사전은 각 인스턴스에 고유한 데이터를 저장한다. 다음 예를 보자.

```
>>> a = Account('Guido', 1100.0)
>>> a.__dict__
{'balance': 1100.0, 'name': 'Guido'}
```

다음과 같이 언제든지 인스턴스에 새로운 속성을 추가할 수 있다.

```
a.number = 123456 # 속성 'number'를 a.__dict__에 추가한다.
```

인스턴스에 가한 변화는 항상 내부 `__dict__` 속성에 반영된다. 마찬가지로 `__dict__`에 직접 변화를 가해도 이 변화가 속성에 반영된다.

인스턴스는 특수한 속성 `__class__`로 자신의 클래스에 다시 연결된다. 클래스 자체도 자신의 `__dict__` 속성을 통해 접근할 수 있는 사전 위에 얹은 층을 제공하고 있을 뿐이다. 클래스의 사전에는 메서드도 들어 있다. 다음 예를 보자.

```
>>> a.__class__
<class '__main__.Account'>
>>> Account.__dict__.keys()
['__dict__', '__module__', 'inquiry', 'deposit', 'withdraw',
 '__del__', 'num_accounts', '__weakref__', '__doc__', '__init__']
>>>
```

마지막으로, 클래스는 기반 클래스들을 담는 튜플인 특수한 속성 `__bases__`를 통해 자신의 기반 클래스에 연결된다. 이러한 내부 구조는 객체의 속성을 얻고 설정하고 삭제하는 모든 연산의 기반이 된다.

`obj.name = value`로 속성을 설정할 때마다 특수한 메서드인 `obj.__setattr__`

(“name”, value)가 호출된다. `del obj.name`으로 속성을 삭제하면 특수한 메서드인 `obj.__delattr__("name")`가 호출된다. 이 메서드들은 요청된 속성이 프로퍼티나 기술자가 아닌 한, `obj`의 내부 `__dict__`에 있는 값을 수정하거나 제거하는 기본 동작을 수행한다. 속성이 프로퍼티나 기술자인 경우에는 설정과 삭제 연산이 프로퍼티에 연결된 설정과 삭제 함수에 의해서 수행된다.

`obj.name` 같은 속성 검색에는 특수한 메서드인 `obj.__getattribute__("name")`가 사용된다. 이 메서드는 보통 먼저 프로퍼티를 찾아보고, 내부 `__dict__` 속성을 살펴보고, 클래스 사전을 살펴보고, 마지막으로 기반 클래스들을 살펴보는, 속성을 찾기 위한 검색을 수행한다. 이 검색이 실패하면 마지막 시도로 클래스의 `__getattr__()` 메서드(정의된 경우)를 호출해서 속성을 찾는다. 이것마저 실패하면 `AttributeError` 예외가 발생한다.

사용자 정의 클래스에서는 속성 검색 함수를 원하는 대로 구현할 수 있다. 다음 예를 보자.

```
class Circle(object):
    def __init__(self, radius):
        self.radius = radius
    def __getattr__(self, name):
        if name == 'area':
            return math.pi * self.radius**2
        elif name == 'perimeter':
            return 2 * math.pi * self.radius
        else:
            return object.__getattribute__(self, name)
    def __setattr__(self, name, value):
        if name in ['area', 'perimeter']:
            raise TypeError("%s is readonly" % name)
        object.__setattr__(self, name, value)
```

앞에서 언급한 메서드들을 재구현하는 클래스에서는 최종 작업을 수행할 때 `object`에 있는 기본 구현을 사용하는 것이 좋다. 그 이유는 기본 구현에서 기술자나 프로퍼티 같은 고급 기능을 알아서 처리해주기 때문이다.

일반적으로 클래스에서 속성 접근 연산자들을 구현하는 일은 상대적으로 드물다. 그렇게 해야 하는 경우 중 하나로 기존 객체에 대한 범용 래퍼나 대리자(proxy)를 작성하는 경우가 있다. `__getattr__()`, `__setattr__()`, `__delattr__()`를 재정의함으로써 대리자에서 속성 접근을 가로채 이 연산을 다른 객체에 투명하게 전달할 수 있다.

__slots__

클래스에서 `__slots__`이라는 특수한 변수를 정의하면 속성 이름에 제약을 가할 수 있다. 다음 예를 보자.

```
class Account(object):
    __slots__ = ('name','balance')
    ...
```

`__slots__`를 정의하면 인스턴스에 할당할 수 있는 속성 이름이 지정된 것으로 제한된다. 그 밖의 속성 이름이 사용되면 `AttributeError` 예외가 발생한다. 이렇게 하면 누군가가 기준 인스턴스에 새로운 속성을 추가하는 것을 막고 속성 이름을 잘못 써서 값을 할당하는 경우 발생하는 문제를 해결할 수 있다.

실제로 `__slots__`이 이러한 보호 기능을 구현하기 위해서 사용되는 경우는 거의 없다. `__slots__`은 메모리와 실행 속도의 성능을 최적화하기 위한 용도로 사용된다. `__slots__`을 사용하는 클래스의 인스턴스는 더 이상 인스턴스 데이터를 저장하는 데 사전을 사용하지 않는다. 그 대신 배열에 기초한 훨씬 간결한 데이터 구조가 사용된다. 많은 수의 객체를 사용하는 프로그램에서 `__slots__`을 사용하면 메모리 사용과 실행 시간을 크게 줄일 수 있다.

`__slots__`을 사용할 때는 상속 기능과의 복잡한 상호 작용을 염두에 두어야 한다. 클래스가 `__slots__`을 사용하는 기반 클래스로부터 상속을 받으면 `__slots__`이 제공하는 이점을 얻기 위해서 자신 또한 `__slots__`을 정의할 필요가 있다(새로운 속성을 전혀 추가하지 않더라도). 이렇게 하는 것을 잊어버리면 파생 클래스가 더 느리게 작동하고 기반 클래스에서 `__slots__`을 전혀 사용하지 않는 경우보다도 더 메모리를 많이 사용하게 된다!

`__slots__`을 사용할 때는 인스턴스에 내부 `__dict__` 속성이 있다는 것을 염두에 두고 작성한 코드가 제대로 작동하지 않을 수 있다. 사용자가 작성한 코드에서 그런 일이 발생하는 일은 드물지만, 다른 객체를 지원하기 위해서 작성된 유ти리티 라이브러리라든지 디버깅, 직렬화나 기타 연산을 위해서 `__dict__`를 들여다보도록 작성된 도구에서는 문제가 발생할 수 있다.

마지막으로, `__slots__`이 있다고 해서 클래스에서 재정의된 `__getattribute__()`, `__getattr__()`와 `__setattr__()` 메서드 호출이 영향을 받지는 않는다. 그렇지만 이 메서드들의 기본 구현은 `__slots__`을 고려한다. 메서드나 프로퍼티는 클래스에 저장되는 것이므로 이들의 이름을 `__slots__`에 추가할 필요는 없다.

연산자 오버로딩

사용자 정의 객체는 3장에서 설명한 특수한 메서드들을 구현함으로써 파이썬의 모든 내장 연산자와 함께 사용될 수 있다. 예를 들어, 파이썬에 새로운 종류의 수 객체를 추가하고 싶다면 표준 수학 연산자와 함께 사용될 수 있도록 `__add__()` 같은 특수한 메서드를 정의한 클래스를 정의하면 된다.

다음은 몇 가지 표준 수학 연산자로 복소수를 나타내는 클래스를 정의하는 예를 보여준다.

Note

파이썬에는 이미 복소수 타입이 있기 때문에 여기에 나온 클래스는 단지 설명을 위해서 만든 것이다.

```
class Complex(object):
    def __init__(self, real, imag=0):
        self.real = float(real)
        self.imag = float(imag)
    def __repr__(self):
        return "Complex(%s,%s)" % (self.real, self.imag)
    def __str__(self):
        return "(%g+%gj)" % (self.real, self.imag)
    # self + other
    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)
    # self - other
    def __sub__(self, other):
        return Complex(self.real - other.real, self.imag - other.imag)
```

앞의 예에서 `__repr__()` 메서드는 평가될 경우 객체를 다시 생성하는 문자열을 생성한다(즉, “`Complex(real, imag)`”). 사용자 정의 객체는 적용 가능할 경우 이 관례를 따라야 한다. 반면에, `__str__()` 메서드는 깔끔한 출력을 위해서 사용된다(`print`문에 의해서 출력될 문자열이다).

`__add__()`나 `__sub__()` 같은 연산자는 수학 연산을 구현한다. 여기서 다소 까다로운 부분은 피연산자의 순서와 타입 강제 변환에 관한 것이다. 앞의 예에서 `__add__()`와 `__sub__()` 연산자는 복소수가 연산자의 왼쪽에 나올 때만 적용된다. 연산자의 오른쪽에 나온다거나 제일 왼쪽 피연산자가 `Complex`가 아닐 경우에는 제대로 작동하지 않는다. 다음 예를 보자.

```
>>> c = Complex(2,3)
>>> c + 4.0
Complex(6.0,3.0)
>>> 4.0 + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Complex'
>>>
```

여기서 연산 $c + 4.0$ 은 어느 정도 우연에 의해서 제대로 실행되었다. 파이썬의 모든 내장 숫자는 이미 `.real`과 `.imag` 속성을 갖고 있기 때문에 이 속성들이 계산에서 사용된 것이다. `other` 객체가 이 속성들을 갖고 있지 않으면 코드는 제대로 돌아가지 않을 것이다. `Complex`의 구현이 이러한 속성들이 없는 객체와도 작동할 수 있게 하려면 필요한 정보(`other` 객체의 타입에 따라 다를 수 있는)를 추출하는 변환 코드를 더 추가해야 한다.

$4.0 + c$ 는 내장 부동 소수점 타입이 `Complex` 클래스에 대해서 전혀 모르기 때문에 제대로 작동하지 않는다. 이 문제를 해결하려면 다음과 같이 역퍼연산자 메서드를 `Complex`에 추가하면 된다.

```
class Complex(object):
    ...
    def __radd__(self, other):
        return Complex(other.real + self.real, other.imag + self.imag)
    def __rsub__(self, other):
        return Complex(other.real - self.real, other.imag - self.imag)
    ...
    ...
```

이 메서드들은 예외적인 경우에 사용된다. 즉, 연산 $4.0 + c$ 가 실패하면 파이썬은 `TypeError`를 발생시키기 전에 `c.__radd__(4.0)`을 먼저 실행해본다.

오래된 버전의 파이썬에서는 타입이 섞여 있는 연산에서 타입을 강제로 변환하기 위해서 다양한 시도를 하였다. 예를 들어, 레거시 파이썬 코드에서 `__coerce__()` 메서드가 구현된 것을 볼 수 있을 것이다. 파이썬 2.6과 3에서 이 메서드는 더 이상 사용되지 않는다. 그리고 `__int__()`, `__float__()`, `__complex__()` 같은 특수한 메서드를 보고 혼동하지 않도록 한다. 이 메서드들은 `int(x)`나 `float(x)` 같이 명시적인 타입 변환을 위해서 호출된다. 하지만, 혼합 타입 산술 연산에서 타입 변환을 수행하는 데 암묵적으로 호출되는 일은 절대 없다. 혼합 타입에 대해서도 작동하는 연산자를 정의하는 클래스를 작성할 때는 각 연산자 구현 코드에서 직접 타입을 변환해주어야 한다.

타입과 클래스 멤버 검사

클래스의 인스턴스를 생성하면 인스턴스의 타입은 그 클래스가 된다. 어떤 클래스에 속하는지를 검사하려면 내장 함수인 `isinstance(obj, cname)`를 사용하면 된다. 이 함수는 객체 `obj`가 클래스 `cname`이나 `cname`에서 파생된 클래스에 속하면 `True`를 반환한다. 다음 예를 보자.

```
class A(object): pass
class B(A): pass
class C(object): pass

a = A( )           # 'A'의 인스턴스
b = B( )           # 'B'의 인스턴스
c = C( )           # 'C'의 인스턴스

type(a)            # 클래스 객체 A를 반환한다.
isinstance(a,A)    # True를 반환한다.
isinstance(b,A)    # True를 반환한다. B는 A로부터 파생되었다.
isinstance(b,C)    # False를 반환한다. B는 C로부터 파생되지 않았다.
```

비슷하게 내장 함수인 `issubclass(A, B)`는 클래스 `A`가 클래스 `B`의 하위 클래스일 경우 `True`를 반환한다. 다음 예를 보자.

```
issubclass(B,A)  # True를 반환한다.
issubclass(C,A)  # False를 반환한다.
```

객체의 타입을 검사할 때는 프로그래머들이 종종 상속을 사용하지 않고 단순히 다른 객체의 작동 방식을 흉내 내기만 하는 객체를 정의하는 경우가 있다는 점을 염두에 두어야 한다. 다음 두 클래스를 살펴보자.

```
class Foo(object):
    def spam(self,a,b):
        pass

class FooProxy(object):
    def __init__(self,f):
        self.f = f
    def spam(self,a,b):
        return self.f.spam(a,b)
```

이 예에서 `FooProxy`는 기능적으로는 `Foo`와 동일하다. `FooProxy`는 동일한 메서드를 구현하고 있고 심지어 내부적으로 `Foo`를 사용하고 있기까지 하다. 그런데도 타입 시스템상에서 `FooProxy`는 `Foo`와 다르다. 다음 예를 보자.

```
f = Foo( )          # Foo를 생성한다.
g = FooProxy(f)     # FooProxy를 생성한다.
isinstance(g, Foo)   # False를 반환한다.
```

프로그램에서 `isinstance()`를 사용해서 `Foo`인지 여부를 직접 검사하는 경우라면 확실히 `FooProxy` 객체는 사용될 수 없다. 하지만, 이 정도의 엄격함은 종종 여러분이 원하는 바가 아니다. 대신 어떤 객체가 `Foo`와 동일한 인터페이스를 가지고 있다면 간단히 `Foo`처럼 사용될 수 있다고 생각하는 것이 더 자연스럽다. 이를 위해서 비슷한 객체들을 그룹 지어 타입 검사를 적절히 수행하기 위한 목적으로 `isinstance()`와 `issubclass()`의 작동 방식을 재정의하는 것이 가능하다. 다음 예를 보자.

```
class IClass(object):
    def __init__(self):
        self.implementors = set()
    def register(self,C):
        self.implementors.add(C)
    def __instancecheck__(self,x):
        return self.__subclasscheck__(type(x))
    def __subclasscheck__(self,sub):
        return any(c in self.implementors for c in sub.mro())
# 앞에서 정의된 클래스를 사용한다.
IFoo = IClass()
IFoo.register(Foo)
IFoo.register(FooProxy)
```

이 예에서 클래스 `IClass`는 단순히 집합 자료 구조를 사용해서 다른 클래스들을 그룹 짓는 객체를 생성한다. `register()` 메서드는 집합에 새로운 클래스를 추가한다. 특수한 메서드 `__instancecheck__()`는 누군가가 `isinstance(x, IClass)` 연산을 수행할 때 호출된다. 특수한 메서드 `__subclasscheck__()`는 `issubclass(C, IClass)`가 호출될 때 호출된다.

`IFoo` 객체와 등록된 구현 객체들을 사용해서 이제 다음과 같이 타입 검사를 수행할 수 있다.

```
f = Foo()          # Foo 객체를 생성한다.
g = FooProxy(f)   # FooProxy 객체를 생성한다.
isinstance(f, IFoo)      # True를 반환한다.
isinstance(g, IFoo)      # True를 반환한다.
issubclass(FooProxy, IFoo) # True를 반환한다.
```

이 예에서 엄격한 타입 검사를 수행하지 않는 것에 주목하기 바란다. 인스턴스 검사 연산들이 오버로딩되어 있기 때문에 `IFoo`에 어떤 클래스가 속하는지 등록할 수 있게 되었다. 실제 프로그래밍 인터페이스에 대한 어떤 정보도 사용되지 않고 기타 다른 검증 작업도 수행되지 않는다. 여러분은 클래스들이 서로 어떻게 관련이 있는지에 상관없이 아무 클래스나 그룹에 등록할 수 있다. 일반적으로 클래스들은 동일한 프로그래밍 인터페이스를 구현하고 있다거나 하는 등 특정한 기준에 따라 묶

여진다. 하지만, `__instancecheck__()`나 `__subclasscheck__()`를 구현하는 경우에는 그런 기준에 얹매일 필요가 없다. 실제로 이 두 메서드를 어떻게 구현하느냐는 구현하는 사람의 자유다.

파이썬에서는 객체들을 그룹 짓고 인터페이스를 정의하며 타입을 검사하기 위한 더 형식화된 메커니즘을 제공한다. 다음 절에서 설명할 추상 기반 클래스가 바로 그것이다.

추상 기반 클래스

앞 절에서 `isinstance()`와 `issubclass()` 연산을 오버로딩할 수 있다는 것을 살펴보았다. 이렇게 함으로써 비슷한 클래스들을 묶는 객체를 생성하고 다양한 종류의 타입 검사를 수행할 수 있다. 추상 기반 클래스(Abstract base class)는 이와 유사하게 객체들을 계층으로 구성하거나 필수 메서드를 강제하는 등의 작업을 할 수 있게 해준다.

`abc` 모듈을 사용해서 추상 기반 클래스를 정의한다. 이 모듈에는 메타클래스(`ABCMeta`)와 장식자들(`@abstractmethod`과 `@abstractproperty`)이 정의되어 있고 다음과 같이 사용할 수 있다.

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Foo:           # 파이썬 3에서는 다음 문법을 사용
    __metaclass__ = ABCMeta      # class Foo(metaclass=ABCMeta)
    @abstractmethod
    def spam(self, a, b):
        pass
    @abstractproperty
    def name(self):
        pass
```

이 예에서 볼 수 있듯이 추상 클래스를 정의하려면 `ABCMeta`를 클래스의 메타클래스로 설정해주어야 한다(파이썬 2와 3은 문법이 약간 다르다는 것에 주의). 그 이유는 추상 클래스의 구현이 메타클래스에 기초하기 때문이다(메타클래스는 다음 절에서 설명한다). 추상 클래스 안에서 `@abstractmethod`과 `@abstractproperty` 장식자로 `Foo`의 하위 클래스에서 반드시 구현해야 하는 메서드나 프로퍼티를 표시한다.

추상 클래스는 인스턴스를 바로 생성하려고 만든 것이 아니다. 앞에서 나온 `Foo` 클래스의 인스턴스를 생성하려고 시도하면 다음과 같은 에러가 발생한다.

```
>>> f = Foo()
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract
methods spam
>>>
```

이 제약 사항은 파생 클래스로도 옮겨간다. 예를 들어, Bar 클래스가 Foo로부터 상속받았고 아직 구현하지 않은 추상 메서드가 있을 경우 Bar 객체를 생성하려고 할 때 비슷한 에러가 발생한다. 이러한 추가적인 검사가 있기 때문에 추상 기반 클래스는 하위 클래스에서 반드시 구현되어야 하는 메서드나 프로퍼티를 강제하는 데 유용하게 쓰인다.

추상 기반 클래스는 반드시 구현해야 하는 메서드나 프로퍼티에 대한 규칙을 검사하지만 인수나 반환 값이 적절한지 여부를 검사하지는 않는다. 즉, 하위 클래스 메서드의 인수가 추상 메서드의 인수와 동일한지 검사하지 않는다. 마찬가지로 하위 클래스 프로퍼티가 추상 프로퍼티와 동일한 연산(get, set, delete)을 지원하는지도 검사하지 않는다.

추상 기반 클래스의 인스턴스를 바로 생성하지는 못하지만 추상 기반 클래스에서는 하위 클래스에서 사용할 메서드나 프로퍼티를 정의할 수 있다. 또한, 하위 클래스에서 추상 메서드를 호출할 수 있다. 예를 들어, 하위 클래스에서 `Foo.spam(a, b)`을 호출할 수 있다.

추상 기반 클래스는 기존 클래스를 등록하는 기능을 제공한다. 다음과 같이 `register()` 메서드를 사용하면 된다.

```
class Grok(object):
    def spam(self,a,b):
        print("Grok.spam")

Foo.register(Grok) # Foo 추상 기반 클래스에 등록한다.
```

클래스가 추상 기반 클래스에 등록되면 추상 기반 클래스와 연관된 타입 검사 연산에 대해서 등록된 클래스의 인스턴스가 `True`를 반환하게 된다. 클래스가 추상 클래스에 등록되면 실제로 그 클래스가 추상 메서드나 프로퍼티를 구현하고 있는지를 검사하지는 않는다. 등록 과정은 단순히 타입 검사에만 영향을 미친다.

다른 객체지향 언어와는 달리 파이썬의 내장 타입들은 상대적으로 평평한 계층으로 구성되어 있다. 예를 들어, `int`나 `float` 같은 내장 타입을 보면 이 타입들은 숫자를 나타내는 중간 기반 클래스 대신 모든 객체의 조상인 `object`로부터 직접 상속 받은 것을 볼 수 있다. 이 때문에 간단히 모든 숫자 인스턴스에 대해서 작업을 수행

하는 등 포괄적인 범주에 기반하여 객체를 검사하거나 다른 프로그램을 작성하기가 쉽지 않다.

추상 클래스 메커니즘은 기존 객체들을 사용자 정의 타입 계층으로 구성할 수 있게 하여 이 문제를 해결한다. 몇몇 라이브러리 모듈에서는 제공하는 기능에 따라 내장 타입들을 적절히 조직화하기도 한다. collections 모듈은 순서열, 집합, 사전의 다양한 연산을 위한 추상 기반 클래스를 포함하고 있다. numbers 모듈은 수 계층을 구성하는 추상 기반 클래스를 포함하고 있다. 더 자세한 내용은 14장과 15장에서 다룬다.

메타클래스

파이썬에서 클래스를 정의하면 클래스 정의 자체도 객체가 된다. 다음 예를 보자.

```
class Foo(object): pass
isinstance(Foo, object) # True를 반환
```

가만히 생각해보면 누군가가 Foo 클래스 객체를 생성한 것 같다. 클래스 객체를 생성하는 일은 메타클래스(metaclass)라고 부르는 특수한 종류의 객체에 의해서 제어된다. 간단하게 말해서 메타클래스는 클래스를 어떻게 생성하고 관리하는지를 아는 객체이다.

앞에 나온 예에서 Foo의 생성을 제어하는 메타클래스는 type이라고 불리는 클래스이다. 사실, Foo의 타입을 출력해보면 Foo 타입은 type이라는 것을 알 수 있다.

```
>>> type(Foo)
<type 'type'>
```

class문으로 새로운 클래스를 정의하면 몇 가지 일이 일어난다. 먼저, 클래스의 몸체 안에 있는 문장이 개인 사전 안에서 실행된다. 개인 멤버의 이름(__로 시작하는 이름) 변형이 일어난다는 것 말고는 보통의 코드가 실행되는 것과 동일하다. 마지막으로, 해당 클래스 객체를 생성하기 위해서 클래스의 이름, 기반 클래스 목록, 생성된 사전이 메타클래스의 생성자에 전달된다. 다음은 이 과정이 어떻게 이루어지는지를 보여준다.

```
class_name = "Foo"          # 클래스의 이름
class_parents = (object,)   # 기반 클래스들
class_body = ""              # 클래스 몸체
def __init__(self,x):
    self.x = x
```

```

def blah(self):
    print("Hello World")
"""
class_dict = { }
# 내부 사전 class_dict 안에서 몸체를 실행한다.
exec(class_body,globals(),class_dict)

# 클래스 객체 Foo를 생성한다.
Foo = type(class_name,class_parents,class_dict)

```

메타클래스 type()이 호출되는 마지막 단계는 커스터마이즈할 수 있다. 클래스를 정의할 때 마지막 단계에서 벌어지는 일을 제어하는 방법에는 몇 가지가 있다. 먼저, `__metaclass__` 클래스 변수를 정의하거나(파이썬 2), 기반 클래스 목록 튜플에 metaclass 키워드 인수를 전달하여 클래스의 메타클래스를 지정할 수 있다.

```

class Foo:                      # 파이썬 3에서는 다음 문법을 사용
    __metaclass__ = type        # class Foo(metaclass=type)
...

```

메타클래스를 명시적으로 지정하지 않으면 class문은 기반 클래스들의 튜플에서 첫 번째 항목을 살펴본다(있는 경우). 이 경우 첫 번째 기반 클래스의 타입이 메타클래스가 된다. 따라서, 다음과 같이 쓸 경우 Foo는 object와 동일한 종류의 클래스가 된다.

```
class Foo(object): pass
```

기반 클래스를 지정하지 않으면 class문은 `__metaclass__`라는 이름의 전역 변수가 있는지를 살펴본다. 이 변수가 있으면 이것이 클래스를 생성하는 데 사용된다. 이 변수를 설정하면 간단한 class문에 대해서 클래스를 어떻게 생성할지를 제어할 수 있게 된다. 다음 예를 보자.

```

__metaclass__ = type
class Foo:
    pass

```

마지막으로, `__metaclass__` 값이 어디에도 없으면 파이썬은 기본 메타클래스를 사용한다. 파이썬 2에서는 기본으로 구형 클래스(old-style class)라고 불리는 types. ClassType이 사용된다. 파이썬에서는 원래 클래스 구현을 위해서 이 메타클래스를 사용했지만 파이썬 2.2부터는 사용이 권장되지 않는다. 파이썬에서 여전히 이 메타클래스를 지원하지만 새 코드에서는 사용하지 말아야 하며 이 메타클래스에 관해서는 이곳에서 더 이상 다루지 않는다. 파이썬 3에서는 기본 메타클래스로 간단히 `type()`이 사용된다.

메타클래스는 사용자가 객체를 정의하는 방식에 제약을 가하고자 하는 프레임워크를 구현하는 데 주로 사용된다. 커스텀 메타클래스를 정의할 때는 보통 type()에서 상속을 받고 __init__()이나 __new__() 같은 메서드를 다시 구현한다. 다음은 모든 메서드에 문서화 문자열이 존재해야 하는 것을 강제하는 메타클래스를 정의한 것이다.

```
class DocMeta(type):
    def __init__(self, name, bases, attrs):
        for key, value in attrs.items():
            # 특수 메서드와 개인 메서드를 건너뜀
            if key.startswith("__"):
                continue
            # 호출 가능하지 않은 것은 건너뜀
            if not hasattr(value, "__call__"):
                continue
            # 문서화 문자열을 검사함
            if not getattr(value, "__doc__"):
                raise TypeError("%s must have a docstring" % key)
        type.__init__(self, name, bases, attrs)
```

이 메타클래스의 __init__() 메서드에서는 클래스 사전의 내용을 검사한다. 사전을 훑으면서 메서드를 찾고 모든 메서드가 문서화 문자열을 가지고 있는지를 검사한다. 그렇지 않은 경우 TypeError 예외를 발생시킨다. 모두 문서화 문자열을 가지고 있으면 기본 구현인 type.__init__()을 호출해서 클래스를 초기화한다.

이 메타클래스를 사용하려면 클래스에서 이것을 사용하도록 지정해주어야 한다. 이를 위해서 다음과 같이 기반 클래스를 먼저 정의하는 방법을 많이 쓴다.

```
class Documented:          # 파이썬 3에서는 다음 문법을 사용한다.
    __metaclass__ = DocMeta  # class Documented(metaclass=DocMeta)
```

그러고 나서 이 기반 클래스를 문서화가 필요한 클래스의 부모로 지정한다. 다음 예를 보자.

```
class Foo(Documented):
    def spam(self, a, b):
        "spam does something"
        pass
```

이것은 클래스 정의를 들여다보고 필요한 정보를 수집하는 데 메타클래스를 사용하는 예이다. 보통 메타클래스는 이런 식으로 사용한다. 즉, 생성되는 클래스는 변경하지 않고 단지 추가적인 검사만 한다.

메타클래스를 좀 더 전문으로 사용하는 경우에는 클래스 정의를 살펴보는 동시에 변경하기도 한다. 클래스를 변경하려면 클래스 자체가 생성되기 전에 실행되는 __new__() 메서드를 재정의해야 한다. 이 메서드에서는 클래스에서 사용될 이름

을 변경할 수 있기 때문에 보통 속성을 기술자나 프로퍼티로 감쌀 때 흔히 사용된다. 한 예로, 다음은 154페이지 ‘기술자’ 절에 나왔던 TypedProperty 기술자를 메타클래스를 사용해서 작성한 버전을 보여준다.

```
class TypedProperty(object):
    def __init__(self, type, default=None):
        self.name = None
        self.type = type
        if default: self.default = default
        else: self.default = type()
    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)
    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance, self.name, value)
    def __delete__(self, instance):
        raise AttributeError("Can't delete attribute")
```

앞에 나왔던 코드와 다른 점은 기술자의 name 속성을 간단히 None으로 설정한 것이다. 이제 name을 자동으로 할당하기 위해 메타클래스를 사용할 것이다. 다음 예를 보자.

```
class TypedMeta(type):
    def __new__(cls, name, bases, dict):
        slots = []
        for key, value in dict.items():
            if isinstance(value, TypedProperty):
                value.name = "_" + key
                slots.append(value.name)
        dict['__slots__'] = slots
        return type.__new__(cls, name, bases, dict)

# 사용자 정의 클래스에서 사용할 기본 클래스
class Typed:          # 파이썬 3에서는 다음 문법을 사용한다.
    __metaclass__ = TypedMeta  # class Typed(metaclass=TypedMeta)
```

이 예에서 메타클래스는 클래스 사전을 뒤어보고 TypedProperty 인스턴스를 찾는다. 찾았으면 name 속성을 설정하고 slots에 이름 목록을 구축한다. 이것이 끝나면 __slots__ 속성이 클래스 사전에 추가되고 type() 메타클래스의 __new__() 메서드를 호출해서 클래스를 생성한다. 다음은 이 새로운 메타클래스를 사용하는 예이다.

```
class Foo(Typed):
    name = TypedProperty(str)
    num = TypedProperty(int, 42)
```

메타클래스로 사용자 정의 클래스의 작동 방식이나 사용 방법을 원하는 대로 변경할 수 있지만 표준 파이썬 문서에서 설명된 방식과 많이 다르게 클래스를 변경하는 것은 좋지 않다. 사용자 입장에서는 일반적인 코딩 규칙과 다르게 클래스를 작성해야 하는 경우 혼란을 느낄 수 있기 때문이다.

클래스 장식자

앞 절에서는 메타클래스를 정의함으로써 클래스를 생성하는 과정을 어떻게 커스터マイ즈할 수 있는지를 살펴보았다. 하지만 여러분이 원하는 것이 단순히 클래스가 정의되고 난 후 클래스를 레지스트리(registry)나 데이터베이스에 등록하는 것과 같은 몇 가지 추가 작업을 수행하고 싶을 뿐인 경우가 있다. 이러한 경우 메타클래스 대신 클래스 장식자를 사용할 수 있다. 클래스 장식자(decorator)는 입력으로 클래스를 받고 출력으로 클래스를 반환하는 함수이다. 다음 예를 보자.

```
registry = {}
def register(cls):
    registry[cls.__clsid__] = cls
    return cls
```

여기에서 등록 함수(register)는 클래스에 __clsid__ 속성이 있는지를 보고 있을 경우 클래스 식별자를 클래스 객체로 매핑하는 사전에 이 클래스를 추가한다. 이 등록 함수를 사용하려면 다음과 같이 클래스 정의 바로 앞에 장식자를 써주면 된다.

```
@register
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
```

이렇게 장식자 문법을 사용하면 편리하다. 다음과 같이 장식자 문법을 사용하지 않을 수도 있다.

```
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
register(Foo) # 클래스를 등록한다.
```

클래스 장식자 함수 안에서 클래스에 별의별 일을 다 할 수 있지만, 클래스에 래퍼를 써운다거나 클래스 내용을 바꾸는 것과 같은 과도한 작업은 하지 않는 것이 좋다.

8장

P y t h o n E s s e n t i a l R e f e r e n c e

모듈, 패키지와 배포

큰 규모의 파이썬 프로그램은 모듈과 패키지로 구성된다. 파이썬 표준 라이브러리에도 많은 모듈이 있다. 이 장에서는 모듈과 패키지 시스템에 관해서 자세하게 설명한다. 써드파티 모듈을 설치하는 방법과 소스 코드를 배포하는 방법도 알아본다.

모듈과 import문

어느 파이썬 소스 파일이든 모듈로서 사용될 수 있다. 다음 예를 보자.

```
# spam.py
a = 37
def foo( ):
    print("I'm foo and a is %s" % a)
def bar( ):
    print("I'm bar and I'm calling foo")
foo()
class Spam(object):
    def grok(self):
        print("I'm Spam.grok")
```

이 코드를 모듈의 형태로 로드하려면 import spam문을 사용하면 된다. 어떤 모듈을 로드하려고 import문을 처음으로 사용했을 때 다음 세 가지 일이 수행된다.

1. 모듈 소스 파일에서 정의된 모든 객체를 담을 컨테이너 역할을 하는 네임스페이스가 생성된다. 모듈 안에서 정의된 함수와 메서드에서 global문을 쓰면 이 네임스페이스를 사용하게 된다.

2. 새로 생성된 네임스페이스 안에서 모듈의 코드를 실행한다.

3. 모듈을 호출한 곳에 이 모듈 네임스페이스를 가리키는 이름을 생성한다. 이 이름은 다음과 같이 사용할 수 있다.

```
import spam      # 모듈 'spam'을 불러와서 실행한다.
x = spam.a      # 모듈 'spam'의 멤버에 접근한다.
spam.foo()      # 모듈 'spam'에 있는 함수를 호출한다.
s = spam.Spam() # spam.Spam()의 인스턴스를 생성한다.
s.grok()
...
```

`import`문은 소스 파일에 있는 모든 문장을 실행한다. 모듈 안에서는 변수, 함수, 클래스를 정의하는 것뿐만 아니라 계산을 수행하고 결과를 출력할 수도 있다. 그런 경우에는 출력되는 결과를 볼 수도 있다. 어떤 모듈에 있는 클래스에 접근하려고 할 때 사람들이 자주 실수하는 것이 있다. 파일 `spam.py`에 `Spam` 클래스가 정의되어 있을 때 이 클래스를 참조하려면 `spam.Spam`이라는 이름을 사용해야 한다는 것을 기억하기 바란다.

여러 모듈을 불러오려면 다음과 같이 모듈 이름을 콤마로 분리해서 써주면 된다.

```
import socket, os, re
```

모듈을 참조할 이름을 바꾸려면 `as` 한정어를 사용하면 된다. 다음 예를 보자.

```
import spam as sp
import socket as net
sp.foo()
sp.bar()
net.gethostname()
```

이렇게 모듈을 다른 이름으로 로드하면 `import`문을 사용한 소스 파일이나 컨텍스트에서만 이 이름이 생성된다. 다른 곳에서는 여전히 원래 모듈 이름을 사용해서 해당 모듈을 로드할 수 있다.

모듈을 임포트하면서 이름을 변경하는 기능은 확장성 있는 코드를 작성하는 데 유용하다. `xmlreader.py`와 `cvsreader.py`라는 두 모듈이 있고 파일에서 데이터를 읽어오는 `read_data(filename)`이라는 함수가 두 모듈 모두에 들어 있다고 하자. 두 함수는 받아들이는 파일 포맷만 다르다. 다음과 같이 코드를 작성하면 상황에 따라 적절하게 파일 읽기 모듈을 선택할 수 있다.

```
if format == 'xml':
    import xmlreader as reader
elif format == 'csv':
    import csvreader as reader
data = reader.read_data(filename)
```

파이썬에서 모듈은 1급 클래스 객체이다. 이 말은 모듈을 변수에 할당하거나 리스트 같은 데이터 구조에 넣거나 데이터로서 전달할 수 있다는 것을 의미한다. 예를 들어, 앞에 나온 예에서 reader 변수는 모듈 객체를 가리킨다. 내부적으로 모듈 객체는 모듈 네임스페이스의 내용을 담은 사전을 감싸는 형태로 구현된다. 모듈의 `__dict__` 속성으로 이 사전에 접근할 수 있고 모듈에서 값을 찾거나 변경할 때마다 이 사전이 쓰인다.

`import`문은 프로그램의 어느 곳이나 나타날 수 있다. 각 모듈 코드는 `import`문이 몇 번 사용되었는지에 상관없이 오직 한 번만 로드되어 실행된다. 이어지는 `import`문들은 단순히 앞에서 `import`문에 의해 이미 생성된 모듈 객체에 이름을 묶기만 한다. 현재 로드된 모든 모듈을 담은 사전은 변수 `sys.modules`에 저장되어 있다. 이 사전은 모듈 이름을 모듈 객체로 매핑한다. `import`문이 모듈의 새 복사본을 로드하는지 여부를 결정하는 데 이 사전이 사용된다.

모듈에서 선택된 기호 임포트

`from`문은 모듈 안에 특정한 정의를 현재 네임스페이스로 불러오는 데 사용한다. `from`문은 새로 생성된 모듈 네임스페이스를 가리키는 이름을 생성하지 않고 대신에 모듈에 정의된 특정 객체에 대한 참조를 현재 네임스페이스로 가져온다. 이 점을 제외하고는 `import`문과 동일하게 작동한다.

```
from spam import foo    # spam을 임포트하고 'foo'를 현재 네임스페이스로 가져온다.
foo()                  # spam.foo( )를 호출한다.
spam.foo()             # NameError: spam
```

`from`문에는 여러 이름을 콤마로 구분해서 쓸 수도 있다. 다음 예를 보자.

```
from spam import foo, bar
```

많은 이름을 임포트해야 할 때는 이름들을 팔호로 둘러싸면 된다. 이렇게 하면 `import`문을 여러 줄로 나눌 수 있다. 다음 예를 보자.

```
from spam import (foo,
                  bar,
                  Spam)
```

as 한정어를 사용하면 `from`으로 임포트되는 객체의 이름을 바꿀 수 있다. 다음 예를 보자.

```
from spam import Spam as Sp
```

```
s = Sp( )
```

별표(*) 와일드카드 문자는 모듈로부터 밑줄 하나로 시작하는 것을 제외한 모든 정의를 불러오는 데 사용된다. 다음 예를 보자.

```
from spam import * # 현재 네임스페이스로 모든 정의를 가져온다.
```

from module import * 문은 모듈의 가장 상위 수준에서만 사용할 수 있다. 특히 이 형태의 임포트를 함수 안에서 사용하는 것은 함수 유효 범위 규칙(함수가 내부 바이트코드로 컴파일될 때 함수 안에 있는 모든 기호는 확실하게 식별될 수 있어야 한다는 규칙)과 서로 작용하는 방식 때문에 적법하지 않다.

리스트 __all__을 모듈에 정의하면 from module import *이 임포트하는 이름을 세세하게 제어할 수 있다. 다음 예를 보자.

```
# 모듈: spam.py
__all__ = [ 'bar', 'Spam' ] # from spam import *로 임포트할 이름
```

from 형태로 어떤 정의를 임포트한다고 해서 유효 범위 규칙이 변경되지는 않는 다. 다음 코드를 보자.

```
from spam import foo
a = 42
foo() # "I'm foo and a is 37"을 출력
```

이 예에서 spam.py에 있는 foo()의 정의를 보면 전역 변수인 a를 참조하고 있다. foo 함수에 대한 참조가 다른 네임스페이스 안으로 들어간다고 해서 foo 함수 안에 있는 변수에 대한 바인딩 규칙이 변경되지는 않는다. 함수의 전역 네임스페이스는 항상 그 함수가 정의된 모듈이지, 함수가 임포트되고 호출되는 모듈이 아니다. 함수를 호출할 때도 마찬가지다. 예를 들어, 다음 코드에서 bar()를 호출하면 앞쪽에 있는 foo()를 재정의한 함수를 호출하는 것이 아니라 spam.foo()를 호출한다.

```
from spam import bar
def foo():
    print("I'm a different foo")
bar() # bar에서 foo()를 호출할 때 위에 정의된 foo()가 아니라
      # spam.foo()를 호출한다.
```

from 형태의 임포트를 사용할 때 흔히 혼동하는 것 중 하나로 전역 변수의 작동 방식과 관련된 부분이 있다. 예를 들어, 다음 코드를 보자.

```
from spam import a, foo # 전역 변수를 임포트한다.
a = 42                 # 변수를 수정한다.
foo()                  # "I'm foo and a is 37"을 출력한다.
print(a)                # "42"를 출력한다.
```

파이썬에서 변수 대입은 저장 연산이 아니라는 점을 이해하는 것이 중요하다. 즉, 앞의 예에서 a에 대한 대입문은 a의 이전 값을 덮어쓰면서 새로운 값을 a에 저장하지 않는다. 대신에 값 42를 담은 새로운 객체가 생성되고 이름 a는 이 객체를 가리키게 된다. a는 더 이상 임포트된 모듈에 있는 값을 가리키지 않고 다른 객체를 가리키게 된다. 이러한 작동 방식 때문에 변수를 C나 Fortran 같은 언어의 전역 변수나 공통 블록처럼 활용하기 위해서 from문을 사용하는 것은 불가능하다. 값을 변경할 수 있는 전역 매개변수를 정의하려면 해당 변수를 모듈에 넣고 import문을 사용함으로써 모듈 이름을 직접 참조하는 방식을 사용해야 한다(즉, spam.a처럼 써야 한다).

메인 프로그램으로 실행

파이썬 소스 파일을 실행하는 방법에는 두 가지가 있다. import문은 주어진 코드를 라이브러리 모듈인 것처럼 자신만의 네임스페이스에서 실행한다. 메인 프로그램이나 스크립트로서 코드를 실행하는 방법도 있다. 다음과 같이 인터프리터에 해당 프로그램을 스크립트 이름으로 주면 된다.

```
% python spam.py
```

각 모듈에는 모듈 이름을 담고 있는 변수 `__name__`이 정의된다. 이 변수의 내용을 보면 현재 실행 중인 모듈이 무엇인지 알 수 있다. 가장 상위 수준에 있는 모듈의 이름은 `__main__`이다. 명령행이나 대화식 모드를 통해 실행 중인 프로그램은 `__main__` 모듈 안에서 실행된다. 모듈로서 임포트된 것인지 `__main__` 안에서 실행 중인지에 따라 다르게 작동하는 코드를 작성할 수도 있다. 예를 들어, 다른 모듈에 의해서 임포트될 때에는 작동하지 않지만 메인 프로그램으로서 실행될 때에는 작동하는 검사 코드를 담은 모듈을 작성할 수 있다. 다음은 그 예를 보여준다.

```
# 메인 프로그램으로서 실행 중인지를 검사한다.
if __name__ == '__main__':
    # 네
    문장들
else:
    # 아니요, 모듈로서 임포트된 것이 틀림없다.
    문장들
```

라이브러리로서 사용될 모듈을 작성할 때에는 이렇게 검사 코드나 예를 담은 코드를 소스 파일에 포함시키는 경우가 많다. 예를 들어, 라이브러리의 기능을 검사하는 코드를 앞에서 본 것처럼 if문 안에 넣고 메인 프로그램으로 실행할 수 있다.

사용자가 모듈을 라이브러리로서 임포트할 때는 해당 코드가 실행되지 않는다.

모듈 검색 경로

모듈을 로드하기 위해서 인터프리터는 sys.path에 있는 디렉터리 목록을 검색한다. sys.path의 첫 번째 항목은 보통 현재 작업 디렉터리를 의미하는 빈 문자열 ‘’이다. sys.path에는 디렉터리 이름, .zip 아카이브 파일, .egg 파일 등을 추가할 수 있다. sys.path에서 항목이 나열된 순서가 모듈이 로드되는 순서를 결정한다. 검색 경로에 새로운 항목을 추가하려면 간단히 sys.path에 추가하면 된다.

경로를 지정할 때 보통 디렉터리 이름을 써주는 경우가 많지만 파이썬 모듈을 담은 zip 아카이브 파일을 써줄 수도 있다. 이렇게 하면 여러 모듈을 하나의 파일로 묶기가 수월하다. 예를 들어, foo.py와 bar.py 모듈이 있고 이 둘을 zip 파일 mymodules.zip에 넣었다고 하자. 다음과 같이 이 파일을 파이썬 검색 경로에 추가 할 수 있다.

```
import sys
sys.path.append("mymodules.zip")
import foo, bar
```

zip 파일 안의 구체적인 경로를 지정할 수도 있다. 또한 zip 파일 경로를 일반 경로와 섞어서 쓸 수도 있다. 다음 예를 보자.

```
sys.path.append("/tmp/modules.zip/lib/python")
```

zip 파일뿐만 아니라 .egg 파일도 검색 경로에 추가할 수 있다. .egg 파일은 setuptools 라이브러리에 의해서 만들어지는 패키지이다. 이 포맷은 써드파티 파일 뿐만 아니라 확장 기능을 설치할 때 자주 볼 수 있다. 실제로 .egg 파일은 단순히 추가 메타데이터(버전 번호, 의존 관계 등)를 담은 .zip 파일이다. 따라서 .zip 파일 관련 도구를 사용해서 .egg 파일을 살펴보고 데이터를 추출하는 등의 작업을 수행할 수 있다.

zip 파일을 임포트할 수 있지만 몇 가지 알아두어야 할 제약 사항이 있다. 첫째, zip 파일에서 임포트할 수 있는 파일 형식은 .py, .pyw, .pyc, .pyo 파일뿐이다. setuptools 같은 패키징 시스템에서 이 제약 사항을 피해가는 방법을 제공하기는 하지만(C 확장 기능을 임시 디렉터리에 풀고 거기서 모듈을 로딩하는 방식으로) C로 작성된 공유 라이브러리나 확장 모듈은 zip 파일에서 직접 로드할 수 없다. 게다가

파이썬은 .py 파일이 zip 파일에서 로드된 것일 경우에는 .pyc와 .pyo 파일을 만들지 않는다는(이 파일들에 관해서는 다음 절에서 설명한다). 따라서 이러한 파일은 미리 만들어서 zip 파일에 넣어두어야 모듈을 로딩할 때 성능이 저하되는 것을 막을 수 있다.

모듈 로딩과 컴파일

이 장에서 지금까지는 모듈을 순수 파이썬 코드만 담는 파일이라고 가정했다. 실제로 import로 로드할 수 있는 모듈은 크게 다음 네 가지로 나뉜다.

- 파이썬으로 작성된 코드(.py 파일)
- 공유 라이브러리나 DLL로 컴파일된 C나 C++ 확장 기능
- 모듈을 담는 패키지
- C로 작성되어 파이썬 인터프리터에 연결된 내장 모듈

모듈을 찾을 때(예를 들어, foo 모듈) 인터프리터는 sys.path에 있는 각 디렉터리에서 다음 파일을 찾는다(나열된 순서대로).

1. 패키지를 정의하는 foo 디렉터리
2. foo.pyd, foo.so, foomodule.so 또는 foomodule.dll(컴파일된 확장 기능)
3. foo.pyo(-O나 -OO 옵션이 사용될 때만)
4. foo.pyc
5. foo.py(윈도에서는 .pyw 파일도 검사)

패키지에 관해서는 곧 설명하도록 하겠다. 컴파일된 확장 기능에 관해서는 26장에서 설명한다. 모듈이 처음으로 임포트될 때 .py 파일은 바이트코드로 컴파일되어 .pyc로 다시 쓰여진다. 이어지는 임포트에서 인터프리터는 현재 .py 파일의 수정 날짜가 더 최신인 경우(이 경우 .pyc 파일이 다시 생성된다)가 아니면, 미리 컴파일된 바이트코드를 로드한다. .pyo 파일은 인터프리터를 -O 옵션으로 실행했을 때 사용된다. 이 파일은 줄 번호, 단언문, 기타 디버깅 정보 등을 벗겨낸 바이트코드를 담는다. 이에 따라 파일의 크기가 조금 작고 인터프리터가 더 빠르게 실행되는 효과가 있다. -O 옵션 대신 -OO 옵션을 사용하면 파일에서 문서화 문자열까지도 벗겨진다. 문서화 문자열은 모듈이 로드될 때 제거되는 것이 아니라 .pyo 파일이 생성될 때 제거된다. sys.path에 있는 디렉터리에 이 파일 중 어느 것도 없으면 인터프리터는 해당 이름이 내장 모듈 이름에 대응되는지를 검사한다. 이마저 실패하면

ImportError 예외가 발생한다.

import문장을 사용할 때만 소스 파일이 .pyc와 .pyo로 자동으로 컴파일된다. 명령행이나 표준 입력을 통해 프로그램을 실행할 때는 이 파일이 생성되지 않는다. 모듈의 .py 파일을 담은 디렉터리가 쓰기가 불가능할 경우에도 이 파일은 생성되지 않는다(권한이 부족하거나 zip 파일에 들어 있을 때). 인터프리터에 -B 옵션을 주어도 역시 파일은 생성되지 않는다.

.pyc와 .pyo 파일이 있으면 해당 .py 파일이 필요 없다. 코드를 패키징할 때 소스를 포함시키고 싶지 않으면 단순히 .pyc 파일만 넣으면 된다. 그래도 파이썬에는 강력한 조사(introspection)와 역어셈블(disassembly) 기능이 제공된다는 것을 명심하라. 잘 아는 사용자라면 소스가 공개되어 있지 않더라도 프로그램을 들여다보고 많은 정보를 얻을 수 있다. .pyc 파일은 버전에 종속적인 편이다. 파이썬의 특정 버전에서 생성된 .pyc 파일이 이후의 릴리스에서는 작동하지 않을 수도 있다.

import문에서 파일을 검색할 때는 윈도나 OS X 같이 바탕 파일 시스템이 파일 이름의 대소문자를 구별하지 않는 운영 체제에서조차도(파일 이름의 대소문자가 유지되기는 한다) 대소문자를 구별한다. 그러므로 import foo는 foo.py 파일을 임포트하지 FOO.PY 파일을 임포트하지는 않는다. 보통 대소문자만 차이 나는 모듈 이름을 사용하지 않는 것이 좋다.

모듈 재로딩과 내리기

파이썬은 이전에 임포트된 모듈을 다시 로드하거나 내리기 위한 기능을 제대로 제공하지 않는다. sys.modules에서 모듈을 제거할 수는 있지만 그렇다고 해서 모듈이 메모리에서 내려지는 것은 아니다. 그 이유는 모듈을 로드하려고 import를 사용한 부분에서 모듈 객체를 여전히 참조하기 때문이다. 게다가 모듈 안에서 정의된 클래스의 인스턴스가 있는 경우 그 인스턴스는 클래스 객체를 참조하고 클래스 객체는 다시 자신이 정의된 모듈을 참조한다.

모듈에 대한 참조가 여러 곳에 있을 수 있기 때문에 일반적으로 모듈의 구현을 변경한 후에 다시 로드하는 것은 사실상 불가능하다. 예를 들어, sys.modules에서 모듈을 제거하고 import로 다시 로드한다고 해도 이 모듈에 대한 모든 이전 참조가 소급해서 갱신되지는 않는다. 대신, 가장 최근에 import문으로 생성된 새로운 모듈에 대한 참조와 다른 곳에서 임포트되어 생성된 예전 모듈에 대한 참조가 함께

존재하게 된다. 이 상황은 여러분이 원하는 바가 아닐 것이며 전체 실행 환경을 세심하게 제어할 수 있는 경우가 아니라면 제대로 된 제품 코드에서 사용하는 것은 안전하지 않다.

파이썬의 예전 버전에는 `reload()` 함수가 있어서 모듈을 다시 로드할 수 있었다. 하지만 이 함수는 안전하지 않으며(이전에 언급한 모든 이유 때문에) 디버깅 목적을 제외하고는 사용하지 않도록 한다. 파이썬 3에서는 이 기능이 완전히 제거되었다. 이 기능은 사용하지 않는 것이 가장 좋다.

파이썬의 C/C++ 확장 기능은 어떻게든 제대로 내리거나 다시 로드하는 것이 어렵다. 가능한 방법도 없고 바탕 운영 체제에서 아예 허용하지 않을 수도 있다. 파이썬 인터프리터를 다시 시작하는 수밖에 없다.

패키지

모듈을 공통의 이름으로 묶는 데 패키지를 사용한다. 패키지를 사용하면 모듈 사이에 이름 충돌 문제를 해결할 수 있다. 패키지를 만들려면 패키지 이름을 가진 디렉터리를 만들고 이 디렉터리에 `__init__.py` 파일을 생성하면 된다. 그런 다음 필요에 따라 소스 파일, 컴파일된 확장 기능, 하위 패키지 등을 추가하면 된다. 다음은 패키지를 구성하는 예를 보여준다.

```

Graphics/
    __init__.py
    Primitive/
        __init__.py
        lines.py
        fill.py
        text.py
        ...
    Graph2d/
        __init__.py
        plot2d.py
        ...
    Graph3d/
        __init__.py
        plot3d.py
        ...
Formats/
    __init__.py
    gif.py
    png.py
    tiff.py
    jpeg.py

```

import문으로 패키지로부터 모듈을 몇 가지 형태로 로드할 수 있다.

- import Graphics.Primitive.fill

하위 모듈인 Graphics.Primitive.fill을 로드한다. 이 모듈의 내용에 접근하려면 Graphics.Primitive.fill.floodfill(img, x, y, color) 같이 이름을 직접 써주어야 한다.

- from Graphics.Primitive import fill

하위 모듈인 fill을 로드하며 앞쪽 패키지 이름 없이 사용할 수 있다. 예를 들면, fill.floodfill(img, x, y, color).

- from Graphics.Primitive.fill import floodfill

하위 모듈 fill을 로드하며 floodfill 함수를 바로 사용할 수 있다. 예를 들어, floodfill(img, x, y, color).

패키지의 일부가 처음으로 로드될 때 __init__.py 파일에 있는 코드가 실행된다. 이 파일은 비어 있을 수 있고 패키지를 초기화하는 코드를 담을 수도 있다. import 를 실행하는 동안 만나게 되는 모든 __init__.py 파일이 차례대로 실행된다. 즉, import Graphics.Primitive.fill이 실행되면 먼저 Graphics 디렉터리에 있는 __init__.py 파일이 실행되고 다음으로 Primitive 디렉터리에 있는 __init__.py 파일이 실행된다.

패키지와 관련해서 다음 문장을 처리할 때 약간의 문제가 있을 수 있다.

```
from Graphics.Primitive import *
```

보통 프로그래머는 패키지와 연관된 모든 하위 모듈을 현재 네임스페이스로 임포트하려고 이 문장을 사용한다. 그러나 시스템마다 파일 이름과 관련된 규칙이 다르기 때문에(특히 대소문자 구별 여부) 어떤 모듈을 임포트해야 할지를 정확하게 결정하기 어렵다. 그 결과, 이 문장은 단순히 Primitive 디렉터리에 있는 __init__.py 파일 안에서 정의된 이름을 임포트한다. 이러한 작동 방식을 바꾸려면 패키지와 연관된 모듈 이름을 담는 리스트인 __all__을 정의하면 된다. 다음과 같이 패키지의 __init__.py 파일에 이 리스트를 정의하면 된다.

```
# Graphics/Primitive/__init__.py
__all__ = ["lines", "text", "fill"]
```

이제 사용자가 from Graphics.Primitive import * 문을 실행하면 나열된 모든 하위 모듈이 기대했던 대로 로드된다.

한 하위 모듈에서 동일한 패키지의 다른 하위 모듈을 임포트하려는 경우에

도 문제가 발생할 수 있다. 예를 들어, Graphics.Primitive.fill 모듈에서 Graphics.Primitive.lines 모듈을 임포트하려고 한다고 하자. 이렇게 하려면 단순하게 완전히 한정된 이름(예를 들어, from Graphics.Primitive import lines)을 사용하거나 아니면 다음과 같이 패키지에 상대적인 임포트를 사용해야 한다.

```
# fill.py
from . import lines
```

앞의 예에서 from . import lines문의 점(.)은 현재 모듈과 동일한 디렉터리를 가리킨다. 따라서, 이 문장은 파일 fill.py와 동일한 디렉터리에 있는 모듈 lines를 찾는다. 패키지의 하위 모듈을 임포트하려고 import module 같은 문장을 사용하지 않도록 한다. 파이썬의 예전 버전에서는 import module문이 표준 라이브러리 모듈을 의미하는지 패키지의 하위 모듈을 의미하는지 명확하지 않았다. 예전 버전의 파이썬에서는 먼저 import문이 나타난 모듈과 동일한 패키지 디렉터리에 있는 모듈을 로드하려고 시도하고, 해당 모듈을 찾지 못했을 경우에만 표준 라이브러리 모듈을 로드하였다. 그러나 파이썬 3에서는 import에 절대 경로가 사용된 것으로 가정하고 module을 표준 라이브러리에서 로드하려고 시도한다. 상대적인 임포트를 사용하면 더 명확하게 의도를 표현할 수 있다.

상대적인 임포트는 동일한 패키지 안에서 다른 디렉터리에 들어 있는 하위 모듈을 로드하는 데도 사용할 수 있다. 예를 들어, 모듈 Graphics.Graph2D.plot2d에서 Graphics.Primitives.lines 모듈을 임포트하려면 다음과 같이 하면 된다.

```
# plot2d.py
from ..Primitives import lines
```

여기서 ..는 디렉터리 밖으로 한 수준 벗어나게 하고 Primitives는 다른 패키지 디렉터리로 들어가게 한다.

import문 중 from module import symbol의 형태로만 상대적인 임포트를 지정할 수 있다. 따라서 import ..Primitives.lines나 import .lines는 문법 오류이다. 또한 symbol은 유효한 식별자여야 한다. 따라서, from .. import Primitives.lines도 적법하지 않다. 마지막으로, 상대적인 임포트는 패키지 안에서만 사용해야 한다. 단순히 다른 디렉터리에 있는 모듈을 가리키는 데 상대적인 임포트를 사용하면 안 된다.

패키지 이름을 임포트한다고 해서 그 안에 있는 모든 하위 모듈이 임포트되는 것은 아니다. 예를 들어, 다음 코드는 제대로 작동하지 않는다.

```
import Graphics
Graphics.Primitive.fill.floodfill(img,x,y,color) # 실패!
```

이 문제를 해결하려면 import Graphics가 Graphics 디렉터리에 있는 `__init__.py` 파일을 실행하기 때문에 다음과 같이 상대적인 임포트를 사용하여 모든 하위 모듈을 자동으로 로드하게 만들어야 한다.

```
# Graphics/__init__.py
from . import Primitive, Graph2d, Graph3d

# Graphics/Primitive/__init__.py
from . import lines, fill, text, ...
```

이제 import Graphics문은 모든 하위 모듈을 임포트하고 완전히 한정된 이름을 사용해서 해당 모듈에 접근할 수 있게 되었다. 상대적인 임포트는 앞에서 설명한 대로 사용해야 한다. 간단히 import module 같은 문장을 사용하면 표준 라이브러리 모듈이 로드된다.

마지막으로, 파이썬에서 패키지를 임포트하면 패키지의 하위 모듈을 검색하는 데 사용되는 디렉터리 목록을 담는 특수한 변수인 `__path__`가 생성된다(`__path__`는 `sys.path` 변수의 패키지용 버전이다). `__path__`는 `__init__.py` 파일 안에서 접근이 가능하다. 처음에는 패키지의 디렉터리 이름만 담고 있지만, 필요한 경우 `__path__` 목록에 이름을 추가하여 하위 모듈을 찾는 데 사용할 검색 경로를 변경할 수 있다. 파일 시스템에서 패키지를 구성하는 일이 복잡하며 디렉터리 구조를 패키지 계층과 깔끔하게 일치시키기 어려울 때 유용하게 쓸 수 있다.

파이썬 프로그램과 라이브러리 배포

파이썬 프로그램을 다른 사람들에게 배포하려면 `distutils` 모듈을 사용해야 한다. 먼저 여러분이 한 일을 `README` 파일, 지원 문서, 소스 코드 등을 포함하는 디렉터리에 넣는다. 보통 이 디렉터리에는 라이브러리 모듈, 패키지, 스크립트가 섞여 존재한다. 모듈과 패키지는 `import`문으로 로드될 소스 파일을 말한다. 스크립트는 인터프리터에 의해 메인 프로그램으로서 실행되는 프로그램을 말한다(예를 들어, `python scriptname` 형태로 실행된다). 다음은 파이썬 코드를 담은 디렉터리의 예를 보여준다.

```
spam/
    README.txt
```

```
Documentation.txt
libspam.py      # 라이브러리 모듈 하나
spampkg/        # 지원 모듈 패키지
    __init__.py
    foo.py
    bar.py
runspam.py     # python runspam.py 형태로 실행될 스크립트
```

보통 최상위 디렉터리에서 파이썬 인터프리터를 실행할 수 있게 디렉터리 내용을 구성하는 것이 좋다. 예를 들어 spam 디렉터리에서 파이썬을 구동하면, 모듈 검색 경로 같은 파이썬 설정을 변경하지 않으며 모듈과 패키지 구성 요소를 임포트하고 스크립트를 실행할 수 있어야 한다.

코드를 정리하였으면 다음으로 최상위 디렉터리(앞의 예에서 spam)에 setup.py 파일을 생성한다. 이 파일 안에 다음 코드를 넣자.

```
# setup.py
from distutils.core import setup

setup(name = "spam",
      version = "1.0",
      py_modules = ['libspam'],
      packages = ['spampkg'],
      scripts = ['runspam.py'],
      )
```

setup()을 호출할 때 py_modules 인수는 파일 하나짜리 모듈 목록을, packages는 패키지 디렉터리 목록을, scripts는 스크립트 파일 목록을 나타낸다. 어느 인수든

표 8.1 setup()의 매개변수

매개변수	설명
name	패키지 이름(필수)
version	버전 번호(필수)
author	저자 이름
author_email	저자의 이메일 주소
maintainer	관리자 이름
maintainer_email	관리자 이메일
url	패키지 홈페이지
description	패키지에 대한 짧은 설명
long_description	패키지에 대한 긴 설명
download_url	패키지를 다운로드할 수 있는 위치
classifiers	문자열 분류자 목록

해당하는 구성 요소가 없는 경우(스크립트가 없다든지) 생략할 수 있다. name은 패키지의 이름을, version은 버전 번호를 문자열로 나타낸다.

setup()을 호출할 때 패키지에 관한 다양한 메타데이터를 전달할 수 있다. 표 8.1은 가장 흔히 사용되는 매개변수를 나열한 것이다. [‘Development Status :: 4 - Beta’, ‘Programming Language :: Python’] 같은 문자열 목록을 담는 classifiers 매개변수를 제외하고 모든 값은 문자열이어야 한다(<http://pypi.python.org>에 전체 인수 목록이 설명되어 있다).

setup.py 파일만 생성해도 소스 배포본을 만드는 데 충분하다. 다음 셸 명령을 입력하면 소스 배포본이 만들어진다.

```
% python setup.py sdist
...
%
```

이렇게 하면 spam/dist 디렉터리에 spam-1.0.tar.gz나 spam-1.0.zip 같은 아카이브 파일이 생성된다. 이 파일을 다른 사람에게 주어서 소프트웨어를 설치하게 하면 된다. 설치를 하려면 사용자는 아카이브 파일을 풀어서 다음 단계를 수행하면 된다.

```
% unzip spam-1.0.zip
...
% cd spam-1.0
% python setup.py install
...
%
```

이렇게 하면 소프트웨어가 지역 파이썬 배포 장소에 설치되고 누구나 사용할 수 있게 된다. 모듈과 패키지는 보통 파이썬 라이브러리 안에 있는 ‘site-packages’ 디렉터리에 설치된다. 이 디렉터리의 정확한 위치는 sys.path를 보면 알 수 있다. 스크립트는 보통 유닉스 기반 시스템에서는 파이썬 인터프리터와 동일한 디렉터리에 설치되고 윈도에서는 ‘Scripts’ 디렉터리에 설치된다(보통의 설치본에서는 ‘C\Python26\Scripts’에서 찾을 수 있다).

유닉스에서 스크립트의 첫 번째 줄이 #!로 시작하고 “python”이라는 텍스트를 담고 있으면 설치 프로그램에서 해당 줄을 파이썬의 지역 설치본을 가리키도록 덮어쓴다. 따라서, 스크립트에 /usr/local/bin/python처럼 특정한 파이썬 위치를 직접 써넣은 경우라도 다른 곳에 파이썬이 설치된 시스템에서도 잘 작동한다.

setup.py 파일은 소프트웨어 배포와 관련해서 기타 명령을 몇 가지 제공한다. ‘python setup.py bdist’라고 입력하면 모든 .py 파일이 .pyc 파일로 미리 컴파일되

고 지역 플랫폼을 흉내 내는 디렉터리 구조에 담겨진 이진 배포본이 생성된다. 이 배포본은 응용 프로그램이 일부 특정 플랫폼에 의존적일 때 필요하다(예를 들어, 컴파일되어야 하는 C 확장 기능이 있을 때). 윈도에서 ‘python setup.py bdist_wininst’를 입력하면 .exe 파일이 만들어진다. 이 파일을 열면 윈도 설치 대화창이 시작되고 소프트웨어를 어디에 설치할 것인지 묻는다. 이 배포본은 나중에 패키지를 제거하는 일이 수월하도록 레지스트리에 정보를 추가하기도 한다.

distutils 모듈은 시스템에 이미 파이썬이 설치되어 있다고 가정한다(따로 다운로드되어). 파이썬 런타임과 소프트웨어를 하나의 이진 실행 파일로 함께 묶어서 소프트웨어 패키지를 생성할 수도 있지만, 이렇게 하는 방법에 관해서 여기서 다루기는 힘들다(더 자세한 정보는 py2exe나 py2app 같은 썬드파티 모듈을 살펴보라). 라이브러리나 간단한 스크립트를 배포하는 경우라면, 일반적으로 코드와 함께 파이썬 인터프리터와 런타임을 함께 패키징하는 일은 불필요하다.

distutils에는 이곳에서 다루지 않은 더 많은 옵션이 제공된다. 26장에서는 distutils를 사용해서 C와 C++ 확장 기능을 어떻게 컴파일할 수 있는지를 설명한다.

표준 파이썬 배포 방식은 아니지만 종종 파이썬 소프트웨어가 .egg 파일로 배포되는 것을 볼 수 있다. 이 포맷은 인기 있는 확장 기능인 setuptools로 생성된 것이다(<http://pypi.python.org/pypi/setuptools>). setup.py 파일의 첫 번째 부분을 다음과 같이 변경해서 setuptools도 지원할 수 있다.

```
# setup.py
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup

setup(name = "spam",
      ...
)
```

썬드파티 라이브러리 설치

썬드파티 라이브러리나 파이썬의 확장 기능을 찾기 가장 좋은 곳은 <http://pypi.python.org>에 있는 파이썬 패키지 색인(PyPI: Python Package Index)이다. 썬드파티 모듈을 설치하는 일은 보통 간단하지만 다른 썬드파티 모듈에 의존하는 규모가 큰 패키지의 경우 꽤 복잡해질 수 있다. 주요 확장 기능을 설치할 때는 플랫폼에 맞게 만들어진 설치 프로그램이 있어서 설치 과정의 각 단계를 안내하는 대화창 화면

을 따라가기만 하면 된다. 기타 모듈은 보통 다운로드한 파일을 풀어서 setup.py 파일을 찾고 python setup.py install을 입력하면 된다.

기본으로 써드파티 모듈은 파이썬 표준 라이브러리의 site-packages 디렉터리에 설치된다. 이 디렉터리에 접근하려면 보통 루트 혹은 관리자 권한이 필요하다. 필요한 권한이 없다면 python setup.py install -user를 입력해서 모듈을 사용자별 라이브러리 디렉터리에 설치할 수 있다. 이렇게 설치되는 패키지는 유닉스에서는 “/Users/beazley/.local/lib/python2.6/site-packages” 같은 사용자별 디렉터리에 설치된다.

소프트웨어를 다른 곳에 설치하려면 setup.py에 –prefix 옵션을 주면 된다. 예를 들어, python setup.py install –prefix=/home/beazley/pypackages를 입력하면 모듈이 /home/beazley/pypackages에 설치된다. 표준 디렉터리가 아닌 곳에 모듈을 설치한 경우에는 새로 설치된 모듈을 찾을 수 있도록 sys.path를 수정해야 할 수도 있다.

파이썬의 많은 확장 기능이 C나 C++ 코드로 되어 있다. 소스 배포본을 다운받은 경우에는 설치를 제대로 하려면 시스템에 C++ 컴파일러가 설치되어 있어야 한다. 유닉스, 리눅스나 OS X에서는 보통 문제가 되지 않는다. 윈도에서는 보통 마이크로소프트 비주얼 스튜디오(Microsoft Visual Studio)가 설치되어 있어야 한다. 윈도에서 작업하는 중이라면 미리 컴파일된 확장 기능을 찾아보는 것이 더 낫다.

setuptools를 설치했으면 패키지를 설치하는 데 사용할 수 있는 easy_install이라는 스크립트가 있을 것이다. 패키지를 설치하려면 간단히 easy_install pkgname을 입력한다. 적절히 설정되어 있는 경우 알아서 필요한 소프트웨어와 기타 의존하는 패키지를 PyPI에서 다운받아서 설치해줄 것이다. 물론 상황에 따라 다르게 작동할 수도 있다.

여러분의 소프트웨어를 PyPI에 추가하고 싶으면 간단히 python setup.py register를 입력한다. 이렇게 하면 소프트웨어의 최신 버전에 관한 메타데이터가 색인에 올라간다(먼저 사용자 이름과 암호를 등록해야 할 것이다).

9장

P y t h o n E s s e n t i a l R e f e r e n c e

입력과 출력

이 장에서는 명령줄 옵션, 환경 변수, 파일 입출력, 유니코드, pickle 모듈로 객체를 직렬화하는 방법 등 파이썬의 입력과 출력에 관련된 기본적인 내용을 설명한다.

명령줄 옵션 읽기

파이썬을 시작하면 명령줄 옵션들이 sys.argv 리스트에 저장된다. 이 리스트의 첫번째 원소는 프로그램의 이름이다. 나머지 원소들은 명령줄에서 프로그램 이름 다음에 나오는 옵션들을 나타낸다. 다음 프로그램은 명령줄 인수를 직접 처리하는 간단한 예를 보여준다.

```
import sys
if len(sys.argv) != 3:
    sys.stderr.write("Usage : python %s inputfile outputfile\n"
                     % sys.argv[0])
    raise SystemExit(1)
inputfile = sys.argv[1]
outputfile = sys.argv[2]
```

이 프로그램에서 sys.argv[0]에는 실행된 스크립트의 이름이 담긴다. 위 코드에서 보듯이 보통 sys.stderr로 에러 메시지를 출력하고 0이 아닌 종료 코드와 함께 SystemExit 예외를 발생시켜서 프로그램을 잘못 실행했을 경우 사용법을 알려준다.

앞에서 본 것처럼 간단한 스크립트로 명령줄 옵션을 직접 처리할 수 있지만, 복잡한 옵션을 처리해야 하는 경우에는 optparse 모듈을 사용하면 된다. 다음 예를 살펴보자.

```

import optparse
p = optparse.OptionParser( )

# 인수를 받아들이는 옵션
p.add_option("-o",action="store",dest="outfile")
p.add_option("--output",action="store",dest="outfile")

# 불리언 플래그를 설정하는 옵션
p.add_option("-d",action="store_true",dest="debug")
p.add_option("--debug",action="store_true",dest="debug")

# 옵션의 기본 값을 설정
p.set_defaults(debug=False)

# 명령줄을 파싱
opts, args = p.parse_args( )

# 옵션 설정값을 추출
outfile = opts.outfile
debugmode = opts.debug

```

이 예에서, 두 가지 종류의 옵션을 추가하였다. 첫 번째 옵션인 -o 또는 --output에는 반드시 추가 인수를 제공해야 한다. p.add_option() 호출에서 action='store'를 지정해서 이 부분을 명시하였다. 두 번째 옵션인 -d 또는 --debug는 단순히 불리언 플래그를 설정한다. p.add_option()에서 action='store_true'를 지정해서 이 부분을 명시하였다. p.add_option()에서 dest 인수는 파싱 후에 인수 값이 저장될 속성의 이름을 지정하는 데 쓰인다. p.set_defaults() 메서드는 하나 이상의 옵션에 대해서 기본 값을 설정한다. 이 메서드에서 사용되는 인수 이름은 옵션에 지정한 인수의 저장 목적지(dest) 이름과 동일하여야 한다. 기본 값을 설정하지 않으면 기본 값으로 None이 사용된다.

앞에 나온 프로그램은 다음에 나오는 명령줄 형식을 모두 인식할 수 있다.

```

% python prog.py -o outfile -d infile1 ... infileN
% python prog.py --output=outfile --debug infile1 ... infileN
% python prog.py -h
% python prog.py --help

```

파싱은 p.parse_args() 메서드를 사용하여 수행한다. 이 메서드는 원소가 2개인 튜플 (opts, args)을 반환한다. 여기서 opts는 파싱된 옵션 값들을 담는 객체이고 args는 옵션이 아닌 항목들을 담는 리스트이다. 옵션 값은 dest가 목적지 이름일 때 opts, dest를 이용해 얻어올 수 있다. 예를 들어, -o 또는 --output의 인수 값은 opts.outfile에 저장되고 args는 ['infile1', ..., 'infileN'] 같은 나머지 인수들을 담은 리스트가 된다. optparse 모듈은 사용자가 요청할 경우 사용 가능한 옵션을 출력하는 -h

또는 `__help` 옵션을 자동으로 제공한다. 잘못된 옵션을 지정하면 에러 메시지가 출력된다.

여기에서는 optparse 모듈의 가장 기본적인 사용법만 살펴보았다. 더 고급 옵션 처리 기법은 19장에서 다룬다.

환경 변수

사전인 `os.environ`을 통해서 환경 변수에 접근할 수 있다. 다음 예를 살펴보자.

```
import os
path = os.environ["PATH"]
user = os.environ["USER"]
editor = os.environ["EDITOR"]
... etc ...
```

환경 변수를 수정하려면 `os.environ` 변수를 설정하면 된다. 다음은 그 예이다.

```
os.environ["FOO"] = "BAR"
```

`os.environ`을 수정하면 현재 실행 중인 프로그램과 파이썬에 의해서 생성된 하위 프로세스가 영향을 받는다.

파일과 파일 객체

내장 함수 `open(name [, mode [, bufsize]])`은 다음 예와 같이 파일을 열고 파일 객체를 생성한다.

```
f = open("foo")           # 읽기용으로 "foo" 열기
f = open("foo",'r')       # 읽기용으로 "foo" 열기(위와 동일)
f = open("foo",'w')       # 쓰기용으로 "foo" 열기
```

파일 모드로서 r은 읽기용, w는 쓰기용, a는 추가용을 의미한다. 이 세 가지 파일 모드 모두 텍스트 모드로 작동하고 암묵적으로 줄바꿈 문자 '\n'에 대한 변환을 수행하기도 한다. 예를 들어, 윈도에서 '\n'을 쓰면 실제로는 두 문자로 구성된 순서열 '\r\n'이 출력된다(그리고 파일을 다시 읽으면 '\r\n'은 단일 '\n'으로 다시 변환된다). 이진 데이터를 가지고 작업을 할 경우에는 'rb' 또는 'wb'와 같이 파일 모드에 'b'를 추가하면 된다. 이렇게 하면 줄바꿈 문자 변환을 수행하지 않는다. 이진 데이터를 처리하는 코드 간 호환성 문제를 고려하고 있다면 모드에 'b'를 추가해야 한다(유닉스에서는 텍스트와 이진 파일 사이에 구분이 없기 때문에 간혹 'b'를 빼

뜨리는 실수를 한다). 이렇게 모드 사이에 차이가 있기 때문에, ‘rt’, ‘wt’, ‘at’와 같이 사용자의 의도를 더 명확히 표시하기도 한다.

파일을 직접 수정하려면 ‘r+’ 또는 ‘w+’와 같이 플러스(+) 문자를 추가하여 파일을 열면 된다. 파일을 수정 모드로 열고 나면, 다음 입력 연산을 수행하기 전에 모든 이전 출력 연산의 결과를 내보내기만 하면 파일에 대해 입력과 출력을 모두 수행할 수 있다. 파일을 ‘w+’ 모드로 열면 파일 길이가 먼저 0으로 줄어든다.

파일을 ‘U’ 또는 ‘rU’ 모드로 열 경우 파일을 읽을 때 보편 줄바꿈 문자 기능(universal newline support)^{o)} 지원된다. 이 기능은 다양한 파일 I/O 함수로부터 반환되는 문자열에서 여러 줄바꿈 인코딩(‘\n’, ‘\r’, ‘\r\n’ 같은)을 표준 문자 ‘\n’으로 바꾸어주기 때문에 서로 다른 플랫폼 사이에 호환이 필요한 작업을 수행할 때 유용하다. 예를 들어, 윈도 프로그램에서 생성된 텍스트 파일을 유닉스 시스템에서 처리하는 스크립트를 작성할 때 유용하다.

옵션인 매개변수 bufsize는 파일의 버퍼링 방식을 지정한다. 0은 버퍼링하지 않는 것, 1은 줄 버퍼링을 의미하며 음수는 시스템 기본 값의 사용을 의미한다. 기타 양수 값은 사용될 대략적인 버퍼 크기를 바이트로 나타낸다.

파이썬 3에서는 open() 함수에 4개의 매개변수가 추가되었고 open(name [, mode [, bufsize [, encoding [, errors [, newline [, closefd]]]]]) 형식으로 호출한다. encoding은 ‘utf-8’ 또는 ‘ascii’ 같은 인코딩 이름을 나타낸다. errors는 인코딩 에러에 대한 에러 처리 정책을 나타낸다(자세한 내용은 이 장의 유니코드에 관한 부분에서 설명한다). newline은 보편 줄바꿈 모드의 작동 방식을 제어하며 None, ‘’, ‘\n’, ‘\r’, ‘\r\n’ 중 하나가 될 수 있다. None으로 설정하면, ‘\n’, ‘\r’, ‘\r\n’ 형태의 줄 끝 표시를 ‘\n’으로 변환한다. ‘’(빈 문자열)로 설정할 경우 줄 끝 표시가 문자로 인식되지만 입력 텍스트에 변환되지 않은 채로 남겨진다. 줄바꿈이 기타 값을 가지면 그것이 줄바꿈 문자로 사용된다. closefd는 close() 메서드가 호출될 때 내부 파일 기술자(file descriptor)를 실제로 닫을 것인지 여부를 제어한다. 이 값은 기본으로 True로 설정된다.

표 9.1은 파일 객체에서 지원하는 메서드를 보여준다.

표 9.1 파일 메서드

메서드	설명
f.read([n])	최대 n 바이트까지 읽음
f.readline([n])	최대 n 개의 문자까지 한 줄을 읽음. n을 생략하면 줄 전체를 읽음
f.readlines([size])	모든 줄을 읽어서 리스트로 반환함. 옵션인 size는 읽기를 중단하기 전 까지 파일에서 읽을 대략적인 문자 개수를 나타냄
f.write(s)	문자열 s를 쓴다
f.writelines(lines)	순서열 lines에 있는 모든 문자열을 쓴다
f.close()	파일을 닫음
f.tell()	현재 파일 포인터를 반환
f.seek(offset [, whence])	새 파일 위치를 찾음
f.isatty()	터미널일 경우 1을 반환
f.flush()	출력 버퍼를 비움
f.truncate([size])	최대 size 바이트로 파일을 자름
f.fileno()	정수 파일 기술자를 반환하거나 파일이 닫힌 경우 ValueError를 발생시킴
f.next()	다음 줄을 반환하거나 StopIteration 예외를 발생시킴. 파이썬 3에서는 f.__next__()

read() 메서드는 최대 문자 개수를 나타내는 옵션인 length 매개변수가 주어지지 않으면 파일 전체를 문자열로 반환한다. readline() 메서드는 종료를 알리는 줄바꿈 문자가 포함된 다음 줄을 읽어서 반환한다. readlines() 메서드는 모든 입력 줄을 문자열 리스트로 반환한다. readline() 메서드는 추가적으로 최대 줄 길이 n을 입력 받는다. 줄이 n 개의 문자보다 길면 최초 n 개의 문자가 반환된다. 줄 데이터에서 남아 있는 부분은 버려지지 않고 이어지는 읽기 연산에 의해 반환된다. readlines() 메서드는 그만둘 때까지 읽을 대략적인 문자 개수를 나타내는 size 매개변수를 입력받는다. 실제로 읽히는 문자 개수는 버퍼에 데이터가 얼마나 들어 있느냐에 따라서 더 많을 수도 있다.

readline()과 readlines() 메서드 모두 플랫폼을 인식하여 다양한 줄바꿈 문자를 적절히 처리한다(예를 들어, '\n' 대 '\r\n'). 파일을 보편 줄바꿈 모드('U'나 'tU')로 열면 줄바꿈 문자가 '\n'으로 변환된다.

read()와 readline()은 빈 문자열을 반환하여 파일의 끝을 표시한다. 다음 코드는 파일 끝을 감지하는 방법을 보여준다.

```

while True:
    line = f.readline( )
    if not line:          # 파일 끝
        break

```

for 루프와 반복을 사용하면 파일에서 모든 줄을 쉽게 읽을 수 있다. 다음 예를 살펴보자.

```

for line in f:           # 파일의 모든 줄에 대해 반복한다.
    # 이번 줄에 대해 무언가를 수행한다.
    ...

```

파이썬 2에서는 파일 모드가 무엇으로 설정되어 있는지(텍스트 또는 이진)에 관계 없이 읽기 연산은 항상 8비트 문자열을 반환한다. 파이썬 3에서는 파일을 텍스트 모드로 열었을 경우 유니코드 문자열을 반환하고 이진 모드로 열었을 경우 바이트 문자열을 반환한다.

`write()` 메서드는 문자열을 파일에 쓰고 `writelines()` 메서드는 문자열 리스트를 파일에 쓴다. `write()`와 `writelines()`는 줄바꿈 문자를 알아서 추가하지 않기 때문에 출력하는 내용에 미리 포맷을 적용해놓아야 한다. 이 메서드들로 무가공(raw) 바이트 문자열을 파일에 쓸 수 있지만 파일을 이진 모드로 열었을 경우에만 가능하다.

내부적으로 각 파일 객체는 다음 읽기 연산 또는 쓰기 연산이 발생할 곳의 바이트 오프셋을 저장하는 파일 포인터를 유지한다. `tell()` 메서드는 파일 포인터의 현재 값을 긴 정수로 반환한다. `seek()` 메서드는 `offset`과 배치 규칙 `whence`를 지정해서 파일의 일부분에 임의로 접근하는 데 사용한다. `whence`가 0이면(기본 값) `seek()`는 `offset`이 파일 시작 위치에 상대적으로 지정되었다고 가정한다. `whence`가 1이면 현재 위치에 상대적으로 위치가 결정된다. `whence`가 2면 `offset`은 파일의 끝에서부터 계산된다. `seek()`는 파일 포인터의 새 값을 정수로 반환한다. 파일 포인터는 파일 자체와 연결된 것이 아니라 `open()`에 의해서 반환되는 파일 객체에 연결된다. 같은 프로그램 안에서(또는 다른 프로그램에서) 같은 파일을 한 번 이상 열 수 있다. 이때 각 열린 파일의 인스턴스는 독립적으로 조작이 가능한 자신만의 파일 포인터를 갖는다.

`fileno()` 메서드는 파일에 대한 정수 파일 기술자를 반환하며 몇몇 라이브러리 모듈에서 저수준 I/O 연산을 수행하는 데 사용된다. 예를 들어, `fcntl` 모듈은 UNIX 시스템에서 저수준 파일 제어 연산을 지원하기 위해서 파일 기술자를 사용한다.

파일 객체는 표 9.2에 나와 있는 읽기 전용 데이터 속성을 가진다.

표 9.2 파일 객체 속성

속성	설명
f.closed	파일의 상태를 나타내는 불리언 값. 파일이 열려 있을 경우 False, 닫혀 있을 경우 True
f.mode	파일 I/O 모드
f.name	open()을 사용하여 생성된 경우 파일 이름. 아니면 파일 소스를 나타내는 문자열
f.softspace	print문에서 다음 값을 출력하기 전에 공백 문자를 출력할 것인지를 나타내는 불리언 값. 파일을 흉내 내는 클래스는 반드시 이 이름에 대해 쓰기 가능한 속성을 가져야 하며 초기에는 0으로 초기화된다(파이썬 2에만 있음).
f.newlines	보편 줄바꿈 모드로 파일을 열었을 때 이 속성은 실제로 파일에서 발견된 줄바꿈 표현 방식을 담는다. 줄바꿈 문자를 발견하지 못했으면 None, 아니면 '\n', '\r' 또는 '\r\n'을 담은 문자열, 또는 발견된 모든 줄바꿈 표현을 담은 튜플.
f.encoding	파일 인코딩을 나타내는 문자열(예를 들어, 'latin-1' 또는 'utf-8'). 사용된 인코딩이 없으면 None

표준 입력, 출력과 에러

인터프리터는 표준 입력, 표준 출력, 표준 에러라고 부르는 세 가지 표준 파일 객체를 제공하며, 각각 sys 모듈에서 sys.stdin, sys.stdout, sys.stderr으로 접근할 수 있다. stdin은 인터프리터에 제공되는 입력 문자 스트림에 해당하는 파일 객체이다. stdout은 print에서 생성하는 출력을 받아들이는 파일 객체이다. stderr는 에러 메시지를 받아들이는 파일 객체이다. 대체로 stdin은 사용자 키보드에 대응되고 stdout과 stderr는 화면에 텍스트를 출력한다.

사용자와 관련된 미가공 I/O를 수행하는 데 앞 절에서 설명한 메서드들을 사용할 수 있다. 예를 들어, 다음은 표준 출력으로 쓰고 표준 입력으로부터 한 줄을 읽는 코드이다.

```
import sys
sys.stdout.write("Enter your name : ")
name = sys.stdin.readline()
```

다른 방법으로는 stdin으로부터 텍스트 한 줄을 읽고 추가로 프롬프트를 출력하기도 하는 내장 함수 raw_input(prompt)를 사용할 수 있다.

```
name = raw_input("Enter your name : ")
```

raw_input에서 읽은 줄은 줄 마지막에 있는 줄바꿈 문자를 포함하지 않는다. 이것은 줄바꿈 문자가 입력 텍스트에 포함되는 sys.stdin에서 직접 읽는 방식과는 다르다. 파이썬 3에서는 raw_input()의 이름이 input()으로 변경되었다.

키보드 인터럽트(일반적으로 Ctrl+C에 의해 발생됨)는 KeyboardInterrupt 예외를 발생시키며 예외 처리기로 잡을 수 있다. 필요할 경우 sys.stdout, sys.stdin, sys.stderr는 다른 파일 객체로 대체될 수 있다. 이 경우 print문이나 입력을 받는 함수에서는 대체된 객체를 사용하게 된다. sys.stdout의 기본 값을 복원해야 할 필요가 있는 경우에는 대체하기 전에 먼저 저장해두어야 한다. 인터프리터가 시작할 때의 sys.stdout, sys.stdin과 sys.stderr의 기본 값은 각각 sys.__stdout__, sys.__stdin__, sys.__stderr__에 들어 있다.

통합 개발 환경(IDE)을 사용할 때 sys.stdin, sys.stdout, sys.stderr이 변경되는 경우도 있다. 예를 들어, IDLE에서 파이썬을 실행하면 sys.stdin은 파일과 유사하게 작동 하지만 실제로는 개발 환경에 속하는 객체로 대체된다. 이러한 경우 read와 seek() 같은 몇몇 저수준 메서드는 사용할 수 없다.

print문

파이썬 2에서는 sys.stdout에 저장된 파일로 출력하는 데 특수한 print문을 사용한다. print는 다음과 같이 콤마로 구분되는 객체 목록을 받아들인다.

```
print "The values are", x, y, z
```

각 객체에 대해 결과 문자열 생성하기 위해서 str() 함수가 호출된다. 생성된 결과 문자열들은 최종 결과 문자열을 생성하기 위해서 하나의 공백 문자로 분리하여 합쳐진다. print문의 끝에 콤마가 나타나지 않으면 결과는 줄바꿈 문자로 종료된다. 끝에 콤마가 있을 경우에는 다음 print문은 항목을 더 출력하기 전에 공백 문자를 하나 출력한다. 출력에 사용되는 파일의 softspace 속성에 의해서 출력되는 공백 문자가 결정된다.

```
print "The values are ", x, y, z, w
# 두 개의 print문으로 동일한 텍스트를 출력한다.
print "The values are ", x, y,      # 끝의 줄바꿈 문자를 생략한다.
      print z, w                  # z 앞에 공백 문자가 출력된다.
```

포맷이 적용된 결과를 생성하려면 4장 “연산자와 표현식”에서 설명한 문자열 포맷 연산자(%)나 .format() 메서드를 사용하면 된다. 다음 예를 보자.

```
print "The values are %d %7.5f %s" % (x,y,z) # 포맷이 적용된 I/O
"The values are {0:d} {1:7.5f} {2}" .format(x,y,z)
```

file이 쓰기 가능한 파일 객체일 때 특수한 >>file 변경자와 콤마를 추가해서 print 문의 목적지를 변경할 수 있다. 다음 예를 살펴보자.

```
f = open("output","w")
print >>f, "hello world"
...
f.close()
```

print() 함수

파이썬 3에서 가장 중요한 변화 중 하나는 print문이 함수로 변했다는 점이다. 파이썬 2.6에서도 모듈에 from __future__ import print_function문을 넣어서 print를 함수로 사용할 수 있다. print() 함수는 앞 절에서 설명한 print문과 거의 동일하게 작동한다.

스페이스로 구분된 일련의 값들을 출력하려면 다음과 같이 단순히 print()에 인자로 넘겨주면 된다.

```
print("The values are", x, y, z)
```

줄 바꿈 문자를 생략하거나 변경하려면 다음 예와 같이 키워드 인수 end=ending 을 사용하면 된다. 만약 end 인수에 줄바꿈 문자가 아닌 다른 값을 지정했다면, 출력 결과를 보기 위해 sys.stdout 파일의 출력 버퍼를 비워야(flush) 할 수도 있다.

```
print("The values are", x, y, z, end="")      # 줄바꿈 문자를 생략
```

파일에 출력하려면 다음 예와 같이 키워드 인수 file=outfile을 사용하면 된다.

```
print("The values are", x, y, z, file=f) # 파일 객체 f로 돌림
```

항목을 구분하는 분리자를 변경하려면 다음 예와 같이 키워드 인수 sep=sepchr 를 사용하면 된다.

```
print("The values are", x, y, z, sep=',') # 값 사이에 콤마 넣기
```

텍스트 출력에서 변수 보간

변수 치환부를 담은 큰 텍스트를 생성해야 하는 일이 종종 있다. 펌이나 PHP 같은 스크립트 언어에서는 달러 변수 치환을 사용해서 변수를 문자열에 넣을 수 있다 (\$name, \$address 등). 파이썬에는 이 기능에 바로 대응되는 기능이 없지만 삼중 따

음표 문자열과 포맷을 적용한 I/O를 결합하여 이 기능을 흉내 낼 수 있다. 예를 들어, 다음과 같이 name, item, amount를 적절히 채우는 짧은 편지 형식을 작성할 수 있다.

```
# 참고: """ 다음의 역슬래시(\)는 첫 번째 줄에 빈 줄이 나타나지 않게 한다.  
form = """\\  
Dear %(name)s,  
  
Please send back my %(item)s or pay me $%(amount)0.2f.  
Sincerely yours,  
  
Joe Python User  
"""  
  
print form % { 'name': 'Mr. Bush',  
                'item': 'blender',  
                'amount': 50.00,  
            }
```

앞의 코드는 다음 결과를 생성한다.

```
Dear Mr. Bush,  
  
Please send back my blender or pay me $50.00.  
Sincerely yours,  
  
Joe Python User
```

format() 메서드는 좀 더 최신 기법으로서 앞의 코드에서 몇몇 부분을 더 깔끔하게 만들어준다. 다음 예를 보자.

```
form = """\\  
Dear {name},  
Please send back my {item} or pay me {amount:0.2f}.  
Sincerely yours,  
  
Joe Python User  
"""  
  
print form.format(name='Mr. Bush', item='blender', amount=50.0)
```

특정 종류의 형식을 생성하는 데는 다음과 같이 Template 문자열을 사용할 수도 있다.

```
import string  
form = string.Template("""\\  
Dear $name,  
Please send back my $item or pay me $amount.  
Sincerely yours,  
  
Joe Python User  
"""  
  
print form.substitute({'name' = 'Mr. Bush', 'item' = 'blender',
```

```
'amount' = "%0.2f" % 50.0})
```

앞의 코드에서 문자열에 있는 특수한 \$ 변수는 치환될 부분을 나타낸다. form, substitute() 메서드는 대체할 문자열을 담은 사전을 받아서 새로운 문자열을 반환한다. 여기에서 소개한 방법들은 간단한 반면 텍스트를 생성하는 가장 강력한 기법은 아니다. 보통 웹 프레임워크나 기타 대규모 애플리케이션 프레임워크에서는 제어 흐름 내장, 변수 치환, 파일 포함 기능과 기타 고급 기능을 지원하는 자신만의 템플릿 문자열 엔진을 제공한다.

출력 생성

프로그래머에게 가장 익숙한 I/O 모델은 파일을 직접 다루는 것이다. 그렇지만 생성기 함수도 연속된 소량의 데이터를 I/O 스트림으로 내보내는 데 사용할 수 있다. 이를 위해서는 yield문을 간단히 write() 또는 print문을 사용하는 것처럼 사용하면 된다. 다음 예를 살펴보자.

```
def countdown(n):
    while n > 0:
        yield "T-minus %d\n" % n
        n -= 1
    yield "Kaboom!\n"
```

이러한 식으로 출력 스트림을 생성하면 출력 스트림을 생성하는 부분이 실제로 스트림의 목적지를 지정하는 코드와 분리될 수 있기 때문에 유연성이 커진다. 앞에서 예로 든 코드의 출력을 파일 f로 보내고자 할 경우 다음과 같이 하면 된다.

```
count = countdown(5)
f.writelines(count)
```

또는 소켓 s로 보내고 싶으면 다음과 같이 하면 된다.

```
for chunk in count:
    s.sendall(chunk)
```

간단히 모든 결과를 하나의 문자열에 담고 싶으면 다음과 같이 하면 된다.

```
out = "".join(count)
```

고급 응용 프로그램에서는 자신만의 I/O 버퍼링 기능을 구현하는 데 이 기법을 사용할 수 있다. 예를 들어, 다음에서 보듯이 생성기에서는 작은 텍스트 데이터들을 내보내지만 또 다른 함수에서 더 단위가 크고 효율적인 I/O 연산을 위해 이 데이터

들을 큰 버퍼에 모으는 것을 생각해볼 수 있다.

```
chunks = []
buffered_size = 0
for chunk in count:
    chunks.append(chunk)
    buffered_size += len(chunk)
    if buffered_size >= MAXBUFFERSIZE:
        outf.write("".join(chunks))
        chunks.clear()
        buffered_size = 0
outf.write("".join(chunks))
```

출력을 파일이나 네트워크 연결로 보내는 프로그램에서 생성기를 사용하는 접근법을 사용하면 전체 결과 출력 스트림을 먼저 하나의 큰 출력 문자열이나 문자열 리스트에 모으지 않고 작은 데이터 단위로 생성되어 처리되기 때문에 메모리 사용량을 크게 줄일 수 있다. 이 접근법은 몇몇 웹 프레임워크에서 컴포넌트 사이에 통신할 때 사용되는 파이썬 웹 서비스 게이트웨이 인터페이스(WSGI: Web Services Gateway Interface)와 상호작용하는 프로그램을 작성할 때 종종 쓰인다.

유니코드 문자열 처리

I/O 처리와 관련해서 유니코드로 표현된 국제 문자를 다루는 일이 자주 있다. 유니코드 문자열을 인코딩한 것을 담은 미가공 바이트들의 문자열 `s`가 있을 때 `s.decode([encoding [,errors]])` 메서드는 이것을 적절한 유니코드 문자열로 변환한다. 유니코드 문자열 `u`를 인코딩된 바이트 문자열로 변환하려면 문자열 메서드 `u.encode([encoding [, errors]])`를 사용하면 된다. 이 두 변환 메서드 모두 유니코드 문자 값들을 바이트 문자열의 일련의 8비트 문자들로 매핑하거나 그 반대로 매핑하는 방법을 지정하는 특별한 인코딩 이름을 받는다. `encoding` 매개변수는 문자열로 지정해야 하고 수백 개 문자 인코딩 중 하나가 될 수 있다. 다음은 그 중에서 가장 널리 사용되는 인코딩을 나열한 것이다.

값	설명
'ascii'	7 비트 ASCII
'latin-1' 또는 'iso-8859-1'	ISO 8859-1 Latin-1
'cp1252'	윈도 1252 인코딩
'utf-8'	8 비트 가변 길이 인코딩
'utf-16'	16 비트 가변 길이 인코딩(리틀 엔디언 또는 빅 엔디언)
'utf-16-le'	UTF-16, 리틀 엔디언 인코딩

'utf-16-be'	UTF-16, 빅 엔디언 인코딩
'unicode-escape'	유니코드 상수 u"string"과 같은 포맷
'raw-unicode-escape'	무가공 유니코드 상수 ur"string"과 같은 포맷

기본 인코딩은 site 모듈에서 설정하고 sys.getdefaultencoding()을 사용하여 얻을 수 있다. 많은 경우 기본 인코딩은 ‘ascii’이며 이 경우 [0x00, 0x7f] 범위에 있는 ASCII 문자가 [0x00, 0x7f] 범위에 있는 유니코드 문자로 직접 매핑된다. ‘utf-8’ 또한 널리 사용된다. 자주 사용되는 인코딩과 관련된 기술적인 내용은 이 절의 나중에 설명한다.

s.decode() 메서드를 사용할 때 s는 항상 바이트 문자열로 가정한다. 이 말은 파이썬 2에서는 s가 표준 문자열이라는 것을 의미하지만 파이썬 3에서는 s가 특수한 bytes 타입이어야 한다는 것을 의미한다. 비슷하게 t.encode()의 결과도 항상 바이트 순서열이다. 호환성에 신경 쓰고 있다면 파이썬 2에서 이 메서드들이 섞여 있다는 점을 염두에 두기 바란다. 예를 들어, 파이썬 2에서 문자열은 decode()와 encode() 메서드를 모두 가지고 있는 반면에 파이썬 3에서는 문자열은 encode() 메서드만 가지고 bytes 타입은 decode() 메서드만 가진다. 파이썬 2에서 코드를 간단하게 만들려면 유니코드 문자열에 대해서는 encode()만 사용하고 바이트 문자열에 대해서는 decode()만 사용하도록 한다.

문자열을 변환할 때 변환할 수 없는 문자를 만나면 UnicodeError 예외가 발생할 수 있다. 예를 들어, U+1F28과 같은 유니코드 문자를 담은 문자열을 ‘ascii’로 인코딩하려고 시도하면 ASCII 문자 집합으로 표현하기에는 문자 값이 너무 커서 인코딩 에러가 발생한다. encode()와 decode() 메서드의 errors 매개변수는 어떻게 인코딩 에러를 처리할지를 지정한다. 다음 문자열 중 하나가 될 수 있다.

값	설명
'strict'	인코딩과 디코딩 에러가 발생하면 UnicodeError 예외를 발생시킨다.
'ignore'	유효하지 않은 문자를 무시한다.
'replace'	유효하지 않은 문자를 대체 문자로 바꾼다.(유니코드에서는 U+FFFD, 표준 문자열에서는 '?')
'backslashreplace'	유효하지 않은 문자를 파이썬 탈출 문자 순서열로 대체한다.(예를 들어, 문자 U+1234는 '\u1234'로 바꾼다)
'xmlcharrefreplace'	유효하지 않은 문자를 XML 문자 참조로 대체(예를 들어, 문자 U+1234는 'ሴ'로 바꾼다)

기본 에러 처리 방식은 ‘strict’이다.

에러 처리 정책 중 ‘xmlcharrefreplace’는 웹 페이지에서 ASCII로 인코딩된 텍스트에 국제 문자를 집어넣을 때 유용하게 쓰인다. 예를 들어, 유니코드 문자열 ‘Jalape\u00f1o’를 ‘xmlcharrefreplace’ 처리 정책을 사용해서 ASCII로 인코딩하여 출력하면 브라우저는 이것을 알아볼 수 없는 텍스트로 출력하지 않고 “Jalapeño”로 올바르게 표시한다.

골치 아픈 일을 겪지 않으려면 인코딩된 바이트 문자열과 인코딩되지 않은 문자열을 표현식에서 절대로 같이 섞어서 사용하지 않도록 한다(예를 들어 +를 사용하여 연결한다든지). 파이썬 3에서는 아예 이렇게 할 수 없게 되어 있지만 파이썬 2에서는 조용히 바이트 문자열을 기본 인코딩에 따라 유니코드 문자열로 자동으로 변환해버린다. 이 때문에 예상 못한 결과가 나오거나 불가사의한 에러 메시지가 출력되는 일이 있다. 따라서 인코딩된 문자 데이터와 인코딩 안 된 문자 데이터를 엄격하게 분리해서 다루도록 주의를 기울여야 한다.

유니코드 I/O

유니코드 문자열로 작업할 때 미가공 유니코드 데이터를 파일에 직접 쓰는 것은 불가능하다. 그 이유는 유니코드 문자는 내부적으로 다중 바이트 정수로 표현되며 이 정수를 직접 출력 스트림에 쓰게 되면 바이트 순서와 관련된 문제가 발생할 수 있기 때문이다. 예를 들어, 유니코드 문자 U+1F601를 ‘리틀 엔디안’ 포맷인 LL HH 로 쓸지 ‘빅 엔디안’ 포맷인 HH LL로 쓸지 어느 쪽으로든 정해야 한다. 또한 유니코드를 다루는 도구에서도 여러분이 사용한 인코딩에 관해서 알고 있어야 한다.

유니코드 문자열 표현 방식은 항상 유니코드 문자들을 바이트 순서열로 어떻게 나타낼 것인지를 정확하게 정의하는 인코딩 규칙에 따라서 결정된다. 유니코드 I/O를 지원하기 위해서 이전 장에서 설명한 인코딩과 디코딩 개념이 파일로 확장된다. 내장 codecs 모듈은 다양한 데이터 인코딩 방식에 따라서 바이트 지향(byte-oriented) 데이터를 유니코드 문자열 또는 그 반대로 변환하기 위한 함수들을 담고 있다.

유니코드 파일을 다루기 위한 가장 간단한 방법으로 다음과 같이 codecs.open(filename [, mode [, encoding [, errors]]]) 함수를 사용하는 방법이 있다.

```
f = codecs.open('foo.txt','r','utf-8','strict')      # 읽기
g = codecs.open('bar.txt','w','utf-8')                # 쓰기
```

이렇게 하면 유니코드 문자열을 읽거나 쓰는 데 사용할 수 있는 파일 객체가 생성된다. encoding 매개변수는 파일을 읽거나 쓸 때 데이터를 변환하기 위해 사용할 내부 문자 인코딩을 지정한다. errors 매개변수는 에러를 어떻게 처리할 것인지를 결정하며 ‘strict’, ‘ignore’, ‘replace’, ‘backslashreplace’, ‘xmlcharrefreplace’ 중 하나의 값을 가질 수 있다.

이미 파일 객체를 가지고 있으면 `codecs.EncodedFile(file, inputenc [, outputenc [, errors]])`을 사용하여 그 파일에 래퍼를 씌울 수 있다. 다음 예를 보자.

```
f = open("foo.txt", "rb")
...
fenc = codecs.EncodedFile(f,'utf-8')
```

이 경우 파일에서 읽어오는 데이터를 inputenc로 지정한 인코딩에 따라 해석하게 된다. 파일에 있는 데이터는 inputenc의 인코딩에 따라 해석되고 outputenc에 따라 쓰여진다. outputenc를 생략하면 inputenc가 기본으로 사용된다. errors는 앞에서 설명한 의미를 지닌다. 기존의 파일에 EncodedFile 래퍼를 씌우는 경우 파일을 이진 모드로 열어야 한다. 그러지 않으면 줄바꿈 변환 과정에서 인코딩이 깨질 수 있다.

유니코드 파일을 다룰 때에는 종종 파일 자체에 데이터 인코딩이 써져 있는 경우도 있다. 예를 들어, XML 파서는 문서의 인코딩을 결정하기 위해서 먼저 문자열 ‘<?xml ...>’의 처음 몇 바이트를 살펴본다. 첫 네 개 값이 3C 3F 78 6D (‘<?xm’)이면 인코딩을 UTF-8로 가정한다. 첫 네 개의 값이 00 3C 00 3F 또는 3C 00 3F 00 이면 인코딩을 각각 UTF-16 빅 엔디안 또는 UTF-16 리틀 엔디안으로 가정한다. 인코딩은 MIME 헤더나 기타 문서 엘리먼트의 속성으로 나타날 수도 있다. 다음 예를 보자.

```
<?xml ... encoding="ISO-8859-1" ... ?>
```

이와 비슷하게 유니코드 파일은 문자 인코딩 속성을 나타내는 특별한 바이트 순서 표시(BOM: byte-order marker)를 담을 수도 있다. 유니코드 문자 U+FEFF는 이러한 목적을 위해 쓰이도록 예약되어 있다. 보통 이 표시는 파일에서 첫 번째 문자로 나타난다. 프로그램에서는 이 문자를 읽어서 인코딩을 결정하기 위해 이 문자의 정렬 방식을 살펴본다(예를 들어, UTF-16-LE에 대해서는 ‘\xff\xfe’, UTF-16-BE에 대해서는 ‘\xfe\xff’). 일단 인코딩이 결정되면 BOM 문자는 버려지고 파일의 남은 부분이 처리된다. 불행히도 이렇게 BOM을 추가적으로 처리하는 일은 알아서 수행되지 않는다. 프로그램에서 여러분이 직접 해야 한다.

문서에서 인코딩을 읽고 나면 다음과 같은 코드를 사용해서 입력 파일을 인코딩 된 스트림으로 변환할 수 있다.

```
f = open("somefile","rb")
# 파일의 인코딩을 결정
...
# 파일에 적절한 인코딩 래퍼를 씌움
# BOM이 이전 문장에서 이미 제거되었다고 가정한다.
fenc = codecs.EncodedFile(f,encoding)
data = fenc.read()
```

유니코드 데이터 인코딩

표 9.3은 codecs 모듈에서 가장 흔히 사용되는 인코더들의 목록을 보여준다.

표 9.3 codecs 모듈에서 사용되는 인코드

인코더	설명
'ascii'	7 비트 ASCII
'latin-1'	또는 'iso-8859-1' ISO 8859-1 Latin-1
'cp437'	원도 437 인코딩
'cp1252'	원도 1252 인코딩
'utf-8'	8 비트 가변 길이 인코딩
'utf-16'	16 비트 가변 길이 인코딩(리틀 엔디언 또는 빅 엔디언)
'utf-16-le'	UTF-16, 리틀 엔디언 인코딩
'utf-16-be'	UTF-16, 빅 엔디언 인코딩
'unicode-escape'	유니코드 상수 u "string" 과 같은 포맷
'raw-unicode-escape'	무가공 유니코드 상수 ur "string"과 같은 포맷

이어서 각 인코더에 대해 자세히 설명한다.

'ascii' 인코딩

'ascii' 인코딩에서는 문자 값이 [0x00, 0x7f]와 [U+0000, U+007F] 사이에 한정된다. 이 범위 밖의 문자는 유효하지 않다.

'iso-8859-1', 'latin-1' 인코딩

문자는 [0x00, 0xff]와 [U+0000, U+00FF] 사이에 있는 값이 될 수 있다.

[0x00, 0x7f] 범위에 있는 값은 ASCII 문자 집합에 있는 문자에 대응된다. [0x80, 0xff] 범위의 값은 ISO-8859-1 또는 확장 ASCII 문자 집합에 있는 문자에 대응된다. [0x00, 0xff] 범위 밖에 있는 값을 가진 문자를 사용하면 에러가 발생한다.

'cp437' 인코딩

이 인코딩은 'iso-8859-1'과 비슷하지만 파이썬이 윈도 콘솔 응용 프로그램으로 실행될 때 기본 인코딩으로 사용되는 인코딩이다.* [x80, 0x9f] 범위에 있는 몇몇 문자는 레거시 DOS 응용 프로그램에서 메뉴, 창, 프레임을 보여주고자 할 때 사용되던 문자에 대응된다.

'cp1252' 인코딩

이 인코딩은 윈도에서 사용되는 'iso-8859-1'과 유사하다. 하지만 이 인코딩은 'iso-8859-1'에서 정의되지 않은 [0x80-0x9f] 범위에 있는 문자를 정의하고 있고 이 부분은 유니코드에서 다른 코드 포인트를 가진다.

'utf-8' 인코딩

UTF-8은 가변 길이 인코딩으로서 모든 유니코드 문자를 표현할 수 있게 한다. 단일 바이트는 0-127의 범위에 있는 ASCII 문자를 표현하는 데 사용된다. 다른 모든 문자는 2 바이트나 3 바이트의 다중 바이트 순서열로 표현된다. 다음은 바이트들을 인코딩하는 방식을 보여준다.

유니코드 문자	바이트 0	바이트 1	바이트 2
U+0000 - U+007F	0nnnnnnn		
U+007F - U+07FF	110nnnnn	10nnnnnn	
U+0800 - U+FFFF	1110nnnn	10nnnnnn	10nnnnnn

2 바이트 순서열에서 첫 번째 바이트는 항상 비트 순서열 110으로 시작한다. 3 바이트 순서열에서 첫 번째 바이트는 비트 순서열 1110으로 시작한다. 다중 바이트 순서열에서 나머지 데이터 바이트는 모두 비트 순서열 10으로 시작한다.

UTF-8 포맷은 최대 6 바이트의 길이를 가진 다중 바이트 순서열을 지원한다. 파이썬에서는 4 바이트 UTF-8 순서열을 대리자 쌍(surrogate pair)이라고 부르는 유니코드 문자 쌍을 인코딩하기 위해서 사용한다. 두 문자 모두 [U+D800, U+DFFF]

* (옮긴이) 한글 윈도에서는 'cp 949'가 기본 인코딩이다.

범위에 있는 값을 가지며 20 비트 문자 값의 인코딩을 위해 결합된다. 대리자 인코딩은 다음과 같이 이루어 진다. 4 바이트 순서열 11110nnn 10nnnnnn 10nmffff 10mmmmmm은 U+D800 + N, U+DC00 + M 쌍으로 인코딩되는데 여기서 N과 M은 각각 4 바이트 UTF-8 순서열에서 인코딩된 20비트 문자의 상위 10비트와 하위 10비트를 나타낸다. 5 바이트와 6 바이트 UTF-8 순서열(시작 비트 순서열이 각각 111110과 1111110)은 길이가 최대 32 비트인 문자 값을 인코딩하는 데 사용된다. 파이썬에서는 현재 이 값을 지원하지 않기 때문에 인코딩된 데이터 스트림에 나타날 경우 UnicodeError 예외가 발생한다.

UTF-8 인코딩은 오래된 소프트웨어에서도 사용할 수 있게 하는 유용한 속성을 몇 가지 갖고 있다. 첫 번째로 표준 ASCII 문자는 표준 인코딩 안에서 표현된다. 따라서 UTF-8로 인코딩된 ASCII 문자열과 전통적인 ASCII 코드를 구별할 수 없다. 두 번째로 UTF-8은 다중 바이트 문자 순서열을 위해서 NULL 바이트를 포함시키지 않았다. 그러므로 NULL로 끝나는 8-비트 문자열을 기대하는 C 라이브러리나 프로그램에 기반한 기존의 소프트웨어에서도 UTF-8 문자열을 사용할 수 있다. 마지막으로 UTF-8 인코딩은 문자열의 사전 순서를 지킨다. 즉, a와 b가 유니코드 문자열이고 a < b라면, a와 b를 UTF-8로 변경하여도 a < b가 유지된다. 그러므로 8 비트 문자열에 대해서 쓰여진 정렬 알고리즘이나 기타 순서를 정하는 알고리즘도 UTF-8에 대해서 잘 작동한다.

‘utf-16’, ‘utf-16-be’, ‘utf-16-le’ 인코딩

UTF-16은 가변 길이 16-비트 인코딩으로 유니코드 문자를 16 비트 값으로 쓴다. 바이트 순서를 따로 지정하지 않는 이상 빅 엔디안으로 가정한다. 추가로 U+FEFF인 바이트 순서 표시를 사용해서 UTF-16 바이트 스트림에서 바이트 순서를 명시적으로 나타낼 수 있다. 빅 엔디안 인코딩에서 U+FEFF는 길이가 0인 끊어지지 않는 공백(nonbreaking space)을 나타내는 유니코드 문자인 반면에 그 반대 값인 U+FFFE는 적법하지 않은 유니코드 문자이다. 그러므로 인코더는 데이터 스트림의 바이트 순서를 결정하기 위해서 바이트 순서열 FE FF 또는 FF FE를 사용할 수 있다. 유니코드 데이터를 읽을 때 파이썬은 최종 유니코드 문자열에서 바이트 순서 표시 문자를 제거한다.

‘utf-16-be’ 인코딩은 명시적으로 UTF-16 빅 엔디안 인코딩을 선택한다. ‘utf-16-le’는 명시적으로 UTF-16 리틀 엔디안 인코딩을 선택한다.

16 비트보다 큰 문자 값을 지원하기 위해 UTF-16을 확장한 버전도 있지만 현재 지원되는 확장 버전은 없다.

'unicode-escape'와 'raw-unicode-escape' 인코딩

이 인코딩 방식은 유니코드 문자열을 파이썬 유니코드 문자열 상수나 유니코드 무가공 문자열 상수에서 사용되는 것과 동일한 포맷으로 변경한다. 다음 예를 보자.

```
s = u'\u14a8\u0345\u2a34'  
t = s.encode('unicode-escape')           #t = '\u14a8\u0345\u2a34'
```

유니코드 문자 속성

I/O를 수행하는 것 말고도 유니코드를 사용하는 프로그램에서는 유니코드 문자에 대해서 대문자, 숫자, 공백인지 여부 등 다양한 속성을 검사할 일이 있다. unicodedata 모듈을 사용하면 문자 속성 데이터베이스에 접근할 수 있다. 일반 문자 속성은 unicodedata.category(c) 함수로 얻을 수 있다. 예를 들어, unicodedata.category(u "A")는 주어진 문자가 대문자라는 것을 의미하는 'Lu'를 반환한다.

유니코드 문자를 다룰 때에는 한 유니코드 문자에 대해 여러 가지 다른 표현이 있다는 것을 고려해야 한다. 예를 들어, 문자 U+00F1 (ñ)는 온전히 하나로 구성된 문자 U+00F1로 표현하거나 다중 문자 순서열인 U+006e U+0303 (n, ~)로 분해할 수 있다. 일관된 방식으로 유니코드 문자열을 처리하고 싶다면 일관된 문자 표현 방식을 보장하는 unicodedata.normalize() 함수를 사용하도록 하라. 예를 들어, unicodedata.normalize('NFC', s)는 s에 있는 모든 문자를 조합 문자들의 순서열이 아니라 온전한 하나의 문자로 표현한다.

유니코드 문자 데이터베이스와 unicodedata 모듈과 관련된 자세한 내용은 16장에서 다룬다.

객체 영속화와 pickle 모듈

객체의 내용을 파일에 저장하거나 복원해야 할 일이 종종 있다. 이를 위한 한 가지 방법으로 특수한 포맷에 따라서 파일에서 데이터를 읽고 쓰는 함수 한 쌍을 작성하는 것을 생각해볼 수 있다. 아니면 pickle과 shelve 모듈을 사용할 수도 있다.

pickle 모듈은 객체를 파일에 썼다가 나중에 복원할 수 있도록 객체를 바이트의

스트림으로 직렬화한다. pickle의 인터페이스는 dump()와 load() 연산 두 개로 간단히 구성되어 있다. 예를 들어, 다음 코드는 파일에 객체를 쓴다.

```
import pickle
obj = SomeObject()
f = open(filename,'wb')
pickle.dump(obj,f)      # 객체를 f에 저장
f.close()
```

객체를 복원하려면 다음 코드처럼 하면 된다.

```
import pickle
f = open(filename,'rb')
obj = pickle.load(f)      # 객체 복원
f.close()
```

객체 순서열을 저장하려면 하나씩 차례대로 dump()를 호출하면 된다. 다시 복원하려면 비슷한 순서대로 load()를 실행해주면 된다.

shelve 모듈은 pickle 모듈과 유사하지만 객체들을 사전 같은 데이터베이스에 저장한다.

```
import shelve
obj = SomeObject()
db = shelve.open('filename')
db['key'] = obj      # shelve 열기
...                  # shelve에 객체 저장
obj = db['key']      # 추출
db.close()           # shelve 닫기
```

비록 shelve에 의해 생성되는 객체는 사전처럼 보이지만 사전과 달리 몇 가지 제약을 가진다. 첫 번째로 키는 반드시 문자열이어야 한다. 두 번째로 shelve에 저장되는 값은 반드시 pickle과 호환이 되어야 한다. 파이썬 객체 대부분을 저장할 수 있지만 파일이나 네트워크 연결 같이 내부 상태를 저장하는 특별한 목적에 사용되는 객체는 shelve를 통해 저장하거나 복원할 수 없다.

pickle에서 사용하는 데이터 포맷은 파이썬에 특화되어 있다. 하지만, 이 포맷은 파이썬 버전이 바뀌면서 점차 진화해왔다. dump(obj, file, protocol) 연산에 추가 프로토콜 매개변수를 주어서 저장 프로토콜을 선택할 수 있다. 기본으로 프로토콜 0이 사용된다. 이는 가장 오래된 pickle 데이터 포맷으로 거의 모든 파이썬 버전에서 사용할 수 있도록 객체를 저장한다. 하지만, 이 포맷은 좀 더 최신 파이썬에서 지원하는 slot 같은 사용자 정의 클래스 관련 기능과 호환되지 않는다. 프로토콜 1과 2는 좀 더 효율적인 이진 데이터 표현을 위해 사용된다. 이 프로토콜들을 사용하려

면 다음과 같이 하면 된다.

```
import pickle
obj = SomeObject( )
f = open(filename,'wb')
pickle.dump(obj,f,2)                                # 프로토콜 2를 사용하여 저장
pickle.dump(obj,f,pickle.HIGHEST_PROTOCOL)          # 가장 최신 프로토콜을 사용
f.close( )
```

`load()`를 사용하여 객체를 복원할 때는 프로토콜을 지정할 필요가 없다. 내부적으로 사용되는 프로토콜이 파일 자체에 저장되어 있기 때문이다.

비슷하게 `shelve`에서도 다음과 같이 특정한 pickle 프로토콜을 사용해서 파일 객체를 저장하게 할 수 있다.

```
import shelve
db = shelve.open(filename,protocol=2)
...
```

보통 사용자 정의 객체에서 pickle 또는 `shelve`를 사용할 때 추가로 해야 하는 일은 없다. 그러나 특수 메서드인 `__getstate__()`와 `__setstate__()`를 사용해서 피클링 과정을 도울 수 있다. `__getstate__()` 메서드는 정의되어 있을 경우 객체의 상태를 표현하는 값을 생성하기 위해서 호출된다. `__getstate__()`가 반환하는 값은 보통 문자열, 튜플, 리스트, 사전이어야 한다. `__setstate__()`는 언피클링하는 동안 값을 넘겨 받아서 객체의 상태를 복원할 수 있어야 한다. 다음 예는 내부적으로 네트워크 연결을 수행하는 객체에 대해서 이 메서드들을 어떻게 사용하는지를 보여준다. 실제 연결 자체가 피클링될 수는 없지만 나중에 언피클링될 때 연결을 다시 생성하는 데 충분한 정보를 저장하면 된다.

```
import socket
class Client(object):
    def __init__(self,addr):
        self.server_addr = addr
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(addr)
    def __getstate__(self):
        return self.server_addr
    def __setstate__(self,value):
        self.server_addr = value
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(self.server_addr)
```

pickle에 의해 사용되는 데이터 포맷은 파일에 특화되어 있기 때문에 다른 프로그램 언어로 작성된 응용 프로그램 사이에 데이터를 주고받는 용도로는 pickle을 사용하지 않는 것이 좋다. 그리고 보안 문제가 있기 때문에 신뢰하지 않는 소스에

서 온 피클링된 데이터를 처리하지 않도록 한다(똑똑한 공격자라면 pickle의 데이터 포맷을 조작해서 언피클링을 수행할 때 임의의 시스템 명령을 실행하게 만들 수도 있다).

pickle과 shelve 모듈은 커스터마이즈할 수 있는 더 많은 기능과 고급 옵션을 제공한다. 자세한 내용은 13장을 참고하라.

10장

P y t h o n E s s e n t i a l R e f e r e n c e

실행 환경

이 장에서는 파이썬 프로그램의 실행 환경에 관해서 설명한다. 프로그램 시작, 설정, 종료 등과 관련해서 런타임에 인터프리터가 어떻게 작동하는지 알아본다.

인터프리터 옵션과 환경

인터프리터는 런타임 작동 방식과 환경을 제어하기 위한 옵션을 여러 개 제공한다. 다음과 같이 명령줄에서 인터프리터에 옵션을 주면 된다.

```
python [options] [-c cmd | filename | - ] [args]
```

다음 표는 가장 자주 사용되는 명령줄 옵션을 나열한 것이다.

표 10.1 인터프리터 명령줄 인수

옵션	설명
-3	파이썬 3에서 제거되거나 변경될 기능에 대한 경고를 활성화함
-B	import를 수행할 때 .pyc나 .pyo 파일이 생성되는 것을 막음
-E	환경 변수를 무시
-h	사용 가능한 모든 명령줄 옵션을 출력
-i	프로그램을 실행한 후 대화식 모드로 들어감
-m module	라이브러리 모듈 module을 스크립트로서 실행
-O	최적화 모드
-OO	최적화 모드에 더하여 .pyo 파일을 생성할 때 문서화 문자열 제거

-Q arg	파이썬 2에서 나누기 연산자의 작동 방식을 지정. -Qold(기본 값), -Qnew, -Qwarn, -Qwarnall 중 하나
-s	사용자 사이트 디렉터리를 sys.path에 추가하는 것을 막음
-S	site 초기화 모듈 포함을 막음
-t	일관성 없는 탭 사용에 대해 경고를 출력
-tt	일관성 없게 탭을 사용할 경우 TabError 예외가 발생
-u	버퍼링하지 않는 이진 stdout과 stdin
-U	유니코드 상수. 모든 문자열 상수이 유니코드로 취급됨(파이썬 2에만 있음)
-v	상세 출력 모드. import문에 대한 추적 정보를 제공
-V	버전 번호를 출력하고 종료
-x	소스 프로그램의 첫 번째 줄을 건너뜀
-c cmd	cmd를 문자열로서 실행

-i 옵션은 프로그램이 종료되자마자 대화식 세션을 시작하게 하며, 디버깅할 때 유용하게 쓰인다. -m 옵션은 메인 스크립트를 실행하기 전에 주어진 라이브러리 모듈을 __main__ 모듈 안에서 스크립트로서 실행한다. -O와 -OO 옵션은 바이트 컴파일된 파일에 최적화를 수행하며 8장에서 설명했다. -S 옵션은 나중에 ‘사이트 설정 파일’ 절에서 설명한 site 초기화 모듈을 제외한다. -t, -tt, -v 옵션은 추가적으로 경고와 디버깅 정보를 출력한다. -x는 첫 번째 줄이 유효한 파이썬 문장이 아닌 경우 첫 번째 줄을 무시한다(예를 들어, 첫 번째 줄에서 파이썬 인터프리터를 실행하는 경우).

인터프리터 옵션을 지정한 다음에 프로그램의 이름을 써준다. 프로그램의 이름을 지정하지 않거나 파일 이름 대신 하이픈(-) 문자를 지정하면, 인터프리터가 프로그램을 표준 입력에서 읽어 들인다. 표준 입력이 대화식 터미널인 경우 배너와 프롬프트가 출력된다. 프로그램 이름을 입력한 경우에는 해당 파일을 열어서 파일 끝 표시에 도달할 때까지 문장을 실행한다. -c cmd 옵션은 명령줄 옵션을 통해 짧은 프로그램을 실행한다. 예를 들어, python -c “print('hello world')”.

프로그램 이름 또는 하이픈 문자(-) 이후에 나오는 명령줄 옵션은 프로그램에서 sys.argv에 저장된다. 여기에 관해서는 9장의 ‘명령줄 옵션 읽기’와 ‘환경 변수’ 절에서 자세히 설명했다.

인터프리터는 다음의 환경 변수들을 참조한다.

표 10.2 인터프리터 환경 변수

환경 변수	설명
PYTHONPATH	콜론으로 구분된 모듈 검색 경로
PYTHONSTARTUP	대화식 모드를 시작할 때 실행되는 파일
PYTHONHOME	파이썬이 설치된 위치
PYTHONINSPECT	-o 옵션을 활성화
PYTHONUNBUFFERED	-u 옵션을 활성화
PYTHONIOENCODING	stdin, stdout, stderr에 대한 인코딩과 에러 처리 방식을 지정. “utf-8” 또는 “utf-8:ignore” 같이 “encoding[:errors]” 형태의 문자열
PYTHONDONTWRITEBYTECODE	-B 옵션을 활성화
PYTHONOPTIMIZE	-O 옵션을 활성화
PYTHONNOUSERSITE	-s 옵션을 활성화
PYTHONVERBOSE	-v 옵션을 활성화
PYTHONUSERBASE	사용자별 사이트 패키지의 루트 디렉터리
PYTHONCASEOK	import에서 모듈 이름을 검색할 때 대소문자 구별을 하지 않도록 함

PYTHONPATH는 9장에서 설명한 것처럼 sys.path의 맨 앞에 추가될 모듈 검색 경로를 지정한다. PYTHONSTARTUP은 인터프리터가 대화식 모드로 실행될 때 실행할 파일을 지정한다. PYTHONHOME 변수는 파이썬이 설치된 디렉터리를 설정하는 데 사용한다. 일반적으로 라이브러리나 확장 기능이 설치되는 디렉터리인 site-packages 디렉터리를 찾는 방법을 파이썬에서 알고 있기 때문에 보통 이 변수를 설정할 필요가 없다. 이 변수를 /usr/local 같은 단일 디렉터리로 지정하면 인터프리터가 해당 위치에서 모든 파일을 찾는다. /usr/local:/usr/local/sparc-solaris-2.6 같이 두 디렉터리를 지정하면 인터프리터는 플랫폼에 독립적인 파일은 첫 번째 디렉터리에서, 플랫폼에 종속적인 파일은 두 번째 디렉터리에서 찾는다. 해당 위치에 파이썬이 설치되어 있지 않으면 PYTHONHOME을 설정해도 아무런 효과가 없다.

PYTHONIOENCODING 환경 변수는 표준 I/O 스트림의 인코딩과 에러 처리 방식을 설정하는 데 사용되며 파이썬 3에서 유용하게 쓰인다. 파이썬 3에서는 대화식 인터프리터 프롬프트에서 유니코드를 그대로 출력한다. 이 때문에 데이터를 살펴볼 때 예상하지 못한 예외가 발생할 수 있다. 다음 예를 보자.

```
>>> a = 'Jalape\xf1o'
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/lib/python3.0/io.py", line 1486, in write
    b = encoder.encode(s)
  File "/tmp/lib/python3.0/encodings/ascii.py", line 22, in encode
    return codecs.ascii_encode(input, self.errors)[0]
UnicodeEncodeError: 'ascii' codec can't encode character '\xf1' in
position 7:
ordinal not in range(128)
>>>
```

이 문제를 해결하려면 환경 변수 PYTHONIOENCODING을 ‘ascii: backslashreplace’나 ‘utf-8’ 같은 것으로 설정해주면 된다. 이제 다음 결과를 얻게 된다.

```
>>> a = 'Jalape\xf1o'
>>> a
'Jalape\xf1o'
>>>
```

윈도에서는 PYTHONPATH 같은 환경 변수의 값을 HKEY_LOCAL_MACHINE/Software/Python에 있는 레지스트리 항목에서도 찾는다.

대화식 세션

인터프리터에 프로그램 이름을 지정하지 않았고 또 인터프리터의 표준 입력이 대화식 터미널일 경우 파이썬은 대화식 모드에서 시작된다. 이 모드에서는 배너 메시지가 출력되고 프롬프트가 나타난다. 또한 PYTHONSTARTUP 환경 변수로 지정한 스크립트가 실행된다(설정된 경우에만). 이 스크립트는 입력 프로그램의 일부인 것처럼 실행된다(즉, import문으로 로드되는 것이 아니라). 이 스크립트로 .pythonrc 같은 설정 파일을 읽는 등의 작업을 수행할 수 있다.

대화식 모드에 있을 때 두 종류의 프롬프트를 볼 수 있다. >>> 프롬프트는 새로운 문장이 시작할 때 나타나고 ... 프롬프트는 문장이 이어질 때 나타난다. 다음 예를 보자.

```
>>> for i in range(0,4):
...     print i,
...
0 1 2 3
>>>
```

sys.ps1과 sys.ps2의 값을 수정해서 이 프롬프트의 모양을 변경할 수 있다.

어떤 시스템에서는 파이썬이 GNU readline 라이브러리를 사용하도록 컴파일된 경우도 있다. 이 경우 대화식 모드에서 이전에 실행하였던 명령 보기나 자동 완성 같은 기능을 추가로 사용할 수 있다.

대화식 모드에서 명령을 실행하면 해당 결과에 내장 함수 repr()를 적용한 것이 출력된다. 결과 출력을 담당하는 함수를 sys.displayhook 변수에 설정해주면 이 기본 출력 방식을 변경할 수 있다. 다음은 긴 결과를 잘라내는 예를 보여준다.

```
>>> def my_display(x):
...     r = repr(x)
...     if len(r) > 40: print(r[:40]+"..."+r[-1])
...     else: print(r)
>>> sys.displayhook = my_display
>>> 3+4
7
>>> range(100000)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1...]
```

마지막으로, 대화식 모드에서 최종 연산의 결과는 특수한 변수인 _에 저장된다. 이 변수는 이어지는 연산에서 이전 결과를 사용해야 할 일이 있을 때 쓰인다. 다음 예를 보자.

```
>>> 7 + 3
10
>>> _ + 2
12
>>>
```

_ 변수를 설정하는 일은 앞에서 본 displayhook() 함수에서 이루어진다. displayhook()을 재정의하는 경우 이 기능을 유지시키려면 적절히 _를 설정해주어야 한다.

파이썬 응용 프로그램 실행

인터프리터를 직접 실행할 필요 없이 프로그램에서 인터프리터를 자동으로 실행해 주면 좋을 것이다. 유닉스에서 이렇게 하려면 프로그램에 실행 권한을 주고 첫 번째 줄을 다음과 같이 작성하면 된다.

```
#!/usr/bin/env python
# 여기서부터 파이썬 코드
print "Hello world"
...
```

윈도에서는 .py, .pyw, .wpy, .pyc, .pyo 파일을 더블 클릭하면 인터프리터가 자동으로 실행된다. 보통 .pyw로 끝나는 파일이 아닌 한(이 경우 프로그램이 조용히 실행된다) 콘솔 창에서 프로그램이 실행된다. 인터프리터에 옵션을 주고 싶을 때는 .bat 파일로 파이썬을 실행하면 된다. 예를 들어, 다음 .bat 파일은 명령 프롬프트에 전달된 옵션을 인터프리터에 넘겨주면서 파이썬 스크립트를 실행한다.

```
:: foo.bat
:: (있을 경우) 제공된 명령줄 옵션을 넘겨주면서 foo.py 스크립트를 실행한다.
c:\python26\python.exe c:\pythonscripts\foo.py %*
```

사이트 설정 파일

보통 파이썬 설치본에는 써드파티 모듈과 패키지가 포함되어 있다. 인터프리터는 이러한 모듈과 패키지를 설정하기 위해 site 모듈을 먼저 임포트한다. site 모듈은 패키지 파일을 찾고 모듈 검색 경로인 sys.path에 적절히 디렉터리를 추가하는 일을 수행한다. 또한 site 모듈은 유니코드 문자열 변환을 위한 기본 인코딩 값을 설정하는 일도 담당한다.

site 모듈은 먼저 다음에 나온 것처럼 sys.prefix와 sys.exec_prefix 값을 사용해서 디렉터리 이름 목록을 생성한다.

```
[ sys.prefix,           # 윈도에서만
  sys.exec_prefix,     # 윈도에서만
  sys.prefix + 'lib/pythonvers/site-packages', *
  sys.prefix + 'lib/site-python',
  sys.exec_prefix + 'lib/pythonvers/site-packages',
  sys.exec_prefix + 'lib/site-python' ]
```

또한 활성화된 경우, 사용자별 사이트 패키지 디렉터리도 이 목록에 추가한다(다음 절에서 설명).

다음으로 이 목록에 있는 각 디렉터리에 대해 해당 디렉터리가 있는지를 검사한다. 디렉터리가 있으면 sys.path 변수에 추가한다. 이어서 각 디렉터리가 경로 설정 파일(.pth로 끝나는 파일)을 담고 있는지를 검사한다. 경로 설정 파일에는 디렉터리, zip 파일, .egg 파일 등을 상대적인 경로로 기술한다. 다음 예를 보자.

```
# foo 패키지 설정 파일 'foo.pth'
foo
bar
```

* (옮긴이) pythonvers는 'python 2.6' 또는 'python 3.1' 같은 문자열로 치환된다.

경로 설정 파일에서 각 디렉터리는 별개 줄로 나와야 한다. 주석이나 빈 줄은 무시된다. site 모듈은 이 파일을 로드하고 각 디렉터리가 존재하는지를 검사한다. 만약 디렉터리가 있으면 sys.path에 추가한다. 중복된 항목은 한 번만 추가한다.

모든 경로가 sys.path에 추가된 후에는 sitecustomize라는 이름의 모듈을 임포트하려고 시도한다. 이 모듈은 추가적으로(그리고 임의로) 사이트와 관련된 설정을 변경하는 작업을 수행한다. sitecustomize 모듈의 임포트가 ImportError를 내면서 실패하면 에러는 조용히 무시된다. sys.path에 사용자 디렉터리가 추가되기 전에 sitecustomize 모듈 임포트가 수행된다. 따라서, 이 모듈 파일을 사용자 디렉터리에 두면 아무런 효과가 없다.

site 모듈은 기본 유니코드 인코딩도 설정한다. 기본으로 이 인코딩은 ‘ascii’로 설정된다. sitecustomize.py 파일에서 ‘utf-8’ 같은 새 인코딩 인수로 sys.setdefaultencoding()을 호출해주면 이 인코딩을 변경할 수 있다. 한번 시도해볼 생각이 있다면 site의 소스 코드를 변경해서 현재 머신의 로케일 설정에 기초해서 인코딩을 자동으로 설정하는 코드를 작성해보기 바란다.

사용자별 사이트 패키지

보통 써드파티 모듈은 모든 사용자가 접근할 수 있는 곳에 설치된다. 하지만, 개별 사용자도 모듈과 패키지를 사용자별 사이트 디렉터리에 설치할 수 있다. 유닉스와 맥킨토시에서는 ~/.local 디렉터리 아래에 있는 ~/.local/lib/python2.6/site-packages 같은 곳에서 사용자별 사이트 디렉터리를 찾을 수 있다. 윈도에서는 이 디렉터리가 %APPDATA% 환경 변수가 가리키는 디렉터리 아래에 있다. 보통 %APPDATA%는 C:\Documents and Settings\David Beazley\Application Data와 유사한 디렉터리를 가리키게 된다. 이 디렉터리 아래에서 ‘Python\Python26\site-packages’ 디렉터리를 발견할 수 있을 것이다.

여러분만의 파이썬 모듈이나 패키지를 작성하였고 이를 라이브러리로 사용하려는 경우 모듈이나 패키지를 사용자별 사이트 디렉터리에 넣어두면 된다. 써드파티 모듈을 설치하는 경우에도 setup.py에 –user 옵션을 주면 사용자별 사이트 디렉터리에 설치할 수 있다. 예를 들어, python setup.py install –user라고 입력하면 된다.

미래 기능 활성화

예전 버전의 파이썬과 호환성 문제가 있을 수 있는 새로운 언어 기능은 파이썬의 새로운 릴리스에서 보통 비활성화되어 있다. 이 기능을 활성화하려면 `from __future__ import feature` 문을 사용하면 된다. 다음 예를 보자.

```
# 새로운 나누기 작동 방식을 활성화한다.
from __future__ import division
```

이 문장은 모듈이나 프로그램에서 가장 앞에 나와야 한다. 또한 `__future__ import`의 유효 범위는 이 문장이 사용된 모듈에 국한된다. 따라서 미래 기능을 임포트한다고 하더라도 파이썬 라이브러리 모듈의 작동 방식이 변경되지 않으며, 인터프리터의 예전 작동 방식에 의존하는 오래된 코드의 작동 방식도 변경되지 않는다.

현재 `__future__` 모듈에는 다음 기능이 정의되어 있다.

표 10.3 `__future__` 모듈에 있는 기능 이름

기능 이름	설명
<code>nested_scopes</code>	함수에서 중첩 유효 범위 지원. 파이썬 2.1에서 처음 도입되었고 파이썬 2.2에서 기본으로 지원됨.
<code>generators</code>	생성기 지원. 파이썬 2.2에서 처음으로 도입되었고 파이썬 2.3에서 기본으로 지원됨.
<code>division</code>	정수 나누기가 부동 소수점 수를 반환하도록 나누기 연산의 작동 방식을 변경. 예를 들어, $1/4$ 은 0 대신 0.25를 반환한다. 파이썬 2.2에서 도입되었고 파이썬 2.6에서도 여전히 옵션인 기능이다. 파이썬 3.0에서는 기본이 됨.
<code>absolute_import</code>	패키지 상대적인 임포트의 작동 방식을 변경. 현재 패키지의 하위 모듈에서 <code>import string</code> 같은 import문을 사용할 경우 먼저 패키지의 현재 디렉터리에서 찾고 다음으로 <code>sys.path</code> 에서 찾는다. 하지만, 이렇게 되면 패키지에서 이름 충돌이 일어날 때 표준 라이브러리에 있는 모듈을 로드할 수 없다. 이 기능이 활성화되면 import문은 항상 절대적인 임포트를 수행한다. 따라서, <code>import string</code> 은 항상 표준 라이브러리에 있는 <code>string</code> 모듈을 임포트한다. 파이썬 2.5에서 처음 도입되었고 파이썬 2.6에서 여전히 비활성화되어 있다. 파이썬 3.0에서는 기본으로 활성화된다.
<code>with_statement</code>	컨텍스트 관리자와 <code>with</code> 문을 지원. 파이썬 2.5에서 도입되었고 파이썬 2.6에서 기본으로 지원됨.
<code>print_function</code>	<code>print</code> 문 대신 파이썬 3.0의 <code>print()</code> 함수를 사용한다. 파이썬 2.6에서 처음 도입되었고 파이썬 3.0에서 기본으로 활성화된다.

`__future__`에서는 어떤 기능 이름도 제거된 적이 없다. 따라서 나중 파이썬 버전에서 어떤 기능이 기본으로 활성화된다 하더라도, 이 기능을 사용하는 기존 코드는 문제가 발생하지 않는다.

프로그램 종료

프로그램에서 더 이상 실행할 문장이 없거나, 잡히지 않은 SystemExit 예외가 발생했거나(sys.exit()에 의해서 발생됨), 인터프리터가 SIGTERM이나 SIGHUP 시그널을 받은 경우(유닉스에서) 프로그램은 종료된다. 인터프리터는 프로그램이 종료될 때 현재 알고 있는 모든 네임스페이스에 있는 객체의 참조 횟수를 감소시킨다(각 네임스페이스는 파괴된다). 객체의 참조 횟수가 영에 도달하면 그 객체는 파괴되고 객체의 __del__() 메서드가 호출된다.

어떤 경우 프로그램이 종료될 때 __del__() 메서드가 호출되지 않을 수도 있다. 객체들 사이에 순환 참조가 있을 때 이런 일이 발생할 수 있다(객체가 생성되었지만 알려진 네임스페이스에서 접근할 수 없다). 프로그램을 실행하는 도중에 사용되지 않는 순환 참조를 파이썬의 쓰레기 수집기에서 회수하기는 하지만, 프로그램이 종료될 때는 보통 쓰레기 수집기가 호출되지 않는다.

프로그램이 종료될 때 __del__()이 호출되리라는 보장이 없기 때문에, 열린 파일이라든지 네트워크 연결 같은 객체는 직접 정리하는 것이 좋다. 이를 위해서는 사용자 정의 객체에 특별한 청소 메서드를 추가해주면 된다(예를 들어, close() 같은 메서드). 다음과 같이 atexit 모듈을 사용해서 종료 함수를 등록하는 방법도 있다.

```
import atexit
connection = open_connection("deaddot.com")

def cleanup():
    print "Going away..."
    close_connection(connection)

atexit.register(cleanup)
```

쓰레기 수집기를 다음과 같이 호출할 수도 있다.

```
import atexit, gc
atexit.register(gc.collect)
```

마지막으로, 프로그램이 종료될 때 어떤 객체의 __del__ 메서드에서 다른 모듈에 정의된 전역 데이터나 메서드에 접근하려는 경우 문제가 발생할 수 있다. 접근하려는 객체가 이미 파괴되었을 수 있기 때문에 NameError 예외가 발생할 수 있고 그런 경우 다음과 같은 에러 메시지가 출력된다.

```
Exception exceptions.NameError: 'c' in <method Bar.__del__
of Bar instance at c0310> ignored
```

이런 일이 발생했다면 `__del__` 메서드가 도중에 실행이 중지되었다고 생각하면 된다. 이 말은 중요한 연산이 실패했을 수 있다는 말이기도 하다(서버 연결을 깨끗하게 닫는 등). 이런 일이 문제가 될 수 있다면 프로그램 종료 때 인터프리터의 객체를 청소 작업에 의존하지 말고 직접 종료 과정을 수행하는 것이 좋다. `NameError` 예외가 발생하는 것을 막으려면 `__del__()` 메서드 선언부에 다음과 같이 기본 인수를 설정하면 된다.

```
import foo
class Bar(object):
    def __del__(self, foo=foo):
        foo.bar() # 모듈 foo에 있는 무언가를 사용한다.
```

어떤 경우에는 아무런 청소 작업도 하지 않고 프로그램을 종료하는 것이 나을 수도 있다. 이렇게 하고 싶을 때는 `os._exit(상태)`를 호출하면 된다. 이 함수는 파이썬 인터프리터 프로세스를 바로 종료시키는 저수준 `exit()` 시스템 호출에 대한 인터페이스 역할을 수행한다. 이 함수를 호출하면 프로그램이 추가적인 처리나 청소 작업 없이 즉시 종료된다.

11장

Python Essential Reference

테스트, 디버깅, 프로파일링과 튜닝

C 또는 Java 언어로 작성된 프로그램과 달리, 파이썬 프로그램은 실행 프로그램을 생성하는 컴파일러에 의해 처리되지 않는다. 다른 언어에서 컴파일러는 인수 개수를 틀리게 하여 함수를 호출하거나 변수에 부적절한 값을 할당하는(즉, 타입 검사) 등의 프로그래밍 에러를 잡아내는 일차 방어선의 역할을 수행한다. 하지만 파이썬에서는 프로그램이 실행되기 전까지 이러한 검사가 수행되지 않는다. 이 때문에 여러분은 프로그램을 실행하고 테스트해보기 전까지는 프로그램이 제대로 작동하지 여부를 알지 못한다. 모든 가능한 내부 제어 흐름을 다 실행해보지 않고서는 언제 숨어 있던 에러가 나타날지 모른다(이것을 운이 좋다고 해야 할지는 모르겠지만 보통 이런 일은 제품을 전달하고 나서 며칠이 지난 후에야 발생한다).

이러한 문제를 해결하기 위해 이 장에서는 테스트, 디버깅, 프로파일링을 수행하는 데 사용할 수 있는 기술과 모듈에 관해서 다룬다. 그리고 이 장의 끝에서는 파이썬 코드를 최적화하기 위한 몇 가지 전략을 논의한다.

문서화 문자열과 doctest 모듈

함수, 클래스, 모듈의 첫 번째 줄이 문자열이면 이를 문서화 문자열(documentation string)이라고 부른다. 문서화 문자열은 파이썬 소프트웨어 개발 도구에 유용한 정보를 제공하기 때문에 항상 문서화 문자열을 추가해주는 것이 좋다. 예를 들어, help() 명령이 문서화 문자열을 참고하고 파이썬 IDE도 문서화 문자열을 사용한다. 보통 프로그래머들이 대화식 셸에서 이것저것 실험해볼 때 문서화 문자열을 들

여다보는 경우가 있으므로 종종 문자화 문자열에 대화식 셸에서 실행해볼 수 있는 간단한 예를 포함시키곤 한다. 다음 예를 살펴보자.

```
# splitter.py
def split(line, types=None, delimiter=None):
    """텍스트 줄을 분할하고 추가적으로 타입 변환을 수행한다.
    예를 들어:
    >>> split('GOOG 100 490.50')
    ['GOOG', '100', '490.50']
    >>> split('GOOG 100 490.50',[str, int, float])
    ['GOOG', 100, 490.5]
    >>>
    기본으로 공백을 기준으로 분할되지만 delimiter 키워드 인수를 통해 다른 구분자를 선택할 수도 있다.

    >>> split('GOOG,100,490.50',delimiter=',')
    ['GOOG', '100', '490.50']
    >>>
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

문서화 문자열을 작성할 때 실제 함수 구현과 문서화 문자열 내용이 일치하지 않는 경우가 흔히 발생한다. 예를 들어, 함수를 수정하고 문서화 문자열을 수정하는 일을 잊어버릴 수 있다.

이 문제를 해결하려면 doctest 모듈을 사용하면 된다. doctest는 문서화 문자열을 수집하고 여기서 대화식 세션 텍스트를 검색하여 일련의 테스트로서 실행한다. doctest를 사용할 때는 보통 테스트용 모듈을 따로 만든다. 예를 들어, 앞에 나온 함수가 splitter.py에 들어 있다고 할 때 다음과 같이 테스트용 testsplitter.py 파일을 생성할 수 있다.

```
# testsplitter.py
import splitter
import doctest

nfail, ntests = doctest.testmod(splitter)
```

이 코드에서 doctest.testmod(module)를 호출하면 지정된 모듈에 대해서 테스트가 수행되고 실패 횟수와 총 실행된 테스트 개수가 반환된다. 모든 테스트가 통과하면 아무런 결과도 출력되지 않는다. 그렇지 않으면 기대한 결과와 얻은 결과의 차이를 보여주는 실패 보고서가 출력된다. 테스트 결과를 더 자세하게 보고 싶으면

testmod(module, verbose=True)를 사용하면 된다.

이렇게 테스트용 파일을 별개로 생성하는 대신 파일 끝에 다음과 같은 코드를 두어 라이브러리 모듈이 직접 자신을 테스트하게 할 수도 있다.

```
...
if __name__ == '__main__':
    # 스스로를 테스트
    import doctest
    doctest.testmod()
```

이렇게 하면 파일이 메인 프로그램으로서 실행될 때 문서화 문자열을 사용해서 테스트가 수행된다. 이 파일이 import로 로드된 경우 테스트는 무시된다.

doctest는 함수의 실행 결과가 대화식 인터프리터에서 얻게 되는 결과와 문자 그대로 정확히 일치할 것을 기대한다. 그 결과, 공백이나 수치 정밀도와 관련해서 주의를 기울여야 한다. 예로서 다음 함수를 보자.

```
def half(x):
    """x를 반으로 나눈다. 예를 들면,
    >>> half(6.8)
    3.4
    >>>
    return x/2
```

이 함수에 대해 doctest를 실행하면 다음과 같은 에러 보고를 받게 된다.

```
*****
File "half.py", line 4, in __main__.half
Failed example:
    half(6.8)
Expected:
    3.4
Got:
    3.399999999999999
*****
```

이 문제를 해결하려면 문서화 문자열을 출력 결과와 정확히 일치하도록 작성하거나 좀 더 나은 예를 선택할 필요가 있다.

doctest를 사용하는 일은 매우 간단하기 때문에 여러분의 프로그램에서 사용하지 않을 이유가 없다. 하지만 doctest는 보통 프로그램을 철저하게 테스트하는 데 사용되지는 않는다는 것을 명심하라. 그렇게 할 경우 과도하게 길고 복잡한 문서화 문자열을 작성하게 되고 이는 문서화의 의도 자체를 해칠 수 있다(예를 들어, 도움말을 불렀는데 온갖 복잡하고 까다로운 특별한 경우를 다루는 50개의 예가 출

력된다면 짜증이 날 수밖에 없을 것이다). 철저한 테스트를 수행하기를 원한다면 unittest 모듈을 사용하도록 한다.

마지막으로, doctest 모듈은 테스트를 어떻게 수행하고 결과를 어떻게 보고할 것인지 등 다양한 부분을 조절할 수 있는 많은 수의 설정 옵션을 제공한다. 이러한 옵션들은 대부분의 경우 필요하지 않으므로 여기서 다루지 않는다. 자세한 정보는 <http://docs.python.org/library/doctest.html>를 참고하라.

단위 테스트와 unittest 모듈

좀 더 철저하게 프로그램을 테스트하려면 unittest 모듈을 사용해야 한다. 개발자는 단위 테스트를 통해 프로그램의 구성 요소(함수, 메서드, 클래스, 모듈 등)에 대한 격리된 테스트를 작성할 수 있다. 작성된 테스트들은 큰 프로그램을 구성하는 기본 빌딩 블록들이 올바르게 작동하는지를 검사하기 위해서 실행된다. 프로그램의 크기가 커짐에 따라 다양한 구성 요소에 대한 단위 테스트들이 합쳐져서 큰 규모의 테스트 프레임워크와 테스트 도구를 형성하게 된다. 이렇게 되면 프로그램이 올바르게 작동하는지를 검사하는 작업이 쉬워질 뿐만 아니라 문제가 발생했을 때 문제를 격리시키고 고치는 작업이 간단해진다. 앞 절에 나왔던 코드를 사용해서 unittest 모듈을 어떻게 사용하는지 알아보자.

```
# splitter.py
def split(line, types=None, delimiter=None):
    """ 텍스트 줄을 분할하고 추가적으로 타입 변환을 수행한다.
    ...
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty, val in zip(types, fields) ]
    return fields
```

split() 함수를 단위 테스트들을 작성해서 다양하게 테스트하기 위해 다음과 같이 별개의 모듈 testsplitter.py를 생성한다.

```
# testsplitter.py
import splitter
import unittest

# 단위 테스트
class TestSplitFunction(unittest.TestCase):
    def setUp(self):
        # 설정 작업을 수행한다.(존재하는 경우)
        pass
```

```

def tearDown(self):
    # 청소 작업을 수행한다.(존재하는 경우)
    pass
def testsimplestring(self):
    r = splitter.split('GOOG 100 490.50')
    self.assertEqual(r,['GOOG','100','490.50'])
def testtypeconvert(self):
    r = splitter.split('GOOG 100 490.50',[str, int, float])
    self.assertEqual(r,['GOOG', 100, 490.5])
def testdelimiter(self):
    r = splitter.split('GOOG,100,490.50',delimiter=',')
    self.assertEqual(r,['GOOG','100','490.50'])

# 단위 테스트를 실행한다.
if __name__ == '__main__':
    unittest.main()

```

테스트를 실행하려면 다음과 같이 파일으로 testsplitter.py 파일을 실행하면 된다.

```
% python testsplitter.py
...
-----
```

Ran 3 tests in 0.014s

OK

unittest를 사용할 때는 기본적으로 unittest.TestCase로부터 상속받은 클래스를 정의한다. 이 클래스 안에 각 테스트를 ‘test’ 이름으로 시작하는 메서드로 정의한다. 예를 들어, ‘testsimplestring’, ‘testtypeconvert’ 등으로 정의할 수 있다(이름이 ‘test’로 시작하기만 하면 되고 나머지 부분은 사용자가 원하는 대로 정할 수 있다). 각 테스트 안에서는 다양한 조건을 검사하기 위해서 단언문을 사용한다.

unittest.TestCase 인스턴스 t에는 테스트를 작성하고 테스트 과정을 제어하는 데 사용할 수 있는 다음 메서드들이 있다.

t.setUp()

테스트 메서드를 실행하기 전에 설정 단계를 수행하기 위해서 호출된다.

t.tearDown()

테스트들을 실행한 다음 청소 단계를 수행하기 위해서 호출된다.

t.assert_(expr [, msg])
t.failUnless(expr [, msg])

expr이 False로 평가되면 테스트 실패를 알린다. msg는 실패에 대한 설명을 담은 문자열이다(주어진 경우).

```
t.assertEqual(x, y [,msg])
t.failUnlessEqual(x, y [, msg])
```

x와 y가 서로 다르면 테스트 실패를 알린다. msg는 실패를 설명하는 메시지이다(주어진 경우).

```
t.assertNotEqual(x, y [, msg])
t.failIfEqual(x, y, [, msg])
```

x와 y가 서로 같다면 테스트 실패를 알린다. msg는 실패를 설명하는 메시지이다(주어진 경우).

```
t.assertAlmostEqual(x, y [, places [, msg]])
t.failUnlessAlmostEqual(x, y, [, places [, msg]])
```

x와 y가 서로 소수 자리 places 차이 안에 있지 않은 경우 테스트 실패를 알린다. x와 y의 차이를 계산한 다음 결과를 주어진 소수점 자리로 반올림함으로써 검사를 수행한다. 이 결과가 영이면 x와 y는 거의 동일하다. msg는 실패를 설명하는 메시지이다(주어진 경우).

```
t.assertNotAlmostEqual(x, y, [, places [, msg]])
t.failIfAlmostEqual(x, y [, places [, msg]])
```

x와 y가 적어도 소수 자리 places만큼 떨어져 있지 않으면 테스트 실패를 알린다. msg는 실패를 설명하는 메시지이다(주어진 경우).

```
t.assertRaises(exc, callable, ...)
t.failUnlessRaises(exc, callable, ...)
```

호출 가능한 객체 callable이 예외 exc를 발생시키지 않으면 테스트 실패를 알린다. 나머지 인수들은 callable에 인수로서 전달된다. 여러 예외를 검사하려면 예외들을 담은 튜플을 exc에 사용하면 된다.

```
t.failIf(expr [, msg])
```

expr가 True로 평가되면 테스트 실패를 알린다. msg는 실패를 설명하는 메시지이다(주어진 경우).

```
t.fail([msg])
```

테스트 실패를 알린다. msg는 실패를 설명하는 메시지이다(주어진 경우).

```
t.failureException
```

이 속성은 테스트 안에서 마지막으로 잡힌 예외 값으로 설정된다. 이 속성은 예외

가 발생하였다는 사실을 확인하는 것뿐만 아니라 적절한 값을 담은 예외가 발생하였는지를 확인하고자 할 때 유용하게 쓰인다. 예를 들어, 예외 발생으로 생성된 여러 메시지를 확인하고자 할 때 사용할 수 있다.

unittest 모듈은 테스트를 그룹 짓고 테스트 끝음을 생성하며 테스트가 실행될 환경을 조절하는 등 커스터마이즈를 위한 다수의 고급 옵션을 제공한다. 이러한 기능들은 코드에 대한 테스트를 작성하는 과정과 직접 관련은 없다(앞에서 본 것처럼 사용자는 보통 테스트가 어떻게 실행될지를 신경 쓰지 않고 테스트 클래스를 작성 한다). 큰 규모의 프로그램에 대한 테스트를 조직화하는 방법에 관해서 알고 싶다면 <http://docs.python.org/library/unittest.html>를 참고하도록 한다.

파이썬 디버거와 pdb 모듈

파이썬은 pdb 모듈을 통해 간단한 명령 기반의 디버거를 제공한다. pdb 모듈은 사후(post-mortem) 디버깅, 스택 프레임 검사, 중단점, 소스 줄 단위의 단일 스텝 진행과 코드 평가를 지원한다.

프로그램이나 대화식 파이썬 셀에서 디버거를 불러내기 위해서 사용할 수 있는 함수가 몇 가지 있다.

`run(statement [, globals [, locals]])`

디버거의 제어하에서 문자열 statement를 실행한다. 코드가 실행되기 전에 디버거 프롬프트가 바로 먼저 나타난다. ‘continue’를 입력하면 실행이 진행된다. globals과 locals는 각각 코드가 실행되는 전역 네임스페이스와 지역 네임스페이스를 담는다.

`runeval(expression [, globals [, locals]])`

디버거의 제어하에서 문자열 expression을 평가한다. 코드가 실행되기 전에 디버거 프롬프트가 바로 먼저 나타난다. run()에서처럼 ‘continue’를 입력하면 실행이 진행된다. 성공할 경우 표현식의 값이 반환된다.

`runcall(function [, argument, ...])`

디버거 안에서 함수를 호출한다. function은 호출 가능한 객체이다. 추가적인 인수는 함수의 인수로 전달된다. 코드가 실행되기 전에 디버거 프롬프트가 나타난다. 종료 시점에 함수의 반환 값이 반환된다.

set_trace()

이 함수가 호출된 곳에서 디버거를 시작한다. 코드의 특정 지점에 디버거 중단점을 직접 집어넣는 데 사용한다.

post_mortem(traceback)

역추적 객체의 사후 디버깅을 시작한다. traceback은 보통 sys.exc_info() 같은 함수를 사용하여 얻는다.

pm()

마지막 예외의 역추적 정보를 사용하여 죽은뒤 디버깅을 시작한다.

디버거를 구동하는 함수 중에서 set_trace() 함수가 일반적으로 실전에서 사용하기 가장 쉽다. 복잡한 응용 프로그램을 작성하는 중에 어느 부분에서 문제를 발견하면 코드에 set_trace() 호출을 삽입한 다음에 간단히 응용 프로그램을 실행하기만 하면 된다. set_trace()를 만나면 프로그램이 중단되고 실행 환경을 조사할 수 있는 디버거로 바로 넘어간다. 디버거에서 빠져나오면 실행이 다시 시작된다.

디버거 명령

디버거를 구동하면 다음과 같이 (Pdb) 프롬프트가 나타난다.

```
>>> import pdb
>>> import buggymodule
>>> pdb.run('buggymodule.start( )')
><string>(0)?()
(Pdb)
```

(Pdb)는 디버거 프롬프트로서 다음에 나오는 명령을 인식하게 된다. 몇몇 명령은 짧고 긴 두 가지 형식을 가진다. 여기서는 두 형태를 모두 표시하는 데 팔호를 사용할 것이다. 예를 들어, h(elp)는 h나 help 모두 쓰일 수 있다는 것을 의미한다.

[!]statement

현재 스택 프레임의 컨텍스트에서 (한 줄) statement를 실행한다. 느낌표는 생략할 수 있지만 문장의 첫 단어가 디버거 명령과 비슷한 경우에는 모호성을 없애기 위해서 사용해야 한다. 전역 변수를 생성하려면 같은 줄에서 대입 명령 앞에 “global” 명령을 사용하면 된다.

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

a(rgs)

현재 함수의 인수 목록을 출력한다.

alias [name [command]]

command를 실행하는 별명 name을 생성한다. 별명을 입력할 때 command 문자열 안에 있는 부분 문자열 '%1', '%2' 등은 매개변수로 대체된다. '%*'은 모든 매개변수를 의미한다. command가 주어지지 않으면 현재 별명 목록이 출력된다. 별명은 중첩될 수 있고 Pdb 프롬프트에서 입력 가능한 것이라면 어떤 것인지를 알 수 있다. 다음 예를 보자.

```
# 인스턴스 변수를 출력 (사용법 "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.-
dict_[
# self 안에 있는 인스턴스 변수를 출력
alias ps pi self
```

b(reak) [loc [, condition]]

loc 위치에 중단점을 설정한다. loc는 파일 이름과 줄 번호이거나 모듈 내부의 함수 이름일 수 있다. 다음 문법이 사용된다

설정	설명
n	현재 파일의 줄 번호
filename:n	filename의 줄 번호
function	현재 모듈 내부의 함수 이름
module.function	module 내부의 함수 이름

loc를 생략하면 모든 현재 중단점이 출력된다. condition은 중단점이 작동하기 위해서 참으로 평가되어야 하는 표현식이다. 모든 중단점은 이 명령이 종료될 때 출력되는 번호를 할당받는다. 이 번호는 다음에 나오는 다른 디버거 명령에서 사용된다.

c(lear) [bpnumber [bpnumber ...]]

중단점 번호들을 지운다. 아무 인수도 주어지지 않으면 모든 중단점이 지워진다.

commands [bpnumber]

중단점 bpnumber를 만났을 때 자동으로 실행할 디버거 명령들을 설정한다. 실행할 명령을 입력할 때는 간단히 여러 줄에 걸쳐서 입력한 다음에 끝을 표시하기 위해 end를 사용한다. continue 명령이 들어 있으면 프로그램이 중단점에 도달했을 때 자동으로 다시 시작된다. bpnumber를 생략하면 마지막으로 설정한 중단점이

사용된다.

c(ont|inue)

중단점에 조건 condition을 설정한다. condition은 중단점을 작동하기 위해서 참으로 평가되어야 하는 표현식이다. condition을 생략하면 이전에 설정한 조건이 제거된다.

c(ont|inue)

다음 중단점을 만나기 전까지 계속 실행한다.

disable [bpnumber [bpnumber ...]]

지정된 중단점을 비활성화한다. clear와 달리 중단점을 나중에 다시 활성화 할 수 있다.

d(own)

스택 추적 정보에서 현재 프레임을 한 수준 아래로 이동시킨다.

enable [bpnumber [bpnumber ...]]

지정된 중단점을 활성화한다.

h(elp) [command]

사용 가능한 명령 목록을 보여준다. command를 지정하면 해당 command에 대한 도움말을 보여준다.

ignore bpnumber [count]

count 번 실행될 때까지 중단점을 무시한다.

j(ump) lineno

실행할 다음 줄을 설정한다. 동일한 실행 프레임에 있는 문장들에 대해서만 사용할 수 있다. 루프 안 문장 같이 문장 속으로 이동할 수는 없다.

l(list) [first [, last]]

소스 코드를 출력한다. 인수가 주어지지 않으면 현재 줄 주위의 11줄을 출력한다 (현재 줄, 앞쪽 다섯 줄, 뒤쪽 다섯 줄). 인수가 하나 주어지면 해당 줄 주위의 11줄을 출력한다. 인수가 두 개 주어지면 주어진 범위 안의 줄들을 출력한다. last가 first 보다 작으면 last는 줄 개수로 해석된다.

n(ext)

현재 함수에서 다음 줄을 만날 때까지 실행한다. 함수 호출 부분은 건너뛴다.

p expression

현재 컨텍스트에서 표현식을 평가하고 그 값을 출력한다.

pp expression

p 명령과 동일하지만 보기 좋은 출력을 생성하는 모듈(pprint)을 사용해서 결과를 출력한다.

q(uit)

디버거를 종료한다.

return

현재 함수가 반환되기 전까지 실행한다.

run [args]

프로그램을 재시작하고 args에 있는 명령줄 인수를 사용해서 sys.argv를 새로 설정한다. 모든 중단점과 디버거 설정이 유지된다.

s(tep)

소스 한 줄을 실행하며 함수가 호출될 경우 그 안에서 멈춘다.

tbreak [loc [, condition]]

한번 도달하면 제거되는 임시 중단점을 설정한다.

u(p)

현재 프레임을 스택 추적 정보 안에서 한 수준 위로 이동시킨다.

unalias name

지정된 별명을 지운다.

until

제어가 현재 실행 프레임을 벗어나거나 현재 줄 번호보다 더 큰 줄 번호에 도달 할 때까지 실행을 재개한다. 예를 들어, 디버거가 루프 몸체에서 마지막 줄에서 멈추었다면 until을 입력할 경우 루프가 종료될 때까지 루프 안에 있는 모든 문장을 실행하게 된다.

w(here)

스택 추적 정보를 출력한다.

명령줄에서 디버깅

명령줄에서 디버거를 실행할 수도 있다. 다음 예를 보자.

```
% python -m pdb someprogram.py
```

이렇게 하면 디버거가 프로그램이 시작할 때 자동으로 실행되고 중단점 설정이 라든지 기타 설정을 자유롭게 변경할 수 있게 된다. 프로그램을 실행하려면 간단히 continue 명령을 입력하면 된다. 예를 들어, 프로그램에서 split() 함수를 사용하고 있고 이 함수를 디버깅하고 싶다면 다음과 같이 하면 된다.

```
% python -m pdb someprogram.py
> /Users/beazley/Code/someprogram.py(1)<module>()
-> import splitter
(Pdb) b splitter.split
Breakpoint 1 at /Users/beazley/Code/splitter.py:1
(Pdb) c
> /Users/beazley/Code/splitter.py(18)split()
-> fields = line.split(delimiter)
(Pdb)
```

디버거 설정

사용자 홈 디렉터리나 현재 디렉터리에 .pdbrc 파일이 있으면 그 내용이 디버거 프롬프트에서 실제로 입력한 것처럼 실행된다. 디버거를 구동할 때마다 매번 실행하고 싶은 디버깅 명령들을 이곳에 두면 된다(매번 직접 명령을 입력하는 것이 아니라).

프로그램 프로파일링

profile과 cProfile 모듈은 프로파일 정보를 수집하는 데 사용된다. 두 모듈이 동일한 방식으로 작동하기는 하지만 cProfile이 C의 확장 기능으로 구현되어 있기 때문에 상당히 빠르고 좀 더 최신 모듈이다. 각 모듈은 도달 범위(coverage) 분석(즉, 어떤 함수가 실행되었는지)뿐만 아니라 성능 통계 정보 수집에도 사용된다. 가장 손쉽게 프로그램을 분석하고자 한다면 다음과 같이 명령줄에서 cProfile을 실행하면 된다.

```
% python -m cProfile someprogram.py
```

profile 모듈에 있는 다음 함수를 사용할 수도 있다.

```
run(command [, filename])
```

이 함수는 프로파일러 속에서 exec문으로 command의 내용을 실행한다. filename은 미가공 프로파일링 데이터가 저장될 파일의 이름을 나타낸다. 생략할 경우 표준 출력으로 결과가 출력된다.

프로파일러를 실행하면 다음과 같은 보고를 받게 된다.

```
126 function calls (6 primitive calls) in 5.130 CPU seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall   filename:lineno(function)
      1    0.030    0.030    5.070    5.070   <string>:1(?)
 121/1    5.020    0.041    5.020    5.020   book.py:11(process)
      1    0.020    0.020    5.040    5.040   book.py:5(?)
      2    0.000    0.000    0.000    0.000   exceptions.py:101(__init__)
      1    0.060    0.060    5.130    5.130   profile:0(execfile('book.py'))
      0    0.000        0.000        0.000        0.000   profile:0(profiler)
```

run()에 의해 생성되는 보고서의 각 부분은 다음과 같은 의미를 갖는다.

구역	설명
primitive calls	비재귀 함수 호출 횟수
ncalls	총 호출 횟수(자신에 대한 호출도 포함)
tottime	이 함수에서 머무른 시간(하위 함수는 고려하지 않음)
percall	tottime/ncalls
cumtime	이 함수에서 머무른 총 시간
percall	cumtime/(primitive calls)
filename:lineno(function)	각 함수의 위치와 이름

첫 번째 열에 숫자가 두 개 나타나면(예를 들어 “121/1”) 후자는 primitive call의 수를, 전자는 실제 호출 수를 나타낸다.

대부분의 응용 프로그램에 대해서 생성된 프로파일러의 보고서를 살펴보는 것 만으로 충분하다(예를 들어, 간단히 프로그램이 어떻게 시간을 쓰고 있는지 알고 싶을 때). 데이터를 저장하고 분석을 더 하고 싶다면 pstats 모듈을 사용할 수 있다. 이와 관련한 자세한 정보는 <http://docs.python.org/library/profile.html>에서 찾아볼 수 있다.

튜닝과 최적화

이 절에서는 파이썬 프로그램을 좀 더 빠르게 실행되게 하고 메모리를 보다 적게 사용하게 만드는 데 사용할 수 있는 몇 가지 경험적인 법칙을 다룬다. 모든 기법을 설명하지는 않고 있지만 그래도 코드를 들여다볼 때 써먹을 수 있는 몇 가지 아이디어는 얻을 수 있을 것이다.

시간 측정

오래 실행되는 파이썬 프로그램의 시간을 간단히 측정하고 싶다면 UNIX의 time과 같은 명령을 사용해 프로그램을 실행하는 것이 가장 쉽다. 또는 시간을 측정하려는 오래 실행되는 문장 블록이 있는 경우 time.clock() 호출을 추가해서 현재 CPU 경과 시간 값을 얻어오거나 time.time()을 호출해서 실제 실행 시간(wall-clock time)을 읽어올 수도 있다. 다음 예를 보자.

```
start_cpu = time.clock( )
start_real= time.time( )
문장들
문장들
end_cpu = time.clock( )
end_real = time.time( )
print("%f Real Seconds" % (end_real - start_real))
print("%f CPU seconds" % (end_cpu - start_cpu))
```

이 기법은 측정하고자 하는 코드가 어느 적정 시간 이상 동안 실행될 때만 사용이 가능하다. 더 짧게 실행되는 문장에 대해서 벤치마크를 수행하려면 다음 예와 같이 timeit 모듈의 timeit(code[, setup]) 함수를 사용하면 된다.

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2.0)', 'import math')
0.20388007164001465
>>> timeit('sqrt(2.0)', 'from math import sqrt')
0.14494490623474121
```

이 예에서 timeit()의 첫 번째 인수는 벤치마크하고자 하는 코드이다. 두 번째 인수는 실행 환경을 설정하기 위해 한 번만 실행될 문장을 나타낸다. timeit() 함수는 제공된 문장을 100만 번 실행하고 경과 시간을 보고한다. 반복 실행 횟수는 timeit()에 number=count 키워드 인수를 제공해서 변경할 수 있다.

timeit 모듈에는 시간을 측정하는 데 사용할 수 있는 repeat()이라는 함수도 있다. 이 함수는 시간을 세 번 측정하고 그 결과를 리스트로 반환하는 것 말고는

timeit()과 동일하게 작동한다. 다음 예를 보자.

```
>>> from timeit import repeat
>>> repeat('math.sqrt(2.0)', 'import math')
[0.20306601524353027, 0.19715800285339355, 0.20907392501831055]
>>>
```

성능을 측정할 때는 보통 속도 향상(speedup)에 관해서 이야기하는 경우가 많다. 속도 향상 값은 일반적으로 원래 실행 시간을 새로운 실행 시간으로 나누는 식으로 구한다. 예를 들어, 앞에서 시간 측정을 했을 때 math.sqrt(2.0) 대신 sqrt(2.0)을 사용하는 것이 0.20388/0.14494, 즉 약 1.41배 빨라진 것을 확인할 수 있었다. 이것을 퍼센트 단위를 사용해서 약 41퍼센트의 속도 향상이 있었다고 말하기도 한다.

메모리 측정

sys 모듈에는 개별 파이썬 객체의 메모리 사용량(바이트 단위)을 살펴보는 데 사용할 수 있는 getsizeof() 함수가 있다. 다음 예를 살펴보자.

```
>>> import sys
>>> sys.getsizeof(1)
14
>>> sys.getsizeof("Hello World")
52
>>> sys.getsizeof([1,2,3,4])
52
>>> sum(sys.getsizeof(x) for x in [1,2,3,4])
56
```

리스트, 튜플, 사전 같은 컨테이너에 대해서 보고되는 크기는 단지 컨테이너 자체 그 자체의 크기이지 컨테이너 안에 있는 모든 객체의 크기의 합을 의미하지 않는다. 예를 들어, 앞선 예에서 list[1, 2, 3, 4]의 사이즈는 실제로 네 개의 정수를 위해 필요한 공간보다 더 작다(각각 14 바이트). 그 이유는 리스트에 담긴 내용의 크기가 고려되지 않기 때문이다. 리스트에 담긴 내용의 총 크기를 계산하려면 앞에 나온 것처럼 sum()을 사용해야 한다.

getsizeof() 함수는 다양한 객체에 대해서 사용되는 전체적인 메모리량에 대한 대략적인 정보만을 제공한다. 내부적으로 인터프리터는 참조 횟수 세기를 통해 객체들을 적극적으로 공유하기 때문에 어떤 객체에 대해서 소모되는 실제 메모리가 여러분이 생각한 것보다 훨씬 작을 수 있다. 또한 파이썬의 C 확장 기능은 인터프리터 외부의 메모리를 사용할 수 있기 때문에 전체 메모리 사용량을 정확히 측정하기는 어렵다. 실제 메모리 사용량을 측정하는 이차적인 방법으로 운영 체제의 프로세서

뷰어나 작업 관리자를 통해 실행 중인 프로그램을 살펴보는 방법도 있다.

사실 메모리 사용량을 제대로 파악하려면 자리에 앉아서 제대로 분석해보는 것 이 더 나을 수도 있다. 프로그램에서 어떤 종류의 데이터 구조를 사용하고 각 데이 터 구조에 어떤 종류의 데이터(정수, 실수, 문자열 등)를 담을 것인지를 알고 있다면 `getsizeof()` 함수를 사용해서 메모리 사용량의 상한 값을 계산하기 위한 수치를 얻을 수 있다. 아니면 적어도 봉투 뒷면에 간단한 계산을 해보는 데 필요한 충분한 정 보를 얻을 수 있다.

분해

`dis` 모듈은 함수, 메서드, 클래스를 저수준 인터프리터 명령어로 분해하는 데 사용 된다. 이 모듈에는 `dis()` 함수가 있고 다음과 같이 사용한다.

```
>>> from dis import dis
>>> dis(split)
 2      0 LOAD_FAST          0 (line)
 3      3 LOAD_ATTR           0 (split)
 6      6 LOAD_FAST          1 (delimiter)
 9     9 CALL_FUNCTION       1
12    12 STORE_FAST          2 (fields)
15    15 LOAD_GLOBAL          1 (types)
18    18 JUMP_IF_FALSE      58 (to 79)
21    21 POP_TOP
4      22 BUILD_LIST          0
25    25 DUP_TOP
26    26 STORE_FAST          3 (_[1])
29    29 LOAD_GLOBAL          2 (zip)
32    32 LOAD_GLOBAL          1 (types)
35    35 LOAD_FAST           2 (fields)
38    38 CALL_FUNCTION        2
41    41 GET_ITER
>> 42 FOR_ITER            25 (to 70)
45    45 UNPACK_SEQUENCE     2
48    48 STORE_FAST          4 (ty)
51    51 STORE_FAST          5 (val)
54    54 LOAD_FAST           3 (_[1])
57    57 LOAD_FAST           4 (ty)
60    60 LOAD_FAST           5 (val)
63    63 CALL_FUNCTION        1
66    66 LIST_APPEND
67    67 JUMP_ABSOLUTE       42
>> 70 DELETE_FAST          3 (_[1])
73    73 STORE_FAST          2 (fields)
76    76 JUMP_FORWARD         1 (to 80)
>> 79 POP_TOP
5    >> 80 LOAD_FAST           2 (fields)
83    83 RETURN_VALUE
```

>>>

전문 프로그래머는 이 정보를 두 가지 용도로 사용한다. 첫째, 분해를 수행하면 함수를 실행할 때 어떠한 연산이 관련되어 있는지를 정확히 알 수 있다. 세심하게 분석하면 속도를 향상시킬 수 있는 기회를 발견할 수도 있다. 둘째, 스레드 프로그래밍을 할 때 분해 결과로 출력되는 각 줄은 단일 인터프리터 연산(각각은 원자적 atomic으로 실행된다)을 나타낸다. 까다로운 경쟁 조건(race condition)을 추적하고자 할 때 이 정보가 유용할 수 있다.

튜닝 전략

여기서 필자가 생각하기에 파이썬에서 유용하게 쓸 수 있다고 생각하는 몇 가지 최적화 전략에 대해서 간략하게 설명한다.

프로그램을 이해하라

무언가를 최적화하기 전에, 프로그램 최적화로 얻을 수 있는 속도 향상은 해당 부분이 전체 실행 시간에 기여하는 정도와 밀접한 관련이 있다는 것을 알아야 한다. 예를 들어, 어떤 함수를 10배 빠르게 만들었다고 하더라도 해당 함수가 전체 실행 시간에 10퍼센트밖에 기여하지 못한다고 하면 전체적인 속도 향상은 9~10퍼센트밖에 되지 않는다. 최적화에 들어가는 노력 정도에 따라 최적화를 수행하는 일이 가치 있는 일이 될 수도 있고 아닐 수도 있다.

최적화를 하려는 코드를 항상 먼저 프로파일링해보는 것이 좋다. 여러분의 프로그램에서 가장 시간을 많이 잡아먹는 함수나 메서드에 초점을 맞추는 것이 좋으며, 드물게 호출되는 확실하지 않은 연산을 최적화하는 것은 좋지 않다.

알고리즘을 이해하라

형편없이 구현된 $O(n \log n)$ 알고리즘이라고 할지라고 잘 구현된 $O(n^3)$ 보다는 월등히 빠르다. 비효율적인 알고리즘을 최적화하려고 시도하지 말고 먼저 더 나은 알고리즘을 찾도록 한다.

내장 타입을 사용하라

파이썬의 내장 투플, 리스트, 집합, 사전 타입은 모두 C로 구현되어 있고 인터프리터에 가장 잘 튜닝되어 있는 자료 구조이기도 하다. 프로그램에서 데이터를 저장하

거나 조작할 때 이러한 내장 타입을 활발하게 사용하고 자신만의 데이터 구조를 만들어서 내장 데이터 타입을 흉내 내려고 하지 않는 것이 좋다(이진 검색 트리나 연결 리스트 등을 구현하는 식으로).

내장 타입 말고도 표준 라이브러리에 있는 타입들도 잘 살펴볼 필요가 있다. 몇몇 라이브러리 모듈에서는 특정 작업을 수행할 때 내장 타입보다 성능이 더 좋은 새로운 타입을 제공하기도 한다. 예를 들어, `collection.deque` 타입은 리스트와 유사한 기능을 제공하지만 앞뒤로 새로운 항목을 추가하는 작업에 최적화되어 있다. 반면에 리스트는 항목을 뒤에 추가할 때만 효율적으로 작동한다. 리스트 앞에 항목을 추가하면 새로운 공간을 만들기 위해서 리스트에 존재하는 모든 항목을 옆으로 옮겨야 한다. 여기에 드는 시간은 리스트가 커질수록 더 많다. 다음은 리스트와 `dequeue`의 앞쪽에 100만 개 항목을 추가하는 데 걸리는 시간을 보여준다.

```
>>> from timeit import timeit
>>> timeit('s.appendleft(37)',
...           'import collections; s = collections.deque( )',
...           number=1000000)
0.24434304237365723
>>> timeit('a.insert(0,37)', 's=[]', number=1000000)
612.95199513435364
```

계층을 추가하지 않는다

추상화를 위해서나 객체나 함수에 대해서 편의 기능을 제공하려고 계층을 하나 추가할 때마다 프로그램은 더 느려진다. 사용 편의성과 성능은 서로 상충 관계에 있다. 예를 들어, 계층을 하나 추가하는 것은 속도를 느리게 하지만 코딩이 더 간단해지는 이점이 있다.

이 점을 간단한 예를 통해서 알아보자. 다음 코드는 `dict()` 함수를 사용해서 문자열 키를 갖는 사전을 생성한다.

```
s = dict(name='GOOG', shares=100, price=490.10)
# s = {'name': 'GOOG', 'shares': 100, 'price': 490.10 }
```

이렇게 하면 사전을 생성할 때 타이핑 시간을 줄일 수 있다(키 이름에 따옴표를 두르지 않아도 된다). 하지만 추가적인 함수 호출이 있기 때문에 실행 속도가 훨씬 느린다.

```
>>> timeit("s = {'name': 'GOOG', 'shares':100,'price':490.10}")
0.38917303085327148
>>> timeit("s = dict(name='GOOG',shares=100,price=490.10)")
0.94420003890991211
```

프로그램에서 실행 도중에 수백만 개의 사전을 생성하는 경우라면 첫 번째 방식이 빠르다. 몇몇 예외가 있기는 하지만 대체로 기존 파이썬 객체의 작동 방식을 개선하거나 변경한 기능을 사용하면 프로그램은 더 느려진다.

클래스와 인스턴스가 사전을 토대로 만들어진 것을 안다

사용자 정의 클래스와 인스턴스는 사전을 사용하여 만들어진다. 따라서 인스턴스 데이터를 검색하고 설정하며 지우는 연산은 직접 사전에 대고 이 작업을 수행하는 것보다 대부분의 경우 느리다. 단지 데이터를 저장하기 위한 간단한 자료 구조를 원하는 것뿐이라면 클래스를 정의하는 것보다 사전을 사용하는 것이 더 효율적일 수 있다.

두 방식이 얼마나 차이 나는지 알아보기 위해서 다음과 같이 주식을 나타내는 간단한 클래스를 살펴보자.

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

이 클래스를 사용하는 경우와 사전을 사용하는 경우의 성능을 비교해보면 재미 있는 결과를 얻을 수 있다. 먼저 단순히 인스턴스를 생성할 때의 성능을 비교해보자.

```
>>> from timeit import timeit
>>> timeit("s = Stock('GOOG',100,490.10)","from stock import Stock")
1.3166780471801758
>>> timeit("s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")
0.37812089920043945
>>>
```

새로운 객체를 생성할 때 약 3.5배 속도 향상이 있었다. 다음으로 간단한 계산을 수행해보자.

```
>>> timeit("s.shares*s.price",
...         "from stock import Stock; s = Stock('GOOG',100,490.10)")
0.29100513458251953
>>> timeit("s['shares']*s['price']",
...         "s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")
0.23622798919677734
>>>
```

약 1.2배의 속도 향상이 있었다. 여기서 얻을 수 있는 교훈은 클래스만이 데이터

를 저장하는 유일한 방법이 아니라는 점이다. 튜플과 사전만으로 충분한 경우가 많다. 이 자료 구조들을 사용하면 프로그램이 더 빨리 실행되고 메모리도 덜 사용하게 된다.

`__slots__`을 사용하라

프로그램에서 사용자 정의 클래스의 인스턴스를 다수 생성한다면 클래스 정의에 `__slots__` 속성을 사용하는 것을 고려해 보도록 하라. 다음 예를 보자.

```
class Stock(object):
    __slots__ = ['name','shares','price']
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

`__slots__` 은 속성 이름에 제약을 가하기 때문에 보통 보안을 위한 기능으로서 취급된다. 하지만, 오히려 성능 최적화를 위한 기능에 가깝다. `__slots__`을 사용한 클래스에서는 인스턴스 데이터를 저장하는 데 사전을 쓰지 않는다(대신에 효율적인 내부 자료 구조를 사용한다). 따라서 인스턴스가 훨씬 적은 메모리를 사용할 뿐만 아니라 인스턴스 데이터에 접근하는 일도 효율적으로 이루어진다. 어떤 경우에는 단순히 `__slots__`을 추가하는 것만으로도 프로그램이 눈에 띄게 빠르게 수행되기도 한다.

`__slots__`을 사용할 때 한 가지 주의할 점이 있다. 이 기능을 클래스에 추가하면 다른 코드가 알 수 없는 이유로 제대로 작동하지 않을 수 있다. 예를 들어, 인스턴스 데이터는 보통 `__dict__` 속성으로 접근할 수 있는 사전에 저장된다고 알고 있다. 하지만 `slots`을 정의하면 `__dict__` 속성이 존재하지 않게 되고 `__dict__`에 의존하는 코드는 작동하지 않게 된다.

점(.) 연산자의 사용을 피하라

객체의 속성을 찾는 데 점(.) 연산자를 사용할 때마다 항상 이름 검색이 수행된다. 예를 들어, `x.name`을 입력하면 먼저 변수 이름 “`x`”를 찾고 그리고 나서 `x`에 “`name`” 속성이 있는지를 찾는다. 사용자 정의 객체의 경우 속성 검색은 인스턴스 사전, 클래스 사전, 기반 클래스들의 사전에 대한 검색으로 이어질 수 있다.

메서드 검색이나 모듈 검색이 찾은 경우에는 대부분의 경우 지역 변수를 사용해

서 속성 검색 연산을 먼저 제거하는 것이 좋다. 제곱근을 구하는 연산을 여러 번 수행한다고 할 때 ‘from math import sqrt’과 ‘sqrt(x)’를 사용하는 것이 ‘math.sqrt(x)’를 사용하는 것보다 더 빠르다. 이 절의 앞 부분에서 약 1.4배의 속도 향상이 있다 는 것을 확인할 수 있었다.

당연히 프로그램의 모든 곳에서 속성 검색을 제거하려고 해서는 안 된다. 그렇게 하면 코드를 읽기가 어렵다. 하지만 성능이 중요한 곳에서는 유용하게 사용할 수 있다.

흔하지 않은 경우를 처리하기 위해서 예외를 사용하라
에러를 피하려고 다음과 같이 프로그램에서 추가 검사를 수행하는 경우가 종종 있다.

```
def parse_header(line):
    fields = line.split(":")
    if len(fields) != 2:
        raise RuntimeError("Malformed header")
    header, value = fields
    return header.lower( ), value.strip( )
```

이렇게 하지 않고 다음과 같이 간단히 프로그램에서 예외를 발생시키게 한 다음 예외를 잡는 방식으로 에러를 처리할 수도 있다.

```
def parse_header(line):
    fields = line.split(":")
    try:
        header, value = fields
        return header.lower( ), value.strip( )
    except ValueError:
        raise RuntimeError("Malformed header")
```

두 코드의 성능을 비교하면 두 번째 코드가 약 10퍼센트 빠르다. 예외를 잘 발생시키지 않는 코드를 try 블록으로 둘러싸는 것이 if문을 실행하는 것보다 더 빠르다.

흔한 경우에 대한 예외 처리를 피하라

흔하게 발생하는 경우에 대해서는 예외 처리 코드를 작성하지 않도록 한다. 예를 들어, 사전을 많이 검색하는 프로그램이 있을 때 대부분의 경우 키가 존재하지 않는다고 하자. 다음과 같은 두 가지 접근법이 있을 수 있다.

```
# 첫 번째 방법: 검색을 수행하고 예외를 잡는다.
try:
    value = items[key]
except KeyError:
    value = None

# 두 번째 방법: 키 값이 존재하는지 확인하고 검색을 수행한다.
if key in items:
    value = items[key]
else:
    value = None
```

키를 찾을 수 없는 경우 간단한 성능 비교를 하면 두 번째 방법이 17배 빠르다!
궁금한 사람들을 위해 말하자면 후자의 방법이 `items.get(key)`를 사용하는 것보다
도 약 2배 빠르다. 그 이유는 `in` 연산이 메서드 호출보다 더 빠르기 때문이다.

함수형 프로그래밍과 반복을 채택하라

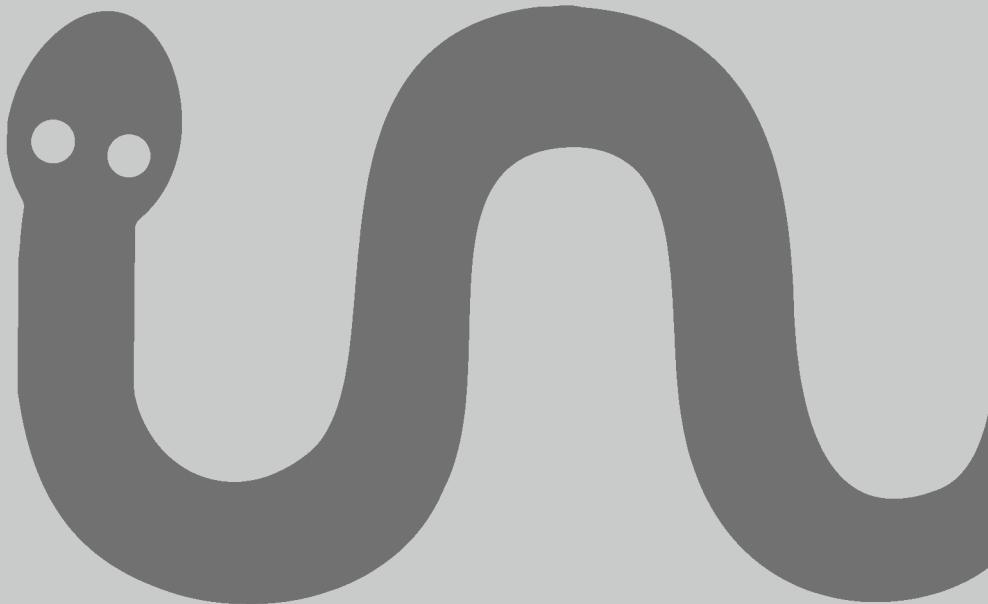
리스트 내포, 생성기 표현식, 생성기, 코루틴, 클로저는 대부분의 파이썬 프로그래머
가 인식하고 있는 것보다 더 효율적으로 작동한다. 특히 데이터 처리를 수행할 때
리스트 내포와 생성기 표현식은 직접 데이터에 대해 반복을 수행하면서 유사한 연
산을 수행하는 것보다 훨씬 빠르다. 이들은 `map()`과 `filter()` 같은 함수를 사용하
는 데거시 파이썬 코드보다도 훨씬 빠르다. 생성기를 사용하면 더 빠를 뿐만 아니
라 메모리도 더 효율적으로 사용하는 코드를 작성할 수 있다.

장식자와 메타클래스를 사용하라

장식자와 메타클래스는 함수와 클래스를 수정하는 데 사용된다. 이들은 함수 정의
나 클래스 정의 시점에 작동하기 때문에 성능을 향상시키는 데도 사용할 수 있다.
특히 프로그램에서 켜고 끌 수 있는 옵션인 기능을 많이 제공하는 경우 유용하게
쓰인다. 6장에서 로깅을 수행하지 않을 때는 성능에 영향을 주지 않으면서 함수 로
깅을 활성화하는 데 장식자를 사용하는 예를 살펴보았다.

2부

파이썬 라이브러리



P y t h o n E s s e n t i a l R e f e r e n c e

12장

Python Essential Reference

내장 함수와 예외

이 장에서는 파이썬의 내장 함수와 예외를 설명한다. 여기 나와 있는 내용은 이 책의 앞 부분에서 공식적이지 않은 형태로 다루었다. 이 장에서는 모든 정보를 한 절에 모으고 몇몇 이해하기 어려운 함수는 더 자세히 설명한다. 파이썬 2에는 이제 쓸모 없는 것으로 생각되어 파이썬 3에서 제거된 몇몇 내장 함수가 있다. 이러한 함수는 이 장에서 다루지 않는다. 이 장에서는 파이썬의 최신 기능에만 초점을 맞춘다.

내장 함수와 타입

파이썬에는 인터프리터에서 항상 접근할 수 있고 어느 모듈에서나 사용할 수 있는 타입, 함수, 변수들이 있다. 다음에 나오는 함수에 접근할 때는 임포트가 필요 없으며, 이들은 파이썬 2에는 `__builtin__` 모듈에, 파이썬 3에서는 `builtins` 모듈에 들어 있다. 여러분이 임포트할 모듈 안에서도 `__builtins__`로 이 모듈에 접근할 수 있다.

abs(x)

`x`의 절대값을 반환한다.

all(s)

반복 가능한 `s`에 들어 있는 모든 값이 `True`로 평가되면 `True`를 반환한다.

any(s)

반복 가능한 `s`에 들어 있는 아무 값이나 `True`로 평가되면 `True`를 반환한다.

ascii(x)

`repr()`처럼 객체 x의 출력 가능한 표현을 생성하지만 결과를 생성하는 데 ASCII 문자만 사용한다. ASCII가 아닌 문자는 적절한 탈출 순서열로 변환된다. 유니코드를 지원하지 않는 터미널이나 셸에서 유니코드 문자열을 출력하는 데 사용할 수 있다. 파이썬 3에만 있다.

basestring

파이썬 2에서 모든 문자열(str과 unicode)의 상위 클래스인 추상 데이터 타입이다. 문자열 검사를 할 때만 쓰인다. 예를 들어서, `isinstance(s, basestring)`은 s가 문자열의 한 종류일 때 `True`를 반환한다. 파이썬 2에만 있다.

bin(x)

정수 x의 이진 표현을 담은 문자열을 반환한다.

bool([x])

불리언 값 `True`와 `False`를 표현하는 타입. x를 변환하기 위해 사용되면 x가 보통의 진리값 검사 작동 방식(영어 아닌 숫자, 비어 있지 않은 리스트 등)에 따라 참으로 평가될 경우 `True`를 반환한다. 아니면 `False`를 반환한다. `bool()`을 인수 없이 호출하면 기본으로 `False`가 반환된다. `bool` 클래스는 `int`에서 상속받았기 때문에 불리언 값 `True`와 `False`는 수리 계산을 할 때 값 1과 0을 갖는 정수로서 사용될 수 있다.

bytearray([x])

변경 가능한 바이트 배열을 표현하는 타입. 인스턴스를 생성하는 데 사용할 때 x는 0에서 255까지의 범위를 가지는 숫자들을 담은 반복 가능한 순서열이거나, 8비트 문자열 또는 바이트 상수이거나, 바이트 배열의 크기를 지정하는 정수(이 경우 모든 항목은 0으로 초기화된다)일 수 있다. `bytearray` 객체 a는 정수 배열과 비슷하다. `a[i]`로 검색을 수행하면 색인 i에 있는 바이트 값을 표현하는 정수를 얻게 된다. `a[i] = v`로 대입을 수행할 때 v는 정수 바이트 값이어야 한다. `bytearray`는 보통 문자열과 관련 있는 연산도 모두 지원한다(`분할`, `find()`, `split()`, `replace()` 등). 이러한 문자열 연산을 사용할 때는 문자열 상수 앞에 b를 붙여주어서 바이트를 다루고 있다는 것을 알려야 한다. 예를 들어서, 콤마 문자를 분리자로 하여 바이트 배열을 여러 필드로 나누고 싶은 경우라면 `a.split(' , ')`이 아니라 `a.split(b ' , ')`라고 써야 한다. 이러한 연산의 결과는 문자열이 아니라 항상 새로운 `bytearray` 객체이다. `bytearray` 객체 a를 문자열로 바꾸려면 `a.decode(encoding)` 메서드를 사용하면 된

다. encoding 값으로 ‘latin-1’을 사용하면 내부 문자 값을 수정하지 않고 8비트 문자로 구성된 bytearray를 그대로 문자열로 변환하게 된다.

bytearray(s,encoding)

encoding을 변환에 사용할 문자 인코딩으로 하여 문자열 s에 들어 있는 문자들로 bytearray 인스턴스를 생성하는 방법.

bytes([x])

변경 불가능한 바이트 배열을 표현하는 타입. 파이썬 2에서 이 타입은 8비트 문자들로 구성되는 표준 문자열을 생성하는 str()의 별명이다. 파이썬 3에서 bytes는 앞에서 설명한 bytearray 타입의 변경 불가능한 타입으로서 완전히 별개의 타입이다. 인수 x는 bytearray의 경우와 동일하게 해석된다. 파이썬 2에서 bytes가 정의되어 있기는 하지만 결과 객체가 파이썬 3의 것과 작동하는 방식이 동일하지 않은 호환성 문제가 있다. 예를 들어, bytes()로 생성된 인스턴스 a가 있을 때 파이썬 2에서는 a[i]가 문자를 담은 문자열을 반환하지만, 파이썬 3에서는 정수를 반환한다.

bytes(s,encoding)

encoding을 변환에 사용할 문자 인코딩으로 하여 문자열 s에 들어 있는 문자들로부터 bytes 인스턴스를 생성하는 방법. 파이썬 3에만 있다.

chr(x)

정수값 x를 문자 하나짜리 문자열로 변환한다. 파이썬 2에서는 x의 범위가 $0 \leq x \leq 255$ 이어야 하고 파이썬 3에서는 x가 유효한 유니코드 코드 포인트를 나타내야 한다. x가 범위를 벗어나면 ValueError 예외가 발생한다.

classmethod(func)

함수 func에 대한 클래스 메서드를 생성한다. 보통 @classmethod 장식자에 의해 서 간접적으로 호출되며 클래스 정의 안에서만 사용된다. 보통의 메서드와 달리, 클래스 메서드는 첫 번째 인수로 인스턴스가 아니라 클래스를 받는다. 예를 들어, 클래스 Foo의 인스턴스인 객체 f가 있을 때, f에 클래스 메서드를 호출하면 인스턴스 f가 아니라 클래스 Foo가 첫 번째 인수로 전달된다.

cmp(x, y)

x와 y를 비교하고 $x < y$ 이면 음수를, $x > y$ 이면 양수를, $x == y$ 이면 0을 반환한다. 어떠한 두 객체도 비교할 수 있다. 두 객체를 의미 있게 비교할 수 없는 경우 결과는

아무런 의미가 없다(예를 들어, 숫자를 파일 객체와 비교한다든지). 어떤 경우에는 의미 없는 비교를 수행하면 예외가 발생할 수도 있다. (파이썬 3에서는 제거됨)

compile(string, filename, kind [, flags [, dont_inherit]])

exec()나 eval()에서 사용할 수 있도록 string을 코드 객체로 변환한다. string은 유효한 파이썬 코드를 담은 문자열이다. 이 코드가 여러 줄로 구성되면 각 줄은 플랫폼에 종속적인 문자가 아닌(예를 들어, 윈도에서 '\r\n') 단일 줄바꿈 문자('\n')로 종료되어야 한다. filename은 string을 읽어올 파일 이름이다. kind의 값으로는 일련의 문장들에 대해서는 'exec', 단일 표현식에 대해서는 'eval', 단일 실행 가능한 문장에 대해서는 'single'를 사용해야 한다. flags는 어떤 추가 기능을 활성화할 것인지를 지정하며(__future__ 모듈과 관련해서) __future__ 모듈에 정의된 플래그들을 비트 OR하여 기술한다. 예를 들어, 새로운 나누기 작동 방식을 활성화하려면 flags를 __future__.division.compiler_flag로 설정하면 된다. flags를 생략하거나 0으로 설정하면 현재 활성화되어 있는 기능을 사용하여 코드를 컴파일한다. flags를 제공하면 지정된 기능이 현재 활성화되어 있는 기능에 추가된다. dont_inherit이 설정되면 flags으로 지정한 기능만 활성화된다. 이때 현재 활성화되어 있는 기능은 무시된다.

complex([real [, imag]])

아무 숫자 타입이 올 수 있는 실수부 real과 허수부 imag로 구성되는 복소수를 나타내는 타입. imag가 생략되면 허수부가 영으로 설정된다. real이 문자열로 전달되면 문자열이 파싱되어 복소수로 변환된다. 이 경우 imag는 지정하면 안 된다. 아무 인수도 주어지지 않으면 0j가 반환된다.

delattr(object, attr)

객체의 속성을 삭제한다. attr는 문자열이다. del object.attr와 동일하다.

dict([m]) or dict(key1 = value1, key2 = value2, ...)

사전을 나타내는 타입. 전달 인수가 없으면 빈 사전을 반환한다. m이 매핑 객체이면(사전 같은) m과 동일한 키와 값을 갖는 새로운 사전을 반환한다. 예를 들어, m이 사전이면 dict(m)은 간단히 얇은 복사본을 생성한다. m이 매핑 객체가 아니면 m은 일련의 (key,value) 쌍을 생성하는 반복을 지원하는 객체이어야 한다. 이 쌍들이 사전의 내용을 채우는 데 쓰인다. dict()를 키워드 인수로 호출할 수도 있다. 예

를 들어, `dict(foo=3,bar=7)`은 사전 `{ 'foo' : 3, 'bar' : 7 }`를 생성한다.

dir([object])

속성 이름 목록을 반환한다. object가 모듈이면 모듈 안에서 정의된 기호 목록을 반환한다. object가 타입이나 클래스 객체이면 속성 이름 목록을 반환한다. 보통 객체에 `__dict__` 속성이 정의되어 있으면 이 속성에서 이름이 추출되지만 다른 곳에서 추출될 수도 있다. 전달 인수가 없으면 현재 지역 기호 표(symbol table)에서 이름이 반환된다. 이 함수는 주로 정보를 제공하는 목적으로 사용되어야 한다(예를 들어, 대화식 명령줄에서 사용된다든지). 이 함수가 반환하는 정보가 불완전할 수 있기 때문에 프로그램을 엄밀하게 분석하는 데 사용해서는 안 된다. 사용자 정의 클래스에서는 이 함수의 호출 결과를 변경하기 위해서 특수 메서드 `__dir__()`를 정의 할 수 있다.

divmod(a, b)

나누기의 몫과 나머지를 튜플로서 반환한다. 정수에 대해서는 `(a // b, a% b)`가 반환된다. 실수에 대해서는 `(math.floor(a / b), a % b)`가 반환된다. 이 함수를 복소수에 대해서 호출하면 안 된다.

enumerate(iter [, initial value])

반복 가능한 객체인 iter가 주어졌을 때 카운트와 iter가 생성한 값을 담은 튜플을 생성하는 새로운 반복자(타입은 `enumerate`)를 반환한다. iter가 `a, b, c`를 생성한다고 하면, `enumerate(iter)`는 `(0, a), (1, b), (2, c)`를 생성한다.

eval(expr [, globals [, locals]])

표현식을 평가한다. expr는 문자열이나 `compile()`에 의해서 생성되는 코드 객체이다. globals와 locals는 각각 평가에 쓰일 전역 네임스페이스와 지역 네임스페이스를 정의하는 매핑 객체이다. 생략될 경우 표현식은 호출하는 쪽의 네임스페이스에서 평가된다. 보통 globals와 locals에 사전을 넘겨주지만 고급 응용 프로그램에서 는 자신만의 매핑 객체를 사용하기도 한다.

exec(code [, global [, locals]])

파이썬 문장들을 실행한다. code는 문자열, 파일, 또는 `compile()`에 의해서 생성되는 코드 객체일 수 있다. globals와 locals는 각각 전역 네임스페이스와 지역 네임스페이스를 나타낸다. 생략되면 코드는 호출하는 쪽의 네임스페이스에서 실행된

다. `globals`와 `locals`가 주어졌을 때 이 함수가 작동하는 방식은 파이썬 버전마다 다를 수 있다. 파이썬 2에서 `exec`는 특수한 문장으로서 구현되어 있는 반면에 파이썬 3에서는 표준 라이브러리 함수로서 구현되어 있다. 이러한 구현상의 차이점으로 인해 파이썬 2에서는 `exec`에 의해 평가되는 코드에서 호출하는 쪽의 네임스페이스에 있는 지역 변수를 원하는 대로 수정할 수 있다. 파이썬 3에서도 그러한 수정을 할 수 있지만 `exec()` 호출을 넘어서까지 효과가 미치지는 않는 것 같다. 그 이유는 파이썬 3에서는 `locals`가 주어지지 않을 경우 지역 네임스페이스를 얻어오기 위해 `locals()`를 호출하기 때문이다. `locals()`의 문서를 보면 반환되는 사전은 들여다 볼 수만 있지 수정할 수 없다고 되어 있다.

filter(function, iterable)

파이썬 2에서 이 함수는 `iterable`에서 `function`이 참으로 평가되는 객체로만 구성된 리스트를 생성한다. 파이썬 3에서는 리스트 대신 반복자를 반환한다. `function`이 `None`이면 항등 함수가 사용되고 `iterable`에서 거짓으로 평가되는 원소가 제거된다. `iterable`은 반복을 지원하는 어떤 객체든 될 수 있다. 일반적으로 데이터를 거르기 위해서는 생성기 표현식이나 리스트 내포를 사용하는 것이 훨씬 빠르다(6장 참조).

float([x])

부동 소수점 수를 나타내는 타입. `x`가 숫자이면 실수로 변환된다. `x`가 문자열이면 실수로 파싱된다. 전달 인수가 없으면 0.0이 반환된다.

format(value [, format_spec])

`format_spec`으로 지정된 포맷 지정 문자열에 따라 `value`를 포맷이 적용된 문자열로 변환한다. 이 연산은 `value.__format__()`을 호출하며, 이 메서드 안에서 원하는 대로 포맷 지정 내용을 해석할 수 있다. 간단한 타입의 데이터에 대해서 포맷 지정자는 보통 ‘‘(,’’ ’^’ 같은 정렬 문자, 숫자(필드 폭을 지정), 그리고 각각 정수, 부동 소수점, 문자열 값을 나타내는 문자 코드 ‘d’, ‘f’, ‘s’를 포함한다. 예를 들어, 포맷 지정자 ‘d’는 정수로 포맷을 적용하고 ‘8d’는 여덟 문자 필드에서 정수를 오른쪽 정렬하며 ‘<8d’는 여덟 문자 필드에서 정수를 왼쪽 정렬한다. `format()`과 포맷 지정자는 3장과 4장에 자세하게 설명되어 있다.

frozenset([items])

반복 가능한 객체이어야 하는 `items`로부터 값을 가져와서 만든 변경 불가능한 집

합 객체를 나타내는 타입. 값들 또한 변경 불가능해야 한다. 전달 인수가 주어지지 않으면 빈 집합이 반환된다.

getattr(object, name [,default])

객체에서 주어진 이름을 가진 속성을 반환한다. name은 속성 이름을 담은 문자열이다. 생략 가능한 default는 해당 속성이 없을 경우 반환될 값이다. 아니면 AttributeError 예외가 발생한다. object.name과 동일하다.

globals()

전역 네임스페이스를 나타내는 현재 모듈의 사전을 반환한다. 다른 함수나 메서드 안에서 호출되면 해당 함수나 메서드가 정의된 모듈의 전역 네임스페이스를 반환한다.

hasattr(object, name)

name이 object의 속성이라면 True를 반환한다. 아니면 False를 반환한다. name은 문자열이다.

hash(object)

객체의 정수 해시 값을 반환한다(가능할 경우). 해시 값은 주로 사전, 집합, 기타 매핑 객체를 구현하는 데 사용된다. 비교해서 같은 두 객체는 해시 값이 동일하다. 변경 가능한 객체에는 해시 값이 없다. 사용자 정의 클래스에서는 __hash__() 메서드를 정의해서 이 연산을 지원할 수 있다.

help([object])

대화식 세션에서 내장 도움말 시스템을 호출한다. object는 모듈, 함수, 메서드 이름 또는 키워드나 주제를 담은 문자열이어야 한다. object가 이것들 말고 다른 객체이면 해당 객체와 관련된 도움말 화면이 나타난다. 인수가 주어지지 않으면 대화식 도움말 도구가 구동되어 추가 정보를 제공한다.

hex(x)

정수 x의 16진수 문자열을 생성한다.

id(object)

object의 고유한 심원을 나타내는 정수를 반환한다. 반환 값을 어떤 식으로든 해석해서는 안 된다(메모리 위치로 해석한다든지).

input([prompt])

파이썬 2에서는 prompt를 출력하고 입력을 한 줄 읽어서 eval()로 처리한다(즉, eval(raw_input(prompt))와 동일하다). 파이썬 3에서는 prompt가 표준 출력으로 출력된 후 평가나 수정 없이 한 줄을 읽는다. 반환되는 줄에는 줄바꿈 문자가 포함되지 않는다.

int(x [,base])

정수를 나타내는 타입. x가 숫자이면 0 쪽으로 자르면서 정수로 변환한다. 문자열이면 정수 값으로 파싱된다. base는 추가적으로 문자열을 정수로 변환할 때 사용할 기본수를 지정한다. 파이썬 2에서는 값이 int 타입의 32비트 범위를 벗어날 경우 긴 정수가 생성된다.

isinstance(object, classobj)

object가 classobj 또는 classobj의 하위 클래스의 인스턴스이거나 추상 기반 클래스인 classobj에 속할 경우 True를 반환한다. classobj 매개변수는 타입이나 클래스들의 튜플일 수도 있다. 예를 들어, isinstance(s, (list,tuple))은 s가 튜플이거나 리스트이면 True를 반환한다.

issubclass(class1, class2)

class1이 class2의 하위 클래스(또 파생 클래스일 경우)이거나 class1이 추상 기반 클래스인 class2에 등록되어 있는 경우 True를 반환한다. class2는 클래스들의 튜플일 수 있다. 이 경우 각 클래스에 대해서 검사가 수행된다. issubclass(A, A)는 참이다.

iter(object [,sentinel])

object에 있는 항목을 생성하는 반복자를 반환한다. sentinel 매개변수가 생략되면, object가 반복자를 생성하는 __iter__() 메서드를 제공하거나, 0부터 시작하는 정수를 받는 __getitem__()을 구현하든지 둘 중 하나는 반드시 해야 한다. sentinel이 지정되면 object는 다르게 해석된다. 이 경우 object는 매개변수를 받지 않는 호출 가능한 객체이여야 한다. 반환되는 반복자 객체는 반환되는 값이 sentinel과 동일할 때까지 이 함수를 반복적으로 호출하고 이 시점이 되면 반복을 멈춘다. object가 반복을 지원하지 않으면 TypeError 예외가 발생한다.

len(s)

s에 들어 있는 항목 개수를 반환한다. s는 리스트, 튜플, 문자열, 집합 또는 사전

이어야 한다. `s`가 생성기 같은 반복 가능한 객체일 경우에는 `TypeError`가 발생한다.

`list([items])`

리스트를 나타내는 타입. `items`는 아무 반복 가능한 객체나 될 수 있고 이 반복 가능한 객체의 값이 리스트를 채우는 데 사용된다. `items`가 이미 리스트이면 복사본이 만들어진다. 전달 인수가 없으면 빈 리스트가 반환된다.

`locals()`

호출하는 쪽의 지역 네임스페이스에 해당하는 사전을 반환한다. 이 사전은 실행 환경을 살펴보는 데만 사용해야 한다. 이 사전의 내용을 수정하는 일은 안전하지 않다.

`long([x [, base]])`

파이썬 2에서 긴 정수를 나타내는 타입. `x`가 숫자이면 0 쪽으로 자르면서 정수로 변환한다. `x`가 문자열이면 긴 정수 값으로 파싱된다. 인수가 주어지지 않으면 이 함수는 `OL`을 반환한다. 호환성 문제가 있을 수 있기 때문에 `long`을 직접 사용하지 않아야 한다. `int(x)`를 사용하면 필요에 따라 `long`이 생성된다. `x`가 정수 타입인지를 검사하려면 `isinstance(x, numbers.Integral)`을 사용하도록 한다.

`map(function, items, ...)`

파이썬 2에서 이 함수는 `function`을 `items`의 각 항목에 적용한 결과 리스트를 반환한다. 파이썬 3에서는 동일한 결과를 생성하는 반복자가 생성된다. 여러 입력 순서열을 제공하면 `function`이 그만큼의 인수를 받는다고 가정하고 각 인수를 각 순서열에서 가져온다. 여러 입력 순서열을 처리할 때 파이썬 2와 파이썬 3의 작동 방식이 다르다. 파이썬 2에서 결과는 가장 긴 입력 순서열의 길이와 같으며 짧은 입력 순서열이 다 소진될 경우 `None`이 채우기 값으로 사용된다. 파이썬 3에서 결과는 가장 짧은 순서열의 길이가 된다. `map()`이 제공하는 기능은 거의 모든 경우에 생성기 표현식이나 리스트 내포를 사용해서 표현하는 것이 더 낫다(둘 다 더 좋은 성능을 보인다). 예를 들어, `map(function, s)`은 보통 `[function(x) for x in s]`로 쓸 수 있다.

`max(s [, args, ...])`

`s`라는 인수가 하나만 주어지면 이 함수는 `s`에 있는 항목 중 가장 큰 값을 반환한다. `s`는 반복 가능한 아무 객체라도 될 수 있다. 여러 인수가 주어지면 인수들 중에 가장 큰 값이 반환된다.

min(s [, args, ...])

s라는 인수가 주어지면 이 함수는 s에 있는 항목 중 가장 작은 값을 반환한다. s는 반복 가능한 아무 객체라도 될 수 있다. 여러 인수가 주어지면 인수들 중에 가장 작은 값이 반환된다.

next(s [, default])

반복자 s에서 다음 항목을 반환한다. 반복자에 항목이 더 없으면, default 인수로 값을 지정하지 않은 이상 StopIteration 예외가 발생한다. default 인수가 주어지면 default가 대신 반환된다. 호환성 문제가 발생할 수 있기 때문에 반복자 s에 직접 s.next()를 호출하는 대신 항상 이 함수를 사용하여야 한다. 파이썬 3에서는 내부 반복자 메서드의 이름이 s.__next__()로 변경되었다. 코드에서 next() 함수를 사용하는 경우에는 이 차이점에 대해서 신경쓸 필요가 없다.

object()

파이썬에서 모든 객체의 기본 클래스. 인스턴스를 생성하는 데 이 함수를 호출할 수 있지만 특별히 흥미로운 일이 일어나지는 않는다.

oct(x)

정수 x를 8진수 문자열로 변환한다.

open(filename [, mode [, bufsize]])

파이썬 2에서 이 함수는 filename 파일을 열고 새로운 파일 객체를 반환한다(9장을 참고할 것). mode는 파일을 어떻게 열지를 지시하는 문자열이다. ‘r’은 읽기용으로, ‘w’는 쓰기용으로, ‘a’는 추가용으로 파일을 연다. 두 번째 문자인 ‘t’나 ‘b’는 텍스트 모드(기본 모드)인지 이진 모드인지를 나타낸다. 예를 들어, ‘r’이나 ‘rt’는 파일을 텍스트 모드에서 열고 ‘rb’는 파일을 이진 모드에서 연다. 옵션인 ‘+’를 모드에 추가하면 파일을 수정을 위해서 열게 된다(읽기와 쓰기 둘 다 가능하다). ‘w+’ 모드는 파일이 이미 있으면 파일의 길이를 0으로 줄인다. ‘r+’와 ‘a+’는 파일을 읽기와 쓰기 모두를 위해 열지만 파일이 열릴 때 원래 내용을 그대로 둔다. ‘U’나 ‘rU’가 사용되면 보편 줄바꿈 모드로 파일이 열린다. 이 모드에서는 모든 줄바꿈 문자(‘\n’, ‘\r’, ‘\r\n’)가 표준 ‘\n’ 문자로 변환된다. 모드를 생략하면 ‘rt’가 기본으로 사용된다. bufsize 인수는 버퍼링의 작동 방식을 결정하며, 0은 버퍼링하지 않는 것을 의미하고 1은 줄 버퍼링을 의미하며 기타 다른 값은 바이트로 버퍼의 대략적인 크기를 가리킨다. 음수 값은 시스템의 기본 버퍼링 방식을 사용한다는 것을 의미한다(버퍼

링 방식을 지정하지 않으면 기본으로 시스템 기본 버퍼링 방식을 사용한다).

open(filename [, mode [, bufsize [, encoding [, errors [, newline [, closefd]]]]])

파이썬 3에서 이 함수는 filename 파일을 열고 새로운 파일 객체를 반환한다. 앞쪽 세 개의 인수는 앞에서 설명한 파이썬 2 버전의 open()과 동일한 의미를 가진다. encoding은 ‘utf-8’ 같은 인코딩 이름을 나타낸다. errors는 에러 처리 정책을 나타내며, ‘strict’, ‘ignore’, ‘replace’, ‘backslashreplace’, ‘xmlcharrefreplace’ 중 하나의 값이 될 수 있다. newline은 보편 줄바꿈 모드의 작동 방식을 제어하며, None, ‘’, ‘\n’, ‘\r’, ‘\r\n’ 중 하나가 될 수 있다. closefd는 close()가 실행될 때 내부 파일 기술자를 닫을 것인지를 나타내는 불리언 플래그이다. 파이썬 2와 달리, 선택된 I/O 모드에 따라 다른 종류의 객체가 반환된다. 예를 들어, 이진 모드로 파일을 열면 read()나 write() 같은 I/O 연산을 문자열이 아니라 바이트 배열에 대고 수행하는 객체를 얻게 된다. 파일 I/O는 파이썬 2와 3에서 크게 다른 부분 중 하나이다. 더 자세한 정보는 부록을 참고하기 바란다.

ord(c)

단일 문자 c의 정수 서열 값을 반환한다. 보통의 문자에 대해서 [0, 255]의 범위에 있는 값이 반환된다. 단일 유니코드 문자에 대해서는 [0, 65535] 범위에 있는 값이 보통 반환된다. 파이썬 3에서 c는 유니코드 대리자 쌍일 수 있다. 이 경우 적절한 유니코드 코드 포인트로 변환된다.

pow(x, y [, z])

x ** y를 반환한다. z가 주어지면 (x ** y) % z를 반환한다. 모든 인수가 주어질 때 모두 정수여야 하고 y는 음수가 아니어야 한다.

print(value, ... [, sep=separator, end=ending, file=outfile])

여러 값을 출력하는 파이썬 3 함수. 입력으로 임의의 개수의 값을 줄 수 있으며 모두 동일한 줄에 출력된다. sep 키워드 인수는 분리 문자를 지정한다(기본은 공백 문자이다). end 키워드 인수는 줄 종료 문자를 지정한다(기본은 ‘\n’). file 키워드 인수는 출력을 파일 객체로 돌린다. 파이썬 2에서도 from __future__ import print_function문을 코드에 추가해서 이 함수를 사용할 수 있다.

property([fget [,fset [,fdel [,doc]]]])

클래스에 프로퍼티 속성을 생성한다. fget은 속성 값을 반환하는 함수이고

fset은 속성 값을 설정하는 함수이며 fdel은 속성을 삭제하는 함수이다. doc는 문서화 문자열이다. 이 인수들은 키워드 인수로도 지정할 수 있다. 예를 들어, `property(fget=getX, doc="some text")`.

range([start,] stop [, step])

파이썬 2에서 이 함수는 start부터 stop까지 완전히 채워진 정수 리스트를 생성한다. step은 보폭을 나타내고 생략될 경우 1이 된다. start를 생략하면(`range()`)가 인수 하나로 호출될 때) 기본으로 0으로 설정된다. 음의 step 값은 감소하는 순서로 숫자를 생성한다. 파이썬 3에서 `range()`는 필요에 따라 값을 계산하는 특수한 `range` 객체를 생성한다(이전 파이썬 버전에서 `xrange()`처럼).

raw_input([prompt])

표준 입력(`sys.stdin`)에서 한 줄을 읽어서 문자열로 반환하는 파이썬 2 함수. `prompt`가 제공되면 이것이 먼저 표준 출력(`sys.stdout`)으로 출력된다. 끝에 있는 줄 바꿈 문자는 벗겨지며 EOF(파일 끝)가 읽히면 `EOFError` 예외가 발생한다. `readline` 모듈이 로드되면 이 함수는 고급 줄 편집이나 명령 완성 기능 같은 기능을 제공하기 위해 `readline` 모듈을 사용한다. 파이썬 3에서 입력을 받아들이려면 `input()`을 사용해야 한다.

repr(object)

`object`의 문자열 표현을 반환한다. 대부분 반환되는 문자열은 `eval()`에 넘길 경우 객체를 다시 생성하는 표현식이다. 파이썬 3에서는 이 함수의 반환값이 터미널이나 셸 창에 출력할 수 없는(예외가 발생한다) 유니코드 문자열일 수도 있다. `object`의 ASCII 표현을 생성하려면 `ascii()` 함수를 사용하도록 한다.

reversed(s)

순서열 `s`의 역순 반복자를 생성한다. 이 함수는 `s`에서 순서열 메서드인 `__len__()`과 `__getitem__()`을 구현하고 있을 때만 작동한다. 또한 `s`는 항목의 색인을 0부터 시작해야 한다. 생성기나 반복자에 대해서는 작동하지 않는다.

round(x [, n])

부동 소수점 수 `x`를 10의 `n` 제곱의 가장 가까운 배수로 반올림한다. `n`을 입력하지 않으면 `n`은 0으로 설정된다. `x`가 두 배수에 동일하게 가까우면 파이썬 2에서는 `x`가 0에서 더 멀리 떨어진 배수로 반올림된다(0.5의 경우 1.0으로 반올림되며, -0.5는 -1.0으로 반올림된다). 파이썬 3에서는 앞의 숫자가 짹수이면 0 쪽으로 반올림되고

아닐 경우에는 0에서 멀어지는 쪽으로 반올림된다(예를 들어, 0.5는 0.0으로 반올림되며, 1.5는 2.0으로 반올림된다).

set([items])

반복 가능한 객체 items에서 가져온 항목으로 채워진 집합을 생성한다. 항목들은 수정이 불가능해야 한다. items에 집합이 들어 있다면 그 집합은 frozenset 타입이어야 한다. items가 생략되면 빈 집합이 반환된다.

setattr(object, name, value)

객체의 속성을 설정한다. name은 문자열이다. object.name = value와 동일하다.

slice([start,] stop [, step])

주어진 범위에 있는 정수들을 나타내는 분할 객체를 반환한다. 분할 객체는 확장 분할 문법 a[i:j:k]에 의해서도 생성된다. 자세한 내용은 3장 ‘순서열과 매핑 메서드’절을 참고하기 바란다.

sorted(iterable [, key=keyfunc [, reverse=reverseflag]])

iterable에 있는 항목들로부터 정렬된 리스트를 생성한다. 키워드 인수인 key는 비교 함수로 전달되기 전에 값을 변환하는 데 사용할 단일 인수 함수이다. 키워드 인수 reverse는 결과 리스트가 역순으로 정렬되어야 하는지 여부를 지정한다. key와 reverse 인수는 반드시 키워드 인수로 지정해야 한다. 예를 들어, sorted(a, key=get_name).

staticmethod(func)

클래스에서 사용할 정적 메서드를 생성한다. @staticmethod 장식자에 의해서 암묵적으로 호출된다.

str([object])

문자열을 나타내는 타입. 파이썬 2에서 문자열은 8 비트 문자들을 담고 파이썬 3에서는 문자열이 유니코드 문자열이다. object가 제공되면 object의 __str__() 메서드를 호출해서 문자열 표현을 생성한다. 이 문자열은 해당 객체를 print문으로 출력했을 때 보게 되는 것과 같다. 인수가 주어지지 않으면 빈 문자열이 생성된다.

sum(items [,initial])

반복 가능한 객체 items로부터 추출된 항목들의 합을 계산한다. initial은 시작 값을 나타내고 기본 값은 0이다. 이 함수는 숫자에 대해서만 제대로 작동한다.

super(type [, object])

type의 상위 클래스를 나타내는 객체를 반환한다. 주로 기반 클래스에 있는 메서드를 호출할 때 사용된다. 다음은 한 예를 보여준다.

```
class B(A):
    def foo(self):
        super(B, self).foo()
```

object가 객체이면 `isinstance(object, type)`은 반드시 참이어야 한다. object가 타입이면 type의 하위 클래스이어야 한다. 더 자세한 내용은 7장에 나와 있다. 파이썬 3에서는 `super()`를 메서드에서 인수 없이 호출할 수도 있다. 이 경우 type은 메서드가 정의된 클래스로 설정되고 object는 메서드의 첫 번째 인수로 설정된다. 이렇게 되면 문법이 깔끔해지지만 파이썬 2와는 호환되지 않기 때문에 호환성을 고려한다면 사용하지 않는 것이 좋다.

tuple([items])

튜플을 나타내는 타입. items가 주어지면 items는 튜플의 내용을 채우는 데 사용될 반복 가능한 객체이어야 한다. items가 이미 튜플이면 간단히 수정 없이 그대로 반환된다. 아무런 인수도 주어지지 않으면 빈 튜플이 반환된다.

type(object)

파이썬에서 모든 타입의 기반 클래스이다. 함수처럼 호출되면 object의 타입을 반환한다. 이렇게 반환되는 타입은 object의 클래스와 동일하다. 정수, 실수, 리스트 같은 공통 타입에 대해서 반환되는 타입은 int, float, list 같은 내장 클래스 중 하나이다. 사용자 정의 객체에 대해서 반환되는 타입은 연관된 클래스가 된다. 파이썬의 내부 작동과 관련 있는 객체에 대해서 반환되는 타입은 보통 types 모듈에 정의된 클래스 중 하나가 된다.

type(name,bases,dict)

새로운 type 객체를 생성한다(새로운 클래스를 정의하는 것과 동일하다). name은 타입의 이름, bases는 기반 클래스들의 튜플, dict는 클래스 몸체에 있는 정의들을 담은 사전이다. 이 함수는 메타클래스와 관련해서 가장 많이 사용된다. 이 함수에 관해서는 7장에 자세하게 설명되어 있다.

unichr(x)

$0 \leq x \leq 65535$ 인 정수나 긴 정수 x를 단일 유니코드 문자로 변환한다. 파이썬

2에만 있다. 파이썬 3에서는 간단히 chr(x)를 사용한다.

unicode(string [,encoding [,errors]])

파이썬 2에서 이 함수는 string을 유니코드 문자열로 변환한다. encoding은 string의 데이터 인코딩을 기술한다. 생략하면 sys.getdefaultencoding()이 반환하는 기본 인코딩이 사용된다. error는 인코딩 에러를 어떻게 처리할 것인지를 기술하며 ‘strict’, ‘ignore’, ‘replace’, ‘backslashreplace’, ‘xmlcharrefreplace’ 중 하나가 될 수 있다. 더 자세한 내용은 9장과 3장을 참고하도록 한다. 파이썬 3에는 없다.

vars([object])

object의 기호 표(symbol table)를 반환한다(보통 __dict__ 속성에 저장되어 있는 것). 인수가 주어지지 않으면 지역 네임스페이스에 해당하는 사전을 반환한다. 이 함수에 의해서 반환되는 사전은 읽기 전용으로 취급되어야 한다. 내용을 변경하는 것은 안전하지 못하다.

xrange([start,] stop [, step])

start에서 stop에 해당하는 정수 값 범위를 나타내는 타입. stop은 범위에 포함되지 않는다. 값들이 실제로 저장되는 것은 아니고 접근될 때마다 필요에 따라 계산된다. 파이썬 2에서 xrange()는 정수 값 범위에 대해 루프를 작성할 때 선호되는 함수이다. 파이썬 3에서는 xrange()가 range()로 이름이 바뀌었고 xrange()는 이제 없다. start, stop과 step에는 머신에서 지원하는 정수만 올 수 있다(보통 32비트).

zip([s1 [, s2 [,...]])]

파이썬 2에서 이 함수는 n번째 튜플이 (s1[n], s2[n], ...)인 튜플 리스트를 반환한다. 결과로 생성되는 리스트는 가장 짧은 순서열의 길이로 잘린다. 인수가 주어지지 않으면 빈 리스트가 반환된다. 파이썬 3에서는 작동 방식이 비슷하지만 결과로 튜플들을 생성하는 반복자가 반환된다. 파이썬 2에서 긴 입력 순서열에 대해 zip()을 사용할 경우 의도하지 않게 많은 메모리를 사용하게 될 수도 있다. itertools.izip()을 대신 사용하는 것을 고려해보도록 한다.

내장 예외

내장 예외는 exceptions 모듈에 들어 있고 프로그램이 실행되기 전에 항상 로드된다. 예외는 클래스로서 정의된다.

예외 기반 클래스

다음 예외들은 다른 모든 예외의 기반 클래스 역할을 수행한다.

BaseException

모든 예외의 조상 클래스. 모든 내장 클래스는 이 클래스에서 파생된다.

Exception

프로그램과 관련된 모든 예외를 위한 기반 클래스. SystemExit, GeneratorExit 와 KeyboardInterrupt를 제외한 모든 내장 예외를 나타낸다. 사용자 정의 예외는 Exception에서 상속해서 정의해야 한다.

ArithmetricError

OverflowError, ZeroDivisionError와 FloatingPointError 등 모든 산술 예외를 위한 기반 클래스.

LookupError

IndexError와 KeyError를 포함한 색인과 키 에러를 위한 기반 클래스.

EnvironmentError

IOError와 OSError를 포함한 파일의 외부에서 발생하는 에러를 위한 클래스.

앞에 나온 예외는 실제로 던져지지 않으며 특정 종류의 에러를 잡는 용도로 사용된다. 예를 들어, 다음 코드는 모든 수와 관련된 에러를 잡아낸다.

```
try:
    # 몇몇 연산
    ...
except ArithmetricError as e:
    # 수리 에러
```

예외 인스턴스

예외가 발생하면 예외의 인스턴스가 생성된다. 이 인스턴스는 except문에서 추가로 지정한 변수에 저장된다. 다음 예를 보자.

```
except IOError as e:
    # 에러를 처리한다.
    # 'e'는 IOError의 인스턴스를 담는다.
```

예외 e의 인스턴스에는 응용 프로그램에서 예외의 내용을 살펴보고 수정하는 데 유용한 몇몇 표준 속성이 있다.

e.args

예외가 발생될 때 전달된 인수들의 튜플. 대부분 예러를 설명하는 문자열을 담은 항목 한 개짜리 튜플이다. EnvironmentError 예외에 대해서는 정수 예러 숫자, 문자열 예러 메시지와 옵션인 파일 이름을 담은 항목 2 개 또는 3개짜리 튜플이다. 튜플의 내용은 예외를 다른 컨텍스트에서 다시 생성하고자 할 때 유용하게 쓰일 수 있다. 예를 들어, 다른 파이썬 인터프리터 프로세스에서 예외를 다시 던지는 경우를 생각해볼 수 있다.

e.message

예외를 출력할 경우 보게 되는 예러 메시지를 담은 문자열(파이썬 2에만 있다).

e.__cause__

명시적 연쇄 예외를 사용할 때 이전 예외(파이썬 3에만 있다). 부록 참고.

e.__context__

암묵적 연쇄 예외를 사용할 때 이전 예외(파이썬 3에만 있다). 부록 참고.

e.__traceback__

예외와 연관된 역추적 객체(파이썬 3에만 있다)). 부록 참고.

미리 정의된 예외 클래스

프로그램에서는 다음 예외들이 발생한다.

AssertionError

assert문 실패.

AttributeError

속성 참조 또는 할당 실패.

EOFError

파일의 끝. 내장 함수인 input()과 raw_input()에 의해서 생성된다. 파일 객체의 read()나 readline() 메서드 같은 대부분의 I/O 연산은 파일의 끝을 나타내는 데 예외를 던지지 않고 빈 문자열을 반환한다는 사실에 주의한다.

FloatingPointError

부동 소수점 연산 실패. 부동 소수점 예외 처리는 쉽지 않은 문제이며 파이썬이

예외를 던지도록 활성화되어 있을 경우 이 예외가 던져진다. 부동 소수점 관련 에러가 조용히 float('nan')이나 float('inf') 같은 결과를 생성하는 경우가 더 흔하다. ArithmeticError의 하위 클래스이다.

GeneratorExit

종료를 표시하기 위해 생성기 함수 안에서 발생한다. 생성기가 일찍 파괴될 경우(생성기의 모든 값이 소모되기 전)나 생성기의 close() 메서드가 호출될 경우에 발생한다. 생성기에서 이 함수를 무시할 경우 생성기는 종료되고 예외는 조용히 무시된다.

IOError

I/O 연산 실패. IOError 인스턴스는 속성 errno, strerror와 filename을 갖는다. errno는 정수 예러 숫자를 나타내고 strerror은 문자열 예러 메시지를, filename은 옵션인 파일 이름을 나타낸다. EnvironmentError의 하위 클래스이다.

ImportError

import문이 모듈을 찾지 못했거나 from이 모듈에서 이름을 찾지 못할 경우 발생한다.

IndentationError

들여쓰기 예러. SyntaxError의 하위 클래스.

IndexError

순서열의 색인이 범위를 벗어남. LookupError의 하위 클래스.

KeyError

매핑 객체에서 키가 발견되지 않음. LookupError의 하위 클래스.

KeyboardInterrupt

사용자가 인터럽트 키(보통 Ctrl+C)를 입력했을 때 발생.

MemoryError

회복이 가능한 메모리 부족 예러.

NameError

지역 네임스페이스나 전역 네임스페이스에서 이름을 찾을 수 없음.

NotImplementedError

구현 안 된 기능. 파생 클래스에서 필요한 메서드를 구현하지 않았을 경우 기반 클래스에 의해서 발생될 수 있다. RuntimeError의 하위 클래스.

OSError

운영체제 에러. os 모듈에 있는 함수에 의해서 주로 발생한다. OSError의 인스턴스는 IOError와 동일한 속성을 갖는다.

OverflowError

정수 값이 너무 커서 표현할 수 없음. 보통 구현상 내부적으로 고정 정밀도 머신 정수에 의존하는 경우에만 발생됨. 예를 들어, range나 xrange 객체에 시작 값이나 종료 값으로 32비트보다 더 큰 값을 줄 때 발생할 수 있다. ArithmeticError의 하위 클래스.

ReferenceError

내부 객체가 파괴된 후 약한 참조에 접근하려고 할 때 발생. weakref 모듈을 참고한다.

RuntimeError

특별한 범주에 들지 않는 일반적인 에러.

StopIteration

반복의 끝을 표시하기 위해 발생됨. 보통 객체의 next() 메서드나 생성기 함수에서 발생함.

SyntaxError

파서 문법 에러. SyntaxError의 인스턴스에는 추가 정보를 제공하는 filename, lineno, offset과 text 속성이 있다.

SystemError

인터프리터의 내부 에러. 문제를 설명하는 문자열 값을 담는다.

SystemExit

sys.exit() 함수에 의해 발생된다. 반환 코드를 나타내는 정수를 담는다. 즉시 종료해야 할 경우에는 os._exit()를 사용할 수 있다.

TabError

일관성 없는 탭 문자 사용. -tt 옵션으로 파이썬이 실행되었을 때 발생한다.

SyntaxError의 하위 클래스.

TypeError

연산이나 함수가 적절하지 않은 타입의 객체에 적용되었을 경우 발생한다.

UnboundLocalError

안 묶인 지역 변수가 참조되었다. 이 에러는 함수 안에서 정의되기 전에 변수가 참조되었을 때 발생한다. NameError의 하위 클래스.

UnicodeError

유니코드 인코딩 또는 디코딩 에러. ValueError의 하위 클래스.

UnicodeEncodeError

유니코드 인코딩 에러. UnicodeError의 하위 클래스.

UnicodeDecodeError

유니코드 디코딩 에러. UnicodeError의 하위 클래스.

UnicodeTranslateError

변환 도중 유니코드 관련 에러가 발생. UnicodeError의 하위 클래스.

ValueError

함수나 연산의 인수로 적절한 타입이 전달되었으나 적절하지 못한 값일 때 발생.

WindowsError

윈도에서 시스템 호출이 실패했을 때 발생. OSError의 하위 클래스.

ZeroDivisionError

영으로 나누기. ArithmeticError의 하위 클래스.

내장 경고

파이썬에는 warnings 모듈이 있다. 이 모듈은 보통 프로그래머에게 사용이 권장되지 않는 기능을 알리는 데 사용된다. 다음과 같은 코드로 경고를 줄 수 있다.

```
import warnings
warnings.warn("The MONDO flag is no longer supported",
DeprecationWarning)
```

라이브러리 모듈을 통해 경고를 생성하기는 하지만 경고 이름 자체는 내장 이름이다. 경고는 예외와 어느 정도 비슷하다. 내장 경고는 모두 Exception에서 상속을 받고 계층 구조를 가진다.

Warning

모든 경고의 기반 클래스. Exception의 하위 클래스.

UserWarning

일반적인 사용자 정의 경고. Warning의 하위 클래스.

DeprecationWarning

사용이 권장되지 않는 기능에 대한 경고. Warning의 하위 클래스.

SyntaxWarning

사용이 권장되지 않는 파이썬 문법에 대한 경고. Warning의 하위 클래스.

RuntimeWarning

잠재적인 런타임 문제에 대한 경고. Warning의 하위 클래스.

FutureWarning

미래에 바뀔 수 있는 기능의 작동 방식에 대한 경고. Warning의 하위 클래스.

warn() 함수로 경고를 생성한다고 하더라도 프로그램이 멈출 수도 있고 멈추지 않을 수도 있다는 측면에서 경고는 예외와 다르다. 예를 들어, 경고는 간단히 무언가를 출력하기만 하기도 하고 예외를 생성하기도 한다. 실제 작동 방식은 warnings 모듈을 사용하거나 인터프리터에 -W 옵션을 통해서 설정할 수 있다. 경고를 생성하는 다른 사람의 코드를 사용하는 중이고 경고를 받더라도 계속 진행하고 싶은 경우에는 예외로 바뀐 경고를 try와 except로 잡으면 된다. 다음 예를 보자.

```
try:
    import md5
except DeprecationWarning:
    pass
```

이런 식으로 코드를 작성하는 일은 드물다. 예외로 바뀐 경고를 잡는다고 하더라도 경고 메시지가 출력되는 것을 막지는 못한다(이 부분까지 제어하려면 warnings 모듈을 사용해야 한다). 또한 경고를 무시하면 새로운 버전의 파이썬이 릴리스되었을 때 코드가 제대로 작동하지 않을 수도 있다.

future_builtins

파이썬 2에만 있는 future_builtins 모듈은 파이썬 3에서 작동 방식이 변경된 내장 함수 구현을 모아놓은 것이다. 다음 함수들이 정의되어 있다.

ascii(object)

repr()와 동일한 결과를 생성한다. 이 장의 ‘내장 함수’ 절에 있는 설명을 참고하도록 한다.

filter(function, iterable)

리스트 대신 반복자를 생성한다. itertools.ifilter()와 동일하다.

hex(object)

16진수 문자열을 생성하지만 정수 값을 얻는 데 __hex__()를 호출하는 대신 특수 메서드 __index__()를 사용한다.

map(function, iterable, ...)

리스트 대신 반복자를 생성한다. itertools imap()와 동일하다.

oct(object)

8 진수 문자열을 생성하지만 정수 값을 얻는 데 __oct__() 대신 특수 메서드 __index__()를 사용한다.

zip(iterable, iterable, ...)

리스트 대신 반복자를 생성한다. itertools izip()와 동일하다.

내장 모듈에서 변경된 부분은 여기에 나열되어 있는 함수가 전부는 아니다. 예를 들어, 파이썬 3에서는 raw_input() input()으로, xrange()는 range()로 이름이 변경되었다.

13장

Python Essential Reference

파이썬 런타임 서비스

이 장에서는 파이썬 인터프리터 런타임과 관련 있는 모듈을 설명한다. 여기에는 쓰레기 수집, 기본적인 객체 관리(복사, 마샬링marshalling 등), 약한 참조, 인터프리터 환경 등이 포함된다.

atexit

atexit 모듈은 파이썬 인터프리터가 종료될 때 실행될 함수를 등록하는 데 사용한다. 이 모듈에는 함수가 하나 들어 있다.

```
register(func [,args [,kwargs]])
```

인터프리터가 종료될 때 실행될 함수 목록에 func을 추가한다. args는 함수에 전달될 인수들의 튜플이다. kwargs는 키워드 인수들의 사전이다. 함수는 func(*args, **kwargs) 형태로 호출된다. 인터프리터 종료 시 함수는 등록된 순서의 역순으로 호출된다(가장 최근에 추가된 종료 함수가 먼저 호출된다). 에러가 발생하면 예외 메시지가 표준 에러로 출력되지만 에러는 무시된다.

copy

copy 모듈은 리스트, 튜플, 사전, 사용자 정의 클래스 인스턴스 등 복합 객체에 대한 얕은 복사본이나 깊은 복사본을 생성하는 데 사용할 수 있는 함수들을 제공한다.

copy(x)

새로운 복합 객체를 만든 후 참조를 통해 x의 구성 요소를 복제함으로써 x에 대한 얇은 복사본을 생성한다. 내장 타입에 대해서 이 함수를 사용하는 일은 드물다. 대신 x에 대한 얇은 복사본을 생성할 때는 list(x), dict(x), set(x) 같은 함수를 호출한다(이렇게 타입 이름을 바로 쓰는 것이 copy()를 사용하는 것보다 속도도 훨씬 빠르다).

deepcopy(x [, visit])

새로운 복합 객체를 만든 후 x의 구성 요소를 재귀적으로 복제함으로써 x의 깊은 복사본을 생성한다. visit는 옵션인 인수로서 재귀적으로 정의되는 데이터 구조에서 순환 참조를 감지하여 피하기 위해서 방문 객체들을 추적하는 데 사용될 사전을 나타낸다. 이 인수는 이 장에서 나중에 설명하듯이 deepcopy()가 재귀적으로 호출될 때만 보통 사용한다. 보통은 필요하지 않지만, 클래스에서 얇은 복사 연산과 깊은 복사 연산을 구현하는 `__copy__(self)`와 `__deepcopy__(self, visit)`를 구현함으로써 복사 연산을 커스터마이즈할 수 있다. `__deepcopy__()` 메서드는 복사 과정 동안 만나게 되는 객체들을 추적하는 데 사용되는 사전 visit를 반드시 받아야 한다. `__deepcopy__()`에서는 구현상 호출하게 되는 다른 deepcopy()에 visit를 넘겨주는 일 말고 특별히 해야 할 일은 없다.

pickle 모듈에 의해서 사용되는 `__getstate__()`와 `__setstate__()`를 어떤 클래스에서 구현하고 있으면 copy 모듈을 통해 복사본을 생성하는 데 이 두 메서드가 대신 사용된다.

Note

- 이 모듈을 정수나 문자열 같은 간단한 타입에도 사용할 수 있지만 그래야 할 필요는 거의 없다.
- 복사 함수는 모듈, 클래스 객체, 함수, 메서드, 역추적, 스택 프레임, 파일, 소켓이나 기타 이와 비슷한 타입에 대해서는 작동하지 않는다. 객체가 복사될 수 없으면 `copy.error` 예외가 던져진다.

gc

gc 모듈은 리스트, 튜플, 사전, 인스턴스 같은 객체에 있는 순환 참조를 수집할 때

사용할 수 있는 쓰레기 수집기를 제어하기 위한 인터페이스를 제공한다. 다양한 컨테이너 객체들이 생성됨에 따라 이들은 인터프리터 내부에 있는 리스트에 저장된다. 컨테이너 객체는 반납될 때 이 리스트에서도 제거된다. 할당 수가 반납 수보다 사용자가 정의할 수 있는 임계값 이상이면 쓰레기 수집기가 호출된다. 쓰레기 수집기는 내부 리스트를 들여다보면서 더 이상 사용되지 않지만 순환 의존성 때문에 반납이 되지 못한 객체들을 찾는다. 쓰레기 수집기는 초기 쓰레기 수집 단계에서 살아남은 객체들을 덜 자주 검사하는 세 수준에 걸친 세대 기법을 사용한다. 이로 인해 오래 생존하는 객체의 수가 많은 경우 더 좋은 성능을 보인다.

collect([generation])

전체 쓰레기 수집을 수행한다. 이 함수는 모든 세대를 검사하고 도달할 수 없는 객체 수를 반환한다. generation은 수집할 세대를 나타내는 0에서 -2 사이의 범위를 가지는 옵션인 정수이다.

disable()

쓰레기 수집을 비활성화한다.

enable()

쓰레기 수집을 활성화한다.

garbage

더 이상 사용되지 않지만, 순환 참조에 묶여 있고 __del__() 메서드를 정의하고 있기 때문에 쓰레기 수집될 수 없는 사용자 정의 인스턴스들의 읽기 전용 리스트를 담은 변수. 이러한 객체들은 쓰레기 수집될 수 없다. 그 이유는 순환 참조를 깨기 위해서는 인터프리터에서 임의의 객체를 먼저 파괴해야 하지만, 순환 참조상에 있는 나머지 객체의 __del__() 메서드에서 지금 파괴할 객체에 대해 중요한 연산을 수행해야 할 필요가 있는지 여부를 알 수가 없기 때문이다.

get_count()

현재 각 세대에 있는 객체 수를 담은 튜플(count0, count1, count2)을 반환한다.

get_debug()

현재 설정된 디버깅 플래그를 반환한다.

get_objects()

쓰레기 수집기에 의해 추적되고 있는 모든 객체를 담은 리스트를 반환한다. 반환

되는 리스트는 포함되지 않는다.

get_referrers(obj1, obj2, ...)

객체 obj1, obj2 등을 직접 참조하는 모든 객체를 담은 리스트를 반환한다. 반환되는 리스트에는 아직 쓰레기 수집되지 않은 객체나 부분적으로만 구축된 객체가 포함될 수 있다.

get_referents(obj1, obj2, ...)

객체 obj1, obj2 등이 참조하는 객체 리스트를 반환한다. 예를 들어, obj1이 컨테이너이면 컨테이너 안에 있는 객체들의 리스트가 반환된다.

get_threshold()

현재 수집 임계값을 튜플로서 반환한다.

isenabled()

쓰레기 수집이 활성화되어 있으면 True를 반환한다.

set_debug(flags)

쓰레기 수집기의 작동 방식을 디버깅하는 데 사용되는 쓰레기 수집 디버깅 플래그를 설정한다. flags는 상수 DEBUG_STATS, DEBUG_COLLECTABLE, DEBUG_UNCOLLECTABLE, DEBUG_INSTANCES, DEBUG_OBJECTS, DEBUG_SAVEALL 와 DEBUG_LEAK를 비트 OR한 것이다. DEBUG_LEAK 플래그는 수집기에서 메모리 누수가 있는 프로그램을 디버깅하는 데 도움이 되는 정보를 출력하게 하기 때문에 아마 가장 유용한 플래그일 것이다.

set_threshold(threshold0 [, threshold1[, threshold2]])

쓰레기 수집의 수집 빈도를 설정한다. 객체는 세 세대로 나뉘어진다. 0세대는 가장 최신 객체를 담고 2세대는 가장 오래된 객체를 담는다. 쓰레기 수집 단계에서 살아남은 객체는 다음으로 오래된 세대로 이동한다. 객체가 일단 2세대에 접어들면 그곳에 계속 머무른다. threshold0은 쓰레기 수집이 0세대에서 발생하기 전에 도달해야 하는 할당 횟수와 반납 횟수의 차이를 나타낸다. threshold1은 세대 1을 탐색하기 전에 발생해야 하는 세대 0의 수집 횟수를 나타낸다. threshold2는 세대 2가 수집되기 전에 발생해야 하는 세대 1에서의 수집 횟수를 나타낸다. 기본 임계값은 (700, 10, 10)으로 설정된다. threshold0를 0으로 설정하면 쓰레기 수집이 꺼진다.

Note

- `__del__()` 메서드를 가진 객체가 관련되어 있는 순환 참조는 쓰레기 수집되지 않고 리스트 `gc.garbage`에 저장된다(수집할 수 없는 객체). 이 객체들은 객체를 파괴할 때의 어려움 때문에 수집되지 않는다.
- 함수 `get_referrers()`와 `get_referents()`는 쓰레기 수집을 지원하는 객체에만 적용된다. 이 함수들은 디버깅 목적으로 사용되어야 한다. 다른 목적으로는 사용하지 않아야 한다.

inspect

`inspect` 모듈은 속성, 문서화 문자열, 소스 코드, 스택 프레임 등 라이브 파이썬 객체에 대한 정보를 수집하는 데 사용된다.

`cleandoc(doc)`

문서화 문자열에 있는 모든 탭을 공백으로 바꾸고 함수나 메서드 안에 있는 다른 코드와 줄을 맞추기 위해서 삽입되었을 수 있는 들여쓰기를 제거하여 문서화 문자열 `doc`를 정리한다.

`currentframe()`

호출자의 스택 프레임에 해당하는 프레임 객체를 반환한다.

`formatargspec(args [, varags [, varkw [, defaults]]])`

`getargspec()`이 반환하는 값을 나타내는 깔끔하게 포맷된 문자열을 생성한다.

`formatargvalues(args [, varargs [, varkw [, locals]]])`

`getargvalues()`이 반환하는 값을 나타내는 깔끔하게 포맷된 문자열을 생성한다.

`getargspec(func)`

함수 `func`가 주어질 때, 이름 있는 튜플(named tuple)인 `ArgSpec(args, varargs, varkw, defaults)`를 반환한다. `args`는 인수 이름 리스트이고 `varargs`는 * 인수의 이름이다(있을 경우). `varkw`는 ** 인수의 이름이다(있을 경우). `defaults`는 기본 인수 값을 나타내는 튜플이며 기본 인수 값이 없는 경우 `None`이다. 기본 인수 값이 있으면 `defaults` 튜플은 `n`을 `len(defaults)`이라고 할 때 `args`에 있는 마지막 `n`개의 인수 값을 나타낸다.

getargvalues(frame)

실행 프레임 frame을 갖는 함수에 제공된 인수 값들을 반환한다. 튜플 ArgInfo(args, varargs, varkw, locals)를 반환한다. args는 인수 이름 리스트이고 varargs는 * 인수의 이름이며(있을 경우) varkw는 ** 인수의 이름이다(있을 경우). locals는 프레임의 locals 사전을 나타낸다.

getclasstree(classes [, unique])

관련된 클래스들의 리스트인 classes가 주어질 때 이 함수는 이 클래스들을 상속 구조에 따라 계층화한다. 이 계층은 중첩된 리스트들로 표현되고 리스트에서 각 항목은 리스트 바로 앞에 나온 클래스로부터 상속받은 클래스들의 목록을 나타낸다. 리스트에서 각 항목은 2 항목짜리 튜플인 (cls, bases)이다. 여기서 cls는 클래스 객체이고 bases는 기반 클래스들의 튜플이다. unique가 True이면 각 클래스는 반환되는 리스트에서 오직 한 번만 나타난다. 아니면 다중 상속이 사용될 경우 클래스는 여러 번 나타날 수 있다.

getcomments(object)

파이썬 소스 코드에서 object의 정의 바로 앞에 나오는 주석들로 구성되는 문자열을 반환한다. object가 모듈이면 모듈의 제일 위에 있는 주석들이 반환된다. 주석을 찾을 수 없으면 None이 반환된다.

getdoc(object)

object의 문서화 문자열을 반환한다. 문서화 문자열은 반환되기 전에 먼저 cleandoc() 함수를 사용해서 처리된다.

getfile(object)

object가 정의된 파일의 이름을 반환한다. 파일에 저장되는 것이 아니거나 이 정 보가 제공되지 않는 경우에는 TypeError를 반환할 수도 있다(예를 들어, 내장 함수의 경우).

getframeinfo(frame [, context])

프레임 객체 frame에 대한 정보를 담은 이름 있는 튜플인 Traceback(filename, lineno, function, code_context, index)를 반환한다. filename과 line은 소스 코드 상의 위치를 기술한다. context 매개변수는 소스 코드에서 추출할 컨텍스트의 줄 수를 나타낸다. 반환되는 튜플에서 contextlist 필드는 이 컨텍스트에 해당하는 소

스 줄들의 리스트를 담는다. index 필드는 이 리스트 안에서의 frame에 해당하는 줄에 대한 숫자 색인이다.

getinnerframes(traceback [, context])

역추적 프레임과 모든 내부 프레임에 대한 프레임 레코드 리스트를 반환한다. 각 프레임 레코드는 6개 항목짜리 튜플인 (frame, filename, line, funcname, contextlist, index)이다. filename, line, context, contextlist와 index는 getframeinfo()에서와 의미가 동일하다.

getmembers(object [, predicate])

object의 모든 멤버를 반환한다. 보통은 객체의 `__dict__` 속성에서 멤버들을 찾지만 다른 곳에 저장된 object의 속성을 반환할 수도 있다(예를 들어, `__doc__`에 있는 문서화 문서열, `__name__`에 있는 객체의 이름 등). 멤버들은 (name, value) 쌍의 리스트로 반환된다. predicate는 멤버 객체를 인수로 받고 True나 False를 반환하는 옵션인 함수이다. predicate가 True를 반환하는 멤버만 반환된다. predicate 인수에 `isfunction()`이나 `isclass()` 같은 함수가 사용될 수 있다.

getmodule(object)

object가 정의된 모듈을 반환한다(가능할 경우).

getmoduleinfo(path)

파이썬이 파일 path를 어떻게 해석하는지에 관한 정보를 반환한다. path가 파이썬 모듈이 아니면 `None`이 반환된다. 아니면 이름 있는 튜플인 `ModuleInfo(name, suffix, mode, module_type)`이 반환된다. 여기서 name은 모듈의 이름, suffix는 파일 이름의 접미사, mode는 모듈을 여는 데 사용할 파일 모드, module_type은 모듈의 종류를 기술하는 정수 코드를 나타낸다. 모듈 종류는 `imp` 모듈에 다음과 같이 정의되어 있다.

모듈 종류	설명
<code>imp.PY_SOURCE</code>	파이썬 소스 코드
<code>imp.PY_COMPILED</code>	파이썬의 컴파일된 객체 파일 (.pyc)
<code>imp.C_EXTENSION</code>	동적으로 로드할 수 있는 C 확장 기능
<code>impPKG_DIRECTORY</code>	패키지 디렉터리
<code>imp.C_BUILTIN</code>	내장 모듈
<code>imp.PY_FROZEN</code>	동결된(frozen) 모듈

getmodulename(path)

파일 path에 대해서 사용될 모듈 이름을 반환한다. path가 파이썬 모듈이 아닌 것 같으면 None을 반환한다.

getmro(cls)

클래스 cls에서 메서드를 찾는 데 사용될 메서드 분석 순서를 나타내는 클래스들의 튜플을 반환한다. 자세한 내용은 7장을 참고하도록 한다.

getouterframes(frame [, context])

frame과 모든 외부 프레임에 대한 프레임 레코드 목록을 반환한다. 이 목록은 호출 순서를 나타내며 첫 번째 항목이 frame에 대한 정보를 담는다. 각 프레임 레코드는 6개 항목짜리 튜플인 (frame, filename, line, funcname, contextlist, index)이고 의미는 getinnerframes() 함수의 것과 동일하다. context 인수는 getframeinfo()의 것과 동일하다.

getsourcefile(object)

object가 정의된 파이썬 소스 파일 이름을 반환한다.

getsourcelines(object)

object의 정의에 해당하는 (sourcelines, firstline) 튜플을 반환한다. sourcelines는 소스 코드 줄 목록을 나타내고 firstline은 소스 코드에서 첫 번째 줄의 줄 번호를 나타낸다. 소스 코드를 찾을 수 없으면 IOError를 발생시킨다.

getsource(object)

object의 소스 코드를 단일 문자열로 반환한다. 소스 코드를 찾을 수 없으면 IOError를 발생시킨다.

isabstract(object)

object가 추상 기반 클래스이면 True를 반환한다.

isbuiltin(object)

object가 내장 함수이면 True를 반환한다.

isclass(object)

object가 클래스이면 True를 반환한다.

iscode(object)

object가 코드 객체이면 True를 반환한다.

isdatadescriptor(object)

object가 데이터 기술자 객체이면 True를 반환한다. object가 `__get__()`과 `__set__()` 메서드 모두를 구현한 경우를 말한다.

isframe(object)

object가 프레임 객체이면 True를 반환한다.

isfunction(object)

object가 함수 객체이면 True를 반환한다.

isgenerator(object)

object가 생성기 객체이면 True를 반환한다.

isgeneratorfunction(object)

object가 생성기 함수이면 True를 반환한다. 생성기를 만드는 함수인지를 검사한다는 점에서 `isgenerator()`와 다르다. object가 현재 활발히 구동되고 있는 생성기인지 여부를 검사하지는 않는다.

ismethod(object)

object가 메서드이면 True를 반환한다.

ismethoddescriptor(object)

object가 메서드 기술자 객체이면 True를 반환한다. object가 메서드, 클래스나 함수가 아니고 `__get__()` 메서드는 정의하지만 `__set__()`을 정의하지 않는 경우를 말한다.

ismodule(object)

object가 모듈 객체인 경우 True를 반환한다.

isroutine(object)

object가 사용자 정의 함수 또는 메서드이거나 내장 함수 또는 메서드인 경우 True를 반환한다.

istraceback(object)

object가 역추적 객체인 경우 True를 반환한다.

stack([context])

호출자의 스택에 해당하는 프레임 레코드 목록을 반환한다. 각 프레임 레코드는 6개 항목짜리 튜플인 (frame, filename, line, funcname, contextlist, index)이고 getinnerframes()에 의해 반환되는 값과 동일한 정보를 담는다. context는 각 프레임 레코드에서 반환할 소스 컨텍스트 줄 수를 기술한다.

trace([context])

현재 프레임과 현재 예외가 발생한 프레임 사이의 스택에 대한 프레임 레코드 목록을 반환한다. 첫 번째 프레임 레코드는 호출한 곳이고 마지막 프레임 레코드는 예외가 발생한 곳이다. context는 각 프레임 레코드에서 반환할 소스 컨텍스트의 줄 수를 기술한다.

marshal

marshal 모듈은 문서화되지 않은, 파이썬에 특화된 데이터 포맷으로 파이썬 객체를 직렬화하는 데 사용된다. marshal은 pickle^o나 shelve 모듈과 비슷하지만 이들보다는 기능이 떨어지고 단순한 객체에 대한 직렬화만 지원한다. 일반적인 객체 영속화 기능을 구현하는 데 사용하지 않는 것이 좋다. 그래도 단순한 내장 타입에 대해서는 marshal 모듈을 사용하면 빠르게 데이터를 저장하고 읽을 수 있다.

dump(value, file [, version])

객체 값을 열린 파일 객체 file에 쓴다. value가 지원되지 않는 타입이면 ValueError 예외가 던져진다. version은 사용할 데이터 포맷을 기술하는 정수이다. 기본 포맷은 marshal.version에서 찾을 수 있고 현재는 2로 설정되어 있다. 버전 0은 이전 버전의 파이썬에서 사용되던 오래된 포맷이다.

dumps(value [, version])

dump()함수에 의해서 쓰여질 바이트 문자열을 반환한다. value가 지원되지 않는 타입이면 ValueError 예외가 발생한다. version은 이전에 설명한 것과 동일하다.

load(file)

열린 파일 객체 file에서 다음 값을 읽어와서 반환한다. 유효한 값이 없으면 EOFError, ValueError나 TypeError 예외가 발생한다. 입력 데이터의 포맷은 자동으로 감지된다.

loads(bytes)

바이트 문자열 bytes에서 다음 값을 읽어와서 반환한다.

Note

- 데이터는 이진 아키텍처에 의존적인 포맷으로 저장된다.
- None, 정수, 긴 정수, 실수, 복소수, 문자열, 유니코드 문자열, 튜플, 리스트, 사전, 코드 객체만 지원된다. 리스트, 튜플, 사전에는 지원되는 객체만 담을 수 있다. 리스트, 튜플, 사전에 클래스 인스턴스나 재귀 참조가 들어 있으면 안 된다.
- 내장 정수 타입이 충분한 정밀도를 갖고 있지 않은 경우에는 정수가 긴 정수로 상향 타입 변환되지 않는다. 예를 들어, 마샬링된 데이터가 64비트 정수를 담고 있지만 데이터를 32비트 머신에서 읽을 때 이러한 상황이 발생한다.
- marshal은 어려가 있을 수 있거나 악의적으로 만들어진 데이터에 대해서 안전하게 작동하도록 작성되지 않았기 때문에 의심 가는 소스로부터 온 데이터를 언마샬링하는 데 사용해서는 안 된다.
- marshal은 pickle보다 훨씬 빠르지만 덜 유연하다

pickle

pickle 모듈은 파이썬 객체를 파일에 저장하거나 네트워크를 통해 전송하거나 데이터베이스에 넣는 데 적합하도록 바이트 스트림으로 직렬화하는 데 사용된다. 이 과정은 피클링, 직렬화, 마샬링 또는 평면화 등으로 다양하게 불린다. 결과 바이트 스트림은 언피클링 과정을 통해서 다시 일련의 파이썬 객체로 변환할 수 있다.

다음 함수들은 객체를 바이트 스트림으로 바꾸는 데 사용한다.

dump(object, file [, protocol])

object의 피클링된 표현을 파일 객체 file에 쓴다. protocol은 데이터의 출력 포맷을 지정한다. 프로토콜 0(기본 값)은 초기 버전의 파이썬과 역호환되는 텍스트 기반 포맷이다. 프로토콜 1은 대부분의 이전 파이썬 버전과 호환되는 이전 프로토콜이다. 프로토콜 2는 클래스와 인스턴스를 더 효율적으로 피클링하는 새로운 프로토콜이다. 프로토콜 3은 파이썬 3에서 쓰이는 것으로서 하위 호환성이 없다. protocol이 음수이면 가장 최신 프로토콜이 사용된다. 변수 pickle.HIGHEST_PROTOCOL은 사용 가능한 가장 높은 프로토콜을 담는다. object가 피클링을 지원

하지 않으면 pickle.PicklingError 예외가 발생한다.

dumps(object [, protocol])

dump()와 동일하지만 피클링된 데이터를 담은 바이트 문자열을 반환한다.

다음 예는 이 함수들을 사용하여 객체를 파일에 저장하는 방법을 보여준다.

```
f = open('myfile', 'wb')
pickle.dump(x, f)
pickle.dump(y, f)
... 더 많은 객체를 쓴다. ...
f.close()
```

다음 함수들은 피클링된 객체를 복구하는 데 사용한다.

load(file)

파일 객체 file에서 객체의 피클링된 표현을 읽어와서 반환한다. 입력 프로토콜은 자동으로 감지되기 때문에 지정할 필요가 없다. 파일에 디코딩할 수 없는 깨진 데이터가 들어 있는 경우 pickle.UnpicklingError 예외가 발생한다. 파일의 끝을 만나면 EOFError 예외가 발생한다.

loads(bytes)

load()와 동일하지만 바이트 문자열로부터 객체의 피클링된 표현을 읽는다.

다음 예는 이 함수들을 사용하여 데이터를 읽는 방법을 보여준다.

```
f = open('myfile', 'rb')
x = pickle.load(f)
y = pickle.load(f)
... 더 많은 객체를 읽는다 ...
f.close()
```

로딩을 할 때는 프로토콜이라든지 로딩될 객체의 타입에 관한 정보를 기술할 필요가 없다. 이 정보는 데이터 포맷 자체에 들어 있다.

둘 이상의 파일 객체를 피클링하려면 앞의 예에서 나왔듯이 간단히 dump()나 load()를 반복적으로 호출하면 된다. 여러 번 호출할 때 일련의 load() 호출이 파일에 쓸 때 사용했던 dump() 호출의 순서와 일치해야 한다.

순환 참조나 공유된 참조가 있는 복잡한 데이터 구조를 다룰 때는 dump()와 load()가 이미 피클링되었거나 복구된 객체의 내부 상태를 관리해주지 않기 때문에 문제가 될 수 있다. 이런 경우 출력 파일이 과도하게 커지거나, 로드할 때 객체 사이의 관계가 제대로 복구되지 않을 수 있다. 대안으로 Pickler와 Unpickler를 사용하

는 방법이 있다.

Pickler(file [, protocol])

주어진 피클링 protocol로 데이터를 쓰는 파일 객체 file에 피클링 객체를 생성한다. Pickler의 인스턴스 p는 객체 x를 file에 쓰는 p.dump(x) 메서드를 갖는다. x가 일단 써지고 나면 x의 신원이 기억된다. 나중에 같은 객체를 쓰는 데 p.dump() 연산을 사용하면 새로운 복사본을 쓰는 대신 이전에 쓰여진 객체에 대한 참조를 쓰게 된다. p.clear_memo() 메서드는 이전에 쓰여진 객체를 추적하는 데 사용되는 내부 사전을 청소한다. 이전에 쓰여진 객체의 새로운 복사본을 쓰고자 할 때 이 메서드를 사용하면 된다(즉, 마지막 dump() 연산 이후에 값이 변경된 경우).

Unpickler(file)

파일 객체 file에서 데이터를 읽는 언피클링 객체를 생성한다. Unpickler의 인스턴스 u는 file에서 새 객체를 읽어와서 반환하는 u.load()를 갖는다. 입력 파일에 Pickler 객체에 의해서 생성된 객체 참조가 들어 있을 수 있으므로 Unpickler는 반환한 객체를 기록한다. 객체 참조를 만나면 u.load()는 이전에 로드된 객체에 대한 참조를 반환하게 된다.

pickle 모듈은 다음 목록에 나와 있는 보통의 파일 객체를 대부분 지원한다.

- None
- 숫자와 문자열
- 피클링이 가능한 객체만을 담은 튜플, 리스트, 사전
- 모듈의 최상위 수준에서 정의된 사용자 정의 클래스의 인스턴스

사용자 정의 클래스의 인스턴스가 피클링되면 피클링되는 것은 오직 인스턴스 데이터뿐이다. 관련된 클래스 정의는 저장되지 않는다. 그 대신 피클링된 데이터는 단순히 관련된 클래스와 모듈의 이름만을 담게 된다. 인스턴스가 언피클링되면 인스턴스를 다시 생성할 때 클래스 정의에 접근할 수 있도록 클래스가 정의된 모듈이 자동으로 임포트된다. 인스턴스를 재구성할 때 클래스의 __init__() 메서드는 호출되지 않는다. 대신 복구되는 인스턴스 데이터와 다른 수단을 통해서 인스턴스가 다시 생성된다.

인스턴스에 가해지는 한 가지 제약으로서 관련된 클래스 정의가 모듈의 최상위 수준에 나타나야 한다는 점이 있다(즉, 중첩된 클래스는 허용되지 않는다). 또한 인

스턴스의 클래스가 원래 `__main__`에서 정의된 것이라면 그 클래스 정의는 저장된 객체를 언피클링하기 전에 직접 다시 로드해주어야 한다(그 이유는 언피클링을 할 때 필요한 클래스 정의를 다시 `__main__`으로 자동으로 로드하는 방법을 인터프리터가 알 수 있는 방법이 없기 때문이다).

보통은 pickle을 사용자 정의 클래스에 대해 사용하기 위해서 특별히 해주어야 하는 일은 없다. 그래도 클래스에서 특수 메서드인 `__getstate__()`와 `__setstate__()` 메서드를 구현함으로써 상태를 저장하고 복구하는 데 사용될 커스터마이즈된 메서드를 정의할 수 있다. `__getstate__()` 메서드는 객체의 상태를 나타내는 피클링이 가능한 객체(문자열이나 튜플 같은)를 반환해야 한다. `__setstate__()` 메서드는 피클링된 객체를 받아서 상태를 복구한다. 이 메서드들이 정의되지 않으면 기본으로 인스턴스의 내부 `__dict__` 속성을 피클링하게 된다. 이 메서드들은 정의될 경우 얇은 복사와 깊은 복사 연산을 수행하기 위해서 copy 모듈에 의해서도 사용된다.

Note

- 파이썬 2에서 `cPickle`이라도 불리는 모듈은 pickle 모듈에 있는 함수의 C 구현 버전을 담고 있다. pickle보다 훨씬 빠르지만 Pickler와 Unpickler에서 상속을 허용하지 않는다. 파이썬 3에서도 C 구현을 담은 지원 모듈이 제공되지만 암묵적으로 사용된다(pickle이 필요에 따라 알아서 적절히 활용한다).
- pickle에 의해서 사용되는 데이터 포맷은 파이썬에 특화된 것으로서 XML 같은 어떠한 외부 표준과도 호환성을 기대해서는 안 된다.
- pickle은 더 유연하고 데이터 인코딩 방식이 문서화되어 있으며 추가적인 에러 검사를 수행하기 때문에 가능하면 언제든지 pickle 모듈을 marshal 대신 사용하도록 한다.
- 보안상의 문제가 있을 수 있으므로 의심 가는 소스에서 받은 데이터는 언피클링하지 않아야 한다.
- 확장 모듈에 정의된 타입에 대해서 pickle 모듈을 사용하는 일은 이곳에서 설명한 것보다 훨씬 더 복잡하다. 확장 타입을 구현하고자 한다면 확장 타입을 pickle과 함께 쓰일 수 있게 하기 위해서 필요한 저수준 프로토콜과 관련된 세부 사항을 설명해 놓은 온라인 문서를 참고하기 바란다. 특히, pickle에서 직렬화된 바이트 순서열을 생성하는 데 사용하는 특수 메서드인 `__reduce__()`와 `__reduce_ex__()`을 어떻게 구현하는지에 관해서 자세히 살펴보기 바란다.

sys

sys 모듈은 인터프리터의 작동과 환경에 관련된 변수와 함수를 담고 있다.

변수

다음 변수들이 정의되어 있다.

`api_version`

파이썬 인터프리터의 C API 버전을 나타내는 정수. 확장 모듈을 다룰 때 사용함.

`argv`

프로그램에 전달된 명령줄 옵션 리스트. `argv[0]`은 프로그램의 이름을 담는다.

`builtin_module_names`

파이썬 실행 파일에 내장된 모듈 이름들을 담은 튜플.

`byteorder`

기계의 고유한 바이트 순서. ‘little’은 리틀 엔디안을 ‘big’은 빅 엔디안을 나타냄.
둘 중 하나.

`copyright`

저작권을 담은 문자열.

`__displayhook__`

`displayhook()` 함수의 원래 값.

`dont_write_bytecode`

파이썬이 모듈을 임포트할 때 바이트코드(.pyc나 .pyo 파일)를 쓸 것인지 여부를
결정하는 불리언 플래그. 인터프리터에 -B 옵션을 주지 않는 한 초기값은 True. 프
로그램에서 필요한 경우 이 설정을 바꿀 수 있다.

`dllhandle`

파이썬 DLL(윈도)에 대한 정수 핸들.

`__excepthook__`

`excepthook()` 함수의 원래 값.

exec_prefix

플랫폼 종속적인 파이썬 파일이 설치될 디렉터리.

executable

인터프리터 실행 파일의 이름을 담은 문자열.

flags

파이썬 인터프리터에 제공된 명령줄 옵션들의 설정을 나타내는 객체. 다음 표는 flags의 속성과 각 속성에 대응되는 플래그를 활성화하는 명령줄 옵션을 나열한 것이다. 모두 읽기 전용 속성이다.

속성	명령줄 옵션
flags.debug	-d
flags.py3k_warning	-3
flags.division_warning	-Q
flags.division_new	-Qnew
flags.inspect	-i
flags.interactive	-i
flags.optimize	-O 또는 -OO
flags.dont_write_bytecode	-B
flags.no_site	-S
flags.ignore_environment	-E
flags.tabcheck	-t 또는 -tt
flags.verbose	-v
flags_unicode	-U

float_info

부동 소수점 수의 내부 표현에 관한 정보를 담은 객체. 다음 속성들의 값은 C 헤더 파일 float.h에서 읽어온다.

속성	설명
float_info.epsilon	1.0과 다음 가장 큰 실수 사이의 차이
float_info.dig	반올림하고 나서도 변경 없이 표현할 수 있는 10진수 숫자의 개수
float_info.mant_dig	float_info.radix로 지정된 기본수를 사용해서 표현할 수 있는 10진수 숫자의 개수
float_info.max	최대 부동 소수점 수
float_info.max_exp	float_info.radix로 지정된 기본수의 최대 지수값
float_info.max_10_exp	기본수 10의 최대 지수값
float_info.min	가장 작은 양의 부동 소수점 값
float_info.min_exp	float_info.radix로 지정된 기본수의 최소 지수값

<code>float_info.min_10_exp</code>	기본수 10의 최소 지수값
<code>float_info.radix</code>	지수에 대해 사용되는 기본수
<code>float_info.rounds</code>	반올림 방식(-1은 결정되지 않음을, 0은 영을 향해, 1은 가장 가까운 쪽으로, 2는 양의 무한대로, 3은 음의 무한대로)

hexversion

`sys.version_info`에 들어 있는 버전 정보를 16진수로 인코딩한 정수. 이 정수 값은 새로운 버전의 인터프리터가 나올 때마다 항상 증가한다.

last_type, last_value, last_traceback

이 변수들은 처리되지 않은 예외가 있어서 인터프리터에서 에러 메시지를 출력할 때 생성된다. `last_type`은 최종 예외 타입을, `last_value`는 최종 예외 값을, `last_traceback`은 스택 추적 내용을 나타낸다. 이 변수들을 사용할 때는 스레드 안전성이 보장되지 않는다. 스레드 안전성을 보장하려면 `sys.exec_info()`를 대신 사용해야 한다.

maxint

정수 타입에 의해서 지원되는 가장 큰 정수(파이썬 2에만 있다).

maxsize

시스템에서 C의 `size_t` 데이터 타입에 의해서 지원되는 가장 큰 정수 값. 이 값은 문자열, 리스트, 사전 및 기타 내장 타입의 최대 길이를 결정한다.

maxunicode

표현할 수 있는 가장 큰 유니코드 코드 포인트를 나타내는 정수. 16 비트 UCS-2 인코딩에 대해서 기본 값은 65535이다. 파이썬이 UCS-4를 사용하도록 설정되면 더 큰 값을 갖게 된다.

modules

모듈 이름을 모듈 객체로 매핑하는 사전.

path

모듈을 찾기 위한 검색 경로를 기술하는 문자열 리스트. 첫 번째 항목은 항상 파이썬을 실행하는 데 사용된 스크립트가 있는 디렉터리로 설정된다(있을 경우). 8장을 참고하라.

platform

‘linux-i386’ 같은 플랫폼 식별 문자열.

prefix

플랫폼에 특화된 파이썬 파일을 설치할 디렉터리.

ps1, ps2

인터프리터의 주 프롬프트와 이차적인 프롬프트를 나타내는 텍스트를 담은 문자열. 초기에 ps1은 ‘>>>’으로 설정되고 ps2는 ‘...’로 설정된다. 이 두 값에 할당된 객체에 대해서 str() 메서드를 호출한 결과 값이 프롬프트 텍스트를 출력하는 데 사용된다.

py3kwarning

인터프리터가 -3 옵션으로 실행되었을 때 파이썬 2에서 True로 설정되는 플래그.

stdin, stdout, stderr

각각 표준 입력, 표준 출력, 표준 에러에 대응되는 파일 객체. stdin은 raw_input()과 input() 함수에 의해서 사용된다. stdout은 print와 raw_input() 및 input()의 프롬프트 출력에 사용된다. stderr는 인터프리터의 프롬프트와 에러 메시지 출력에 사용된다. 각 변수에는 단일 문자열 인수를 받는 write() 메서드를 갖고 있는 어떤 객체든 할당할 수 있다.

__stdin__, __stdout__, __stderr__

각각 인터프리터가 시작될 때 stdin, stdout과 stderr의 값을 담는 파일 객체

tracebacklimit

처리되지 않은 예외가 발생했을 때 출력될 역추적 정보의 최대 수준 개수. 기본 값은 1000. 0이 주어지면 역추적 정보의 출력을 막고 예외 타입과 값만을 출력한다.

version

버전 문자열.

version_info

버전 정보를 튜플 (major, minor, micro, releaselevel, serial)로 표현. ‘alpha’, ‘beta’, ‘candidate’ 또는 ‘final’ 중 하나의 값을 가지는 releaselevel을 제외하고는 모두 정수로 표현된다.

warnoptions

-W 명령줄 옵션으로 인터프리터에 제공된 경고 옵션 목록.

winver

윈도에서 레지스트리 키를 만드는 데 사용되는 버전 번호.

함수

sys 모듈에는 다음 함수들이 존재한다.

_clear_type_cache()

내부 타입 캐시를 비운다. 메서드 검색 속도를 높이기 위해서 최근 사용된 메서드를 모은 작은 1024 항목 캐시가 인터프리터에서 관리된다. 이 캐시는 반복된 메서드 검색의 속도를 향상시킨다. 특히 깊은 상속 계층이 있는 코드에서 더욱 그렇다. 보통 이 캐시를 비워야 할 필요는 없지만 아주 까다로운 메모리 참조 횟수 세기 문제를 해결하고자 할 때 그렇게 해야 하는 경우가 있다. 예를 들어, 파괴되기를 기대하는 객체에 대한 참조를 캐시에 있는 메서드가 갖고 있는 경우를 생각해볼 수 있다.

__current_frames()

스레드 식별자들을 호출 시점 실행 스레드의 가장 상위 스택 프레임으로 매핑하는 사전을 반환한다. 이 정보는 스레드 디버깅과 관련된 도구를 작성할 때 유용하다(즉, 교착 상태deadlock를 찾을 때). 이 함수에 의해서 반환되는 값들은 호출 시점에 인터프리터의 스냅샷을 나타낼 뿐이라는 것에 주의한다. 여러분이 반환된 데이터를 들여다볼 때쯤 되면 스레드가 다른 지점에서 실행되고 있을 수 있다.

displayhook([value])

이 함수는 인터프리터가 대화식 모드에서 실행 중일 때 표현식의 결과를 출력하려고 호출되는 함수이다. 기본으로 repr(value)의 값이 표준 출력으로 출력되고 value가 변수 __builtins__에 저장된다. 원할 경우 다르게 작동하도록 displayhook 함수를 재정의할 수 있다.

excepthook(type, value, traceback)

이 함수는 잡히지 않은 예외가 발생할 때 호출된다. type은 예외 클래스를, value는 raise문에서 제공된 값을, traceback은 역추적 객체를 나타낸다. 기본으로 예외와 역추적 정보를 표준 에러에 출력한다. 잡히지 않은 예외를 다르게 처리하기 위해서 이 함수를 재정의할 수 있다(디버거나 CGI 스크립트 같은 특수한 용용 프로그램에서 유용할 수 있다).

exec_clear()

발생된 최종 예외와 관련된 모든 정보를 지운다. 호출 스레드에 국한된 정보만 지운다.

exec_info()

현재 처리 중인 예외에 관한 정보를 담은 튜플 (type, value, traceback)을 반환한다. type은 예외 타입을, value는 raise문에 전달된 예외 매개변수를, traceback은 예외가 발생한 시점의 호출 스택을 담은 역추적 객체이다. 아무런 예외도 처리되고 있지 않으면 None을 반환한다.

exit([n])

SystemExit 예외를 발생시켜서 파이썬을 종료한다. n은 상태 코드를 나타내는 정수이다. 0(기본 값)은 이상이 없다고 간주되고 0이 아닌 값은 이상이 있다고 간주된다. n에 정수가 아닌 값이 주어지면 그 값이 sys.stderr로 출력되고 종료 코드 1이 사용된다.

getcheckinterval()

시그널이나 스레드 컨텍스트 전환 또는 기타 주기적인 이벤트를 인터프리터가 얼마나 자주 검사할지를 지정하는 검사 주기 값을 반환한다.

getdefaultencoding()

유니코드 변환에 사용할 기본 문자열 인코딩을 얻어온다. ‘ascii’나 ‘utf-8’ 같은 값을 반환한다. 기본 인코딩은 site 모듈에 의해서 설정된다.

getdlopenFlags()

유닉스에서 확장 모듈을 로딩할 때 C 함수 dlopen() 함수에 제공되는 플래그 매개변수를 반환한다. dl 모듈을 참고하도록 한다.

getfilesystemencoding()

유니코드 파일 이름을 내부 운영체제에 의해서 사용되는 파일 이름으로 매핑하는 데 사용되는 문자 인코딩을 반환한다. 윈도에서는 ‘mbcs’를, 맥킨토시 OS X에서는 ‘utf-8’을 반환한다. 유닉스에서는 인코딩이 로케일 설정에 따라 달라질 수 있으며 로케일 CODESET 매개변수의 값을 반환한다. 시스템 기본 인코딩이 사용될 경우 None을 반환할 수도 있다.

_getframe([depth])

호출 스택에서 프레임 객체를 반환한다. depth가 생략되거나 영이면 최상위 프레임이 반환된다. 아니면 현재 프레임에서 지정된 호출 횟수만큼 아래에 있는 프레임이 반환된다. 예를 들어, `_getframe(1)`은 호출자의 프레임을 반환한다. depth가 유효하지 않은 값이면 `ValueError` 예외가 발생한다.

getprofile()

`setprofile()` 함수에 의해서 설정된 프로파일 함수를 반환한다.

getrecursionlimit()

함수의 재귀 한도를 반환한다.

getrefcount(object)

`object`의 참조 횟수를 반환한다.

getsizeof(object [, default])

`object`의 크기를 바이트로 반환한다. `object`의 특수한 메서드 `__sizeof__()` 메서드를 호출해서 계산한다. 이 메서드가 정의되지 않으면 `default` 인수로 기본 값을 지정하지 않은 이상 `TypeError` 예외가 발생한다. `__sizeof__()`를 마음대로 정의할 수 있기 때문에 이 함수의 반환 값이 정확한 메모리 사용량을 나타낼 것이라는 것이 보장되지 않는다. 리스트나 문자열 같은 내장 타입에 대해서는 올바른 값이 반환된다.

gettrace()

`settrace()` 함수에 의해서 설정된 추적 함수를 반환한다.

getwindowsversion()

사용 중인 윈도 버전을 설명하는 튜플 (`major`, `minor`, `build`, `platform`, `text`)를 반환한다. `major`는 주 버전 번호이다. 예를 들어, 4는 윈도 NT 4.0을 가리키고 5는 윈도 2000과 윈도 XP를 가리킨다. `minor`는 부 버전 번호이다. 예를 들어, 0은 윈도 2000을 1은 윈도 XP를 나타낸다. `build`는 윈도 빌드 번호를 나타낸다. `platform`은 플랫폼을 나타내고 0(윈도 3.1에서 Win32), 1(윈도 95, 98 또는 Me), 2(윈도 NT, 2000, XP) 또는 3(윈도 CE) 중 하나인 정수 값이다. `text`는 “Service Pack 3” 같은 추가 정보를 담은 문자열이다.

setcheckinterval(n)

시그널이나 스레드 컨텍스트 전환 같은 주기적인 이벤트를 검사하기 전까지 인

터프리터에서 실행해야 하는 파이썬 가상 머신 명령의 수를 설정한다. 기본 값은 10이다.

setdefaultencoding(enc)

기본 인코딩을 설정한다. enc는 ‘ascii’나 ‘utf-8’ 같은 문자열이다. 이 함수는 site 모듈 안에서만 정의된다. 사용자가 정의할 수 있는 sitecustomize 모듈에서도 호출될 수 있다.

setdlopenflags(flags)

유닉스에서 확장 모듈을 로딩하는 데 사용되는 C dlopen() 함수에 전달할 플래그를 설정한다. 라이브러리와 기타 확장 모듈 사이에 기호를 해석하는 방식에 영향을 준다. flags는 dl 모듈에서 찾을 수 있는 값들을 비트 OR한 것이다(19장 참고). 예를 들어서, sys.setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL).

setprofile(pfunc)

소스 코드 프로파일러를 구현하는 데 사용될 수 있는 시스템 프로파일 함수를 설정한다.

setrecursionlimit(n)

함수의 재귀 한도를 변경한다. 기본 값은 1000이다. 운영체제에서 스택 크기에 엄격한 한도를 설정할 수 있기 때문에 이 값을 너무 높게 설정하면 파이썬 인터프리터 프로세스가 Segmentation Fault나 Access Violation 에러와 함께 멈춰버릴 수 있다.

setattrace(tfunc)

디버거를 구현하는 데 사용할 수 있는 시스템 추적 함수를 설정한다. 파이썬 디버거에 관해서는 11장을 참고하도록 한다.

traceback

traceback 모듈은 예외가 발생되고 나서 프로그램의 스택 추적 정보를 모아서 출력하는 데 사용된다. 이 모듈에 있는 함수들은 sys.exc_info() 함수에 의해서 세 번째 항목으로서 반환되는 것 같은 역추적 객체에 대해서 작동한다. 이 모듈은 주로 에러를 표준적이지 않은 방식으로 보고하고자 할 때 사용된다. 예를 들어, 네트워크 서버의 내부 깊숙한 곳에서 실행되는 파이썬 프로그램에서 역추적 정보를 로그 파일에 기록하거나, 디버거에서 역추적 정보를 출력하는 등이다.

일로 보내려고 할 때 쓸 수 있다.

print_tb([traceback [, limit [, file]]])

traceback에서 limit만큼의 스택 추적 항목을 파일 file에 출력한다. limit를 생략하면 모든 항목이 출력된다. file을 생략하면 출력이 sys.stderr로 보내진다.

print_exception(type, value, traceback [, limit [, file]]))

예외 정보와 스택 추적 정보를 file에 출력한다. type은 예외 타입, value는 예외 값 을 나타낸다. limit와 file은 print_tb()의 것과 동일하다.

print_exc([limit [, file]])

sys.exc_info() 함수에 의해서 반환되는 정보에 print_exception을 적용한 것과 같다.

format_exc([limit [, file]])

print_exc()에 의해서 출력되는 것과 동일한 정보를 담은 문자열을 반환한다.

print_last([limit [, file]])

print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file)과 동일하다.

print_stack([frame [, limit [, file]]])

이 함수가 호출된 지점부터 스택 추적 정보를 출력한다. frame은 시작할 스택 프레임을 추가로 지정한다. limit와 file은 print_tb()의 것과 동일하다.

extract_tb(traceback [, limit])

print_tb()에 의해서 사용되는 스택 추적 정보를 추출한다. 반환되는 값은 스택 추적 정보에 보통 나타나는 것과 동일한 정보를 담은 (filename, line, funcname, text) 형태의 튜플 리스트이다. limit는 반환할 항목 개수를 나타낸다.

extract_stack([frame [, limit]])

print_stack()에 의해서 출력되는 것과 동일한 스택 추적 정보를 추출하지만 스택 프레임 frame에서 정보를 얻어온다. frame을 생략하면 호출자의 현재 스택 프레임 이 사용되고 limit는 반환할 항목 수를 나타낸다.

format_list(list)

출력을 위해서 스택 추적 정보의 포맷을 지정한다. list는 extract_tb()나 extract_

`stack()`에 의해서 반환되는 튜플 리스트이다.

format_exception_only(type, value)

출력을 위해서 예외 정보의 포맷을 지정한다.

format_exception(type, value, traceback [, limit])

출력을 위해서 예외와 스택 추적 정보의 포맷을 지정한다.

format_tb(traceback [, limit])

`format_list(extract_tb(traceback, limit))`과 동일하다

format_stack([frame [,limit]])

`format_list(extract_stack(frame, limit))`과 동일하다.

tb_lineno(traceback)

역추적 객체에 설정된 줄 번호를 반환한다.

types

types 모듈은 함수, 모듈, 생성기, 스택 프레임 및 기타 프로그램 구성 요소에 대응하는 내장 타입의 이름을 정의한다. 이 모듈은 보통 내장 함수 `isinstance()`나 기타 타입과 관련된 연산과 함께 사용된다.

변수	설명
<code>BuiltinFunctionType</code>	내장 함수 타입
<code>CodeType</code>	코드 객체 타입
<code>FrameType</code>	실행 프레임 객체 타입
<code>FunctionType</code>	사용자 정의 함수와 람다(lambda) 타입
<code>GeneratorType</code>	생성기, 반복자 객체
<code>GetSetDescriptorType</code>	getset 기술자 객체 타입
<code>LambdaType</code>	<code>FunctionType</code> 의 다른 이름
<code>MemberDescriptorType</code>	멤버 기술자 객체 타입
<code>MethodType</code>	사용자 정의 클래스 메서드 타입
<code>ModuleType</code>	모듈 타입
<code>TracebackType</code>	역추적 객체 타입

앞에 나온 대부분의 타입 객체는 생성자로 사용되어 해당 타입의 객체를 생성하는 데 사용될 수 있다. 여기서는 함수, 모듈, 코드 객체, 메서드를 생성하는 데 사용되는 매개변수만을 나열한다. 생성되는 객체의 속성과 함수에 전달해야 하는 인수에 관해서는 3장에서 자세하게 설명했다.

FunctionType(code, globals [, name [, defarags [, closure]]])

새로운 함수 객체를 생성한다.

CodeType(argcount, nlocals, stacksize, flags, codestring, constants, names, varnames, filename, name, firstlineno, lnotab [, freevars [, cellvars]])

새로운 코드 객체를 생성한다.

MethodType(function, instance, class)

새로운 뮤인 인스턴스 메서드를 생성한다.

ModuleType(name [, doc])

새로운 모듈 객체를 생성한다.

Note

- types 모듈은 정수, 리스트, 사전 같은 내장 객체의 타입을 참조하는 데 사용해서는 안 된다. 파이썬 2에서 types은 IntType이나 DictType 같은 이름도 담고 있다. 이 이름들은 단순히 내장 타입 이름인 int나 dict의 별칭일 뿐이다. 파이썬 3에서 이 모듈에는 앞에서 나열한 이름만 들어 있기 때문에 최신 코드에서는 내장 타입 이름을 바로 사용해야 한다.

warnings

warnings 모듈은 경고 메시지를 출력하고 걸러내는 기능을 제공한다. 예외와는 달리, 경고는 예외를 생성하거나 실행을 멈추게 하지 않고 잠재적인 문제를 사용자에게 알리는 것이 목적이다. warnings 모듈의 주 사용처 중 하나는 미래의 파이썬 버전에서 지원되지 않을 수 있는 사용이 권장되지 않는 언어 기능에 관해서 사용자에게 알리는 데 있다. 다음 예를 보자.

```
>>> import regex
__main__:1: DeprecationWarning: the regex module is deprecated; use
the re module
>>>
```

예외와 마찬가지로 경고들도 클래스 계층으로 구성되어 분류된다. 다음 목록은 현재 지원되는 경고의 종류를 나열한 것이다.

경고 객체

Warning

UserWarning

설명

모든 경고 타입의 기본 클래스

사용자 정의 경고

<code>DeprecationWarning</code>	사용이 권장되지 않는 기능의 사용에 대한 경고
<code>SyntaxWarning</code>	잠재적인 문법 문제
<code>RuntimeWarning</code>	잠재적인 런타임 문제
<code>FutureWarning</code>	특정 기능의 작동 방식이 미래 릴리즈에서 변경될 수 있음

각 클래스는 `__builtin__` 모듈에도 있고 exceptions 모듈에도 있다. 또한 모든 경고는 `Exception`의 인스턴스이기 때문에 경고를 에러로 변경하는 일이 쉽다.

경고는 `warn()` 함수를 사용해서 알릴 수 있다. 다음 예를 보자.

```
warnings.warn("feature X is deprecated.")
warnings.warn("feature Y might be broken.", RuntimeWarning)
```

원할 경우 경고를 필터링할 수 있다. 필터링은 경고 메시지의 출력 방식을 변경하거나 경고를 무시하거나 경고를 예외로 변경하는 데 사용할 수 있다. `filterwarnings()` 함수는 특정 타입의 경고에 대한 필터를 추가하는 데 사용된다. 다음 예를 보자.

```
warnings.filterwarnings(action="ignore",
                        message=".*regex.*",
                        category=DeprecationWarning)
import regex # 경고 메시지가 사라진다.
```

제한적이기는 하지만 인터프리터에 `-W` 옵션을 주어서 필터를 지정할 수도 있다.

```
% python -W ignore:the\ regex:DeprecationWarning
```

`warnings` 모듈에는 다음 함수들이 정의되어 있다.

```
warn(message[, category[, stacklevel]])
```

경고를 보낸다. `message`는 경고 메시지를 담은 문자열이고 `category`는 경고 클래스(`DeprecationWarning` 같은)이며, `stacklevel`은 경고 메시지가 생기는 스택 프레임을 지정하는 정수이다. 기본으로 `category`는 `UserWarning`이고 `stacklevel`은 1이다.

```
warn_explicit(message, category, filename, lineno[, module[, registry]])
```

`warn()` 함수의 저수준 버전이다. `message`와 `category`는 `warn()`과 의미가 같다. `filename`과 `lineno`, `module`은 경고의 위치를 명시적으로 지정한다. `registry`는 현재 작동 중인 모든 필터를 나타내는 객체이다. `registry`를 생략하면 경고 메시지는 억제되지 않는다.

```
showwarning(message, category, filename, lineno[, file])
```

경고를 파일에 쓴다. `file`을 생략하면 `sys.stderr`에 써진다.

formatwarning(message, category, filename, lineno)

경고를 알릴 때 출력되는 메시지에 포맷을 적용한 문자열을 생성한다.

filterwarnings(action[, message[, category[, module[, lineno[, append]]]]])

경고 필터 목록에 항목을 하나 추가한다. action은 ‘error’, ‘ignore’, ‘always’, ‘default’, ‘once’, ‘module’ 중 하나가 될 수 있다. 다음은 각 action을 설명한다.

action 값	설명
‘error’	경고를 예외로 변환한다.
‘ignore’	경고를 무시한다.
‘always’	항상 경고 메시지를 출력한다.
‘default’	경고가 발생한 각 위치에 대해서 경고를 한 번씩 출력한다.
‘module’	경고가 발생한 각 모듈에서 경고를 한 번씩 출력한다.
‘once’	어디서 발생되었는지에 상관없이 경고를 한 번씩 출력한다.

message는 경고 메시지를 매칭하는 데 사용할 정규 표현식 문자열이다. category는 DeprecationWarning 같은 경고 클래스이다. module은 모듈 이름에 매칭할 정규 표현식 문자열이다. lineno는 특정 줄 번호이거나 모든 줄에 매칭을 할 경우 0이다. append는 필터를 필터 목록의 가장 끝에 추가할 것인지를 지정한다(끝에 있는 필터가 가장 나중에 검사된다). 기본으로 새 필터는 필터 목록의 가장 앞에 추가된다. 인수를 생략하면 기본 값으로 모든 경고에 매치되는 값이 사용된다.

resetwarnings()

모든 경고 필터를 초기화한다. filterwarnings() 호출과 -W로 지정한 모든 옵션

Note

- 현재 사용 중인 필터 목록은 warnings.filters에 들어 있다.
- 경고가 예외로 변환되면 경고가 예외 타입이 된다. 예를 들어, DeprecationWarning에 대한 에러는 DeprecationWarning 예외를 발생시킨다.
- W 옵션은 명령줄에서 경고 필터를 추가하는 데 사용된다. 이 옵션의 가장 일반적인 형식은 다음과 같다.

-Waction : message : category : module:lineno

각 부분은 filterwarning() 함수에서 의미와 같다. 여기서 다른 점은 message와 module이 정규 표현식 대신에 필터링될 경고 메시지나 모듈 이름의 앞부분에 대한 부분 문자열을 지정한다는 점이다.

을 버린다.

weakref

weakref 모듈은 약한 참조에 대한 지원을 제공한다. 일반적으로 객체에 대한 참조는 참조 횟수를 증가시킨다. 이로 인해 객체는 자신에 대한 참조가 사라지기 전까지 살아 있게 된다. 반면에 약한 참조는 참조 횟수를 증가시키지 않고 객체를 참조할 수 있는 방법을 제공한다. 이 모듈은 객체를 일반적이지 않은 방식으로 관리해야 하는 특정 응용에서 유용하게 쓰인다. 예를 들어, 객체지향 프로그램에서 관찰자(Observer) 패턴 같은 객체 사이의 관계를 구현해야 할 때 약한 참조를 사용하면 순환 참조를 방지할 수 있다. 여기에 관한 예는 7장 ‘객체 메모리 관리’ 절에서 다루었다.

약한 참조는 다음과 같이 weakref.ref() 함수를 사용해서 생성한다.

```
>>> class A: pass
>>> a = A()
>>> ar = weakref.ref(a) # a에 대한 약한 참조를 생성한다.
>>> print ar
<weakref at 0x135a24; to 'instance' at 0x12ce0c>
```

일단 약한 참조가 만들어지면 이것을 인수 없이 함수로서 호출해서 원래 객체를 얻을 수 있다. 원래 객체가 아직 존재하면 그것이 반환된다. 아니면 원래 객체가 더 이상 존재하지 않는다는 것을 알리기 위해서 None이 반환된다. 다음 예를 보자.

```
>>> print ar() # 원래 객체를 출력한다.
<__main__.A instance at 12ce0c>
>>> del a # 원래 객체를 삭제한다.
>>> print ar() # a가 없어졌으므로 None을 반환한다.
None
>>>
```

weakref 모듈에는 다음 함수들이 정의되어 있다.

ref(object [, callback])

object에 대한 약한 참조를 생성한다. callback은 생략 가능한 함수로서 object가 막 파괴되려고 할 때 호출된다. 이 함수를 제공할 경우 이 함수는 약한 참조 객체에 해당하는 단일 인수를 받아야 한다. 둘 이상의 약한 참조가 같은 객체를 가리킬 수 있다. 이 경우 역호출(callback) 함수들은 가장 최근의 참조부터 가장 오래된 참조의 순서대로 호출된다. 반환되는 약한 참조를 인수 없이 함수처럼 호출해서 약한

참조에서 object를 얻을 수 있다. 원래 객체가 더 이상 존재하지 않으면 None이 반환된다. 실제로 ref()는 타입 검사를 하거나 하위 클래스를 생성하는 데 사용할 수 있는 타입 ReferenceType을 나타낸다.

proxy(object [, callback])

object에 대한 약한 참조를 사용하여 대리자를 생성한다. 반환되는 대리자 객체는 원래 객체의 속성과 메서드에 대한 접근을 제공하는 원래 객체를 감싸는 래퍼이다. 원래 객체가 존재하는 한 대리자 객체를 조작할 경우 내부 객체의 작동 방식을 투명하게 흉내 내게 된다. 반면에 원래 객체가 파괴되면 대리자에 가한 연산은 원래 객체가 더 이상 존재하지 않는다는 것을 알리기 위해서 weakref.ReferenceError 예외를 발생시킨다. callback은 ref() 함수의 역호출 함수와 동일한 의미를 가진다. 대리자 객체의 타입은 원래 객체가 호출 가능한지 여부에 따라서 ProxyType이거나 CallableProxyType이다.

getweakrefcount(object)

object를 가리키는 약한 참조와 대리자 수를 반환한다.

getweakrefs(object)

object를 가리키는 모든 약한 참조와 대리자 객체의 리스트를 반환한다.

WeakKeyDictionary([dict])

키가 약하게 참조되는 사전을 생성한다. 어떤 키에 대해서 강한 참조가 더 이상 없으면 해당 항목이 사전에서 알아서 제거된다. dict가 주어지면 dict에 있는 항목들이 반환되는 WeakKeyDictionary 객체에 추가된다. 특정한 종류의 객체만이 약하게 참조될 수 있기 때문에 키 값에는 많은 제약이 가해진다. 특히, 내장 문자열은 약한 키로서 사용할 수 없다. __hash__() 메서드를 정의한 사용자 정의 클래스의 인스턴스는 키로서 사용할 수 있다. WeakKeyDictionary의 인스턴스에는 약한 키 참조를 반환하는 두 메서드, iterkeyrefs()와 keyrefs()가 있다.

WeakValueDictionary([dict])

값이 약하게 참조되는 사전을 생성한다. 어떤 값에 대해서 강한 참조가 더 이상 없으면 해당 항목이 사전에서 알아서 제거된다. dict가 주어지면 dict에 있는 항목들이 반환되는 WeakValueDictionary 객체에 추가된다. WeakValueDictionary의 인스턴스에는 약한 값 참조를 반환하는 두 메서드, itervaluerefs()와 valuerefs()가 있다.

ProxyTypes

어떤 객체가 proxy() 함수에 의해서 생성되는 두 종류의 대리자 객체 중 하나인지 를 검사하는 데 사용할 수 있는 튜플 (ProxyType, CallableProxyType)이다. 예를 들어 isinstance(object, ProxyTypes).

예

약한 참조의 응용으로 최근에 계산된 결과에 대한 캐시를 생성하는 것을 생각해볼 수 있다. 예를 들어, 어떤 함수에서 결과를 계산하는 데 시간이 오래 걸릴 경우 결과를 캐싱해 두었다가 다른 곳에서 재사용하는 것이다. 다음 예를 보자.

```
_resultcache = { }
def foocache(x):
    if x in _resultcache:
        r = _resultcache[x]()
        if r is not None: return r
    r = foo(x)
    _resultcache[x] = weakref.ref(r)
    return r
```

Note

- 클래스 인스턴스, 함수, 메서드, 집합, frozenset, 파일, 생성기, 타입 객체, 그리고 라이브러리 모듈에 정의된 특정 객체 타입(소켓, 배열, 정규 표현식 패턴 등)만 약한 참조를 지원한다. 내장 함수와 리스트, 사전, 문자열, 숫자 같은 대부분의 내장 타입은 사용할 수 없다.
- WeakKeyDictionary나 WeakValueDictionary에 대해서 반복을 사용할 생각이라면 사전의 크기를 변경하지 않도록 주의해야 한다. 사전의 크기가 변할 경우 어떤 항목이 알 수 없는 이유로 사라진다거나 하는 이상한 일이 벌어질 수 있다.
- ref()나 proxy()로 등록한 역호출 함수가 실행되는 중에 예외가 발생하면 해당 예외는 표준 예외로 출력된 후 무시된다.
- 약한 참조는 원래 객체가 해싱 가능한 한 해싱이 가능하다. 게다가 약한 참조는 원래 객체가 존재하는 동안 원래 객체의 해시 값이 계산되었다면 원래 객체가 삭제된 후에도 자신의 해시 값을 유지한다.
- 약한 참조는 서로 동일한지 검사할 수 있지만 순서를 정할 수는 없다. 원래 객체가 아직 살아 있는 경우에 두 약한 참조의 내부 객체가 동일한 값을 갖는 한, 두 약한 참조는 동일하다. 또는 두 참조가 같은 참조이면 동일하다.

14장

Python Essential Reference

수학

이 장에서는 다양한 종류의 수리 연산을 위한 모듈들을 설명한다. 범용 십진수 부동 소수점 수를 지원하는 decimal 모듈도 설명한다.

decimal

파이썬의 float 데이터 타입은 배정밀도 이진 부동 소수점 인코딩을 사용하여 표현된다(보통 IEEE 754 표준을 따른다). 이 인코딩의 특성 때문에 0.1 같은 십진수 값을 정확하게 표현할 수 없다. 0.1에 가장 가까운 값은 0.1000000000000001이다. 부동 소수점 수와 관련된 계산에서 이러한 부정확성이 영향을 주기 때문에 예상 밖의 결과를 얻기도 한다(예를 들어, $3 * 0.1 == 0.3$ 은 False로 평가된다).

decimal 모듈은 십진수를 정확하게 표현할 수 있게 하는 IBM 범용 십진수 계산 표준(IBM General Decimal Arithmetic Standard)을 구현한다. 이 모듈은 수리 정밀도, 유효 숫자, 반올림의 작동 방식을 원하는 대로 제어할 수 있는 기능을 제공한다. 이 기능은 특정한 십진수 작동 방식을 기대하는 외부 시스템과 상호작용이 필요할 때 유용하게 쓰인다. 한 예로서 비즈니스 응용 프로그램과 상호작용하는 예를 생각해볼 수 있다.

decimal 모듈은 두 가지 기본 데이터 타입을 정의한다. Decimal 타입은 십진수를 나타내고 Context 타입은 정밀도나 반올림 에러 처리 같은 계산에 영향을 주는 다양한 매개변수를 나타낸다. 다음에 나오는 간단한 예는 이 모듈의 기본적인 사용 방식을 보여준다.

```

import decimal
x = decimal.Decimal('3.4')    # 십진수들을 생성한다.
y = decimal.Decimal('4.5')

# 기본 컨텍스트를 사용해서 몇 가지 수리 계산을 수행한다.
a = x * y      # a = decimal.Decimal('15.30')
b = x / y      # b = decimal.Decimal('0.75555555555555555555555556')

# 정밀도를 수정하고 계산을 수행한다.
decimal.getcontext().prec = 3
c = x * y      # c = decimal.Decimal('15.3')
d = x / y      # d = decimal.Decimal('0.756')

# 단일 문장 블록에 대해서만 정밀도를 변경한다.
with decimal.localcontext(decimal.Context(prec=10)):
    e = x * y    # e = decimal.Decimal('15.30')
    f = x / y    # f = decimal.Decimal('0.7555555556')

```

Decimal 객체

십진수는 다음 클래스로 표현한다.

Decimal([value [, context]])

value는 숫자 값으로서 정수이거나 ‘4.5’ 같은 십진수 값을 담은 문자열이거나 튜플(sign, digits, exponent)일 수 있다. 튜플이 사용되면 sign 값 0은 양수를, 1은 음수를 나타낸다. digits는 정수로 표현되는 아라비아 숫자(0에서 9 사이 값)들의 튜플이다. exponent는 정수 지수를 나타낸다. 특수 문자열인 ‘Infinity’, ‘-Infinity’, ‘NaN’ 그리고 ‘sNaN’은 각각 양의 무한대, 음의 무한대, 숫자 아님(NaN: Not a Number)을 나타낸다. ‘sNaN’은 NaN의 한 종류로서 이후의 계산에서 사용될 경우 예외를 발생시킨다. 보통 float 객체는 그 값이 정확하지 않을 수 있기 때문에 초기 값으로 사용될 수 없다(애초에 decimal을 사용하는 목적에 위배된다). context 매개 변수는 나중에 설명할 Context 객체를 나타낸다. 제공될 경우 context는 초기 값이 유효한 숫자가 아닌 경우 무슨 일이 벌어질지를 결정한다. 예외가 발생될 수도 있고 값 NaN을 가진 십진수가 반환될 수도 있다.

다음 예는 십진수를 생성하는 다양한 방법을 보여준다.

```

a = decimal.Decimal(42)                      # Decimal("42")를 생성한다.
b = decimal.Decimal("37.45")                  # Decimal("37.45")를 생성한다.
c = decimal.Decimal((1,(2,3,4,5),-2)) # Decimal("-23.45")를 생성한다.
d = decimal.Decimal("Infinity")
e = decimal.Decimal("NaN")

```

Decimal 객체는 변경이 불가능하고 내장 타입인 int와 float이 가지는 일반적인 수리적 속성을 모두 갖는다. 또한 사전의 키로 사용할 수 있고 집합에 넣을 수 있고 정렬할 수 있다. 보통 일반적인 파이썬의 표준 수학 연산자를 사용해서 Decimal 객체를 다룬다. 또한 다음에 나오는 메서드를 사용하면 자주 사용되는 몇몇 수리 연산을 수행할 수 있다. 모든 연산이 정밀도, 반올림, 그리고 기타 계산 과정의 여러 측면을 제어하는 생략 가능한 context 매개변수를 받는다. 이 매개변수가 생략되면 현재 컨텍스트가 사용된다.

메서드	설명
<code>x.exp([context])</code>	자연 지수 e^{x}
<code>x.fma(y, z [, context])</code>	$x*y + z$. $x*y$ 는 반올림하지 않는다.
<code>x.ln([context])</code>	x 의 자연 로그 (기본수 e)
<code>x.log10([context])</code>	x 의 기본수 10 로그
<code>x.sqrt([context])</code>	x 의 제곱근

Context 객체

반올림이나 정밀도 같은 십진수 숫자의 다양한 속성을 제어하는 데 Context 객체가 사용된다.

```
Context(prec=None, rounding=None, traps=None, flags=None, Emin=None, Emax=None, capitals=1)
```

새로운 십진수 컨텍스트를 생성한다. 매개변수는 여기 나온 이름의 키워드 인수로 전달해야 한다. prec는 수리 연산에 적용될 정밀도 숫자 개수를 설정하는 정수이고, rounding은 반올림 방식을 결정하며, traps는 계산 과정에서 어떤 사건이 발생했을 때(영으로 나누기 같은) 파이썬 예외를 생성하는 신호 목록을 나타낸다. flags는 컨텍스트의 초기 상태를 가리키는 신호 목록을 나타낸다(오버플로우 같은). 보통 flags는 설정하지 않는다. Emin과 Emax는 각각 지수의 최소 범위와 최대 범위를 나타내는 정수다. capitals는 불리언 플래그로서 지수에 'E'를 사용할지 'e'를 사용할지를 지정한다. 기본으로 1('E')이 사용된다.

보통 새로운 Context 객체를 직접 생성하지는 않는다. 대신에 `getcontext()`나 `localcontext()` 함수를 사용해서 현재 사용 중인 Context 객체를 얻는다. 그리고 이 객체를 필요에 따라 수정한다. 이 절에서 나중에 관련 예를 살펴볼 때 이해를 돋기 위해서 먼저 컨텍스트 매개변수들을 좀 더 자세히 살펴볼 필요가 있다.

`rounding` 매개변수에 의해서 설정되는 반올림 방식은 다음 값 중 하나를 가진다.

상수	설명
ROUND_CEILING	양의 무한대로 반올림한다. 예를 들어, 2.52는 2.6으로 올림하고 -2.58은 -2.5로 올림한다.
ROUND_DOWN	영 쪽으로 반올림한다. 예를 들어, 2.58은 2.5로 내림하고 -2.58는 -2.5으로 올림한다.
ROUND_FLOOR	음의 무한대로 반올림한다. 예를 들어, 2.58은 2.5로 내림하고 -2.52는 -2.6으로 내림한다.
ROUND_HALF_DOWN	소수점 부분이 절반보다 크면 영에서 먼 쪽으로 반올림하고 그렇지 않으면 영 쪽으로 반올림한다. 예를 들어, 2.58은 2.6으로 올림하고 2.55는 2.5로 내림한다. -2.55는 -2.5로 올림하고 -2.58은 -2.6으로 내림한다.
ROUND_HALF_EVEN	소수점 부분이 정확히 절반일 경우 바로 앞 숫자가 짹수이면 내림하고 홀수이면 올림한다는 점을 제외하고 ROUND_HALF_DOWN과 동일하다.
ROUND_HALF_UP	소수점 부분이 정확히 절반인 경우 영에서 먼 쪽으로 반올림된다 는 것을 제외하고 ROUND_HALF_DOWN과 동일하다. 예를 들어, 2.55는 2.6으로 올림하고 -2.55는 -2.6으로 내림한다.
ROUND_UP	영에서 먼쪽으로 반올림한다. 예를 들어, 2.52는 2.6으로 올림하고 -2.52는 -2.6으로 내림한다.
ROUND_05UP	영 쪽으로 반올림했을 때 마지막 숫자가 0이나 5이면 영에서 먼 쪽 으로 반올림하고 아니면 영 쪽으로 반올림한다. 예를 들어, 2.54는 2.6으로 올림하고 2.64는 2.6으로 내림한다.

Context()의 traps와 flags 매개변수는 신호 목록을 나타낸다. 신호는 계산 중에 발생할 수 있는 산술 예외를 가리킨다. traps에 없으면 신호는 무시된다. 아니면 예외가 발생한다. 다음 신호들이 정의되어 있다.

신호	설명
Clamped	허용된 범위를 맞추기 위해서 지수가 조정되었다.
DivisionByZero	유한 수가 0에 의해 나누어졌다.
Inexact	반올림 에러가 발생하였다.
InvalidOperation	유효하지 않은 연산이 수행되었다.
Overflow	반올림 후에 지수가 Emax를 넘어섰다. Inexact와 Rounded도 발생 시킨다.
Rounded	반올림이 일어났다. 아무런 정보를 잃어버리지 않았어도 발생될 수 있다(예를 들어, “1.00”이 “1.0”으로 반올림될 때).
Subnormal	지수가 반올림 이전에 Emin보다 작다.
Underflow	수리 언더플로우. 결과가 0으로 반올림됨. Inexact와 Subnormal도 생성함.

신호 이름은 여러 검사에 사용할 수 있는 파이썬 예외에 대응된다. 다음 예를 보자.

```

try:
    x = a/b
except decimal.DivisionByZero:
    print "Division by zero"

```

예외처럼 신호들도 계층 구조를 이룬다.

```

ArithError (내장 예외)
  DecimalException
    Clamped
    DivisionByZero
  Inexact
    Overflow
    Underflow
  InvalidOperation
  Rounded
    Overflow
    Underflow
  Subnormal
    Underflow

```

여기서 Overflow와 Underflow는 부모 신호도 발생시키기 때문에 한 번 이상씩 나타난다. decimal.DivisionByZero 신호는 내장 예외 DivisionByZero에서 상속받는다.

많은 경우 산술 신호는 조용히 무시된다. 예를 들어, 계산 과정 동안 버림(round-off) 에러가 발생할 수 있지만 예외는 발생되지 않는다. 이럴 때는 신호 이름을 사용하여 계산 상태를 나타내는 플래그를 검사할 수 있다. 다음 예를 보자.

```

ctxt = decimal.getcontext() # 현재 컨텍스트
x = a + b
if ctxt.flags[Rounded]:
    print "Result was rounded!"

```

플래그가 일단 설정되면 clear_flags()가 호출되어 제거될 때까지 설정된 채로 남아 있게 된다. 따라서 전체 계산 과정을 수행한 후 마지막에 한 번만 예러 검사를 수행해도 된다.

다음 속성과 메서드를 사용하여 기존 Context 객체 c에 대한 설정을 변경할 수 있다.

c.capitals

지수 문자로 E가 사용되는지 e가 사용되는지를 나타내기 위해 1이나 0으로 설정되는 플래그.

c.Emax

최대 지수를 나타내는 정수.

c.Emin

최소 지수를 나타내는 정수.

c.prec

정밀도 숫자 개수를 나타내는 정수.

c.flags

신호의 현재 플래그 값을 담는 사전. 예를 들어, c.flags[Rounded]는 Rounded 신호의 현재 플래그 값을 반환한다.

c.rounding

적용되는 반올림 규칙. 예를 들어, ROUND_HALF_EVEN.

c.traps

파이썬 예외를 발생시킬 신호들을 나타내는 True나 False 값을 담은 사전. 예를 들어, c.traps[DivisionByZero]는 보통 True로 설정되는 반면에 c.traps[Rounded]는 False로 설정된다.

c.clear_flags()

모든 플래그를 재설정한다(c.flags를 초기화한다).

c.copy()

컨텍스트 c의 복사본을 반환한다.

c.create_decimal(value)

c를 컨텍스트로 사용하여 새로운 Decimal 객체를 생성한다. 기본 컨텍스트의 정밀도나 반올림 방식을 덮어쓰면서 십진수를 생성하고자 할 때 유용하다.

함수와 상수

decimal 모듈에는 다음 함수와 상수들이 있다.

getcontext()

현재 십진수 컨텍스트를 반환한다. 각 스레드는 자신만의 십진수 컨텍스트를 갖기 때문에 이 함수는 호출 스레드의 컨텍스트를 반환한다.

localcontext([c])

c의 복사본을 with문의 몸체 안에 있는 문장들을 위한 컨텍스트로 설정하는 컨

텍스트 관리자를 생성한다. `c`를 생략하면 현재 컨텍스트의 복사본이 사용된다. 다음은 이 함수를 사용하여 일련의 문장들에 대해 소수점 다섯 자리 정밀도를 사용하는 예를 보여준다.

```
with localcontext( ) as c:
    c.prec = 5
    문장들
```

setcontext(c)

호출 스레드의 십진수 컨텍스트를 `c`로 설정한다.

BasicContext

아홉 개 숫자 정밀도를 갖는 미리 만들어진 컨텍스트. 반올림 방식은 ROUND_HALF_UP이다. `Emin`은 -999999999이다. `Emax`는 999999999이다. `traps`는 Inexact, Rounded 그리고 Subnormal을 제외하고는 모두 활성화된다.

DefaultContext

새로운 컨텍스트를 생성할 때 사용할 기본 컨텍스트(여기서 저장된 값은 새로운 컨텍스트를 만들 때 기본 값으로 사용된다). 28개 숫자 정밀도를 사용한다. ROUND_HALF_EVEN 반올림 방식이 사용된다. `traps`로는 Overflow, InvalidOperation 그리고 DivisionByZero가 설정된다.

ExtendedContext

미리 만들어진 아홉 개 숫자 정밀도를 가지는 컨텍스트. 반올림 방식은 ROUND_HALF_EVEN이다. `Emin`은 -999999999이다. `Emax`는 999999999이다. `traps`는 모두 비활성화된다. 예외를 발생시키는 대신 `NaN`이나 `Infinity` 결과 값을 생성한다.

Inf

`Decimal("Infinity")`과 동일하다.

negInf

`Decimal("-Infinity")`과 동일하다.

NaN

`Decimal("NaN")`과 동일하다.

예

다음은 십진수의 기본적인 사용 방식을 보여준다.

```
>>> a = Decimal("42.5")
>>> b = Decimal("37.1")
>>> a + b
Decimal("79.6")
>>> a / b
Decimal("1.145552560646900269541778976")
>>> divmod(a,b)
(Decimal("1"), Decimal("5.4"))
>>> max(a,b)
Decimal('42.5')
>>> c = [Decimal("4.5"), Decimal("3"), Decimal("1.23e3")]
>>> sum(c)
Decimal("1237.5")
>>> [10*x for x in c]
[Decimal("45.0"), Decimal("30"), Decimal("1.230e4")]
>>> float(a)
42.5
>>> str(a)
'42.5'
```

다음은 컨텍스트의 값을 변경하는 예를 보여준다.

```
>>> getcontext().prec = 4
>>> a = Decimal("3.4562384105")
>>> a
Decimal("3.4562384105")
>>> b = Decimal("5.6273833")
>>> getcontext().flags[Rounded]
0
>>> a + b
9.084
>>> getcontext().flags[Rounded]
1
>>> a / Decimal("0")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    decimal.DivisionByZero: x / 0
>>> getcontext().traps[DivisionByZero] = False
>>> a / Decimal("0")
Decimal("Infinity")
```

Note

- Decimal과 Context 객체에는 십진수 연산의 표현과 작동 방식을 저수준에서 제어하는 여러 메서드가 있다. 이 모듈을 사용할 때 필수적이지는 않기 때문에 이곳에서 다루지는 않는다. 자세한 정보는 온라인 문서인 <http://docs.python.org/library/decimal.html>를 참고하라.
- 십진수 컨텍스트는 스레드별로 고유하다. 컨텍스트를 바꾸면 현재 스레드에만 영향이 있다.
- 특별한 숫자인 Decimal("sNaN")은 신호를 생성하는 NaN으로서 사용할 수 있다. 이 숫자는 내장 함수에 의해 생성되지는 않는다. 하지만 계산 과정에서 나타날 경우 에러가 항상 신호를 발생시키게 된다. 이 값은 조용히 무시하면 안 되는 에러를 발생시켜야 하는 유효하지 않은 계산을 나타내는 데 사용할 수 있다. 예를 들어, 어떤 함수에서 결과로 sNaN을 반환할 수 있다.
- 값 0은 양수나 음수일 수 있다(즉, Decimal(0)과 Decimal("-0")). 이 두 영은 같다고 평가된다.
- 계산에 드는 부하가 아주 크기 때문에 이 모듈은 고성능 과학 계산에 적합하지 않을 수 있다. 또한 그러한 응용에서 이진 부동 소수점 대신 십진수 부동 소수점을 사용하는 것이 실제적인 이점이 별로 없는 경우가 많다.
- 부동 소수점 표현이라든지 에러 분석에 관한 심도 있는 논의는 이 책의 범위를 벗어난다. 더 자세히 알고 싶은 독자는 수치 해석에 관련된 책을 참고하기 바란다. 1991년 3월 Association for Computing Machinery의 Computing Surveys에 실린 데이비드 골드버그가 쓴 '부동 소수점 계산에 관해서 모든 컴퓨터 과학자가 알아야 할 내용'(What Every Computer Scientist Should Know About Floating-Point Arithmetic)이라는 기사가 읽을 만하다(인터넷에서 제목으로 검색하면 쉽게 찾을 수 있다).
- IBM 범용 십진수 계산 명세서(IBM General Decimal Arithmetic Specification)에도 더 자세한 정보가 나와 있고 이 문서 역시 온라인 검색 엔진을 통해 쉽게 찾을 수 있다.

fractions

fractions 모듈은 유리수를 나타내는 Fraction 클래스를 정의한다. Fraction 클래스의 인스턴스는 다음 생성자들을 사용하여 세 가지 방식으로 생성할 수 있다.

Fraction([numerator [,denominator]])

새로운 유리수를 생성한다. numerator(분자)와 denominator(분모)는 정수 값이고 기본으로 각각 0과 1이 사용된다.

Fraction(fraction)

fraction^o] numbers.Rational의 인스턴스이면 fraction과 동일한 값을 갖는 유리수

를 생성한다.

Fraction(s)

s가 “3/7”이나 “4/7” 같은 유리수를 담고 있으면 동일한 값을 가진 유리수가 생성된다. s가 “1.25” 같은 실수를 담고 있으면 해당 값을 갖는 유리수가 생성된다 (예를 들어, Fraction(5,4)).

다음 클래스 메서드들은 다른 타입의 객체로부터 Fraction 인스턴스를 생성하는데 사용된다.

Fraction.from_float(f)

부동 소수점 수 f의 정확한 값을 나타내는 유리수를 생성한다.

Fraction.from_decimal(d)

Decimal 객체 d의 정확한 값을 나타내는 유리수를 생성한다.

다음은 이 함수들을 사용하는 예를 보여준다.

```
>>> f = fractions.Fraction(3,4)
>>> g = fractions.Fraction("1.75")
>>> g
Fraction(7, 4)
>>> h = fractions.Fraction.from_float(3.1415926)
Fraction(3537118815677477, 1125899906842624)
>>>
```

Fraction의 인스턴스 f는 일반적인 수학 연산을 모두 지원한다. 분자와 분모는 각각 f.numerator와 f.denominator 속성에 저장된다. 인스턴스에는 다음 메서드도 있다.

f.limit_denominator([max_denominator])

f에 가장 가까운 값을 갖는 유리수를 반환한다. max_denominator는 사용할 가장 큰 분모를 지정하고 기본으로 1000000이 사용된다.

다음은 Fraction 인스턴스를 사용하는 예를 보여준다(앞의 예에서 생성한 값을 사용하여).

```
>>> f + g
Fraction(5, 2)
>>> f * g
Fraction(21, 16)
>>> h.limit_denominator(10)
Fraction(22, 7)
>>>
```

fractions 모듈에는 다음 함수 하나가 있다.

gcd(a, b)

정수 a와 b의 최대 공약수를 계산한다. 결과는 b가 영이 아닐 경우 b와 같은 부호를 갖고 b가 영일 경우 a 와 같은 부호를 갖는다.

math

math 모듈에는 다음 표준 수학 함수들이 들어 있다. 이 함수들은 정수와 실수에 대해서 작동하고 복소수에 대해서는 작동하지 않는다(별개의 모듈인 cmath가 복소수에 대한 비슷한 연산을 수행하는 데 사용된다). 모든 함수의 반환 값은 실수이다. 모든 삼각 함수는 라디안(radian)을 가정한다.

함수	설명
<code>acos(x)</code>	x의 아크 코사인(arc cosine)을 반환한다.
<code>acosh(x)</code>	x의 쌍곡(hyperbolic) 아크 코사인을 반환한다.
<code>asin(x)</code>	x의 아크 사인(arc sine)을 반환한다.
<code>asinh(x)</code>	x의 쌍곡 아크 사인을 반환한다.
<code>atan(x)</code>	x의 아크 탄젠트(tangent)를 반환한다.
<code>atan2(y, x)</code>	atan(y / x)를 반환한다.
<code>atanh(x)</code>	x의 쌍곡 아크 탄젠트를 반환한다.
<code>ceil(x)</code>	x의 천장 값을 반환한다.
<code>copysign(x,y)</code>	y와 동일한 부호를 가진 x를 반환한다.
<code>cos(x)</code>	x의 코사인을 반환한다.
<code>cosh(x)</code>	x의 쌍곡 코사인을 반환한다.
<code>degrees(x)</code>	x를 라디안에서 각도(degree)로 바꾼다.
<code>radians(x)</code>	x를 각도에서 라디안으로 바꾼다.
<code>exp(x)</code>	$e^{**} x$ 를 반환한다.
<code>fabs(x)</code>	x의 절대값을 반환한다.
<code>factorial(x)</code>	x의 계승을 반환한다.
<code>floor(x)</code>	x의 바닥 값을 반환한다.
<code>fmod(x, y)</code>	C로 작성된 fmod() 함수에 의해서 계산되는 것처럼 x % y를 반환한다.
<code>frexp(x)</code>	x의 양의 가수(mantissa)와 지수를 튜플로 반환한다.
<code>fsum(s)</code>	반복 가능한 순서열 s에 있는 부동 소수점 값들의 최대 정밀도 합을 반환한다. 뒤에 나오는 참고 부분에서 더 설명한다.
<code>hypot(x, y)</code>	유클리드 거리(Euclidean distance)인 $\sqrt{x^2 + y^2}$ 를 반환한다.
<code>isinf(x)</code>	x가 무한대이면 True를 반환한다.
<code>isnan(x)</code>	x가 NaN이면 True를 반환한다.

<code>ldexp(x, i)</code>	$x * (2 ** i)$ 를 반환한다.
<code>log(x [, base])</code>	주어진 <code>base</code> 에 대한 <code>x</code> 의 로그를 반환한다. <code>base</code> 를 생략하면 자연 로그를 계산한다.
<code>log10(x)</code>	x 의 기본수 10 로그를 반환한다.
<code>log1p(x)</code>	$1+x$ 의 자연 로그를 반환한다.
<code>modf(x)</code>	x 의 소수 부분과 정수 부분을 튜플로서 반환한다. 둘 다 <code>x</code> 와 동일한 부호를 가진다.
<code>pow(x, y)</code>	$x ** y$ 를 반환한다.
<code>sin(x)</code>	x 의 사인을 반환한다.
<code>sinh(x)</code>	x 의 쌍곡 사인을 반환한다.
<code>sqrt(x)</code>	x 의 제곱근을 반환한다.
<code>tan(x)</code>	x 의 탄젠트를 반환한다.
<code>tanh(x)</code>	x 의 쌍곡 탄젠트를 반환한다.
<code>trunc(x)</code>	x 를 0 쪽으로 가장 가까운 정수로 자른다.

다음 상수도 정의되어 있다.

상수	설명
<code>pi</code>	수학 상수 pi
<code>e</code>	수학 상수 e

Note

- 부동 소수점 수 `+inf`, `-inf`, `nan`은 `float()` 함수에 문자열을 전달하여 생성할 수 있다. 예를 들어, `float("inf")`, `float("-inf")` 그리고 `float("nan")`을 사용하면 된다.
- `math.fsum()` 함수는 소거 효과에 의한 부동 소수점 에러를 피하는 알고리즘을 사용하기 때문에 내장 함수인 `sum()`보다 더 정확한 결과를 낸다. 예를 들어, 순서열 `s = [1, 1e100, -1e100]`이 있다고 하자. 여기서 `sum(s)`은 0.0을 반환한다(큰 값 1e100에 10이 더해질 때 1이 사라지기 때문이다). `math.fsum(s)`는 정확한 값인 1.0을 반환한다. `math.fsum()`에 의해 사용되는 알고리즘은 1996년에 조나단 리처드 슈체(Jonathan Richard Shewchuk) 이 쓴 Carnegie Mellon University School of Computer Science Technical Report CMU-CS-96-140에 설명되어 있다.

numbers

`numbers` 모듈은 다양한 종류의 수를 조직화하기 위한 추상 기반 클래스들을 정의 한다. 숫자 클래스들은 계층 구조를 가지며 수준이 깊어짐에 따라 점진적으로 더 많은 기능이 제공된다.

Number

숫자 계층에서 최상위 클래스.

Complex

복소수를 나타내는 클래스. 이 타입의 숫자는 real과 imag 속성을 갖는다. 이 클래스는 Number에서 상속받는다.

Real

실수를 표현하는 클래스. Complex에서 상속받는다.

Rational

분수를 나타내는 클래스. 이 타입의 숫자는 numerator와 denominator 속성을 갖는다. Real에서 상속받는다.

Integral

정수를 나타내는 클래스. Rational에서 상속받는다.

이 모듈에 있는 클래스들은 인스턴스를 생성하는 데 사용하지 않는다. 대신 주어진 값에 대해 다양한 타입 검사를 수행하는 데 사용된다. 다음 예를 보자.

```
if isinstance(x, numbers.Number)      # x는 아무 숫자나 가능
    문장들
if isinstance(x, numbers.Integral)    # x는 정수 값
    문장들
```

타입 검사가 True를 반환하면 x는 해당 타입에 적용되는 모든 일반적인 수리 연산에 쓰일 수 있다는 것을 의미하고 또 complex(), float()나 int() 같은 내장 타입으로 변환할 수 있다는 것을 의미한다.

여기에서 정의된 추상 기반 클래스는 숫자를 흉내 내는 사용자 정의 클래스의 기반 클래스로도 사용된다. 이렇게 하면 타입 검사를 하기 쉽고 또 필요한 모든 메서드를 반드시 구현해야 하기 때문에 추가적인 안정성 검사도 이루어진다. 다음 예를 보자.

```
>>> class Foo(numbers.Real): pass
...
>>> f = Foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract
methods
__abs__, __add__, __div__, __eq__, __float__, __floordiv__,
__le__, __lt__, __mod__, __mul__, __neg__, __pos__, __pow__,
```

```
--radd__, __rdiv__, __rfloordiv__, __rmod__, __rmul__, __rpow__,
__rtruediv__, __truediv__, __trunc__
>>>
```

Note

- 주상 기반 클래스에 관해서는 7장을 참고하도록 한다.
- 이 모듈의 타입 계층과 적절한 사용법에 관한 정보는 PEP 3141(<http://www.python.org/dev/peps/pep-3141>)에서 찾을 수 있다.

random

random 모듈은 유사 무작위(pseudo-random) 수를 생성하거나 다양한 분포에 따라서 실수 값을 생성하는 데 사용되는 함수들을 제공한다. 이 모듈에 있는 대부분의 함수는 메르세네 트위스터(Mersenne Twister) 생성기를 사용해서 [0.0, 1.0] 범위 안에서 균일하게 분포된 수들을 생성하는 random() 함수에 의존한다.

씨 뿌리기와 초기화

다음 함수는 내부 무작위 수 생성기의 상태를 제어하는 데 사용된다.

seed([x])

무작위 수 생성기를 초기화한다. x를 생략하거나 None을 사용하면 시스템 시간이 생성기에 씨를 뿌리는 데 사용된다. 그렇지 않고 x가 정수이거나 긴 정수이면 그 값이 사용된다. x가 정수가 아니면 x는 해싱이 가능한 객체이어야 하고 hash(x)의 값이 씨로서 사용된다.

getstate()

생성기의 현재 상태를 나타내는 객체를 반환한다. 이 객체는 나중에 setstate()에 전달해서 상태를 복구하는 데 사용할 수 있다.

setstate(state)

getstate()에 의해서 반환된 객체로부터 무작위 수 생성기의 상태를 복구한다.

jumpahead(n)

생성기의 상태를 random()을 연속으로 n 번 호출했을 때의 상태로 빠르게 변경한다. n은 음수가 아닌 정수이어야 한다.

무작위 정수

다음 함수는 무작위 정수를 생성하는 데 사용된다.

getrandbits(k)

k 개의 무작위 비트를 담은 긴 정수를 생성한다.

randint(a,b)

범위 $a \leq x \leq b$ 에 있는 무작위 정수 x를 생성한다.

randrange(start,stop [,step])

range(start, stop, step)에 있는 무작위 정수를 반환한다. 마지막 값은 포함되지 않는다.

무작위 순서열

다음 함수는 순서열 데이터를 무작위화하는 데 사용된다.

choice(seq)

비어 있지 않은 순서열 seq에서 무작위로 선택된 원소 하나를 반환한다.

sample(s, len)

순서열 s에서 무작위로 선택된 원소들을 담은 길이 len인 순서열을 반환한다. 결과 순서열에 있는 원소들은 선택된 순서대로 배열되어 있다.

shuffle(x [,random])

리스트 x에 있는 항목들을 무작위로 섞는다. random은 옵션인 인수로서 무작위 생성 함수를 지정한다. 제공될 경우 이 함수는 인수를 받지 않고 범위 [0.0, 1.0)에 속하는 부동 소수점 수를 반환해야 한다.

실수 무작위 분포

다음 함수들은 무작위로 실수를 생성한다. 분포의 이름이나 매개변수의 이름은 확률 및 통계 분야에서 사용하는 표준 이름에 대응된다. 더 자세한 정보는 관련도서를 참고하기 바란다.

random()

범위 [0.0, 1.0)에 속하는 무작위 수를 반환한다.

uniform(a, b)

범위 [a, b)에 속하는 균일 분포 무작위 수를 반환한다.

betavariate(alpha, beta)

베타 분포로부터 0과 1 사이에 있는 값을 반환한다. $\alpha > -1$ 이고 $\beta > -1$ 이어야 한다.

cunifvariate(mean, arc)

원형(circular) 균일 분포. mean은 평균 각을, arc는 평균 각을 중심으로 한 분포의 범위를 나타낸다. 두 값 모두 0과 pi 사이에 속하는 라디안으로 지정해야 한다. 반환되는 값은 범위 ($mean - \frac{arc}{2}$, $mean + \frac{arc}{2}$)에 속한다.

expovariate(lambd)

지수 분포. lambd는 1.0을 원하는 평균값으로 나눈 값이어야 한다. 반환되는 값은 범위 [0, +Infinity)에 속한다.

gammavariate(alpha, beta)

감마 분포. $\alpha > -1$, $\beta > 0$.

gauss(mu, sigma)

평균값 mu와 표준 편차 sigma를 갖는 가우스 분포. normalvariate()보다 약간 더 빠르다.

lognormvariate(mu, sigma)

로그 정규 분포. 이 분포에 자연 로그를 취하면 평균값 mu와 표준 편차 sigma를 갖는 정규 분포가 된다.

normalvariate(mu, sigma)

평균값 mu와 표준 편차 sigma를 갖는 정규 분포.

paretovariate(alpha)

형상 모수(shape parameter)가 alpha인 파레토 분포.

triangular([low [, high [, mode]]])

삼각 분포. 무작위 수 n은 모드 mode를 갖고 $low \leq n \leq high$ 범위에 속한다. 기본으로 low는 0, high는 1.0 그리고 mode는 low와 high의 중간값으로 설정된다.

vonmisesvariate(mu, kappa)

mu는 0과 $2 * \pi$ 사이에 속하는 라디안으로 표현된 평균 각도이고 kappa는 음이 아닌 농축 계수(concentration factor)인 폰 미제스(von Mises) 분포이다. kappa가 0이면 이 분포는 0에서 $2 * \pi$ 에 속하는 균일 무작위 각도를 생성하게 된다.

weibullvariate(alpha, beta)

척도 모수(scale parameter) alpha와 형상 모수 beta를 갖는 와이블 분포.

Note

- 이 모듈에 있는 함수들은 스레드 안전이 보장되지 않는다. 여러 스레드에서 무작위 수를 생성하는 경우라면 동시 접근을 막기 위해서 락을 사용해야 한다.
- 무작위 수 생성기의 주기(반복이 시작되기 전까지 숫자 개수)는 $2^{**19937-10}$ 이다.
- 이 모듈에 의해서 생성되는 무작위 수들은 결정적(deterministic) 특성을 갖기 때문에 임호 기술에 사용해서는 안 된다.
- random.Random에서 상속받고 random(), seed(), getstate()와 jumpahead() 메서드를 구현해서 새로운 무작위 수 생성기 타입을 정의할 수 있다. 이 모듈에 있는 다른 함수들은 내부적으로 Random의 메서드로서 구현되어 있다. 이 메서드들은 새로운 무작위 수 생성기 인스턴스의 메서드로서 접근할 수 있다.
- 이 모듈에는 두 개의 다른 무작위 수 생성기 클래스, WichmannHill과 SystemRandom이 들어 있다. 해당 클래스의 인스턴스를 생성하고 앞에 나온 함수를 메서드로서 호출하면 된다. WichmannHill 클래스는 파이썬의 초기 릴리스 버전에서 사용되었던 위치만 힐(Wichmann-Hill) 생성기를 구현한다. SystemRandom 클래스는 시스템 무작위 수 생성기인 os.urandom()를 사용하여 무작위 수를 생성한다.

15장

P y t h o n E s s e n t i a l R e f e r e n c e

데이터 구조, 알고리즘과 코드 단순화

이 장에서는 프로그램을 작성할 때 도움이 되는 데이터 구조와 알고리즘에 대해서 알아보고 반복, 함수형 프로그램, 컨텍스트 관리자, 클래스 등을 작성할 때 코드를 단순하게 만드는 데 사용할 수 있는 모듈을 살펴본다. 여기서 다루는 모듈은 파이썬 내장 타입과 함수를 확장한 것이라고 생각하면 된다. 효율적으로 구현되어 있기 때문에 어떤 종류의 문제에 대해서는 내장 타입이나 함수를 쓰는 것보다 이 장에서 다루는 모듈을 사용하는 것이 나은 경우도 있다.

abc

abc 모듈은 새로운 추상 기반 클래스를 정의하는 데 사용할 수 있는 메타클래스와 장식자 한 쌍을 담고 있다.

ABCMeta

추상 기반 클래스를 나타내는 메타클래스이다. 추상 클래스를 정의하려면 ABCMeta를 메타클래스로 사용하여 클래스를 정의하면 된다. 다음 예를 보자.

```
import abc
class Stackable: # 파이썬 3에서는 다음 문법을 사용한다.
    __metaclass__ = abc.ABCMeta # class Stackable(metaclass=abc.ABCMETA)
    ...
    
```

이렇게 생성된 클래스는 보통의 클래스와 크게 다른 점이 몇 가지 있다.

- 첫째, 나중에 설명할 `abstractmethod`와 `abstractproperty` 장식자를 사용하여 추상 클래스에 메서드나 프로퍼티를 정의하면 파생 클래스에서는 해당 메서드나 프로퍼티에 대해서 추상이 아닌 구현을 제공할 경우에만 인스턴스를 생성할 수 있다.
- 둘째, 추상 클래스는 어떤 타입을 논리적 하위 클래스로 등록하는 데 사용할 수 있는 클래스 메서드 `register(subclass)`를 갖고 있다. 이 함수로 `subclass` 타입을 등록했다면 `isinstance(x, AbstractClass)` 연산은 `x`가 `subclass`의 인스턴스일 경우 `True`를 반환한다.
- 마지막으로 추상 클래스에는 특수한 클래스 메서드인 `__subclasshook__(cls, subclass)`를 추가로 정의할 수 있다. 이 메서드를 구현할 때는 타입 `subclass`가 하위 클래스로 간주될 경우 `True`를 반환하고 `subclass`가 하위 클래스가 아닐 경우 `False`를 반환하며 아무런 정보가 없을 경우 `NotImplemented` 예외를 발생시켜야 한다.

abstractmethod(method)

`method`를 추상 메서드로 선언하는 장식자. 추상 기반 클래스에서 사용하면 직접 상속을 통해 정의한 파생 클래스에서 해당 메서드에 대한 추상이 아닌 구현을 제공할 때만 인스턴스를 생성할 수 있다. 이 장식자는 추상 기반 클래스의 `register()` 메서드로 등록한 하위 클래스에는 아무런 영향을 주지 않는다.

abstractproperty(fget [, fset [, fdel [, doc]]])

추상 프로퍼티를 생성한다. 매개변수들은 일반 `property()` 함수의 것과 동일하다. 추상 기반 클래스에서 쓰일 경우 직접 상속을 통해 정의한 파생 클래스에서는 해당 프로퍼티에 대한 추상이 아닌 구현을 제공할 때만 인스턴스를 생성할 수 있다.

다음 코드는 간단히 추상 클래스를 정의하는 예를 보여준다.

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Stackable:          # 파이썬 3에서는 다음 문법을 사용한다.
    __metaclass__ = ABCMeta      # class Stackable(metaclass=ABCMeta)
    @abstractmethod
    def push(self, item):
        pass
    @abstractmethod
    def pop(self):
        pass
    @abstractproperty
```

```
def size(self):
    pass
```

다음 예에서는 Stackable을 상속하여 클래스를 만든다.

```
class Stack(Stackable):
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
```

Stack의 인스턴스를 생성하려고 하면 다음 에러 메시지가 출력된다.

```
>>> s = Stack()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Stack with abstract
methods size
>>>
```

Stack에 size() 프로퍼티를 추가하면 에러가 사라진다. size() 프로퍼티는 Stack에 추가해도 되고 다음과 같이 Stack에서 상속받은 클래스에 추가해도 된다.

```
class CompleteStack(Stack):
    @property
    def size(self):
        return len(self.items)
```

다음은 CompleteStack 객체를 사용하는 예이다.

```
>>> s = CompleteStack()
>>> s.push("foo")
>>> s.size
1
>>>
```

참고

7장 클래스와 객체지향 프로그래밍, numbers(310페이지), collections(323페이지)

array

array 모듈은 거의 리스트처럼 작동하지만 한 종류의 타입만 담는 새로운 객체 타입인 array를 정의한다. 배열(array)이 담을 타입은 표 15.1에 나와 있는 타입 코드 중 하나를 사용하여 생성 시점에 지정한다.

표 15.1 타입 코드

타입 코드	설명	C 타입	최소 크기(바이트)
'b'	8비트 정수	signed char	1
'B'	8비트 부호 없는 정수	unsigned char	1
'u'	유니코드 문자	PY_UNICODE	2 또는 4
'h'	16비트 정수	short	2
'H'	16비트 부호 없는 정수	unsigned short	2
'i'	정수	int	4 또는 8
'I'	부호 없는 정수	unsigned int	4 또는 8
'l'	긴 정수	long	4 또는 8
'L'	부호 없는 긴 정수	unsigned long	4 또는 8
'f'	단밀도 실수	float	4
'd'	배밀도 실수	double	8

정수와 긴 정수 표현 방식은 머신 아키텍처에 의해 결정된다(32비트이거나 64비트일 수 있다). 파이썬 2에서는 'L'이나 'l'로 저장된 값을 반환할 때 긴 정수로 반환한다.

이 모듈에는 다음 타입이 정의되어 있다.

`array(typecode [, initializer])`

`typecode` 타입인 배열을 생성한다. `initializer`는 배열의 값을 초기화하는 데 사용할 문자열 또는 값 리스트이다. 다음 속성과 메서드를 `array` 객체 `a`에 대해서 사용할 수 있다.

항목	설명
<code>a.typecode</code>	배열을 만드는 데 사용한 타입 코드 문자
<code>a.itemsize</code>	배열에 저장된 항목들의 크기(바이트)
<code>a.append(x)</code>	<code>x</code> 를 배열의 끝에 추가한다.
<code>a.buffer_info()</code>	배열을 저장한 메모리 위치와 버퍼의 길이 정보를 담은 (<code>address</code> , <code>length</code>)를 반환
<code>a.byteswap()</code>	배열에 있는 모든 항목의 바이트 순서를 빅 엔디안에서 리틀 엔디안 또는 그 반대로 바꾼다. 정수 값만 지원한다.
<code>a.count(x)</code>	<code>a</code> 에서 <code>x</code> 가 나타난 횟수를 반환한다.
<code>a.extend(b)</code>	<code>b</code> 를 배열 <code>a</code> 의 끝에 추가한다. <code>b</code> 는 <code>a</code> 에 있는 것과 동일한 타입의 원소를 가지는 배열이거나 반복 가능한 객체일 수 있다.

<code>a.fromfile(f, n)</code>	파일 객체 f에서 n개의 항목을 읽어온 다음 (이진 형식으로) 배열의 끝에 추가한다. f는 반드시 파일 객체이어야 한다. 항목 개수가 n개 보다 작으면 EOFError 예외가 발생한다.
<code>a.fromlist(list)</code>	list에 있는 항목들을 배열의 끝에 추가한다. list는 아무 반복 가능한 객체나 될 수 있다.
<code>a.fromstring(s)</code>	fromfile()로 읽을 때처럼 s를 이진 값들의 문자열로 해석하여 문자열 s에 있는 항목들을 추가한다.
<code>a.index(x)</code>	a에서 x가 처음으로 나타난 색인을 반환한다. 찾을 수 없으면 ValueError를 발생시킨다.
<code>a.insert(i, x)</code>	x를 위치 i 바로 앞에 삽입한다.
<code>a.pop([i])</code>	배열에서 항목 i를 제거하면서 반환한다. i를 생략하면 마지막 원소가 제거된다.
<code>a.remove(x)</code>	배열에서 처음으로 나타나는 x를 제거한다. 찾지 못하면 ValueError를 발생시킨다.
<code>a.reverse()</code>	배열의 순서를 뒤집는다.
<code>a.tofile(f)</code>	모든 항목을 파일 f에 쓴다. 데이터는 머신에 특화된 이진 포맷으로 저장된다.
<code>a.tolist()</code>	배열을 보통의 리스트로 변환한다.
<code>a.tostring()</code>	배열을 이진 데이터 문자열로 변환한다. tofile()로 쓰여지는 데이터와 동일하다.
<code>a.tounicode()</code>	배열을 유니코드 문자열로 변환한다. 배열의 타입이 'u'가 아니면 ValueError가 발생한다.

어떤 항목을 배열에 추가할 때 항목의 타입이 배열을 생성할 때의 타입과 일치하지 않으면 TypeError 예외가 발생한다.

array 모듈은 데이터 리스트를 저장할 때 공간을 효율적으로 사용할 필요가 있고 리스트의 모든 항목이 동일한 타입을 가질 것이라는 것을 알고 있을 때 유용하게 쓸 수 있다. 예를 들어, 리스트에 천만 개의 정수를 저장하려면 약 160MB의 메모리가 필요하지만 배열을 사용하면 40MB의 메모리만 있으면 된다. 공간이 크게 절약되기는 하지만 배열에 대한 기본 연산이 리스트에 대응하는 연산보다 빠른 편은 아니다. 사실, 더 느릴 수도 있다.

배열을 사용해서 계산할 때 리스트 생성 연산을 주의해서 사용해야 한다. 예를 들어, 배열에 리스트 내포를 사용하면 먼저 배열이 리스트로 변환되기 때문에 공간 절약의 이점이 사라지게 된다. 생성기 표현식을 사용해서 새로운 배열을 만드는 것 이 더 낫다. 다음 예를 보자.

```
a = array.array("i", [1,2,3,4,5])
b = array.array(a.typecode, (2*x for x in a)) # a로 새로운 배열을 만든다.
```

공간을 절약하기 위해서 배열을 사용하는 것이기 때문에 “제자리(in-place)” 연산을 수행하는 것이 좋다. 다음과 같이 enumerate()를 사용하면 이를 효율적으로 수행할 수 있다.

```
a = array.array("i", [1,2,3,4,5])
for i, x in enumerate(a):
    a[i] = 2*x
```

큰 배열을 직접 수정하는 것이 생성기 표현식으로 새 배열을 생성하는 것보다 약 15 퍼센트 더 빠르다.

Note

- 이 모듈로 생성한 배열은 행렬이나 벡터 연산 같은 계산에 적합하지 않다. 예를 들어, 더하기 연산자는 두 배열의 대응하는 원소를 더하는 것이 아니라 단순히 한 배열을 다른 배열에 추가한다. 공간 및 계산 효율적인 배열을 생성하려면 <http://numpy.sourceforge.net/>에 있는 numpy 확장 기능을 사용하도록 한다. numpy API는 인터페이스가 많이 다르다.
- += 연산자는 다른 배열의 내용을 기준 배열에 추가한다. *= 연산자는 반복된 배열을 만든다.

참고

16장 struct (357페이지)

bisect

bisect 모듈은 리스트를 정렬된 순서로 유지하는 데 필요한 기능을 제공한다. 대부분의 작업은 다음의 이등분 알고리즘으로 이루어진다.

bisect(list, item [, low [, high]])

list를 정렬된 순서로 유지하기 위해서 item을 list에 삽입할 곳의 색인을 반환한다. low와 high는 리스트에서 어느 부분을 들여다볼 것인지를 지정하는 색인이다. item이 리스트에 이미 있으면 삽입될 곳은 항상 기존 항목의 바로 오른쪽이다.

bisect_left(list, item [, low [, high]])

list를 정렬된 순서로 유지하기 위해서 item을 list에 삽입할 곳의 색인을 반환한다. low와 high는 리스트에서 어느 부분을 들여다볼 것인지를 지정하는 색인이다. item이 리스트에 이미 있으면 삽입될 곳은 항상 기존 항목의 바로 왼쪽이다.

bisect_right(list, item [, low [, high]])

bisect()와 동일하다.

insort(list, item [, low [, high]])

item을 list에 정렬된 순서에 맞게 삽입한다. item이 이미 리스트에 있으면 새로운 항목은 기존 항목의 오른쪽에 삽입된다.

insort_left(list, item [, low [, high]])

item을 list에 정렬된 순서에 맞게 삽입한다. item이 이미 리스트에 있으면 새로운 항목은 기존 항목의 왼쪽에 삽입된다.

insort_right(list, item [, low [, high]])

insort()와 동일하다.

collections

collections 모듈은 유용한 몇 가지 컨테이너 타입에 대한 고성능 구현, 다양한 컨테이너를 위한 추상 기반 클래스, 이름 있는 튜플 객체를 생성하기 위한 유ти리티 함수를 담고 있다. 이어지는 절에서 각각을 설명한다.

deque와 defaultdict

collections 모듈에는 deque와 defaultdict, 두 가지 새로운 컨테이너가 정의되어 있다.

deque([iterable [, maxlen]])

양 끝을 가진 큐(deque: ‘데크’라고 읽는다) 객체를 나타내는 타입이다. iterable은 deque를 채우는 데 사용할 반복 가능한 객체이다. 데크는 항목을 큐의 아무 쪽 끝이나 삽입하거나 삭제할 수 있다. 데크 구현은 이 두 연산이 대략 $O(1)$ 과 같도록 최적화되어 있다. 이 점은 리스트 앞쪽에 연산을 수행하면 뒤쪽 모든 원소를 옆으로 밀어야 하는 리스트와 다르다. 옵션인 maxlen 인수가 주어지면 결과 deque 객체는 해당 크기의 원형 버퍼가 된다. 즉, 새로운 항목을 추가하려는데 공간이 부족하면 공간을 만들기 위해서 반대편에 있는 항목이 삭제된다.

deque의 인스턴스 d에는 다음 메서드들이 있다.

d.append(x)

x를 d의 오른쪽에 추가한다.

d.appendleft(x)

x를 d의 왼쪽에 추가한다.

d.clear()

d에서 모든 항목을 제거한다.

d.extend(iterable)

iterable에 있는 모든 항목을 오른쪽에 추가하여 d를 확장한다.

d.extendleft(iterable)

iterable에 있는 모든 항목을 왼쪽에 추가하여 d를 확장한다. 왼쪽에 차례대로 추가하기 때문에 iterable에 있는 항목은 d에서 반대 순서로 나타난다.

d.pop()

d의 오른쪽에서 항목을 하나 반환하고 제거한다. d가 비었으면 IndexError를 발생시킨다.

d.popleft()

d의 왼쪽에서 항목을 하나 반환하고 제거한다. d가 비었으면 IndexError를 발생시킨다.

d.remove(item)

처음으로 나타나는 item을 제거한다. item이 없으면 ValueError를 발생시킨다.

d.rotate(n)

모든 항목을 오른쪽으로 n번 옮긴다. n이 음수이면 왼쪽으로 회전시킨다.

많은 파이썬 프로그래머들이 데크를 잘 모른다. 이 타입은 장점이 많다. 먼저 구현이 아주 효율적이다. 심지어 프로세서 캐시의 작동 방식을 흉내 내기 위한 내부 데이터 구조까지 가지고 있다. 뒤쪽 끝에 항목을 추가하는 일은 내장 list 타입보다 약간 느리지만 앞쪽에 항목을 추가하는 일은 훨씬 빠르다. 데크에 새로운 항목을 추가하는 작업은 스레드 안전이 보장되기 때문에 큐를 구현하는 데 알맞다. 데크는 pickle 모듈로 직렬화할 수도 있다.

defaultdict([default_factory], ...)

없는 키를 처리하는 부분을 제외하고는 사전과 완전히 동일한 타입이다. 키를 검색했는데 없다면 default_factory로 지정한 함수가 호출되어 기본 값이 생성되고 이 값이 해당 키의 값으로 저장된다. defaultdict의 나머지 인수들은 내장 dict() 함수의 것과 동일하다. defaultdict의 인스턴스는 내장 사전과 동일한 연산을 지원한다. d.default_factory 속성은 첫 번째 인수로 전달된 함수를 저장하고 필요에 따라 수정할 수 있다.

defaultdict 객체는 데이터를 추적하기 위해서 사전을 컨테이너로서 쓸 때 유용하다. 예를 들어, 문자열 s에 있는 각 단어의 위치를 기록한다고 하자. 다음은 defaultdict를 사용해서 이것을 구현하는 방법을 보여준다.

```
>>> from collections import defaultdict
>>> s = "yeah but no but yeah but no but yeah"
>>> words = s.split( )
>>> wordlocations = defaultdict(list)
>>> for n, w in enumerate(words):
...     wordlocations[w].append(n)
...
>>> wordlocations
defaultdict(<type 'list'>, {'yeah': [0, 4, 8], 'but': [1, 3, 5, 7],
'no': [2, 6]})
```

이 예에서 어떤 단어를 처음 만났을 때 wordlocations[w] 검색은 실패한다. KeyError가 발생하는 대신에 default_factory에 지정한 list 함수가 호출되어 새 값을 생성한다. 내장 사전에는 비슷한 결과를 내는 setdefault() 메서드가 있지만 보통 코드가 이해하기 어렵고 더 느리다. 예를 들어, 앞에서 본 새 항목을 추가하는 문장은 wordlocations.setdefault(w, []).append(n)처럼 쓸 수 있다. 이해하기도 어렵고 간단히 속도를 재보면 defaultdict 객체를 사용하는 것보다 거의 두 배 느린다.

이름 있는 튜플

튜플은 간단한 데이터 구조를 표현하는 데 흔히 사용한다. 예를 들어, 네트워크 주소는 튜플 addr = (hostname, port)로 나타낼 수 있다. 튜플을 사용할 때 불편한 점은 개별 항목을 숫자 색인으로만 접근할 수 있다는 점이다. addr[0]이나 addr[1]처럼 말이다. 코드를 읽기 어려울뿐더러 모든 색인 값이 의미하는 바를 기억하고 있지 않는 한 관리하기도 어렵다(튜플 크기가 클수록 문제는 더 심각해진다).

collections 모듈에는 속성 이름으로 튜플 원소에 접근할 수 있는 tuple의 하위 클

래스를 생성하는 데 사용하는 namedtuple() 함수가 있다.

namedtuple(typename, fieldnames [, verbose])

이름이 typename인 tuple의 하위 클래스를 생성한다. 문자열인 fieldnames는 속성 이름 목록을 지정한다. 이 목록에 있는 이름들은 유효한 파이썬 식별자이어야 하고 밑줄 문자로 시작할 수 없으며 튜플에 나타날 항목과 동일한 순서로 정해야 한다. 예를 들어, ['hostname', 'port']처럼 지정한다. 또는 fieldnames을 'hostname port'나 'hostname, port' 같은 문자열로 지정해도 된다. 이 함수가 반환하는 값은 typename으로 지정한 값을 이름으로 갖는 클래스이다. 이 클래스는 이름 있는 튜플을 만드는 데 사용한다. verbose 플래그를 True로 설정하면 생성되는 클래스의 정의가 표준 출력으로 출력된다.

다음은 이 함수를 사용하는 예를 보여준다.

```
>>> from collections import namedtuple
>>> NetworkAddress = namedtuple('NetworkAddress', ['hostname', 'port'])
>>> a = NetworkAddress('www.python.org', 80)
>>> a.hostname
'www.python.org'
>>> a.port
80
>>> host, port = a
>>> len(a)
2
>>> type(a)
<class '__main__.NetworkAddress'>
>>> isinstance(a, tuple)
True
>>>
```

이 예에서 이름 있는 튜플인 NetworkAddress는 튜플의 구성 요소에 접근하는 데 a.hostname이나 a.port 같은 속성 검색을 사용할 수 있다는 점을 제외하고는 보통의 튜플과 모든 면에서 구별하기 힘들다. 내부 구현은 아주 효율적이다. 생성되는 클래스는 인스턴스 사전을 사용하지 않으며 내장 튜플보다 메모리 오버헤드가 적다. 그럼에도 여전히 모든 튜플 연산이 지원된다.

이름 있는 튜플은 데이터 구조로만 사용될 객체를 정의할 때 유용하다. 예를 들어 다음과 같이 클래스를 정의하는 대신,

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
```

```
self.price = price
```

다음과 같이 이름 있는 튜플을 정의하면 된다.

```
import collections
Stock = collections.namedtuple('Stock', 'name shares price')
```

두 버전 다 거의 동일하게 작동한다. 두 경우 모두 s.name, s.shares 등을 써서 필드에 접근할 수 있다. 이름 있는 튜플을 사용하면 메모리를 더 효율적으로 사용하고 풀어 헤치기 같은 다양한 튜플 연산을 사용할 수 있다는 장점이 있다. 예를 들어, 이름 있는 튜플 리스트가 있으면 for 루프에서 for name, shares, price in stockList 같은 문장을 사용해서 값을 풀어헤칠 수 있다. 이름 있는 튜플을 사용할 때 단점은 속성 검색이 클래스보다 덜 효율적이라는 점이다. s.shares는 s가 보통의 클래스일 때보다 이름 있는 튜플의 인스턴스일 때 두 배 이상 느린다.

이름 있는 튜플은 파이썬 표준 라이브러리에서 자주 사용된다. 여기에는 역사적인 이유가 있다. 원래 많은 라이브러리 모듈에서 파일, 스택 프레임이나 기타 저수준 정보를 반환하는 다양한 함수의 반환 값으로 튜플을 사용하였다. 이렇게 튜플을 사용하는 코드는 그리 멋지지 않다. 그래서 하위 호환성을 깨지 않으면서 코드를 더 깔끔하게 만들기 위해 이름 있는 튜플을 사용하게 되었다. 튜플을 사용할 때 다른 문제로 일단 튜플을 사용하게 되면 기대하는 필드 개수가 영원히 고정된다는 점 때문이다(예를 들어, 새 필드를 추가하면 오래된 코드가 작동하지 않을 수 있다). 라이브러리에서는 몇몇 함수에서 반환되는 데이터에 새로운 필드를 추가하려고 이름 있는 튜플의 일종을 사용한다. 가령 어떤 객체가 레거시 튜플 인터페이스를 지원하고 있을 때 나중에 필요에 따라 이름 있는 속성으로만 접근할 수 있는 값을 더 추가할 수 있다.

추상 기반 클래스

`collections` 모듈은 일련의 추상 기반 클래스들도 정의한다. 이 클래스들은 리스트, 집합, 사전 같은 다양한 컨테이너의 프로그래밍 인터페이스를 정의한다. 이 클래스들은 크게 두 가지 용도로 사용한다. 먼저, 내장 컨테이너 타입을 흉내 내려고 만든 사용자 정의 객체의 기반 클래스로 사용할 수 있다. 둘째, 타입 검사를 위해서 사용한다. 예를 들어, `s`가 순서열로서 작동하는지를 검사하려면 `isinstance(s, collections.Sequence)`처럼 하면 된다.

Container

모든 컨테이너를 위한 기반 클래스이다. in 연산자를 구현하는 추상 메서드인 __contains__() 하나를 정의한다.

Hashable

해시 테이블 키로서 사용할 수 있는 객체를 위한 기반 클래스. 단일 추상 메서드인 __hash__()를 정의한다.

Iterable

반복 프로토콜을 지원하는 객체를 위한 기반 클래스. 단일 추상 메서드인 __iter__()를 정의한다.

Iterator

반복자 객체를 위한 기반 클래스. 추상 메서드인 next()를 정의하고 Iterable에서 상속받았으며 아무것도 하지 않는 __iter__()의 기본 구현을 제공한다.

Sized

크기를 알 수 있는 컨테이너를 위한 기반 클래스. 추상 메서드인 __len__()을 정의한다.

Callable

함수 호출을 지원하는 객체를 위한 기반 클래스. 추상 메서드인 __call__()을 정의한다.

Sequence

순서열처럼 보이는 객체를 위한 기반 클래스. Container, Iterable과 Sized에서 상속 받았고 추상 메서드인 __getitem__()과 __len__()을 정의한다. __getitem__()과 __len__() 메서드만 사용하여 구현한 __contains__(), __iter__(), __reversed__(), index(), count()의 기본 구현을 제공한다.

MutableSequence

변경 가능한 순서열을 위한 기반 클래스. Sequence에서 상속받고 추상 메서드인 __setitem__()과 __delitem__()을 추가한다. append(), reverse(), extend(), pop(), remove(), __iadd__()의 대한 기본 구현을 제공한다.

Set

집합처럼 작동하는 객체를 위한 기반 클래스. Container, Iterable과 Sized에서 상속 받고 추상 메서드인 `__len__()`, `__iter__()`, `__contains__()`를 정의한다. 집합 연산자인 `__le__()`, `__lt__()`, `__eq__()`, `__ne__()`, `__gt__()`, `__ge__()`, `__and__()`, `__or__()`, `__xor__()`, `__sub__()`, `isdisjoint()`의 기본 구현을 제공한다.

MutableSet

변경 가능한 집합을 위한 기반 클래스. Set에서 상속받고 추상 메서드인 `add()`와 `discard()`를 추가한다. `clear()`, `pop()`, `remove()`, `__ior__()`, `__iand__()`, `__ixor__()`, `__isub__()`의 기본 구현을 제공한다.

Mapping

매핑(사전) 검색을 지원하는 객체를 위한 기반 클래스. Sized, Iterable과 Container에서 상속받고 추상 메서드인 `__getitem__()`, `__len__()`, `__iter__()`를 정의한다. `__contains__()`, `keys()`, `values()`, `items()`, `get()`, `__eq__()`, `__ne__()`의 기본 구현을 제공한다.

MutableMapping

변경 가능한 매핑 객체를 위한 기반 클래스. Mapping에서 상속받고 추상 메서드 `__setitem__()`과 `__delitem__()`을 정의한다. `pop()`, `popitem()`, `clear()`, `update()`, `setdefault()`의 구현을 제공한다.

MapView

매핑 뷰를 위한 기반 클래스. 매핑 뷰(mapping view)는 집합의 형태로 매핑 객체의 내부에 접근하는 데 사용하는 객체이다. 예를 들어, 키 뷰는 매핑 안에 있는 키들을 보여주는 집합 같은 객체이다. 더 자세한 내용은 부록을 참고하도록 한다.

KeysView

매핑의 키 뷰를 위한 기반 클래스. MappingView와 Set에서 상속한다.

ItemsView

매핑의 항목 (키, 값) 뷰를 위한 기반 클래스. MappingView와 Set에서 상속한다.

ValuesView

매핑의 값 뷰를 위한 기반 클래스. MappingView와 Set에서 상속한다.

파이썬의 내장 타입들은 이미 적절한 기반 클래스에 등록되어 있다. 이 기반 클래스들을 사용하면 프로그램에서 타입 검사를 더욱 정확하게 할 수 있다. 다음에 나오는 몇 가지 예를 보자.

```
# 순서열의 마지막 항목을 가져온다.
if isinstance(x, collections.Sequence):
    last = x[-1]

# 크기를 알 수 있는 경우에만 객체에 대해 반복을 수행한다.
if isinstance(x, collections.Iterable) and isinstance(x, collections.Sized):
    for item in x:
        statements

# 집합에 새 항목을 추가한다.
if isinstance(x, collections.MutableSet):
    x.add(item)
```

참고

[7장 클래스와 객체지향 프로그래밍](#)

contextlib

contextlib 모듈은 with문과 함께 사용할 컨텍스트 관리자를 생성하기 위한 장식자와 유ти리티 함수를 제공한다.

Contextmanager(func)

생성기 함수 func를 가지고 컨텍스트 관리자를 생성하는 장식자. 이 장식자는 다음에 나오는 것처럼 사용한다.

```
@contextmanager
def foo(args):
    문장들
    try:
        yield 값
    except Exception as e:
        예외 처리(있으면)
    문장들
```

with foo(args) as value문을 만나면 생성기 함수가 주어진 인수로 호출되어 첫 번째 yield문을 만날 때까지 실행된다. yield에 의해 반환되는 값은 변수 value에 저장된다. 이 시점에서 with문의 몸체가 실행된다. 몸체 실행이 끝나면 생성기 함수가 다시 시작된다. with 몸체 안에서 예외가 발생되면 이 예외는 생성기 함수 안에서 발생되고 적절히 처리된다. 예외를 전파하려면 생성기 함수에서 예외를 다시 발생시켜

야 한다. 이 장식자를 사용하는 예는 5장 ‘컨텍스트 관리자’ 절에서 살펴보았다.

nested(mgr1, mgr2, ..., mgrN)

두 개 이상의 컨텍스트 관리자 mgr1, mgr2 등을 단일 연산으로서 호출하는 함수. with문들의 반환 값을 담은 튜플을 반환한다. “with nested(m1,m2) as (x,y): 문장들”은 “with m1 as x: with m2 as y: 문장들”과 동일하다. 안쪽 컨텍스트 관리자에서 예외를 잡으면 예외 정보는 바깥쪽 관리자로 전달되지 않는다.

closing(object)

실행이 with문의 몸체를 벗어나면 자동으로 object.close()를 실행하는 컨텍스트 관리자를 생성한다. with문은 object를 반환한다.

functools

functools 모듈은 고차 함수, 함수형 프로그래밍, 장식자를 생성하는 데 유용하게 사용할 수 있는 함수와 장식자를 담고 있다.

partial(function [, *args [, **kwargs]])

위치 인수 args, 키워드 인수 kwargs, 그리고 호출 시점에 추가로 제공한 위치 인수나 키워드 인수로 함수 function을 호출하는 함수 같은 객체인 partial을 생성한다. 추가 위치 인수는 args의 끝에 추가되고 추가 키워드 인수는 이전에 정의된 값이 있으면 그것을 덮어쓰면서 kwargs에 합쳐진다. partial() 함수는 많은 인수의 값을 고정한 채로 여러 번 함수를 호출할 때 보통 쓰인다. 다음 예를 보자.

```
from functools import partial
mybutton = partial(Button, root, fg="black", bg="white", font="times",
                   size=12)
b1 = mybutton(text="Ok")      # text="Ok"와 위에서 partial()에 제공된 모든
b2 = mybutton(text="Cancel") # 추가 인수를 가지고 Button()를 호출한다.
b3 = mybutton(text="Restart")
```

partial에 의해서 생성된 인스턴스 p는 다음 속성을 갖는다.

항목	설명
p.func	p를 호출할 때 호출되는 함수
p.args	호출할 때 p.func에 전달할 앞쪽 위치 인수들을 담은 튜플. 추가 위치 인수는 이 값 끝에 연결된다.
p.keywords	호출할 때 p.func에 전달할 키워드 인수들을 담은 사전. 추가 키워드 인수는 이 사전에 합쳐진다.

보통 함수 대신 partial 객체를 사용할 경우 주의해야 한다. 보통 함수와 같지 않기 때문이다. 예를 들어, 클래스를 정의할 때 partial()을 사용하면 해당 메서드는 인스턴스 메서드가 아니라 정적 메서드처럼 작동한다.

reduce(function, items [, initial])

반복 가능한 객체 items에 있는 항목에 함수 function을 누적하는 방식으로 적용하고 값 하나를 반환한다. function은 두 인수를 받아야 하며 items에 있는 처음 두 항목에 적용된다. 여기서 나온 결과와 items에 있는 다음 항목이 다시 function에 적용되고 items에 들어 있는 모든 항목이 소진될 때까지 비슷하게 반복한다. initial은 옵션인 시작 값으로 첫 번째 계산에서 사용하거나 items가 비었을 때 사용한다. 이 함수는 파이썬 2의 내장 함수인 reduce() 함수와 같다. 앞으로 호환성을 위해서 이 버전을 사용하도록 하라.

update_wrapper(wrapper, wrapped [, assigned [, updated]])

장식자를 만들 때 유용하게 쓸 수 있는 유틸리티 함수이다. 함수 wrapped에서 래퍼 함수인 wrapper로 속성을 복사하여 래퍼로 둘러싸인 함수가 원래 함수처럼 보이게 만든다. assigned는 복사할 속성 이름들의 튜플이고 기본으로 ('__name__', '__module__', '__doc__')로 설정된다. updated는 래퍼에서 값이 합쳐지기를 원하는 사전인 함수 속성 이름들을 담는 튜플이다. 기본으로 이 튜플은 ('__dict__')이다.

wraps(function [, assigned [, updated]])

자신이 적용될 함수에 update_wrapper()와 같은 일을 하는 장식자이다. assigned와 updated는 같은 의미를 가진다. 이 장식자는 다른 장식자를 만들 때 주로 쓰인다. 다음 예를 보자.

```
from functools import wraps
def debug(func):
    @wraps(func)
    def wrapped(*args,**kwargs):
        print("Calling %s" % func.__name__)
        r = func(*args,**kwargs)
        print("Done calling %s" % func.__name__)
        return wrapped

    @debug
    def add(x,y):
        return x+y
```

참고

6장 함수와 함수형 프로그래밍

heapq

heapq 모듈은 힙(heap)으로 우선 순위 큐를 구현한다. 힙은 간단히 힙 조건에 따라 정렬된 항목들의 리스트이다. 구체적으로는 0에서 시작하는 모든 n에 대해서 $\text{heap}[n] \leq \text{heap}[2 * n + 1]$ 와 $\text{heap}[n] \leq \text{heap}[2 * n + 2]$ 를 만족한다. $\text{heap}[0]$ 은 항상 가장 작은 항목을 담는다.

heapify(x)

리스트 x를 제자리에서 힙으로 변환한다.

heappop(heap)

힙 조건을 유지하면서 heap에서 가장 작은 항목을 반환하고 제거한다. heap이 비어 있으면 IndexError를 발생시킨다.

heappush(heap, item)

힙 조건을 유지하면서 item을 힙에 추가한다.

heappushpop(heap, item)

item을 힙에 추가하고 heap에서 가장 작은 항목을 제거하는 일을 한 번에 한다. heappush()와 heappop()을 따로 호출하는 것보다 더 빠르다.

heareplace(heap, item)

힙에서 가장 작은 아이템을 반환하고 제거한다. 이와 동시에 새로운 항목 item이 추가된다. 이러는 동안 힙 조건이 유지된다. 이 함수는 heappop()과 heappush()를 차례로 호출하는 것보다 더 빠르다. 반환되는 값은 새로운 항목을 추가하기 전에 얻어진다. 따라서 반환되는 값은 item보다 더 클 수 있다. heap이 비어 있으면 IndexError가 발생한다.

merge(s1, s2, ...)

정렬된 반복 가능한 객체 s1, s2 등을 하나의 정렬된 순서열로 합치는 반복자를 생성한다. 이 함수는 입력들을 소모하지 않으며 점진적으로 데이터를 처리하는 반복자를 반환한다.

nlargest(n, iterable [, key])

iterable에 있는 가장 큰 n개의 항목으로 구성된 리스트를 생성한다. 반환되는 리스트에서 가장 큰 항목이 제일 앞에 나온다. key는 옵션인 함수로 하나의 입력 매개변수를 받고 iterable에 있는 각 항목의 비교 키를 계산하는 함수이다.

nsmallest(n, iterable [, key])

iterable에 있는 가장 작은 n개의 항목으로 구성된 리스트를 생성한다. 반환되는 리스트에서 가장 작은 항목이 제일 앞에 나온다. key는 옵션인 키 함수이다.

Note

알고리즘에 관한 책 대부분에 힙 우선 큐의 이론과 구현이 나와 있다.

itertools

itertools 모듈은 데이터에 대해서 다양한 반복을 수행하는 데 사용할 수 있는 효율적인 반복자를 생성하는 함수를 담고 있다. 이 모듈에 있는 모든 함수는 for문이나 기타 생성기, 생성기 표현식 같은 반복자와 관련 있는 함수와 함께 사용할 수 있는 반복자를 반환한다.

chain(iter1, iter2, ..., iterN)

반복자 그룹(iter1, ..., iterN)이 주어질 때 이 함수는 모든 반복자를 연결하는 새로운 반복자를 생성한다. 반환되는 반복자는 iter1에서 모든 항목이 소진될 때까지 항목을 생성한다. 다음으로 iter2에서 항목을 생성된다. 이 과정은 iterN에 있는 모든 항목을 소진할 때까지 계속된다.

chain.from_iterable(iterables)

iterables이 반복 가능한 객체들의 순서열을 생성하는 반복 가능한 객체일 때, 이들을 연결하는 생성자. 이 연산의 결과는 다음 생성기 코드를 실행하는 것과 동일하다.

```
for it in iterables:
    for x in it:
        yield x
```

combinations(iterable, r)

iterable에서 길이 r인 모든 하위 순서열을 반환하는 반복자를 생성한다. 반환되는 하위 순서열에 있는 항목들은 입력 iterable에서 순서와 동일한 순서를 가진다. 예를 들어, iterable이 리스트 [1, 2, 3, 4]라면 combinations([1, 2, 3, 4], 2)에 의해서 생성되는 순서열은 [1, 2], [1, 3], [1, 4], [2, 3], [3, 4]이다.

count([n])

n에서 시작하는 연속적인 정수를 생성하는 반복자를 생성한다. n을 생략하면 0에서 시작한다.(이 반복자는 긴 정수를 지원하지 않는다. 만약 sys.maxint를 넘으면 오버플로우가 나서 -sys.maxint - 1에서 다시 시작한다.)

cycle(iterable)

iterable에 있는 원소들을 계속해서 순환하는 반복자를 생성한다. 내부에서 iterable에 있는 원소들의 복사본이 만들어진다. 이 복사본은 순환하면서 항목들을 반복적으로 반환하는 데 사용된다.

dropwhile(predicate, iterable)

함수 predicate(item)이 True로 평가되는 동안 iterable에서 항목을 버리는 반복자를 생성한다. 일단 predicate가 False를 반환하면 해당 항목과 iterable에 있는 나머지 항목이 생성된다.

groupby(iterable [, key])

iterable에서 생성되는 연속적인 항목을 그룹 짓는 반복자를 생성한다. 그룹을 짓기 위해서 중복된 항목을 찾는다. 예를 들어, iterable에서 동일한 항목이 여러 번 나오면 그룹이 만들어진다. 이 함수를 정렬된 리스트에 적용하면 그룹들은 리스트에 있는 모든 고유한 항목을 나타내게 된다. key는 제공될 경우 각 항목에 적용되는 함수이다. key가 주어지면 항목을 직접 비교하는 것이 아니라 이 함수가 반환하는 값이 연속되는 항목을 비교하는 데 사용된다. 이 함수에서 반환되는 반복자는 (key, group) 튜플들을 생성하는데, key는 그룹의 키 값을, group은 해당 그룹을 구성하는 모든 항목을 생성하는 반복자이다.

ifilter(predicate, iterable)

iterable에서 predicate(item)이 True인 항목만 생성하는 반복자를 만든다. predicate가 None이면 iterable에 있는 모든 항목이 True로 평가되어 반환된다.

`ifilterfalse(predicate, iterable)`

`iterable`에서 `predicate(item)`이 `False`인 항목만 생성하는 반복자를 만든다. `predicate`가 `None`이면 `iterable`에 있는 모든 항목이 `False`로 평가되어 반환된다.

`imap(function, iter1, iter2, ..., iterN)`

`function(i1, i2, .. iN)`들을 생성하는 반복자를 만든다. 여기서 `i1, i2, ..., iN`은 각각 반복자 `iter1, iter2, ..., iterN`에서 얻는다. `function`이 `None`이면 `(i1, i2, ..., iN)` 형태의 튜플이 반환된다. 주어진 반복자 중 하나라도 값을 더 이상 생성하지 않으면 반복이 멈춘다.

`islice(iterable, [start,] stop [, step])`

`iterable[start:stop:step]`이 반환하는 것과 비슷하게 항목을 생성하는 반복자를 만든다. 첫 `start` 개의 항목을 건너뛰고 `stop` 위치에서 반복을 멈춘다. `step`은 항목을 건너뛸 보폭을 말한다. 분할과는 달리 `start, stop`이나 `step` 값으로 음수를 사용할 수 없다. `start`를 생략하면 0에서 반복을 시작한다. `step`을 생략하면 1값이 사용된다.

`izip(iter1, iter2, ... iterN)`

`(i1, i2, ..., iN)`들을 생성하는 반복자를 만든다. 여기서 `i1, i2, ..., iN`은 각각 반복자 `iter1, iter2, ..., iterN`에서 얻는다. 주어진 반복자 중 하나라도 값을 더 이상 생성하지 않으면 반복을 멈춘다. 이 함수는 내장 `zip()` 함수와 동일한 값을 생성한다.

`izip_longest(iter1, iter2, ..., iterN [, fillvalue=None])`

반복자 `iter1, iter2` 등을 모두 소진할 때까지 반복이 이어진다는 것을 제외하고는 `izip()`과 동일하다. `fillvalue` 키워드 인수로 값을 주지 않으면 다 소진된 반복자의 값을 채우는 데 `None`이 사용된다.

`permutations(iterable [, r])`

`iterable`에서 길이 `r`인 모든 순열을 반환하는 반복자를 생성한다. `r`을 생략하면 `iterable`에 있는 항목 개수와 동일한 길이를 가지는 순열이 반환된다.

`product(iter1, iter2, ... iterN, [repeat=1])`

`iter1, iter2` 등에 있는 항목들의 대카르트곱(Cartesian product)을 나타내는 튜플을 생성하는 반복자를 만든다. `repeat`는 키워드 인수로서 생성된 순서열을 몇 번 반복할지를 지정한다.

repeat(object [, times])

object를 계속해서 생성하는 반복자를 생성한다. times은 반복 횟수를 나타낸다. times가 주어지지 않으면 object를 끝없이 반복한다.

starmap(func [, iterable])

func(*item)들을 생성하는 반복자를 만든다. 여기서 item은 iterable에서 얻는다. 이런 식으로 함수를 호출할 수 있는 항목을 생성하는 iterable에 대해서만 제대로 작동한다.

takewhile(predicate [, iterable])

predicate(item)이 True인 한 iterable에서 항목을 생성하는 반복자를 만든다. predicate가 False로 평가되는 즉시 반복을 멈춘다.

tee(iterable [, n])

iterable에서 n개의 독립적인 반복자를 생성한다. 생성된 반복자들은 n개 항목 튜플로 반환된다. n은 기본으로 2다. 이 함수는 아무 반복 가능한 객체에나 적용할 수 있다. 원래 반복자를 복제하기 위해서, 생성되는 항목들을 캐싱해 두었다가 모든 새로 생성된 반복자에서 사용한다. tee()를 호출한 다음에 원래 반복자를 사용하지 않도록 조심해야 한다. 잘못하면 캐싱 메커니즘이 제대로 작동하지 않는다.

예

다음 예들은 itertools 모듈에 있는 함수들을 사용하는 방법을 보여준다.

```
from itertools import *
# 끝없이 0,1,...,10,9,8,...,1를 반복한다.
for i in cycle(chain(range(10), range(10,0,-1))):
    print i

# a에 있는 고유한 항목들의 리스트를 생성한다.
a = [1,4,5,4,9,1,2,3,4,5,1]
a.sort( )
b = [k for k,g in groupby(a)]    # b = [1,2,3,4,5,9]

# x와 y의 값을 조합할 수 있는 모든 쌍에 대해서 반복한다.
x = [1,2,3,4,5]
y = [10,11,12]
for r in product(x,y):
    print(r)
# (1,10),(1,11),(1,12), ... (5,10),(5,11),(5,12)를 출력한다.
```

operator

operator 모듈은 3장에서 설명한 내장 연산자와 인터프리터의 특수 메서드에 접근하는 데 사용할 수 있는 함수를 제공한다. 예를 들어, `add(3, 4)`는 $3 + 4$ 와 같다. 제 자리 연산이 지원되는 연산에 대해서는 `x += y`와 동일한 `iadd(x, y)` 같은 함수를 사용할 수 있다. 다음 목록은 operator 모듈에 정의된 함수 목록과 이들이 어떤 연산자에 매핑되는지를 보여준다.

함수	설명
<code>add(a, b)</code>	숫자에 대해서 $a + b$ 를 반환한다.
<code>sub(a, b)</code>	$a - b$ 를 반환한다.
<code>mul(a, b)</code>	숫자에 대해서 $a * b$ 를 반환한다.
<code>div(a, b)</code>	a / b (옛 나누기)를 반환한다.
<code>floordiv(a, b)</code>	$a // b$ 를 반환한다.
<code>truediv(a, b)</code>	a / b (새 나누기)를 반환한다.
<code>mod(a, b)</code>	$a \% b$ 를 반환한다.
<code>neg(a)</code>	$-a$ 를 반환한다.
<code>pos(a)</code>	$+a$ 를 반환한다.
<code>abs(a)</code>	a 의 절대값을 반환한다.
<code>inv(a), invert(a)</code>	a 의 비트 역수를 반환한다.
<code>shift(a, b)</code>	$a \ll b$ 를 반환한다.
<code>rshift(a, b)</code>	$a \gg b$ 를 반환한다.
<code>and_(a, b)</code>	$a \& b$ (비트 AND)를 반환한다.
<code>or_(a, b)</code>	$a \mid b$ (비트 OR)를 반환한다.
<code>xor(a, b)</code>	$a ^ b$ (비트 XOR)를 반환한다.
<code>not_(a)</code>	<code>not a</code> 를 반환한다.
<code>lt(a, b)</code>	$a < b$ 를 반환한다.
<code>le(a, b)</code>	$a \leq b$ 를 반환한다.
<code>eq(a, b)</code>	$a == b$ 를 반환한다.
<code>ne(a, b)</code>	$a != b$ 를 반환한다.
<code>gt(a, b)</code>	$a > b$ 를 반환한다.
<code>ge(a, b)</code>	$a \geq b$ 를 반환한다.
<code>truth(a)</code>	a 가 참이면 <code>True</code> 를 아니면 <code>False</code> 를 반환한다.
<code>concat(a, b)</code>	순서열에 대해서 $a + b$ 를 반환한다.
<code>repeat(a, b)</code>	순서열 a 와 정수 b 에 대해서 $a * b$ 를 반환한다.
<code>contains(a, b)</code>	b in a 의 결과를 반환한다.
<code>countOf(a, b)</code>	b 가 a 에서 나타난 횟수를 반환한다.
<code>indexOf(a, b)</code>	b 가 a 에서 나타낸 첫 번째 색인을 반환한다.
<code>getitem(a, b)</code>	$a[b]$ 를 반환한다.
<code>setitem(a, b, c)</code>	$a[b] = c$
<code>delitem(a, b)</code>	<code>del a[b]</code>

<code>getslice(a, b, c)</code>	a[b:c]를 반환한다.
<code>setslice(a, b, c, v)</code>	a[b:c] = v로 설정한다.
<code>delslice(a, b, c)</code>	del a[b:c]
<code>is_(a, b)</code>	a is b
<code>is_not(a, b)</code>	a is not b

처음에는 왜 간단한 문법을 두고 이 함수를 사용하는지 이해가 잘 안 갈 것이다. 이 함수들은 역호출 함수가 필요한 코드라든지 lambda로 익명 함수를 정의해야 하는 상황에서 유용하게 쓰인다. `functools.reduce()` 함수의 시간을 측정하는 벤치마크 코드를 보자.

```
>>> from timeit import timeit
>>> timeit("reduce(operator.add,a)","import operator; a =
range(100)")
12.055853843688965
>>> timeit("reduce(lambda x,y: x+y,a)","import operator; a =
range(100)")
25.012306928634644
>>>
```

이 예에서 `operator.add`를 역호출 함수로서 사용하는 것인 `lambda x,y: x+y`를 사용하는 버전보다 두 배 정도 빠른 것을 볼 수 있다.

`operator` 모듈에는 속성 접근, 항목 검색, 메서드 호출을 감싸는 다음 함수들도 있다.

`attrgetter(name [, name2 [, ... [, nameN]]])`

호출 가능한 객체 `f`를 생성한다. `f(obj)`를 호출하면 `obj.name`을 반환한다. 인수가 두 개 이상 주어지면 결과들의 튜플을 반환한다. 예를 들어, `attrgetter('name', 'shares')`는 호출할 경우 (`obj.name`, `obj.shares`)를 반환한다. `name`은 추가로 점 검색을 담을 수 있다. 예를 들어, `nameo "address.hostname"`이면 `f(obj)`는 `obj.address.hostname`을 반환한다.

`itemgetter(item [, item2 [, ... [, itemN]]])`

호출 가능한 객체 `f`를 생성한다. `f(obj)`를 호출하면 `obj[item]`을 반환한다. 인수가 두 개 이상 주어지면 `f(obj)`를 호출하면 (`obj[item]`, `obj[item2]`, ..., `obj[itemN]`) 튜플을 반환한다.

`methodcaller(name [, *args [, **kwargs]])`

호출 가능한 객체 `f`를 생성한다. `f(obj)`를 호출하면 `obj.name(*args, **kwargs)`를

반환한다.

이 함수들은 역호출 함수가 필요한 연산의 성능을 최적화하는 데 사용할 수 있다. 특히 정렬 같이 자주 쓰이는 데이터 처리 연산에 유용하다. 예를 들어, 튜플 리스트를 두 번째 열에 대해서 정렬하려고 할 때 `sorted(rows, key=lambda r: r[2])`로 하거나 `sorted(rows, key=itemgetter(2))`로 할 수 있다. 두 번째 버전이 속도가 느린 `lambda`를 사용하지 않기 때문에 훨씬 빠르다.

16장

P y t h o n E s s e n t i a l R e f e r e n c e

문자열과 텍스트 처리

이 장에서는 기초적인 문자열과 텍스트 처리를 위해서 사용되는 파이썬 모듈을 다룬다. 텍스트 처리, 정규 표현식 패턴 매칭, 텍스트 포맷 지정 같은 많이 사용되는 문자열 연산을 중점적으로 다룬다.

codecs

codecs 모듈은 유니코드 텍스트 I/O에 사용되는 여러 문자 인코딩을 처리하는 데 사용한다. 이 모듈은 새로운 문자 인코딩을 정의하거나 UTF-8, UTF-16 같은 다양한 기존 인코딩에 기반하여 문자 데이터를 처리하는 데 모두 사용된다. 단순히 기존 인코딩 중 하나를 사용하는 일이 훨씬 많기 때문에 여기서는 기존 인코딩에 관해서만 다룬다. 새로운 인코딩을 정의하는 방법을 알고 싶으면 온라인 문서를 찾아보기 바란다.

저수준 codecs 인터페이스

각 문자 인코딩은 ‘utf-8’이나 ‘big5’ 같이 보통 사용되는 이름이 있다. 다음 함수는 이를 검색에 사용한다.

`lookup(encoding)`

코덱 레지스트리에서 코덱을 찾는다. encoding은 ‘utf-8’ 같은 문자열이다. 요청한 인코딩에 대한 정보가 아무것도 없으면 LookupError 예외가 발생한다. 아니면 CodecInfo의 인스턴스 `c`가 반환된다.

CodecInfo 인스턴스 c는 다음 메서드들을 갖는다.

c.encode(s [, errors])

유니코드 문자열 s를 인코딩하여 튜플 (bytes, length_consumed)을 반환하는 상태 유지 않는(stateless) 인코딩 함수. bytes는 인코딩된 데이터를 담은 8비트 문자열 또는 바이트 배열이다. length_consumed는 s에서 인코딩된 문자 개수를 나타낸다. errors는 에러 처리 정책을 나타내고 기본으로 ‘strict’이다.

c.decode(bytes [, errors])

바이트 문자열 bytes를 디코딩하여 튜플 (s, length_consumed)을 반환하는 상태 유지 않는 인코딩 함수. s는 유니코드 문자열이고 length_consumed는 디코딩 과정에서 소모한 bytes 안에 있는 바이트 수이다. errors는 에러 처리 정책이고 기본으로 ‘strict’이다.

c.streamreader(bytestream [, errors])

인코딩된 데이터를 읽는 데 사용할 수 있는 StreamReader 인스턴스를 반환한다. bytestream은 이진 모드로 열린 파일 같은 객체이다. errors는 에러 처리 정책을 나타내고 기본으로 ‘strict’이다. StreamReader의 인스턴스 r은 다음 저수준 I/O 연산을 지원한다.

메서드

r.read([size [, chars [, firstline]]])

설명

디코딩된 텍스트에서 최대 chars 개의 문자를 반환한다. size는 저수준 바이트 스트림에서 읽을 최대 바이트 수를 나타내며 내부 버퍼링을 제어한다. firstline은 플래그로서 설정될 경우 파일에서 나중에 디코딩 에러가 발생하더라도 첫 번째 줄을 반환하게 한다.

r.readline([size [, keepends]])

디코딩된 텍스트에서 한 줄을 반환한다. keepends는 플래그로서 줄 종료 문자를 제거할지 말지를 제어한다(기본 값은 True).

r.readlines([size [, keepends]])

모든 줄을 리스트로 읽는다.

r.reset()

내부 버퍼와 상태 정보를 재설정한다.

c.streamwriter(bytestream [, errors])

인코딩된 데이터를 쓰는 데 사용할 수 있는 StreamWriter 인스턴스를 반환한다.

bytestream은 이진 모드에서 열린 파일 같은 객체이다. errors는 에러 처리 정책으로 기본으로 ‘strict’이다. StreamWriter의 인스턴스 w는 다음 저수준 I/O 연산을 지원 한다.

메서드	설명
w.write(s)	문자열 s의 인코딩된 표현을 쓴다.
w.writelines(lines)	lines에 있는 문자열 리스트를 파일에 쓴다.
w.reset()	내부 버퍼와 상태 정보를 재설정한다.

c.incrementalencoder([errors])

여러 단계에 걸쳐서 문자열들을 인코딩하는 데 사용할 수 있는 IncrementalEncoder 인스턴스를 반환한다. errors는 기본으로 ‘strict’이다. IncrementalEncoder의 인스턴스 e는 다음 메서드들을 갖는다.

메서드	설명
e.encode(s [, final])	문자열 s의 인코딩된 표현을 바이트 문자열로 반환한다. final은 마지막 encode() 호출에서 True로 설정해야 하는 플래그이다.
e.reset()	내부 버퍼와 상태 정보를 재설정한다.

c.incrementaldecoder([errors])

여러 단계에 걸쳐서 문자열들을 디코딩하는 데 사용할 수 있는 IncrementalDecoder 인스턴스를 반환한다. errors는 기본으로 ‘strict’이다. IncrementalDecoder의 인스턴스 d는 다음 메서드들을 갖는다.

메서드	설명
d.decode(bytes [, final])	bytes에 있는 인코딩된 바이트에서 디코딩된 문자열을 반환한다. final은 마지막 decode() 호출에서 True로 설정해야 하는 플래그이다.
d.reset()	내부 버퍼와 상태 정보를 재설정한다.

I/O 관련 함수

codecs 모듈은 인코딩된 텍스트와 관련된 I/O를 간단하게 만들어주는 고수준 함수들을 제공한다. 보통은 앞에서 설명한 저수준 코덱 인터페이스 대신 이 함수들을 사용한다.

open(filename, mode[, encoding[, errors[, buffering]]])

주어진 mode에서 filename을 열고 encoding으로 지정한 인코딩에 따라서 자

동으로 데이터 인코딩과 디코딩을 수행한다. errors는 ‘strict’, ‘ignore’, ‘replace’, ‘backslashreplace’, ‘xmlcharrefreplace’ 중 하나이다. 기본은 ‘strict’이다. buffering은 내장 open() 함수에 있는 것과 동일하다. mode로 지정한 모드와 관계 없이 내부 파일은 항상 이진 모드로 열린다. 파이썬 3에서는 codecs.open() 대신 내장 open() 함수를 사용해도 된다.

EncodedFile(file, inputenc[, outputenc [, errors]])

기존 파일 객체 file을 감싸는 인코딩 래퍼 클래스. 파일에 쓰여지는 데이터는 먼저 입력 인코딩 inputenc에 따라서 해석되고 그 다음으로 출력 인코딩인 outputenc에 따라서 파일에 써진다. 파일에서 읽은 데이터는 inputenc에 따라서 디코딩된다. outputenc를 생략하면 inputenc 값이 사용된다. errors는 open()에서와 의미가 같고 기본으로 ‘strict’이다.

iterencode(iterable, encoding [, errors])

주어진 encoding에 따라서 iterable에 있는 모든 문자열을 점진적으로 인코딩하는 생성기 함수이다. errors는 기본으로 ‘strict’이다.

iterdecode(iterable, encoding [, errors])

주어진 encoding에 따라서 iterable에 있는 모든 문자열을 점진적으로 디코딩하는 생성기 함수이다. errors는 기본으로 ‘strict’이다.

유용한 상수

codecs에는 내부 인코딩에 대해서 아무것도 모를 때 파일을 해석하는 데 도움을 주기 위해서 다음에 나오는 바이트 순서 표시들을 정의한다. 이 바이트 순서 표시들은 보통 파일 시작 부분에 쓰여져서 문자 인코딩을 나타내며 적절한 코덱을 고르는 데 사용된다.

상수	설명
BOM	기계에 특화된 바이트 순서 표시(BOM_BE 또는 BOM_LE)
BOM_BE	빅 엔디안 바이트 순서 표시(‘\xfe\xff’)
BOM_LE	리틀 엔디안 바이트 순서 표시(‘\xff\xfe’)
BOM_UTF8	UTF-8 표시(‘\xef\xbb\xbf’)
BOM_UTF16_BE	16비트 UTF-16 빅 엔디안 표시(‘\xfe\xff’)
BOM_UTF16_LE	16비트 UTF-16 리틀 엔디안 표시(‘\xff\xfe’)
BOM_UTF32_BE	32비트 UTF-32 빅 엔디안 표시(‘\x00\x00\xfe\xff’)
BOM_UTF32_LE	32비트 UTF-32 리틀 엔디안 표시(‘\xff\xfe\x00\x00’)

표준 인코딩

다음은 가장 자주 사용되는 문자 인코딩 목록을 보여준다. 인코딩 이름은 `open()`이나 `lookup()` 같은 함수에서 인코딩을 지정하는 데 쓰이는 이름이다. 전체 인코딩 목록은 `codecs` 모듈의 온라인 문서에 나와 있다(<http://docs.python.org/library/codecs>).

코덱 이름	설명
<code>ascii</code>	7비트 ASCII 문자
<code>cp437</code>	MS-DOS의 확장 ASCII 문자 집합
<code>cp1252</code>	윈도의 확장 ASCII 문자 집합
<code>latin-1, iso-8859-1</code>	라틴 문자로 확장된 ASCII
<code>utf-16</code>	UTF-16
<code>utf-16-be</code>	UTF-16 빅 엔디안
<code>utf-16-le</code>	UTF-16 리틀 엔디안
<code>utf-32</code>	UTF-32
<code>utf-32-be</code>	UTF-32 빅 엔디안
<code>utf-32-le</code>	UTF-32 리틀 엔디안
<code>utf-8</code>	UTF-8

Note

- `codecs` 모듈의 사용법은 9장에 더 설명되어 있다.
- 새 문자 인코딩을 만드는 방법은 온라인 문서를 참고한다.
- `encode()`와 `decode()`에 입력을 넘길 때 주의해야 한다. `encode()`에는 유니코드 문자열을 사용해야 하고 `decode()`에는 바이트 문자열을 사용해야 한다. 파일 2에서는 이 부분에 일관성이 있는 편이고 파일 3에서는 두 문자열을 엄격히 구분한다. 예를 들어, 파일 2에는 바이트 문자열을 바이트 문자열로 매핑하는 코덱이 있다("bz2" 같은 코덱). 파일 3에는 이런 것이 없고 호환성을 위해서라면 사용하지 않는 것이 좋다.

re

`re` 모듈은 문자열에 정규 표현 패턴 매칭과 치환을 수행하는 데 사용한다. 정규 표현식 패턴은 텍스트와 특수문자 순서열이 섞인 문자열로 지정한다. 패턴은 보통 특수 문자와 역슬래시를 많이 사용하기 때문에 보통 `r'(P<int>\d+)\.(\\d*)'`처럼 무가공 문자열로 작성한다. 이 절에서는 앞으로 정규 표현식 패턴을 무가공 문자열 문법으로 쓴다.

패턴 문법

정규 표현식 패턴에서는 다음에 나오는 특수 문자 순서열들이 인식된다.

문자(들)	설명
text	상수 문자열 text에 매칭
.	줄바꿈 문자를 제외한 모든 문자
^	문자열 시작에 매칭
\$	문자열 끝에 매칭
*	앞에 나온 표현식의 영 또는 하나 이상의 반복에 매칭. 가능한 한 긴 반복에 매칭.
+	앞에 나온 표현식의 하나 이상의 반복에 매칭. 가능한 한 긴 반복에 매칭.
?	앞에 나온 표현식의 영 또는 하나의 반복에 매칭.
*?	앞에 나온 표현식의 영 또는 하나 이상의 반복에 매칭. 가능한 한 짧은 반복에 매칭.
+?	앞에 나온 표현식의 하나 이상의 반복에 매칭. 가능한 한 짧은 반복에 매칭.
??	앞에 나온 표현식의 영 또는 하나의 반복에 매칭. 가능한 한 짧은 반복에 매칭.
{m}	앞에 나온 표현식의 정확히 m번 반복에 매칭.
{m, n}	앞에 나온 표현식의 m번에서 n번 반복에 매칭. 가능한 한 긴 반복에 매칭. m을 생략하면 기본으로 0, n을 생략하면 기본으로 무한대.
{m, n}?	앞에 나온 표현식의 m번에서 n번 반복에 매칭. 가능한 한 짧은 반복에 매칭.
[...]	r'[abcdef]' 또는 '[a-zA-Z]' 같은 문자 집합에 매칭. 집합 안에서 * 같은 특수 문자는 활성화되지 않음.
[^...]	r'[^0-9]'처럼 집합에 없는 문자에 매칭
A B	A와 B가 모두 정규 표현식일 때 A 또는 B에 매칭
(...)	괄호 안에 있는 정규 표현식에 한 그룹으로 매칭되고 매칭된 부분 문자열을 저장한다. 그룹 내용은 매칭 과정에서 얻은 MatchObject 객체의 group() 메서드로 얻을 수 있다.
(?aiLmsux)	문자 “a”, “i”, “L”, “m”, “s”, “u”, “x”를 각각 re.compile()에 주어지는 re.A, re.I, re.L, re.M, re.S, re.U, re.X 플래그 설정으로 해석한다. “a”는 파이썬 3에만 있다.
(?:....)	괄호 안에 있는 정규 표현식에 매칭시키지만 매칭된 부분 문자열을 버린다.
(?P<name>....)	괄호 안에 있는 정규 표현식에 매칭시키고 이름 있는 그룹을 만든다. 그룹 이름은 유효한 파이썬 식별자이어야 한다.
(?P=name)	앞선 이름 있는 그룹에 매칭된 것과 동일한 텍스트에 매칭.
(?#...)	주석. 괄호 안에 있는 내용은 무시된다.
(=?...)	괄호 안에 있는 패턴이 따라 나올 때만 앞선 표현식을 매칭.

	예를 들어, r'Hello (?=World)'는 뒤에 'World'가 나올 때만 'Hello'에 매칭.
(?!...)	괄호 안에 있는 패턴이 따라 나오지 않을 때만 앞선 표현식에 매칭. 예를 들어, r'Hello (?!=World)'는 뒤에 'World'가 없을 때만 'Hello'에 매칭.
(?<=...)	괄호 안에 있는 패턴에 매칭 되는 텍스트가 앞에 나올 때만 뒤에 나오는 표현식에 매칭. 예를 들어, r'(?<=abc)def'는 앞에 'abc'가 나올 경우에만 'def'에 매칭.
(?<!...)	괄호 안에 있는 패턴에 매칭되는 텍스트가 앞에 나오지 않을 때만 뒤에 나오는 표현식에 매칭. 예를 들어, r'(?<!abc)def'는 앞에 'abc'가 나오지 않을 경우에만 'def'에 매칭.
(?(id name)ypat npat)	id나 name이라는 이름을 갖는 정규 표현식 그룹이 있는지를 검사한다. 있다면 정규 표현식 ypat이 매칭된다. 없다면 옵션인 표현식 npat가 매칭된다. 예를 들어, r'(Hello)?(1) World!Howdy)'는 문자열 'Hello World' 또는 'Howdy'에 매칭된다.

'\n'이나 '\t' 같은 탈출 순서열은 정규 표현식에서 표준 문자로 인식된다(예를 들어, r'\n+' 는 하나 이상의 줄바꿈 문자에 매칭된다. 정규 표현식에서 특수한 뜻을 갖는 상수 기호는 앞에 역슬래시를 붙여서 쓸 수 있다. 예를 들어, r'*'는 문자 '*'에 매칭된다. 몇몇 역슬래시 순서열은 특수한 문자 집합에 대응한다.

문자(들)	설명
\숫자	이전 그룹 숫자에 매칭된 텍스트에 매칭. 그룹들은 왼쪽부터 1에서 99까지 번호가 매겨진다.
\A	문자열의 시작 부분에만 매칭.
\b	단어의 시작 또는 끝에 있는 빈 문자열에 매칭. 여기서 단어란 알파벳과 숫자의 순서열로서 공백 또는 알파벳이나 숫자가 아닌 문자로 끝나는 것을 말한다.
\B	단어의 시작이나 끝에 있지 않은 빈 문자열에 매칭.
\d	십진수에 매칭. r'[0-9]'와 동일.
\D	숫자 아닌 문자에 매칭. r'[^0-9]'와 동일.
\s	공백 문자에 매칭. r'[\t\n\r\f\v]'와 동일.
\S	공백이 아닌 문자에 매칭. r'[^ \t\n\r\f\v]'와 동일.
\w	알파벳이나 숫자에 매칭.
\W	\w가 정의하는 집합에 속하지 않는 문자에 매칭.
\Z	문자열 끝에만 매칭.
\\"	상수 역슬래시에 매칭.

특수문자 \d, \D, \s, \S, \w, \W는 유니코드 문자열을 매칭할 때는 다르게 해석된다. 이 경우 앞에서 설명한 특성을 갖는 모든 유니코드 문자에 매칭된다. 예를 들어, \d는 유럽, 아라비아, 인도 숫자 등으로 인식되는데 이들은 서로 다른 위치에 있는

유니코드 문자들이다.

함수

다음 함수들은 패턴 매칭과 치환을 수행하는 데 사용한다.

`compile(pattern [, flags])`

정규 표현식 패턴 문자열을 정규 표현식 객체로 컴파일한다. 이 객체는 뒤에 나오는 모든 함수의 패턴 인수로서 전달될 수 있다. 이 객체는 곧 설명할 몇 가지 메서드를 갖는다. flags는 다음에 나오는 것들을 비트 OR한 것이다.

플래그	설명
A 또는 ASCII	8비트 ASCII 전용 매칭(파이썬 3에만 있다).
I 또는 IGNORECASE	대소문자를 구별하지 않는 매칭을 수행한다.
L 또는 LOCALE	\w, \W, \b, \B에 대해서 로케일 설정을 사용한다.
M 또는 MULTILINE	^와 \$가 전체 문자열의 시작과 끝뿐만 아니라 각 줄에 적용되게 한다(보통 ^와 \$는 전체 문자열의 시작과 끝에만 적용된다).
S 또는 DOTALL	점(.)문자를 줄바꿈 문자를 포함한 모든 문자에 매칭한다.
U 또는 UNICODE	\w, \W, \b, \B에 대해서 유니코드 문자 속성 데이터베이스의 정보를 사용한다(파이썬 2에만 있다. 파이썬 3는 유니코드를 기본으로 사용한다).
X 또는 VERBOSE	패턴 문자열에서 털출 안 된 공백과 주석을 무시한다.

`escape(string)`

알파벳과 숫자가 아닌 문자에 모두 역슬래시를 붙인 문자열을 반환한다.

`findall(pattern, string [, flag])`

string에서 겹치지 않는 pattern에 대한 모든 매칭들의 리스트를 반환한다. 빈 매칭도 포함된다. 패턴에 그룹이 하나 이상 있으면 그룹에 매칭되는 텍스트 리스트가 반환된다. 그룹이 둘 이상 있으면 반환되는 리스트에 있는 각 항목은 그룹 텍스트들을 담은 튜플이다. flags는 compile()의 것과 같다.

`finditer(pattern, string [, flags])`

findall()과 같지만 반복자 객체를 반환한다. 반복자는 MatchObject 타입의 항목들을 반환한다.

`match(pattern, string [, flags])`

string 시작 부분에서 영 또는 하나 이상의 문자들이 pattern에 매칭되는지 여부를

검사한다. 성공하면 MatchObject 객체를 반환하고 아니면 None을 반환한다. flags는 compile()의 것과 같다.

search(pattern, string [, flags])

string에서 pattern의 첫 번째 매칭을 찾는다. flags는 compile()의 것과 같다. 성공하면 MatchObject 객체를 반환하고 아무것도 매칭되지 않으면 None을 반환한다.

split(pattern, string [, maxsplit = 0])

string을 pattern이 나타나는 곳들을 기준으로 분할한다. 패턴에 그룹이 있으면 그룹에 매칭되는 텍스트도 담은 문자열 목록을 반환한다. maxsplit은 최대 분할 횟수를 나타낸다. 기본으로 가능한 모든 분할이 수행된다.

sub(pattern, repl, string [, count = 0])

repl를 치환에 사용해서 string에서 왼쪽부터 pattern이 겹치지 않게 출현하는 부분들을 치환한다. repl는 문자열이거나 함수일 수 있다. 함수일 경우 MatchObject를 받아서 치환 문자열을 반환해야 한다. repl가 문자열이면 '\6' 같은 역참조가 패턴에 있는 그룹을 가리키는 데 사용된다. '\g{name}'은 이름 있는 그룹을 가리키는 데 사용한다. count는 최대 치환 횟수를 나타낸다. 기본으로 모든 출현이 교체된다. 이 함수와 다음 함수는 compile()처럼 flags는 받지는 않지만 앞에서 설명한 (?iLmsux) 표기법을 사용하여 동일한 효과를 얻을 수 있다.

subn(pattern, repl, string [, count = 0])

sub()와 동일하지만 새 문자열과 치환 횟수를 담은 튜플을 반환한다.

정규 표현식 객체

compile() 함수가 생성하는 컴파일된 정규 표현식 객체 r은 다음 메서드와 속성들을 갖는다.

r.flags

정규 표현식이 컴파일될 때 사용된 flags 인수. 지정되지 않은 경우 0.

r.groupindex

r'(?P{id})'로 정의한 기호 그룹 이름을 그룹 번호로 매핑하는 사전.

r.pattern

정규 표현식 객체를 컴파일할 때 사용한 패턴 문자열.

r.findall(string [, pos [, endpos]])

findall() 함수와 동일하다. pos와 endpos는 검색의 시작과 끝 위치를 나타낸다.

r.finditer(string [, pos [, endpos]])

finditer() 함수와 동일하다. pos와 endpos는 검색의 시작과 끝 위치를 나타낸다.

r.match(string [, pos] [, endpos])

string의 시작 부분에서 영 또는 하나 이상의 문자들이 매칭되는지를 검사한다. pos와 endpos는 string에서 검색을 수행할 범위를 나타낸다. 매칭이 있으면 MatchObject 객체를 반환하고 아니면 None을 반환한다.

r.search(string [, pos] [, endpos])

string에서 매칭이 있는지 검사한다. pos와 endpos는 string에서 검색을 수행할 범위를 나타낸다. 매칭이 있으면 MatchObject 객체를 반환하고 아니면 None을 반환한다.

r.split(string [, maxsplit = 0])

split() 함수와 동일.

r.sub(repl, string [, count = 0])

sub() 함수와 동일.

r.subn(repl, string [, count = 0])

subn() 함수와 동일.

Match Object

search()와 match()가 반환하는 MatchObject 인스턴스는 그룹 내용과 매칭이 어디서 일어났는지에 대한 위치 데이터를 담는다. MatchObject 객체 m은 다음 메서드와 속성을 갖는다.

m.expand(template)

문자열 template에 정규 표현식 역슬래시 치환을 수행한 결과를 담은 문자열을 반환한다. “\1”이나 “\2” 같은 숫자 역참조와 “\g<n>”과 “\g<name>” 같은 이름 역

참조는 해당 그룹의 내용으로 치환된다. 역참조들은 `r'\1'`이나 '`\1`'처럼 미가공 문자열 또는 상수 역슬래시 문자로 써야 한다.

`m.group([group1, group2, ...])`

매칭에서 하나 또는 그 이상의 하위 그룹을 반환한다. 인수로 그룹 번호나 이름을 쓸 수 있다. 그룹 이름을 주지 않으면 전체 매칭 결과가 반환된다. 그룹을 하나만 주면 해당 그룹에 매칭되는 텍스트를 담은 문자열이 반환된다. 나머지 경우에는 요청한 그룹들에 매칭된 텍스트들을 담은 튜플이 반환된다. 유효하지 않은 그룹 번호나 이름을 주면 `IndexError` 예외가 발생한다.

`m.groups([default])`

패턴에 있는 모든 그룹에 대한 매칭 텍스트들을 담은 튜플을 반환한다. `default`는 매칭에 참여하지 않은 그룹에 대해 반환할 값을 나타낸다(기본으로 `None`이다).

`m.groupdict([default])`

매칭의 모든 이름 있는 그룹들을 담은 사전을 반환한다. `default`는 매칭에 참여하지 않은 그룹에 대해 반환할 값을 나타낸다(기본으로 `None`이다).

`m.start([group]) m.end([group])`

이 두 메서드는 그룹이 매칭된 부분 문자열의 시작과 끝을 나타내는 색인들을 반환한다. `group`을 생략하면 매칭된 부분 문자열 전체가 사용된다. 그룹이 존재하지만 매칭에는 참여하지 않았을 경우 `None`을 반환한다.

`m.span([group])`

2 개 항목을 가진 튜플(`m.start(group), m.end(group)`)을 반환한다. `group`이 매칭에 관여하지 않았으면 (`None, None`)을 반환한다. 그룹을 생략하면 매칭된 부분 문자열 전체가 사용된다.

`m.pos`

`search()`나 `match()` 함수에 전달된 `pos`의 값.

`m.endpos`

`search()`나 `match()` 함수에 전달된 `endpos` 값.

`m.lastindex`

매칭된 마지막 그룹의 숫자 색인. 매칭된 그룹이 없으면 `None`.

m.lastgroup

매칭된 마지막 이름 있는 그룹의 이름. 이름 있는 그룹이 매칭되지 않았거나 패턴에 없을 경우 None.

m.re

match()나 search() 메서드가 이 MatchObject 인스턴스를 만든 정규 표현식 객체.

m.string

match()나 search()에 전달된 문자열.

예

다음 예는 re 모듈을 사용해서 문자열에서 텍스트 패턴을 검색하고 데이터를 추출하고 치환하는 방법을 보여준다.

```
import re
text = "Guido will be out of the office from 12/15/2012 - 1/3/2013."
datepat = re.compile('(\d+)/(\d+)/(\d+)')

# 날짜를 나타내는 정규 표현식 패턴
for m in datepat.finditer(text):
    print(m.group( ))

# 날짜를 모두 찾아서 모두 출력한다.
# monthnames = [None,'Jan','Feb','Mar','Apr','May','Jun',
#                 'Jul','Aug','Sep','Oct','Nov','Dec']
for m in datepat.finditer(text):
    print ("%s %s, %s" % (monthnames[int(m.group(1)),
                                    m.group(2), m.group(3))])

# 유럽 형식(일/월/년)인 필드들로 구성되는 모든 날짜를 교체한다.
def fix_date(m):
    return "%s/%s/%s" % (m.group(2),m.group(1),m.group(3))
newtext = datepat.sub(fix_date, text)

# 다른 방식
newtext = datepat.sub(r'\2/\1/\3', text)
```

Note

- 정규 표현식의 이론과 구현에 관한 자세한 내용은 컴파일러 구축을 다루는 책에서 찾아 볼 수 있다. 제프리 프라이들(Jeffrey Friedl)이 쓴 〈정규 표현식 정복(Mastering Regular Expressions)〉(O'Reilly & Associates, 1997)이 도움이 될 것이다.
- re 모듈을 사용할 때 가장 어려운 부분은 정규 표현식 패턴을 작성하는 부분이다. 패턴 작성에 관해서는 Kodos 같은 도구를 사용하는 것을 고려해보라(<http://kodos.sourceforge.net>).

string

string 모듈은 문자열을 다룰 때 유용하게 쓸 수 있는 몇 가지 상수와 함수를 담고 있다. 이 모듈은 새로운 문자열 포맷 지정자를 구현하는 클래스들도 담고 있다.

상수

다음에 나오는 상수들은 다양한 문자열 처리 연산을 수행할 때 유용하게 쓸 수 있는 문자 집합들을 정의한다.

상수	설명
<code>ascii_letters</code>	모든 대소문자 ASCII 문자들을 담은 문자열
<code>ascii_lowercase</code>	문자열 'abcdefghijklmnopqrstuvwxyz'.
<code>ascii_uppercase</code>	문자열 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
<code>digits</code>	문자열 '0123456789'.
<code>hexdigits</code>	문자열 '0123456789abcdefABCDEF'.
<code>letters</code>	lowercase와 uppercase를 연결한 것
<code>lowercase</code>	현재 로케일 설정에 따른 모든 소문자들을 담은 문자열
<code>octdigits</code>	문자열 '01234567'.
<code>punctuation</code>	ASCII 구두점 문자들을 담은 문자열
<code>printable</code>	출력할 수 있는 문자들을 담은 문자열. 문자, 숫자, 구두점, 공백을 합한 것.
<code>uppercase</code>	현재 로케일 설정에 따른 모든 대문자들을 담은 문자열
<code>whitespace</code>	모든 공백 문자를 담은 문자열. 보통 스페이스, 탭, 개행, 리턴, 품퍼드, 수직 탭을 포함한다.

상수 중에서 몇 개는 시스템의 로케일 설정에 따라서 달라진다(예를 들어, letters 와 uppercase).

Formatter 객체

문자열의 str.format() 메서드는 고급 문자열 포맷 지정 연산을 수행한다. 3장과 4장에서 설명했듯이 이 메서드는 순서열이나 매핑의 특정 항목 또는 객체의 속성에 접근할 수 있게 하고 기타 관련 연산을 지원한다. string 모듈은 나름대로의 포맷 지정 연산을 구현하는 데 사용할 수 있는 Formatter 클래스를 정의한다. 이 클래스는 문자열 포맷 지정 연산을 구현하는 데 필요한 부분을 커스터마이즈할 수 있게 노출한다.

fFormatter()

새로운 Formatter 인스턴스를 생성한다. Formatter 인스턴스 f는 다음 연산들을 지원한다.

f.format(format_string, *args, **kwargs)

문자열 format_string의 포맷을 지정한다. 기본 결과는 format_string.format(*args, **kwargs)을 호출한 것과 동일하다. 예를 들어, f.format("{name} is {0:d} years old", 39, name="Dave")는 문자열 "Dave is 39 years old"를 생성한다.

f.vformat(format_string, args, kwargs)

f.format()의 작업을 실제로 수행하는 메서드. args는 위치 인수들의 튜플이고 kwargs는 키워드 인수들의 사전이다. 이미 인수들을 튜플이나 사전으로 가지고 있다면 이 메서드를 호출하는 것이 더 빠르다.

f.parse(format_string)

포맷 문자열 format_string의 내용을 파싱하기 위한 반복자를 생성하는 함수. 이 반복자는 포맷 문자열을 훑으면서 (literal_text, field_name, format_spec, conversion) 형식의 튜플들을 생성한다. literal_text는 대괄호 { ... }로 둘러싸인 다음 포맷 지정자 앞에 오는 상수 텍스트이다. 앞선 텍스트가 없으면 빈 문자열일 수도 있다. field_name은 포맷 지정자에서 필드 이름을 지정하는 문자열이다. 예를 들어, 포맷을 '{0:d}'처럼 지정하면 필드 이름은 '0'이 된다. format_spec은 콜론 다음에 나오는 포맷 지정자 부분이다. 앞의 예에서는 'd'가 된다. 없을 경우 빈 문자열이다. conversion은 변환 지정자를 나타낸다. 앞의 예에서는 None이고 포맷 지정자가 '{0:s:d}'라면 's'로 설정된다. field_name, format_spec, conversion은 포맷 문자열의 마지막에는 모두 None으로 설정된다.

f.get_field(fieldname, args, kwargs)

args와 kwargs에서 주어진 fieldname에 연결된 값을 추출한다. fieldname은 앞에 나온 parse() 메서드가 반환하는 “0”이나 “name”이다. 이 메서드는 튜플 (value, key)을 반환한다. 여기서 value는 필드 값을, key는 args나 kwargs에서 해당 값을 찾는 데 쓰인 키를 나타낸다. key가 정수이면 args에서 색인이다. 키가 문자열이면 kwargs에서 사용된 키이다. fieldname에는 추가로 ‘0.name’이나 ‘0[name]’처럼 색인이나 속성 검색을 담을 수 있다. 이 경우 이 메서드는 추가로 검색을 수행하고 적절한 값을 반환한다. 그리고 반환되는 튜플에서 key의 값은 ‘0’으로 설정된다.

f.get_value(key, args, kwargs)

args나 kwargs에서 key에 해당하는 객체를 추출한다. key가 정수이면 args에서 객체를 얻고 문자열이면 kwargs에서 얻는다.

f.check_unused_args(used_args, args, kwargs)

format() 연산에서 사용되지 않은 인수들을 찾는다. used_args는 포맷 문자열에서 찾아진 사용된 인수 키들의 집합이다(get_field()를 참고). args와 kwargs는 format()에 전달된 위치 인수와 키워드 인수를 나타낸다. 사용되지 않은 인수가 있으면 기본으로 TypeError를 반환한다.

f.format_value(value, format_spec)

주어진 포맷 지정 방식에 따라 값 하나의 포맷을 지정한다. 기본으로 단순히 내장 함수인 format(value, format_spec)을 호출한다.

f.convert_field(value, conversion)

지정된 변환 코드에 따라 get_field()가 반환한 값을 변환한다. conversion이 None이면 수정하지 않고 값을 반환한다. conversion이 ‘s’ 또는 ‘r’이면 각각 값을 str()이나 repr()을 사용해서 변환한다.

자신만의 문자열 포맷 지정기를 만들고 싶으면 Formatter 객체를 생성하고 원하는 대로 포맷을 지정하기 위해서 기본 메서드들을 사용하면 된다. 또는 Formatter에서 상속을 받아서 새로운 클래스를 생성하고 앞에 나온 메서드 몇 개를 다시 구현할 수도 있다.

포맷 지정자의 문법이나 고급 문자열 포맷 지정에 관한 상세한 정보는 3장과 4장에서 찾을 수 있다.

Template 문자열

string 모듈은 문자열 치환을 간단하게 수행할 수 있는 새로운 문자열 타입인 Template을 정의한다. 9장에서 관련 예를 하나 살펴보았다.

다음은 새 템플릿 문자열 객체를 하나 생성한다.

t.Template(s)

여기서 s는 문자열이고 Template은 클래스이다.

Template 객체 t는 다음 메서드들을 지원한다.

t.substitute(m [, **kwargs])

이 메서드는 매핑 객체 m(예를 들어, 사전)과 키워드 인수 리스트를 받아서 문자열 t에 대해서 키워드를 치환한다. 문자열 '\$\$'는 '\$'로 치환되고 키워드 인수가 주어진 경우 문자열 '\$key'나 '\${key}'는 m['key']나 kwargs['key']로 치환된다. key는 유효한 파이썬 식별자이어야 한다. 최종 문자열이 치환이 안 된 '\$key' 패턴을 담고 있으면 KeyError 예외가 발생한다.

t.safe_substitute(m [, **kwargs])

예외나 에러가 발생하지 않는다는 점을 제외하면 substitute()와 동일하다. 대신 치환이 안 된 \$key는 그대로 남겨진다.

t.template

Template()에 전달한 원래 문자열을 담는다.

Template 클래스의 작동 방식은 이 클래스에서 상속을 받은 후 delimiter와 idpattern 속성을 재정의하여 변경할 수 있다. 예를 들어, 다음 코드는 탈출 문자 \$를 @로 바꾸고 키 이름으로 문자만 쓸 수 있게 제한한다.

```
class MyTemplate(string.Template):
    delimiter = '@'          # 탈출 순서열을 나타내는 상수 문자
    idpattern = '[A-Z]*'     # 식별자 정규 표현식 패턴
```

유ти리티 함수

string 모듈은 문자열을 다루는 데 사용할 수 있는 문자열 객체에 메서드로 정의되지 않은 함수 두 개를 담고 있다.

capwords(s)

s에 있는 각 단어의 첫 글자를 대문자로 바꾸고 연속으로 나오는 공백 문자들을

단일 스페이스로 바꾸고 앞뒤에 나오는 공백들을 제거한다.

maketrans(`from`, `to`)

`from`에 있는 각 문자를 `to`의 같은 위치에 있는 문자로 매핑하는 변환 표를 생성한다. `from`과 `to`는 길이가 같아야 한다. 이 함수는 문자열의 `translate()` 메서드에 전달할 인수를 만드는 데 사용할 수 있다.

struct

`struct` 모듈은 파이썬과 이진 데이터 구조(파이썬 바이트 문자열로 표현된다) 사이에 데이터를 변환하는 데 사용한다. 이 데이터 구조는 C로 작성된 함수, 이진 파일 포맷, 네트워크 프로토콜, 시리얼 포트를 통한 이진 통신 등에 사용한다.

포장 함수와 풀기 함수

다음 모듈 수준 함수들은 데이터를 바이트 문자열로 포장하거나 푸는 데 사용한다. 이 작업을 자주 해야 하면 다음 절에 나오는 `Struct` 객체를 사용하는 것을 고려하라.

pack(`fmt`, `v1`, `v2`, ...)

값 `v1`, `v2` 등을 `fmt`으로 지정한 포맷 문자열에 따라 바이트 문자열로 포장한다.

pack_into(`fmt`, `buffer`, `offset`, `v1`, `v2` ...)

값 `v1`, `v2` 등을 바이트 오프셋 `offset`에서 시작해 쓰기 가능한 버퍼 객체에 포장한다. 버퍼 인터페이스를 지원하는 객체에 대해서만 작동한다. `array.array`나 `bytearray` 객체 등이 여기에 포함된다.

unpack(`fmt`, `string`)

`fmt`로 지정한 포맷 문자열에 따라서 바이트 문자열 `string`의 내용을 푼다. 풀린 값들의 튜플을 반환한다. `string`의 길이는 `calcsize()` 함수가 결정하는 포맷의 크기와 정확히 일치해야 한다.

unpack_from(`fmt`, `buffer`, `offset`)

오프셋 `offset`에서 시작해서 `fmt`로 지정한 포맷 문자열에 따라 `buffer` 객체의 내용을 푼다. 풀린 값들의 튜플을 반환한다.

calcsize(fmt)

포맷 문자열 fmt에 대응하는 구조의 크기를 바이트로 계산한다.

Struct 객체

struct 모듈은 포장과 풀기를 위한 또 다른 인터페이스를 제공한다. 포맷 문자열이 한 번만 처리되기 때문에 이 클래스를 사용하는 것이 더 효율적이다.

Struct(fmt)

주어진 포맷 코드에 따라 포장된 데이터를 나타내는 Struct의 인스턴스를 생성한다. Struct 인스턴스 s는 이전 절에서 설명한 함수들과 동일한 일을 수행한다.

메서드	설명
s.pack(v1, v2, ...)	값들을 바이트 문자열로 포장한다.
s.pack_into(buffer, offset, v1, v2, ...)	값들을 버퍼 객체로 포장한다.
s.unpack(bytes)	바이트 문자열에서 값들을 푸다.
s.unpack_from(buffer, offset)	버퍼 객체에서 값들을 푸다.
s.format	사용되는 포맷 코드
s.size	포맷의 크기를 바이트로

포맷 코드

struct 모듈에서 사용되는 포맷 문자열은 다음과 같이 해석할 수 있는 문자들의 순서열이다.

포맷	C 타입	파이썬 타입
'x'	채우기 바이트	값이 없음
'c'	char	길이 1인 String
'b'	signed char	Integer
'B'	unsigned char	Integer
'?'	_Bool (C99)	Boolean
'h'	short	Integer
'H'	unsigned short	Integer
'i'	int	Integer
'I'	unsigned int	Integer
'l'	long	Integer
'L'	unsigned long	Integer
'q'	long long	Long
'Q'	unsigned long long	Long
'f'	float	Float
'd'	double	Float

's'	char[]	String
'p'	char[]	첫 번째 바이트가 길이를 나타내는 String
'P'	void *	Integer

각 포맷 문자 앞에는 정수를 써서 반복 횟수를 지정할 수 있다(예를 들어, '4i'는 'iiii'와 같다). 's' 포맷에 대해서는 문자열의 최대 길이를 나타낸다. 즉, '10s'는 10바이트 문자열을 나타낸다. 포맷 '0s'는 길이 영인 문자열을 가리킨다. 포맷 'p'는 첫 바이트가 길이를 나타내고 나머지가 데이터를 나타내는 문자열을 인코딩하는 데 사용한다. 패스칼 코드나 매킨토시에서 작업을 할 때 유용하게 사용할 수 있다. 문자열의 길이는 최대 255자가 될 수 있다.

값을 푸는 데 포맷 'I'나 'L'을 쓰면 반환되는 값은 긴 정수이다. 포맷 'P'는 머신 워드 크기에 따라서 정수나 긴 정수를 반환할 수 있다.

각 포맷 문자열의 첫 번째 문자는 바이트 순서와 포장된 데이터의 정렬 방식을 지정하는 데 사용할 수 있다. 다음을 보자.

포맷	바이트 순서	크기와 정렬
'@'	기계 특화	기계 특화
'='	기계 특화	표준
'<'	리틀 엔디안	표준
'>'	빅 엔디안	표준
'!'	네트워크(빅 엔디안)	표준

기계 특화 바이트 순서는 머신 아키텍처에 따라서 리틀 엔디안이나 빅 엔디안이 될 수 있다. 기계 특화 크기와 정렬은 C 컴파일러에서 사용하는 값에 대응하고 구현마다 다르다. 표준 정렬은 정렬 자체가 필요 없음을 가정한다. 표준 크기는 short는 2바이트, int는 4바이트, long은 4바이트, float는 32비트, double은 64비트이다. 포맷 'P'는 기계 특화 바이트 순서만 사용할 수 있다.

Note

- 타입별 정렬 요구 사항에 따라서 어떤 구조의 끝 부분을 정렬해야 하는 경우도 있다. 이를 위해서 구조 포맷 문자열을 반복 횟수가 0인 해당 타입 코드로 끝마치면 된다. 예를 들어, 포맷 '#h@'은 4바이트 경계에서 구조를 끝내라고 지정한다(long이 4바이트 경계를 따라 정렬된다고 할 때). 이 경우 'h' 코드로 지정한 short 값 다음에 채우기 바이트가 두 개 추가된다. 이러한 사항은 기계 특화 크기와 정렬이 사용될 때에만 의미가 있다. 표준 크기와 정렬에서는 정렬 규칙을 강제하지 않는다.
- 포맷 'q'와 'Q'는 파이썬을 빌드하는 데 사용한 C 컴파일러가 long long 데이터 타입을 지원할 때만 기계 특화 모드에서 사용할 수 있다.

참고

15장 ‘array’(319페이지), 26장 ‘ctypes’(755페이지)

unicodedata

unicodedata 모듈은 모든 유니코드 문자에 대한 문자 속성을 담고 있는 유니코드 문자 데이터베이스에 접근하는 데 사용한다.

bidirectional(unichr)

unichr에 할당된 양방향 범주^{*}를 문자열로 반환하거나 값이 정의되어 있지 않으면 빈 문자열을 반환한다. 다음 문자열 중 하나를 반환한다.

값	설명
L	왼쪽에서 오른쪽으로
LRE	왼쪽에서 오른쪽으로 내장
LR0	왼쪽에서 오른쪽으로 덮어쓰기
R	오른쪽에서 왼쪽으로
AL	오른쪽에서 왼쪽으로 아라비아
RLE	오른쪽에서 왼쪽으로 내장
RLO	오른쪽에서 왼쪽으로 덮어쓰기
PDF	방향 포맷 종료
EN	유럽 숫자
ES	유럽 숫자 구분자
ET	유럽 숫자 종료
AN	아라비아 숫자
CS	공통 숫자 구분자
NSM	스페이스 아닌 표시
BN	경계 중립
B	문단 구분자
S	구획 구분자
WS	공백
ON	기타 중립

category(unichar)

unichr의 일반 범주를 설명하는 문자열을 반환한다. 반환되는 문자열은 다음 중 하나다.

*(옮긴이) 유니코드 문자를 어떤 방향으로 출력해야 하는지를 나타내는 정보다. 오른쪽에서 왼쪽으로 쓰는 아랍어 문장에서 영어 문장을 인용한 경우 이 문장은 양방향성을 띠게 된다(영어는 왼쪽에서 오른쪽으로 쓰므로). 양방향 범주는 이런 문장을 제대로 출력하기 위한 정보를 제공한다.

값	설명
Lu	문자, 대문자
Ll	문자, 소문자
Lt	문자, 제목 대소문자
Mn	표시, 스페이스 아님
Mc	표시, 스페이스 결합
Me	표시, 둘리쌈
Nd	숫자, 십진수 숫자
Nl	숫자, 문자
No	숫자, 기타
Zs	구분자, 스페이스
Zl	구분자, 줄
Zp	구분자, 문단
Cc	기타, 제어
Cf	기타, 포맷
Cs	기타, 대리자
Co	기타, 사적으로 사용
Cn	기타, 할당되지 않음
Lm	문자, 수정자
Lo	문자, 기타
Pc	구두점, 연결자
Pd	구두점, 대시
Ps	구두점, 열기
Pe	구두점, 닫기
Pi	구두점, 처음 따옴표
Pf	구두점, 마지막 따옴표
Po	구두점, 기타
Sm	기호, 수학
Sc	기호, 통화
Sk	기호, 수정자
So	기호, 기타

combining(unichar)

unichar의 결합 부류(combining class)를 설명하는 정수를 반환하거나 결합 부류가 정의되어 있지 않으면 0을 반환한다. 다음 값 중 하나를 반환한다.

값	설명
0	스페이스, 분리, 에워쌈, 레오드란트(reordrant), 티베트 아래 글자(subjoined)
1	위에 겹치거나 내부
7	누크타스(Nuktas)
8	히라가나/가타가나 소리 표시
9	비라마스(Viramas)

10-199	고정 위치 부류
200	왼쪽 아래에 붙음
202	아래에 붙음
204	오른쪽 아래에 붙음
208	왼쪽에 붙음
210	오른쪽에 붙음
212	왼쪽 위에 붙음
214	위에 붙음
216	오른쪽 위에 붙음
218	왼쪽 아래
220	아래
222	오른쪽 아래
224	왼쪽
226	오른쪽
228	왼쪽 위
230	위
232	오른쪽 위
233	아래쪽 두 번
234	위쪽 두 번
240	아래(이오타 첨자(iota subscript))

decimal(unichr[, default])

문자 unichar에 할당된 십진수 정수 값을 반환한다. unichr가 십진수 숫자가 아니면 default가 반환되거나 ValueError가 발생한다.

decomposition(unichar)

unichar의 분해 매핑을 담은 문자열을 반환하거나 이러한 매핑이 정의되지 않을 경우 빈 문자열을 반환한다. 보통 강세 표시가 있는 문자는 여러 문자 순서열로 분해할 수 있다. 예를 들어, decomposition(u"\u00fc")("ü")는 문자 u와 움라우트('') 강세 표시에 대응하는 문자열 "0075 0308"를 반환한다. 이 함수가 반환하는 문자열에는 다음 문자열들이 들어 있을 수 있다.

값	설명
	폰트의 일종(예를 들어, blackletter 형식)
<noBreak>	분리할 수 없는 스페이스나 하이픈
<initial>	초기 표현 형식(아라비아)
<medial>	중간 표현 형식(아라비아)
<final>	최종 표현 형식(아라비아)
<isolated>	고립된 표현 형식(아라비아)
<circle>	애워싼 형식

<code><super></code>	위첨자 형식
<code><sub></code>	아래첨자 형식
<code><vertical></code>	수직 배치 표현 형식
<code><wide></code>	넓은(또는 전각zenkaku) 호환 문자
<code><narrow></code>	좁은(또는 반각hankaku) 호환 문자
<code><small></code>	작은 형식(CNS 호환)
<code><square></code>	한중일(CJK) 정사각형 폰트 일종
<code><fraction></code>	일반 분수 형식
<code><compat></code>	기타 기술되지 않은 호환 문자

digit(unichr [, default])

문자 unichr에 할당된 정수 숫자 값을 반환한다. unichr가 숫자가 아니면 default가 반환되거나 ValueError가 발생한다. 이 함수는 십진수가 아닌 숫자를 표현하는 문자에 대해서도 작동하는 점에서 decimal()과 다르다.

east_asian_width(unichr)

unichr에 할당된 동아시아 넓이(east Asian width)*를 반환한다.

lookup(name)

이름으로 문자를 검색한다. 예를 들어, lookup('COPYRIGHT SIGN')은 대응하는 유니코드 문자를 반환한다. <http://www.unicode.org/charts>에 가면 흔히 사용되는 이름을 볼 수 있다.

mirrored(unichr)

unichr가 양방향 텍스트에서 '반영(mirrored)' 문자이면 1을 반환하고 아니면 0을 반환한다. 반영 문자는 텍스트를 반대 순서로 출력할 경우 제대로 보이는 문자이다. 예를 들어, 문자 '('는 뒤 텍스트가 오른쪽에서 왼쪽으로 쓰여질 경우 ')'로 뒤집는 것이 말이 되기 때문에 반영 문자이다.

name(unichr [, default])

유니코드 문자 unichar를 반환한다. 이름이 정의되어 있지 않으면 ValueError를 발생시키거나 default가 제공된 경우 default를 반환한다. 예를 들어, name(u'\xfc')은 'LATIN SMALL LETTER U WITH DIAERESIS'를 반환한다.

* (옮긴이) 동아시아 넓이에 대해서는 다음 링크를 참고하기 바란다.

http://ko.wikipedia.org/wiki/전각_문자와_반각_문자

normalize(form, unistr)

유니코드 문자열 unistr를 정규 형식인 form에 따라서 정규화한다. form은 ‘NFC’, ‘NFKC’, ‘NFD’, ‘NFKD’ 중 하나가 될 수 있다. 문자열을 정규화하는 작업은 특정 문자를 결합하거나 분해하는 것과 어느 정도 관련이 있다. 예를 들어, 단어 “resumé”의 유니코드 문자열은 u‘resum\u00e9’나 문자열 u‘resume\u0301’로 표현될 수 있다. 첫 번째 문자열에서 강세 문자인 é는 문자 하나로 표현된다. 두 번째 문자열에서는 문자 e와 뒤에 결합되는 강세 표시(’)로 표현되었다. ‘NFC’ 정규화는 문자열 unistr를 변환하여 모든 문자가 완전히 결합되게 한다(예를 들어, é는 글자 하나로 표현된다. ‘NFD’ 정규화는 unistr를 변환하여 문자를 분해한다(예를 들어, 문자 é는 e와 강세로 표현한다). ‘NFKC’와 ‘NFKD’는 ‘NFC’와 ‘NFD’와 같지만 추가로 두 개 이상의 유니코드 문자 값으로 표현되는 특정 문자를 하나의 표준 값으로 변환한다. 예를 들어 로마 숫자는 각각 유니코드 문자 값을 갖지만 I, V, M 같은 라틴문자들로 표현되기도 한다. ‘NFKC’와 ‘NFKD’는 특수한 로마숫자들을 대응하는 라틴 문자들로 변환한다.

numeric(unichr[, default])

유니코드 문자 unichr에 할당된 값을 부동 소수점 숫자로 반환한다. 숫자 값이 정의된 것이 없으면 default가 반환되거나 ValueError가 발생한다. 예를 들어, U+2155(분수 “1/5”에 대한 문자)의 숫자 값은 0.2이다.

unidata_version

사용된 유니코드 데이터베이스 버전을 담은 문자열(예를 들어, ‘5.1.0’).

Note

유니코드 문자 데이터베이스에 관해서 더 알고 싶다면 <http://www.unicode.org>를 참고하도록 한다.

17장

Python Essential Reference

파이썬 데이터베이스 접근

이 장에서는 파이썬으로 관계 데이터베이스나 해시 테이블 스타일 데이터베이스에 접근하는 데 사용하는 프로그램 인터페이스를 설명한다. 특정 라이브러리 모듈을 다루었던 다른 장과는 달리 이 장에서 다루는 내용 중 일부는 써드 파티 확장 기능에 적용되는 내용이다. 예를 들어, MySQL이나 오라클 데이터베이스에 접근하려면 먼저 써드 파티 확장 모듈을 다운로드 받아야 한다. 그런 다음 여기서 설명하는 기본 사용 방식을 따라서 사용하면 된다.

관계 데이터베이스 API 명세서

파이썬 커뮤니티에서는 관계 데이터베이스에 접근하는 데 사용할 수 있는 파이썬 데이터베이스 API 명세서 버전 2.0(Python Database API Specification V2.0) 또는 줄여서 PEP 249라고 하는 표준을 개발하였다(공식 문서는 <http://www.python.org/dev/peps/pep-249/>에 있다). 특정 데이터베이스 모듈은 기본적으로 이 명세서를 따르지만 추가 기능을 제공할 수도 있다. 이 절에서는 대부분의 응용 프로그램에서 필요한 필수 요소를 설명한다.

크게 보아 데이터베이스 API는 데이터베이스 서버에 접속하고 SQL 질의를 실행하며 결과를 얻는 데 사용할 수 있는 함수와 객체를 정의한다. 두 주요 객체가 사용된다. Connection 객체는 데이터베이스 연결을 관리하는 데 사용하고 Cursor 객체는 질의를 수행하는 데 사용한다.

연결

데이터베이스에 연결하기 위해서 모든 데이터베이스 모듈에서는 모듈 수준의 함수인 `connect(parameters)`를 제공한다. 정확한 매개변수는 데이터베이스의 종류에 따라 다를 수 있지만 보통은 데이터 소스 이름, 사용자 이름, 암호, 호스트 이름, 데이터베이스 이름 같은 정보가 필요하다. 이런 정보는 보통 `dsn`, `user`, `password`, `host`, `database` 같은 키워드 인수를 사용하여 전달한다. 예를 들어, `connect()`는 다음과 같이 호출할 수 있다.

```
connect(dsn="hostname:DBNAME", user="michael", password="peekaboo")
```

성공할 경우 `Connection` 객체가 반환된다. `Connection` 인스턴스 `c`는 다음 메서드들을 갖는다.

`c.close()`

서버로의 연결을 닫는다.

`c.commit()`

결정을 기다리는 모든 트랜잭션을 데이터베이스에 커밋한다. 데이터베이스에서 트랜잭션을 지원하면 수정한 내용을 반영하기 위해서 이 메서드를 반드시 호출해야 한다. 데이터베이스가 트랜잭션을 지원하지 않으면 이 메서드는 아무 일도 하지 않는다.

`c.rollback()`

결정을 기다리는 트랜잭션이 있으면 시작 지점으로 데이터베이스를 되돌린다. 이 메서드는 종종 트랜잭션을 지원하지 않는 데이터베이스에서 데이터베이스에 가해진 변화를 되돌리는 데 사용되기도 한다. 예를 들어, 데이터베이스를 사용하는 도중에 코드에서 예외가 발생한 경우 이 메서드를 사용해서 예외가 발생하기 전에 수행한 변화를 되돌릴 수 있다.

`c.cursor()`

이 연결을 사용하는 새로운 `Cursor` 객체를 만든다. 커서(cursor)는 SQL 질의를 실행하고 결과를 얻는 데 사용하는 객체이다.

커서

데이터베이스에 어떤 작업을 수행하려면 먼저 연결 `c`를 만들고 그 다음으로

`c.cursor()` 메서드를 호출해서 Cursor 객체를 생성해야 한다. Cursor 인스턴스 cur는 질의를 실행하는 데 사용할 수 있는 몇 가지 표준 메서드와 속성을 가지고 있다.

cur.callproc(procname [, parameters])

이름이 procname인 저장된 프로시저(stored procedure)를 호출한다. parameters는 프로시저에 인수로서 전달할 값들이다. 이 함수의 결과는 parameters와 항목의 개수가 같은 순서열이다. 이 순서열은 parameters의 복사본인데, 인수 중에서 출력 인수의 값은 실행 후 수정된 값으로 교체된다. 프로시저가 출력 집합을 생성할 경우 나중에 설명할 `fetch()` 메서드로 값을 읽을 수 있다.

cur.close()

커서를 닫아서 추가 연산을 수행할 수 없게 만든다.

cur.execute(query [, parameters])

데이터베이스에 질의나 명령을 실행한다. query는 명령(보통 SQL)을 담은 문자열이고 parameters는 질의 문자열에 들어 있는 변수에 값을 할당하는 데 사용할 순서열이나 매핑이다(여기에 관해서는 다음 절에서 설명한다).

cur.executemany(query [, parametersequence])

질의나 명령을 반복해서 실행한다. query는 질의 문자열이고 parametersequence는 매개변수 순서열이다. 이 순서열에서 각 항목은 앞에서 본 `execute()` 메서드에서 사용하는 순서열이나 매핑 객체이다.

cur.fetchone()

`execute()`나 `executemany()`가 생성하는 결과 집합에서 다음 행을 반환한다. 보통 각 열의 값을 담은 리스트나 튜플이다. 행이 더 없으면 `None`을 반환한다. 남은 결과가 없거나 이전에 실행한 연산이 결과 집합을 만들지 않은 경우 예외가 발생한다.

cur.fetchmany([size])

결과 행들의 순서열을 반환한다(예를 들어, 튜플들의 리스트). size는 반환할 행의 수다. 생략하면 `cur.arraysize`가 기본으로 사용된다. 실제 반환되는 행의 수는 요청한 것보다 작을 수 있다. 더 이상 행이 없으면 빈 순서열을 반환한다.

cur.fetchall()

남은 모든 결과 행을 담은 순서열을 반환한다(예를 들어, 튜플들의 리스트).

cur.nextset()

현재 결과 집합에서 남아 있는 모든 행을 버리고 다음 결과 집합으로 넘어간다 (있는 경우). 결과 집합이 더 없는 경우에는 None을 반환한다. 아니면 True 값이 반환되고 이어지는 fetch*() 연산에서 새로운 집합으로부터 데이터를 반환한다.

cur.setinputsizes(sizes)

이어지는 execute*() 메서드 호출에 전달된 매개변수에 대한 힌트를 커서에 준다. sizes는 타입 객체들의 순서열이거나 각 매개변수에 대해 기대하는 최대 문자열 길이를 주는 정수들일 수 있다. 이 메서드는 내부적으로 데이터베이스에 보내질 질의와 명령을 생성하는 데 사용할 메모리 버퍼를 미리 정의하는 데 사용한다. 이 메서드를 사용하면 이어지는 execute*() 연산의 속도를 높일 수 있다.

cur.setoutputsizes(size [, column])

결과 집합들에 있는 특정 열의 버퍼 크기를 설정한다. column은 결과 행에서 정수 색인을 나타내고 size는 바이트 수를 나타낸다. 이 메서드는 보통 execute*() 호출 전에 문자열, BLOB, LONG 같은 큰 데이터베이스 열에 대해 제한을 걸기 위해서 사용한다. column을 생략하면 결과에 있는 모든 열에 제약이 가해진다.

커서는 현재 결과 집합을 설명하고 커서 자체에 대한 정보를 제공하는 몇 가지 속성을 갖고 있다.

cur.arraysize

fetchmany() 연산에 사용될 기본 값을 주는 정수. 이 값은 데이터베이스 모듈마다 다를 수 있고 내부적으로 모듈에서 최적이라고 생각하는 값으로 설정될 수도 있다.

cur.description

현재 결과 집합에 있는 각 열에 대한 정보를 제공하는 튜플들의 순서열. 각 튜플은 (name, type_code, display_size, internal_size, precision, scale, null_ok) 형태를 갖는다. 첫 번째 필드는 항상 정의되고 열 이름을 나타낸다. type_code는 ‘타입 객체’ 절에서 설명되는 타입 객체와 관련해서 비교를 수행하는 데 사용한다. 다른 필드들은 특정 열에 적용되지 않을 경우 None으로 설정된다.

cur.rowcount

execute*() 메서드가 생성하는 최종 결과에서 행의 수. -1로 설정되면 결과 집합이

없거나 행 수를 알 수 없다는 것을 의미한다.

다음 예는 내장 라이브러이인 sqlite3 데이터베이스 모듈을 사용해서 앞에서 설명한 연산 몇 가지를 사용하는 방법을 보여준다.

```
import sqlite3
conn = sqlite3.connect("dbfile")
cur = conn.cursor( )

# 간단한 질의
cur.execute("select name, shares, price from portfolio where
account=12345")

# 결과에 대해 루프를 돋다
while True:
    row = cur.fetchone( )
    if not row: break
    # 이번 행을 처리한다
    name, shares, price = row
    ...

# 다른 방법(반복을 사용한다)
cur.execute("select name, shares, price from portfolio where
account=12345")
for name, shares, price in cur:
    # 이번 행을 처리한다
    ...
    ...
```

질의 만들기

데이터베이스 API를 사용할 때 핵심적인 부분이 바로 커서 객체의 execute*() 메서드에 전달할 SQL 질의 문자열을 생성하는 부분이다. 여기서 까다로운 것은 사용자가 준 매개변수로 질의 문자열을 채우는 부분이다. 예를 들어, 여러분은 흔히 다음과 같은 코드를 작성하게 된다.

```
symbol = "AIG"
account = 12345
cur.execute("select shares from portfolio where name='%s'
            and account=%d" % (symbol, account))
```

이렇게 해도 돌아가기는 하지만 절대로 파이썬에서 문자열 연산을 사용해서 직접 질의를 작성하지 않도록 한다. 그렇게 하면 여러분의 코드는 잠재적인 SQL 주입 공격에 노출된다. SQL 주입 공격을 하면 누군가가 데이터베이스에 대고 임의의 문장을 실행할 수 있게 된다. 예를 들어, 앞에 나온 코드에서 누군가가 symbol 값으로 “EVIL LAUGH”; drop table portfolio;—처럼 생긴 값을 줄 수 있다. 여러분이 예상하는 것과는 다른 일이 벌어진다.

모든 데이터베이스 모듈은 값 치환을 위한 자신만의 메커니즘을 제공한다. 예를 들어, 앞에서 본 것처럼 전체 질의를 생성하는 대신 다음과 같이 할 수 있다.

```
symbol = "AIG"
account = 12345
cur.execute("select shares from portfolio where name=?  
and account=?", (symbol, account))
```

여기서 자리 표시자 ‘?’는 차례대로 튜플 (symbol, account)에 있는 값으로 교체된다. ‘?’는 값이 올 수 있는 위치에만 사용할 수 있고, 명령어라든지 표 이름에 해당하는 자리에는 사용할 수 없다.

아쉽게도 데이터베이스 모듈 구현들 사이에 자리 표시자를 나타내기 위한 표준적인 방법은 없다. 그래도 각 모듈에는 질의에서 사용할 값 치환 스타일을 나타내는 `paramstyle`이라는 변수를 정의하고 있다. 이 변수는 다음 중에 한 값을 가질 수 있다.

매개변수 스타일 설명

'qmark'	질의에서 각 ?가 순서열에 있는 항목으로 차례대로 교체되는 의문 부호 스타일. 예를 들어, <code>cur.execute("... where name=? and account=?", (symbol, account))</code> . 매개변수는 튜플로 전달 한다.
'numeric'	:n이 색인 n에 있는 매개변수 값으로 채워지는 숫자 스타일. 예를 들면, <code>cur.execute("... where name=:0 and account=:1", (symbol, account))</code> .
'named'	:name이 이름 있는 값으로 채워지는 이름 있는 스타일. 이 스타일에 대해서 매개변수는 매핑으로 주어져야 한다. 예를 들면, <code>cur.execute("... where name=:symbol and account=:account", {'symbol':symbol, 'account':account})</code> .
'format'	%s, %d 같은 printf 스타일 포맷 코드. 예를 들어, <code>cur.execute("... where name=%s and account=%d", (symbol, account))</code> .
'pyformat'	%(name)s 같은 파이썬의 확장 포맷 코드. 'named' 스타일과 비슷하다. 매개변수는 튜플이 아니라 매핑으로 전달해야 한다.

타입 객체

데이터베이스 데이터로 작업할 때는 보통 정수나 문자열 같은 내장 타입이 데이터베이스에 있는 동등한 타입으로 매핑된다. 날짜, 이진 데이터나 기타 특수한 타입은 다루기가 더 까다롭다. 이 매핑을 돋기 위해서 데이터베이스 모듈은 다양한 타입의 객체를 생성하는 데 사용할 수 있는 생성자 함수를 구현한다.

Date(year, month, day)

날짜를 나타내는 객체를 생성한다.

Time(hour, minute, second)

시간을 나타내는 객체를 생성한다.

Timestamp(year, month, day, hour, minute, second)

타임스탬프를 나타내는 객체를 생성한다.

DateFromTicks(ticks)

시스템 시간 값으로부터 날짜 객체를 생성한다. ticks는 time,time() 같은 함수가 반환하는 값으로서 초의 수를 나타낸다.

TimeFromTicks(ticks)

시스템 시간 값으로부터 시간 객체를 생성한다.

TimestampFromTicks(ticks)

시스템 시간 값으로부터 타임스탬프 객체를 생성한다.

Binary(s)

바이트 문자열 s에서 이진 객체를 생성한다.

이 생성자 함수 말고도 다음 타입 객체가 더 정의되어 있을 수 있다. 이 코드의 목적은 현재 결과 집합의 내용을 설명하는 cur.description의 type_code 필드에 대한 타입 검사를 수행하는 것이다.

타입 객체	설명
STRING	문자 또는 텍스트 데이터
BINARY	BLOB 같은 이진 데이터
NUMBER	숫자 데이터
DATETIME	날짜나 시간 데이터
ROWID	행 아이디 데이터

에러 처리

데이터베이스 모듈에는 다른 에러들의 기반 클래스인 최상위 예외 Error가 있다. 다음 예외는 더 구체적인 데이터베이스 에러를 나타낸다.

예외	설명
InterfaceError	데이터베이스 자체가 아니라 데이터베이스 인터페이스와 관련된 에러.
DatabaseError	데이터베이스 자체와 관련된 에러.
DataError	처리된 데이터와 관련된 에러. 예로 잘못된 타입 변환, 영으로 나누기 등.
OperationalError	데이터베이스 자체의 운영에 관련된 에러. 잃어버린 연결 등.
IntegrityError	데이터베이스의 무결성이 깨질 때 에러.
InternalError	데이터베이스의 내부 에러. 예를 들면, 오래된(stale) 커서.
ProgrammingError	SQL 질의 관련 에러.
NotSupportedError	내부 데이터베이스가 지원하지 않는 데이터베이스 API 관련 에러.

데이터베이스 모듈은 개선 과정에서 데이터 잘림 같은 것을 경고하기 위해서 Warning 예외를 정의할 수도 있다.

다중 스레드

다중스레드 프로그래밍에서 데이터베이스에 접근하는 경우라면 내부 데이터베이스 모듈이 스레드 안전을 보장할 수도 있고 하지 않을 수도 있다. 다음 변수는 각 모듈이 이와 관련된 정보를 제공하기 위해 정의한다.

threadsafety

모듈이 스레드 안전을 보장하는지를 나타내는 정수. 다음 값 중 하나.

- 0 스레드 안전을 보장하지 않음. 여러 스레드가 모듈을 공유하면 안 된다.
- 1 모듈은 스레드 안전을 보장한다. 그래도 연결을 공유하면 안 된다.
- 2 모듈과 연결이 스레드 안전이 보장된다. 커서는 공유하면 안 된다.
- 3 모듈, 연결, 커서 모두 스레드 안전이 보장된다.

결과를 사전에 매핑하기

데이터베이스 결과를 다룰 때 흔히 튜플이나 리스트를 이름 있는 필드를 가진 사전으로 매핑한다. 예를 들어, 질의 결과 집합이 많은 수의 열을 담고 있는 경우 튜플에 특정 필드의 숫자 색인을 직접 써주는 것보다 서술적인 필드 이름을 사용하여 데이터에 접근하는 것이 훨씬 쉽다.

이를 위한 방법이 많이 있지만 결과 데이터를 처리하는 가장 좋은 방법은 생성기 함수를 통하는 것이다. 다음 예를 보자.

```
def generate_dicts(cur):
    import itertools
    fieldnames = [d[0].lower() for d in cur.description]
    while True:
        rows = cur.fetchmany()
```

```

        if not row: return
        for row in rows:
            yield dict(itertools.izip(fieldnames, row))

# 사용 예
cur.execute("select name, shares, price from portfolio")
for r in generate_dicts(cur):
    print r['name'], r['shares'], r['price']

```

데이터베이스마다 열의 이름을 짓는 방법이 완전히 일관성이 있는 것은 아니라는 것을 염두에 두도록 한다. 대소문자 구분 같은 문제는 특히 그렇다. 이 기법을 사용할 때 여러 데이터베이스 모듈에 대해서 작동하게 하려면 이런 점을 주의해야 한다.

데이터베이스 API 확장

마지막으로 특정 데이터베이스 모듈에는 여러 확장 기능과 고급 기능이 들어 있을 수 있다. 예를 들어, 두 단계 커밋(two-phase commit)이라든지 확장된 에러 처리 같은 지원 기능이 들어 있을 수 있다. PEP-249는 이러한 기능을 위해서 권장되는 인터페이스에 관한 정보를 담고 있다. 고급 사용자라면 한번 살펴보기를 권한다. 써드 파티 모듈에서는 관계 데이터베이스 인터페이스를 사용하기 더 간단하게 만드는 경우도 있다.

sqlite3 모듈

sqlite3 모듈은 SQLite 데이터베이스 라이브러리(<http://www.sqlite.org>)에 대해 파일 씬 인터페이스를 제공한다. SQLite는 데이터베이스 전체를 하나의 파일이나 메모리에 두는, 필요한 것을 다 포함하고 있는 관계 데이터베이스를 구현하는 C 라이브러리이다. 간단하기는 하지만 몇 가지 이유에서 이 라이브러리는 아주 매력적이다. 첫째로 별개의 데이터베이스 서버가 필요 없고 특별한 설정을 할 필요도 없다. 여러분은 간단히 데이터베이스 파일에 연결해서 프로그램에서 데이터베이스를 바로 사용할 수 있다(파일이 없으면 새로운 파일이 생성된다). 데이터베이스는 또한 개선된 신뢰성(심지어 시스템이 갑자기 멈추는 경우에도)을 위해서 트랜잭션을 지원할 뿐만 아니라 여러 프로세스에서 동시에 데이터베이스에 접근할 수 있도록 하는 락을 지원한다.

이 라이브러리의 프로그램 인터페이스는 데이터베이스 API에 관한 앞 절에서 설명한 일반적인 규칙을 따르기 때문에 이곳에서 자세한 내용을 다시 설명하지는

않겠다. 대신 이 절에서는 이 모듈을 사용할 때 고려해야 하는 기술적인 문제와 sqlite3 모듈에 국한된 기능을 설명한다.

모듈 수준 함수

다음 함수는 sqlite3 모듈에서 정의하는 함수이다.

```
connect(database [, timeout [, isolation_level [, detect_types]]])
```

SQLite 데이터베이스로 연결을 생성한다. database는 데이터베이스 파일 이름을 지정하는 문자열이다. “:memory:”라고 지정할 경우 메모리 데이터베이스가 사용된다(이 데이터베이스는 파일 프로세스가 실행 중일 때만 유지되고 프로그램이 종료되면 없어진다). 매개변수 timeout은 다른 연결에서 데이터베이스를 갱신하고 있을 때 내부 읽기 쓰기 락을 언제까지 기다렸다가 해제할 것인지를 지정한다. 기본으로 timeout은 5초다. INSERT나 UPDATE 같은 SQL문이 사용될 때 이미 트랜잭션이 있지 않으면 새로운 트랜잭션이 자동으로 시작된다. isolation_level 매개변수는 이 트랜잭션을 시작하는 데 사용된 내부 SQL BEGIN문에 추가적인 수정자를 추가한다. “ ”(기본 값), “DEFERRED”, “EXCLUSIVE”, “IMMEDIATE” 중 한 값을 사용할 수 있다. 이 값들의 뜻은 내부 데이터베이스 락과 관련이 있으며 다음과 같다.

격리 수준	설명
""(빈 문자열)	기본 설정을 사용한다(DEFERRED).
"DEFERRED"	새 트랜잭션을 시작하지만 첫 번째 데이터베이스 연산이 실제로 수행될 때까지 락을 회피하지 않는다.
"EXCLUSIVE"	새로운 트랜잭션을 시작하고 수정 사항을 커밋할 때까지 다른 연결에서 데이터베이스를 읽거나 쓸 수 없다.
"IMMEDIATE"	새 트랜잭션을 시작하고 수정 사항을 커밋할 때까지 다른 연결에서 데이터베이스를 수정하지 못하게 한다. 그래도 다른 연결에서 데이터베이스를 읽을 수는 있다.

detect_types 매개변수는 결과를 반환할 때 추가 타입 감지(SQL문을 추가로 파싱하여 구현한다)를 활성화한다. 기본으로 0으로 설정된다(추가 감지를 하지 않는다). 이 매개변수는 PARSE_DECLTYPES와 PARSE_COLNAMES를 비트 OR하여 설정할 수 있다. PARSE_DECLTYPES가 활성화되면 질의를 살펴보고 “integer”나 “number(8)” 같은 SQL 타입 이름을 찾는다. PARSE_COLNAMES가 활성화되면 “[colname]typename”(큰따옴표까지 포함) 형태의 특수한 문자열을 질의에 넣을 수 있다. 여기서 colname은 열 이름이고 typename은 다음에 설명할 register_

converter() 함수에 등록될 타입의 이름이다. 이 문자열은 SQLite 엔진에 전달될 때 간단히 colname으로 변환되지만 추가로 타입 지정자를 사용해서 질의 결과에 있는 값을 변환할 수도 있다. 예를 들어, ‘select price as “price [devimal]” from portfolio’는 ‘select price as price from portfolio’로 해석되고 질의 결과는 “십진수” 변환 규칙에 따라서 변환된다.

register_converter(typename, func)

connect()에 detect_types 옵션으로 사용할 새로운 타입 이름을 등록한다. typename은 질의에서 사용될 타입 이름을 담은 문자열이고 func는 입력으로 바이트 문자열을 하나 받아서 결과로 파이썬 데이터 타입을 반환하는 함수이다. 예를 들어, sqlite3.register_converter(‘decimal’, decimal.Decimal)을 호출하면, ‘select price as “price [decimal]” from stocks’ 같은 질의를 작성하여 값을 Decimal 객체로 변환할 수 있다.

register_adapter(type, func)

주어진 type의 값을 저장할 때 사용할 파이썬 타입 type을 위한 어댑터 함수를 등록한다. func는 타입 type의 인스턴스를 입력으로 받아서 int, float, UTF-8로 인코딩된 바이트 문자열, 유니코드 문자열 또는 베폐를 결과로 반환한다. 예를 들어, Decimal 객체를 저장하고 싶으면 sqlite3.register_adapter(decimal.Decimal, float) 이렇게 하면 된다.

complete_statement(s)

문자열 s가 세미콜론으로 구분된 하나 이상의 완전한 SQL문을 담고 있으면 True를 반환한다. 사용자로부터 질의를 입력받는 대화식 프로그램을 작성할 때 유용하다.

enable_callback_tracebacks(flag)

변환기나 어댑터 같은 사용자 정의 역호출 함수에서 예외를 처리하는 방식을 지정한다. 기본으로 예외는 무시된다. flag가 True로 설정되면 역추적 메시지가 sys.stderr에 출력된다.

Connection 객체

connect() 함수가 반환하는 Connect 객체 c는 데이터베이스 API에서 설명한 표준 연산을 지원한다. 또한 sqlite3 모듈에 국한된 다음 메서드들도 제공된다.

c.create_function(name, num_params, func)

SQL문에서 사용할 수 있는 사용자 정의 함수를 생성한다. name은 함수의 이름을 담은 문자열이고 num_params는 매개변수 수를 나타내는 정수이고 func는 구현을 담은 파이썬 함수이다. 다음의 간단한 예를 보자.

```
def toupper(s):
    return s.upper()
c.create_function("toupper",1,toupper)
# 질의에서 사용하는 예
c.execute("select toupper(name),foo,bar from sometable")
```

파이썬 함수가 사용되기는 하지만 이 함수의 매개변수나 입력 값은 int, float, str, unicode, buffer, None 중 하나여야 한다.

c.create_aggregate(name, num_params, aggregate_class)

SQL문에서 사용할 사용자 정의 집계 함수를 생성한다. name은 함수 이름을 담은 문자열이고 num_params는 매개변수 수를 나타내는 정수이다. aggregate_class는 집계 연산을 구현하는 클래스이다. 이 클래스는 인수 없는 초기화를 지원하고 num_params로 지정한 것과 동일한 개수의 매개변수를 받는 step(params) 메서드를 구현해야 하며 최종 결과를 반환하는 finalize() 메서드를 구현해야 한다. 다음의 간단한 예를 보자.

```
class Averager(object):
    def __init__(self):
        self.total = 0.0
        self.count = 0
    def step(self,value):
        self.total += value
        self.count += 1
    def finalize(self):
        return self.total / self.count

c.create_aggregate("myavg",1,Averager)

# 질의에서 사용하는 예
c.execute("select myavg(num) from sometable")
```

입력 값으로 step() 메서드를 반복적으로 호출해서 집계가 이루어지고 마지막으로 최종 값을 얻기 위해 finalize()를 호출한다.

c.create_collation(name, func)

SQL문에서 사용할 사용자 정의 대조(collection) 함수를 등록한다. name은 대조 함수 이름을 담은 문자열이고 func는 입력 두 개를 받아서 첫 번째 입력이 두 번

째 입력보다 아래인지, 같은지, 또는 위인지에 따라서 -1, 0, 1을 반환하는 함수이다.

“select * from table order by colname collate name”처럼 SQL 표현식에서 사용자 정의 함수를 사용할 수 있다.

c.execute(sql [, params])

c.cursor()로 커서 객체를 생성하고 sql에 있는 SQL문과 params에 있는 매개변수로 커서의 execute() 메서드를 실행하는 단축 함수.

c.executemany(sql [, params])

c.cursor()로 커서 객체를 생성하고 sql에 있는 SQL문과 params에 있는 매개변수로 커서의 executemany() 메서드를 실행하는 단축 함수.

c.executescript(sql)

c.cursor()로 커서 객체를 생성하고 sql에 있는 SQL문과 params에 있는 매개변수로 커서의 executescript() 메서드를 실행하는 단축 함수.

c.interrupt()

이 연결에서 현재 실행 중인 질의를 취소한다. 보통 다른 스레드에서 호출한다.

c.interdump()

전체 데이터베이스 내용을 나중에 데이터베이스를 다시 만들 때 실행할 SQL문들로 만들어내는 반복자를 반환한다. 데이터베이스를 다른 곳으로 옮기거나 메모리 데이터베이스의 내용을 파일로 내보내서 나중에 복구하는 데 사용할 수 있다.

c.set_authorizer(auth_callback)

데이터베이스에 있는 열 데이터에 접근할 때마다 실행되는 인증 역호출 함수를 등록한다. 역호출 함수는 auth_callback(code, arg1, arg2, dbname, innername)처럼 다섯 개의 인수를 받아야 한다. 이 역호출 함수가 반환하는 값은 접근이 허락되면 SQLITE_OK, SQL문이, 예외와 함께 실패하면 SQLITE_DENY, 해당 열을 널 값으로 취급하여 무시해야 하는 경우에는 SQLITE_IGNORE이어야 한다. 첫 번째 인수인 code는 정수 동작 코드이다. arg1과 arg2 매개변수는 code 값에 따라 다른 값을 갖는다. dbname은 데이터베이스의 이름을 담은 문자열이다(보통 “main”). innername은 접근을 시도하는 가장 안쪽 뷰나 트리거의 이름이거나 뷰나 트리거가 활성화되지 않는 경우에는 None이다. 다음은 code의 값과 arg1과 arg2 매개변수의 뜻을 나열한 것이다.

code	arg1	arg2
SQLITE_CREATE_INDEX	인덱스 이름	표 이름
SQLITE_CREATE_TABLE	표 이름	None
SQLITE_CREATE_TEMP_INDEX	인덱스 이름	표 이름
SQLITE_CREATE_TEMP_TABLE	표 이름	None
SQLITE_CREATE_TEMP_TRIGGER	트리거 이름	표 이름
SQLITE_CREATE_TEMP_VIEW	표 이름	None
SQLITE_CREATE_TRIGGER	트리거 이름	표 이름
SQLITE_CREATE_VIEW	뷰 이름	None
SQLITE_DELETE	표 이름	None
SQLITE_DROP_INDEX	인덱스 이름	표 이름
SQLITE_DROP_TABLE	표 이름	None
SQLITE_DROP_TEMP_INDEX	인덱스 이름	표 이름
SQLITE_DROP_TEMP_TABLE	표 이름	None
SQLITE_DROP_TEMP_TRIGGER	트리거 이름	표 이름
SQLITE_DROP_TEMP_VIEW	뷰 이름	None
SQLITE_DROP_TRIGGER	트리거 이름	표 이름
SQLITE_DROP_VIEW	뷰 이름	None
SQLITE_INSERT	표 이름	None
SQLITE_PRAGMA	프래그마(pragma) 이름	None
SQLITE_READ	표 이름	열 이름
SQLITE_SELECT	None	None
SQLITE_TRANSACTION	None	None
SQLITE_UPDATE	표 이름	열 이름
SQLITE_ATTACH	파일 이름	None
SQLITE_DETACH	데이터베이스 이름	None
SQLITE_ALTER_TABLE	데이터베이스 이름	표 이름
SQLITE_REINDEX	인덱스 이름	None
SQLITE_ANALYZE	표 이름	None
SQLITE_CREATE_VTABLE	표 이름	모듈 이름
SQLITE_DROP_VTABLE	표 이름	모듈 이름
SQLITE_FUNCTION	함수 이름	None

c.set_progress_handler(handler, n)

SQLite 가상 머신에서 n번 명령을 수행했을 때마다 실행할 역호출 함수를 등록한다. handler는 인수를 받지 않는 함수이다.

연결 객체에는 다음 속성들도 정의되어 있다.

c.row_factory

각 결과 행의 내용을 나타내는 객체를 생성할 때 호출되는 함수. 이 함수는 결과

를 얻는 데 사용할 커서 객체와 미가공 결과 행을 담은 튜플 이 두 개의 인수를 받는다.

c.text_factory

데이터베이스에서 텍스트 값을 나타내는 객체를 생성할 때 호출되는 함수. 이 함수는 UTF-8로 인코딩된 바이트 문자열인 하나의 인수를 받아야 한다. 반환되는 값은 문자열의 한 종류여야 한다. 기본으로 유니코드 문자열이 반환된다.

c.total_changes

데이터베이스 연결이 열린 이후로 수정된 행의 개수를 나타내는 정수.

연결 객체의 마지막 기능으로 트랜잭션을 자동으로 처리하기 위해서 컨텍스트 관리자와 함께 사용될 수 있다는 특징이 있다. 다음 예를 보자.

```
conn = sqlite.connect("somedb")
with conn:
    conn.execute("insert into sometable values (?,?)",
    ("foo","bar"))
```

이 예에서, with 블록 안에 있는 모든 문장이 실행되고 아무런 에러가 발생하지 않으면 commit() 연산이 자동으로 수행된다. 예외가 발생하는 경우에는 rollback() 연산이 수행되고 예외가 다시 던져진다.

커서와 기본 연산

sqlite3 데이터베이스에 기본 연산을 수행하려면 먼저 연결 객체의 cursor() 메서드로 커서 객체를 생성해야 한다. 그리고 나서 SQL문을 실행하기 위해 커서의 execute(), executemany(), executescript() 메서드를 호출한다. 이 메서드를 사용하는 방법에 관해서는 데이터베이스 API 절을 참고하도록 한다. 여기서 같은 내용을 다시 설명하기보다는 샘플 코드를 통해서 일반적으로 데이터베이스를 어떻게 사용하는지 보여줄 것이다. 커서 객체를 사용하는 방법을 보여주고 SQL 문법을 복습할 필요가 있는 프로그래머들을 위해서 흔히 사용되는 SQL 연산을 살펴본다.

새 데이터베이스 표 만들기

다음 코드는 데이터베이스를 열고 새 표를 만드는 방법을 보여준다.

```
import sqlite3
conn = sqlite3.connect("mydb")
```

```
cur = conn.cursor()
cur.execute("create table stocks
            (symbol text, shares integer, price real)")
conn.commit()
```

표를 정의할 때 기본 SQLite 데이터 타입인 text, integer, real, blob를 사용해야 한다. blob 타입은 바이트 문자열을 나타내고 text 타입은 UTF-8로 인코딩된 유니코드를 나타낸다.

표에 값을 삽입하기

다음 코드는 표에 새 항목을 삽입하는 방법을 보여준다.

```
import sqlite3
conn = sqlite3.connect("mydb")
cur = conn.cursor()
cur.execute("insert into stocks values (?,?,?)",('IBM',50,91.10))
cur.execute("insert into stocks values (?,?,?)",('AAPL',100,123.45))
conn.commit()
```

값을 삽입할 때는 이 예에서 볼 수 있듯이 항상 ? 치환을 사용해야 한다. 각 ?는 매개변수로 전달한 튜플에 있는 것으로 대체된다.

데이터 순서열을 삽입해야 하는 경우에는 다음과 같이 커서의 executemany() 메서드를 사용하면 된다.

```
stocks = [ ('GOOG',75,380.13),
           ('AA',60,14.20),
           ('AIG',125, 0.99) ]
cur.executemany("insert into stocks values (?,?,?)",stocks)
```

기존 행 갱신하기

다음 코드는 기존 행의 열들을 갱신하는 방법을 보여준다.

```
cur.execute("update stocks set shares=? where symbol=?", (50,'IBM'))
```

SQL문에 값을 삽입할 때에는 ? 자리 표시자를 사용하고 매개변수로 값들의 튜플을 전달해야 한다는 것을 기억하기 바란다.

열을 삭제하기

다음 코드는 열을 삭제하는 방법을 보여준다.

```
cur.execute("delete from stocks where symbol=?", ('SCOX',))
```

기본 질의 수행하기

다음 코드는 기본적인 질의를 수행하고 결과를 얻는 방법을 보여준다.

```
# 표에서 모든 열을 선택한다.
for row in cur.execute("select * from stocks"):
    문장들

# 몇 개의 열을 선택한다.
for shares, price in cur.execute("select shares,price from stocks"):
    문장들

# 부합하는 행들을 선택한다.
for row in cur.execute("select * from stocks
                        where symbol=?", ('IBM',)):
    문장들

# 부합하는 행들을 선택하고 정렬한다.
for row in cur.execute("select * from stocks order by shares"):
    문장들

# 부합하는 행들을 선택하고 반대 순서로 정렬한다.
for row in cur.execute("select * from stocks order by shares desc"):
    문장들

# 공통 열 이름(symbol)으로 표들을 조인한다.
for row in cur.execute("""select s.symbol, s.shares, p.price
                        from stocks as s, prices
                        as p using(symbol)"""):
    문장들
```

DBM 스타일 데이터베이스 모듈

파이썬에는 유닉스 DBM 스타일의 데이터베이스 파일을 지원하는 모듈도 몇 가지 있다. 이러한 데이터베이스의 몇몇 표준 타입이 지원된다. dbm 모듈은 표준 유닉스 dbm 데이터베이스 파일을 읽는 데 사용할 수 있는 모듈이다. gdbm 모듈은 GNU dbm 데이터베이스 파일(<http://www.gnu.org/software/gdbm>)을 읽는 데 사용할 수 있다. dbhash 모듈은 버클리 DB 라이브러리(<http://www.oracle.com/database/berkeley-db/index.html>)가 생성하는 데이터베이스 파일을 읽는 데 사용한다. dumbdbm 모듈은 간단한 DBM 스타일의 데이터베이스를 순수 파이썬으로 구현한 모듈이다.

모든 모듈이 영속 문자열 기반 사전을 구현하는 객체를 제공한다. 즉, 이 객체는 키와 값이 문자열만 될 수 있는 파이썬 사전처럼 작동한다. 데이터베이스 파일은 open() 함수의 일종을 사용해서 연다.

open(filename [, flag [, mode]])

이 함수는 데이터베이스 파일 filename을 열고 데이터베이스 객체를 반환한다. flags는 읽기 전용 접근 ‘r’, 읽기 쓰기 접근 ‘w’, 데이터베이스가 없을 때 생성하도록 하는 ‘c’, 새로운 데이터베이스 생성을 강제하는 ‘n’이 될 수 있다. mode는 데이터베이스를 생성할 때 사용할 정수 파일 접근 모드를 나타낸다(유닉스에서 기본으로 0666).

open() 함수가 반환하는 객체는 최소한 다음에 나오는 사전 같은 연산을 지원한다.

연산	설명
d[key] = value	값을 데이터베이스에 삽입한다.
value = d[key]	데이터베이스에서 데이터를 가져온다.
del d[key]	데이터베이스 항목을 삭제한다.
d.close()	데이터베이스를 닫는다.
key in d	키가 있는지 검사한다.
d.sync()	모든 수정 사항을 데이터베이스에 쓴다.

구현마다 추가 기능을 제공할 수도 있다(자세한 내용은 해당 모듈을 참고할 것).

다양한 DBM 스타일 데이터베이스 모듈과 관련해서 한 가지 문제는 어떤 플랫폼에 모든 모듈이 설치되어 있지 않을 수 있다는 점이다. 예를 들어, 윈도에서 파이썬을 사용하면 dbm과 gdbm 모듈은 보통 사용할 수 없다. 그럼에도 프로그램에서 DBM 스타일의 데이터베이스를 만들고 싶은 경우가 있다. 이러한 문제를 해결하기 위해 파이썬에서는 anydbm이라는 모듈을 제공하여 DBM 스타일의 데이터베이스 파일을 열고 생성하는 데 사용할 수 있다. 이 모듈은 앞에 나온 것 같은 open() 함수를 제공하는데 이 함수는 모든 플랫폼에서 작동하는 것이 보장된다. 이것은 사용 가능한 DBM 모듈을 찾은 다음에 그중에서 가장 고급 라이브러리를 선택함으로써 이루어진다(보통 dbhash가 설치되어 있으면 이것이 사용된다). 만일의 경우에는 어디에나 항상 있는 dumbdbm 모듈을 사용한다.

whichdb 모듈은 whichdb(filename) 함수를 제공하며 이 함수는 어떤 DBM 데이터베이스로 파일을 생성했는지 알아내는 데 사용할 수 있다.

일반적으로 호환성이 중요한 응용 프로그램에서는 여기서 설명한 저수준 모듈을 사용하지 않는 것이 좋다. 예를 들어, 한 머신에서 DBM 데이터베이스를 생성한 다음 데이터베이스 파일을 다른 머신에 옮기면 해당 DBM 모듈이 없을 경우 파이썬에서 그 파일을 읽지 못하는 경우가 생길 수 있다. 많은 양의 데이터를 저장하거나, 여

러 파일 프로그램에서 같은 데이터베이스 파일을 동시에 열거나, 높은 신뢰성과 트랜잭션 등이 필요한 경우에 이 모듈을 사용할 때는 주의해야 한다(이런 경우에는 sqlite3 모듈을 사용하는 것이 더 안전하다).

shelve 모듈

shelve 모듈은 특수한 “선반(shelf)” 객체를 사용해서 객체의 지속성을 지원한다. 이 객체는 담고 있는 모든 객체가 dbhash, dbm, gdbm 같은 해시 테이블 기반 데이터베이스를 사용해서 디스크에 저장된다는 것을 제외하고는 사전처럼 작동한다. 해시 테이블 기반 데이터베이스와는 달리 선반에 저장되는 값들은 문자열에만 국한되지 않는다. pickle 모듈과 호환되는 어떤 객체라도 저장할 수 있다. 선반은 shelve, open() 함수로 생성한다.

```
open(filename [, flag='c' [, protocol [, writeback]]])
```

선반 파일을 연다. 파일이 존재하지 않으면 생성된다. filename은 데이터베이스 파일 이름이어야 하고 확장자를 포함하지 않아야 한다. flag는 앞 절에서 설명한 것과 같은 뜻을 가지며 ‘r’, ‘w’, ‘c’, ‘n’ 중에서 하나가 될 수 있다. 데이터베이스 파일이 없으면 생성된다. protocol은 데이터베이스에 저장될 피클 객체에 사용할 프로토콜을 말한다. pickle 모듈에서 설명한 의미를 갖는다. writeback은 데이터베이스 객체의 캐싱 방식을 제어한다. True일 경우 모든 접근되는 항목은 메모리에 캐시되고 선반이 닫힐 때 다시 써진다. 기본 값은 False이다. 선반 객체를 반환한다.

선반 객체를 열었다면 다음 사전 연산을 수행할 수 있다.

연산	설명
d[key] = data	key에 데이터를 저장한다. 기존 데이터를 덮어쓴다.
data = d[key]	key에 있는 데이터를 추출한다.
del d[key]	key에 있는 데이터를 지운다.
d.has_key(key)	key가 존재하는지를 검사한다.
d.keys()	모든 키를 반환한다.
d.close()	선반을 닫는다.
d.sync()	저장되지 않은 데이터를 디스크에 쓴다.

선반의 키는 문자열이어야 한다. 선반에 저장되는 객체는 pickle 모듈을 사용해서 직렬화할 수 있어야 한다.

Shelf(dict [, protocol [, writeback]])

사전 객체 dict 위에 선반의 기능을 구현하는 혼합 클래스. 이 클래스를 사용하면 반환되는 선반 객체에 저장된 객체들이 피클링되고 내부 사전 dict에 저장된다. protocol과 writeback은 shelve.open()에서의 뜻과 같다.

shelve 모듈은 적절한 DBM 모듈을 선택하기 위해 anydbm 모듈을 사용한다. 대부분의 표준 파이썬 설치 본에는 dbhash가 있을 가능성이 높기 때문에 보통 버클리 DB 라이브러리를 사용하는 dbhash를 사용하게 된다.

18장

Python Essential Reference

파일 및 디렉터리 다루기

이 장에서는 고수준 파일 및 디렉터리 처리와 관련된 파이썬 모듈을 다룬다. 여기에서는 gzip 또는 bzip2 파일 같은 다양한 종류의 기본 파일 인코딩을 처리하는 모듈, zip이나 tar 등의 파일 아카이브를 추출하는 모듈, 그리고 파일 시스템을 조작(디렉터리 출력, 이동, 이름 변경, 복사 등)하기 위한 모듈을 다룬다. 파일과 관련된 저수준 운영체제 호출에 관해서는 19장에서 다루고 HTML과 같은 파일의 내용을 파싱하는 모듈은 24장에서 다룬다.

bz2

bz2 모듈은 bzip2 압축 알고리즘을 사용하여 압축된 데이터를 읽거나 쓰는 데 사용한다.

BZ2File(filename [, mode [, buffering [, compresslevel]]])

이름이 filename인 .bz2 파일을 읽어서 파일과 유사한 객체를 반환한다. mode ‘r’은 읽기를, ‘w’는 쓰기를 나타낸다. ‘rU’를 지정하면 보편 줄바꿈이 지원된다. buffering은 바이트 단위로 버퍼의 크기를 나타내며 기본 값은 0이다(버퍼링하지 않음). compresslevel은 1과 9 사이의 숫자로서 기본 값 9는 가장 높은 수준의 압축을 제공하지만 오랜 처리 시간을 요구한다. 반환된 객체는 close(), read(), readline(), readlines(), seek(), tell(), write(), writelines()와 같은 파일 연산을 지원한다.

BZ2Compressor([compresslevel])

일련의 데이터 블록을 순서대로 압축하는 데 사용할 수 있는 압축 객체를 생성한다. compresslevel은 압축 정도를 나타내며 1에서 9(기본 값) 사이의 값을 가진다.

BZ2Compressor의 인스턴스 c는 다음 두 가지 메서드를 가진다.

c.compress(data)

압축 객체 c에 새로운 바이트 문자열 데이터를 추가한다. 추가가 가능하면 압축된 데이터를 바이트 문자열로 반환한다. 압축은 데이터 덩어리 단위로 이루어지기 때문에 반환되는 바이트 문자열은 모든 데이터를 포함하고 있지 않을 수 있으며, 이전 compress() 호출에서 압축된 데이터를 포함할 수도 있다. 입력 데이터를 모두 넘겨주었으면 압축기에 남아 있는 데이터를 반환하기 위해 flush() 메서드를 호출해야 한다.

c.flush()

내부 버퍼를 비우고, 남은 모든 데이터의 압축된 버전을 담은 바이트 문자열을 반환한다. 이 메서드를 호출한 후에는 객체에 더 이상 compress() 메서드를 호출하면 안 된다.

BZ2Decompressor()

압축해제기 객체를 생성한다.

BZ2Decompressor의 인스턴스 d에는 하나의 메서드만 있다.

d.decompress(data)

바이트 문자열 data로 압축된 데이터 덩어리가 주어지면 이 함수는 압축이 해제된 데이터를 반환한다. 데이터는 덩어리 단위로 처리되기 때문에 반환되는 바이트 문자열에는 data로 제공한 데이터의 압축이 해제된 버전을 모두 포함할 수도 있고 포함하지 않을 수도 있다. 이 메서드를 반복해서 호출하면 입력에서 스트림의 끝을 나타내는 표시가 발견될 때까지 데이터 블록들을 압축해제한다. 그 이후에 압축해제를 시도할 경우 EOFError 예외가 발생한다.

compress(data [, compresslevel])

바이트 문자열 data로 제공된 데이터의 압축된 버전을 반환한다. compresslevel은 1에서 9(기본 값) 사이의 숫자다.

decompress(data)

바이트 문자열 data로 제공된 데이터의 압축 해제된 데이터를 담은 바이트 문자열을 반환한다.

filecmp

filecmp 모듈은 파일 및 디렉터리를 비교하는 다음의 함수들을 지원한다:

cmp(file1, file2 [, shallow])

파일 file1과 file2를 비교하고 같을 경우 True를, 그렇지 않을 경우 False를 반환한다. 기본으로 두 파일이 os.stat()에 의해 반환되는 속성이 동일하면 서로 같다고 간주된다. shallow 매개변수가 False이면 동일한지를 결정하기 위해 두 파일의 내용을 비교한다.

cmpfiles(dir1, dir2, common [, shallow])

두 디렉터리 dir1과 dir2에서 리스트 common에 있는 파일들의 내용을 비교한다. 세 종류의 파일 이름 목록을 담은 튜플(match, mismatch, errors)을 반환한다. match는 두 디렉터리에 같이 존재하는 파일 목록이고 mismatch는 다른 파일 목록이며 errors는 어떤 이유로 인해 비교할 수 없는 파일 목록이다. shallow 매개변수는 cmp()에서의 의미와 동일하다.

dircmp(dir1, dir2 [, ignore[, hide]])

디렉터리 dir1과 dir2에 대해 다양한 비교 연산을 수행하는 데 사용할 수 있는 디렉터리 비교 객체를 생성한다. ignore는 무시할 파일 이름 목록이며 기본 값은 ['RCS', 'CVS', 'tags']이다. hide는 감출 파일 이름 목록이며 기본 값은 [os.curdir, os.pardir](유닉스에서는['.', '..'])이다.

dircmp()에 의해서 반환되는 디렉터리 객체 d는 다음 메서드와 속성을 가진다.

d.report()

dir1과 dir2를 비교하고 그 결과를 sys.stdout으로 출력한다.

d.report_partial_closure()

dir1과 dir2를 비교하고 바로 아래 있는 하위 디렉터리들을 비교한다. 결과를 sys.stdout으로 출력한다.

d.report_full_closure()

dir1과 dir2를 비교하고 모든 하위 디렉터리들을 재귀적으로 비교한다. 결과를 sys.stdout으로 출력한다.

d.left_list

dir1에 있는 파일과 하위 디렉터리 목록. hide와 ignore에 의해 걸러진 것이다.

d.right_list

dir2에 있는 파일과 하위 디렉터리 목록. hide와 ignore에 의해 걸러진 것이다.

d.common

dir1과 dir2 둘 모두에 있는 파일과 하위 디렉터리 목록.

d.left_only

dir1에만 있는 파일과 하위 디렉터리 목록.

d.right_only

dir2에만 있는 파일과 하위 디렉터리 목록.

d.common_dirs

dir1과 dir2 둘 모두에 있는 하위 디렉터리 목록.

d.common_files

dir1과 dir2 둘 모두에 있는 파일 목록.

d.common_funny

dir1과 dir2에 있는 파일 중 타입이 다르거나 os.stat()로 정보를 얻을 수 없는 파일 목록.

d.same_files

dir1과 dir2에서 동일한 내용을 가지는 파일 목록.

d.diff_files

dir1과 dir2에서 다른 내용을 가지는 파일 목록.

d.funny_files

dir1과 dir2 둘 모두에 있는 파일 중 어떤 이유로 인해 서로 비교할 수 없는 파일 목록(예를 들어, 파일 접근 권한이 없다거나).

Note

difrcmp 객체의 속성은 객체가 처음으로 생성될 때 값이 할당되는 것이 아니고 나중에 동적으로 값이 정해진다. 따라서, 일부 속성만 사용하더라도 사용하지 않은 속성 때문에 성능 저하가 발생하지는 않는다.

d.subdirs

d.common_dirs에 있는 이름들을 추가 difrcmp 객체들로 매핑하는 사전.

fnmatch

fnmatch 모듈은 와일드카드 문자를 사용한 유닉스 셸 스타일의 파일 이름 매칭을 지원한다. 이 모듈은 파일 이름 매칭만 수행하기 때문에 실제 파일 목록을 얻는 데는 glob 모듈을 사용해야 한다. 다음은 패턴 문법을 보여준다.

문자(열)	설명
*	모든 것에 매칭
?	문자 하나에 매칭
[seq]	seq에 있는 아무 문자에 매칭
[!seq]	seq에 없는 문자에 매칭

다음의 함수들은 와일드카드 매칭 검사를 하는 데 사용한다

fnmatch(filename, pattern)

filename이 pattern에 매칭되었는지에 따라 True와 False를 반환한다. 대소문자 구분은 운영체제에 의해 결정된다(윈도와 같은 특정 플랫폼에서는 대소문자를 구별하지 않는다).

fnmatchcase(filename, pattern)

대소문자를 구분하여 filename을 pattern과 비교한다.

filter(names, pattern)

names 순서열에 있는 모든 이름에 fnmatch() 함수를 실행하고 pattern에 매치된 이름 목록을 반환한다.

예

```

fnmatch('foo.gif', '*.gif')           # True를 반환
fnmatch('part37.html', 'part3[0-5].html') # False를 반환

# 전체 디렉터리 트리에서 파일을 찾는 예
# os.walk, fnmatch, 생성기를 사용
def findall(topdir, pattern):
    for path, files, dirs in os.walk(topdir):
        for name in files:
            if fnmatch.fnmatch(name, pattern):
                yield os.path.join(path, name)

# 모든 .py 파일을 찾기
for pyfile in findall(".", "*.py"):
    print (pyfile)

```

glob

glob 모듈은 디렉터리에서 유닉스 셸 규칙(fnmatch 모듈에서 다룸)으로 기술된 패턴에 매칭되는 모든 파일 이름을 반환한다.

glob(pattern)

pattern에 매칭된 경로 목록을 반환한다.

iglob(pattern)

glob()와 동일한 결과를 반환하지만 반복자가 사용된다.

예

```

htmlfile = glob('*.*html')
imgfiles = glob('image[0-5]*.gif')

```

Note

틸데(~)와 셸 변수 확장은 수행되지 않는다. 이러한 확장을 수행하려면 glob()를 호출하기 전에 os.path.expanduser()와 os.path.expandvars()를 사용하면 된다.

gzip

gzip 모듈은 GNU gzip 프로그램과 호환되는 파일을 읽거나 쓰는 데 사용할 수 있는 GzipFile 클래스를 제공한다. GzipFile 객체는 파일이 자동으로 압축되고 압축해제된다는 점을 제외하고는 보통의 파일처럼 작동한다.

GzipFile([filename [, mode [, compresslevel [, fileobj]]]])

GzipFile을 연다. filename은 파일 이름이고 mode는 ‘r’, ‘rb’, ‘a’, ‘ab’, ‘w’, ‘wb’ 중 하나이다. 기본 값은 ‘rb’이다. compresslevel은 1에서 9 사이의 정수 값을 가지며 압축 레벨을 제어한다. 1은 가장 빠르지만 낮은 압축률을 제공하고 9(기본 값)는 가장 느리지만 높은 압축률을 제공한다. fileobj는 기존 파일 객체를 나타낸다. 주어질 경우 filename이라는 이름을 가지는 파일 대신에 주어진 객체를 사용한다.

open(filename [, mode [, compresslevel]])

GzipFile(filename, mode, compresslevel)과 동일하다. 기본 모드는 ‘rb’이며 기본 compresslevel은 9이다.

Note

- GzipFile 객체에 close() 메서드를 호출해도 fileobj로 지정한 파일을 닫지 않는다. 그렇기 때문에 압축된 데이터 뒤에 추가 정보를 기록할 수 있다.
- 유닉스에 있는 compress 프로그램으로 압축된 파일은 지원되지 않는다.
- 이 모듈은 zlib 모듈을 필요로 한다.

shutil

shutil 모듈은 복사, 삭제, 이름 변경 등 고수준 파일 연산을 수행하는 데 사용한다. 이 모듈에 있는 함수들은 실제 파일과 디렉터리에 대해서만 작동한다. 이름 있는 파일(named pipe), 블록 장치(block device) 같은 파일 시스템에서 특수한 종류의 파일에 대해서는 사용할 수 없다. 또한 이 함수들은 파일의 고급 메타데이터(예를 들어, 자원 포크(resource fork), 생성자 코드(creator code) 등)를 올바르게 처리하지 못할 때도 있다.

copy(src, dst)

파일 권한(permission)을 유지하면서 src 파일을 파일 또는 디렉터리 dst로 복사한다. src와 dst는 문자열이다.

copy2(src, dst)

copy()와 유사하지만 최근 접근(last access) 시간과 수정(modification) 시간도 복사한다.

copyfile(src, dst)

src의 내용을 dst로 복사한다. src과 dst는 문자열이다.

copyfileobj(f1, f2 [, length])

열린 파일 객체 f1의 모든 데이터를 파일 객체 f2로 복사한다. length는 사용할 가장 큰 버퍼 크기를 명시한다. 음수 값은 데이터 전체를 한 번의 연산으로 복사하려고 시도한다(즉, 모든 데이터를 하나의 데어리로 읽은 다음 쓴다).

copymode(src, dst)

src의 권한 비트(permission bit)를 dst로 복사한다.

copystat(src, dst)

src의 권한 비트, 최근 접근 시간, 수정 시간을 dst로 복사한다. dst의 내용, 소유자, 그룹은 변경하지 않는다.

copytree(src, dst, symlinks [, ignore])

src부터 시작하여 모든 디렉터리를 재귀적으로 복사한다. 목적 디렉터리 dst는 새로 생성된다(이미 존재하면 안 된다). 개별 파일들은 copy2()를 사용하여 복사한다. symlinks가 참이면 원본 트리의 심벌릭 링크(symbolic link)들은 새로운 트리의 심벌릭 링크들로 표현된다. symlinks가 거짓이거나 생략되어 있으면 심벌릭 링크가 가리키는 파일이 새로운 디렉터리 트리로 복사된다. ignore는 특정한 파일을 걸러내는 데 사용할 옵션인 함수이다. 이 함수는 디렉터리 이름과 디렉터리 내용 목록을 받아야 한다. 그리고 무시할 파일 이름 목록을 반환해야 한다. 복사 과정에서 에러가 발생할 경우 모아 놓았다가 마지막에 Error 예외를 발생시킨다. 이때 발생한 모든 에러를 담은 (srcname, dstname, exception) 튜플 리스트를 예외의 인수로 받게 된다.

ignore_pattern(pattern1, pattern2, ...)

pattern1, pattern2 등으로 지정한 glob 스타일의 패턴에 매칭되는 파일들을 무시하는 함수를 생성한다. 반환되는 함수는 두 개의 인수를 입력받는데 첫 번째 인수는 디렉터리 이름, 두 번째 인수는 디렉터리 내용 목록이다. 반환되는 함수는 무시할 파일 이름 목록을 반환한다. 반환되는 함수는 앞서 언급한 copytree() 함수의 ignore 매개변수로서 주로 사용한다. os.walk() 함수와 관련된 연산을 수행하는 데에도 사용할 수 있다.

move(src, dst)

파일 또는 디렉터리 src을 dst로 이동시킨다. 다른 파일 시스템으로 이동시킬 경우 재귀적으로 src를 복사한다.

rmtree(path [, ignore_errors [, onerror]])

전체 디렉터리 트리를 삭제한다. ignore_errors가 참이면 예리를 무시한다. 그렇지 않으면 예리는 onerror 함수(제공하면)에 의해서 처리된다. onerror에 지정한 함수는 반드시 세 개의 매개변수(func, path, excinfo)를 입력으로 받아야 한다. func은 예리(os.remove() 또는 os.rmdir())를 발생시킨 함수이고 path는 함수에 전달된 경로 이름이며 excinfo는 sys.exc_info()에 의해 반환된 예외 정보를 나타낸다. 예리가 발생했는데 onerror를 생략한 경우에는 예외가 발생한다.

tarfile

tarfile 모듈은 tar 아카이브 파일을 조작할 때 사용한다. 이 모듈을 사용하면 압축되어 있거나 압축되어 있지 않은 tar 파일을 읽거나 쓸 수 있다.

is_tarfile(name)

name이 이 모듈에서 읽을 수 있는 유효한 tar 파일인 것 같으면 True를 반환한다.

open([name [, mode [, fileobj [, bufsize]]]])

경로 name인 새로운 TarFile 객체를 생성한다. mode는 어떻게 tar 파일을 열 것인지를 나타내는 문자열이다. mode 문자열은 파일 모드(filemode)와 압축 방법(compression)의 조합인 ‘filemode[compression]’로서 표현한다. 다음은 유효한 조합을 보여준다.

모드	설명
'r'	읽기용으로 연다. 파일이 압축되어 있으면 알아서 압축을 해제한다. 기본으로 사용되는 모드이다.
'r:'	압축 없이 읽기용으로 연다.
'r:gz'	gzip 압축을 사용해서 읽기용으로 연다.
'r:bz2'	bzip2 압축을 사용해서 읽기용으로 연다.
'a', 'a:'	압축 없이 추가용으로 연다.
'w', 'w:'	압축 없이 쓰기용으로 연다.
'w:gz'	gzip 압축을 사용해서 쓰기용으로 연다.
'w:bz2'	bzip2 압축을 사용해서 쓰기용으로 연다.

다음 모드들은 순차 I/O 접근(임의 탐색random seek은 허용되지 않는다)만 허용하는 TarFile 객체를 생성할 때 사용한다.

모드	설명
'r '	압축되지 않은 블록 스트림을 읽기용으로 연다.
'r gz'	gzip로 압축된 스트림을 읽기용으로 연다.
'r bz2'	bzip2로 압축된 스트림을 읽기용으로 연다.
'w '	압축되지 않은 블록 스트림을 쓰기용으로 연다.
'w gz'	gzip로 압축된 스트림을 쓰기용으로 연다.
'w bz2'	bzip2로 압축된 스트림을 쓰기용으로 연다.

매개변수 fileobj를 지정했다면 fileobj는 반드시 열린 file 객체이어야 한다. fileobj가 주어지면 name으로 지정한 파일 대신 사용된다. bufsize는 tar 파일에 사용할 블록 크기를 나타낸다. 기본은 20*512 바이트이다.

open()에서 반환되는 TarFile의 인스턴스 t에는 다음의 메서드와 속성이 있다.

t.add(name [, arcname [, recursive]])

새로운 파일을 tar 아카이브에 추가한다. name은 아무 파일(디렉터리, 심벌릭 링크 등) 이름이나 될 수 있다. arcname은 아카이브 파일 안에서 대신 사용할 이름을 지정한다. recursive는 디렉터리의 내용을 재귀적으로 추가할지 여부를 나타내는 불리언 플래그다. 기본 값은 True이다.

t.addfile(tarinfo [, fileobj])

새로운 객체를 tar 아카이브에 추가한다. tarinfo는 아카이브 멤버와 관련된 정보를 담은 TarInfo 구조이다. fileobj는 데이터를 읽어서 아카이브에 저장할 열린 file 객체이다. 읽을 데이터량은 tarinfo의 속성 size에 의해 결정된다.

t.close()

tar 아카이브를 닫는다. 아카이브가 쓰기용으로 열린 것으면 영 값만 담은 두 개의 블록을 끝에 쓴다.

t.debug

생성될 디버깅 정보의 양을 제어한다. 0은 디버깅 메시지를 출력하지 않고 3은 모든 디버깅 메시지를 출력한다. 메시지는 sys.stderr에 쓰여진다.

t.dereference

이 속성을 True로 설정하면 심벌릭 링크와 하드 링크에 대해 역참조(dereference)

가 수행되어 참조된 파일의 전체 내용을 아카이브에 추가한다. False로 설정하면 링크만 추가한다.

t.errorlevel

아카이브의 멤버를 추출할 때 에러를 어떻게 처리할지를 결정한다. 이 속성을 0으로 설정하면 에러를 무시한다. 1로 설정하면 에러는 OSError 또는 IOError 예외를 발생시킨다. 2로 설정하면 치명적이지 않은 에러가 추가로 TarError 예외를 발생시킨다.

t.extract(member [, path])

아카이브에서 멤버를 추출하여 현재 디렉터리에 저장한다. member는 아카이브 멤버 이름이거나 TarInfo 인스턴스일 수 있다. path는 다른 목적 디렉터리를 지정하는 데 사용한다.

t.extractfile(member)

아카이브에서 멤버를 추출한 후 read(), readline(), readlines(), seek(), tell() 연산으로 내용을 읽을 수 있는 파일과 유사한 읽기 전용 객체를 반환한다. member는 아카이브 멤버 이름이거나 TarInfo 인스턴스일 수 있다. member가 링크를 참조하면 링크의 대상을 열려고 시도한다.

t.getmember(name)

아카이브 멤버 name을 찾아서 이 멤버에 관한 정보를 담은 TarInfo 객체를 반환한다. 아카이브 멤버가 존재하지 않으면 KeyError가 발생한다. 멤버 이름이 아카이브 내에서 한 번 이상 나타난다면 가장 마지막 항목의 정보(더 최근의 것이라고 가정하여)를 반환한다.

t.getmembers()

아카이브의 모든 멤버에 대해 TarInfo 객체 리스트를 반환한다.

t.getnames()

모든 아카이브 멤버의 이름 목록을 반환한다.

t.gettarinfo([name [, arcname [, fileobj]]])

파일 시스템의 파일 name 또는 열린 파일 객체 fileobj에 대응하는 TarInfo 객체를 반환한다. arcname은 아카이브 안에서 대신 사용되는 이름을 나타낸다. 이 함

수는 add() 와 같은 메서드에서 사용하려고 적절히 TarInfo 객체를 생성할 때 주로 사용한다.

t.ignore_zeros

이 속성을 True로 설정하면, 아카이브를 읽을 때 빈 블록을 건너뛴다. False로 설정하면(기본 값) 빈 블록은 아카이브의 끝을 알린다. 손상된 아카이브를 읽을 때 이 속성을 True로 설정할 수 있다.

t.list([verbose])

아카이브의 모든 내용을 sys.stdout으로 출력한다. verbose는 출력 내용의 상세 함을 결정한다. verbose를 False로 설정하면 아카이브 이름만 출력한다. 그 외의 경우(기본 설정)에는 모든 정보가 출력된다.

t.next()

아카이브의 멤버들에 대해 반복을 수행하기 위한 메서드. 다음 아카이브 멤버의 TarInfo 구조 또는 None을 반환한다.

t.posix

이 속성을 True로 설정하면 tar 파일은 POSIX 1003.1-1990 표준에 따라 생성된다. 이 경우 파일 이름의 길이와 파일 크기에 제약이 있다(파일 이름은 반드시 256자 이하여야 하고 파일 크기는 8GB 이하여야 한다). 이 속성을 False로 설정하면 아카이브는 이러한 제약 사항을 풀어준 GNU 확장에 따라 생성된다. 기본 값은 False이다.

앞에 나온 많은 메서드에서 TarInfo 인스턴스를 사용한다. 다음 표는 TarInfo 인스턴스 ti의 메서드와 속성을 보여준다.

속성	설명
ti.gid	그룹 아이디
ti.gname	그룹 이름
ti.isblk()	객체가 블록 장치인 경우 True를 반환한다.
ti.ischr()	객체가 문자 장치인 경우 True를 반환한다.
ti.isdev()	객체가 장치(문자, 블록, FIFO)인 경우 True를 반환한다.
ti.isdir()	객체가 디렉터리인 경우 True를 반환한다.
ti.isfifo()	객체가 FIFO인 경우 True를 반환한다.
ti.isfile()	객체가 일반 파일인 경우 True를 반환한다.
ti.islnk()	객체가 하드 링크이면 True를 반환한다.
ti.isreg()	isfile()와 동일하다.
ti.issym()	객체가 심벌릭 링크이면 True를 반환한다.
ti.linkname	하드 링크 또는 심벌릭 링크의 대상 파일 이름

<code>ti.mode</code>	권한 비트
<code>ti.mtime</code>	최근 수정 시간
<code>ti.name</code>	아카이브 멤버 이름
<code>ti.size</code>	바이트 크기
<code>ti.type</code>	파일 타입으로 상수 REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FiFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE 중 하나
<code>ti.uid</code>	사용자 아이디
<code>ti.uname</code>	사용자 이름

예외

다음은 tarfile 모듈에서 정의된 예외들이다.

TarError

다른 모든 예외의 기본 클래스.

ReadError

tar 파일을 여는 도중 에러가 발생했을 때 생성된다(예를 들어, 유효하지 않은 파일을 열려고 할 때).

CompressionError

데이터를 압축해제 할 수 없을 때 발생한다.

StreamError

스트림처럼 작동하는 TarFile 객체에 대해서 지원되지 않는 연산을 시도한 경우 발생한다(예를 들어, 임의 접근이 필요한 연산).

ExtractError

추출 과정에서 발생한 치명적이지 않은 에러에 대해 예외를 발생시킨다(errorlevel이 2로 설정되어 있을 경우).

예

```
# tar 파일을 열고 그 안에 파일을 추가한다.
t = tarfile.open("foo.tar","w")
t.add("README")
import glob
for pyfile in glob.glob("*.py"):
    t.add(pyfile)
t.close()
```

```
# tar 파일을 열고 모든 멤버에 대해 반복을 수행한다.
t = tarfile.open("foo.tar")
for f in t:
    print("%s %d" % (f.name, f.size))

# tar 파일을 훑어보면서 "README" 파일들의 내용을 출력한다.
t = tarfile.open("foo.tar")
for f in t:
    if os.path.basename(f.name) == "README":
        data = t.extractfile(f).read()
        print("**** %s ****" % f.name)
```

tempfile

tempfile 모듈은 임시 파일 이름과 파일을 생성할 때 사용된다.

`mkdtemp([suffix [,prefix [, dir]]])`

호출 프로세스의 소유자만 접근할 수 있는 임시 디렉터리를 생성하고 이 디렉터리에 대한 절대 경로를 반환한다. suffix는 디렉터리 이름 뒤에 추가로 붙일 접미사를 나타내고 prefix는 디렉터리 이름 앞에 추가로 붙일 접두사를 나타내며 dir는 임시 디렉터리가 생성될 곳을 나타낸다.

`mkstemp([suffix [,prefix [, dir [,text]]]])`

임시 파일을 생성하고 튜플(fd, pathname)을 반환한다. fd는 os.open에서 반환된 정수 파일 기술자이고 pathname은 파일의 절대 경로이다. suffix는 파일 이름에 붙일 추가 접미사이고 prefix는 파일 이름에 추가할 접두사이다. dir는 파일이 생성될 곳을 나타내고 text는 파일을 텍스트 모드 또는 이진 모드(기본 값)으로 열지 여부를 나타내는 불리언 플래그이다. 파일 생성 과정은 시스템이 os.open()에서 O_EXCL 플래그를 지원하는 경우 원자적(atomic)이고 안전(secure)하다.

`mktemp([suffix [, prefix [,dir]]])`

고유한 임시 파일 이름을 반환한다. suffix는 파일 이름에 붙일 추가 접미사이고 prefix는 파일 이름의 시작 부분에 추가할 접두사이며 dir는 파일이 생성될 곳을 나타낸다. 이 함수는 고유한 파일 이름을 생성할 뿐 실제로 임시 파일을 생성하거나 열지 않는다. 이 함수는 파일이 실제로 열리기 전에 이름을 생성하기 때문에 잠재적인 보안 문제를 야기한다. 이를 해결하려면 mkstemp()를 대신 사용하는 것을 고려해야 한다.

gettempdir()

임시 파일들이 생성된 곳의 디렉터리를 반환한다.

gettempprefix()

임시 파일을 생성하는 데 사용되는 접두사를 반환한다. 반환되는 접두사에 디렉터리는 포함되지 않는다.

TemporaryFile([mode [, bufsize [, suffix [,prefix [, dir]]]]])

`mkstemp()`를 사용하여 임시 파일을 생성하고 보통의 파일 객체와 동일한 메서드를 지원하는 파일과 유사한 객체를 반환한다. `mode`는 파일 모드를 나타내며 기본 값은 ‘w+b’이다. `bufsize`는 버퍼링 작동 방식을 지정하며 `open()` 함수와 의미가 동일하다. `suffix`, `prefix`, `dir`는 `mkstemp()`와 의미가 동일하다. 반환되는 객체는 이 객체의 `file` 속성을 통해 접근할 수 있는 파일 객체를 둘러싼 래퍼일 뿐이다. 이 함수로 생성된 파일은 임시 파일 객체가 파괴될 때 자동으로 파괴된다.

NamedTemporaryFile([mode [, bufsize [, suffix [,prefix [, dir [, delete]]]]]])

`TemporaryFile()`와 유사하게 임시 파일을 생성하며 파일 시스템에서 생성된 임시 파일 이름을 볼 수 있다. 이 이름은 반환되는 파일 객체의 `name` 속성을 통해 얻을 수 있다. 일부 시스템에서는 임시 파일이 닫힐 때까지 이 이름을 사용하여 파일을 다시 열 수 없다. `delete` 매개변수를 `True`로 설정할 경우(기본 값) 닫는 순간 임시 파일이 지워진다.

SpoooledTemporaryFile([max_size [, mode [, bufsize [, suffix [, prefix [, dir]]]]]])

크기가 `max_size`를 넘지 않는 선까지 파일 내용을 전부 메모리에 가지고 있다는 점을 제외하고는 `TemporaryFile` 같은 임시 파일을 생성한다. 파일 내용을 파일 시스템으로 내보낼 필요가 있을 때까지 `StringIO` 객체에 담아둔다. `fileno()` 메서드가 사용되는 저수준 IO 연산이 수행되면 메모리 내용이 `TemporaryFile` 객체로 접근할 수 있는 적절한 임시 파일에 즉시 쓰여진다. `SpoooledTemporaryFile`에 의해 반환되는 파일 객체는 그 내용을 파일 시스템에 강제로 쓰는 데 사용할 수 있는 `rollover()` 메서드를 가진다.

임시 이름을 생성하는 데 두 전역 변수가 사용된다. 원한다면 새로운 값을 할당할 수 있다. 기본 값은 시스템마다 다르다.

변수	설명
<code>tempdir</code>	<code>mktemp()</code> 에 의해 반환되는 파일 이름에서 디렉터리 부분
<code>template</code>	<code>mktemp()</code> 에 의해 생성되는 파일 이름의 접두사. 고유한 파일 이름을 생성하기 위해 삽진수 숫자들로 구성되는 문자열이 <code>template</code> 에 덧붙여진다.

Note

기본으로 `tempfile` 모듈은 몇몇 표준적인 위치를 검사해보고 파일을 생성한다. 예를 들어, 유닉스에서는 파일이 `/tmp`, `/var/tmp`, `/usr/tmp` 중 한 곳에 생성된다. 윈도에서는 `C:\TEMP`, `C:\WTEMP`, `C:\WTMP` 중 한 곳에 생성된다. 이 디렉터리들은 `TMPDIR`, `TEMP` 또는 `TMP` 환경 변수를 설정함으로써 덮어쓸 수 있다. 어떤 이유로 인해 보통의 위치에 파일을 생성할 수 없는 경우 파일은 현재 작업 디렉터리에 생성된다.

zipfile

`zipfile` 모듈은 널리 알려진 zip 포맷(원래는 PKZIP으로 알려져 있었다) 현재 다양한 프로그램에서 지원된다)으로 인코딩된 파일을 조작하는 데 사용한다. zip 파일은 파일썬에서 많이 사용되며 주로 패키징을 하기 위한 목적으로 사용된다. 예를 들어, 파일썬 소스 코드를 담은 zip 파일이 `sys.path`에 추가되어 있으면 zip 파일에 들어 있는 파일들을 `import`를 통해 로드할 수 있다(직접 사용할 일은 없는 `zipimport` 라이브러리 모듈에서 이 기능을 구현하고 있다). `.egg` 파일(`setuptools` 확장 기능을 통해 생성되는)로 배포되는 패키지도 알고 보면 단순히 zip 파일이다(`.egg` 파일은 실제로 zip 파일에 메타데이터를 조금 추가한 것이다).

`zipfile` 모듈에는 다음 함수와 클래스들이 정의되어 있다.

`is_zipfile(filename)`

`filename` 유효한 zip 파일인지를 검사한다. `filename` zip 파일이면 `True`를 반환한다. 그 외의 경우에는 `False`를 반환한다.

`ZipFile(filename [, mode [, compression [, allowZip64]]])`

이름이 `filename`인 zip 파일을 열어서 `ZipFile` 인스턴스를 반환한다. `mode`에서 ‘r’은 기존 파일에서 읽고 ‘w’는 파일을 비운 후 새로운 파일을 작성하며 ‘a’는 기존의 파일에 추가한다. ‘a’ 모드에서 `filename` zip 파일이면 새로운 파일이 추가된다. `filename` zip 파일이 아니면 아카이브의 내용이 파일 끝에 추가된다. `compression`

은 아카이브에 내용을 쓸 때 사용할 zip 압축 방식을 나타내며 ZIP_STORED 또는 ZIP_DEFLATED 중 하나가 될 수 있다. 기본 값은 ZIP_STORED이다. 인수 allowZip64는 크기가 2GB를 넘어서는 zip 파일을 생성할 때 필요한 ZIP64 확장 기능을 사용할 수 있게 한다. 기본 값은 False이다.

PyZipFile(filename [, mode[, compression [, allowZip64]]])

ZipFile과 유사하게 zip 파일을 열지만 파이썬 소스 파일을 아카이브에 추가할 때 사용할 수 있는 writepy() 메서드가 추가된 특별한 PyZipFile 인스턴스를 반환한다.

ZipInfo([filename [, date_time]])

아카이브 멤버에 관한 정보를 담을 새로운 ZipInfo 인스턴스를 직접 생성한다. 보통 ZipFile 인스턴스의 z.writestr() 메서드(조금 뒤에 설명)를 사용할 때를 제외하고는 이 함수를 호출할 일이 없다. filename과 date_time 인수는 아래에서 설명한 filename과 date_time 속성의 값을 지정한다.

ZipFile 또는 PyZipFile의 인스턴스 z는 다음 메서드와 속성을 지원한다.

z.close()

아카이브 파일을 닫는다. zip 파일의 레코드들을 내보내기 위해서 프로그램이 종료되기 전에 반드시 호출해야 한다.

z.debug

0(출력 없음)에서 3(모두 출력)까지 범위의 디버깅 수준.

z.extract(name [, path [, pwd]])

아카이브로부터 파일을 추출하여 현재 작업 디렉터리에 저장한다. name은 추출할 아카이브 멤버의 이름이거나 ZipInfo 인스턴스일 수 있다. path는 파일을 추출할 디렉터리이고 pwd는 암호화된 아카이브에 대해서 사용할 패스워드이다.

z.extractall([path [, members [, pwd]])])

아카이브의 모든 멤버를 현재 작업 디렉터리로 추출한다. path는 다른 디렉터리를 지정할 때 쓰고 pwd는 암호화된 아카이브에 대해서 쓴 패스워드이다. members는 추출할 멤버 목록이다. 이 목록은 반드시 namelist() 메서드(이후 설명)에 의해 반환되는 목록의 부분집합이어야 한다.

z.getinfo(name)

아카이브 멤버 이름에 관한 정보를 ZipInfo 인스턴스로 반환한다(곧 더 자세히 설명한다).

z.infolist()

아카이브의 모든 멤버에 대한 정보를 담은 ZipInfo 객체 리스트를 반환한다.

z.namelist()

아카이브 멤버 이름 리스트를 반환한다.

z.open(name [, mode [, pwd]])

이름이 name인 아카이브 멤버를 연 다음 그 내용을 읽는 데 사용할 수 있는 파일과 유사한 객체를 반환한다. name은 아카이브 멤버를 설명하는 문자열 또는 ZipInfo 인스턴스이다. mode는 파일 모드로서 반드시 읽기 전용 파일 모드인 ‘r’, ‘rU’, ‘U’ 중 하나이어야 한다. pwd는 암호화된 아카이브 멤버에 대해서 사용할 패스워드이다. 반환된 파일 객체는 read(), readline(), readlines() 메서드를 지원할 뿐만 아니라 for문을 통한 반복도 지원한다.

z.printdir()

아카이브 디렉터리를 sys.stdout으로 출력한다.

z.read(name [,pwd])

이름이 name인 아카이브 멤버의 내용을 읽어서 데이터를 문자열로 반환한다. name은 아카이브 멤버를 설명하는 문자열 또는 ZipInfo 인스턴스이다. pwd는 암호화된 아카이브 멤버에 사용할 패스워드이다.

z.setpassword(pwd)

아카이브로부터 암호화된 파일들을 추출하기 위해 사용할 기본 패스워드를 설정한다.

z.testzip()

아카이브의 모든 파일을 읽어서 CRC 체크섬(checksum)을 확인한다. 훼손된 첫 번째 파일을 반환하거나 손상된 파일이 없으면 None을 반환한다.

z.write(filename[, arcname[, compress_type]])

filename을 arcname 아카이브에 쓴다. compress_type은 압축 방식을 지정하

는 매개변수이며 ZIP_STORED 또는 ZIP_DEFLATED 중 하나이다. ZipFile() 또는 PyZipFile() 함수에 주어진 것이 기본으로 사용된다. 쓰기를 하려면 아카이브는 반드시 ‘w’ 또는 ‘a’ 모드로 열어야 한다.

z.writepy(pathname)

PyZipFile 인스턴스에만 있는 이 메서드는 파이썬 소스 코드(*.py 파일)를 zip 아카이브에 쓰는 데 사용하며 배포를 위해서 파이썬 응용 프로그램을 패키징하는 데 사용할 수 있다. pathname이 파일이면 반드시 .py로 끝나야 한다. pathname이 파일이면 대응하는 .pyo, .pyc, .py 파일 중 하나가 추가된다(이 순서대로). pathname이 디렉터리이고 이 디렉터리가 파이썬 패키지 디렉터리가 아닐 경우 모든 대응하는 .pyo, .pyc, .py 파일이 제일 상위 수준에 추가된다. 디렉터리가 패키지일 경우 파일들은 패키지 이름을 파일 경로로 하여 그 아래에 추가된다. 하위 디렉터리도 패키지 디렉터리이면 재귀적으로 추가된다.

z.writestr(arcinfo, s)

문자열 s를 zip 파일에 쓴다. arcinfo는 데이터를 저장할 아카이브 안의 파일 이름 이거나 파일 이름, 날짜, 시간 정보를 담은 ZipInfo 인스턴스가 될 수 있다.

ZipInfo(), z.getInfo(), z.infolist()에 의해서 반환되는 ZipInfo 인스턴스 i는 다음 속성들을 가진다.

속성	설명
i.filename	아카이브 멤버 이름
i.date_time	최종 수정 시간을 담은 튜플(year, month, day, hours, minutes, seconds). month와 day는 각각 1-12와 1-31의 범위에 속하는 숫자이다. 다른 값들은 0에서 시작한다.
i.compress_type	아카이브 멤버에 대한 압축 타입. 현재 ZIP_STORED와 ZIP_DEFLATE만 이 모듈에서 지원된다.
i.comment	아카이브 멤버 주석
i.extra	추가 파일 속성을 담는 데 사용되는 확장 필드 데이터. 여기에 저장되는 데이터는 파일이 생성된 시스템마다 다르다.
i.create_system	이 아카이브가 생성된 시스템을 나타내는 정수 코드. 0(MS-DOS FAT), 3(유닉스), 7(매킨토시), 10(윈도 NTFS)가 주로 사용된다.
i.create_version	zip 아카이브를 생성한 PKZIP 버전 코드
i.extract_version	이 아카이브를 추출하는 데 필요한 최소 버전
i.reserved	예약된 필드. 현재 0으로 설정되어 있음.
i.flag_bits	암호화나 압축 방식 등을 나타내는 데이터 인코딩을 설명하는 zip 플래그 비트

<code>i.volume</code>	파일 헤더의 블루 번호
<code>i.internal_attr</code>	아카이브 내부 구조를 나타냄. 하위 비트가 1이면 데이터는 ASCII 텍스트이다. 아니면 이진 데이터이다.
<code>i.external_attr</code>	운영체제에 의존적인 외부 파일 속성
<code>i.header_offset</code>	파일 헤더로의 바이트 오프셋
<code>i.file_offset</code>	파일 데이터 시작 부분으로의 바이트 오프셋
<code>i.CRC</code>	압축되지 않은 파일의 CRC 체크섬
<code>i.compress_size</code>	압축된 파일 데이터의 크기
<code>i.file_size</code>	압축되지 않은 파일의 크기

Note

zip 파일의 내부 구조에 관한 자세한 정보는 <http://www.pkware.com/appnote.html>에 있는 PKZIP Application Note에서 찾아볼 수 있다.

 zlib

zlib 모듈은 zlib 라이브러리에 접근할 수 있게 함으로써 데이터 압축 기능을 제공한다.

adler32(string [, value])

string의 Adler-32 체크섬을 계산한다. value는 시작 값(여러 문자열이 연결되어 있는 경우 체크섬을 계산하는 데 사용함). value가 제공되지 않으면 고정 기본 값이 사용된다.

compress(string [, level])

string에 있는 데이터를 압축한다. level은 1에서 9 사이의 정수로서 압축 수준을 제어한다. 1은 낮은 압축률(가장 빠르다)을, 9는 높은 압축률(가장 느리다)을 의미한다. 기본 값은 6이다. 압축된 데이터를 담은 문자열을 반환하거나 에러가 발생한 경우 error 예외를 발생시킨다.

compressobj([level])

압축된 객체를 반환한다. level은 compress() 함수와 의미가 동일하다.

crc32(string [, value])

string의 CRC 체크섬을 계산한다. value가 주어지면 체크섬의 시작 값으로 사용

한다. 그렇지 않으면 고정 값을 사용한다.

decompress(string [, wbits [, bufsize]])

string에 있는 데이터의 압축을 해제한다. wbits는 윈도 버퍼(window buffer) 크기를 제어하고 bufsize는 출력 버퍼의 초기 크기를 나타낸다. 에러가 발생한 경우 error 예외를 발생시킨다.

decompressobj([wbits])

압축 객체를 반환한다. wbits 매개변수는 윈도 버퍼 크기를 제어한다.

압축 객체 c는 다음의 메서드를 가진다.

c.compress(string)

string을 압축한다. string에 있는 데이터의 일부에 대한 압축된 데이터를 담은 문자열을 반환한다. 출력 스트림을 생성하는 경우에는 이전에 호출한 c.compress()에 의해서 생성된 출력에 이 함수로 압축된 데이터를 이어서 출력해야 한다. 입력 데이터의 일부는 추후 처리를 위해 내부 버퍼에 저장될 수도 있다.

c.flush([mode])

남아 있는 모든 입력을 압축한 데이터를 담은 문자열을 반환한다. mode는 Z_SYNC_FLUSH, Z_FULL_FLUSH, Z_FINISH(기본 값)가 될 수 있다. Z_SYNC_FLUSH와 Z_FULL_FLUSH는 압축을 더 할 수 있게 하며 압축을 해제할 때 부분 에러 복구를 가능하게 한다. Z_FINISH는 압축 스트림을 닫는다.

압축해제 객체 d는 다음의 메서드와 속성을 가진다.

d.decompress(string [,max_length])

string의 압축을 해제하고 string의 일부 데이터에 대한 압축되지 않은 데이터를 담은 문자열을 반환한다. 출력 스트림을 생성하려면 이전에 호출된 decompress()에 의해 생성된 데이터에 이어 이 데이터를 출력해주어야 한다. 일부 입력 데이터는 추후 처리를 위해 내부 버퍼에 저장되기도 한다. max_length는 반환되는 데이터의 최대 크기를 나타낸다. 이 값을 초과할 경우 처리되지 않은 데이터는 d.unconsumed_tail 속성에 저장된다.

d.flush()

남은 모든 입력을 처리하여 얻은 압축이 풀린 데이터를 담은 문자열을 반환한다.

압축해제 객체는 이 함수를 호출한 후에 다시 사용할 수 없다.

d.unconsumed_tail

마지막 decompress() 호출 후에 아직 처리되지 않은 데이터를 담은 문자열. 버퍼 크기 제한으로 인해 압축해제를 단계적으로 수행해야 하는 경우에 이 속성은 데이터를 담고 있을 수 있다. 이어지는 decompress() 호출에 이 변수를 전달할 수 있다.

d.unused_data

압축된 데이터 뒤쪽에 남아 있는 추가 바이트들을 담은 문자열.

Note

zlib 라이브러리는 <http://www.zlib.net>에서 받을 수 있다.

19장

Python Essential Reference

운영체제 서비스

이 장에서는 다양한 운영체제 서비스에 대한 접근을 제공하는 모듈을 다룬다. 그 중에서도 저수준 I/O, 프로세스 관리, 운영체제 환경에 관련된 모듈을 중점적으로 다룬다. 설정 파일을 읽거나 로그 파일에 쓰는 등 시스템 프로그램을 작성할 때 자주 사용되는 모듈도 설명한다. 18장에서는 파일과 파일 시스템을 다루는 데 사용 할 수 있는 고수준 모듈을 살펴보았다면 여기서는 더 저수준 내용을 다룬다.

파이썬에서 운영체제 관련 모듈은 대부분 POSIX 인터페이스에 기반한다. POSIX 는 핵심 운영체제 인터페이스를 정의하는 표준이다. 유닉스 시스템은 대부분 POSIX를 지원하고 윈도 같은 플랫폼에서도 이 인터페이스를 많은 부분 지원한다. 이 장 전체에 걸쳐서 특정 플랫폼에만 적용되는 함수나 모듈이 있다면 따로 언급하겠다. 유닉스 시스템에는 리눅스와 Mac OS X가 포함된다. 윈도 시스템은 특별한 언급이 없는 한 모든 버전의 윈도를 포함한다.

여기서 다루는 내용에 대해 추가 참고 자료를 원할지도 모르겠다. 브라이언 커니핸(Brian W. Kernighan)과 데니스 리치(Dennis M. Ritchie)가 쓴 《C 언어 프로그래밍(The C Programming Language)》 제2판(Prentice Hall, 1989)에는 이 장에서 다 른 많은 모듈의 기초가 되는 파일과 파일 기술자 및 저수준 인터페이스에 관한 정보를 잘 간추려 놓았다. 이미 기본 개념에 익숙한 독자들은 리처드 스티븐스(W. Richard Stevens)와 스티븐 레이고(Stephen Rago)가 쓴 《유닉스 환경에서 고급 프로그래밍》(Advanced Programming in the UNIX Environment) 제2판(Addison Wesley, 2005) 같은 책을 참고하면 된다. 일반적인 개념을 개략적으로 살펴보기 위

해 운영체제 관련 대학 교재를 보려는 생각을 할 수 있을 것이다. 그렇지만 가격도 비싸고 실제로 매일 옆에 두고 볼 것도 아니기 때문에 전산학을 전공하는 친구가 있다면 관련 책을 빌려서 보는 것이 나을 것이다.

commands

commands 모듈은 문자열로 된 간단한 시스템 명령을 실행하는 데 사용하며 결과를 문자열로 반환한다. 유닉스 시스템에서만 사용할 수 있다. 이 모듈에서 제공하는 기능은 유닉스 셸 스크립트에서 역따옴표()를 사용하는 것과 비슷하다. 예를 들어, `x = commands.getoutput('ls -l')`은 `x = `ls -l``와 비슷하다.

`getoutput(cmd)`

셀에서 cmd를 실행하고 해당 명령의 표준 출력과 표준 에러 스트림 두 가지의 출력 내용을 담은 문자열을 반환한다.

`getstatusoutput(cmd)`

2개 항목짜리 튜플 (status, output)을 반환하는 것을 제외하고 `getoutput()`과 비슷하다. `status`는 `os.wait()` 함수가 반환하는 종료 코드이고 `output`은 `getoutput()`이 반환하는 문자열이다.

Note

- 이 모듈은 파이썬 2에만 있다. 파이썬 3에서는 두 함수 모두 `subprocess` 모듈에 있다.
- 간단한 셸 연산을 수행하는 데 이 모듈을 사용할 수 있지만 거의 모든 경우에 `subprocess` 모듈을 사용해 하위 프로세스를 생성하고 출력 결과를 모으는 방식이 더 낫다.

참고

[subprocess\(495페이지\)](#)

ConfigParser, configparser

ConfigParser 모듈(파이썬 3에서는 `configparser`)은 윈도 INI 형식으로 된 .ini 설정 파일을 읽는 데 사용한다. .ini 파일은 이름 있는 구역들로 구성되며 다음과 같이 각 구역은 자신만의 변수 대입문들을 담는다.

```
# 주석
; 주석
[구역1]
이름1 = 값1
이름2 = 값2

[구역2]
; 값을 대입하는 다른 문법
이름1: 값1
이름2: 값2
...
```

ConfigParser 클래스

다음 클래스는 설정 변수들을 관리하는 데 사용한다.

ConfigParser([defaults [, dict_type]])

새로운 ConfigParser 인스턴스를 생성한다. defaults는 설정 변수에서 참조할 수 있는 값들의 사전이며 설정 변수에서 '%(key)s' 같은 문자열 포맷 지정자로 값을 참조할 수 있다. 여기서 key는 defaults에 있는 키이다. dict_type은 내부적으로 설정 변수들을 저장하는 데 사용할 사전의 타입을 나타낸다. 기본 값은 dict(내장 사전)이다.

ConfigParser 인스턴스 c는 다음 연산들을 지원한다.

c.add_section(section)

저장된 설정 매개변수에 새로운 구역을 추가한다. section은 구역 이름을 담은 문자열이다.

c.defaults()

기본 값을 담은 사전을 반환한다.

c.get(section, option [, raw [, vars]])

구역 section에 있는 옵션 option의 값을 문자열로 반환한다. 반환된 문자열은 기본으로 '%(option)s' 같은 포맷 문자열을 확장하는 채움 단계(interpolation step)를 통해 처리된다. 이 경우 option은 같은 구역에 있는 구성 옵션의 이름이거나 ConfigParser에 제공된 defaults 매개변수에 들어 있는 기본 값 중 하나가 될 수 있다. raw는 채움 기능을 사용하지 않고 옵션을 그대로 반환하게 하는 불리언 플래그이다. vars는 '%' 확장에 사용할 추가 값을 담은 옵션인 사전이다.

c.getboolean(section, option)

구역 section의 옵션 option의 값을 불리언 값으로 변환하여 반환한다. “0”, “true”, “yes”, “no”, “on”, “off” 같은 값들을 모두 이해하며 대소문자를 구별하지 않는다. 값 채움을 항상 수행한다(c.get() 참고).

c.getfloat(section, option)

구역 section의 옵션 option의 값을 값채움을 사용하여 실수로 변환하여 반환한다.

c.getint(section, option)

구역 section의 옵션 option의 값을 값채움을 사용하여 정수로 변환하여 반환한다.

c.has_option(section, option)

구역 section에 옵션 이름 option이 있으면 True를 반환한다.

c.has_section(section)

section이란 이름의 구역이 있으면 True를 반환한다.

c.items(section [, raw [, vars]])

구역 section의 (option, value) 리스트를 반환한다. raw는 True로 설정할 경우 채움 기능을 사용하지 않게 하는 불리언 플래그이다. vars는 ‘%’ 확장에서 사용할 추가 값들을 담은 사전이다.

c.options(section)

구역 section의 모든 옵션을 담은 리스트를 반환한다.

c.optionxform(option)

옵션 이름 option을 해당 옵션을 참조할 때 사용할 수 있는 문자열로 변환한다. 기본으로 소문자로 변환한다.

c.read(filenames)

여러 파일에서 설정 옵션을 읽는다. filenames는 읽을 파일의 이름을 담은 단일 문자열이거나 파일 이름들을 담은 리스트이다. 주어진 파일을 찾을 수 없으면 해당 파일은 무시한다. 구성 파일을 여러 곳에서 찾지만 해당 구성 파일이 있을지 없을지 모르는 경우에 사용하기 좋다. 성공적으로 파싱한 파일 이름들을 담은 리스트를 반환한다.

c.readfp(fp [, filename])

이미 열린 파일과 비슷한 객체 fp에서 설정 옵션을 읽는다. filename은 fp의 파일 이름을 지정한다(있는 경우). 기본으로 파일 이름을 fp.name에서 얻으며 이 속성이 정의되어 있지 않으면 파일 이름은 ‘`<??>`’로 설정된다.

c.remove_option(section, option)

구역 section에서 option을 제거한다.

c.remove_section(section)

구역 section을 제거한다.

c.sections()

모든 구역의 이름을 담은 리스트를 반환한다.

c.set(section, option, value)

구역 section의 설정 옵션 option을 value로 설정한다. value는 문자열이어야 한다.

c.write(file)

현재 보관 중인 모든 데이터를 file에 쓴다. file은 이미 열린 파일과 비슷한 객체이다.

예

ConfigParser 모듈은 자주 간과되지만 아주 복잡한 사용자 설정이나 런타임 환경이 필요한 프로그램을 제어하는 데 매우 유용하게 사용할 수 있다. 예를 들어, 큰 프레임워크 안에서 쓰일 컴포넌트를 개발하는 경우 설정 파일을 사용하면 런타임 매개변수를 손쉽게 설정할 수 있다. 또한 설정 파일을 사용하면 프로그램에서 argparse 모듈을 사용해 많은 수의 명령줄 옵션을 읽는 것보다 훨씬 낫다. 파이썬 소스 파일에서 설정 데이터를 읽는 것과 설정 파일에서 읽는 것 사이에는 미묘하지만 중요한 차이가 있다.

다음은 ConfigPaser 모듈의 몇 가지 흥미로운 기능을 보여주는 예들이다. 먼저 sample.ini 파일을 살펴보자.

```
# appconfig.ini
# 거대한 응용 프로그램을 위한 설정 파일
```

```
[output]
LOGFILE=%(LOGDIR)s/app.log
LOGGING=on
LOGDIR=%(BASEDIR)s/logs

[input]
INFILE=%(INDIR)s/initial.dat
INDIR=%(BASEDIR)s/input
```

다음 코드는 구성 파일을 읽고 몇몇 변수들의 기본 값을 지정하는 방법을 보여 준다.

```
from configparser import ConfigParser # 파이썬 2에서는 from
ConfigParser를 사용한다.

# 기본 변수 설정들을 담은 사전
defaults = {
    'basedir' : '/Users/beazley/app'
}

# ConfigParser 객체를 생성하고 .ini 파일을 읽는다.
cfg = ConfigParser(defaults)
cfg.read('appconfig.ini')
```

설정 파일을 읽었다면 get() 메서드를 사용해 옵션 값을 얻을 수 있다. 다음 예를 보자.

```
>>> cfg.get('output','logfile')
'/Users/beazley/app/logs/app.log'
>>> cfg.get('input','infile')
'/Users/beazley/app/input/initial.dat'
>>> cfg.getboolean('output','logging')
True
>>>
```

여기서 몇 가지 흥미로운 특징을 볼 수 있다. 먼저, 설정 매개변수는 대소문자를 구별하지 않는다. 따라서 프로그램에서 매개변수 'logfile'을 읽을 때 설정 파일에서 'logfile', 'LOGFILE', 'LogFile' 중 어느 것을 사용하든지 상관없다. 두 번째로 앞에 나온 파일에서 볼 수 있듯이 설정 매개변수는 '%(BASEDIR)s'나 '%(LOGDIR)s' 같은 변수 치환을 담을 수 있다. 이러한 치환을 수행할 때도 대소문자를 구별하지 않는다. 게다가 설정 매개변수들의 순서도 중요하지 않다. 예를 들어, appconfig.ini에서 LOGFILE 매개변수는 나중에 정의된 LOGDIR 매개변수를 참조하고 있다. 마지막으로 설정 파일의 값은 파이썬 문법이나 데이터 타입과 맞지 않더라도 그 값이 정확하게 해석된다. 예를 들어, LOGGING 매개변수의 'on' 값은 cfg.getboolean() 메서드에서 True로 해석된다.

설정 파일들을 합칠 수도 있다. 예를 들어, 사용자 설정을 담은 별개의 설정 파일이 있다고 하자.

```
; userconfig.ini
;
; 사용자별 설정

[output]
Logging=off

[input]
BASEDIR=/tmp
```

다음과 같이 앞에서 읽은 설정 파일의 내용에 이 파일의 내용을 합칠 수 있다.

```
>>> cfg.read('userconfig.ini')
['userconfig.ini']
>>> cfg.get('output','logfile')
'/Users/beazley/app/logs/app.log'
>>> cfg.get('output','logging')
'off'
>>> cfg.get('input','infile')
'/tmp/input/initial.dat'
>>>
```

여기서 새로 불러온 설정은 미리 정의된 매개변수들을 대체할 수 있음을 볼 수 있다. 게다가, 다른 설정 매개변수의 변수 치환에서 사용된 설정 매개변수를 변경하면 이 변경이 올바르게 전달된다. 예를 들어, input 구역에서 BASEDIR를 새로 설정하였는데 이것이 같은 구역에 있는 이전에 정의된 구성 매개변수 INFILE에 영향을 미쳤다. 이러한 작동 방식은 단순히 파이썬 스크립트에서 프로그램 매개변수를 정의 할 때와 미묘하지만 중요하게 다른 점이다.

Note

ConfigParser 자리에 다음 두 클래스를 사용할 수 있다. RawConfigParser 클래스는 ConfigParser의 모든 기능을 제공하지만 변수 치환을 수행하지 않는다. SafeConfigParser 클래스는 ConfigParser와 동일한 기능을 제공하지만 설정 값 자체에 채움 기능 구현에 사용되는 특수한 포맷 문자(예를 들어, '%')가 들어 있는 경우에 발생할 수 있는 문제를 해결한다.

datetime

datetime 모듈은 날짜와 시간을 표현하고 조작하는 데 사용할 수 있는 다양한 클래스들을 제공한다. 이 모듈의 많은 부분이 날짜와 시간 정보를 생성하고 출력하

는 다양한 방법을 제공하는 것에 초점이 맞추어져 있다. 다른 주요한 기능으로 시간 변화량을 비교하고 계산하는 수학 연산들이 있다. 날짜 조작과 관련한 주제는 복잡하기 때문에 이 모듈의 설계와 관련한 배경 지식을 쌓기 위해 파이썬 온라인 문서를 꼭 읽어보기 바란다.

date 객체

date 객체는 연, 월, 일로 구성되는 간단한 날짜를 표현한다. 다음 네 개의 함수들을 날짜를 생성하는 데 사용한다.

`date(year, month, day)`

새로운 date 객체를 생성한다. year는 `datetime.MINYEAR`에서 `datetime.MAXYEAR`의 범위 안에 있는 정수이다. month는 1에서 12 사이의 정수이고 day는 1에서 주어진 월의 마지막 날까지의 범위 안에 있는 정수이다. 반환되는 date 객체는 변경할 수 없으며 주어진 인수의 값에 해당하는 year, month, day 속성을 가진다.

`date.today()`

현재 날짜에 해당하는 date 객체를 반환하는 클래스 메서드.

`date.fromtimestamp(timestamp)`

타임스탬프 timestamp에 해당하는 date 객체를 반환하는 클래스 메서드. timestamp는 `time.time()` 함수에서 반환되는 값이다.

`date.fromordinal(ordinal)`

최소 날짜로부터 ordinal 개수의 날이 지난 날짜에 해당하는 date 객체를 반환하는 클래스 메서드(1년 1월 1일은 ordinal 값 1을 가지고 2006년 1월 1일은 ordinal 값 732312를 가진다).

다음 클래스 속성들은 date 인스턴스의 최대 최솟값과 해상력을 나타낸다.

`date.min`

표현할 수 있는 가장 이른 날짜를 나타내는 클래스 속성(`datetime.date(1, 1, 1)`).

`date.max`

표현할 수 있는 가장 늦은 날짜를 나타내는 클래스 속성(`datetime.date(9999, 12, 31)`).

date.resolution

동일하지 않은 날짜 객체 사이의 나눌 수 있는 가장 작은 차이(datetime.timedelta(1)).

date 인스턴스 d는 읽기 전용 d.year, d.month, d.day 속성을 가지며 추가로 다음 메서드들을 제공한다.

d.ctime()

time.ctime() 함수에서 사용되는 것과 같은 형식으로 데이터를 표현한 문자열을 반환한다.

d.isocalendar()

날짜를 튜플 (iso_year, iso_week, iso_weekday)로 반환한다. iso_week는 1에서 53까지 범위를 가지며 iso_weekday는 1(월요일)에서 7(일요일)의 범위를 가진다. 첫 번째 iso_week는 한 해에서 목요일을 포함하는 첫 주를 나타낸다. 튜플의 세 값이 가질 수 있는 범위는 ISO 8601 표준에 따라 결정된다.

d.isoformat()

ISO 8601 형식인 ‘YYYY-MM-DD’로 표현된 날짜를 담은 문자열을 반환한다.

d.isoweekday()

1(월요일)에서 7(일요일) 범위 내의 요일을 반환한다.

d.replace([year [, month [, day]]])

지정한 값으로 하나 이상의 구성 요소를 대체한 새로운 date 객체를 반환한다. 예를 들어 d.replace(month=4)는 월이 4로 바뀐 새로운 날짜를 반환한다.

d.strftime(format)

time.strftime() 함수에서 사용되는 규칙과 동일하게 포맷된 날짜를 나타내는 문자열을 반환한다. 이 함수는 1900년 이후의 날짜에만 사용할 수 있다. date 객체에 없는 구성 요소(시, 분 등)에 대해서는 포맷 코드를 사용하지 않아야 한다.

d.timetuple()

time 모듈에 있는 함수에서 사용하기 적합하도록 time.struct_time 객체를 반환한다. 시간(시, 분, 초)과 관련된 값은 0으로 설정된다.

d.toordinal()

d를 서수 값(ordinal value)으로 바꾼다. 1년 1월 1일은 서수 값 1을 가진다.

d.weekday()

0(월요일)에서 6(일요일) 범위 내의 요일을 반환한다.

time 객체

time 객체는 시간을 시, 분, 초, 마이크로초로 표현하는 데 사용한다. 시간은 다음 생성자로 생성한다.

time(hour [, minute [, second [, microsecond [, tzinfo]]]])

$0 \leq \text{hour} < 24$, $0 \leq \text{minute} < 60$, $0 \leq \text{second} < 60$, $0 \leq \text{microsecond} < 1000000$ 인 시간을 나타내는 time 객체를 생성한다. tzinfo는 이 장의 나중에 설명할 tzinfo 클래스 인스턴스로 시간대(time zone) 정보를 제공한다. 반환되는 time 객체는 인수로 제공한 값을 담은 hour, minute, second, microsecond, tzinfo 속성을 가진다.

다음의 time 클래스 속성들은 time 인스턴스의 허용 가능한 시간 범위와 해상력을 나타낸다.

time.min

표현할 수 있는 최소 시간을 나타내는 클래스 속성(datetime.time(0, 0)).

time.max

표현할 수 있는 최대 시간을 나타내는 클래스 속성(datetime.time(23, 59, 59, 999999)).

time.resolution

동일하지 않은 time 객체 사이에 나눌 수 있는 가장 작은 차이(datetime.timedelta(0, 0, 1)).

time 객체 인스턴스 t는 t.hour, t.minute, t.second, t.microsecond, t.tzinfo 속성과 다음 메서드들을 가진다.

t.dst()

t.tzinfo.dst(None) 값을 반환한다. 반환되는 객체는 timedelta 객체이다. 시간대가 설정되어 있지 않으면 None을 반환한다.

t.isoformat()

‘HH:MM:SS,mmmmmm’으로 표현된 시간을 문자열로 반환한다. 마이크로초가 0이면 문자열에서 마이크로초를 나타내는 부분은 생략된다. 시간대 정보가 주어지면 시간에 오프셋(offset)이 추가될 수 있다(예로, ‘HH:MM:SS, mmmmmmm+HH:MM’).

t.replace([hour [, minute [, second [, microsecond [, tzinfo]]]]])

지정한 값으로 하나 이상의 구성 요소 값을 대체한 새로운 time 객체를 반환한다. 예를 들어, t.replace(second=30)는 초 필드를 30으로 바꾼 새로운 time 객체를 반환한다. 인수들은 앞서 살펴본 time() 함수에 제공되는 것과 같은 의미를 가진다.

t.strftime(format)

time 모듈의 time.strftime() 함수에서 사용되는 규칙에 따라 포맷을 지정한 문자열을 반환한다. 날짜 정보를 사용할 수 없기 때문에 시간 관련 포맷 코드만 사용해야 한다.

t.tzname()

t.tzinfo.tzname() 값을 반환한다. 시간대가 설정되어 있지 않으면 None을 반환한다.

t.utcoffset()

t.tzinfo.utcoffset(None) 값을 반환한다. 반환되는 객체는 timedelta 객체이다. 시간대가 설정되어 있지 않으면 None을 반환한다.

datetime 객체

datetime 객체는 날짜와 시간을 동시에 표현하는 데 사용한다. datetime 인스턴스는 여러 가지 방법으로 생성할 수 있다.

datetime(year, month, day [, hour [, minute [, second [, microsecond [, tzinfo]]]]])

날짜와 시간 객체의 모든 특징을 결합한 새로운 datetime 객체를 생성한다. 인수들은 date()와 time()에 제공되는 인수들과 같다.

datetime.combine(date, time)

date 객체 date와 time 객체 time의 내용을 합쳐 datetime 객체를 생성하는 클래스 메서드.

datetime.fromordinal(ordinal)

서수인 날(정수로 나타낸 datetime.min 아래 날 수)이 주어질 때 datetime 객체를 생성하는 클래스 메서드. 시간 요소들은 모두 0으로 설정되고 tzinfo는 None으로 설정된다.

datetime.fromtimestamp(timestamp [, tz])

time.time() 함수에서 반환되는 타임스탬프로부터 datetime 객체를 생성하는 클래스 메서드. tz는 tzinfo 인스턴스로 옵션인 시간대 정보이다.

datetime.now([tz])

현재 지역 날짜와 시간으로부터 datetime 객체를 생성하는 클래스 메서드. tz는 tzinfo 인스턴스로 옵션인 시간대 정보이다.

datetime.strptime(datestring, format)

format으로 지정한 날짜 포맷에 따라 날짜 문자열 datestring을 파싱하여 datetime 객체를 생성하는 클래스 메서드. 파싱은 time 모듈의 strftime() 함수에 의해 수행된다.

datetime.utcnow(timestamp)

time.gmtime()에서 반환되는 타임스탬프로부터 datetime 객체를 생성하는 클래스 메서드.

datetime.utcnow()

현재 협정 세계시(UTC: Universal Time Coordinated) 날짜와 시간으로부터 datetime을 생성하는 클래스 메서드.

다음 클래스 속성들은 허용되는 날짜 범위와 해상력을 설명한다.

datetime.min

표현할 수 있는 가장 이른 날짜 및 시간(datetime.datetime(1, 1, 1, 0, 0)).

datetime.max

표현할 수 있는 가장 늦은 날짜 및 시간(datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)).

datetime.resolution

동일하지 않은 datetime 객체 사이에 나눌 수 있는 가장 작은 차이(datetime.

`timedelta(0, 0, 1)).`

datetime 객체 인스턴스 d는 date와 time 객체에서 제공되는 메서드들을 가진다. 추가로 다음 메서드들을 사용할 수 있다.

d.astimezone(tz)

시간대 tz인 새로운 datetime 객체를 반환한다. 새로운 객체의 멤버들은 시간대 tz에서 같은 UTC 시간을 표현하기 위해 보정된다.

d.date()

같은 날짜를 가진 date 객체를 반환한다.

d.replace([year [, month [, day [, hour [, minute [, second [, microsecond [,tzinfo]]]]]]])

하나 이상의 구성 요소를 지정된 값으로 바꾼 새로운 datetime 객체를 반환한다. 개별 값을 바꾸려면 키워드 인수를 사용하면 된다.

d.time()

같은 시간을 가진 time 객체를 반환한다. 반환되는 time 객체에는 시간대 정보가 설정되지 않는다.

d.timetz()

같은 시간과 시간대를 가진 time 객체를 반환한다.

d.utctimetuple()

UTC 시간으로 정규화된 날짜와 시간 정보를 담은 `time.struct_time` 객체를 반환 한다.

timedelta 객체

timedelta 객체는 날짜 또는 시간의 차이를 표현한다. timedelta 객체는 보통 – 연산을 사용하여 두 datetime 인스턴스의 차이를 계산한 결과로 생성된다. 다음 클래스를 사용하여 직접 생성할 수도 있다.

timedelta([days [, seconds [, microseconds [, milliseconds [, minutes [, hours [, weeks]]]]]])

날짜나 시간 차이를 나타내는 timedelta 객체를 생성한다. 여기서 주요한 매개변수들은 내부적으로 차이 값을 표현하기 위해 사용되는 days, seconds,

microseconds뿐이다. 다른 매개변수들을 제공하면 이들은 일, 초, 마이크로초로 변환된다. 반환되는 timedelta 객체의 days, seconds, microseconds 속성에 이 값들이 저장된다.

다음 클래스 속성들은 timedelta 인스턴스의 최대 범위와 해상력을 설명한다.

timedelta.min

표현할 수 있는 가장 음의 timedelta 객체(timedelta(-999999999))

timedelta.max

표현할 수 있는 가장 양의 timedelta 객체(timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)).

timedelta.resolution

동일하지 않은 timedelta 객체 사이의 나눌 수 있는 가장 작은 차이를 나타내는 timedelta 객체(timedelta(microseconds=1)).

날짜 관련 수학 연산

datetime 모듈의 중요한 특징으로 날짜와 관련된 수학 연산을 지원한다는 점이 있다. date와 datetime 객체 모두 다음 연산들을 지원한다.

연산

```
td = date1 - date2
date2 = date1 + td
date2 = date1 - td
date1 < date2
date1 <= date2
date1 == date2
date1 != date2
date1 > date2
date1 >= date2
```

설명

timedelta 객체를 반환한다.
timedelta 객체에 date를 더한다.
timedelta 객체에 date를 뺀다.
날짜 비교

날짜를 비교할 때 시간대 정보가 제공된 경우 주의를 기울여야 한다. 날짜가 tzinfo 정보를 가지고 있으면 tzinfo를 가진 날짜하고만 비교할 수 있다. 그렇지 않으면 TypeError가 발생한다. 시간대가 서로 다른 날짜들을 비교할 때는 비교 전에 이들을 먼저 UTC에 맞추어야 한다.

timedelta 객체는 다양한 수학 연산을 지원한다.

연산	설명
<code>td3 = td2 + td1</code>	두 차이(delta)를 더한다.
<code>td3 = td2 - td1</code>	두 차이를 뺀다.
<code>td2 = td1 * i</code>	정수를 곱한다.
<code>td2 = i * td2</code>	
<code>td2 = td1 // i</code>	정수 i로 바닥 나누기
<code>td2 = -td1</code>	단항 빼기, 더하기
<code>td2 = +td1</code>	
<code>abs(td)</code>	절대값
<code>td1 < td2</code>	비교
<code>td1 <= td2</code>	
<code>td1 == td2</code>	
<code>td1 != td2</code>	
<code>td1 > td2</code>	
<code>td1 >= td2</code>	

몇 가지 예를 보자.

```
>>> today = datetime.datetime.now()
>>> today.ctime()
'Thu Oct 20 11:10:10 2005'
>>> oneday = datetime.timedelta(days=1)
>>> tomorrow = today + oneday
>>> tomorrow.ctime()
'Fri Oct 21 11:10:10 2005'
>>>
```

date, datetime, time, timedelta는 모두 변경할 수 없다. 따라서 사전의 키로 사용할 수 있고 집합에 넣을 수 있으며 기타 다양한 연산에 사용할 수 있다.

tzinfo 객체

datetime 모듈에 있는 많은 메서드에서 시간대 정보를 나타내는 특별한 tzinfo 객체를 사용한다. tzinfo는 단순히 기본 클래스일 뿐이다. 개별 시간대는 tzinfo을 상속한 후 다음 메서드들을 구현함으로써 생성한다.

`tz.dst(dt)`

적용 가능한 경우 일광 절약 시간(DST: daylight savings time)을 위한 조정분을 나타내는 timedelta 객체를 반환한다. DST에 관한 아무런 정보도 얻을 수 없으면 None을 반환한다. 인수 dt는 datetime 객체이거나 None이다.

tz.fromutc(dt)

datetime 객체 dt를 UTC 시간에서 지역 시간대로 변환한 새로운 datetime 객체를 반환한다. 이 메서드는 datetime 객체의 astimezone() 메서드를 호출할 때 호출된다. tzinfo에 기본 구현이 미리 제공되므로 보통 이 메서드를 재정의할 필요가 없다.

tz.tzname(dt)

시간대 이름(예를 들어, “US/Central”)을 담은 문자열을 반환한다. 인수 dt는 datetime 객체 또는 None이다.

tz.utcoffset(dt)

지역 시간이 UTC에서 동쪽 방향으로 몇 분 차이가 나는지를 나타낸 오프셋을 담은 timedelta 객체를 반환한다. 이 오프셋은(적용 가능한 경우) 일광 절약 시간을 포함한 지역 시간을 구성하는 모든 요소를 고려하여 계산된 것이다. 인수 dt는 datetime 객체 또는 None이다.

다음 예는 시간대를 정의할 때 사용할 수 있는 기본 틀을 보여준다.

```
# 반드시 정의되어야 하는 변수들
# TZOFFSET - UTC에서 시로 나타낸 시간대의 오프셋. 예를 들어, US/CST는 -6 시이다.
# DSTNAME - DST가 효력이 있을 때 시간대 이름
# STDNAME - DST가 효력이 없을 때 시간대 이름

class SomeZone(datetime.tzinfo):
    def utcoffset(self,dt):
        return datetime.timedelta(hours=TZOFFSET) + self.dst(dt)
    def dst(self,dt):
        # is_dst()는 지역 시간대 규칙에 의해 DST가 적용되는지를
        # 반환하는 함수로서 반드시 구현해야 한다.
        if is_dst(dt):
            return datetime.timedelta(hours=1)
        else:
            return datetime.timedelta(0)
    def tzname(self,dt):
        if is_dst(dt):
            return DSTNAME
        else:
            return STDNAME
```

시간대를 정의하는 다른 예는 datetime 온라인 문서에서 찾을 수 있다.

날짜와 시간 파싱

날짜를 다룰 때 어떻게 다양한 시간과 날짜 문자열을 적절한 datetime 객체로 파싱 할 수 있느냐 하는 질문을 많이 한다. datetime 모듈에서 실제로 제공하는 파싱 함수는 datetime.strptime()밖에 없다. 이 함수를 사용할 때는 포맷 코드를 다양하게 조합하여 정확하게 날짜 포맷을 지정해야 한다(time.strptime() 참고). 예를 들어, 날짜 문자열 s=“Aug 23, 2008”을 파싱하려면 d = datetime.datetime.strptime(s, “%b %d, %Y”)처럼 해야 한다.

흔히 사용되는 날짜 포맷 몇 가지를 자동으로 이해하는 유연한 날짜 파싱 기능이 필요하다면 써드 파티 모듈을 사용해야 한다. datetime 모듈의 기능을 확장하는 다양한 유ти리티 모듈을 찾고 싶다면 파이썬 패키지 색인(Python Package Index) (<http://pypi.python.org>)에 가서 “datetime” 키워드로 검색해보기 바란다.

참고

[time\(498페이지\)](#)

errno

errno 모듈은 os와 socket 모듈에서 주로 사용되는 다양한 운영체제 시스템 호출로부터 반환되는 정수 에러 코드의 기호 이름을 정의한다. 보통 여기 나오는 코드들은 OSError 또는 IOError 예외의 errno 속성에 사용된다. os.strerror() 함수는 에러 코드를 문자열 에러 메시지로 변환하는 데 사용한다. 다음 사전을 정수 에러 코드를 기호 이름으로 변환하는 데 사용할 수도 있다.

errorcode

errno 정수를 기호 이름(‘EPERM’ 같은)으로 매핑하는 사전.

POSIX 에러 코드

다음 표는 일반적인 시스템 에러 코드의 POSIX 기호 이름을 보여준다. 여기 나온 에러 코드는 거의 모든 버전의 유닉스, 맥킨토시 OS X, 윈도에서 지원된다. 기타 유닉스 시스템에서는 여기 나오지 않은 덜 일반적인 에러 코드를 추가로 사용할 수도 있다. 그런 에러가 발생하면 errorcode 사전을 통해 프로그램에서 사용하기 적절한 기호 이름을 찾을 수 있다.

에러 코드	설명
E2BIG	인수 목록이 너무 깊음
EACCES	권한 없음
EADDRINUSE	주소가 이미 사용 중임
EADDRNOTAVAIL	요청한 주소에 할당 불가
EAFNOSUPPORT	프로토콜이 지원하지 않는 주소 체계
EAGAIN	다시 시도하라
EALREADY	연산이 이미 진행 중임
EBADF	잘못된 파일 번호
EBUSY	장치나 자원이 사용 중임
ECHILD	자식 프로세스가 없음
ECONNABORTED	소프트웨어가 연결을 중단시킴
ECONNREFUSED	연결이 거부됨
ECONNRESET	상대방이 연결을 초기화함
EDEADLK	자원 데드락 발생
EDEADLOCK	파일 잠금 데드락 에러
EDESTADDRREQ	목적지 주소가 필요함
EDOM	함수의 수학 인수가 정의역을 벗어남
EDQUOT	할당량을 초과함
EEXIST	파일이 이미 존재함
EFAULT	잘못된 주소
EFBIG^T	파일이 너무 큼
EHOSTDOWN	호스트가 다운됨
EHOSTUNREACH	호스트로 가는 경로 없음
EILSEQ	적절하지 않은 바이트 순서열
EINPROGRESS	연산이 현재 진행 중임
EINTR	인터럽트된 시스템 호출
EINVAL	유효하지 않은 인수
EIO	에러
EISCONN	전송 종점이 이미 연결된 상태임
EISDIR	디렉터리임
ELOOP	너무 많은 심벌릭 링크를 만남
EMFILE	너무 많은 파일을 염
EMLINK	너무 많은 링크
EMSGSIZE	메시지가 너무 깊음
ENETDOWN	네트워크가 다운됨
ENETRESET	재설정으로 네트워크 연결이 끊김
ENETUNREACH	네트워크에 연결할 수 없음
ENFILE	파일 표(file table) 오버플로우
ENOBUFS	사용할 수 있는 버퍼 공간 없음
ENODEV	장치가 없음
ENOENT	파일 또는 디렉터리가 없음

ENOEXEC	실행 파일 형식 에러
ENOLCK	레코드 락을 사용할 수 없음
ENOMEM	메모리 부족
ENOPROTOOPT	프로토콜을 사용할 수 없음
ENOSPC	장치에 남은 공간이 없음
ENOSYS	함수가 구현되지 않음
ENOTCONN	전송 종점이 연결되지 않음
ENOTDIR	디렉터리가 아님
ENOTEMPTY	디렉터리가 비어 있지 않음
ENOTSOCK	소켓 아닌 것에 소켓 연산
ENOTTY	터미널이 아님
ENXIO	장치 또는 주소가 없음
EOPNOTSUPP	전송 종점에서 지원하지 않는 연산
EPERM	허용되지 않는 연산
EPFNOSUPPORT	지원하지 않는 프로토콜 체계
EPIPE	깨진 파일
EPROTONOSUPPORT	지원하지 않는 프로토콜
EPROTOTYPE	소켓에 잘못된 타입의 프로토콜 사용
ERANGE	표현할 수 없는 계산 결과
EREMOTE	원격 객체임
EROFS	읽기 전용 파일 스트림
ESHUTDOWN	전송 종점이 닫힌 후에는 전송할 수 없음
ESOCKTNOSUPPORT	지원하지 않는 소켓 타입
ESPIPE	적절하지 않은 탐색
ESRCH	프로세스가 없음
ESTALE	오염된 NFS 파일 핸들
ETIMEDOUT	연결 타임아웃
ETOOMANYREFS	너무 많은 참조: 이을 수 없음
EUSERS	너무 많은 사용자
EWOULDBLOCK	연산이 블록됨
EXDEV	장치 사이 링크

윈도 에러 코드

다음 표에 나온 에러 코드는 윈도에서만 사용할 수 있다.

에러 코드	설명
WSAEACCES	권한이 없음
WSAEADDRINUSE	주소가 이미 사용 중임
WSAEADDRNOTAVAIL	요청한 주소를 할당할 수 없음
WSAEAFNOSUPPORT	프로토콜 체계가 주소 체계를 지원하지 않음
WSAEALREADY	연산이 이미 진행 중임
WSAEBADF	잘못된 파일 핸들

WSAECONNABORTED	연결 중단을 발생시킨 소프트웨어
WSAECONNREFUSED	연결 거부됨
WSAECONNRESET	상대방이 연결을 초기화함
WSAEDESTADDRREQ	목적지 주소가 필요함
WSAEDISCON	원격 종료
WSAEDQUOT	디스크 할당량 초과
WSAEFAULT	잘못된 주소
WSAEHOSTDOWN	호스트가 다운됨
WSAEHOSTUNREACH	호스트로 가는 경로 없음
WSAEINPROGRESS	현재 진행 중인 연산
WSAEINTR	인터럽트된 시스템 호출
WSAEINVAL	잘못된 인수
WSAEISCONN	소켓이 이미 연결됨
WSAELoop	이름을 해석할 수 없음
WSAEMFILE	너무 많은 열린 파일
WSAEMSGSIZE	메시지가 너무 깊
WSAENAMETOOLONG	이름이 너무 깊
WSAENETDOWN	네트워크가 다운됨
WSAENETRESET	초기화로 네트워크 연결이 끊김
WSAENETUNREACH	네트워크에 연결할 수 없음
WSAENOBUFS	사용 가능한 버퍼 공간이 없음
WSAENOPROTOOPT	잘못된 프로토콜 옵션
WSAENOTCONN	소켓이 연결되지 않음
WSAENOTEMPTY	비어 있지 않은 디렉터리를 삭제할 수 없음
WSAENOTSOCK	소켓 아닌 것에 소켓 연산
WSAEOPNOTSUPP	지원하지 않는 연산
WSAEPFNOSUPPORT	지원하지 않는 프로토콜 체계
WSAEPROCLIM	너무 많은 프로세스
WSAEPROTOSUPPORT	지원하지 않는 프로토콜
WSAEPROTOTYPE	소켓에 잘못된 타입의 프로토콜 사용
WSAEREMOTE	항목이 지역 머신에 없음
WSAESHUTDOWN	소켓이 닫힌 후에는 전송할 수 없음
WSAESOCKTNOSUPPORT	지원하지 않는 소켓 타입
WSAESTALE	더 이상 사용할 수 없는 파일 헤더
WSAETIMEDOUT	연결 타임아웃
WSAETOOMANYREFS	커널 객체에 대한 너무 많은 참조
WSAEUSERS	할당량 초과
WSAEWOULDBLOCK	자원을 일시적으로 사용할 수 없음
WSANOTINITIALISED	WSA 구동이 성공적으로 수행되지 않음
WSASYSNOTREADY	네트워크 하위 시스템을 사용할 수 없음
WSAVERNOTSUPPORTED	범위를 벗어난 Winsock.dll 버전

fcntl

fcntl은 유닉스 파일 기술자에 대해 파일 및 I/O 방식을 제어할 수 있게 하는 모듈이다. 파일 기술자는 파일이나 소켓 객체의 fileno() 메서드로 얻을 수 있다.

fcntl(fd, cmd [, arg])

열린 파일 기술자 fd에 명령 cmd를 수행한다. cmd는 정수 명령 코드이다. arg는 옵션인 인수로 정수 또는 문자열일 수 있다. arg가 정수이면 이 함수의 반환 값은 정수이다. arg가 문자열이면 이 문자열을 이진 데이터 구조로 해석하고 반환 값은 문자열 객체로 변환한 버퍼 내용이다. 이 경우 데이터 손상을 방지하기 위해, 제공된 인수와 반환 값은 1,024 바이트를 넘지 않아야 한다. 다음 명령을 사용할 수 있다.

명령어	설명
F_DUPFD	파일 기술자를 복제한다. arg는 새로운 파일 기술자가 가질 수 있는 최소 숫자를 나타낸다. 시스템 호출인 os.dup()와 비슷하다.
F_SETFD	close-on-exec 플래그를 arg(0 또는 1)로 설정한다. 1로 설정하면 exec() 시스템 호출 때 파일이 닫힌다.
F_GETFD	close-on-exec 플래그를 반환한다.
F_SETFL	상태 플래그를 다음을 비트 OR한 arg로 설정한다.
O_NDELAY-년블로킹 I/O(System V)	
O_APPEND-추가 append 모드(System V)	
O_SYNC-동기적 쓰기(System V)	
FNDELAY-년블로킹 I/O (BSD)	
FAPPEND-추가 모드(BSD)	
FASYNC-I/O가 가능할 때 프로세스 그룹에 SIGIO 신호를 보낸다(BSD).	
F_GETFL	F_SETFL로 설정한 상태 플래그를 얻는다.
F_GETOWN	SIGIO와 SIGURG 신호를 수신하는 프로세스 ID나 프로세스 그룹 ID를 얻는다(BSD).
F_SETOWN	SIGIO와 SIGURG 신호를 수신할 프로세스 ID나 프로세스 그룹 ID를 설정한다(BSD).
F_GETLK	파일 잠금(file-locking) 연산에 사용되는 플록(flock) 구조를 반환한다.
F_SETLK	파일을 잠근다. 파일이 이미 잠겨 있으면 -1을 반환한다.
F_SETLKW	파일을 잠그지만 락을 획득할 수 없으면 기다린다.

fcntl() 함수 실행이 실패하면 IOError 예외가 발생한다. F_GETLK와 F_SETLK 명령을 구현하는 데 flockf() 함수가 사용된다.

ioctl(fd, op [, mutate_flag])

op로 지정한 연산이 라이브러리 모듈 termios에 정의되어 있다는 점을 제외하고는 fcntl() 함수처럼 작동한다. 옵션인 mutate_flag는 인수로 변경 가능한 버퍼 객체를 지정했을 때 이 함수가 작동하는 방식을 제어한다. 자세한 정보는 온라인 문서를 참고하기 바란다. ioctl()은 장치 드라이버나 운영체제의 저수준 구성 요소와 상호 작용하려는 목적으로 주로 사용하기 때문에 기반 플랫폼에 매우 의존적이다. 이 식성을 염두에 둔 코드에서는 사용하지 않도록 한다.

flock(fd, op)

파일 기술자 fd에 접근 연산 op를 수행한다. op는 fcntl 모듈에 있는 다음 제약 조건들을 비트 OR 한 것이다.

항목	설명
LOCK_EX	상호 배타적 락. 락을 획득하려는 모든 시도는 락이 해제될 때까지 막힌다.
LOCK_NB	넌블로킹 모드. 락이 이미 사용 중이면 IOError와 함께 즉시 반환한다.
LOCK_SH	공유 락. 배타적 락(LOCK_EX)을 획득하려는 시도는 막히지만 공유 락은 획득할 수 있다.
LOCK_UN	락 해제. 기존에 소유하던 락을 해제한다.

넌블로킹 모드에서는 락을 획득할 수 없을 경우 IOError 예외가 발생한다. 어떤 시스템에서는 os.open()에 특수 플래그를 지정하여 파일을 여는 작업과 잠그는 작업을 단일 연산으로 수행할 수 있다. 자세한 내용은 os 모듈을 참고하기 바란다.

lockf(fd, op [, len [, start [, whence]]])

파일 일부분에 레코드 잡금 또는 범위 잡금을 수행한다. op는 flock() 함수에서 의미와 같다. len은 잡글 바이트 수이다. start는 whence 값에 상대적인 락 시작 위치를 나타낸다. whence는 파일 시작은 0, 현재 위치는 1, 파일 끝은 2 값을 갖는다.

예

```
import fcntl

# 파일을 연다.
f = open("foo", "w")

# 파일 객체 f에 close-on-exec 비트를 설정한다.
fcntl.fcntl(f.fileno(), fcntl.F_SETFD, 1)
```

```
# 파일을 잠근다(블로킹)
fcntl.flock(f.fileno(), fcntl.LOCK_EX)

# 파일의 첫 8192 바이트를 잠근다(넌블로킹)
try:
    fcntl.lockf(f.fileno(), fcntl.LOCK_EX | fcntl.LOCK_NB, 8192, 0, 0)
except IOError as e:
    print("Unabled to acquire lock %s" % e)
```

Note

- 사용할 수 있는 fcntl() 명령과 옵션은 시스템에 따라 다르다. 어떤 플랫폼에서는 fcntl 모듈이 100 개 이상의 상수를 포함하기도 한다.
- 보통 다른 모듈에서 정의된 잠금 연산은 컨텍스트 관리 프로토콜을 지원하지만 파일 잠금은 그렇지 않다. 파일 락을 획득했다면 잊지 말고 락을 적절히 해제하는 코드를 작성해야 한다.
- 이 모듈에 있는 많은 함수들은 소켓 파일 기술자에 적용할 수 있다.

io

io 모듈은 파이썬 3에서 사용되는 다양한 형태의 I/O와 내장 open() 함수를 구현하는 클래스를 제공한다. 이 모듈은 파이썬 2.6에서도 사용할 수 있다.

io 모듈은 기본 I/O의 여러 형태를 차이 없이 처리하는 것을 핵심 과제로 삼고 있다. 예를 들어, 텍스트로 작업할 때는 줄바꿈이나 문자 인코딩과 관련된 문제 때문에 이진 데이터로 작업할 때와 약간 다르다. 이러한 차이점에서 발생하는 문제를 해결하기 위해 이 모듈은 일련의 계층을 구축하여 각 계층에서 바로 전 계층에 기능을 더 추가하는 방식을 취하고 있다.

기반 I/O 인터페이스

io 모듈에서는 모든 파일과 비슷한 객체들이 구현해야 하는 기본 I/O 프로그래밍 인터페이스를 정의한다. 이 인터페이스는 기반 클래스 IOBase에 정의되어 있다. IOBase 인스턴스 f는 다음 기본 연산들을 지원한다.

속성	설명
f.closed	파일이 닫혔는지를 나타내는 플래그
f.close()	파일을 닫는다.
f.fileno()	정수 파일 기술자를 반환한다.

<code>f.flush()</code>	I/O 버퍼를 비운다(가능하면).
<code>f.isatty()</code>	f가 터미널이면 True를 반환한다.
<code>f.readable()</code>	f가 읽기용으로 열렸으면 True를 반환한다.
<code>f.readline([limit])</code>	스트림에서 한 줄을 읽는다. limit는 읽을 최대 바이트 수이다.
<code>f.readlines([limit])</code>	f에서 모든 줄을 읽어 리스트로 반환한다. limit 값이 주어지면 이는 중단하기 전까지 읽을 최대 바이트 수를 나타낸다. 실제 읽은 바이트 수는 마지막 줄을 손상시키지 않고 읽기 위해 약간 더 커질 수도 있다.
<code>f.seek(offset, [whence])</code>	파일 포인터를 whence로 지정한 위치에 상대적인 새로운 위치로 옮긴다. offset은 바이트 수이다. whence 값 0은 파일 시작을, 1은 현재 위치를, 2는 파일의 끝을 나타낸다.
<code>f.seekable()</code>	f가 탐색할 수 있는 경우 True를 반환한다.
<code>f.tell()</code>	파일 포인터의 현재 값을 반환한다.
<code>f.truncate([size])</code>	최대 size 바이트가 되도록 파일 크기를 줄인다. size를 생략하면 파일을 0 바이트로 줄인다.
<code>f.writable()</code>	f가 쓰기용으로 열렸으면 True를 반환한다.
<code>f.writelines(lines)</code>	줄들을 담은 순서열을 f에 쓴다. 줄바꿈 문자를 알아서 추가하지 않기 때문에 각 줄 끝에 반드시 줄바꿈 문자를 포함시켜야 한다.

무가공 I/O

I/O 시스템의 가장 저수준에서는 무가공 바이트에 대해 직접 I/O를 수행하게 된다. 이를 위한 핵심 객체는 `FileIO`이다. 이 객체는 `read()`나 `write()` 같은 저수준 시스템 호출에 대한 꽈 직접적인 인터페이스를 제공한다.

`FileIO(name [, mode [, closefd]])`

파일 또는 시스템 파일 기술자에 무가공 저수준 I/O를 수행하기 위한 클래스. `name`은 파일 이름이 될 수도 있고 `os.open()` 함수나 기타 파일 객체의 `fileno()` 메서드가 반환하는 정수 파일 기술자일 수도 있다. `mode`는 ‘r’(기본 값), ‘w’, ‘a’ 중 하나가 될 수 있고 각각 읽기, 쓰기, 추가를 나타낸다. ‘+’를 추가하면 읽기 쓰기를 모두 지원하는 개신 모드가 된다. `closefd`는 `close()` 메서드에서 실제로 내부 파일을 닫을지를 결정하는 플래그이다. 기본 값은 `True`이지만 이미 어디선가 열린 파일에 대한 래퍼를 생성하느라 `FileIO`를 사용하고 있는 경우에는 `False`로 설정할 수도 있다. 파일 이름을 준 경우에는 결과로 생성되는 파일 객체에 대해 운영체제의 `open()`을 호출하여 연다. 내부 버퍼링은 수행되지 않고 모든 데이터는 무가공 바이트 문

자열로 처리된다. FileIO 인스턴스 f는 앞에서 언급한 모든 기본 I/O 연산을 지원하며 추가로 다음 속성과 매서드들을 가진다.

속성	설명
f.closefd	f.close()를 호출할 때 내부 파일 기술자를 닫을지를 결정하는 플래그(읽기 전용)
f.mode	파일을 열 때 사용한 파일 모드(읽기 전용)
f.name	파일 이름(읽기 전용)
f.read([size])	시스템 호출 한 번으로 최대 size 바이트만큼 읽는다. size를 생략하면 f.readall()로 읽을 수 있는 만큼 읽는다. 이 연산은 요청한 것보다 적은 바이트를 반환할 수 있으므로 len()으로 크기를 확인해야 한다. 널블로킹 모드에서는 읽을 수 있는 데이터가 없을 경우 None을 반환한다.
f.readall()	읽을 수 있는 만큼 데이터를 읽고 결과를 바이트 문자열로 반환한다. EOF일 경우 빈 문자열을 반환한다. 널블로킹 모드에서는 즉시 읽을 수 있는 만큼의 데이터만 반환한다.
f.write(bytes)	시스템 호출 한 번으로 바이트 문자열 또는 바이트 배열을 f에 쓴다. 실제 쓴 바이트의 수를 반환한다. 이 값은 bytes보다 작을 수 있다.

FileIO 객체는 아주 저수준에서 작동하기 때문에 read()나 write() 같은 운영체제 시스템 호출 위에 다소 얇은 계층을 제공하는 데 그친다. 특히, 이 객체를 사용할 때는 f.read() 또는 f.write() 연산이 요청한 데이터를 모두 읽거나 쓴다는 보장이 없기 때문에 반환 코드를 항상 확인할 필요가 있다. fcntl 모듈로 파일 락, 잠금 방식 등 파일의 저수준 특성을 변경할 수 있다.

FileIO 객체는 텍스트 데이터 같은 줄 기반 데이터에는 사용하지 않아야 한다. f.readline()과 f.readlines() 같은 메서드가 있지만 이들은 IOBase 기반 클래스에 있는 것들로 f.read() 연산을 사용해 한 번에 한 바이트씩 읽는 형태로 파이썬에서 직접 구현한 것들이다. 말할 것도 없이 성능이 아주 안 좋다. 예를 들어, FileIO 객체에 f.readline()을 사용하는 것은 파이썬 2.6에서 open() 함수로 생성한 표준 파일 객체에 f.readline()을 사용하는 것보다 750배 느린다.

버퍼 이진 I/O

버퍼 I/O 계층에는 무가공 이진 데이터를 메모리 버퍼를 사용해 읽고 쓰는 많은 수의 파일 객체들이 있다. 이들은 입력으로 앞 절에서 살펴본 FileIO 객체 같은 무가공 I/O를 구현한 파일 객체를 받는다. 이 절에 나오는 모든 클래스는 BufferdIOBase를 상속한다.

BufferedReader(raw [, buffer_size])

raw로 지정한 무가공 파일에 버퍼 이진 읽기를 수행하는 클래스. buffer_size는 사용할 버퍼 크기를 바이트로 지정한다. 이를 생략할 경우 DEFAULT_BUFFER_SIZE가 사용된다(이 책을 쓰는 지금은 크기가 8,192 바이트). BufferedReader 인스턴스 f는 IOBase가 지원하는 모든 연산과 더불어 다음 연산들을 제공한다.

메서드	설명
f.peek([n])	파일 포인터 움직임 없이 I/O 버퍼에서 최대 n 바이트의 데이터를 반환한다. n을 생략한 경우 한 바이트만 반환한다. 필요하다면 버퍼가 현재 비어 있는 경우 버퍼를 채우기 위해 읽기 연산을 수행한다. 이 연산은 현재 버퍼 크기보다 더 많은 바이트를 반환하지 않기 때문에 결과가 요청한 바이트 n보다 작을 수 있다.
f.read([n])	n 바이트를 읽고 바이트 문자열로 반환한다. n을 생략한 경우 읽을 수 있는 모든 데이터를(최대 EOF까지) 읽어 반환한다. 내부 파일이 넌블로킹이면 읽을 수 있는 데이터까지 읽고 반환한다. 넌블로킹 파일을 읽는 데 사용 가능한 데이터가 없으면 BlockingIOError 예외가 발생한다.
f.read1([n])	한 번의 시스템 호출로 n 바이트까지 읽고 바이트 문자열로 반환한다. 버퍼에 이미 데이터가 있으면 간단히 이것을 반환한다. 그렇지 않을 경우 데이터를 반환하기 위해 무가공 파일에 read()를 한 번 수행한다. f.read()와 달리 이 연산은 내부 파일이 EOF에 도달하지 않았더라도 요청한 데이터보다 적은 데이터를 반환할 수 있다.
f.readinto(b)	파일에서 len(b) 바이트 데이터를 bytearray 객체 b로 읽는다. 실제 읽은 바이트 수를 반환한다. 내부 파일이 넌블로킹 모드에 있고 사용 가능한 데이터가 없으면 BlockingIOError 예외가 발생한다.

BufferedWriter(raw [, buffer_size [, max_buffer_size]])

raw로 지정한 무가공 파일에 버퍼 이진 쓰기를 수행하는 클래스. buffer_size는 데이터를 내부 I/O 스트림으로 보내기 전까지 버퍼에 저장할 바이트 수를 나타낸다. 기본 값은 DEFAULT_BUFFER_SIZE이다. max_buffer_size는 넌블로킹 스트림에 쓸 출력 데이터를 저장하는 데 사용할 버퍼의 최대 크기를 나타내며 기본 값은 buffer_size의 두 배이다. 이렇게 더 큰 값을 사용하는 이유는 운영체제가 버퍼의 이전 내용을 I/O 스트림에 쓰는 동안 계속 쓰기가 가능하도록 하기 위해서다. BufferedWriter 인스턴스 f는 다음 연산들을 지원한다.

메서드	설명
f.flush()	버퍼에 저장된 모든 바이트를 내부 I/O 스트림에 쓴다. 파일이 넌블로킹 모드이고 연산이 막히면(예를 들어, 스트림이 현재 새로운 테

이터를 받을 수 없을 때) BlockingIOError 예외가 발생한다.

f.write(bytes)

bytes로 지정한 바이트들을 I/O 스트림에 쓰고 실제 쓴 바이트 수를 반환한다. 내부 스트림이 널블로킹이면 쓰기 연산이 막힌 경우 BlockingIOError 예외가 발생한다.

BufferedRWPair(reader, writer [, buffer_size [, max_buffer_size]])

무가공 I/O 스트림 한 쪽에 버퍼 이진 읽기와 쓰기를 수행하는 클래스. reader는 읽기를 지원하는 무가공 파일이고 writing은 쓰기를 지원하는 무가공 파일이다. 이들은 서로 다를 수 있는데 이 점은 파이프나 소켓을 사용하는 특정한 종류의 통신을 할 때 유용하다. 버퍼 크기 관련 매개변수는 BufferedWriter와 같은 의미를 가진다. BufferedRWPair 인스턴스 f는 BufferedReader와 BufferedWriter의 모든 연산을 지원한다.

BufferedRandom(raw [, buffer_size [, max_buffer_size]])

임의 접근(예를 들어, 탐색seek)을 지원하는 무가공 I/O 스트림에 버퍼 이진 읽기와 쓰기를 수행하는 클래스. raw는 읽기, 쓰기, 탐색 연산을 모두 지원하는 무가공 파일이어야 한다. 버퍼 크기 관련 매개변수는 BufferedWriter와 같은 의미를 가진다. BufferedRandom 인스턴스 f는 BufferedReader와 BufferedWriter의 모든 연산을 지원한다.

BytesIO([bytes])

버퍼 I/O 스트림의 기능을 구현하는 메모리 파일(in-memory file). bytes는 파일의 초기 내용을 담은 바이트 문자열이다. BytesIO 인스턴스 b는 BufferedReader와 BufferedWriter 객체의 모든 연산을 지원한다. 추가로 파일의 현재 내용을 바이트 문자열로 반환하는 데 b.getvalue() 메서드를 사용할 수 있다.

FileIO 객체와 마찬가지로 이 절에서 다룬 모든 파일 객체는 텍스트 데이터 같은 줄 기반 데이터에 사용하면 안 된다. 버퍼링 때문에 그나마 낫지만 그래도 성능은 여전히 좋지 않다(파이썬 2.6 내장 open() 함수를 사용하여 파일에서 줄을 읽는 것 보다 50배 정도 느리다). 또한, 내부 버퍼링 때문에 쓰기를 할 때 flush() 연산에 주의를 기울여야 한다. 예를 들어, 파일 포인터를 새로운 위치로 옮기려고 f.seek()를 사용할 때는 먼저 이전에 쓴 데이터를 비우기 위해(존재하면) f.flush()를 호출해야 한다.

버퍼 크기 관련 매개변수는 쓰기가 일어나는 순간에 대한 제약을 가할 뿐 내부

자원 사용에 제약을 가하는 것은 아니다. 예를 들어, 버퍼 파일 f에 f.write(data)를 호출하면 data에 있는 모든 바이트를 내부 버퍼로 복사한다. data가 아주 큰 바이트 배열일 경우 프로그램의 메모리 사용량이 상당히 증가한다. 따라서, 큰 데이터는 write() 연산을 한 번으로 쓰는 것보다 적절한 데어리로 나누어 쓰는 것이 낫다. io 모듈은 상대적으로 최근에 나온 것이기 때문에 이러한 작동 방식은 나중 버전에서 달라질 수 있다는 점을 유념하기 바란다.

텍스트 I/O

텍스트 I/O 계층은 줄 기반 문자 데이터를 처리하는 데 사용한다. 이 장에 나오는 클래스들은 버퍼 I/O 스트림을 기반으로 하며 줄 기반 처리와 유니코드 인코딩 및 디코딩을 지원한다. 모든 클래스는 TextIOBase를 상속한다.

```
TextIOWrapper(buffered [, encoding [, errors [, newline [, line_buffering]]]])
```

버퍼 텍스트 스트림용 클래스. buffered는 이전 절에 나온 버퍼 I/O 객체이다. encoding은 ‘ascii’ 또는 ‘utf-8’ 같은 텍스트 인코딩을 지정하는 문자열이다. errors는 유니코드 에러 처리 정책을 나타내고 기본 값은 ‘strict’이다(9장 참고). newline은 줄바꿈을 나타내는 문자 순서열이며 None, ‘\n’, ‘\r’, ‘\r\n’ 등이 될 수 있다. newline을 None으로 지정하면 읽을 때 줄 끝 문자가 ‘\n’으로 변환되고 os.linesep를 출력 때 줄바꿈 문자로 사용하는 보편 줄바꿈 모드를 사용한다. newline이 다른 값이면 출력 때 모든 ‘\n’ 문자를 지정한 줄바꿈 문자로 변환한다. line_buffering은 쓰기 연산에 줄바꿈 문자가 포함될 때 flush()를 수행할지를 제어하는 플래그이다. 기본 값은 False이다. TextIOWrapper 인스턴스 f는 IOBase에 정의된 모든 연산뿐만 아니라 다음 연산들도 지원한다.

메서드 설명

f.encoding	사용되는 텍스트 인코딩의 이름
f.errors	인코딩과 디코딩 에러 처리 정책
f.line_buffering	줄 버퍼링 방식을 결정하는 플래그
f.newlines	None, 문자열 또는 변환할 줄바꿈 문자의 여러 형태를 담은 튜플
f.read([n])	내부 스트림에서 최대 n개 문자를 읽고 문자열로 반환한다. n을 생략한 경우 파일 끝에 도달할 때까지 읽을 수 있는 모든 데이터를 읽는다. EOF를 만나면 빈 문자열을 반환한다. 반환된 문자열은 f.encoding으로 설정한 인코딩에 따라 디코딩된다.
f.readline([limit])	텍스트 한 줄을 읽고 문자열로 반환한다. EOF를 만나면 빈 문자

<code>f.write(s)</code>	열을 반환한다. limit는 읽을 최대 바이트 수를 나타낸다. 텍스트 인코딩 f.encoding을 사용하여 문자열 s를 내부 스트림에 쓴다.
-------------------------	---

`StringIO([initial [, encoding [, errors [, newline]]]])`

TextIOWrapper와 같은 방식으로 작동하는 메모리 파일 객체. initial은 파일의 초기 내용을 지정하는 문자열이다. 다른 매개변수는 TextIOWrapper와 같은 의미를 가진다. StringIO 인스턴스 s는 모든 일반 파일 연산을 지원하며 추가로 메모리 버퍼의 현재 내용을 반환하는 s.getvalue() 메서드를 가진다.

open() 함수

io 모듈에는 파일 3의 내장 open 함수와 동일한 다음 open 함수가 있다.

`open(file [, mode [, buffering [, encoding [, errors [, newline [, closefd]]]]])`

file을 열고 적절한 I/O 객체를 반환한다. file은 파일 이름을 담은 문자열이거나 이미 열린 I/O 스트림에 대한 정수 파일 기술자이다. 이 함수는 mode와 buffering 설정에 따라 io 모듈에 정의된 I/O 클래스 중 하나를 반환한다. mode가 ‘r’, ‘w’, ‘a’, ‘U’ 같은 텍스트 모드이면 TextIOWrapper 인스턴스를 반환한다. mode가 ‘rb’ 또는 ‘wb’이면 결과는 buffering 설정에 따라 다르다. buffering이 0이면 버퍼링 없는 무가공 I/O 수행을 위해 FileIO 인스턴스를 반환한다. buffering이 다른 값일 경우 파일 모드에 따라 BufferedReader, BufferedWriter, BufferedRandom 인스턴스를 반환한다. encoding, errors, newline 매개변수는 텍스트 모드로 열린 파일에만 적용되며 TextIOWrapper의 생성자에 전달된다. closefd는 file이 정수 파일 기술자일 경우 적용되며 FileIO의 생성자에 전달된다.

추상 기반 클래스

io 모듈은 타입 검사나 새로운 I/O 클래스 정의에 사용할 수 있는 다음 추상 기반 클래스들을 정의한다.

추상 클래스	설명
IOBase	모든 I/O 클래스를 위한 기반 클래스
RawIOBase	무가공 이진 I/O를 지원하는 객체를 위한 기반 클래스. IOBase를 상속함.
BufferedIOBase	버퍼 이진 I/O를 지원하는 객체를 위한 기반 클래스. IOBase를 상속

함.

TextIOWrapper	텍스트 스트림을 지원하는 객체를 위한 기반 클래스. IOWrapper를 상속함.
---------------	--

이 클래스들을 직접 사용하는 경우는 드물다. 이들의 사용법과 정의에 관련한 자세한 정보는 온라인 문서를 참고하도록 하라.

Note

io 모듈은 파이썬에 새로 추가된 것으로 파이썬 3에서 처음 도입되었고 파이썬 2.6에 역이식된 것이다. 이 책을 쓰는 지금 이 모듈은 아직 덜 다듬어져 있으며 극도로 낮은 런타임 성능을 보인다. 특히 아주 많은 양의 텍스트 I/O를 수행하는 응용 프로그램에서 더욱 그렇다. 파이썬 2를 사용하는 중이라면 io 모듈에 정의된 I/O 클래스를 사용하는 것보다 내장 open() 함수를 사용하는 것이 낫다. 파이썬 3를 사용한다면 다른 마땅한 대안이 없다. 나중 릴리스에서 성능이 향상될 가능성이 높지만 유니코드 디코딩과 결합된 이 I/O에 대한 계층 기반 접근법은 파이썬 2 I/O의 기반인 C 표준 라이브러리에 있는 무기공 I/O 성능에는 미치지 못한다.

logging

logging 모듈은 응용 프로그램에서 이벤트, 에러, 경고 및 디버깅 정보를 로깅할 수 있게 하는 유연한 기능을 제공한다. 로깅 정보를 수집하고 거르고 파일에 쓰거나 시스템 로그로 보낼 수 있으며 심지어 네트워크를 통해 원격 머신에 보낼 수도 있다. 이 절에서는 이 모듈에서 주로 사용되는 필수적인 기능을 다룬다.

로깅 수준

logging 모듈은 주로 로그 메시지 생성 및 처리에 초점이 맞추어져 있다. 각 메시지는 텍스트와 심각성을 나타내는 수준 값으로 구성된다. 수준 값은 다음과 같이 기호 이름과 숫자 값을 가진다.

수준	값	설명
CRITICAL	50	심각한 에러/메시지
ERROR	40	에러
WARNING	30	경고 메시지
INFO	20	알림 메시지
DEBUG	10	디버깅
NOTSET	0	수준 설정 안됨

이 수준 값들은 logging 모듈 전반에 걸쳐서 다양한 함수와 메서드에서 사용된다. 예를 들어, 각 수준별 로그 메시지 생성을 위한 메서드가 있으며 특정 수준을 넘지 않는 메시지를 막는 필터도 존재한다.

기본 설정

logging 모듈의 함수를 사용하기 전에 먼저 루트 로거(root logger)라고 부르는 특수한 객체에 기본 설정을 해주어야 한다. 루트 로거는 로깅 수준, 출력 목적지, 메시지 형식, 기타 기본적인 세부 사항 등 로그 메시지의 기본 작동 방식을 관리한다. 관련 설정은 다음 함수로 수행한다.

basicConfig(kwargs)**

루트 로거에 기본 설정을 수행한다. 이 함수는 다른 로깅 관련 호출을 수행하기 전에 호출해야 한다. 이 함수는 몇 가지 키워드 인수를 입력으로 받는다.

키워드 인수	설명
<code>filename</code>	주어진 파일 이름을 가지는 파일에 로그 메시지를 추가한다.
<code>filemode</code>	파일을 열 때 사용할 모드를 지정한다. 기본으로 모드 ‘a’(추가)가 사용된다.
<code>format</code>	로그 메시지를 생성하는 데 사용할 포맷 문자열
<code>datefmt</code>	날짜와 시간을 출력하는 데 사용할 포맷 문자열
<code>level</code>	루트 로거의 수준을 설정한다. 이 수준과 같거나 더 높은 수준에 있는 로그 메시지를 처리한다. 더 낮은 수준에 있는 메시지는 조용히 무시한다.
<code>stream</code>	로그 메시지를 쓸 열린 파일을 지정한다. 기본 값은 <code>sys.stderr</code> 이다. 이 매개변수는 <code>filename</code> 매개변수와 동시에 사용할 수 없다.

대부분의 매개변수는 설명 없이 이해할 수 있을 것이다. `format` 인수는 파일 이름, 수준, 줄 번호 같은 추가적인 문맥 정보를 담도록 로그 메시지의 포맷을 지정하는 데 사용한다. `datefmt`은 `time.strftime()` 함수와 호환되는 날짜 포맷 문자열이어야 한다. 생략할 경우 날짜 형식은 ISO8601 형식으로 설정된다.

`format` 인수에서는 다음 확장 연산을 인식한다.

포맷	설명
<code>%(name)s</code>	로거의 이름
<code>%(levelname)s</code>	숫자 로깅 수준
<code>%(levelname)s</code>	로깅 수준의 텍스트 이름
<code>%(pathname)s</code>	로깅을 수행한 소스 파일의 경로
<code>%(filename)s</code>	로깅을 수행한 소스 파일의 파일 이름

<code>%(funcName)s</code>	로깅을 수행한 함수 이름
<code>%(module)s</code>	로깅을 수행한 모듈 이름
<code>%(lineno)d</code>	로깅을 수행한 줄 번호
<code>%(created)f</code>	로깅을 수행한 시간. <code>time.time()</code> 이 반환하는 것과 같은 숫자
<code>%(asctime)s</code>	로깅을 수행한 날짜와 시간을 ASCII 형식으로
<code>%(msecs)s</code>	로깅을 수행한 시간의 밀리초 부분
<code>%(thread)d</code>	스레드 ID
<code>%(threadName)s</code>	스레드 이름
<code>%(process)d</code>	프로세스 ID
<code>%(message)s</code>	로깅 메시지(사용자가 제공)

다음은 INFO 또는 그 이상의 수준을 가진 로그 메시지를 파일에 쓰도록 설정하는 예를 보여준다.

```
import logging
logging.basicConfig(
    filename = "app.log",
    format = "%(levelname)-10s %(asctime)s %(message)s",
    level = logging.INFO
)
```

이렇게 설정하면 CRITICAL 로그 메시지 “Hello World”는 로그 파일 ‘app.log’에 다음과 같이 나타난다.

```
CRITICAL 2005-10-25 20:46:57,126 Hello World
```

Logger 객체

로그 메시지를 생성하려면 Logger 객체를 얻어야 한다. 여기에서는 이 객체의 생성, 설정 및 사용 과정을 설명한다.

로거 생성

새로운 Logger 객체를 생성하는 데 다음 함수를 사용한다.

```
getLogger([logname])
```

`logname` 이름에 연결된 Logger 인스턴스를 반환한다. 객체가 존재하지 않으면 새로운 Logger 인스턴스를 생성하여 반환한다. `logname`은 이름 또는 점으로 구분된 이름들(예를 들어 ‘app’ 또는 ‘app.net’)을 담은 문자열이다. `logname`을 생략할 경우 루트 로거와 연결된 Logger 객체를 얻게 된다.

Logger 인스턴스를 생성하는 과정은 라이브러리 모듈에서 하는 방식과 다르다.

Logger를 생성할 때는 항상 getLogger()에 logname 매개변수로 이름을 주어야 한다. 내부적으로 getLogger()는 Logger 인스턴스들과 연결된 이름들의 캐시를 유지한다. 프로그램의 다른 부분에서 같은 이름을 가진 로거를 요청하는 경우 이전에 생성된 인스턴스를 반환한다. 이렇게 하면 프로그램 모듈 사이에 Logger 인스턴스를 전달하는 방법에 관해 고민할 필요가 없기 때문에 큰 규모의 응용 프로그램을 작성할 때 로그 메시지 처리 과정이 아주 간단해진다. 로깅을 원하는 각 모듈에서 간단히 getLogger()를 사용하여 적절한 Logger 객체를 얻기만 하면 된다.

이름 고르기

뒤에서 더 확실하게 그 이유를 설명할 테지만, getLogger()를 사용할 때는 항상 의미 있는 이름을 골라야 한다. 예를 들어, 응용 프로그램을 ‘app’라고 부르는 경우 응용 프로그램을 구성하는 각 모듈의 제일 앞 부분에서 getLogger(‘app’)를 호출해야 한다. 예를 들면 다음과 같다.

```
import logging
log = logging.getLogger('app')
```

getLogger(‘app.net’) 또는 getLogger(‘app.user’) 같이 로그 메시지의 소스를 확실히 나타내기 위해 모듈 이름을 덧붙일 수도 있다. 다음과 같이 하는 것이다.

```
import logging
log = logging.getLogger('app.'+__name__)
```

나중에 설명하겠지만, 모듈 이름을 추가하면 선택적으로 로그 메시지를 끄거나 특정 모듈의 로깅 설정만 다르게 하는 등의 작업이 수월해진다.

로그 메시지 생성

Logger 객체 인스턴스가 log일 때(이전 절에 나온 getLogger()를 사용하여 생성) 다음 메서드들을 사용하여 다양한 로깅 수준에서 로그 메시지를 생성할 수 있다.

로깅 수준	메서드
CRITICAL	log.critical(fmt [, *args [, exc_info [, extra]]])
ERROR	log.error(fmt [, *args [, exc_info [, extra]]])
WARNING	log.warning(fmt [, *args [, exc_info [, extra]]])
INFO	log.info(fmt [, *args [, exc_info [, extra]]])
DEBUG	log.debug(fmt [, *args [, exc_info [, extra]]])

fmt 인수는 로그 메시지의 포맷을 지정하는 포맷 문자열이다. args로 지정한 인수들은 포맷 문자열에서 포맷 지정자들을 대체하는 데 사용된다. 이러한 인수들로 결과 메시지를 생성하는 데 문자열 포맷 연산자 %가 사용된다. 인수를 여러 개 지정할 경우 이들을 튜플로 묶어 전달한다. 인수가 하나일 때는 % 바로 다음에 나온다. 인수로 사전을 지정하면 포맷 문자열에 키 이름을 포함시킬 수 있다. 다음은 이 과정이 어떻게 이루어지는지 보여준다.

```
log = logging.getLogger("app")
# 위치 인수로 포맷을 지정한 로그 메시지
log.critical("Can't connect to %s at port %d", host, port)

# 사전으로 포맷을 지정한 로그 메시지
parms = {
    'host' : 'www.python.org',
    'port' : 80
}
log.critical("Can't connect to %(host)s at port %(port)d", parms)
```

키워드 인수 exc_info를 True로 설정하면 sys.exc_info()에서 예외 정보를 얻어 로그 메시지에 추가한다. exc_info를 sys.exc_info()가 반환하는 것 같은 예외 튜플로 설정하면 지정한 예외 정보를 사용한다. extra 키워드 인수는 로그 메시지 포맷 문자열에 사용할 추가 값을 제공한다(나중에 설명). exc_info와 extra 모두 반드시 키워드 인수로 지정해야 한다.

로그 메시지를 생성할 때, 메시지를 생성하는 시점에는 문자열 포맷 지정을 수행하지 않는 것이 좋다(즉, 메시지의 포맷을 먼저 지정하고 그 결과를 로깅 모듈로 전달하지 않는 것이 좋다). 다음 예를 보자.

```
log.critical("Can't connect to %s at port %d" % (host, port))
```

함수나 메서드의 인수들은 먼저 평가되기 때문에 문자열 포맷 지정 연산은 항상 log.critical()를 호출하기 전에 이루어진다. 더 앞에 나온 예에서는 문자열 포맷 지정 연산에 사용하는 매개변수들이 단순히 logging 모듈에 전달될 뿐이고, 실제로 나중에 로그 메시지를 처리할 때에야 사용된다. 아주 미묘한 차이지만 많은 응용 프로그램에서 로그 메시지를 거르거나 디버깅 도중에만 로그 메시지를 출력하기 때문에, 메시지 생성 시점에 문자열 포맷 지정을 하지 않는 방법이 더 적은 일을 수행하며 로깅이 꺼져 있을 때는 더 빠르다.

이 메서드들 말고도 Logger 인스턴스 log에는 로그 메시지를 생성하기 위한 메서드들이 몇 가지 있다.

log.exception(fmt [, *args])

ERROR 수준의 메시지를 생성하며 현재 처리 중인 예외 정보도 포함시킨다. except 블록 안에서만 사용 가능하다.

log.log(level, fmt [, *args [, exc_info [, extra]]])

level로 지정한 수준의 로깅 메시지를 출력한다. 변수를 사용해 로깅 수준을 지정하고 싶거나 다섯 개의 기본 수준 이외의 로깅 수준을 사용하려는 경우 쓸 수 있다.

log.findCaller()

호출하는 쪽의 소스 파일 이름, 줄 번호, 함수 이름을 담은 튜플 (filename, lineno, funcname)을 반환한다. 이 정보는 로그 메시지를 생성할 때 유용할 때가 있다. 예를 들어, 이 정보를 가지고 로깅 호출이 일어난 곳의 정보를 로그 메시지에 추가할 수 있다.

로그 메시지 필터링

각 Logger 객체 log는 어떤 로그 메시지를 처리할지를 결정하기 위해서 내부 수준 정보와 그에 따른 필터링 메커니즘을 구현한다. 로그 메시지의 숫자로 나타낸 수준을 바탕으로 단순한 필터링을 수행하는 데 다음 두 메서드를 사용한다.

log.setLevel(level)

log의 수준을 설정한다. level보다 같거나 높은 수준의 로깅 메시지만 처리한다. 모든 다른 메시지는 무시한다. 기본 수준은 모든 로그 메시지를 처리하는 logging.NOTSET이다.

log.isEnabledFor(level)

level 수준을 가진 로그 메시지를 처리하는 경우 True를 반환한다.

로그 메시지를 메시지 자체와 관련된 정보(예를 들어, 파일 이름, 줄 번호나 기타 세부 정보)를 바탕으로 거를 수도 있다. 이를 위해 다음 메서드들을 사용한다.

log.addfilter(filter)

필터 객체 filter를 로거에 추가한다.

log.removeFilter(filter)

필터 객체 filter를 로거에서 제거한다.

앞의 두 메서드에서 filt는 Filter 객체 인스턴스이다.

Filter(logname)

logname 또는 그 자식에서 생성한 로그 메시지만 허용하는 필터를 생성한다. 예를 들어 logname이 ‘app’이면 ‘app’, ‘app.net’, ‘app.user’ 같은 로거에서 생성한 메시지는 통과되지만 ‘spam’과 같은 로거에서 생성한 메시지는 통과되지 않는다.

Filter를 상속한 다음 로그 메시지에 관한 정보를 담은 레코드를 입력으로 받는 filter(record) 메서드를 구현함으로써 커스텀 필터를 생성할 수 있다. 결과로 로그 메시지를 처리하는지 아닌지에 따라 True 또는 False를 반환해야 한다. 이 메서드에 전달되는 record 객체는 다음 속성들을 가진다.

속성	설명
record.name	로거 이름
record.levelname	수준 이름
record.levelno	숫자 수준
record.pathname	모듈 경로
record.filename	기반 파일 이름
record.module	모듈 이름
record.exc_info	예외 정보
record.lineno	로그 메시지가 생성된 줄 번호
record.funcName	로그 메시지가 생성된 함수 이름
record.created	생성 시간
record.thread	스레드 식별자
record.threadName	스레드 이름
record.process	현재 실행 중인 프로세스의 PID

다음 예는 커스텀 필터를 생성하는 방법을 보여준다.

```
class FilterFunc(logging.Filter):
    def __init__(self, name):
        self.funcName = name
    def filter(self, record):
        if record.funcName == self.funcName: return False
        else: return True
```

```
log.addFilter(FilterFunc('foo'))    # foo()에서 생성한 모든 메시지를 무시한다.
log.addFilter(FilterFunc('bar'))    # bar()에서 생성한 모든 메시지를 무시한다.
```

메시지 전달과 계층 로거

고급 로깅 응용에서는 Logger 객체를 계층적으로 조직할 수 있다. 로거 객체 이름을 ‘app.net.client’ 같이 주면 된다. 이때 실제로는 ‘app’, ‘app.net’, ‘app.net.client’

로 부르는 세 가지 Logger 객체가 만들어지게 된다. 어떤 메시지가 어느 한 로거에서 생성되어 필터를 무사히 통과하면 이 메시지는 부모 쪽으로 전달되어 처리된다. 예를 들어, 메시지가 ‘app.net.client’에서 성공적으로 생성되면 이 메시지는 ‘app.net’, ‘app’, 그리고 루트 로거에 전달된다.

Logger 객체 log에 있는 다음 속성과 메서드들을 사용하여 메시지 전파 여부를 제어한다.

log.propagate

메시지를 부모 로거로 전파할지 나타내는 불리언 플래그. 기본으로 True로 설정된다.

log.getEffectiveLevel()

로거의 실제 수준(effective level)을 반환한다. setLevel()을 사용하여 수준을 설정했으면 해당 수준을 반환한다. 수준을 직접 설정하지 않았다면(이 경우 수준은 logging.NOTSET) 부모의 실제 수준을 반환한다. 부모 로거 중 수준이 설정된 것이 하나도 없으면 루트 로거의 실제 수준을 반환한다.

계층 로거는 주로 큰 규모의 응용 프로그램 안 여러 곳에서 발생하는 로그 메시지를 손쉽게 필터링하기 위해서 사용한다. 예를 들어, 응용 프로그램 ‘app.net.client’에서 발생한 로그 메시지를 끄려면 다음 설정 코드를 추가하면 된다.

```
import logging
logging.getLogger('app.net.client').propagate = False
```

다음 코드처럼 가장 심각한 메시지만 처리하고 나머지는 무시하게 할 수도 있다.

```
import logging
logging.getLogger('app.net.client').setLevel(logging.CRITICAL)
```

계층 로거에서 특이한 점은 로그 메시지가 부모 로거의 수준이나 필터가 아니라 전적으로 로그 메시지를 생성한 Logger 객체의 수준과 필터에 기반하여 처리된다는 점이다. 따라서, 메시지가 초기 필터들의 집합을 통과하면 부모 로거의 필터나 수준에 관계 없이 모든 부모 로거에 전달되어 처리된다. 부모 로거의 필터에 걸리질 메시지라도 이렇게 처리된다. 언뜻 보기에 이러한 방식은 직관에 반하며 벼그인 것처럼 느껴진다. 하지만, 이 때문에 자식 로거의 수준을 부모 수준보다 하나 작은 값으로 설정하여 부모 설정을 덮어쓰는 일종의 수준 상승이 가능하다. 다음 예를 보자.

```

import logging

# 최상위 로거 'app'
log = logging.getLogger('app')
log.setLevel(logging.CRITICAL)    # CRITICAL 수준 메시지만 받음

# 자식 로거 'app.net'
net_log = logging.getLogger('app.net')
net_log.setLevel(logging.ERROR) # 'app.net'에서 ERROR 메시지를 받음
                                # 'app' 로거의 메시지 수준이 CRITICAL이더라도
                                # 'app' 로거에서 처리됨

```

계층 로거를 사용할 때는 필터링이나 전달 방식을 변경하기를 원하는 로깅 객체만을 설정하면 된다. 메시지들은 최종적으로 루트 로거에 전달되기 때문에 루트 로거는 출력을 생성할 책임이 있고 그곳에서는 basicConfig() 설정이 적용된다.

메시지 처리

보통 메시지는 루트 로거에서 처리된다. 하지만, 어느 Logger 객체에서든 특수한 처리기를 추가하여 로그 메시지를 받아서 처리할 수 있다. 이를 위해 Logger 인스턴스 log에 있는 다음 메서드들을 사용한다.

log.addHandler(handler)

로거에 Handler 객체를 추가한다.

log.removeHandler(handler)

로거에서 Handler 객체를 제거한다.

logging 모듈에는 로그 메시지를 파일, 스트림, 시스템 로그 등에 쓰는 다양한 내장 처리기가 있다. 여기에 관해서는 다음 절에서 자세히 설명한다. 다음 예는 앞에 나온 메서드를 사용해 로거와 처리기를 연결하는 방법을 보여준다.

```

import logging
import sys

# 'app'라는 최상위 로거를 생성한다.
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.propagate = False

# 'app' 로거에 메시지 처리기를 몇 개 추가한다.
app_log.addHandler(logging.FileHandler('app.log'))
app_log.addHandler(logging.StreamHandler(sys.stderr))

# 메시지 몇 개를 생성한다. 이들은 app.log와 sys.stderr로 보내진다.
app_log.critical("Creeping death detected!")

```

```
app_log.info("FYI")
```

보통 루트 로거의 기본 작동 방식을 변경하기 위해서 하위 로거에 처리기를 추가 한다. 그래서 앞 예에서도 app 로거의 메시지 전달 기능을 껐다(즉, ‘app’ 로거는 모든 메시지를 자신이 처리한다).

처리기 객체

logging 모듈에는 다양한 방법으로 로그 메시지를 처리하는 내장 처리기들이 있다. Logger 객체의 addHandler() 메서드로 이들을 추가할 수 있다. 각 처리기에는 필터와 수준을 설정할 수 있다.

내장 처리기

다음은 내장 처리기 객체들을 나열한 것이다. 몇몇 처리기는 하위 모듈 logger.handlers에 정의되어 있다. 필요에 따라 적절히 임포트해서 써야 한다.

handlers DatagramHandler(host, port)

주어진 host와 port에 있는 UDP 서버에 로그 메시지를 보낸다. 로그 메시지를 나타내는 LogRecord 객체의 속성 사전을 pickle 모듈로 인코딩하여 전달한다. 전달되는 네트워크 메시지는 네트워크 순서(빅 엔디안)로 된 4바이트 크기 길이와 피클링된 레코드 데이터로 구성된다. 메시지 재구성을 위해 수신자는 길이 부분을 벗기고 전체 메시지를 읽은 다음, 내용을 역피클링하고 logging.makeLogRecord()를 호출해야 한다. UDP는 신뢰성을 보장하지 않으므로 네트워크 에러가 발생하면 로그 메시지가 손실될 수 있다.

FileHandler(filename [, mode [, encoding [, delay]]])

파일 filename에 로그 메시지를 쓴다. mode는 파일을 열 때 사용할 파일 모드이며 기본 값은 ‘a’이다. encoding은 파일 인코딩을 나타낸다. delay는 불리언 플래그로서 True로 설정하면 첫 로그 메시지가 생성될 때까지 파일 열기를 미룬다. 기본 값은 False이다.

handlers HTTPHandler(host, url [, method])

HTTP GET 또는 POST 방식을 사용하여 로그 메시지를 HTTP 서버에 업로드 한다. host는 호스트 머신을 나타내고 url은 사용할 URL을 나타내며 method는

‘GET’(기본 값) 또는 ‘POST’ 중 하나이다. 로그 메시지를 보내기 위해 로그 메시지를 나타내는 LogRecord 객체의 속성 사전을 가져와서 urllib.urlencode() 함수를 사용해 URL 질의 문자열 변수로 인코딩한다.

handlers.MemoryHandler(capacity [, flushLevel [, target]])

이 처리기는 메모리에 로그 메시지를 수집하였다가 다른 처리기 target으로 주기적으로 내보낸다. capacity는 바이트 단위로 메모리 버퍼 크기를 나타낸다. flushLevel은 숫자 로깅 수준으로 해당 수준 이상의 메시지가 나타날 경우 강제로 메모리 내보내기를 수행하게 한다. 기본 값은 ERROR이다. target은 메시지를 받을 다른 Handler 객체이다. target을 생략한 경우 이 처리기가 일을 시작하게 만들려면 setTarget() 메서드로 대상 처리기를 설정해야 한다.

handlers.NTEventLogHandler(appname [, dllname [, logtype]])

원도 NT, 원도 2000, 원도 XP에서 메시지를 이벤트 로그로 보낸다. appname은 이벤트 로그에서 사용할 응용 프로그램 이름이다. dllname은 로그 메시지에 담길 데이터 정의를 담은 .DLL 또는 .EXE 파일의 절대 경로를 나타낸다. 이 값을 생략할 경우 dllname은 ‘win32service.pyd’로 설정된다. logtype은 ‘Application’, ‘System’, ‘Security’ 중 하나이다. 기본 값은 ‘Application’이다. 이 처리기는 파이썬 Win32 확장 기능이 설치된 경우에만 사용할 수 있다.

handlers.RotatingFileHandler(filename [, mode [, maxBytes [, backupCount [, encoding [, delay]]]]])

파일 filename에 로그 메시지를 쓴다. 파일이 maxBytes로 지정된 크기보다 클 경우 기존 파일을 filename.1로 옮기고 새로운 로그 파일 filename을 연다. backupCount는 생성할 백업 파일의 최대 개수를 지정한다. backupCount의 기본 값은 0이다. 이 값을 지정하면 백업 파일을 filename.1, filename.2, ..., filename.N 순서로 교체한다. 여기서 filename.1은 가장 최신 백업 파일, filename.N은 가장 오래된 백업 파일을 나타낸다. mode는 파일을 열 때 사용할 파일 모드를 지정한다. 기본 값은 ‘a’이다. maxBytes가 0이면(기본 값) 로그 파일을 새로운 파일로 옮기지 않기 때문에 무한히 커질 수 있다. encoding과 delay는 FileHandler에서 의미와 같다.

handlers.SMTPHandler(mailhost, fromaddr, toaddrs, subject [, credentials])

이메일을 사용하여 로그 메시지를 원격 호스트로 보낸다. mailhost는 메시지

를 받을 SMTP 서버 주소이다. 이 주소는 간단한 호스트 이름을 담은 문자열이거나 튜플 (host, port)일 수 있다. fromaddr은 보낸 주소이고 toaddrs은 목적지 주소이며 subject는 메시지 제목이다. credentials은 사용자 이름과 암호를 담은 튜플 (username, password)이다.

handlers.SocketHandler(host, port)

TCP 소켓 연결을 사용하여 로그 메시지를 원격 호스트로 보낸다. host와 port는 목적지를 지정한다. 메시지는 DatagramHandler에서 설명한 것과 같은 형식으로 전달된다. DatagramHandler와 달리 이 처리기는 로그 메시지를 신뢰성 있게 전달한다.

StreamHandler([fileobj])

이미 열린 파일과 비슷한 객체 fileobj에 로그 메시지를 쓴다. 아무 인수를 제공하지 않으면 메시지를 sys.stderr에 쓴다.

handlers.SysLogHandler([address [, facility]])

로그 메시지를 유닉스 시스템 로깅 데몬에 보낸다. address는 (host, port)로 목적지를 나타낸다. 생략하면 ('localhost', 512)를 사용한다. facility는 정수 시설 코드로 기본 값으로 SysLogHandler.LOG_USER를 사용한다. 전체 시설 코드 목록은 SysLogHandler 정의에 나와 있다.

handlers.TimedRotatingFileHandler(filename [, when [, interval [, backupCount [, encoding [, delay [, utc]]]]]])

RotatingFileHandler와 같지만 파일 크기가 아닌 시간에 따라 파일을 교체한다. interval은 숫자이고 when은 단위를 담은 문자열이다. when에 사용할 수 있는 값은 'S'(초), 'M'(분), 'H'(시간), 'D'(날짜), 'W'(주), 'midnight'(자정마다 교체)이다. 예를 들어, interval을 3으로, when을 'D'로 설정하면 매 3일마다 로그 파일을 교체한다. backupCount는 유지할 백업 파일의 최대 개수를 나타낸다. utc는 지역 시간(기본 값)을 사용할지 UTC 시간을 사용할지를 지정하는 불리언 플래그이다.

handlers.WatchedFileHandler(filename [, mode [, encoding [, delay]]])

FileHandler와 같지만 열린 로그 파일의 아이노드(inode)와 장치를 감시한다. 로그 메시지가 마지막으로 생성된 아래로 파일이 변경된 경우 파일을 닫고 같은 파일 이름으로 파일을 다시 연다. 프로그램 외부에서 로그 교체 연산이 수행되어 로그 파일이 삭제되었거나 옮겨졌을 때 이런 일이 발생할 수 있다. 이 처리기는 유닉스 시

스템에서만 작동한다.

처리기 설정

각 Handler 객체 h는 자신만의 수준과 필터를 가질 수 있다. 이를 설정하기 위해 다음 메서드들을 사용한다.

h.setLevel(level)

처리할 메시지의 임계 수준을 설정한다. level은 ERROR 또는 CRITICAL과 같은 숫자 코드이다.

h.addFilter(filt)

처리기에 Filter 객체 filt를 추가한다. 더 자세한 설명은 Logger 객체의 addFilter() 메서드를 참고한다.

h.removeFilter(filt)

Filter 객체 filt를 처리기에서 제거한다.

처리기가 붙어 있는 Logger 객체의 수준이나 필터와 별도로 처리기에 수준과 필터를 설정할 수 있다는 점을 유념하기 바란다. 다음 예를 보자.

```
import logging
import sys

# CRITICAL 수준 메시지를 stderr로 출력하는 처리기를 생성한다.
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)

# 'app'로 불리는 최상위 로거를 생성한다.
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.addHandler(logging.FileHandler('app.log'))
app_log.addHandler(crit_hand)
```

이 예에서는 INFO 수준을 사용하는 ‘app’라고 불리는 로거 하나가 있다. 이 로거에 처리기 두 개를 추가하였는데 하나(crit_hand)는 수준이 CRITICAL로 설정되었다. 이 처리기는 INFO 수준 이상인 로그 메시지를 받지만 CRITICAL이 아닌 것은 버린다.

처리기 정리

다음 메서드들은 처리기를 정리하는 데 사용한다.

h.flush()

모든 로그 출력을 비운다.

h.close()

처리기를 닫는다.

메시지 포맷 지정

기본으로 Handler 객체는 로깅 요청이 들어오면 지정한 형식대로 로그 메시지를 방출한다. 하지만 종종 타임스탬프, 파일 이름, 줄 번호 등과 같은 추가 문맥 정보를 메시지에 추가하고 싶을 때가 있다. 이 절에서는 이러한 정보를 로그 메시지에 자동으로 추가하는 방법을 설명한다.

포맷 지정기 객체

로그 메시지 포맷을 변경하려면 먼저 Formatter 객체를 생성해야 한다.

Formatter([fmt [, datefmt]])

새로운 Formatter 객체를 생성한다. fmt는 메시지에 적용할 포맷 문자열이다. fmt에는 앞에서 basicConfig() 함수를 설명할 때 언급한 다양한 확장 연산을 지원한다. datefmt는 time.strftime() 함수와 호환되는 날짜 포맷 문자열이어야 한다. 이 값을 생략할 경우 날짜 포맷은 ISO8601로 설정된다.

Formatter 객체를 처리기 객체에 붙여야 포맷 지정 효과가 나타낸다. 이를 위해 Handler 인스턴스 h의 h.setFormatter() 메서드를 사용하면 된다.

h.setFormatter(format)

Handler 인스턴스 h에 로그 메시지를 생성할 때 사용할 메시지 포맷 지정기 객체를 설정한다. format은 반드시 Formatter 인스턴스여야 한다.

다음은 처리기에서 로그 메시지 포맷을 커스터마이즈하는 방법을 보여준다.

```
import logging
import sys

# 메시지 포맷 설정
format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")
```

```
s")  
  
# CRITICAL 수준인 메시지를 stderr에 출력하는 처리기를 생성한다.  
crit_hand = logging.StreamHandler(sys.stderr)  
crit_hand.setLevel(logging.CRITICAL)  
crit_hand.setFormatter(format)
```

이 예에서 crit_hand 처리기에 커스텀 포맷 지정기를 설정했다. 이 처리기는 로그 메시지 “Creeping death detected.”를 다음과 같이 출력한다.

```
CRITICAL 2005-10-25 20:46:57,126 Creeping death detected.
```

추가 문맥 정보를 메시지에 더하기

특정 응용에서는 로그 메시지에 추가 문맥 정보를 더하는 것이 유용할 때가 있다. 추가 정보를 더하는 데는 두 가지 방법 중 하나를 사용할 수 있다. 먼저, 모든 기본 로깅 연산(log.critical(), log.warning() 등)에는 메시지 포맷 문자열에서 사용할 추가 필드들을 담은 사전인 extra 매개변수를 지정할 수 있다. 이 필드들은 앞에서 언급한 Formatter 객체와 함께 사용된다. 다음 예를 보자.

```
import logging, socket  
logging.basicConfig(  
    format = "%(hostname)s %(levelname)-10s %(asctime)s %(message)s"  
)  
  
# 추가 문맥 정보  
netinfo = {  
    'hostname' : socket.gethostname(),  
    'ip' : socket.gethostbyname(socket.gethostname())  
}  
  
log = logging.getLogger('app')  
  
# 추가 문맥 데이터를 담은 로그 메시지를 생성한다.  
log.critical("Could not connect to server", extra=netinfo)
```

이 방법은 추가 정보를 모든 로깅 연산에 지정해주어야 하고 그렇지 않을 경우 프로그램이 다운될 수 있다는 단점이 있다. 다른 방법으로는 LogAdapter 클래스를 기준 로거의 래퍼로서 사용하는 방법이 있다.

LogAdapter(log [, extra])

Logger 객체 log를 감싼 래퍼를 생성한다. extra는 메시지 포맷 지정기에 제공할 추가 문맥 정보를 담은 사전이다. LogAdapter 인스턴스는 Logger 객체와 동일한 인터페이스를 가진다. 로그 메시지를 생성하는 연산에 자동으로 extra로 제공한 추

가 정보가 더해진다.

다음은 LogAdapter 객체를 사용하는 예이다.

```
import logging, socket
logging.basicConfig(
    format = "%(hostname)s %(levelname)-10s %(asctime)s %(message)s"
)

# 추가 맴버 정보
netinfo = {
    'hostname' : socket.gethostname(),
    'ip' : socket.gethostbyname(socket.gethostname())
}

# 로거 생성
log = logging.LogAdapter(logging.getLogger("app"), netinfo)

# 로그 메시지 생성. LogAdapter가 추가 맴버 데이터를 알아서 더한다.
log.critical("Could not connect to server")
```

기타 유ти리티 함수

logging에 있는 다음 함수들은 로깅의 몇 가지 측면을 제어한다.

disable(level)

수준 level 아래인 모든 로깅 메시지를 전역적으로 비활성화시킨다. 응용 프로그램 전체에 걸쳐 로깅을 끄는 데 사용할 수 있다. 예를 들어, 임시로 로깅을 끈다거나 로깅 출력량을 줄이는 데 사용할 수 있다.

addLevelName(level, levelName)

완전히 새로운 로깅 수준과 이름을 생성한다. level은 숫자이고 levelName은 문자열이다. 내장 수준의 이름을 변경하거나 기본으로 제공되는 것 이외에 수준을 더 추가할 때 사용할 수 있다.

getLevelName(level)

숫자 값 level에 해당하는 수준 이름을 반환한다.

shutdown()

모든 로깅 객체를 종료시키고 필요한 경우 출력을 비운다.

로깅 설정

일반적으로 응용 프로그램에서 logging 모듈을 설정할 때는 다음에 나오는 기본적인 단계를 따른다.

1. getLogger()를 사용해 다양한 Logger 객체를 생성한다. 수준과 같은 매개변수들을 적절히 설정한다.
2. 다양한 처리기 타입들(FileHandler, StreamHandler, SocketHandler 등) 중 하나를 골라 인스턴스를 생성하여 Handler 객체를 얻고 적절히 수준을 설정한다.
3. 메시지 포맷 지정자 객체를 생성하고 setFormatter() 메서드로 처리기 객체에 붙인다.
4. addHandler() 메서드로 Handler 객체를 Logger 객체에 붙인다.

각 단계에서 해야 하는 작업이 꽤 복잡하기 때문에 모든 설정 부분을 하나의 잘 문서화된 파일에 넣어두는 것이 좋다. 예를 들어, 다음과 같이 applogconfig.py 파일을 생성하고 응용 프로그램의 주 프로그램에서 임포트하는 방법을 사용할 수 있다.

```
# applogconfig.py
import logging
import sys

# 메시지 포맷 설정
format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")

# CRITICAL 수준인 메시지를 stderr에 출력하는 처리기를 생성한다.
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(format)

# 메시지를 파일에 출력하는 처리기를 생성한다.
applog_hand = logging.FileHandler('app.log')
applog_hand.setFormatter(format)

# 'app'라고 부르는 최상위 로거를 생성한다.
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.addHandler(applog_hand)
app_log.addHandler(crit_hand)

# 'app.net' 로거의 수준을 변경한다.
logging.getLogger("app.net").setLevel(logging.ERROR)
```

모든 것을 한곳에 두게 되면 특정 로깅 설정을 변경해야 하는 경우에 편리하다.

프로그램에서 해당 파일을 한 번만 임포트하면 된다. 기타 다른 곳에서 로그 메시지를 생성하고 싶다면 간단히 다음과 같이 한다.

```
import logging
app_log = logging.getLogger("app")
...
app_log.critical("An error occurred")
```

하위 모듈 `logging.config`

로깅 설정 부분을 파이썬 코드로 짤 수도 있지만 로깅 모듈을 INI 형식 설정 파일을 사용하여 설정할 수도 있다. `logging.config`에 있는 다음 함수들을 사용하면 된다.

```
fileConfig(filename [, defaults [, disable_existing_loggers]])
```

설정 파일 `filename`에서 로깅 설정을 읽는다. `defaults`는 설정 파일에서 사용할 기본 설정 매개변수들을 담은 사전이다. 지정된 파일을 `ConfigParser` 모듈을 사용하여 읽는다. `disable_existing_loggers`는 새로운 설정 데이터를 읽었을 때 기존 로거를 비활성화 할 것인지를 지정하는 불리언 플래그이다. 기본 값은 `True`이다.

`logging` 모듈의 온라인 문서에는 설정 파일을 어떻게 작성해야 하는지에 관해 자세하게 논의하고 있다. 경험 많은 프로그래머라면 다음에 나오는 앞 절에서 본 `applogconfig.py`를 설정 파일 버전으로 바꾼 것을 보면 어떻게 하면 되는지를 알 수 있을 것이다.

```
; applogconfig.ini
;
; 로깅 설정을 위한 설정 파일
;
; 다음 구역들은 파일에서 나중에 설정할 Logger, Handler, Formatter
; 객체들의 이름을 지정한다.

[loggers]
keys=root,app,app_net

[handlers]
keys=crit,applog

[formatters]
keys=format

[logger_root]
level=NOTSET
handlers=

[logger_app]
level=INFO
```

```

propagate=0
qualname=app
handlers=crit,applog

[logger_app_net]
level=ERROR
propagate=1
qualname=app.net
handlers=

[handler_crit]
class=StreamHandler
level=CRITICAL
formatter=format
args=(sys.stderr,)

[handler_applog]
class=FileHandler
level=NOTSET
formatter=format
args=('app.log',)

[formatter_format]
format=%(levelname)-10s %(asctime)s %(message)s
datefmt=

```

이 설정 파일을 읽어 로깅 기능을 설정하려면 다음과 같이 한다.

```
import logging.config
logging.config.fileConfig('applogconfig.ini')
```

이전과 마찬가지로 로그 메시지를 생성하려는 모듈에서는 로깅 설정 정보를 어떻게 읽어와야 하는지에 관해 걱정할 필요 없다. 간단히 logging 모듈을 임포트하고 적절한 Logger 객체의 참조를 얻기만 하면 된다. 다음 예를 보자.

```
import logging
app_log = logging.getLogger("app")
...
app_log.critical("An error occurred")
```

성능 고려

응용 프로그램에 로깅 기능을 추가할 때 주의를 기울이지 않으면 성능이 크게 나빠질 수 있다. 이곳에서는 로깅으로 인한 오버헤드를 감소시킬 수 있는 몇 가지 방법을 소개한다.

첫째, 파이썬 최적화 모드(-O 옵션)는 `if __debug__:` 문장들 같은 문장으로 작성한 조건적으로 실행되는 모든 코드를 제거한다. 디버깅을 위해서만 로깅을 사용하고 있는 중이라면 모든 로깅 부분을 조건적으로 작성하고 최적화 모드를 통해 제

거하는 것이 좋다.

둘째, 로깅을 아예 사용하지 않을 것이라면 Logger 객체 자리에 Null 객체를 사용한다. 이것은 None을 사용하는 경우와 다르다. 우리가 원하는 것은 모든 연산을 조용히 무시하는 객체이다. 다음 예를 보자.

```
class Null(object):
    def __init__(self, *args, **kwargs): pass
    def __call__(self, *args, **kwargs): return self
    def __getattribute__(self, name): return self
    def __setattr__(self, name, value): pass
    def __delattr__(self, name): pass

log = Null()
log.critical("An error occurred.") # 아무 것도 안 함
```

좀 더 고민하면 장식자나 메타클래스를 사용하여 로깅 기능을 관리할 수도 있다. 파이썬에서 이들은 함수, 메서드, 클래스가 정의되는 순간에 실행되기 때문에 로깅을 사용하지 않을 때는 성능에 영향을 주지 않으면서 프로그램의 특정 부분에서 로깅 기능을 선택적으로 추가하거나 제거할 수 있다. 자세한 내용은 6장과 7장을 참고한다.

Note

- logging 모듈에는 여기서 다루지 않은 커스터마이즈를 위한 수많은 옵션을 제공한다. 자세한 내용은 온라인 문서를 참고하도록 한다.
- 스레드를 사용하는 프로그램에서는 logging 모듈을 사용하는 것이 안전하다. 특히, 로그 메시지를 생성하는 코드 주위에 락 연산을 추가할 필요가 없다.

mmap

mmap 모듈은 메모리 매핑(memory-mapped) 파일 객체를 지원한다. 이 객체는 파일 및 바이트 문자열과 비슷하게 작동하며 보통 파일 또는 바이트 문자열이 사용되는 곳에 대부분 사용할 수 있다. 또한 메모리 매핑 파일의 내용을 변경할 수 있다. 따라서 색인 대입과 분할 대입을 사용하여 수정할 수 있다. 파일에 개인(private) 매핑이 적용된 경우가 아니면 이러한 수정 사항은 내부 파일에 바로 반영된다.

메모리 매핑 파일은 mmap() 함수로 생성하며 유닉스와 윈도에서 사용법이 약간 다르다.

```
mmap(fileno, length [, flags, [prot [, access [, offset]]]])
```

(유닉스). 정수 파일 기술자 fileno를 가진 파일에서 length 바이트를 메모리로 매핑한 mmap 객체를 반환한다. fileno가 -1이면 익명 메모리에 매핑된다. flag는 매핑 특성을 나타내며 다음 중 하나가 될 수 있다.

플래그	의미
MAP_PRIVATE	쓸 때 복사(copy-on-write) 방식의 개인 매핑을 생성한다. 이 매핑에 대한 변경은 이 프로세스에만 보인다.
MAP_SHARED	파일에서 같은 지역을 매핑하는 다른 프로세스들과 매핑을 공유한다. 이 매핑에 대한 변경은 모든 다른 매핑에 영향을 준다.

기본 flags 설정은 MAP_SHARED이다. prot는 객체 메모리 보호 방식을 나타내며 다음 값들을 비트 OR한 값이다.

설정	의미
PROT_READ	객체에서 데이터를 읽을 수 있다.
PROT_WRITE	객체를 변경할 수 있다.
PROT_EXEC	객체가 실행 가능한 명령을 포함할 수 있다.

prot의 기본 값은 PROT_READ | PROT_WRITE이다. prot로 지정한 모드는 내부 파일 기술자 fileno을 열 때 사용한 파일 접근 권한과 일치해야 한다. 보통 파일을 읽기 쓰기 모드로 여는 것이 좋다(예를 들어, os.open(name, os.O_RDWR)).

flags와 prot 대신 옵션인 access 매개변수를 사용할 수 있다. 이 매개변수는 다음 값 중 하나를 가질 수 있다(값이 주어지면).

접근 모드	의미
ACCESS_READ	읽기 전용 접근
ACCESS_WRITE	바로 쓰기(write-through) 방식으로 읽기 쓰기 접근. 변경한 내용은 파일에 영향을 준다.
ACCESS_COPY	쓸 때 복사 방식으로 읽기 쓰기 접근. 변경은 메모리에만 영향을 주고 파일을 변경하지는 않는다.

access는 보통 키워드 인수로 지정한다(예를 들어, mmap(fileno, length, access=ACCESS_READ)). access와 flags 값 둘 모두를 지정하면 에러가 발생한다. offset 매개변수는 파일 시작에서 바이트 수를 나타내고 기본 값은 0이다. 반드시 mmap.ALLOCATIONGRANULARITY의 배수여야 한다.

```
mmap(fileno, length[, tagname [,access [, offset]]])
```

(원도). 정수 파일 기술자 fileno를 가진 파일에서 length 바이트를 매핑한 mmap

객체를 반환한다. 익명 메모리에 대해서는 fileno로 -1을 사용하면 된다. length가 현재 파일 크기보다 크면 파일이 length 바이트로 확장된다. length가 0이면 파일이 비어 있지 않은 한 현재 파일 길이를 길이로 사용한다(파일이 비어 있으면 예외가 발생한다). tagname은 매핑 이름으로 사용되는 옵션인 문자열이다. tagname이 기존 매핑을 가리키면 해당 매핑이 열린다. 그렇지 않으면 새로운 매핑이 생성된다. tagname이 None이면 이름 없는 매핑이 생성된다. access는 옵션인 매개변수로 접근 모드를 나타낸다. 앞에서 본 유닉스 버전 mmap()과 동일한 값을 가진다. 기본으로 access는 ACCESS_WRITE이다. offset은 파일 시작에서 바이트 수를 나타내고 기본은 0이다. 반드시 mmap.ALLOCATIONGRANULARITY의 배수여야 한다.

메모리 매핑 파일 객체 m은 다음 메서드들을 제공한다.

m.close()

파일을 닫는다. 이어지는 연산은 예외를 발생시킨다.

m.find(string [, start])

string이 첫 번째로 나타낸 곳의 색인을 반환한다. start는 옵션으로서 시작 위치를 지정한다. 매칭된 것이 없을 경우 -1을 반환한다.

m.flush([offset, size])

메모리 복사본에 가해진 수정 사항을 파일 시스템으로 복사한다. 옵션인 offset과 size는 파일 시스템으로 복사할 바이트 범위를 지정한다. 지정하지 않으면 모든 매핑 내용이 쓰여진다.

m.move(dst, src, count)

src에서 시작해서 count 바이트만큼을 dst로 복사한다. 복사를 수행하는 데 C memmove() 함수를 사용하고 두 지역이 겹쳐 있어도 제대로 작동하는 것이 보장된다.

m.read(n)

현재 파일 위치에서 최대 n 바이트까지 읽은 데이터를 문자열로 반환한다.

m.read_byte()

현재 파일 위치에서 한 바이트를 읽고 길이가 1인 문자열로 반환한다.

m.readline()

현재 파일 위치에서 시작하는 한 줄을 반환한다.

m.resize(newsize)

메모리 매핑 객체가 newsize 바이트만큼 담도록 크기를 조정한다.

m.seek(pos [, whence])

파일 위치를 새로운 값으로 설정한다. pos와 whence는 파일 객체의 seek() 메서드에서 의미와 같다.

m.size()

파일 길이를 반환한다. 이 값은 메모리 매핑 공간 크기보다 더 클 수 있다.

m.tell()

파일 포인터 값을 반환한다.

m.write(string)

바이트 문자열을 현재 파일 포인터 위치에 쓴다.

m.write_byte(byte)

한 바이트를 현재 파일 포인터에 해당하는 메모리에 쓴다.

Note

- mmap() 함수가 유닉스와 윈도에서 약간 다르지만 두 함수 모두에 있는 access 매개변수를 사용하면 이 모듈을 이식성 있게 사용할 수 있다. 예를 들어, mmap(fileno,length,access=ACCESS_WRITE)은 유닉스와 윈도 둘 모두에서 작동한다.
- 어떤 경우에는 메모리 매핑이 상수 mmap.PAGESIZE에 저장된 시스템 페이지 크기의 배수 길이일 때만 작동한다.
- 유닉스 SVR 4에서 익명 매핑 메모리는 파일 /dev/zero를 적절한 권한으로 열고 mmap()을 호출하여 얻을 수 있다.
- 유닉스 BSD 시스템에서 익명 매핑 메모리는 mmap()을 음수 파일 기술자와 플래그 mmap.MAP_ANON으로 호출하여 얻을 수 있다.

msvcrt

msvcrt 모듈은 마이크로소프트 비주얼 C 런타임 라이브러리에 있는 유용한 몇몇 함수에 접근할 수 있는 기능을 제공한다. 이 모듈은 윈도에서만 사용할 수 있다.

getch()

키 놀림(keypress)를 읽고 결과 문자를 반환한다. 키 놀림이 없으면 기다린다. 입력된 키가 특수 기능 키이면 '\000' 또는 '\xe0'을 반환하고 다음 호출 때 키 코드를 반환한다. 이 함수는 문자를 콘솔에 출력하지 않으며 Ctrl+C를 인식하는 데 사용할 수 없다.

getwch()

유니코드 문자를 반환하는 것을 제외하면 getch()와 같다.

getche()

입력한 문자를 출력(출력할 수 있는 경우)하는 것 말고는 getch()와 비슷하다.

getwche()

유니코드 문자를 반환하는 것을 제외하면 getche() 와 같다.

get_osfhandle(fd)

파일 기술자 fd에 대한 파일 핸들러를 반환한다. fd를 인식할 수 없으면 IOError가 발생한다.

heapmin()

내부 파이썬 메모리 관리자에서 사용하지 않는 블록을 운영체제에 강제로 반환하게 한다. 이는 윈도 NT에서만 작동하며 실패하면 IOError가 발생한다.

kbhit()

키 놀림을 기다리고 있으면 True를 반환한다.

locking(fd, mode, nbytes)

C 런타임에서 온 파일 기술자가 주어졌을 때 파일의 일부를 잠근다. nbytes는 현재 파일 포인터에 상대적인 잠글 바이트 수이다. mode는 다음 정수 중 하나가 될 수 있다.

설정	설명
0	파일 영역을 해제(LK_UNLCK)
1	파일 영역을 잠금(LK_LOCK)
2	파일 영역을 잠금. 넌블로킹(LK_NBLOCK)
3	쓰기용 잠금(LK_RLOCK)
4	쓰기용 잠금 넌블로킹(LK_NBRLCK)

락을 획득하는 데 대략 10초 이상 걸리면 IOError 예외가 발생한다.

open_osfhandle(handle, flags)

파일 핸들 handle로 C 런타임 파일 기술자를 생성한다. flags는 os.O_APPEND, os.O_RDONLY, os.O_TEXT 값은 비트 OR한 것이다. 파일 객체를 생성하기 위한 os.fdopen()의 매개변수로 사용할 수 있는 정수 파일 기술자를 반환한다.

putch(char)

문자 char를 버퍼링 없이 콘솔에 출력한다.

putwch(char)

char가 유니코드 문자인 것을 제외하고 putch()와 같다.

setmode(fd, flags)

파일 기술자 fd에 줄 끝 변환 모드를 설정한다. flags는 텍스트 모드일 경우 os.O_TEXT, 이진 모드일 경우 os.O_BINARY이다.

ungetch(char)

문자 char를 콘솔 버퍼에 다시 집어넣는다. 다음 getch() 또는 getche()에서 이 문자를 읽는다.

ungetwch(char)

char가 유니코드 문자인 것을 제외하고 ungetch()와 같다.

Note

MFC(Windows Foundation Classes), COM 컴포넌트, GUI 등에 접근할 수 있게 하는 다양한 Win32 확장 기능이 존재한다. 여기에 관한 주제는 이 책의 범위를 많이 벗어난다. 마크 해몬드(Mark Hammond)와 앤디 로빈슨(Andy Robinson)이 쓴 책, 《Win32에서 파이썬 프로그래밍 (Python Programming on Win32)》(O'Reilly & Associates, 2000)이 이 주제에 관해서 다세히 다루고 있다. 또한 <http://www.python.org>에 가면 윈도에서 사용할 수 있는 모듈에 관한 광범위한 정보를 얻을 수 있다.

참고

winreg(502페이지)

optparse

optparse 모듈은 sys.argv에 들어 있는 유닉스 스타일 명령줄 옵션을 처리하는 데 사용할 수 있는 고수준 기능을 제공한다. 9장에서 이 모듈의 간단한 사용 예를 살펴보았다. optparse를 사용할 때는 OptionParser 클래스를 주로 사용한다.

OptionParser([args])**

새로운 명령 옵션 파서를 생성하고 OptionParser 인스턴스를 반환한다. 설정을 위해 옵션인 다양한 키워드 인수들을 사용할 수 있다. 다음 목록은 키워드 인수들을 설명한다.

키워드 인수	설명
<code>add_help_option</code>	특수 도움말 옵션(<code>--help</code> , <code>-h</code>)을 지원할지를 지정한다. 기본으로 True이다.
<code>conflict_handler</code>	서로 충돌하는 명령을 처리하는 방식을 지정한다. ‘error’(기본 값) 또는 ‘resolve’ 중 하나로 설정한다. ‘error’ 모드에서는 서로 충돌하는 옵션 문자열이 파서에 추가되면 optparse.OptionConflictError 예외가 발생한다. ‘resolve’ 모드에서는 나중에 추가된 옵션이 우선 순위를 가진다. 하지만, 먼저 추가된 옵션이 여러 이름으로 추가되었고 충돌이 발생하지 않은 이름이 적어도 하나 이상 있다면 해당 옵션을 여전히 사용할 수 있다.
<code>description</code>	도움말에서 보여줄 프로그램 설명을 담은 문자열. 이 문자열은 출력될 때 화면에 맞게 적절히 포맷된다.
<code>formatter</code>	도움말을 출력할 때 텍스트 포맷을 지정하는 데 사용할 optparse.HelpFormatter 인스턴스. optparse.IndentedHelpFormatter(기본 값) 또는 optparse.TitledHelpFormatter 중 하나이다.
<code>option_class</code>	각 명령줄 옵션에 대한 정보를 나타내는 파이썬 클래스. 기본 클래스는 optparse.Option이다.
<code>option_list</code>	파서를 채울 옵션 리스트. 기본으로 이 리스트는 비어 있으며 옵션은 add_option() 메서드로 추가한다. 이 인수가 주어질 경우 타입이 Option인 객체들을 담고 있어야 한다.
<code>prog</code>	도움말에서 ‘%prog’ 부분을 대체할 프로그램 이름
<code>usage --help</code>	옵션을 사용했거나 적절하지 않은 옵션이 전달되었을 때 출력되는 사용법을 담은 문자열. 기본 값은 ‘%prog [options]’ 문자열이다. 여기서 ‘%prog’ 키워드는 os.path.basename
<code>(sys.argv[0])</code>	값 또는 (제공될 경우) prog 키워드 인수 값으로 대체된다. optparse.SUPPRESS_USAGE 값을 주면 사용법을 아예 출력하지 않는다.
<code>version</code>	<code>--version</code> 옵션을 사용했을 때 출력할 버전 문자열. 기본으로 version은 None이고 <code>--version</code> 옵션은 사용할 수 없다. 이 문자열을 제공하면 <code>--version</code> 옵션이 자동으로 추가된다. 키워드 ‘%prog’는 프로그램 이름으로 대체된다.

아무 커스터마이즈를 하지 않은 경우에는 보통 인수 없이 OptionParser를 생성한다. 다음 예를 보자.

```
p = optparse.OptionParser()
```

OptionParser 인스턴스 p는 다음 메서드들을 제공한다.

```
p.add_option(name1, ..., nameN [, **parms])
```

p에 새로운 옵션을 추가한다. 인수 name1, name2 등은 이 옵션에 사용할 다양한 이름을 나타낸다. 예를 들어, ‘-f’와 ‘-file’ 같은 짧거나 긴 옵션 이름을 사용할 수 있다. 옵션 이름들 다음에는 옵션을 어떻게 처리할지를 지정하는 생략할 수 있는 키워드 인수들이 나온다. 다음은 키워드 인수들을 설명한다.

키워드 인수	설명
<code>action</code>	옵션을 파싱할 때 수행할 동작. 가능한 값은 다음과 같다. ‘store’ – 옵션이 읽어서 저장할 인수를 가진다. action을 지정하지 않으면 기본으로 사용된다. ‘store_const’ – 옵션에는 인수가 없지만 옵션이 나타나면 const 키워드 인수로 지정한 상수 값이 저장된다. ‘store_true’ – ‘store_const’와 비슷하지만 옵션을 파싱할 때 불리언 True를 저장한다. ‘store_false’ – ‘store_true’와 비슷하지만 False를 저장한다. ‘append’ – 옵션이 리스트에 추가될 인수를 가진다. 같은 명령 줄 옵션으로 여러 값을 지정할 때 사용한다. ‘count’ – 옵션에 인수가 없으며 카운터 값이 저장된다. 옵션이 나타날 때마다 카운터 값이 하나씩 증가한다. ‘callback’ – 옵션을 만나면 callback 키워드 인수로 지정한 역호출 함수를 호출한다. ‘help’ – 옵션을 파싱할 때 도움말을 출력한다. 표준 옵션 <code>-h</code> 또는 <code>--help</code> 가 아닌 다른 옵션으로 도움말을 출력하려고 할 때 필요하다. ‘version’ – OptionParser()에 제공된 버전 번호를 출력한다(값이 있으면). 표준 옵션 <code>-v</code> 또는 <code>--version</code> 대신 다른 옵션을 사용하여 버전 정보를 출력하려고 할 때 필요하다.
<code>callback</code>	옵션을 만났을 때 호출할 역호출 함수를 지정한다. 이 역호출 함수는 호출 가능한 파이썬 객체로서 <code>callback(option, opt_str, value, parser, *args, **kwargs)</code> 형태로 호출된다. 여기서 option 인수는 optparse.Option 인스턴스이다. opt_str은 명령줄에서 역호출을 발생 시킨 옵션 문자열이고 value는 옵션 값이다(존재하면). parser는 실행 중인 OptionParser 인스턴스이고 args는 <code>callback_args</code> 키워드 인수로 지정한 위치 인수들이며 kwargs는 <code>callback_kwargs</code> 키워드 인수로 지정한 키워드 인수들이다.

<code>callback_args</code>	callback 인수로 지정한 역호출 함수에 제공되는 옵션인 위치 인수들.
<code>callback_kwargs</code>	callback 인수로 지정한 역호출 함수에 제공되는 옵션인 키워드 인수들.
<code>choices</code>	모든 가능한 옵션 값을 담은 문자열 리스트. 옵션이 제한된 값만 가질 수 있을 때 사용한다(예를 들어 ['small', 'medium', 'large']).
<code>const</code>	'store_const' 동작을 수행할 때 저장할 상수 값
<code>default</code>	옵션이 주어지지 않았을 때 사용할 기본 값. 기본으로 None.
<code>dest</code>	파싱 도중에 옵션 값을 저장할 속성 이름. 보통 이 이름에 옵션 이름을 사용한다.
<code>help</code>	이 옵션의 도움말. 지정하지 않으면 도움말을 출력할 때 설명 없이 옵션을 나열한다. 옵션을 숨기려면 optparse.SUPPRESS_HELP를 사용한다. 도움말에서 '%default' 부분은 옵션의 기본 값으로 대체된다.
<code>metavar</code>	도움말을 출력할 때 사용할 옵션 인수 이름을 지정한다.
<code>nargs</code>	인수를 기대하는 동작에 대한 옵션 인수의 개수를 지정한다. 기본 값은 1이다. 1보다 큰 값을 사용하면 옵션 인수들을 튜플에 저장했다가 나중에 인수를 처리할 때 사용한다.
<code>type</code>	옵션 타입을 지정한다. 유효한 타입은 'string'(기본 값), 'int', 'long', 'choice', 'float', 'complex'이다.

p.disable_interspersed_args()

위치 인수들과 단순 옵션들을 섞는 것을 금지한다. 예를 들어, '-x'와 '-y'가 인수 없는 옵션일 경우 반드시 인수가 나오기 전에 나타나야 한다(예를 들어, 'prog -x -y arg1 arg2 arg3').

p.enable_interspersed_args()

위치 인수들과 단순 옵션들을 섞는 것을 허락한다. 예를 들어, '-x'와 '-y'가 인수 없는 옵션일 때 'prog -x arg1 arg2 -y arg3'처럼 인수와 섞을 수 있다. 기본 방식이다.

p.parse_args([arglist])

명령줄 옵션을 파싱하고 튜플 (options, args)를 반환한다. 여기서 options은 모든 옵션 값을 담은 객체이고 args는 남겨진 모든 위치 인수를 담은 리스트이다. options 객체는 모든 옵션에 대해 옵션 이름에 해당하는 속성에 옵션 데이터를 저장한다. 예를 들어, 옵션 '--output' 값은 options.output에 저장된다. 옵션을 지정하지 않으면 해당 값은 None이 된다. 속성 이름은 add_option()에 dest 키워드 인수를 사용하여 설정할 수도 있다. 기본으로 sys.argv[1:]에서 인수들을 얻는다. 옵션인 인수 arglist로 인수들을 지정할 수도 있다.

p.set_defaults(dest=value, ... dest=value)

옵션 목적지의 기본 값을 설정한다. 설정할 목적지를 키워드 인수로 지정한다. 키워드 인수 이름은 add_option()에 dest 매개변수를 사용하여 지정한 이름과 같아야 한다.

p.set_usage(usage)

-help 옵션으로 생성되는 텍스트에 사용할 사용법 문자열을 변경한다.

예

```
# foo.py
import optparse
p = optparse.OptionParser()

# 인수 없는 단순 옵션
p.add_option("-t", action="store_true", dest="tracing")

# 문자열 인수를 받는 옵션
p.add_option("-o", "--outfile", action="store", type="string",
dest="outfile")

# 정수 인수를 받는 옵션
p.add_option("-d", "--debuglevel", action="store", type="int",
dest="debug")

# 몇 가지 선택을 제공하는 옵션
p.add_option("--speed", action="store", type="choice", dest="speed",
choices=["slow","fast","ludicrous"])

# 인수를 여러 개 가질 수 있는 옵션
p.add_option("--coord", action="store", type="int", dest="coord",
nargs=2)

# 같은 목적지를 갖는 여러 옵션
p.add_option("--novice", action="store_const", const="novice",
dest="mode")
p.add_option("--guru", action="store_const", const="guru", dest="mode")

# 여러 옵션 목적지의 기본 값을 설정
p.set_defaults(tracing=False,
               debug=0,
               speed="fast",
               coord=(0,0),
               mode="novice")

# 인수들을 파싱한다
opt, args = p.parse_args()

# 옵션 값을 출력한다
print "tracing :", opt.tracing
```

```

print "outfile :", opt.outfile
print "debug :", opt.debug
print "speed :", opt.speed
print "coord :", opt.coord
print "mode :", opt.mode

# 남은 인수들을 출력한다
print "args :", args

```

다음은 앞에 나온 코드가 어떻게 작동하는지를 보여주는 짧은 대화식 유닉스 세션을 보여준다.

```

% python foo.py -h
usage: foo.py [options]

options:
  -h, --help show this help message and exit
  -t
  -o OUTFILE, --outfile=OUTFILE
  -d DEBUG, --debuglevel=DEBUG
  --speed=SPEED
  --coord=COORD
  --novice
  --guru

% python foo.py -t -o outfile.dat -d 3 --coord 3 4 --speed=ludicrous
blah
tracing    : True
outfile    : outfile.dat
debug      : 3
speed      : ludicrous
coord      : (3, 4)
mode       : novice
args       : ['blah']

% python foo.py --speed=insane
usage: foo.py [options]

foo.py:error:option --speed:invalid choice:'insane'
(choose from 'slow', 'fast', 'ludicrous')

```

Note

- 옵션 이름을 정할 때 ‘-x’ 같은 단순한 이름에는 대시 하나, ‘—exclude’ 같이 긴 이름에는 대시 두 개를 사용하라. ‘-exclude’ 같이 두 가지 스타일을 섞으면 OptionError 예외가 발생 한다.
- 파이썬에는 C 라이브러리 getopt와 같은 이름을 갖는 명령 줄 파싱을 지원하는 getopt 모듈이 있다. 훨씬 고수준이고 코딩을 덜하게 하는 optparse 대신 이 모듈을 사용할 특별한 이유는 없다.
- optparse 모듈은 커스터마이즈와 특정 명령줄 옵션을 다루기 위한 많은 고급 기능을 제공 한다. 하지만 이런 기능은 일반적으로 사용되는 명령줄 옵션을 파싱하는 데 필요하지 않다. 더 자세한 내용과 추가 예제는 온라인 라이브러리 문서를 참고하기 바란다.

os

os 모듈은 공통 운영체제 서비스에 대한 이식성 있는 인터페이스를 제공한다. nt나 posix 같은 OS 종속적인 내장 모듈을 찾아서 거기에 있는 함수와 데이터를 공개하여 이를 가능하게 한다. 특별한 언급이 없으면 여기 나온 함수들은 윈도와 유닉스 모두에서 사용할 수 있다. 유닉스 시스템은 리눅스와 Mac OS X를 포함한다.

다음과 같은 일반적인 용도에 사용되는 변수들이 정의되어 있다.

environ

현재 환경 변수를 나타내는 매팅 객체. 매팅 객체를 변경하면 현재 환경에 반영된다. `putenv()` 함수를 사용할 수 있으면 변경 사항이 하위 프로세스에도 보인다.

linesep

현재 플랫폼에서 줄을 구분하는 데 사용하는 문자열. POSIX에 대해서는 ‘\n’ 같은 단일 문자이고 윈도에 대해서는 ‘\r\n’ 같은 여러 개 문자일 수 있다.

name

임포트할 OS 종속 모듈 이름: ‘posix’, ‘nt’, ‘dos’, ‘mac’, ‘ce’, ‘java’, ‘os2’, ‘riscos’.

path

경로 연산에 사용할 OS 종속 표준 모듈. 이 모듈은 import os.path를 사용하여 불러올 수도 있다.

프로세스 환경

다음 함수들은 프로세스가 실행되는 환경과 관련된 다양한 매개변수에 접근하거나 이를 조작하는 데 사용한다. 프로세스, 그룹, 프로세스 그룹, 세션 ID는 특별한 언급이 없는 한 정수이다.

chdir(path)

현재 작업 디렉터리를 path로 변경한다.

chroot(path)

현재 프로세스의 루트 디렉터리를 변경한다(유닉스).

ctermid()

프로세스에 대한 제어 터미널의 파일 이름을 담은 문자열을 반환한다(유닉스).

fchdir(fd)

현재 작업 디렉터리를 변경한다. fd는 열린 디렉터리를 나타내는 파일 기술자이다(유닉스).

getcwd()

현재 작업 디렉터리를 담은 문자열을 반환한다.

getcwd()

현재 작업 디렉터리를 담은 유니코드 문자열을 반환한다.

getegid()

유효 그룹(effective group) ID를 반환한다(유닉스).

geteuid()

유효 사용자(effective user) ID를 반환한다(유닉스).

getgid()

프로세스의 실제 그룹 ID를 반환한다(유닉스).

getgroups()

프로세스 소유자가 속한 정수 그룹 ID들을 담은 리스트를 반환한다(유닉스).

getlogin()

유효 사용자 ID와 관련된 사용자 이름을 반환한다(유닉스).

getpgid(pid)

프로세스 ID pid를 가진 프로세스의 프로세스 그룹 ID를 반환한다. pid가 0이면 호출 프로세스의 프로세스 그룹이 반환된다(유닉스).

getpgrp()

현재 프로세스 그룹의 ID를 반환한다. 일반적으로 프로세스 그룹은 보통 작업 제어에 사용된다. 프로세스 그룹은 프로세스의 그룹 ID와 같을 필요가 없다(유닉스).

getpid()

현재 프로세스의 실제 프로세스 ID를 반환한다(유닉스와 윈도).

getppid()

부모 프로세스의 프로세스 ID를 반환한다(유닉스).

getsid(pid)

프로세스 pid의 프로세스 세션 식별자를 반환한다. pid가 0이면 현재 프로세스의 식별자가 반환된다(유닉스).

getuid()

현재 프로세스의 실제 사용자 ID를 반환한다(유닉스).

putenv(varname, value)

환경 변수 varname을 value로 설정한다. 이러한 변경은 os.system(), popen(), fork(), execv()로 생성한 하위 프로세스에 영향을 미친다. os.environ에 항목을 대입하면 자동으로 putenv()가 호출된다. 그렇지만 putenv()를 호출해도 os.environ이 갱신되지는 않는다(유닉스와 윈도).

setegid(egid)

유효 그룹 ID를 설정한다(유닉스).

seteuid(euid)

유효 사용자 ID를 설정한다(유닉스).

setgid(gid)

현재 프로세스의 그룹 ID를 설정한다(유닉스).

setgroups(groups)

현재 프로세스의 그룹 접근 리스트를 설정한다. groups는 그룹 식별자 정수들을 담은 순서열이다. root만 호출할 수 있다(유닉스).

setpgrp()

시스템 호출 setpgrp() 또는 setpgrp(0, 0) 중 구현된 것을 호출하여(있을 경우) 새로운 프로세스 그룹을 생성한다. 새로운 프로세스 그룹의 ID를 반환한다(유닉스).

setpgid(pid, pgrp)

프로세스 그룹 pgrp에 프로세스 pid를 할당한다. pid가 pgrp와 같으면 프로세스는 새로운 프로세스 그룹 리더가 된다. pid가 pgrp와 다르면 프로세스는 기존 그룹에 참여한다. pid가 0이면 호출한 프로세스의 프로세스 ID를 사용한다. pgrp가 0이면 pid로 지정한 프로세스는 프로세스 그룹 리더가 된다(유닉스).

setreuid(ruid,euid)

호출 프로세스의 실제 사용자 ID와 유효 사용자 ID를 설정한다.

setregid(rgid,egid)

호출 프로세스의 실제 그룹 ID와 유효 그룹 ID를 설정한다.

setsid()

새로운 세션을 생성하고 생성된 세션 ID를 반환한다. 일반적으로 세션은 세션 안에서 시작된 프로세스들의 터미널 장치와 작업 제어에 연결된다(유닉스).

setuid(uid)

현재 프로세스의 실제 사용자 ID를 설정한다. 이 함수는 아무나 실행할 수 없고 보통은 root로 실행 중인 프로세스에서만 실행할 수 있다(유닉스).

strerror(code)

정수 에러 코드 code에 해당하는 에러 메시지를 반환한다(유닉스와 윈도). errno 모듈에 에러 코드에 대한 기호 이름이 정의되어 있다.

umask(mask)

현재 숫자 umask를 설정하고 이전 값을 반환한다. umask는 프로세스가 생성한 파일의 권한 비트들을 청소하는 데 사용한다(유닉스와 윈도).

uname()

시스템 타입을 식별하는 문자열들을 담은 튜플 (sysname, nodename, release, version, machine)을 반환한다(유닉스).

unsetenv(name)

환경 변수 name을 지운다.

파일 생성과 파일 기술자

다음 함수들은 파일과 파일프로세스를 조작하는 데 사용하는 저수준 인터페이스를 제공한다. 이들 함수에서 파일은 정수 파일 기술자 fd로 조작한다. 파일 기술자는 파일 객체에 fileno() 메서드를 호출하여 얻을 수 있다.

close(fd)

이전에 open() 또는 pipe()가 반환한 파일 기술자 fd를 닫는다.

closerange(low, high)

low <= fd < high 범위에 있는 모든 파일 기술자 fd를 닫는다. 에러는 무시된다.

dup(fd)

파일 기술자 fd를 복제한다. 사용되지 않은 가장 낮은 숫자 값을 갖는 새로운 파일 기술자를 반환한다. 새로운 파일 기술자와 기존의 파일 기술자는 서로 바꾸어 쓸 수 있다. 게다가 이들은 현재 파일 포인터와 락 같은 상태를 공유한다(유닉스와 윈도).

dup2(olddfd, newfd)

파일 기술자 oldfd를 newfd로 복제한다. newfd가 유효한 파일 기술자이면 먼저 닫는다(유닉스와 윈도).

fchmod(fd, mode)

fd와 연관된 파일 모드를 mode로 변경한다. mode 설명은 os.open()을 참고하도록 한다(유닉스).

fchown(fd, uid, gid)

fd와 연관된 파일 소유자와 그룹 ID를 uid와 gid로 변경한다. uid 또는 gid 값을 변경하지 않으려면 -1 값을 사용한다(유닉스).

fdatasync(fd)

fd에 쓴 캐싱된 모든 데이터를 강제로 디스크에 쓴다(유닉스).

fdopen(fd [, mode [, bufsize]])

파일 기술자 fd에 연결된 열린 파일 객체를 생성한다. mode와 bufsize 인수는 내장 open() 함수에서 의미와 같다. mode는 ‘r’, ‘w’, ‘a’ 같은 문자열이어야 한다. 파일 3에서 이 함수는 내장 open() 함수에 쓰이는 인코딩과 줄 끝 지정 등을 위한 추가 매개변수를 받는다. 파일 2와 이식성을 고려한다면 여기에서 설명한 use와 bufsize만 사용하도록 한다.

fpathconf(fd, name)

파일 기술자 fd에 대한 열린 파일에 연관된 설정 가능한 경로 변수를 반환한다. name은 추출할 값 이름을 담은 문자열이다. 해당 값은 보통 <limits.h>나 <unistd.h>와 같은 시스템 헤더 파일에서 가져온다. POSIX에서는 name으로 다음 상수들을 정의한다.

상수	설명
"PC_ASYNC_IO"	fd에 비동기 I/O를 수행할 수 있는지를 나타낸다.
"PC_CHOWN_RESTRICTED"	chown() 함수를 사용할 수 있는지를 나타낸다. fd가 디렉터리를 가리키면 디렉터리에 있는 모든 파일에 적용된다.
"PC_FILESIZEBITS"	파일의 최대 크기
"PC_LINK_MAX"	파일 링크 횟수의 최댓값
"PC_MAX_CANON"	포맷된 입력 줄의 최대 길이. fd는 터미널을 가리킨다.
"PC_MAX_INPUT"	입력 줄의 최대 길이. fd는 터미널을 가리킨다.
"PC_NAME_MAX"	디렉터리에서 파일 이름의 최대 길이
"PC_NO_TRUNC"	디렉터리에 PC_NAME_MAX보다 긴 이름을 가진 파일을 생성하려는 시도가 ENAMETOOLONG 에러를 발생 시키는지를 나타낸다.
"PC_PATH_MAX"	디렉터리 fd가 현재 작업 디렉터리일 때 상대 경로의 최대 길이
"PC_PIPE_BUF"	fd가 파일 또는 FIFO를 가리킬 때 파일 버퍼의 크기
"PC_PRIO_IO"	우선순위 I/O를 fd에 수행할 수 있는지를 나타낸다
"PC_SYNC_IO"	동기 I/O를 fd에 수행할 수 있는지를 나타낸다.
"PC_VDISABLE"	fd에 특수문자 처리를 비활성화할 수 있는지를 나타낸다. fd는 반드시 터미널을 가리켜야 한다.

모든 이름을 모든 플랫폼에서 사용할 수 있는 것은 아니며 어떤 시스템에는 추가 설정 매개변수가 있을 수도 있다. 운영체제에 알려진 이름은 os.pathconf_names 사

전에 들어 있다. 설정 이름이 os.pathconf_names에 들어 있지 않더라도 해당 정수 값을 name에 지정할 수 있다. 파일에서 이름을 인식하더라도 호스트 운영체제가 인식하지 못하거나 파일 fd에 연결하지 않은 경우에는 OSError 예외가 발생할 수 있다. 이 함수는 몇몇 유닉스 버전에서만 사용할 수 있다.

fstat(fd)

파일 기술자 fd의 상태를 반환한다. os.stat() 함수와 같은 값을 반환한다(유닉스와 윈도).

fstatvfs(fd)

파일 기술자에 연결된 파일을 담은 파일 시스템에 관한 정보를 반환한다. os.statvfs()함수와 같은 값을 반환한다(유닉스와 윈도).

fsync(fd)

fd에 대해 아직 쓰지 않은 데이터를 강제로 디스크에 쓴다. 베퍼 I/O를 수행하는 객체(예를 들어, 파일 file 객체)를 사용한다면 fsync()를 호출하기 전에 먼저 데이터를 비워야(flush) 한다. 유닉스와 윈도에서 사용할 수 있다.

ftruncate(fd, length)

파일 기술자 fd에 해당하는 파일을 크기가 최대 length 바이트가 되도록 자른다(유닉스).

isatty(fd)

fd가 터미널처럼 TTY 같은 장치와 연결되어 있으면 True를 반환한다.

lseek(fd, pos, how)

파일 기술자 fd의 현재 위치를 pos로 설정한다. how의 값은 다음과 같다. SEEK_SET은 파일의 시작 위치에 상대적으로 위치를 설정한다. SEEK_CUR은 현재 위치에 상대적으로 위치를 설정한다. SEEK_END는 파일 끝에 상대적으로 위치를 설정한다. 오래된 파일 코드에서는 이러한 상수들 대신 각각 0, 1, 2의 숫자 값을 사용하는 것을 볼 수 있다.

open(file [, flags [, mode]])

파일 file을 연다. flags는 다음 상수들을 비트 OR한 것이다.

값	설명
O_RDONLY	읽기용으로 파일을 연다.

O_WRONLY	쓰기용으로 파일을 연다.
O_RDWR	읽기 쓰기용으로 연다(갱신).
O_APPEND	바이트들을 파일 끝에 추가한다.
O_CREAT	파일이 존재하지 않으면 생성한다.
O_NONBLOCK	열기, 읽기, 쓰기 때 기다리지(block) 않는다(유닉스).
O_NDELAY	O_NONBLOCK와 같다(유닉스).
O_DSYNC	동기 쓰기(유닉스).
O_NOCTTY	장치를 열 때 제어 터미널을 설정하지 않는다(유닉스).
O_TRUNC	파일이 존재하면 길이가 0이 되도록 자른다.
O_RSYNC	동기 읽기(유닉스).
O_SYNC	동기 쓰기(유닉스).
O_EXCL	O_CREAT이고 파일이 이미 존재하면 에러.
O_EXLOCK	파일에 상호 배타적 락 설정.
O_SHLOCK	파일에 공유 락 설정.
O_ASYNC	입력이 있을 때 SIGIO 신호가 발생하는 비동기 입력 모드 활성화.
O_DIRECT	운영체제 읽기 쓰기 캐시 대신 직접 디스크에 읽거나 쓰는 직접 I/O 모드 사용
O_DIRECTORY	파일이 디렉터리가 아니면 에러를 발생시킴.
O_NOFOLLOW	심벌릭 링크를 따라가지 않음.
O_NOATIME	파일의 가장 최근 접근 시간을 갱신하지 않음.
O_TEXT	텍스트 모드(윈도).
O_BINARY	이진 모드(윈도).
O_NOINHERIT	자식 프로세스에서 파일을 상속하지 않음(윈도).
O_SHORT_LIVED	파일을 단기 저장용으로 사용할 것이라고 시스템에 알림(윈도).
O_TEMPORARY	닫을 때 파일 삭제(윈도).
O_RANDOM	파일을 임의 접근에 사용할 것이라고 시스템에 알림(윈도).
O_SEQUENTIAL	파일을 순차 접근할 것이라고 시스템에 알림(윈도).

동기 I/O 모드(O_SYNC, O_DSYNC, O_RSYNC)는 I/O 연산을 하드웨어 수준에서 완료될 때까지 막는다(예를 들어, 쓰기 작업은 바이트들을 물리적으로 디스크에 다 쓸 때까지 막힌다). mode 매개변수는 다음 8진수 값들(stat 모듈에 상수로 정의됨)을 비트 OR하여 표현하는 파일 권한을 담는다.

모드	의미
0100	사용자가 실행 권한을 가짐(stat.S_IXUSR).
0200	사용자가 쓰기 권한을 가짐(stat.S_IWUSR).
0400	사용자가 읽기 권한을 가짐(stat.S_IRUSR).
0700	사용자가 읽기 쓰기 실행 권한을 가짐(stat.S_IRWXU).
0010	그룹이 실행 권한을 가짐(stat.S_IXGRP).
0020	그룹이 쓰기 권한을 가짐(stat.S_IWGRP).
0040	그룹이 읽기 권한을 가짐(stat.S_IRGRP).
0070	그룹이 읽기 쓰기 실행 권한을 가짐(stat.S_IRWXG).

0001	다른 사용자가 실행 권한을 가짐(stat.S_IXOTH).
0002	다른 사용자가 쓰기 권한을 가짐(stat.S_IWOTH).
0004	다른 사용자가 읽기 권한을 가짐(stat.S_IROTH).
0007	다른 사용자가 읽기 쓰기 실행 권한을 가짐(stat.S_IRWXO).
4000	UID 모드 설정(stat.S_ISUID).
2000	GID 모드 설정(stat.S_ISGID).
1000	스티키 비트(sticky bit) 설정(stat.S_ISVTX).

파일 기본 모드는 (0777 & ~umask)이며 umask 설정은 해당 권한을 제거하는 데 사용된다. 예를 들어, umask 0022는 그룹과 다른 사용자의 쓰기 권한을 제거한다. umask는 os.umask() 함수를 사용하여 변경할 수 있다. umask 설정은 윈도에서는 효력이 없다.

openpty()

유사 터미널(psuedo-terminal)을 열고 PTY와 TTY에 대한 파일 기술자 쌍(master, slave)을 반환한다. 몇몇 유닉스 버전에서만 사용할 수 있다.

pipe()

다른 프로세스와 단방향 통신을 맺는 데 사용할 수 있는 파일을 생성한다. 읽기와 쓰기에 사용할 수 있는 파일 기술자 쌍 (r, w)를 반환한다. 이 함수는 보통 fork() 함수를 실행하기 전에 호출한다. fork() 후 송신 프로세스는 파일 읽기 끝을 닫고 수신 프로세스는 파일 쓰기 끝을 닫는다. 이 시점에서 파일이 활성화되고 read()와 write() 함수를 통해 한 프로세스에서 다른 프로세스로 데이터를 보낼 수 있게 된다(유닉스).

read(fd, n)

파일 기술자 fd에서 최대 n 바이트 읽는다. 읽은 바이트들을 담은 바이트 문자열을 반환한다.

tcgetpgrp(fd)

fd로 주어진 제어 터미널과 연결된 프로세스 그룹을 반환한다(유닉스).

tcsetpgrp(fd, pg)

fd로 주어진 제어 터미널과 연결된 프로세스 그룹을 설정한다(유닉스).

ttyname(fd)

파일 기술자 fd와 연결된 터미널 장치를 나타내는 문자열을 반환한다. fd가 터미

널 장치와 연결되어 있지 않으면 OSError 예외가 발생한다(유닉스).

write(fd, str)

파일 기술자 fd에 바이트 문자열 str을 쓴다. 실제로 쓰여진 바이트 수를 반환한다.

파일과 디렉터리

다음 함수와 변수들은 파일 시스템에서 파일과 디렉터리를 조작하는 데 사용한다. 다음은 파일 이름을 정하는 방식의 차이를 다루기 위해 경로 생성에 관련한 정보를 담은 변수들을 보여준다.

변수 설명

altsep	OS에서 경로 구성을 구분하는 데 사용하는 기타 문자. 구분 문자가 하나만 있다면 None. sep가 역슬래시인 DOS나 윈도 시스템에서는 ‘/’로 설정된다.
curdir	현재 작업 디렉터리를 가리키는 데 사용하는 문자열. 유닉스와 윈도에서는 ‘.’, Macintosh에서는 ‘:’.
devnull	널(null) 장치 경로(예를 들어, /dev/null)
extsep	기반 파일 이름을 터미널과 구분하는 문자(예를 들어, ‘foo.txt’에서 ‘.’)
pardir	부모 디렉터리를 가리키는 데 사용하는 문자열. 유닉스와 윈도에서 ‘..’, Macintosh에서는 ‘::’.
pathsep	검색 경로(\$PATH 환경 변수에 담긴)의 구성을 구분하는 데 사용하는 문자. 유닉스에서는 ‘:’, DOS와 윈도에서는 ‘;’.
sep	경로 구성을 구분하는 데 사용하는 문자. 유닉스와 윈도에서는 ‘/’, Macintosh에서는 ‘:’.

다음 함수들은 파일을 조작하는 데 사용한다.

access(path, accessmode)

이 프로세스의 주어진 파일 path에 대한 읽기 쓰기 실행 권한을 검사한다. accessmode에는 읽기, 쓰기, 실행, 존재를 가리키는 데 각각 R_OK, W_OK, X_OK, F_OK가 사용된다. 접근이 허락되면 1을, 그렇지 않으면 0을 반환한다.

chflags(path, flags)

path의 파일 플래그를 변경한다. flags는 다음 목록에 나오는 상수들을 비트 OR 한 것이다. UF_로 시작하는 플래그는 아무 사용자나 설정할 수 있는 반면 SF_ 플래그들은 슈퍼유저(superuser)만 변경할 수 있다.

플래그	의미
<code>stat.UF_NODUMP</code>	파일을 덤프하지 않는다.
<code>stat.UF_IMMUTABLE</code>	파일이 읽기 전용이다.
<code>stat.UF_APPEND</code>	파일이 추가 연산만 지원한다.
<code>stat.UF_OPAQUE</code>	디렉터리가 불투명하다.
<code>stat.UF_NOUNLINK</code>	파일을 지우거나 이름을 변경할 수 없다.
<code>stat.SF_ARCHIVED</code>	파일을 보관할 수 있다.
<code>stat.SF_IMMUTABLE</code>	파일이 읽기 전용이다.
<code>stat.SF_APPEND</code>	파일이 추가 연산만 지원한다.
<code>stat.SF_NOUNLINK</code>	파일을 지우거나 이름을 변경할 수 없다.
<code>stat.SF_SNAPSHOT</code>	파일이 스냅샷 파일이다.

chmod(path, mode)

path의 모드를 변경한다. mode는 open() 함수에서 살펴본 것과 같은 값을 가진다(유닉스와 윈도).

chown(path, uid, gid)

path의 소유자와 그룹 ID를 숫자 uid와 gid로 변경한다. uid 또는 gid의 값을 -1로 설정하면 값이 변하지 않는다(유닉스).

lchflags(path, flags)

chflags()와 같지만 심벌릭 링크를 따라가지 않는다(유닉스).

lchmod(path, mode)

path가 심벌릭 링크이면 링크가 가리키는 파일이 아닌 링크 자체를 조작하는 것을 제외하고 chmod()와 같다.

lchown(path, uid, gid)

chown()과 같지만 심벌릭 링크를 따라가지 않는다(유닉스).

link(src, dst)

src를 가리키는 dst라는 이름을 가진 하드 링크를 생성한다(유닉스).

listdir(path)

디렉터리 path에 있는 개체들의 이름을 담은 리스트를 반환한다. 이 리스트는 순서가 뒤바뀌어 있으며 '.'과 '..' 같은 특수 개체는 담지 않는다. path가 유니코드이면 결과 리스트는 유니코드 문자열만 담는다. 디렉터리에 있는 파일을 유니코드로 적절히 인코딩할 수 없으면 조용히 건너뛴다. path를 바이트 문자열로 주면 모든 파일

이름을 바이트 문자열 리스트로 반환한다.

lstat(path)

stat()와 비슷하지만 심벌릭 링크를 따라가지 않는다(유닉스).

makedev(major, minor)

주(major)와 부(minor) 장치 번호가 주어질 때 무가공 장치 번호를 생성한다.

major(devicenum)

makedev()로 생성한 무가공 장치 번호 devicenum에서 주 장치 번호를 반환한다.

minor(devicenum)

makedev()로 생성한 무가공 장치 번호 devicenum에서 부 장치 번호를 반환한다.

makedirs(path [, mode])

재귀 디렉터리 생성 함수. mkdir()과 비슷하지만 말단 디렉터리를 담는 데 필요한 모든 중간 디렉터리를 생성한다. 말단 디렉터리가 이미 존재하거나 생성할 수 없으면 OSError 예외가 발생한다.

mkdir(path [, mode])

숫자 모드 mode로 이름이 path인 디렉터리를 생성한다. 기본 모드는 0777이다. 유닉스 시스템이 아닌 곳에서는 모드 설정이 아무 효과가 없거나 무시된다.

mkfifo(path [, mode])

숫자 모드 mode로 이름이 path인 FIFO(fifo)를 생성한다. 기본 모드는 0666이다(유닉스).

mknod(path [, mode, device])

장치 특수 파일을 생성한다. path는 파일 이름이고 mode는 파일 권한과 타입을 지정하며 device는 os.makedev()로 생성한 무가공 장치 번호이다. 파일 접근 권한을 설정할 때 open()의 매개변수를 mode 매개변수에 사용할 수 있다. 추가로 파일 타입을 지정하기 위해 stat.S_IFREG, stat.S_IFCHR, stat.S_IFBLK, stat.S_IFIFO 플래그를 mode에 추가할 수 있다(유닉스).

pathconf(path, name)

경로 path와 연관된 설정 가능한 시스템 매개변수들을 반환한다. name은 매개변수 이름을 담은 문자열이고 fpathconf() 함수에서 언급한 것과 같다(유닉스).

readlink(path)

심벌릭 링크 path가 가리키는 경로를 나타내는 문자열을 반환한다(유닉스).

remove(path)

파일 path를 삭제한다. unlink() 함수와 동일하다.

removedirs(path)

재귀 디렉터리 삭제 함수. 말단 디렉터리가 성공적으로 삭제되면 남은 경로에서 가장 오른쪽 구성 요소에 해당하는 디렉터리를 반복적으로 지워나간다. 전체 경로를 다 지웠거나 에러가 발생(이 말은 부모 디렉터리가 비어 있지않다는 것을 의미하기 때문에 보통 조용히 무시된다)할 때까지 수행하고 이것 말고는 rmdir()과 같다. 말단 디렉터리를 성공적으로 제거하지 못했으면 OSError 예외가 발생한다.

rename(src, dst)

파일 또는 디렉터리 src의 이름을 dst로 변경한다.

renames(old, new)

디렉터리 또는 파일 이름 재귀 변경 함수. 새로운 경로를 구축하는 데 필요한 중간 디렉터리를 생성하는 점을 제외하고 rename()과 같다. 이름 변경 후 남은 원래 이름에서 removedirs()로 가장 오른쪽 경로 구성 요소에 해당하는 디렉터리를 반복적으로 제거한다.

rmdir(path)

디렉터리 path를 삭제한다.

stat(path)

주어진 path에 stat() 시스템 호출을 수행하여 파일 정보를 추출한다. 파일 정보를 속성에 담은 객체가 반환된다. 다음은 흔히 사용되는 속성이다.

속성	설명
st_mode	아이노드(inode) 보호 모드
st_ino	아이노드 개수
st_dev	아이노드가 있는 장치
st_nlink	아이노드로 링크 개수
st_uid	소유자의 사용자 ID
st_gid	소유자의 그룹 ID
st_size	바이트 단위 파일 크기
st_atime	마지막 접근 시간

<code>st_mtime</code>	마지막 수정 시간
<code>st_ctime</code>	마지막 상태 변경 시간

시스템에 따라 속성이 더 있을 수 있다. `stat()`에서 반환되는 객체는 10개 항목짜리 튜플 (`st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`)처럼 취급할 수도 있다. 이 형식은 하위 호환성을 위해 제공된다. `stat` 모듈에는 이 튜플에서 필드를 추출하는 데 사용할 수 있는 상수들이 정의되어 있다.

`stat_float_times([newvalue])`

`stat()`에서 반환하는 시간이 정수가 아니라 부동 소수점 수일 경우 `True`를 반환한다. `newvalue` 불리언 값을 설정하면 이 작동 방식을 변경할 수 있다.

`statvfs(path)`

주어진 `path`에 `statvfs()` 시스템 호출을 수행하여 파일 시스템 정보를 얻는다. 파일 시스템에 대한 설명을 속성에 담은 객체가 반환된다. 다음 속성이 흔히 사용된다.

속성	설명
<code>f_bsize</code>	선호 시스템 블록 크기
<code>f_frsize</code>	기본 파일 시스템 블록 크기
<code>f_blocks</code>	파일 시스템에 있는 전체 블록 개수
<code>f_bfree</code>	자유 블록 개수
<code>f_bavail</code>	비슈퍼유저가 사용할 수 있는 자유 블록들
<code>f_files</code>	파일 아이노드 전체 개수
<code>f_ffree</code>	자유 파일 아이노드 전체 개수
<code>f_favail</code>	비슈퍼유저가 사용할 수 있는 자유 노드들
<code>f_flag</code>	플래그들(시스템에 따라 다름)
<code>f_namemax</code>	파일 이름의 최대 길이

반환된 객체는 여기 나온 순서대로 이 속성들을 담은 튜플처럼 취급할 수도 있다. 모듈 `statvfs`에는 반환된 `statvfs` 데이터에서 정보를 추출하는 데 사용할 수 있는 상수들이 정의되어 있다(유닉스).

`symlink(src, dst)`

이름이 `dst`인 `src`를 가리키는 심벌릭 링크를 생성한다.

`unlink(path)`

파일 `path`를 삭제한다. `remove()`과 같다.

utime(path, (atime, mtime))

파일 접근 시간과 수정 시간을 주어진 값으로 설정한다.(두 번째 인수는 항목 두 개짜리 튜플이다.) 시간은 time.time() 함수에서 반환하는 숫자 형식으로 지정한다.

walk(top [, topdown [, onerror [, followlinks]]])

디렉터리 트리를 순회하는 생성기 객체를 반환한다. top는 최상위 디렉터리를 지정하고 topdown은 디렉터리를 위에서 아래로(기본 값) 순회할 것인지 아래에서 위로 순회할 것인지를 지정하는 불리언 값이다. 반환되는 생성기는 튜플 (dirpath, dirnames, filenames)를 생성하며 여기서 dirpath는 디렉터리 경로를 담은 문자열이고 dirnames는 dirpath에 있는 모든 하위 디렉터리를 담은 리스트이며 filenames는 dirpath에 있는 모든 파일(디렉터리를 포함하지 않음)을 담은 리스트이다. onerror 매개변수는 인수 하나를 입력으로 받는 함수이다. 순회하는 동안 에러가 발생하면 os.error의 인스턴스로 이 함수가 호출된다. 기본으로 중간에 발생한 에러를 무시한다. 디렉터리를 위에서 아래로 순회할 경우 dirnames를 수정하면 순회 과정이 달라질 수 있다. 예를 들어, dirnames에서 디렉터리를 지우면 해당 디렉터리를 건너뛴다. followlinks 인수가 True로 설정되지 않는 한 기본으로 심벌릭 링크를 따라가지 않는다.

프로세스 관리

다음 함수와 변수들은 프로세스를 생성, 파괴, 관리하는 데 사용한다.

abort()

현재 프로세스에 SIGABRT 시그널을 생성한다. 이 시그널을 시그널 처리기로 처리하지 않으면 기본으로 프로세스가 에러와 함께 종료한다.

defpath

환경변수 ‘PATH’가 없는 경우 exec*p*() 함수에서 사용할 기본 검색 경로를 담은 변수.

execl(path, arg0, arg1, ...)

execv(path, (arg0, arg1, ...))와 같다.

execle(path, arg0, arg1, ..., env)

execve(path, (arg0, arg1, ...), env)와 같다.

execlp(path, arg0, arg1, ...)

execvp(path, (arg0, arg1, ...))와 같다.

execv(path, args)

현재 프로세스(즉, 파이썬 인터프리터)를 대체하면서 인수 리스트 args로 프로그램 path를 실행한다. 인수 리스트는 튜플이나 문자열 리스트로 지정할 수 있다.

execve(path, args, env)

execv()처럼 새로운 프로그램을 실행하지만 프로그램 실행 때 사용할 환경변수들을 정의한 사전 env를 추가로 입력받는다. env는 반드시 문자열을 문자열로 매핑하는 사전이어야 한다.

execvp(path, args)

eexecv(path, args)와 같지만 실행 파일을 디렉터리 목록에서 검색하는 셀의 동작을 따라한다. 디렉터리 목록은 environ['PATH']에서 얻는다.

execvpe(path, args, env)

eexecvp()와 같지만 execve() 함수처럼 추가 환경 변수를 받는다.

_exit(n)

정리 작업 없이 상태 n으로 종료한다. 일반적으로 fork()로 생성한 자식 프로세스에서 사용한다. 적절하게 인터프리터를 종료하는 sys.exit() 호출과 다르다. 종료 코드 n은 응용 프로그램에 따라 다를 수 있지만 보통 0은 성공을 나타내고 0이 아닌 값은 특정 종류의 에러를 나타낸다. 시스템마다 몇몇 표준 종료 코드 값을 정의한다.

값	설명
EX_OK	에러 없음
EX_USAGE	틀린 명령어 사용
EX_DATAERR	틀린 입력 데이터
EX_NOINPUT	입력 없음
EX_NOUSER	사용자 없음
EX_NOHOST	호스트 없음
EX_NOTFOUND	발견되지 않음
EX_UNAVAILABLE	서비스 사용 불가
EX_SOFTWARE	내부 소프트웨어 에러
EX_OSERR	운영체제 에러
EX_OSFILE	파일 시스템 에러

<code>EX_CANTCREATE</code>	출력 생성 불가능
<code>EX_IOERR</code>	I/O 에러
<code>EX_TEMPFAIL</code>	임시 실패
<code>EX_PROTOCOL</code>	프로토콜 에러
<code>EX_NOPERM</code>	불충분한 권한
<code>EX_CONFIG</code>	설정 에러

fork()

자식 프로세스를 생성한다. 새로 생성된 자식 프로세스는 0을 반환하고 기존 프로세스는 자식 프로세스 ID를 반환한다. 자식 프로세스는 원래 프로세스의 복제본으로서 열린 파일 같은 다양한 자원을 공유한다(유닉스).

forkpty()

새로운 유사 터미널을 자식 제어 터미널로 사용하여 자식 프로세스를 생성한다. (pid, fd)의 쌍을 반환하며 자식 프로세스에서 pid는 0이고 fd는 유사 터미널의 마스터(master) 끝을 나타내는 파일 기술자이다. 이 함수는 특정 유닉스 버전에서만 사용할 수 있다.

kill(pid, sig)

프로세스 pid에 시그널 sig를 보낸다. 시그널 이름 목록은 signal 모듈에 나와 있다(유닉스).

killpg(pgid, sig)

프로세스 그룹 pgid에 시그널 sig를 보낸다. 시그널 이름 목록은 signal 모듈에 나와 있다(유닉스).

nice(increment)

프로세스의 스케줄링 우선순위(“좋음(niceness)”)를 증가시킨다. 새로운 좋음 값을 반환한다. 보통 프로세스의 우선순위를 높이는 것은 루트 권한을 필요로 하기 때문에 사용자는 프로세스의 우선순위를 낮출 수만 있다. 우선순위 변경의 효과는 시스템마다 다르며 흔히 다른 프로세스의 성능에 눈에 띄는 영향을 끼치지 않으면서 프로세스를 백그라운드에서 실행하려고 할 때 우선순위를 낮춘다(유닉스).

plock(op)

프로그램 세그먼트를 메모리에 잠궈 스왑되지 않게 한다. op의 값은 어느 세그먼트를 잠글지를 가리키는 정수이다. op의 값은 플랫폼마다 다르지만 일반적으로

UNLOCK, PROCLOCK, TXTLOCK, DATLOCK 중 하나이다. 이 상수들은 파일에 정의하는 것이 아니고 <sys/lock.h> 헤더 파일에 정의된다. 이 함수는 모든 플랫폼에서 사용할 수 있는 것은 아니며 유효 사용자 ID가 0인(root) 프로세스에서만 사용할 수 있다(유닉스)

open(command [, mode [, bufsize]])

command로 또는 command로부터 파일을 연다. 반환 값은 파일에 연결된 열린 파일 객체이며, mode가 ‘r’(기본 값) 또는 ‘w’인지에 따라 읽거나 쓸 수 있다. bufsize는 내장 open() 함수에서 의미와 같다. 반환된 파일의 close() 메서드를 호출하면 명령의 종료 상태가 반환된다. 단, 종료 상태가 0이면 close()는 None을 반환한다.

spawnv(mode, path, args)

args로 지정한 인수들을 명령줄 매개변수로 전달하면서 프로그램 path를 새로운 프로세스로 실행한다. args는 리스트 또는 튜플이다. args의 첫 번째 항목은 프로그램 이름이어야 한다. mode는 다음 상수 중 하나이다.

상수	설명
P_WAIT	프로그램을 실행하고 종료될 때까지 기다린다. 프로그램의 종료 코드를 반환한다.
P_NOWAIT	프로그램을 실행하고 프로세스 핸들을 반환한다.
P_NOWAITO	P_NOWAIT와 같다.
P_OVERLAY	프로그램을 실행하고 호출 프로세스를 파괴한다(exec 함수와 같다).
P_DETACH	프로그램을 실행하고 떼어낸다. 호출 프로그램은 계속 실행되지만 생성된 프로세스를 기다릴 수 없게 된다.

spawnv()은 윈도와 유닉스의 몇몇 버전에서만 사용할 수 있다.

spawnve(mode, path, args, env)

args로 지정한 인수들을 명령줄 매개변수로 전달하고 매팅 env의 내용을 환경 변수로 전달하면서 프로그램 path를 새로운 프로세스로 실행한다. args는 리스트 또는 튜플이다. mode는 spawnv()에서 설명한 것과 같은 의미를 가진다.

spawnl(mode, path, arg1, ..., argn)

추가 매개변수 형태로 인수를 지정하는 것 말고는 spawnv()와 같다.

spawnle(mode, path, arg1, ... , argn, env)

인수를 매개변수로 지정하는 것 말고는 spawnve()와 같다. 마지막 매개변수는 환경 변수들을 담은 매핑이다.

spawnlp(mode, file, arg1, ... , argn)

spawnl()와 같지만 환경 변수 PATH 설정을 사용하여 file을 찾는다(유닉스).

spawnlpe(mode, file, arg1, ... , argn, env)

spawnle()와 같지만 환경 변수 PATH 설정을 사용하여 file을 찾는다(유닉스).

spawnvp(mode, file, args)

spawnnv()와 같지만 환경 변수 PATH 설정을 사용하여 file을 찾는다(유닉스).

spawnvpe(mode, file, args, env)

spawnvne()와 같지만 환경 변수 PATH 설정을 사용하여 file을 찾는다(유닉스).

startfile(path [, operation])

파일 path에 연관된 응용 프로그램을 실행한다. 윈도 탐색기에서 파일을 더블 클릭한 것과 같다. 응용 프로그램이 시작되자마자 함수가 반환된다. 또한 종료 코드를 얻거나 응용 프로그램이 끝날 때까지 기다릴 방법이 없다. path는 현재 디렉터리에 상대적으로 지정한다. 옵션인 operation은 path를 열 때 수행할 동작을 나타내는 문자열이다. 기본 값은 ‘open’이지만 ‘print’, ‘edit’, ‘explore’, ‘find’로 설정할 수도 있다(정확한 목록은 path 타입에 따라 다르다(윈도)).

system(command)

하위 셸에서 command(문자열)를 실행한다. 유닉스에서 반환 값은 wait()에서 반환하는 프로세스 종료 상태이다. 윈도에서 종료 코드는 항상 0이다. subprocess 모듈이 훨씬 강력한 기능을 제공하기 때문에 하위 프로세스를 생성할 때는 subprocess 모듈을 사용하도록 한다.

times()

누적 시간(accumulated time)을 초로 나타낸 부동 소수점 수 다섯 개를 담은 튜플을 반환한다. 유닉스에서 이 튜플은 사용자 시간, 시스템 시간, 자식 사용자 시간, 자식 시스템 시간, 실제 경과 시간을 순서대로 담는다. 윈도에서 이 튜플은 사용자 시간, 시스템 시간을 담고 나머지 세 항목에 대해서는 0을 담는다.

wait([pid])

자식 프로세스가 끝나기를 기다렸다가 자식 프로세스 ID와 종료 상태를 담은 튜플을 반환한다. 종료 상태는 16비트 숫자로 숫자의 아래쪽(low) 바이트는 프로세스를 죽인 시그널 번호이고 숫자의 위쪽(high) 바이트는 종료 상태이다(시그널 번호가 0이면). 코어 파일이 생성된 경우에는 아래쪽 바이트의 위쪽 비트가 설정된다. pid가 주어지면 기다릴 프로세스를 나타낸다. pid를 생략하면 wait()는 아무 자식 프로세스나 종료되면 반환된다(유닉스).

waitpid(pid, options)

주어진 프로세스 ID pid에 해당하는 자식 프로세스 상태가 변할 때까지 기다리고 wait()에서처럼 자식 프로세스 ID와 종료 상태를 담은 튜플을 반환한다. options은 정상 연산에는 0을 사용하고 자식 프로세스 상태를 즉시 알 수 없을 때 기다리는 것을 막으려면 WNOHANG을 사용한다. 이 함수는 특별한 이유로 실행이 중단된 자식 프로세스의 정보를 모으는 데 사용하기도 한다. options을 WCONTINUED로 설정하면 작업 제어로 중단되었다가 연산을 다시 시작할 때 자식에 관한 정보를 얻을 수 있다. options을 WUNTRACED로 설정하면 중단은 되었지만 아직 상태 정보가 보고되지 않은 자식에 관한 정보를 얻을 수 있다.

wait3([options])

아무 자식 프로세스에나 일어난 변화를 기다리는 것 말고는 waitpid()와 같다. 3개 항목 튜플 (pid, status, rusage)을 반환한다. 여기서 pid는 자식 프로세스 ID, status는 종료 상태 코드, rusage는 resource.getrusage()에서 반환하는 자원 사용 정보를 나타낸다. options 매개변수는 waitpid()에서 의미와 같다.

wait4(pid, options)

반환 결과가 wait3()에서 반환하는 것과 같은 튜플이라는 점을 제외하고는 waitpid()와 같다.

다음 함수들은 waitpid(), wait3(), wait4()에서 반환하는 프로세스 상태 코드를 입력받아 프로세스 상태를 검사하는 데 사용한다(유닉스).

WCOREDUMP(status)

프로세스가 코어를 덤프했으면 True를 반환한다.

WIFEXITED(status)

프로세스가 exit() 시스템 호출로 종료되었으면 True를 반환한다.

WEXITSTATUS(status)

WIFEXITED(status)가 참이면 exit() 시스템 호출에 전달된 매개변수를 반환한다.
아니면 반환 값은 아무런 의미가 없다.

WIFCONTINUED(status)

프로세스가 작업 제어 멈춤에서 복귀하였으면 True를 반환한다.

WIFSIGNALED(status)

프로세스가 시그널로 종료되었다면 True를 반환한다.

WIFSTOPPED(status)

프로세스가 멈추었으면 True를 반환한다.

WSTOPSIG(status)

프로세스를 멈춘 시그널을 반환한다.

WTERMSIG(status)

프로세스를 종료시킨 시그널을 반환한다.

시스템 설정

다음 함수들은 시스템 설정 정보를 얻는 데 사용한다.

confstr(name)

시스템 설정 변수 값을 문자열로 반환한다. name은 변수 이름을 지정하는 문자열이다. 받아들이는 이름은 플랫폼에 따라 다르며 os.confstr_names에 호스트 시스템에서 알려진 이름들이 담긴 사전이 저장된다. 해당 이름에 설정 값이 없으면 빈 문자열을 반환한다. name을 알 수 없으면 ValueError가 발생한다. 호스트 시스템에서 해당 설정 이름을 지원하지 않으면 OSError가 발생할 수도 있다. 이 함수에서 반환하는 값은 대부분 호스트 머신의 빌드 환경과 관련 있으며 시스템 유ти리티 경로, 프로그램 설정을 위한 다양한 컴파일 옵션(예를 들어 32비트, 64비트, 큰 파일 지원), 링커 옵션 등이 포함된다(유닉스).

getloadavg()

최근 1분, 5분, 15분간 시스템 실행 큐에 들어 있던 평균 항목 수를 담은 3개 항목 튜플을 반환한다.

sysconf(name)

정수 값을 담은 시스템 설정 변수를 반환한다. name은 변수 이름을 지정하는 문자열이다. 호스트 시스템에서 정의된 이름들은 os.sysconf_names 사전에서 살펴볼 수 있다. 설정 이름이 알려져 있지만 그 값이 정의되어 있지 않으면 -1을 반환한다. 그렇지 않으면, ValueError 또는 OSError가 발생한다. 몇몇 시스템에서는 100개 이상의 시스템 매개변수를 정의하고 있다. 다음 목록은 대부분 유닉스 시스템에서 사용할 수 있는 POSIX.1에 정의된 매개변수를 보여준다.

매개변수	설명
"SC_ARG_MAX"	exec()에 사용할 수 있는 최대 인수 개수
"SC_CHILD_MAX"	사용자 ID당 최대 프로세스 개수
"SC_CLK_TCK"	초당 클럭 톱(clock tick) 수
"SC_NGROUPS_MAX"	동시 추가 그룹(supplementary) ID의 최대 개수
"SC_STREAM_MAX"	프로세스에서 한 번에 열 수 있는 스트림의 최대 개수
"SC_TZNAME_MAX"	시간대 이름의 최대 바이트 수
"SC_OPEN_MAX"	프로세스에서 한 번에 열 수 있는 파일의 최대 개수
"SC_JOB_CONTROL"	시스템이 작업 제어를 지원한다.
"SC_SAVED_IDS"	저장된 set-user-ID와 set-group-ID가 각 프로세스에 있는지를 나타낸다.

urandom(n)

시스템에서 생성한 임의의 n개 바이트를 담은 문자열을 반환한다(예를 들어, 유닉스에서 /dev/urandom). 반환되는 바이트는 암호 기술에 사용하기 적합하다.

예외

os 모듈은 에러를 나타내는 예외 하나를 정의한다.

error

함수가 시스템 관련 에러를 반환하려고 발생시키는 예외. 내장 예외 OSError와 같다. 예외는 errno와 strerror 두 값을 가진다. errno는 errno 모듈을 설명할 때 나온 정수 에러 값을 담는다. strerror는 문자열 에러 메시지이다. 파일 시스템과 관련된 예외일 때에는 함수로 전달된 파일 이름이 세 번째 속성 filename에 담긴다.

os.path

os.path 모듈은 이식성 있게 경로를 조작하는 데 사용한다. os 모듈에 의해서 임포트된다.

abspath(path)

현재 작업 디렉터리를 고려하여 경로 path의 절대 경로를 반환한다. 예를 들어, `abspath('.. / Python / foo')`는 '`/home/beazley/Python/foo`'를 반환한다.

basename(path)

경로 path의 기본 이름을 반환한다. 예를 들어, `basename('/usr/local/python')`은 '`python`'을 반환한다.

commonprefix(list)

list에 있는 모든 문자열에서 가장 긴 공통적인 접두사 문자열을 반환한다. list가 비어 있으면 빈 문자열을 반환한다.

dirname(path)

경로 path의 디렉터리 이름을 반환한다. 예를 들어, `dirname('/usr/local/python')`은 '`/usr/local`'을 반환한다.

exists(path)

path가 존재하는 경로를 가리키면 `True`를 반환한다. path가 깨진 심벌릭 링크를 가리키면 `False`를 반환한다.

expanduser(path)

'`~user`' 형태의 경로를 사용자 홈 디렉터리로 대체한다. 확장에 실패하거나 path가 '~'로 시작하지 않으면 path는 수정되지 않은 채 반환된다.

expandvars(path)

path에 있는 '`$name`' 또는 ' `${name}`' 형태의 환경 변수를 확장한다. 올바르지 않거나 존재하지 않는 변수 이름은 그대로 둔다.

getatime(path)

최근 접근 시간을 에포크(epoch) 아래 초로 반환한다(`time` 모듈 참고). `os.stat_float_times()`가 `True`이면 부동 소수점 수를 반환한다.

getctime(path)

유닉스에서는 최근 수정 시간을, 윈도에서는 생성 시간을 반환한다. 시간은 에포크 아래 초로 반환한다(time 모듈 참고). 특정 경우에 부동 소수점 수가 반환된다(getatime() 참고).

getmtime(path)

최근 수정 시간을 에포크 아래 초로 반환한다(time 모듈 참고). 특정 경우에 부동 소수점 수를 반환한다(getatime() 참고).

getsize(path)

파일 크기를 바이트로 반환한다.

isabs(path)

path가 절대 경로(슬래시로 시작)이면 True를 반환한다.

.isfile(path)

path가 일반(regular) 파일이면 True를 반환한다. 이 함수는 심벌릭 링크를 따라간다. 따라서 같은 경로에 대해 islink()와 isfile() 둘 다 참일 수 있다.

.isdir(path)

path가 디렉터리이면 True를 반환한다. 심벌릭 링크를 따라간다.

islink(path)

path가 심벌릭 링크를 가리키면 True를 반환한다. 심벌릭 링크를 지원하지 않으면 False를 반환한다.

ismount(path)

path가 마운트 지점이면 True를 반환한다.

join(path1 [, path2 [, ...]])

하나 이상의 경로 요소를 지능적으로 결합한다. 예를 들어, join('home', 'beazley', 'Python')는 'home/beazley/Python'을 반환한다.

lexists(path)

path가 존재하면 True를 반환한다. 심벌릭 링크에 대해서는 링크가 깨져 있을지라도 True를 반환한다.

normcase(path)

경로에서 대소문자를 정규화한다. 대소문자를 구별하지 않는 파일 시스템에서는 path를 소문자로 변환한다. 윈도에서는 슬래시를 역슬래시로 변환하는 작업도 한다.

normpath(path)

경로를 정규화한다. 중복 구분자와 상위 수준 참조를 하나로 정리한다. ‘A//B’, ‘A./B’, ‘A/foo../B’는 모두 ‘A/B’로 만든다. 윈도에서는 슬래시를 역슬래시로 변환하는 작업도 한다.

realpath(path)

심벌릭 링크(있는 경우)를 제거한 path의 실제 경로를 반환한다(유닉스).

relpath(path [, start])

현재 작업 디렉터리에서 path까지 상대 경로를 반환한다. start는 다른 시작 디렉터리를 지정하는 데 사용한다.

samefile(path1, path2)

path1과 path2가 같은 파일이나 디렉터리를 가리킬 경우 True를 반환한다(유닉스).

sameopenfile(fp1, fp2)

열린 파일 객체 fp1과 fp2가 같은 파일을 가리키고 있을 경우 True를 반환한다(유닉스).

samestat(stat1, stat2)

fstat(), lstat(), stat()에서 반환되는 상태 튜플 stat1과 stat2가 같은 파일을 가리킬 경우 True를 반환한다(유닉스).

split(path)

path를 (head, tail) 쌍으로 나눈다. 여기서 tail은 경로의 마지막 요소이고 head는 tail을 제외한 앞부분을 나타낸다. 예를 들어, ‘/home/user/foo’는 ('/home/user', 'foo')로 나눠진다. 이 튜플은 (dirname(), basename())과 같다.

splitdrive(path)

path를 (drive, filename) 쌍으로 나눈다. drive는 드라이브 명세 또는 빈 문자열이다. drive는 드라이브 명세 없는 머신에서 항상 빈 문자열이다.

splitext(path)

경로를 기본 파일 이름과 접미사로 나눈다. 예를 들어, splitext('foo.txt')은 ('foo', '.txt')를 반환한다.

splitunc(path)

경로 이름을 (unc, rest) 쌍으로 나눈다. 여기서 unc는 UNC(Universal Naming Convention) 마운트 지점이고 rest는 경로의 남은 부분이다(윈도).

supports_unicode_filenames

파일 시스템이 유니코드 파일 이름을 지원하면 True로 설정된다.

Note

윈도에서 드라이브 문자를 포함한 파일 이름(예를 들어, 'C:spam.txt')을 다룰 때 몇 가지 주의 할 점이 있다. 대부분의 경우, 이러한 파일 이름을 현재 작업 디렉터리에 상대적인 것으로 해석 한다. 예를 들어, 현재 디렉터리가 'C:\Foo\'이면 파일 'C:spam.txt'는 'C:\spam.txt'가 아닌 'C:\Foo\spam.txt'로 해석된다.

참고

`fnmatch(389페이지), glob(390페이지), os(466페이지)`

signal

signal 모듈은 파이썬에서 시그널 처리기를 작성하는 데 사용한다. 시그널은 보통 타이머 만료, 데이터 도착 또는 몇몇 사용자 동작으로 프로그램에 전달되는 비동기 이벤트이다. 시그널 인터페이스는 유닉스의 것을 흉내 내지만 다른 플랫폼에서도 일부 시그널을 지원한다.

alarm(time)

`time=0` 아니면 `time` 초 후에 프로그램에 SIGALRM 시그널을 보내도록 스케 줄링한다. 이전에 스케줄링된 알람은 취소된다. `time=0`이면 아무 알람도 스케줄링 되지 않으며 기존에 설정된 알람도 취소된다. 이전에 스케줄링된 알람이 있다면 남 은 초를 반환하고 없다면 0을 반환한다(유닉스).

getsignal(signalnum)

시그널 signalnum에 대한 시그널 처리기를 반환한다. 반환된 객체는 호출 가능한 파이썬 객체이다. 이 함수는 무시된 시그널에 대해서는 SIG_IGN을 반환하고 기본 시그널 처리기에 대해서는 SIG_DFL을 반환하며 파이썬 인터프리터에서 시그널 처리기를 설치하지 않은 경우에는 None을 반환한다.

getitimer(which)

which가 가리키는 내부 타이머의 현재 값을 반환한다.

pause()

다음 시그널을 수신할 때까지 잠든다(유닉스).

set_wakeup_fd(fd)

시그널을 수신하면 '\0' 바이트를 쓸 파일 기술자 fd를 설정한다. select 모듈에 있는 것 같은 함수로 파일 기술자를 폴링하는 프로그램에서 시그널을 처리하는 데 사용할 수 있다. 제대로 작동하려면 파일 기술자 fd를 반드시 널블로킹 모드에서 열어야 한다.

setitimer(which, seconds [, interval])

seconds 초 후에 시그널을 생성하고 그 다음부터 매 interval 초마다 반복해서 시그널을 생성하는 내부 타이머를 설정한다. 이 두 매개변수 모두 부동 소수점 수로 지정한다. which 매개변수는 ITIMER_REAL, ITIMER_VIRTUAL, ITIMER_PROF 중 하나이다. which 매개변수는 타이머가 만료되었을 때 어떤 시그널을 발생시킬 것인지를 결정한다. ITIMER_REAL은 SIGALRM을, ITIMER_VIRTUAL은 SIGVTALRM을, ITIMER_PROF는 SIGPROF를 생성한다. seconds를 0으로 설정하여 타이머를 제거할 수 있다. 기존 타이머 설정을 담은 튜플 (seconds, interval)을 반환한다.

siginterrupt(signalnum, flag)

주어진 시그널 번호에 대한 시스템 호출 재시작 방식을 설정한다. flag가 False이면 시스템 호출이 시그널 signalnum으로 인터럽트될 경우 시스템 호출을 자동으로 다시 시작한다. flag가 True이면 시스템 호출이 인터럽트된다. 인터럽트된 시스템 호출은 보통 여러 번호가 errno.EINTR 또는 errno.EAGAIN인 OSError 또는 IOError 예외를 발생시킨다.

signal(signalnum, handler)

시그널 signalnum에 시그널 처리기 handler를 설정한다. handler는 인수 두 개 (시그널 번호, 프레임 객체)를 받는 호출 가능한 파이썬 객체이어야 한다. SIG_IGN 와 SIG_DFL를 각각 시그널을 무시하거나 기본 시그널 처리기를 사용하고자 할 때 사용하면 된다. 반환값은 이전 시그널 처리기, SIG_IGN 또는 SIG_DFL이다. 스레드를 사용할 때는 주 스레드에서만 이 함수를 호출할 수 있다. 그렇지 않을 경우 ValueError 예외가 발생한다.

개별 시그널은 SIG* 형태의 기호 상수를 사용하여 식별한다. 이 상수들은 머신 종속적인 정수 값에 대응한다. 다음은 일반적으로 사용되는 값들을 보여준다.

시그널 이름	설명
SIGABRT	비정상 종료
SIGALRM	알람
SIGBUS	버스 에러
SIGCHLD	자식 상태 변화
SIGCLD	자식 상태 변화
SIGCONT	계속
SIGFPE	부동 소수점 에러
SIGHUP	정지
SIGILL	잘못된 명령어
SIGINT	터미널 인터럽트 문자
SIGIO	비동기 I/O
SIGIOT	하드웨어 결함
SIGKILL	종료
SIGPIPE	파이프에 쓴. 수신자 없음
SIGPOLL	폴링 가능 이벤트
SIGPROF	프로파일링 알람
SIGPWR	파워 결함
SIGQUIT	터미널 종료 문자
SIGSEGV	세그멘테이션 결함
SIGSTOP	멈춤
SIGTERM	종료
SIGTRAP	하드웨어 결함
SIGTSTP	터미널 중단 문자
SIGTTIN	TTY 제어
SIGTTOU	TTY 제어
SIGURG	긴급한 상태
SIGUSR1	사용자 정의
SIGUSR2	사용자 정의

SIGVTALRM	가상 시간 알람
SIGWINCH	창 크기 변경
SIGXCPU	CPU 제한 초과
SIGXFSZ	파일 크기 제한 초과

이 모듈에는 다음 변수들도 있다.

변수	설명
SIG_DFL	기본 시그널 처리기를 호출하는 시그널 처리기
SIG_IGN	시그널을 무시하는 시그널 처리기
NSIG	가장 높은 시그널 번호보다 하나 큰 값

예

다음은 네트워크 연결을 맺을 때 타임아웃 기능을 시그널로 구현하는 예이다 (socket 모듈에 이미 타임아웃 옵션이 있으며 이 예는 단지 signal 모듈의 기본 사용법을 보여주기 위한 것이다).

```
import signal, socket
def handler(signum, frame):
    print 'Timeout!'
    raise IOError, 'Host not responding.'
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)          # 5초짜리 알람
sock.connect('www.python.org', 80)  # 연결
signal.alarm(0)          # 알람 제거
```

Note

- SIGCHLD(작동 방식이 구현마다 다름)를 제외하고 시그널 처리기는 명시적으로 재설정하지 않는 한 설치된 채로 남아 있다.
- 임시로 시그널을 비활성화하는 것은 불가능하다.
- 시그널은 파일 인터프리터의 원자적 명령 수행 사이에만 처리된다. C에서 오랜 계산이 이루어질 경우(확장 모듈에서 이런 일이 가능) 시그널 전달이 늦어질 수 있다.
- I/O 연산 중에 시그널이 발생하면 I/O 연산은 예외를 발생시키며 실패할 수 있다. 이 경우 인터럽트된 시스템 호출을 알리기 위해 errno 값이 errno.EINTR로 설정된다.
- SIGSEGV 같은 몇몇 시그널은 파일에서 처리할 수 없다.
- 파일에서는 기본으로 몇 가지의 시그널 처리기를 설치한다. SIGPIPE는 무시되고, SIGINT는 KeyboardInterrupt 예외로 변환되며, SIGTERM은 정리 작업을 수행하고 sys.exitfunc을 호출하기 위해 잡는다.
- 같은 프로그램에서 시그널과 스레드를 함께 사용할 때 특별히 주의해야 한다. 현재 주 실행 스레드에서만 새로운 시그널 처리기를 설정하거나 시그널을 받을 수 있다.
- 윈도에서 시그널 처리 기능은 제한적이다. 지원하는 시그널 개수가 극히 적다.

subprocess

subprocess 모듈은 새로운 프로세스 생성, 입출력 스트림 제어, 반환 코드 처리 작업을 일반화한 함수와 객체들을 제공한다. 이 모듈은 os, popen2, commands 같은 다양한 다른 모듈에 있는 기능을 한곳에 모아놓았다.

Popen(args, **parms)

새 명령을 하위 프로세스로 실행하고 새 프로세스를 나타내는 Popen 객체를 반환한다. args에 있는 명령은 ‘ls -l’ 같은 문자열 또는 [‘ls’, ‘-l’] 같은 문자열 리스트로 지정한다. parms는 하위 프로세스의 다양한 속성을 제어하기 위해 설정하는 키워드 인수들을 나타낸다. 다음 키워드 인수들을 인식한다.

키워드	설명
bufsize	버퍼링 방식을 지정한다. 0은 버퍼링하지 않고, 1은 줄 버퍼링을 하며, 음수 값은 시스템 기본 값을 사용한다. 그 밖에 양수 값은 대략적인 버퍼 크기를 나타낸다. 기본 값은 0이다.
close_fds	True이면 자식 프로세스를 실행하기 전에 0, 1, 2를 제외한 모든 파일 기술자를 닫는다. 기본 값은 False이다.
creation_flags	윈도에서 프로세스 생성 플래그를 지정한다. 현재 사용할 수 있는 플래그는 CREATE_NEW_CONSOLE뿐이다. 기본 값은 0이다.
cwd	명령을 실행할 디렉터리. 실행 전에 자식 프로세스의 현재 디렉터리를 cwd로 변경한다. 기본 값은 None이고 부모 프로세스의 현재 디렉터리를 사용한다.
env	새로운 프로세스의 환경 변수를 담은 사전. 기본 값은 None이고 부모 프로세스의 환경 변수를 사용한다.
executable	사용할 실행 프로그램의 이름을 지정한다. 프로그램 이름이 이미 args에 담겨져 있기 때문에 사용할 일이 거의 없다. shell이 주어지면 이 매개변수는 셸의 이름을 지정한다. 기본 값은 None이다.
pexec_fn	명령을 실행하기 바로 전에 자식 프로세스에서 호출할 함수를 지정한다. 이 함수는 인수를 받지 않는다.
shell	True이면 os.system() 함수처럼 작동하는 유닉스 셸을 사용하여 명령을 실행한다. 기본 셸은 /bin/sh이지만 executable 설정을 통해 변경할 수 있다. shell의 기본 값은 None이다.
startupinfo	윈도에서 프로세스를 생성할 때 사용하는 시동 플래그를 제공한다. 기본 값은 None이다. 가능한 값은 STARTF_USESHOWWINDOW와 STARTF_USESTDHANDLERS이다.
stderr	자식 프로세스에서 stderr에 사용할 파일을 나타내는 파일 객체. open()로 생성한 파일 객체, 정수 파일 기술자 또는 새로운 파일파이프가 생성되었음을 나타내는 특수 값 PIPE가 될 수 있다. 기본 값은 None이다.
stdin	자식 프로세스에서 stdin에 사용할 파일을 나타내는 파일 객체. stderr

<code>stdout</code>	에 사용할 수 있는 값을 사용할 수 있다. 기본 값은 <code>None</code> 이다.
<code>universal_newlines</code>	자식 프로세스에서 <code>stdout</code> 에 사용할 파일을 나타내는 파일 객체. <code>stderr</code> 에 사용할 수 있는 값을 사용할 수 있다. 기본 값은 <code>None</code> 이다.
	<code>True</code> 이면 <code>stdin</code> , <code>stdout</code> , <code>stderr</code> 를 나타내는 파일들을 보편 줄바꿈 모드를 사용한 텍스트 모드로 연다. 자세한 설명은 <code>open()</code> 함수를 참고 한다.

`call(args, **parms)`

단순히 명령을 실행하고 `status` 코드를 반환하는 것(즉, `Popen` 객체를 반환하는 것이 아니라)을 제외하고는 `Popen()`과 같다. 이 함수는 출력을 캡처하거나 기타 다른 식으로 제어하려는 경우가 아닌 간단히 명령을 실행하는 데 좋다. 매개변수는 `Popen()`에서와 같은 의미를 가진다.

`check_call(args, **parms)`

종료 코드가 0이 아닐 때 `CalledProcessError` 예외가 발생하는 것을 제외하고는 `call()`과 같다. 이 예외는 `returncode` 속성에 종료 코드로 저장한다.

`Popen()`에서 반환되는 `Popen` 객체 `p`는 하위 프로세스와 상호작용하는 데 사용할 수 있는 다양한 메서드와 속성을 제공한다.

`p.communicate([input])`

`input`로 주어진 데이터를 자식 프로세스의 표준 입력으로 전달하여 자식 프로세스와 통신한다. 일단 데이터를 전달하고 나면 표준 출력과 표준 에러에서 결과를 수집하면서 프로세스가 종료되기를 기다린다. 문자열인 `stdout`, `stderr`를 담은 튜플 (`stdout`, `stderr`)를 반환한다. 자식 프로세스로 보내는 데이터가 없으면 `input`을 `None`으로 설정한다(기본 값).

`p.kill()`

유닉스에서는 `SIGKILL` 시그널을 전달하여 하위 프로세스를 죽이고 윈도에서는 `p.terminate()` 메서드를 호출하여 하위 프로세스를 죽인다.

`p.poll()`

`p`가 종료되었는지를 확인한다. 종료되었으면 하위 프로세스의 반환 코드를 반환한다. 그렇지 않으면 `None`을 반환한다.

`p.send_signal(signal)`

하위 프로세스로 시그널을 보낸다. `signal`은 `signal` 모듈에 정의된 시그널 번호이

다. 윈도에서 SIGTERM 시그널만 지원한다.

p.terminate()

유닉스에서는 SIGTERM 시그널을 전달하여 하위 프로세스를 종료하고 윈도에서는 Win32 API의 TerminateProcess 함수를 호출하여 하위 프로세스를 종료한다.

p.wait()

p가 종료되기를 기다리고 반환 코드를 반환한다.

p.pid

자식 프로세스의 프로세스 ID

p.returncode

프로세스의 숫자 반환 코드. None이면 프로세스는 아직 종료되지 않았음을 의미한다. 음수이면 시그널로 프로세스가 종료되었음을 나타낸다(유닉스).

p.stdin, p.stdout, p.stderr

해당하는 I/O 스트림이 파일로 열릴 때마다 예를 들어, Popen()에 stdout 인수를 PIPE로 설정) 이 속성들이 열린 파일 객체들로 설정된다. 이 파일 객체들은 파일을 다른 프로세스에 연결하는 데 사용한다. 이 속성들은 파일이 사용되지 않으면 None으로 설정된다.

예

```
# 기본 시스템 명령을 실행한다. os.system()과 비슷하다.
ret = subprocess.call("ls -l", shell=True)

# 기본 시스템 명령을 조용히 실행한다.
ret = subprocess.call("rm -f *.java", shell=True,
                      stdout=open("/dev/null"))

# 시스템 명령을 실행하고 결과를 캡처한다.
p = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
out = p.stdout.read()

# 명령을 실행하고 입력을 전달하고 결과를 받는다.
p = subprocess.Popen("wc", shell=True, stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE, stderr=subprocess.PIPE)
out, err = p.communicate(s)      # 문자열 s를 프로세스에 보낸다

# 두 하위 프로세스를 생성하고 파일을 통해 서로 연결한다.
p1 = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
p2 = subprocess.Popen("wc", shell=True, stdin=p1.stdout,
```

```
    stdout=subprocess.PIPE)  
out = p2.stdout.read()
```

Note

- 보통 셸 명령을 명령줄에 전달할 때는 단일 문자열 대신 문자열 리스트로 전달하는 것이 좋다(예를 들어, 'wc filename' 대신 ['wc', 'filename']). 많은 시스템에서 이상한 문자나 스페이스를 파일 이름에 넣을 수 있기 때문이다(예를 들어, 윈도에서 "Documents and Settings" 폴더). 명령을 리스트로 지정하면 아무 문제 없이 잘 작동한다. 셸 명령을 직접 지정하는 경우에는 특수문자나 스페이스를 적절히 탈출시켜야 한다.
- 윈도에서 파이프는 이진 파일 모드로 열린다. 따라서 하위 프로세스에서 텍스트 결과를 읽을 때 줄 끝에 추가 캐리지 리턴 문자('rn' 대신에 'rnrn')가 포함된다. 이 부분이 신경 쓰이면 Popen()에 universal_newlines 옵션을 지정하면 된다.
- subprocess 모듈은 터미널이나 TTY에서 실행될 프로세스를 제어하는 데 사용할 수 없다. 대표적인 예로 사용자가 비밀번호를 입력하기를 기다리는 프로그램(ssh, ftp, svn 등)을 들 수 있다. 이러한 프로그램을 제어하려면 널리 알려진 유닉스 유틸리티인 'Expect'를 바탕으로 하는 써드 파티 모듈을 살펴보도록 하라.

time

time 모듈은 시간과 관련된 다양한 함수를 제공한다. 파이썬에서 시간은 에포크(epoch) 아래 초로 시간을 측정한다. 에포크(epoch)는 시간의 출발점(time=0인 시점)을 나타낸다. 유닉스에서 에포크는 1970년 1월 1일이며 다른 시스템에서는 time.gmtime(0)을 호출하여 얻을 수 있다.

이 모듈에는 다음 변수들이 정의되어 있다.

accept2dyear

두 자리 연도를 받는지 아닌지를 나타내는 불리언 값. 일반적으로 True이지만 환경 변수 \$PYTHON2K를 빈 문자열이 아닌 값으로 설정하면 False가 된다. 이 값을 직접 변경할 수도 있다.

altzone

일광 절약 시간(DST: Daylight Saving Time) 동안 사용할 시간대(적용될 경우).

daylight

DST 시간대가 정의된 경우 0이 아닌 값으로 설정됨.

timezone

지역(DST가 아닌) 시간대.

tzname

지역 시간대 이름과 지역 일광 절약 시간대 이름(정의되면)을 담은 튜플.

다음 함수들을 사용할 수 있다.

asctime([tuple])

gmtime() 또는 localtime()에서 반환되는 시간을 나타내는 튜플을 ‘Mon Jul 12 14:45:23 1999’ 형식의 문자열로 변환한다. 인수를 지정하지 않으면 현재 시간을 사용한다.

clock()

초로 나타낸 현재 CPU 시간을 부동 소수점 수로 반환한다.

ctime([secs])

에포크 이래 초로 표현한 시간을 문자열로 된 지역 시간으로 변환한다. ctime(secs)은 asctime(localtime(secs))과 같다. secs를 생략하거나 None으로 설정하면 현재 시간을 사용한다.

gmtime([secs])

에포크 이래 초로 표현한 시간을 UTC 시간(그리니치 표준시라고도 함)으로 변환한다. 이 함수는 다음 속성을 가진 struct_time 객체를 반환한다.

속성	값
tm_year	1998처럼 네 자리 값
tm_mon	1-12
tm_mday	1-31
tm_hour	0-23
tm_min	0-59
tm_sec	0-61
tm_wday	0-6(0=월요일)
tm_yday	1-366
tm_isdst	-1, 0, 1

tm_isdst 속성은 DST가 적용되면 1, 적용되지 않으면 0, 관련된 정보가 없을 경우 -1이다. secs를 생략하거나 None을 지정하면 현재 시간을 사용한다. 하위 호환성을 위해 반환되는 struct_Time 객체는 앞에서 나열한 순서대로 속성들을 담은 9개 항

목 투플처럼 작동하기도 한다.

localtime([secs])

지역 시간대에 해당하는, gmtime()에서 반환하는 것과 같은 struct_time 객체를 반환한다. secs를 생략하거나 None을 지정하면 현재 시간을 사용한다.

mktime(tuple)

이 함수는 struct_time 객체 또는 지역 시간대 시간을 나타내는 투플(localtime())에서 반환하는 것과 같은 형식으로 된)을 입력으로 받고 에포크 아래 초를 나타내는 부동 소수점 수를 반환한다. 입력 값이 올바른 시간이 아니면 OverflowError 예외가 발생한다.

sleep(secs)

호출 스레드를 secs 초 동안 잠재운다. secs는 부동 소수점 수이다.

strftime(format [, tm])

gmtime() 또는 localtime()에서 반환하는 것 같은 struct_time 객체 tm을 문자열로 변환한다(하위 호환성을 위해 tm은 시간 값을 담은 투플일 수도 있다). format 은 다음 포맷 코드를 담을 수 있는 포맷 문자열이다.

지시문	의미
%a	로케일에 따른 축약 요일 이름
%A	로케일에 따른 비축약 요일 이름
%b	로케일에 따른 축약 달 이름
%B	로케일에 따른 비축약 달 이름
%C	로케일에 따른 날짜와 시간 표현
%d	십진수로 월일[01-31]
%H	십진수로 시(24시간 시계)[00-23]
%I	십진수로 시(12시간 시계)[01-12]
%j	십진수로 연일[001-366]
%m	십진수로 달[01-12]
%M	십진수로 분[00-59]
%p	AM 또는 PM에 대응하는 로케일에 따른 표현
%S	십진수로 초[00-61]
%U	일년 중 주 번호를 나타내는 수[00-53](일요일이 첫날)
%w	십진수로 요일[0-6](0 = 일요일)
%W	일년 중 주 번호를 나타내는 수(월요일이 첫날)
%x	로케일에 따른 날짜 표현
%X	로케일에 따른 시간 표현
%y	세기(century)를 제외한 십진수 연도[00-99]

%Y	세기를 포함한 십진수 연도
%Z	시간대 이름(시간대 없으면 문자 안 나옴)
%%	% 문자

포맷 코드에는 문자열에서 %연산을 사용할 때와 동일하게 폭과 정밀도를 지정 할 수 있다. tm에서 범위를 벗어난 필드가 있으면 ValueError가 발생한다. tm을 생략하면 현재 시간을 나타내는 튜플을 사용한다.

strftime(string [, format])

시간을 표현한 문자를 파싱하여 localtime() 또는 gmtime()에서 반환되는 것과 같은 struct_time 객체를 반환한다. format 매개변수는 strftime()에서 사용되는 것과 같은 포맷 지정자이고 기본 값은 ‘%a %b %d %H:%M:%S %Y’이다. ctime() 함수에서 생성하는 것과 같은 형식이다. 문자열을 파싱할 수 없으면 ValueError 예외가 발생한다.

time()

현재 시간을 UTC에서 에포크 아래 초로 반환한다.

tzset()

유닉스에서 TZ 환경 변수 값을 바탕으로 시간대를 재설정한다. 다음 예를 보자.

```
os.environ['TZ'] = 'US/Mountain'
time.tzset()

os.environ['TZ'] = "CST+06CDT,M4.1.0,M10.5.0"
time.tzset()
```

Note

- 두 자리 연도 입력을 받은 경우에는 입력된 연도를 POSIX X/Open 표준에 따라 네 자리 연도로 변환한다. 69–99는 1969–1999로, 0–68은 2000–2068로 변환된다.
- 시간 함수의 정확도는 시간을 표현하는 데 사용한 단위 이상으로 떨어질 수 있다. 예를 들어, 운영체제는 시간을 초당 50번에서 100번 정도만 갱신한다.

참고

datetime(413페이지)

winreg

winreg 모듈(파이썬 2에서는 _winreg)은 윈도 레지스트리에 대한 저수준 인터페이스를 제공한다. 레지스트리는 각 노드를 키(key)라고 부르는 커다란 계층 트리이다. 특정 키의 자식을 하위 키(subkeys)라고 부르고 하위 키는 다시 추가로 하위 키 또는 값을 가질 수 있다. 예를 들어, 보통 파이썬 sys.path 변수의 설정 정보는 다음 레지스트리에 들어 있다.

```
\HKEY_LOCAL_MACHINE\Software\Python\PythonCore\2.6\PythonPath
```

여기서 Software는 HKEY_LOCAL_MACHINE의 하위 키이고 Python은 Software의 하위 키이다. PythonPath 키 값은 실제 경로 설정 정보를 담는다.

키는 열림과 닫힘 연산으로 접근한다. 열린 키는 특수 핸들(윈도에서 보통 사용하는 정수 핸들 식별자를 감싸를 래퍼)로 표현한다.

CloseKey(key)

이전에 열린 핸들 key에 해당하는 레지스트리 키를 닫는다.

ConnectRegistry(computer_name, key)

다른 컴퓨터에 미리 정의되어 있는 레지스트리 키에 대한 핸들을 반환한다. computer_name은 \\computername처럼 원격 머신의 이름을 담은 문자열이다. computer_name이 None이면 지역 레지스트리를 사용한다. key는 HKEY_CURRENT_USER 또는 HKEY_USERS 같은 미리 정의된 핸들이다. 실패하면 EnvironmentError가 발생한다. 다음 목록은 _winreg 모듈에 정의된 모든 HKEY_* 값을 보여준다.

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_DYN_DATA
- HKEY_LOCAL_MACHINE
- HKEY_PERFORMANCE_DATA
- HKEY_USERS

CreateKey(key, sub_key)

키를 생성하거나 열고 핸들을 반환한다. key는 이전에 열린 키이거나 HKEY_* 상수로 미리 정의된 키이다. sub_key는 생성하거나 열 키 이름이다. key가 미리 정의된 키이면 sub_key는 None이 될 수 있다. 이 경우 key를 반환한다.

DeleteKey(key, sub_key)

sub_key를 삭제한다. key는 열린 키 또는 미리 정의된 HKEY_* 상수 중 하나이다. sub_key는 삭제할 키를 식별하는 문자열이다. sub_key는 하위 키를 가지고 있지 않아야 한다. 그렇지 않으면 EnvironmentError가 발생한다.

DeleteValue(key, value)

레지스트리 키에서 이름 있는 값을 삭제한다. key는 열린 키 또는 미리 정의된 HKEY_* 상수 중 하나이다. value는 삭제할 값의 이름을 담은 문자열이다.

EnumKey(key, index)

색인을 사용해 하위 키의 이름을 반환한다. key는 열린 키 또는 미리 정의된 HKEY_* 상수 중 하나이다. index는 추출할 키를 지정하는 정수이다. index가 범위를 벗어난 경우 EnvironmentError가 발생한다.

EnumValue(key, index)

열린 키 값을 반환한다. key는 열린 키 또는 미리 정의된 HKEY_* 상수이다. index는 추출한 값을 지정하는 정수이다. 이 함수는 튜플 (name, data, type)을 반환한다. 여기서 name은 값 이름이고 data는 값 데이터를 담은 객체이며 type은 값 데이터의 타입을 나타내는 정수이다. 현재 다음 타입 코드들이 정의되어 있다.

코드	설명
REG_BINARY	이진 데이터
REG_DWORD	32비트 숫자
REG_DWORD_LITTLE_ENDIAN	32비트 리틀 엔디언 숫자
REG_DWORD_BIG_ENDIAN	32비트 빅 엔디언 숫자
REG_EXPAND_SZ	확장되지 않은 환경 변수에 대한 참조를 담은 널로 종료되는 문자열
REG_LINK	유니코드 심벌릭 링크
REG_MULTI_SZ	널로 종료되는 문자열들의 순서열
REG_NONE	정의되지 않은 값 타입
REG_RESOURCE_LIST	장치 드라이버 자원 목록
REG_SZ	널로 종료되는 문자열

ExpandEnvironmentStrings(s)

유니코드 문자열 s에서 %name% 형태의 환경 변수 문자열들을 확장한다.

FlushKey(key)

key의 속성들을 레지스트리에 쓰고 변경 사항을 디스크에 쓴다. 레지스트리 데이터를 확실히 디스크에 저장해야 할 필요가 있을 때에 사용한다. 데이터를 다 쓰기 전까지 반환하지 않는다. 보통 경우에는 이 함수를 사용할 일이 없다.

RegLoadKey(key, sub_key, filename)

하위 키를 생성하고 파일에서 정보를 읽어서 하위 키에 저장한다. key는 열린 키 또는 미리 정의된 HKEY_* 상수이다. sub_key는 불러올 하위 키를 식별하는 문자열이다. filename은 데이터를 읽을 파일 이름이다. 이 파일의 내용은 반드시 SaveKey() 함수를 통해 쓰여져야 하고 이 작업을 수행하기 위해 호출 프로세스가 SE_RESTORE_PRIVILEGE 권한을 갖고 있어야 한다. key가 ConnectRegistry()에서 반환된 것이면 filename을 원격 컴퓨터에 상대적인 경로로 지정해야 한다.

OpenKey(key, sub_key [, res [, sam]])

키를 연다. key는 열린 키 또는 HKEY_* 상수이다. sub_key는 열고자 하는 하위 키를 식별하는 문자열이다. res는 예약된 정수로서 반드시 0이어야 한다(기본 값). sam은 키를 위해 보안 접근 마스크를 정의하는 정수이다. 기본 값은 KEY_READ이다. 다음은 sam에 사용할 수 있는 값들을 보여준다.

- KEY_ALL_ACCESS
- KEY_CREATE_LINK
- KEY_CREATE_SUB_KEY
- KEY_ENUMERATE_SUB_KEYS
- KEY_EXECUTE
- KEY_NOTIFY
- KEY_QUERY_VALUE
- KEY_READ
- KEY_SET_VALUE
- KEY_WRITE

OpenKeyEx()

OpenKey()와 같다.

QueryInfoKey(key)

key와 관련된 정보를 튜플 (num_subkeys, num_values, last_modified)로 반환한다. 여기서 num_subkeys는 하위 키의 개수이고 num_values는 값의 개수를 나타내며 last_modified는 최종 수정 시간을 담은 긴 정수이다. 시간은 1601년 1월 1일부터 측정되며 단위는 100나노초이다.

QueryValue(key, sub_key)

key에 대해서 이름 없는 값을 문자열로 반환한다. key는 열린 키 또는 HKEY_* 상수이다. sub_key는 사용할 하위 키 이름이다(있을 경우). 생략하면 key와 연관된 값을 대신 반환한다. 이 함수는 이름 없는 값 중 첫 번째 것을 반환한다. 타입은 반환하지 않는다(원할 경우 QueryValueEx를 대신 사용).

QueryValueEx(key, value_name)

key에 대한 데이터 값과 타입을 담은 튜플 (value, type)을 반환한다. key는 열린 키 또는 HKEY_* 상수이다. value_name은 반환할 값의 이름이다. 반환되는 타입은 앞에 나온 EnumValue() 함수에 설명한 정수 코드 중 하나이다.

SaveKey(key, filename)

key와 모든 하위 키를 파일에 저장한다. key는 열린 키 또는 미리 정의된 HKEY_* 상수이다. filename은 반드시 미리 존재하지 않아야 하며 파일 확장자를 포함하면 안 된다. 또한 연산이 성공하려면 호출자는 백업 권한을 가지고 있어야 한다.

SetValue(key, sub_key, type, value)

키 값을 설정한다. key는 열린 키 또는 HKEY_* 상수이다. sub_key는 값에 연결 할 하위 키 이름이다. type은 정수 타입 코드이며 현재 REG_SZ만 사용할 수 있다. value는 값 데이터를 담은 문자열이다. sub_key는 존재하지 않으면 생성된다. 이 함수를 실행하려면 key가 KEY_SET_VALUE 접근 권한으로 열린 것이어야 한다.

SetValueEx(key, value_name, reserved, type, value)

키 값을 설정한다. key는 열린 키 또는 HKEY_* 상수이다. value_name은 값 이름이다. type은 EnumValue() 함수에서 설명한 정수 타입 코드이다. value는 새로운 값을 담은 문자열이다. 숫자 타입 값(예를 들어, REG_DWORD)을 설정할 때도 value는 무가공 데이터를 담은 문자열이다. 이 문자열은 struct 모듈을 사용하여 생성할 수 있다. reserved는 현재 무시되며 아무 값이나 설정할 수 있다(값이 사용되지 않는다).

Note

- 원도 HKEY 객체를 반환하는 함수는 클래스 PyHKEY로 정의되는 특수한 레지스트리 핸들 객체를 반환한다. 이 객체는 int()를 사용하여 원도 핸들 값으로 변환할 수 있다. 또한 컨텍스트 관리 프로토콜에 사용하면 내부 핸들이 자동으로 닫힌다. 다음 예를 보자.

```
with winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE, "spam") as key:  
    문장들
```

20장

Python Essential Reference

스레드와 동시성

이 장에서는 파이썬에서 동시에 실행되는 프로그램을 작성하는 데 필요한 라이브러리 모듈을 설명한다. 스레드, 메시지 전달, 다중 처리, 코루틴 등의 주제를 다룬다. 구체적인 라이브러리 모듈을 다루기 전에 기본 개념을 먼저 익혀보자.

기본 개념

실행되고 있는 프로그램을 프로세스(process)라고 부른다. 각 프로세스는 메모리, 열린 파일 목록, 실행 중인 명령을 추적하는 프로그램 카운터, 함수의 지역 변수들을 저장하기 위해 사용되는 스택 등을 포함하는 자신만의 시스템 상태 정보를 가진다. 보통 프로세스는 주 스레드(main thread)라고 부르는 제어 흐름 순서에 따라 명령들을 하나씩 실행한다. 주어진 시점에서 프로그램은 한 가지 일만 한다.

프로그램 안에서 os 또는 subprocess 모듈에 있는 라이브러리 함수(os.fork(), subprocess.Popen() 등)를 사용하여 새로운 프로세스들을 생성할 수 있다. 하위 프로세스라고 부르는 이 프로세스들은 완전히 독립된 개체로서 실행된다. 각 하위 프로세스는 자신만의 개인 시스템 상태와 주 실행 스레드를 가진다. 하위 프로세스는 독립되어 있기 때문에 기존 프로세스와 동시에 실행된다. 즉, 하위 프로세스를 생성한 프로세스가 다른 작업을 계속하는 동안 하위 프로세스는 무대 뒤에서 자신만의 일을 수행한다.

프로세스들은 분리되어 있지만 서로 통신할 수 있는데 이를 프로세스 사이 통신 (IPC: interprocess communication)이라고 한다. IPC의 가장 대표적인 형식은 메시

지 전달(message passing)에 기초한다. 메시지(message)는 간단히 무가공 바이트 버퍼이다. send()와 recv() 같은 기본 연산을 사용해서 파일이나 네트워크 소켓 같은 I/O 채널을 통해 메시지를 전달 또는 수신할 수 있다. 덜 자주 사용되는 다른 IPC 메커니즘은 메모리에 매핑된 지역(memory-mapped region)을 기반으로 한다 (mmap 모듈 참고). 메모리 매핑에서는 프로세서들이 공유된 메모리 지역을 생성할 수 있다. 공유 지역을 수정하면 이 지역을 관찰하고 있는 모든 프로세스를 볼 수 있다.

동시에 여러 작업을 수행해야 하는 응용 프로그램에서 다중 프로세스를 사용할 수 있다. 이때 각 프로세스는 처리 과정의 일부를 책임진다. 한 가지 일을 여러 작업으로 나누어 수행하는 또 다른 방법으로 스레드를 사용하는 방법이 있다. 스레드(thread)는 자신만의 제어 흐름과 실행 스택을 가진다는 점에서 프로세스와 비슷하다. 하지만, 스레드는 자신을 생성한 프로세스 안에서 실행되며 모든 데이터와 시스템 자원을 공유한다. 스레드는 응용 프로그램에서 작업을 동시에 수행하기를 원하지만 잠재적으로 시스템 상태의 많은 부분을 공유해야 할 필요가 있는 경우에 유용하게 쓸 수 있다.

다중 프로세스나 스레드를 사용할 때 호스트 운영체제는 작업 스케줄링을 해야 한다. 운영체제에서는 각 프로세스(또는 스레드)에 작은 시간 조각을 할당하고 모든 활성 작업들을 빠르게 반복해서 방문한다. 그러는 동안 활용할 수 있는 CPU 사이클의 일부를 각 프로세스에 부여한다. 예를 들어, 여러분의 시스템에 10개의 실행 중인 프로세스가 있다면 운영체제는 각 프로세스에 대략 10분의 1씩 CPU 시간을 할당하고 빠른 속도로 차례로 프로세스들을 방문한다. 하나 이상의 CPU 코어를 가진 시스템에서 운영체제는 각 CPU를 바쁘게 만들기 위해서 프로세서들을 병렬로 실행하는 스케줄링 작업을 수행할 수도 있다.

동시 실행의 이점을 활용하는 프로그램을 작성하는 것은 태생적으로 복잡한 일이다. 이러한 복잡함은 주로 동기화와 공유 데이터 접근으로 인해 발생한다. 특히 여러 작업에서 데이터 구조를 동시에 업데이트하게 되면 프로그램 상태가 손상되거나 일관성이 없어질 수 있다(경쟁 조건race condition이라고 부르는 문제). 이러한 문제를 해결하기 위해서 동시 실행 프로그램에서는 코드에서 임계 구역(critical section)을 식별하고 상호 배타적 락(mutual exclusion lock)이나 기타 동기화 기능을 사용하여 이를 보호하여야 한다. 예를 들어, 여러 스레드에서 동시에 같은 파일에 데이터를 쓰는 경우 상호 배타적 락을 사용하여 일단 한 스레드가 쓰기 시작하

면 다른 스레드들은 쓰기가 종료될 때까지 기다리게 만들므로써 이 연산을 동기화 할 수 있다. 다음은 이 시나리오를 구현하는 코드의 전형적인 모습을 보여준다.

```
write_lock = Lock()
...
# 쓰기가 수행되는 임계 구역
write_lock.acquire()
f.write("Here's some data.\n")
f.write("Here's more data.\n")
...
write_lock.release()
```

제이슨 휘팅턴(Jason Whittington)이 처음 했다고 알려진 농담이 있다. “왜 멀티스레드 닭이 도로를 건넜지? 가려고 으로 반대편(Why did the multithreaded chicken cross the road? to To other side, get the.)” 이 유머는 작업 동기화와 동시 실행 프로그램에서 발생할 수 있는 문제를 잘 나타낸다. 만약 여러분이 머리를 긁으면서 “이해가 잘 안 되는데”라고 말한다면 이 장의 나머지 부분을 계속 읽기 전에 여기에 관해 좀 더 공부를 하는 게 좋을 것이다.

동시 실행 프로그래밍과 파이썬

대부분의 시스템에서 파이썬은 메시지 전달과 스레드 기반 동시 실행 프로그래밍을 모두 지원한다. 프로그래머들은 보통 파이썬 스레드 인터페이스를 좀 더 친숙하게 느끼지만 실제로 파이썬 스레드에는 제약 사항이 좀 있다. 최소한으로 스레드 안전을 보장하기는 하지만 파이썬 인터프리터에서는 내부적으로 전역 인터프리터 락(GIL: global interpreter lock)을 사용하기 때문에 특정 시점에서는 파이썬 스레드 하나만 실행된다. 이 때문에 시스템에 여러 개의 CPU 코어가 있다라도 파이썬 프로그램은 한 개의 프로세서에서만 실행된다. GIL은 파이썬 커뮤니티에서 종종 열띤 논의의 대상이 되는 주제이지만 당분간은 사라질 가능성은 낮다.

GIL의 존재는 많은 파이썬 프로그래머들이 동시 실행 프로그래밍 문제를 다루는 방법에 직접적인 영향을 미친다. 응용 프로그램이 I/O에 의존하는 경우라면 프로세서를 추가로 사용하더라도 대부분의 시간을 이벤트를 기다리는 데 쓰는 프로그램에서는 도움이 되지 않기 때문에 일반적으로 스레드를 사용해도 좋다. 많은 양의 CPU 처리가 필요한 응용 프로그램에서는 스레드를 사용하여 작업을 나누어봐야 이점이 없고 오히려 프로그램을 더 느리게 만들 수 있다(종종 여러분이 그럴 거라고 생각한 것보다 훨씬 느리다). 이런 경우에는 하위 프로세스와 메시지 전달을 사용

하는 것이 나을 수 있다.

스레드를 사용하고 있더라도 많은 프로그래머들이 확장성에 의문을 제기한다. 예를 들어, 스레드 100개에서는 잘 작동하던 스레드 네트워크 서버가 스레드 10,000 개가 될 때는 끔찍한 성능을 보일 수 있다. 일반적으로 스레드 10,000개가 필요한 프로그램은 작성하지 않는 것이 좋다. 그 이유는 각 스레드는 자신만의 시스템 자원을 소모하고 스레드 사이 컨텍스트 전환, 락, 그리고 기타 문제들로 인한 오버헤드가 심각할 정도로 커질 수 있기 때문이다(모든 스레드가 한 CPU에서 수행되어야 한다는 사실은 말할 것도 없이). 이런 문제를 해결하기 위해 응용 프로그램을 비동기 이벤트 처리 시스템 형태로 재구성하는 경우를 종종 볼 수 있다. 예를 들어, 중앙 이벤트 루프에서 select 모듈을 사용하여 모든 I/O 소스를 감시하고 발생하는 비동기 이벤트들을 IO 처리기들로 전달하게 할 수 있다. asyncio 같은 라이브러리 모듈이나 Twisted (<http://twistedmatrix.com>) 같은 유명한 써드 파티 모듈이 이를 기반으로 하고 있다.

나중을 생각했을 때 메시지 전달은 파이썬에서 어떤 종류의 동시 실행 프로그램을 작성하든지 수용할 필요가 있는 개념이다. 스레드로 작업할 때일지라도 종종 추천되는 방식은 응용 프로그램을 메시지 큐를 통해 데이터를 서로 주고받는 독립 스레드들로 구성하는 것이다. 이 접근법은 락이라든지 기타 동기화 관련 기능을 사용해야 할 필요성을 줄여주기 때문에 애러가 발생할 가능성이 적다. 메시지 전달은 네트워킹과 분산 시스템으로도 자연스럽게 확장된다. 예를 들어, 처음부터 프로그램의 일부를 스레드로 만들고 이 스레드에 메시지를 보내는 형태로 작성하면 나중에 이 부분을 별개의 프로세스로 만들거나, 메시지를 네트워크 연결을 통해 보냄으로써 다른 머신으로 옮길 수도 있다. 메시지 전달은 코루틴 등 고급 파이썬 기능과도 연관이 있다. 코루틴은 자신에게 전달된 메시지를 받아서 처리하는 함수이다. 메시지 전달이라는 개념을 익힘으로써 여러분은 훨씬 유연한 프로그램을 작성할 수 있다.

이 장의 나머지 부분에서는 동시 실행 프로그래밍을 지원하는 여러 라이브러리 모듈에 대해 살펴본다. 끝부분에서는 일반적인 프로그래밍 스타일에 관해서 더 자세히 설명한다.

multiprocessing

multiprocessing 모듈은 하위 프로세스에서 작업 수행, 데이터 교환 및 공유, 그리고

다양한 형식의 동기화를 지원한다. 프로그래밍 인터페이스는 threading 모듈에 있는 스레드용 프로그래밍 인터페이스를 따른다. 하지만 스레드와 달리 프로세스는 공유 상태를 갖지 않는다. 따라서, 어떤 프로세스에서 데이터를 수정하면 그것은 다른 프로세스에 보이지 않는다.

multiprocessing 모듈은 매우 다양한 기능을 제공하며 내장 라이브러리 중에서 규모가 크고 상당히 고급 기능을 제공한다. 여기에서 이 모듈에 대한 모든 것을 다루는 일은 불가능하기 때문에 몇 가지 예들과 함께 핵심적인 부분만을 다룬다. 경험 많은 프로그래머라면 이곳에 나온 예를 확장하여 더 큰 문제에 적용할 수 있을 것이다.

프로세스

multiprocessing 모듈의 기능은 프로세스에 맞추어져 있다. 프로세스는 다음 클래스로 표현한다.

```
Process([group [, target [, name [, args [, kwargs]]]]])
```

위의 프로세스에서 실행되는 작업을 나타내는 클래스. 인수들은 항상 키워드 인수로 지정해야 한다. target은 프로세스가 시작될 때 실행할 호출 가능한 객체이고 args는 target으로 전달할 위치 인수들의 튜플이며 kwargs는 target으로 전달할 키워드 인수들의 사전이다. args와 kwargs를 생략하면 target은 인수 없이 호출된다. name은 프로세스의 서술적 이름을 나타내는 문자열이다. group은 사용되지 않고 항상 None으로 설정된다. group 인수가 있는 이유는 단순히 threading 모듈로 스레드를 생성하는 형태를 흉내내기 위함이다.

Process 인스턴스 p는 다음 메서드들을 가진다.

```
p.is_alive()
```

p가 아직 실행 중이면 True를 반환한다.

```
p.join([timeout])
```

프로세스 p가 종료되기를 기다린다. 옵션인 timeout은 타임아웃 시간을 지정한다. 이 함수를 원하는 만큼 여러 번 호출할 수 있지만 자기 자신에 대고 호출하면 안 된다.

p.run()

프로세스가 시작될 때 실행되는 메서드. 기본으로 Process를 생성할 때 전달된 target을 호출한다. 아니면 Process에서 상속받고 run()을 재구현하여 비슷한 일을 수행할 수 있다.

p.start()

프로세스를 시작한다. 해당 프로세스를 나타내는 하위 프로세스를 시작시키고 하위 프로세스에 p.run()을 호출한다.

p.terminate()

프로세스를 강제로 종료시킨다. 이 메서드를 호출하면 프로세스 p는 아무런 정리 작업 없이 즉시 종료된다. 프로세스 p가 자신의 하위 프로세서들을 생성한 경우라면 이들은 좀비로 변한다. 이 메서드를 사용할 때는 주의할 점이 있다. p가 락을 획득하고 있거나 프로세스 사이 통신에 참여하고 있는데 p를 종료할 경우 교착 상태(deadlock)나 I/O 오류를 일으킬 수 있다.

Process 인스턴스 p는 다음 데이터 속성들을 가진다.

p.authkey

프로세스의 인증 키. 직접 설정하지 않으면 os.urandom()으로 생성한 32개 문자를 담은 문자열로 설정된다. 이 키는 네트워크 연결을 통한 저수준 프로세스 사이 통신을 수행할 때 보안 기능을 제공하는 데 쓰인다. 네트워크 연결은 양쪽 끝이 동일한 인증 키를 가지고 있는 경우에만 작동한다.

p.daemon

프로세스가 데몬 프로세스인지 아닌지를 나타내는 불리언 플래그. 데몬 프로세스(daemonic process)는 자신을 생성한 파이썬 프로세스가 종료되면 자동으로 종료된다. 또한 데몬 프로세스는 새로운 프로세스를 생성할 수 없다. p.daemon의 값은 반드시 p.start()로 프로세스를 시작하기 전에 설정해야 한다.

p.exitcode

프로세스의 정수 종료 코드. 프로세스가 실행 중이면 이 값은 None이다. 이 값이 음수이면 -N의 값은 프로세스가 시그널 N에 의해 종료되었음을 나타낸다.

p.name

프로세스의 이름.

p.pid

프로세스의 정수 프로세스 ID.

다음 예는 함수(또는 기타 호출 가능한 객체)를 별개의 프로세스로 실행하는 방법을 보여준다.

```
import multiprocessing
import time

def clock(interval):
    while True:
        print("The time is %s" % time.ctime())
        time.sleep(interval)

if __name__ == '__main__':
    p = multiprocessing.Process(target=clock, args=(15,))
    p.start()
```

다음은 Process에서 상속받은 클래스로 같은 일을 하는 방법을 보여준다.

```
import multiprocessing
import time

class ClockProcess(multiprocessing.Process):
    def __init__(self,interval):
        multiprocessing.Process.__init__(self)
        self.interval = interval
    def run(self):
        while True:
            print("The time is %s" % time.ctime())
            time.sleep(self.interval)

if __name__ == '__main__':
    p = ClockProcess(15)
    p.start()
```

두 예 모두 하위 프로세스에서 15초마다 시간을 출력한다. 플랫폼 사이 이식성을 위해서는 앞에서 보았듯이 주 프로그램 안에서 새 프로세스를 생성해야 한다. 유닉스에서는 꼭 그렇게 하지 않아도 되지만 윈도에서는 그렇다. 또한 윈도에서는 IDLE 같은 파이썬 IDE 말고 명령 셸(cmd.exe)에서 앞에 나온 예를 실행해야 할 것이다.

프로세스 사이 통신

multiprocessing 모듈은 프로세스 사이 통신 방법의 두 가지 주요 형태, 즉 파이프(pipe)와 큐(queue)를 제공한다. 둘 모두 메시지 전달을 사용하여 구현된다. 큐 인터페이스는 스레드 프로그램에서 널리 사용되는 큐 사용 방식을 흡내낸다.

Queue([maxsize])

공유 프로세스 큐를 생성한다. maxsize는 큐 안에 넣을 수 있는 최대 아이템 개수이다. 생략하면 크기에 제한이 없다. 내부 큐는 파이프와 락을 사용하여 구현된다. 또한 큐에 들어 있는 데이터를 내부 파이프에 전달하기 위한 지원 스레드가 하나 시작된다.

Queue 인스턴스 q는 다음 메서드들을 가진다.

q.cancel_join_thread()

프로세스 종료 시 자동으로 백그라운드 스레드에 참여하지 않는다. 이렇게 하면 join_thread() 메서드가 블로킹되지 않는다.

q.close()

큐를 닫아서 더 이상의 데이터가 큐에 추가되지 못하게 한다. 이 메서드가 호출되면 백그라운드 스레드는 아직 쓰지 않은 데이터를 계속해서 쓰고 이 일을 완료하자마자 종료된다. 이 메서드는 q가 쓰레기 수집될 경우 자동으로 호출된다. 큐를 닫는다고 해서 큐 소비자(consumer) 쪽에서 데이터의 끝을 알리는 신호가 전달되거나 예외가 발생되지 않는다. 예를 들어, 소비자가 get() 연산에서 블로킹되어 있는 경우 생성자에서 큐를 종료해도 get()이 에러를 내면서 반환되지 않는다.

q.empty()

호출할 당시 q가 비어 있으면 True를 반환한다. 다른 프로세스나 스레드가 큐에 항목을 추가하고 있으면 이 메서드의 결과는 신뢰할 수 없다(예를 들어, 결과를 받은 다음 사용하려고 하는 사이에 큐에 새로운 항목이 추가될 수 있다).

q.full()

q가 가득 차 있으면 True를 반환한다. 다른 스레드로 인해 이 결과 또한 신뢰할 수 없다. (q.empty() 참고)

q.get([block [, timeout]])

q로부터 항목을 반환한다. q가 비어 있으면 새로운 항목이 나타날 때까지 기다린다. block은 블로킹의 작동 방식을 제어하며 기본 값은 True이다. False로 설정하면 queue가 빈 경우 QueueEmpty 예외(Queue 라이브러리 모듈에 정의된)가 발생한다. 옵션인 timeout은 블로킹 모드에서 사용할 타임아웃 시간을 나타낸다. 지정된 시간 안에 사용할 수 있는 항목이 나타나지 않는 경우 QueueEmpty 예외가 발생한다.

q.get_nowait()

q.get(False)와 동일하다.

q.join_thread()

큐의 백그라운드 스레드에 참여한다. q.close()가 호출된 후 큐의 모든 항목이 소비될 때까지 기다리는 데 사용한다. 이 메서드는 q를 처음으로 생성하지 않은 모든 프로세스에서 기본으로 호출된다. q.cancel_join_thread()를 호출하면 이렇게 작동하지 않게 된다.

q.put(item [, block [, timeout]])

큐에 item을 추가한다. 큐가 가득 차 있으면 공간이 생길 때까지 기다린다. block은 블로킹의 작동 방식을 제어하고 기본 값은 True이다. False로 설정하면 queue가 가득 차 있을 때 Queue.Full 예외(Queue 라이브러리 모듈에 정의된)가 발생한다. timeout은 블로킹 모드에서 공간이 생길 때까지 얼마 동안 기다릴지를 지정한다. 지정된 시간이 지나면 Queue.Full 예외가 발생한다.

q.put_nowait(item)

q.put(item, False)와 동일하다.

q.qsize()

현재 큐의 대략적인 항목의 개수를 반환한다. 결과를 반환한 다음 프로그램에서 사용하기 전에 새로운 항목이 추가되거나 삭제될 수 있기 때문에 결과는 신뢰성이 없다. 어떤 시스템에서는 이 메서드를 호출하면 NotImplementedError 예외가 발생할 수도 있다.

JoinableQueue([maxsize])

참여할 수 있는 공유 프로세스 큐를 생성한다. Queue와 거의 같지만 소비자가 생산자에게 항목을 성공적으로 소비하였다는 것을 알릴 수 있다. 이 통지 과정은 공유 세마포어와 조건 변수를 사용하여 구현된다.

JoinableQueue 인스턴스 q는 Queue와 같은 메서드들을 가진다. 또한 다음 추가 메서드들도 가진다.

q.task_done()

소비자 쪽에서 q.get()에서 반환된 항목을 처리하였다는 것을 알리기 위해 사용한다. 큐에서 제거한 것보다 더 많이 호출할 경우 ValueError 예외가 발생한다.

q.join()

큐에 있는 모든 항목이 처리될 때까지 기다리기 위해서 생산자 쪽에서 사용한다.
큐에 있는 모든 항목에 대해 q.task_done()가 호출될 때까지 기다린다.

다음 예는 큐에 있는 항목을 소비하고 처리하면서 영원히 실행되는 프로세스를 생성하는 방법을 보여준다. 생산자는 큐에 항목을 공급하고 이들이 처리될 때까지 기다린다.

```
import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        # 항목을 처리한다.
        print(item)      # 유용한 작업을 수행하도록 변경
        # 작업 완료를 알림
        input_q.task_done()

def producer(sequence, output_q):
    for item in sequence:
        # 항목을 큐에 추가
        output_q.put(item)

# 설정
if __name__ == '__main__':
    q = multiprocessing.JoinableQueue()
    # 소비자 프로세스를 실행
    cons_p = multiprocessing.Process(target=consumer, args=(q,))
    cons_p.daemon=True
    cons_p.start()

    # 항목을 생성한다. sequence는 소비자로 보낼 항목들을 담은
    # 순서열이다. 실제로는 생성기나 기타 다른 방법으로 생성한다.
    sequence = [1,2,3,4]
    producer(sequence, q)

    # 모든 항목이 처리될 때까지 기다린다.
    q.join()
```

이 예에서 소비자 프로그램은 영원히 실행되며 주 프로그램이 종료되었을 때 소비자 프로세스를 종료시키기 위해서 소비자 프로세스를 데몬으로 설정하였다(이렇게 하지 않으면 프로그램이 멈추어 버린다). 생산자에서 언제 큐에 있는 모든 항목이 처리되었는지를 알기 위해서 JoinableQueue를 사용하였다. join() 연산은 이 일을 한다. join() 호출을 잊어버리면 소비자는 모든 작업을 끝마치기 전에 종료된다.

원활 경우 여러 프로세스가 동일한 큐에서 항목을 추가하거나 얻을 수 있다. 소비자 프로세스 풀이 있다면 다음과 같이 코드를 작성할 수 있다.

```

if __name__ == '__main__':
    q = multiprocessing.JoinableQueue()
    # 얼마간의 소비자 프로세스들을 구동한다.
    cons_p1 = multiprocessing.Process(target=consumer, args=(q,))
    cons_p1.daemon=True
    cons_p1.start()

    cons_p2 = multiprocessing.Process(target=consumer, args=(q,))
    cons_p2.daemon=True
    cons_p2.start()

    # 항목을 생성한다. sequence는 소비자로 보낼 항목들을 담은
    # 순서열이다. 실제로는 생성기나 기타 다른 방법으로 생성한다.
    sequence = [1,2,3,4]
    producer(sequence, q)

    # 모든 항목이 처리될 때까지 기다린다.
    q.join()

```

이 같은 코드를 작성할 때 큐에 있는 모든 항목이 피클링되어 파이프나 소켓 연결을 통해 프로세스로 전달된다는 점을 염두에 둔다. 이 경우 일반적으로 많은 수의 작은 객체들을 전달하는 것보다 적은 수의 큰 객체들을 전달하는 것이 낫다.

어떤 응용 프로그램에서는 생산자에서 더 이상 항목을 생성하지 않을 것이기 때문에 소비자가 종료되어야 한다는 사실을 알고 싶은 경우가 있다. 이렇게 하려면 코드를 작성할 때 종료를 알리는 특별한 값인 표지(sentinel)를 사용하면 된다. 다음은 None을 표지로 사용하여 이 개념을 보여준다.

```

import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        if item is None:
            break
        # 항목을 처리한다
        print(item)      # 유용한 작업을 수행하도록 변경
    # 종료
    print("Consumer done")

def producer(sequence, output_q):
    for item in sequence:
        # 항목을 큐에 추가
        output_q.put(item)

if __name__ == '__main__':
    q = multiprocessing.Queue()
    # 소비자 프로세스를 실행
    cons_p = multiprocessing.Process(target=consumer, args=(q,))
    cons_p.start()

```

```
# 항목들을 생성한다.
sequence = [1,2,3,4]
producer(sequence, q)

# 큐에 표지를 추가하여 종료를 알린다.
q.put(None)
# 소비자 프로세스가 종료할 때까지 기다린다.
cons_p.join()
```

이 예에서처럼 표지를 사용할 때는 모든 개별 소비자에 대해서 큐에 표지를 추가해야 한다. 예를 들어, 큐에서 항목들을 소비하는 소비자 프로세스가 세 개 있는 경우 생산자는 모든 소비자를 종료시키기 위해 큐에 표지 세 개를 추가해야 한다.

프로세스 사이에 메시지를 전달하는 데는 큐 대신 파이프를 사용할 수도 있다.

Pipe([duplex])

프로세스들 사이에 파이프를 생성하고 튜플 (conn1, conn2)을 반환한다. 여기서 conn1과 conn2는 파이프 양 끝을 나타내는 Connection 객체이다. 기본으로 파이프는 양방향성을 가진다. duplex를 False로 설정하면 conn1은 수신을 위해서만 사용할 수 있고 conn2는 송신을 위해서만 사용할 수 있다. 파이프를 사용할 Process 객체들을 생성하여 구동하기 전에 Pipe()를 반드시 먼저 호출해야 한다.

Pipe()가 반환하는 Connection 객체의 인스턴스 c는 다음 메서드와 속성들을 가진다.

c.close()

연결을 종료한다. c가 쓰레기 수집되면 자동으로 호출된다.

c.fileno()

연결에 사용되는 정수 파일 기술자를 반환한다.

c.poll([timeout])

연결을 통해 사용할 수 있는 데이터가 있으면 True를 반환한다. timeout은 기다릴 최대 시간을 나타낸다. 이 값을 생략하면 이 메서드는 결과를 즉시 반환한다. timeout을 None으로 설정하면 데이터가 도착할 때까지 무한정 기다린다.

c.recv()

c.send()로 보내진 객체를 받는다. 연결의 다른 끝 부분이 닫혔거나 더 이상 데이터가 없는 경우 EOFError가 발생한다.

c.recv_bytes([maxlength])

c.send_bytes()로 보낸 바이트 메시지 전체를 받는다. maxlength는 받을 최대 바이트 수를 나타낸다. 받은 메시지가 더 클 경우 IOError가 발생하며 연결을 통해 데이터를 더 읽을 수 없게 된다. 연결의 다른 끝 부분이 닫혔거나 더 이상 데이터가 없는 경우 EOFError가 발생한다.

c.recv_bytes_into(buffer [, offset])

바이트 메시지 전체를 받아서 쓰기 가능한 버퍼 인터페이스를 지원하는 객체 buffer(예를 들어, bytearray 객체나 이와 유사한 객체)에 저장한다. offset은 버퍼에서 메시지를 담을 바이트 오프셋을 지정한다. 이 메서드는 받은 바이트 수를 반환한다. 메시지의 길이가 버퍼의 남은 공간보다 클 경우 BufferTooShort 예외가 발생한다.

c.send(obj)

연결을 통해 객체를 보낸다. obj는 피클링할 수 있는 객체이어야 한다.

c.send_bytes(buffer [, offset [, size]])

연결을 통해 버퍼에 있는 바이트 데이터를 보낸다. buffer는 버퍼 인터페이스를 지원하는 객체이고 offset는 버퍼에서 바이트 오프셋이며 size는 보낼 바이트의 수를 나타낸다. c.recv_bytes()를 한 번 호출하여 받을 수 있도록 데이터를 하나의 메시지로 보낸다.

파이프는 큐와 비슷하게 사용할 수 있다. 다음은 앞에서 본 생산자 소비자 문제를 파이프로 구현한 예이다.

```
import multiprocessing
# 파이프를 통해 항목을 소비한다.
def consumer(pipe):
    output_p, input_p = pipe
    input_p.close()    # 파이프의 입력 쪽 끝을 닫는다.
    while True:
        try:
            item = output_p.recv()
        except EOFError:
            break
        # 항목을 처리한다.
        print(item)    # 유용한 작업을 수행하도록 변경
    # 종료한다.
    print("Consumer done")

# 항목들을 생성하여 파이프에 추가한다. sequence는 처리되어야 할
# 항목들을 나타내는 반복 가능한 객체이다.
```

```

def producer(sequence, input_p):
    for item in sequence:
        # 항목을 파이프에 추가함
        input_p.send(item)

if __name__ == '__main__':
    (output_p, input_p) = multiprocessing.Pipe()
    # 소비자 프로세스를 구동한다.
    cons_p = multiprocessing.Process(target=consumer,
                                    args=((output_p, input_p),))
    cons_p.start()

# 생산자의 출력 파이프를 닫는다.
output_p.close()

# 항목들을 생성한다.
sequence = [1,2,3,4]
producer(sequence, input_p)

# 입력 파이프를 닫아서 종료를 알린다.
input_p.close()

# 소비자 프로세스가 종료될 때까지 기다린다.
cons_p.join()

```

파이프의 끝 부분을 적절하게 관리하는 데 각별한 주의를 기울여야 한다. 파이프의 한 끝을 생산자나 소비자에서 사용하지 않는 경우 반드시 닫아야 한다. 이 때문에 앞의 예에서 생산자 쪽에서는 파이프의 출력 끝을 닫고 소비자 쪽에서는 파이프의 입력 끝을 닫았다. 이 단계를 생략하면 프로그램은 소비자의 recv() 연산에서 멈출 수 있다. 운영체제는 파이프에 대한 참조 횟수를 세며 모든 프로세스에서 파이프를 닫아야 EOFError 예외가 발생한다. 따라서, 소비자 쪽에서 같은 끝 부분을 닫지 않는 한 생산자 쪽에서 파이프를 닫아도 아무런 효과가 없다.

파이프는 양방향 통신에 사용할 수 있다. 클라이언트 서버 컴퓨팅이나 원격 프로시저 호출에서 주로 사용하는 요청 응답 모델을 통해 프로세스와 상호작용하는 프로그램을 작성하고자 할 때 파이프를 사용할 수 있다. 다음 예를 보자.

```

import multiprocessing
# 서버 프로세스
def adder(pipe):
    server_p, client_p = pipe
    client_p.close()
    while True:
        try:
            x,y = server_p.recv()
        except EOFError:
            break
        result = x + y
        server_p.send(result)

```

```

# 종료한다
print("Server done")

if __name__ == '__main__':
    (server_p, client_p) = multiprocessing.Pipe()
    # 서버 프로세스를 구동한다.
    adder_p = multiprocessing.Process(target=adder,
                                      args=((server_p, client_p),))
    adder_p.start()

    # 클라이언트에서 서버 파일을 닫는다.
    server_p.close()

    # 서버에 요청들을 보낸다.
    client_p.send((3,4))
    print(client_p.recv())

    client_p.send(('Hello', 'World'))
    print(client_p.recv())

    # 끝. 파일을 닫는다.
    client_p.close()

    # 소비자 프로세스가 종료될 때까지 기다린다.
    adder_p.join()

```

이 예에서 adder() 함수는 파일의 한쪽 끝으로 도착하는 메시지를 기다리는 서버로서 실행된다. 메시지를 받으면 몇 가지 처리를 수행하고 파일을 통해 결과를 다시 보낸다. send()와 recv()는 pickle 모듈을 사용해 객체를 직렬화한다. 이 예에서 서버는 튜플 (x, y)를 입력으로 받아서 결과 x + y를 반환한다. 원격 프로시저 호출을 사용하는 고급 응용에서는 다음에 설명할 프로세스 풀(process pool)을 사용해야 한다.

프로세스 풀

다음 클래스는 다양한 종류의 데이터 처리 작업을 수행하는 프로세스 풀을 생성하는 데 사용한다. 풀이 제공하는 기능은 리스트 내포와 맵리듀스(map-reduce) 같은 함수형 프로그래밍에서 제공하는 기능과 어느 정도 닮았다.

Pool([numprocess [,initializer [, initargs]]])

일꾼(worker) 프로세스들의 풀을 생성한다. numprocess는 생성할 프로세스의 수를 나타낸다. 생략하면 cpu_count()의 값이 사용된다. initializer는 시작할 때 각 일꾼 프로세스에 대해 실행할 호출 가능한 객체이다. initargs는 initializer에 전달할 인수들의 튜플이다. initializer의 기본 값은 None이다.

Pool의 인스턴스 p는 다음 연산들을 지원한다.

p.apply(func [, args [, kwargs]])

func(*args, **kwargs)를 풀에 있는 한 일꾼에서 실행하고 결과를 반환한다. 이렇게 한다고 해서 func이 모든 풀 일꾼 안에서 병렬로 실행되는 것은 아니다. 서로 다른 인수로 func을 동시에 실행하고 싶다면 다른 스레드에서 p.apply()를 호출하거나 p.apply_async()를 사용해야 한다.

p.apply_async(func [, args [, kwargs [, callback]]])

func(*args, **kwargs)를 풀에 있는 한 일꾼에서 실행하고 비동기적으로 결과를 반환한다. 이 메서드는AsyncResult의 인스턴스를 반환하며 나중에 최종 결과를 얻는 데 사용할 수 있다. callback은 인수 하나를 받는 호출 가능한 객체이다. func의 결과를 사용할 수 있게 되면 즉시 그 결과가 callback으로 전달된다. callback에서 는 블로킹 연산을 수행하면 안 된다. 그럴 경우 다른 비동기 연산에 결과가 전달되는 것을 막게 된다.

p.close()

프로세스 풀을 닫고 추가 연산을 막는다. 아직 실행 중인 연산이 있으면 일꾼 프로세스들이 종료되기 전에 완료된다.

p.join()

모든 일꾼 프로세스가 종료될 때까지 기다린다. close() 또는 terminate()를 호출한 후에만 호출할 수 있다.

p imap(func, iterable [, chunksize])

결과로 리스트 대신에 반복자를 반환하는 map() 버전.

p imap_unordered(func, iterable [, chunksize])

일꾼 프로세스에서 결과를 반환한 순서에 따라 결과가 반활될 수 있다는 것을 제외하고는 imap()과 같다.

p.map(func, iterable [, chunksize])

호출 가능한 객체 func을 iterable에 있는 모든 항목에 적용하고 결과를 리스트로 반환한다. 이 연산은 iterable을 여러 덩어리로 나누고 작업을 여러 일꾼 프로세스에 맡김으로써 병렬로 수행된다. chunksize는 각 덩어리에 들어갈 항목 수를 지정한

다. 데이터 양이 많은 경우 chunksize를 증가시키면 성능 향상을 기대할 수 있다.

p.map_async(func, iterable [, chunksize [, callback]])

결과가 비동기적으로 반환되는 것을 제외하고는 map과 같다. 나중에 결과를 얻을 때 사용하는 AsyncResult 인스턴스가 반환된다. callback은 호출 가능한 객체로 인수 하나를 받는다. 지정할 경우 결과를 사용할 수 있게 되었을 때 callback이 해당 결과를 가지고 호출된다.

p.terminate()

정리 작업을 수행하지 않고 아직 진행 중인 작업을 끝마치지 않은 채로 모든 일꾼 프로세스를 즉시 종료한다. p가 쓰레기 수집될 때 호출된다.

apply_async()와 map_async() 메서드는 결과로 AsyncResult 인스턴스를 반환한다. AsyncResult 인스턴스 a는 다음 메서드들을 가진다.

a.get([timeout])

결과를 반환한다. 필요에 따라 결과가 도착할 때까지 기다린다. timeout은 옵션인 타임아웃 시간이다. 결과가 주어진 시간 안에 도착하지 않으면 multiprocessing.TimeoutError 예외가 발생한다. 원격에서 연산을 수행하는 도중에 예외가 발생하면 이 메서드를 호출할 때 해당 예외가 다시 발생한다.

a.ready()

호출이 완료되면 True를 반환한다.

a.successful()

예외 없이 호출이 완료되면 True를 반환한다. 결과가 준비되기 전에 이 함수를 호출하면 AssertionError가 발생한다.

a.wait([timeout])

결과를 사용할 수 있을 때까지 기다린다. timeout은 옵션인 타임아웃 시간이다. 다음은 프로세스 풀을 사용해 특정 디렉터리에 있는 모든 파일에 대해 파일 이름을 SHA512 요약 값으로 매핑하는 사전을 구축하는 예이다.

```
import os
import multiprocessing
import hashlib

# 사용자가 조정할 수 있는 매개변수들
BUFSIZE = 8192    # 읽기 버퍼 크기
```

```

POOLSIZEx = 2    # 일꾼 수

def compute_digest(filename):
    try:
        f = open(filename,"rb")
    except IOError:
        return None
    digest = hashlib.sha512()
    while True:
        chunk = f.read(BUFSIZE)
        if not chunk: break
        digest.update(chunk)
    f.close()
    return filename, digest.digest()

def build_digest_map(topdir):
    digest_pool = multiprocessing.Pool(POOLSIZEx)
    allfiles = (os.path.join(path,name)
                for path, dirs, files in os.walk(topdir)
                for name in files)

    digest_map = dict(digest_pool.imap_unordered(
                      compute_digest,allfiles,20))
    digest_pool.close()
    return digest_map

# 돌려본다. 디렉터리 이름을 원하는 것으로 바꾸어보라.
if __name__ == '__main__':
    digest_map = build_digest_map(
        "/Users/beazley/Software/Python-3.0")
    print(len(digest_map))

```

이 예에서는 지정된 디렉터리 트리의 모든 파일에 대한 파일 이름들의 순서열을 생성하는 데 생성기 표현식을 사용하였다. 이 순서열은 그 다음에 imap_unordered() 함수를 통해 잘게 쪼개진 다음 프로세스 풀에 넘겨진다. 각 풀 일꾼은 compute_digest() 함수를 사용하여 자신에게 할당된 파일의 SHA512 요약 값을 계산한다. 계산된 결과들을 다시 모아서 파일 사전에 저장한다. 과학적인 결과라고 할 수는 없지만, 이 예를 필자의 듀얼 코어 맥북에서 돌렸을 때 단일 프로세스로 작성한 것보다 75% 정도 속도가 향상되었다.

풀 일꾼들의 추가 통신 오버헤드를 합리화할 수 있을 정도의 작업을 수행하는 경우에만 프로세스 풀을 사용하는 것이 적절하다. 보통 단순히 두 숫자를 더하는 것 같은 간단한 연산에 풀을 사용하는 것은 적절하지 않다.

공유 데이터와 동기화

일반적으로 프로세스들은 서로 완전히 격리되어 있으며 서로 통신할 수 있는 방법은 큐나 파일을 통한 방법밖에 없다. 하지만, 데이터를 공유하는 데 사용할 수 있는 두 종류의 객체가 있다. 이 객체들은 여러 프로세스에서 공유 데이터에 접근할 수 있도록 내부적으로 공유 메모리(shared memory)를 사용한다(mmap을 통해).

Value(typecode, arg1, ... argN, lock)

ctypes 객체를 공유 메모리에 생성한다. typecode는 array 모듈에서 사용하는 타입 코드('i', 'd' 등)를 담은 문자열이거나 ctypes 모듈에 있는 타입 객체(ctypes.c_int, ctypes.c_double 등)이다. 나머지 위치 인수들 arg1, arg2, ... argN은 주어진 타입의 생성자로 전달된다. lock은 키워드 전용 인수로서 True로 설정하면(기본 값) 값에 대한 접근을 막기 위해 새로운 락을 생성한다. Lock 또는 RLock 인스턴스 같은 기존 락을 지정하면 동기화를 수행하는 데 해당 락을 사용한다. v가 Value로 생성한 공유된 값 인스턴스인 경우 v.value로 내부 값에 접근할 수 있다. 예를 들어, v.value를 읽어 값을 얻을 수 있고 v.value에 값을 할당하여 값을 변경할 수 있다.

RawValue(typecode, arg1, ..., argN)

락을 걸지 않는다는 점을 제외하고 Value와 같다.

Array(typecode, initializer, lock)

ctypes 배열을 공유 메모리에 생성한다. typecode는 배열 내용을 설명하며 Value()에서 의미와 같다. initializer는 배열의 초기 크기를 지정하는 정수이거나, 내부 값들과 그 크기의 배열을 초기화하는 데 사용할 순서열일 수 있다. lock은 키워드 전용 인수로서 Value()에서 의미와 같다. a가 Array로 생성한 공유 배열 인스턴스일 경우 파이썬의 표준 색인, 분할, 반복 연산을 통해 내용에 접근할 수 있으며 각 연산은 락에 의해 동기화된다. 바이트 문자열일 경우에는 a.value 속성으로 전체 배열을 단일 문자열로 접근할 수 있다.

RawArray(typecode, initializer)

락을 걸지 않는 것을 제외하고 Array와 같다. 많은 수의 배열 항목들을 한꺼번에 다루어야 하는 프로그램을 작성한다면 이 타입을 동기화를 위한 별개의 락(필요할 경우)과 함께 사용하면 훨씬 나은 성능을 얻을 수 있다.

multiprocessing 모듈에는 Value()나 Array()로 생성할 수 있는 공유 값 이외에

도 다음에 나오는 동기화 기본 연산들의 공유된 버전을 제공한다.

기본 연산	설명
Lock	상호 배타적 락
RLock	재진입 상호 배타적 락(동일한 프로세스는 블로킹 없이 여러 번 락을 얻을 수 있다)
Semaphore	세마포어
BoundedSemaphore	경계 세마포어
Event	이벤트
Condition	조건 변수

이 객체들의 작동 방식은 `threading` 모듈에 같은 이름으로 정의되어 있는 동기화 기본 연산을 흉내낸다. 보다 자세한 내용은 `threading` 모듈을 참고하도록 한다.

일반적으로 다중 프로세스 프로그램을 작성할 때는 락, 세마포어, 또는 비슷한 연산을 사용하는 저수준 동기화에 관해서 스레드를 사용할 때와 같은 정도로 신경 쓸 필요는 없다. 파이프에 대한 `send()`과 `receive()` 연산과 큐에 대한 `put()`과 `get()` 연산은 부분적으로 이미 동기화를 지원한다. 그래도 특수한 경우에는 공유 값이나 락을 사용하는 것이 더 좋을 때도 있다. 다음은 파이썬으로 생성한 실수들의 리스트를 파이프 대신 공유 메모리를 사용하여 다른 프로세스로 전달하는 예를 보여준다.

```
import multiprocessing

class FloatChannel(object):
    def __init__(self, maxsize):
        self.buffer      = multiprocessing.RawArray('d', maxsize)
        self.buffer_len = multiprocessing.Value('i')
        self.empty      = multiprocessing.Semaphore(1)
        self.full       = multiprocessing.Semaphore(0)

    def send(self, values):
        self.empty.acquire()          # 버퍼가 비어야 진행된다.
        nitems = len(values)
        self.buffer_len.value = nitems # 버퍼 크기를 설정한다.
        self.buffer[:nitems] = values # 값을 버퍼로 복사한다.
        self.full.release()          # 버퍼가 가득 차를 알린다.

    def recv(self):
        self.full.acquire()          # 버퍼가 가득 차야 진행된다.
        values = self.buffer[:self.buffer_len.value] # 값을 복사한다.
        self.empty.release()         # 버퍼가 비었음을 알린다.
        return values

    # 성능 테스트. 다양한 메시지를 받는다.
    def consume_test(count, ch):
        for i in xrange(count):
            values = ch.recv()
```

```
# 성능 테스트. 다량의 메시지를 보낸다.
def produce_test(count, values, ch):
    for i in xrange(count):
        ch.send(values)

if __name__ == '__main__':
    ch = FloatChannel(100000)
    p = multiprocessing.Process(target=consume_test,
                                args=(1000, ch))
    p.start()
    values = [float(x) for x in xrange(100000)]
    produce_test(1000, values, ch)
    print("Done")
    p.join()
```

이 예를 이해하는 일은 독자들에게 맡긴다. 필자의 컴퓨터에서 성능 테스트를 해보니 대규모 실수 리스트를 `FloatChannel`을 통해 보내는 것이 파이프(모든 값을 피클링하거나 역피클링한다)를 통해 보내는 것보다 80% 정도 빨랐다.

관리 객체

스레드와 달리 프로세스는 공유 객체를 지원하지 않는다. 앞 절에서 보았듯이 공유 값과 배열을 생성할 수는 있지만 사전, 리스트, 사용자 정의 클래스 인스턴스 등 더 고급 파이썬 객체가 지원되지는 않는다. `multiprocessing` 모듈은 이른바 관리자(manager)의 제어 아래 공유 객체로 작업할 수 있는 방법을 제공한다. 관리자란 실제 객체가 존재하는 서버로 작동하는 별개의 하위 프로세스이다. 다른 프로세스는 관리자 서버의 클라이언트로 작동하는 대리자를 사용해 공유 객체에 접근한다.

간단한 관리 객체로 작업을 할 때는 `Manager()` 함수를 사용하는 방법이 가장 쉽다.

`Manager()`

별개 프로세스로 작동하는 관리자 서버를 생성한다. `multiprocessing.managers` 하위 모듈에 정의된 `SynManager` 타입의 인스턴스를 반환한다.

`Manager()`가 반환하는 `SyncManager` 인스턴스 `m`은 공유 객체를 생성하고 공유 객체에 접근하는 데 사용하는 대리자를 반환하는 메서드들을 가진다. 보통 새 프로세스를 생성하기 전에 먼저 관리자를 생성하고, 다음에 나오는 메서드들을 사용해 공유 객체를 생성한다.

m.Array(typecode, sequence)

서버에 공유 Array 인스턴스를 생성하고 이에 대한 대리자를 반환한다. 인수에 대한 설명은 ‘공유 데이터와 동기화’ 절을 참고한다.

m.BoundedSemaphore([value])

서버에 공유 threading.BoundedSemaphore 인스턴스를 생성하고 이에 대한 대리자를 반환한다.

m.Condition([lock])

서버에 공유 threading.Condition 인스턴스를 생성하고 이에 대한 대리자를 반환한다. lock은 m.Lock() 또는 m.Rlock()으로 생성된 대리자 인스턴스이다.

m.dict([args])

서버에 공유 dict 인스턴스를 생성하고 이에 대한 대리자를 반환한다. 이 메서드의 인수들은 내장 dict() 함수의 것과 같다.

m.Event()

서버에 공유 threading.Event 인스턴스를 생성하고 이에 대한 대리자를 반환한다.

m.list([sequence])

서버에 공유 list 인스턴스를 생성하고 이에 대한 대리자를 반환한다. 이 메서드의 인수들은 내장 list() 함수의 것과 같다.

m.Lock()

서버에 공유 threading.Lock 인스턴스를 생성하고 이에 대한 대리자를 반환한다.

m.Namespace()

서버에 공유 네임스페이스 객체를 생성하고 이에 대한 대리자를 반환한다. 네임스페이스(namespace)는 파이썬 모듈과 어느 정도 비슷한 객체이다. 예를 들어, n이 네임스페이스 대리자라면 n.name = value 또는 value = n.name 같이 점(.)을 사용하여 속성을 읽거나 할당할 수 있다. 여기서 name의 형태가 중요하다. name이 문자로 시작하면 해당 값은 관리자가 관리하고 모든 프로세스에서 접근할 수 있는 공유 객체의 일부가 된다. name이 밑줄로 시작하면 대리자 객체의 일부가 되고 공유되지 않는다.

m.Queue()

서버에 공유 Queue.Queue 객체를 생성하고 이에 대한 대리자를 반환한다.

m.RLock()

서버에 공유 threading.Lock 객체를 생성하고 이에 대한 대리자를 반환한다.

m.Semaphore([value])

서버에 공유 threading.Semaphore 객체를 생성하고 이에 대한 대리자를 반환한다.

m.Value(typecode, value)

서버에 공유 Value 객체를 생성하고 이에 대한 프록시를 반환한다. 인수에 대한 설명은 ‘공유 데이터와 동기화’ 절을 참고한다.

다음 예는 관리자를 사용해 프로세스 사이에 공유되는 사전을 생성하는 방법을 보여준다.

```
import multiprocessing
import time

# 전달된 이벤트에 알림이 올 때마다 d를 출력한다.
def watch(d, evt):
    while True:
        evt.wait()
        print(d)
        evt.clear()

if __name__ == '__main__':
    m = multiprocessing.Manager()
    d = m.dict()      # 공유 dict를 생성한다.
    evt = m.Event()   # 공유 Event를 생성한다.

    # 사전을 감시하는 프로세스를 구동한다.
    p = multiprocessing.Process(target=watch, args=(d,evt))
    p.daemon=True
    p.start()

    # 사전을 갱신하고 감시자에게 알린다.
    d['foo'] = 42
    evt.set()
    time.sleep(5)

    # 사전을 갱신하고 감시자에게 알린다.
    d['bar'] = 37
    evt.set()
    time.sleep(5)

    # 프로세스와 관리자를 종료한다.
```

```
p.terminate()
m.shutdown()
```

이 예를 실행하면 전달된 이벤트가 설정될 때마다 watch() 함수에서 d의 값을 출력한다. 주 프로그램에서 공유 사전과 이벤트를 생성하고 조작한다. 이 코드를 실행하면 자식 프로세스에서 데이터를 출력하는 것을 볼 수 있다.

사용자 정의 클래스 인스턴스와 같은 타입의 공유 객체를 원할 경우에는 커스텀 관리자 객체를 작성해야 한다. 이를 위해 하위 모듈 multiprocessing.managers에 정의된 BaseManager에서 상속한 클래스를 생성하면 된다.

```
managers.BaseManager([address [, authkey]])
```

사용자 정의 객체를 위한 커스텀 관리자 서버를 작성하는 데 사용하는 기본 클래스. address는 옵션인 튜플 (hostname, port)로 서버의 네트워크 주소를 지정한다. 이 값을 생략하면 운영체제에서 간단히 빈 포트 번호에 대응하는 주소를 할당한다. authkey는 서버에 연결하는 클라이언트를 인증하는 데 사용할 문자열이다. 이 값을 생략하면 current_process().authkey 값을 사용한다.

mgrclass가 BaseManager에서 상속한 클래스일 때 공유 객체에 대한 대리자를 반환하는 메서드를 생성하는 데 다음 클래스 메서드를 사용한다.

```
mgrclass.register(typeid [, callable [, proxytype [, exposed [, method_to_typeid [,create_method]]]]])
```

관리자 클래스에 새로운 데이터 타입을 등록한다. typeid는 특정한 종류의 공유 객체를 부르는 데 사용할 이름을 담은 문자열이다. 이 문자열은 유효한 파이썬 식별자이어야 한다. callable은 공유될 인스턴스를 생성 또는 반환하는 호출 가능한 객체이다. proxytype은 클라이언트에서 사용될 대리자 객체들의 구현을 제공하는 클래스이다. 일반적으로 기본으로 이러한 클래스들이 생성되기 때문에 이 값은 보통 None으로 설정한다. exposed는 대리자 객체에 공개할 공유 객체의 메서드 이름들을 담은 순서열이다. 생략하면 proxytype._exposed_ 값이 사용되며 이 값 또한 정의되어 있지 않으면 모든 공개 메서드(밑줄로 시작하지 않는 모든 호출 가능한 메서드)가 대리자 객체에 공개된다. method_to_typeid는 어떤 메서드들이 대리자 객체를 사용해 결과를 반환하는지를 지정하는 사전으로 메서드 이름을 타입 ID로 매핑한다. 이 매핑에 메서드 이름이 없으면 반환 값은 복사된 후 반환된다. method_to_typeid가 None이면 proxytype._method_to_typeid_가 정의될 경우 사용된다. create_method는 mgrclass에 typeid이라는 이름의 메서드를 생성할지를 지

정하는 불리언 플래그이다. 기본 값은 True이다.

BaseManager에서 상속한 관리자 인스턴스 m은 반드시 직접 작동시켜야 한다. 이를 위해 다음 속성과 메서드들을 사용한다.

m.address

관리자 서버가 사용하는 주소를 담은 튜플 (hostname, port).

m.connect()

BaseManager 생성자에서 주어진 주소에 있는 원격 관리자 객체에 연결한다.

m.serve_forever()

현재 프로세스에서 관리자 서버를 실행한다.

m.shutdown()

m.start() 메서드로 구동된 관리자 서버를 종료한다.

m.start()

별개의 하위 프로세서를 시작시키고 그 프로세스 안에서 관리자 서버를 구동한다. 다음 예는 사용자 정의 클래스를 위한 관리자를 생성하는 방법을 보여준다.

```
import multiprocessing
from multiprocessing.managers import BaseManager

class A(object):
    def __init__(self,value):
        self.x = value
    def __repr__(self):
        return "A(%s)" % self.x
    def getX(self):
        return self.x
    def setX(self,value):
        self.x = value
    def __iadd__(self,value):
        self.x += value
        return self

class MyManager(BaseManager): pass
MyManager.register("A",A)

if __name__ == '__main__':
    m = MyManager()
    m.start()
    # 관리 객체를 생성한다.
    a = m.A(37)
    ...
    
```

이 예에서 마지막 문장은 관리자 서버에서 인스턴스 A를 생성한다. 여기서 변수 a는 이 인스턴스에 대한 대리자일 뿐이다. 이 대리자는 서버에 있는 참조 대상 (referent)과 비슷하게 작동한다(완전히 같지는 않다). 일단 데이터 속성과 프로퍼티에 접근할 수 없다. 대신 접근 함수를 사용해야 한다.

```
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AutoProxy[A]' object has no attribute 'x'
>>> a.getX()
37
>>> a.setX(42)
>>>
```

대리자에 대해 repr() 함수는 대리자를 나타내는 문자열을 반환하는 반면 str()은 참조 대상에 대한 __repr__()의 출력을 반환한다. 다음 예를 보자.

```
>>> a
<AutoProxy[A] object, typeid 'A' at 0xcef230>
>>> print(a)
A(37)
>>>
```

특수 메서드나 밑줄로 시작하는 메서드에는 대리자를 통해 접근할 수 없다. 예를 들어, a.__iadd__()를 호출하려고 시도하면 제대로 작동하지 않는다.

```
>>> a += 37
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +=: 'AutoProxy[A]' and 'int'
>>> a.__iadd__(37)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AutoProxy[A]' object has no attribute '__iadd__'
>>>
```

고급 응용에서는 보다 더 세밀한 접근 제어를 위해 대리자를 커스터마이즈할 수 있다. multiprocessing.managers에 정의된 BaseProxy를 상속한 클래스를 정의하면 된다. 다음 코드는 앞에 나온 A 클래스에 대한 커스텀 대리자를 생성하는 방법을 보여준다. 이 대리자는 __iadd__() 메서드를 공개하고 프로퍼티를 사용해 속성 x를 공개한다.

```
from multiprocessing.managers import BaseProxy

class AProxy(BaseProxy):
    # 공개할 참조 대상의 모든 메서드 목록
```

```

_exposed_ = ['__iadd__', 'getX', 'setX']
# 대리자의 공개 인터페이스를 구현한다.
def __iadd__(self, value):
    self._callmethod('__iadd__', (value,))
    return self
@property
def x(self):
    return self._callmethod('getX', ())
@x.setter
def x(self, value):
    self._callmethod('setX', (value,))

class MyManager(BaseManager): pass
MyManager.register("A", A, proxytype=AProxy)

```

BaseProxy를 상속한 인스턴스 proxy는 다음 메서드들을 가진다.

proxy._callmethod(name [, args [, kwargs]])

대리자의 참조 대상 객체에 name 메서드를 호출한다. name은 메서드 이름을 담은 문자열이고 args는 위치 인수들을 담은 튜플이며 kwargs는 키워드 인수들을 담은 사전이다. name 메서드는 반드시 직접 공개된 것이어야 한다. 보통 대리자 클래스의 클래스 속성 _exposed_에 이름을 넣는 것으로 메서드를 공개한다.

proxy._getvalue()

호출자 쪽에 참조 대상의 복사본을 반환한다. 다른 프로세스에서 호출될 경우 참조 대상 객체가 피클링되어 호출자로 보내지고 다시 역피클링된다. 참조 대상이 피클링될 수 없으면 예외가 발생한다.

연결

multiprocessing 모듈을 사용하는 프로그램에서는 같은 머신 또는 원격에서 실행되는 프로세스와 메시지를 주고받을 수 있다. 그렇기 때문에 단일 시스템에서 작동하도록 작성한 프로그램을 컴퓨팅 클러스터에서 작동하도록 확장하는 일이 쉽다. 하위 모듈 multiprocessing.connection에는 이런 일을 수행하기 위한 함수와 클래스들이 있다.

connections.Client(address [, family [, authenticate [, authkey]]])

주소 address에 미리 연결을 기다리고 있어야 하는 다른 프로세스와 연결을 맺는다. address는 네트워크 주소를 나타내는 튜플(hostname, port)이거나 유닉스 도메인 소켓을 나타내는 파일 이름이거나 원격 시스템 servername(지역 머

신은 servername에 ‘.’를 사용)에 대한 이름 있는 윈도 파이프를 나타내는 r’ \\servername\pipe\pipename’ 형태의 문자열일 수 있다. family는 주소 형식을 나타내는 문자열이며 보통 ‘AF_INET’, ‘AF_UNIX’, ‘AF_PIPE’ 중 하나이다. 생략하면 family는 address의 형식을 보고 추론하여 정해진다. authentication는 요약 인증을 사용할지를 지정하는 불리언 플래그이다. authkey는 인증 키를 담은 문자열이다. 생략하면 current_process(),authkey 값을 사용한다. 이 함수는 ‘프로세스 사이 통신’의 Pipe에서 설명한 Connection 객체를 반환한다.

```
connections.Listener([address [, family [, backlog [, authenticate [, authkey]]]]])
```

Client() 함수로 생성되는 연결을 기다리고 처리하는 서버를 구현한 클래스. address, family, authenticate, authkey 인수는 Client()에서 의미와 같다. backlog는 address 매개변수가 네트워크 연결을 나타내는 경우 소켓의 listen() 메서드에 전달 할 값을 나타내는 정수이다. backlog의 기본 값은 1이다. address를 생략하면 기본 주소가 사용된다. address와 family 두 값 모두 생략하면 지역 시스템에서 사용할 수 있는 것中最 가장 빠른 통신 방법을 선택한다.

Listener 인스턴스 s는 다음 메서드와 속성을 가진다.

s.accept()

새로운 연결을 수락하고 Connection 객체를 반환한다. 인증에 실패한 경우 AuthenticationError 예외가 발생한다.

s.address

리스너가 사용 중인 주소.

s.close()

리스너가 사용 중인 파이프나 소켓을 닫는다.

s.last_accepted

수락된 가장 최근 클라이언트 주소.

다음은 클라이언트를 기다리는 간단한 원격 연산(덧셈)을 구현한 서버 프로그램의 예이다.

```
from multiprocessing.connection import Listener  
  
serv = Listener(“,15000),authkey=‘12345’  
while True:
```

```

conn = serv.accept()
while True:
    try:
        x,y = conn.recv()
    except EOFError:
        break
    result = x + y
    conn.send(result)
conn.close()

```

다음은 이 서버에 연결하고 메시지를 몇 개 보내는 간단한 클라이언트 프로그램이다.

```

from multiprocessing.connection import Client
conn = Client(('localhost',15000), authkey="12345")

conn.send((3,4))
r = conn.recv()
print(r)      # '7'을 출력

conn.send(("Hello","World"))
r = conn.recv()

print(r)      # 'HelloWorld' 출력
conn.close()

```

기타 유ти리티 함수

이 모듈에는 다음 유ти리티 함수들도 정의되어 있다.

active_children()

모든 활성 자식 프로세스를 나타내는 Process 객체들의 리스트를 반환한다.

cpu_count()

알 수 있다면 시스템의 CPU 수를 반환한다.

current_process()

현재 프로세스를 나타내는 Process 객체를 반환한다.

freeze_support()

py2exe 같은 다양한 패키징 도구를 사용해 “동결(frozen)” 시킬 응용 프로그램에서 주 프로그램의 첫 문장에 포함해야 하는 함수. 동결 응용 프로그램에서 하위 프로세스를 구동할 때 발생할 수 있는 런타임 에러를 방지하는 데 필요하다.

get_logger()

multiprocessing 모듈과 관련된 로깅 객체를 반환한다. 없으면 생성한다. 반환된

로거는 루트 로거로 메시지를 전달하지 않고 logging.NOTSET 수준을 가지며 모든 로깅 메시지를 표준 에러로 출력한다.

set_executable(executable)

하위 프로세스를 실행하는 데 사용할 파이썬 실행 파일의 이름을 설정한다. 윈도에서만 있다.

다중 프로세스 프로그래밍에 관한 일반적인 조언

multiprocessing 모듈은 파이썬 라이브러리에서 가장 강력한 기능을 제공하는 고급 모듈 중 하나다. 다음은 머리가 복잡해지는 것을 방지하는 데 도움이 될 만한 몇 가지 팁을 나열한 것이다.

- 큰 규모의 응용 프로그램을 작성하기 전에 온라인 문서를 주의 깊게 읽는다. 이 책에서 필수적인 기본 기능들을 다루기는 했지만 공식 문서에서는 발생할 수 있는 예상하지 못한 문제들을 다루고 있다.
- 프로세스 사이에 전달할 모든 데이터는 피클링할 수 있어야 한다는 점을 유념 하라.
- 공유 데이터보다는 메시지 전달과 큐를 사랑하는 법을 배워라. 메시지 전달을 사용하면 동기화나 락, 그리고 기타 이슈에 관해 걱정하지 않아도 된다. 프로세스 수가 늘어남에 따라 더 나은 확장성을 제공하기도 한다.
- 별개의 프로세스에서 실행할 함수 안에서 전역 변수를 사용하지 않도록 한다. 대신에 직접 매개변수를 전달하는 것이 낫다.
- 직장에서 잘리고 싶은 것이 아니라면 프로그램 안에서 스레드와 다중 프로세스 프로그래밍을 섞는 시도를 하지 않는다(누가 코드 검토를 하느냐에 따라 잘릴 가능성이 높아질 수 있다).
- 프로세스를 종료하는 방법에 주의를 기울여라. 일반적으로 단순히 쓰레기 수집에 의존하거나 또는 terminate()를 호출하여 프로세스를 강제로 종료하는 것 보다 직접 프로세스를 닫고 잘 정의된 종료 방법 체계를 마련하는 것이 낫다.
- 관리자와 대리자의 사용은 분산 컴퓨팅의 여러 개념(예를 들어, 분산 객체)과 밀접한 관련이 있다. 괜찮은 분산 컴퓨팅 책이 있으면 도움이 될 것이다.
- multiprocessing 모듈은 pyprocessing이라고 부르는 써드 파티 라이브러리에서 유래되었다. 이 라이브러리와 관련한 사용 팁이나 정보를 검색하면 도움이

될 것이다.

- 이 모듈은 윈도에서도 작동하지만 여러 미묘한 사항에 관한 정보를 얻으려면 공식 문서를 주의 깊게 읽어보도록 한다. 예를 들어, 윈도에서 새로운 프로세스를 구동하기 위해 multiprocessing 모듈은 나름대로 유닉스 fork() 연산을 구현하여 프로세스의 상태 정보를 파일을 통해 자식 프로세스로 복사한다. 일반적으로는 이 모듈은 유닉스 시스템에 더 맞게 튜닝되어 있다.
- 그 밖에 가급적 모든 것을 단순하게 만들어라.

threading

threading 모듈은 Thread 클래스와 다중 스레드 프로그램 작성을 위한 다양한 동기화 기능을 제공한다.

Thread 객체

Thread 클래스는 별개의 제어 스레드를 나타내는 데 사용된다. 새로운 스레드는 다음 클래스로 생성한다.

Thread(group=None, target=None, name=None, args=(), kwargs={})

새로운 Thread 인스턴스를 생성한다. group은 None이며 차후 확장을 위해 존재한다. target은 스레드가 시작할 때 run() 메서드에서 호출하는 호출 가능한 객체이다. 기본 값은 None이고 아무것도 호출하지 않는다. name은 스레드 이름이다. 기본으로 “Thread-N” 형식의 고유 이름이 사용된다. args는 target 함수로 전달할 인수들을 담은 튜플이다. kwargs는 target에 전달할 키워드 인수들을 담은 사전이다. Thread 인스턴스 t는 다음 메서드와 속성들을 지원한다.

t.start()

별개의 제어 스레드 안에서 run() 메서드를 호출하여 스레드를 시작한다. 이 메서드는 한 번만 호출할 수 있다.

t.run()

이 메서드는 스레드가 시작할 때 호출된다. 기본으로 생성자에 전달된 target 함수를 호출한다. 이 메서드는 Thread의 하위 클래스에서 재정의할 수 있다.

t.join([timeout])

스레드가 종료되거나 타임아웃이 발생할 때까지 기다린다. timeout은 타임아웃을 초로 지정한 부동 소수점 수이다. 스레드는 자신에게 참여할 수 없으며 아직 시작하지 않은 스레드에 참여하는 것은 예리이다.

t.is_alive()

스레드가 살아 있으면 True를, 그렇지 않으면 False를 반환한다. 스레드는 start() 메서드가 반환할 때부터 run() 메서드가 종료할 때까지 살아 있다. 이 메서드의 오래된 코드에서의 이름은 t.isAlive()이다.

t.name

스레드 이름. 단순히 식별을 위해서만 사용하는 문자열이며 원하면 의미 있는 값으로 변경할 수 있다(디버깅을 수월하게 하기 위해). 오래된 코드에서는 스레드 이름을 조작하는 데 t.getName()과 t.setName(name)을 사용한다.

t.ident

정수 스레드 식별자. 스레드가 아직 시작하지 않았으면 None이다.

t.daemon

불리언인 스레드 데몬 플래그. 반드시 start()를 호출하기 전에 설정해야 하며 초기 값은 이 스레드를 생성한 스레드의 데몬 상태를 상속한다. 데몬 아닌 활성 스레드가 하나도 없을 때 전체 파이썬 프로그램은 종료된다. 모든 프로그램은 데몬이 아닌 초기 제어 스레드를 나타내는 주 스레드를 가진다. 오래된 코드에서는 이 값을 조작하는 데 t.setDaemon(flag)과 t.isDaemon()를 사용한다.

다음은 함수(또는 기타 호출 가능한 객체)를 스레드로 생성하여 구동하는 방법을 보여준다.

```
import threading
import time

def clock(interval):
    while True:
        print("The time is %s" % time.ctime())
        time.sleep(interval)

t = threading.Thread(target=clock, args=(15,))
t.daemon = True
t.start()
```

다음은 동일한 스레드를 클래스로 정의하는 방법을 보여준다.

```
import threading
import time

class ClockThread(threading.Thread):
    def __init__(self,interval):
        threading.Thread.__init__(self)
        self.daemon = True
        self.interval = interval
    def run(self):
        while True:
            print("The time is %s" % time.ctime())
            time.sleep(self.interval)

t = ClockProcess(15)
t.start()
```

스레드를 클래스로 정의하고 여러분만의 `__init__()` 메서드를 정의할 때는 앞에서 보듯이 기본 클래스 생성자 `Thread.__init__()`를 호출하는 것이 매우 중요하다. 그렇지 않으면 아주 까다로운 상황을 맞이할 수 있다. `run()` 말고는 스레드에 정의된 다른 메서드를 재정의하면 안 된다.

백그라운드에서 영원히 실행되는 스레드에는 이 예에서 보듯이 보통 `daemon` 속성을 설정한다. 보통 파이썬에서는 인터프리터를 종료하기 전에 모든 스레드가 종료되기를 기다린다. 하지만, 계속 실행되는 백그라운드 작업에 대해서는 이렇게 하면 안 된다. `daemon` 플래그를 설정하면 주 프로그램이 종료하는 즉시 인터프리터가 종료된다. 이때 데몬 스레드가 파괴된다.

Timer 객체

Timer 객체는 나중에 어떤 시점에 함수를 실행하는 데 사용한다.

Timer(interval, func [, args [, kwargs]])

`interval` 초기 지난 후에 `func` 함수를 실행하는 타이머 객체를 생성한다. `args`과 `kwargs`는 `func`에 전달할 인수와 키워드 인수를 제공한다. 타이머는 `start()` 메서드가 호출되기 전까지 시작하지 않는다.

Timer 객체 `t`는 다음 메서드들을 가진다.

t.start()

타이머를 시작한다. 지정된 시간이 지난 후 `Timer()`에 지정한 `func`이 실행된다.

t.cancel()

함수가 아직 실행되지 않았으면 타이머를 취소한다.

Lock 객체

기본 락(primitive lock, 또는 상호 배타적 락)은 “잠긴(locked)” 또는 “열린(unlocked)” 상태가 되는 동기화 기본 연산이다. 두 메서드 acquire()와 release()를 락 상태를 변경시키는 데 사용한다. 잠긴 상태이면 락을 얻는 시도는 락이 열릴 때까지 막힌다. 하나 이상의 스레드가 락을 얻기 위해 기다리고 있으면 락이 열릴 때 그중 하나만 락을 얻어 진행할 수 있다. 기다리고 있는 스레드 중 어느 스레드가 락을 얻을 것인지는 정의되지 않는다.

다음 생성자로 새로운 Lock 인스턴스를 생성한다.

lock()

초기에 락이 열린 새로운 Lock 객체를 생성한다.

Lock 인스턴스 lock은 다음 메서드들을 지원한다.

lock.acquire([blocking])

락을 획득하며 필요한 경우 락이 열릴 때까지 멈추어 있다. blocking을 False로 설정하면 이 함수는 락을 얻지 못한 경우 False를 즉시 반환하고 락을 얻은 경우 True를 즉시 반환한다.

lock.release()

락을 해제한다. 락이 해제된 상태에 있거나 원래 acquire()를 호출한 스레드가 아닌 다른 스레드가 이 메서드를 호출하면 에러가 발생한다.

RLock

재진입 락(reentrant lock)은 동기화 기본 연산으로서 Lock 객체와 비슷하지만 같은 스레드가 락을 여러 번 얻을 수 있게 한다. 이에 따라 락을 소유하고 있는 스레드에서 acquire()와 release() 연산을 여러 번 수행할 수 있다. 이 경우 가장 바깥쪽 release() 연산만이 락을 해제된 상태로 만들 수 있다.

다음 생성자로 새로운 RLock 인스턴스를 생성한다.

RLock()

새로운 재진입 락 객체를 생성한다. RLock 객체 rlock은 다음 메서드들을 지원한다.

rlock.acquire([blocking])

락을 획득하고 필요할 경우 락이 해제될 때까지 멈춘다. 락을 소유한 스레드가 없으면 락을 잠근 다음 재귀 수준을 1로 설정한다. 스레드가 이미 락을 소유하고 있으면 재귀 수준을 1 증가시키며 즉시 반환한다.

rlock.release()

재귀 수준을 감소시켜 락을 해제한다. 감소 후 재귀 수준이 0이면 락은 해제된 상태로 되돌아간다. 그렇지 않으면 락은 잠긴 상태로 남는다. 현재 락을 소유한 스레드에서만 호출해야 한다.

세마포어와 경계 세마포어

세마포어(semaphore)는 각 acquire() 호출마다 감소하고 각 release() 호출마다 증가하는 카운터에 기반한 동기화 기본 연산이다. 카운터가 0에 도달하면 acquire() 메서드는 다른 스레드가 release() 호출하기 전까지 멈춘다.

Semaphore([value])

새로운 세마포어를 생성한다. value는 카운터의 초기 값이다. 생략하면 카운터는 1로 설정된다.

Semaphore 인스턴스 s는 다음 메서드들을 지원한다.

s.acquire([blocking])

세마포어를 획득한다. 내부 카운터가 0보다 크면 이 메서드는 카운터를 1 감소시키고 즉시 반환한다. 만약 카운터가 0이면 이 메서드는 다른 스레드가 release()를 호출할 때까지 멈춘다. blocking 인수는 Lock과 RLock 객체에서 설명한 것과 같다.

s.release()

내부 카운터를 1 증가시켜 세마포어를 해제한다. 카운터가 0이고 다른 스레드가 기다리고 있으면 그 스레드를 깨운다. 여러 스레드가 기다리고 있으면 그 중 하나만 acquire() 호출에서 깨어난다. 스레드를 깨우는 순서는 정해져 있지 않다.

BoundedSemaphore([value])

새로운 세마포어를 생성한다. value는 카운터의 초기 값이다. 생략하면 카운터는 1로 설정된다. BoundedSemaphore는 release() 연산의 수가 acquire()의 연산 수를 넘을 수 없다는 점을 제외하고는 Semaphore와 똑같이 작동한다.

세마포어는 락과는 달리 신호를 주고받는 데 사용할 수 있다. 예를 들어, 생산자와 소비자 스레드 사이에 통신을 위해 서로 다른 스레드에서 acquire()와 release() 메서드를 호출할 수 있다.

```
produced = threading.Semaphore(0)
consumed = threading.Semaphore(1)

def producer():
    while True:
        consumed.acquire()
        produce_item()
        produced.release()

def consumer():
    while True:
        produced.acquire()
        item = get_item()
        consumed.release()
```

이 예를 통해 살펴본 종류의 신호를 보낼 때는 나중에 설명할 조건 변수를 흔히 대신 사용한다.

이벤트

이벤트(event)는 스레드 사이에 통신을 하는 데 사용한다. 한 스레드에서 “이벤트”를 알리고 하나 이상의 다른 스레드에서 이것을 기다린다. Event 인스턴스는 set() 메서드로 참으로 설정하고 clear() 메서드로 거짓으로 설정할 수 있는 내부 플래그를 관리한다. wait() 메서드는 플래그가 참이 될 때까지 기다린다.

Event()

초기 플래그를 거짓으로 설정한 새로운 Event 인스턴스를 생성한다. Event 인스턴스 e는 다음 메서드들을 지원한다.

e.is_set()

내부 플래그가 참일 경우에만 True를 반환한다. 오래된 코드에서 이 메서드의 이름은 `isSet()`이다.

e.set()

초기 플래그를 참으로 설정한다. 플래그가 참이 되기를 기다리던 모든 스레드를 깨운다.

e.clear()

초기 플래그를 거짓으로 되돌린다.

e.wait([timeout])

내부 플래그가 참이 될 때까지 기다린다. 내부 플래그가 참이면 이 메서드는 즉시 반환한다. 그렇지 않으면 다른 스레드에서 set()을 호출하여 플래그를 참으로 설정하거나 옵션인 타임아웃이 발생하기 전까지 기다린다. timeout은 타임아웃 시간을 초로 나타낸 부동 소수점 수이다.

Event 객체를 다른 스레드에 신호를 보내는 데 사용할 수 있지만 생산자 소비자 문제에서 전형적으로 사용하는 종류의 알림 기능을 구현하는 데는 사용하지 않아야 한다. 예를 들어, 다음과 같은 코드는 작성하지 않아야 한다.

```
evt = Event()

def producer():
    while True:
        # 항목을 생성한다.
        ...
        evt.signal()

def consumer():
    while True:
        # 항목을 기다린다.
        evt.wait()
        # 항목을 소비한다.
        ...
        # 이벤트를 재설정하고 다시 기다린다.
        evt.clear()
```

이 코드는 evt.wait()와 evt.clear() 연산 사이에 생산자가 새로운 항목을 생성할 수 있기 때문에 신뢰성이 없다. 새로운 항목이 생성되었지만 이벤트를 재설정하는 바람에 생산자가 새로운 항목을 다시 생성할 때까지 앞의 항목은 소비자에게 보이지 않게 된다. 운이 좋아봤자 이 프로그램에는 알 수 없이 항목 처리가 늦어지는 현상이 일시적으로 나타나는 데 그친다. 최악의 경우에는 이벤트 알림의 분실로 인해 전체 프로그램이 멈출 수 있다. 이런 종류의 문제에 대해서는 조건 변수를 사용하는 것이 낫다.

조건 변수

조건 변수(condition variable)는 락을 기반으로 만든 것으로서 스레드에서 특정 상태 변화나 이벤트 발생을 알기 위해 사용하는 동기화 기본 연산이다. 보통 한 스레드에서 데이터를 생성하고 다른 한 스레드에서 이를 소비하는 생성자 소비자 문제를 푸는 데 사용한다. 새로운 Condition 인스턴스는 다음 생성자로 생성한다.

Condition([lock])

새로운 조건 변수를 생성한다. lock은 옵션인 Lock 또는 RLock 인스턴스이다. 제공하지 않으면 새로운 RLock 인스턴스를 생성해 조건 변수에 사용한다.

조건 변수 cv는 다음 메서드들을 지원한다.

cv.acquire(*args)

내부 락을 획득한다. 이 메서드는 대응하는 내부 락의 acquire(*args) 메서드를 호출하고 그 결과를 반환한다.

cv.release()

락을 해제한다. 이 메서드는 대응하는 내부 락의 release() 메서드를 호출한다.

cv.wait([timeout])

타임아웃이 발생하거나 알리기 전까지 기다린다. 이 메서드는 호출 스레드에서 락을 이미 획득한 상태에서 호출한다. 호출하면 내부 락이 해제되고 호출 스레드는 다른 스레드에서 notify()나 notifyAll()를 조건 변수에 대해 호출할 때까지 잠든다. 일단 다시 깨면 호출 스레드는 락을 다시 얻고 이 메서드가 반환된다. timeout은 타임아웃을 초로 나타낸 부동 소수점 수이다. 시간이 다 되면 호출 스레드는 깨어나고 락을 다시 얻어 제어권을 가진다.

cv.notify([n])

조건 변수를 기다리는 하나 또는 그 이상의 스레드를 깨운다. 이 메서드는 호출 스레드에서 락을 이미 획득한 상태에서 호출하고 기다리는 스레드가 없으면 아무 일도 하지 않는다. n은 깨울 스레드 수를 지정하며 기본 값은 1이다. 깨어난 스레드가 락을 다시 얻을 때까지 wait() 호출은 반환되지 않는다.

cv.notify_all()

이 조건을 기다리고 있는 모든 스레드를 깨운다. 오래된 코드에서 이 메서드는 notifyAll()이었다.

다음 예는 조건 변수를 사용할 때 활용할 수 있는 틀을 제공한다.

```
cv = threading.Condition()
def producer():
    while True:
        cv.acquire()
        produce_item()
        cv.notify()
        cv.release()

def consumer():
    while True:
        cv.acquire()
        while not item_is_available():
            cv.wait() # 항목이 나타날 때까지 기다린다.
        cv.release()
        consume_item()
```

조건 변수를 사용할 때 미묘한 부분은 조건을 기다리는 여러 스레드가 있는 경우 notify() 연산이 하나 또는 그 이상의 스레드를 깨울 수 있다는 점이다(이 작동 방식은 운영체제에 따라 다를 수 있다). 이 때문에 어떤 스레드가 깨어났지만 원하는 조건이 만족되지 않는 것을 발견하게 될 수 있다. 바로 앞의 코드에서 consumer() 함수 안에서 while 구문을 사용한 이유이다. 스레드가 깨어났지만 생성된 항목이 이미 사라졌으므로 단순히 다음 신호를 기다리며 되돌아간다.

락 다루기

Lock, RLock, 세마포어 같은 락 연산을 사용할 때는 주의를 기울여야 한다. 데드락이나 경쟁 조건 발생의 주 요인이 락을 잘못 관리하는 것이기 때문이다. 락에 의존하는 코드에서는 예외가 발생했을 때 항상 락을 적절히 해제해야 한다. 보통 다음과 같이 코드를 작성한다.

```
try:
    lock.acquire()
    # 임계 구역
    문장들
    ...
finally:
    lock.release()
```

아니면 모든 락이 지원하는 컨텍스트 관리 프로토콜을 사용하면 코드를 더 깔끔하게 작성할 수 있다.

```
with lock:
    # 임계 구역
```

문장들

...

여기서 `with`문은 자동으로 락을 얻고 제어 흐름이 컨텍스트를 벗어나면 락을 해제한다.

특정 시점에 하나 이상의 락을 획득하는 코드를 작성하는 것도 피해야 한다. 다음 예를 보자.

```
with lock_A:
    # 임계 구역
    문장들
    ...
    with lock_B:
        # B에 대한 임계 구역
        문장들
    ...
    ...
```

이렇게 하면 응용 프로그램에서 알 수 없는 데드락이 발생하기 딱 좋다. 이런 상황을 피하는 방법이 있기는 하지만(예를 들어, 계층적 락) 아예 이렇게 코드를 작성하지 않는 것이 더 나은 경우가 많다.

스레드 종료와 정지

스레드를 강제로 종료하거나 정지할 수 있는 방법은 없다. 파이썬에서는 설계 자체를 이렇게 하였고 스레드 프로그램을 작성하는 일이 원래 복잡하기 때문에 그렇기도 하다. 예를 들어, 스레드가 락을 가지고 있는데 락을 해제하기 전에 강제로 종료하거나 정지하면 전체 응용 프로그램이 데드락에 빠질 수 있다. 게다가 복잡한 스레드 동기화를 위해서는 아주 정확한 순서로 락 획득과 해제 연산을 수행해야 하기 때문에 스레드가 종료할 때 간단히 “모든 락을 해제하는” 것은 일반적으로 불가능하다.

여러분이 스레드 종료 또는 정지 기능을 지원하고 싶으면 직접 구현해야 한다. 보통 스레드에서 루프를 돌면서 주기적으로 종료해야 하는지를 확인하게 하면 된다. 다음 예를 보자.

```
class StoppableThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__()
        self._terminate = False
        self._suspend_lock = threading.Lock()
    def terminate(self):
        self._terminate = True
    def suspend(self):
```

```

        self._suspend_lock.acquire()
def resume(self):
    self._suspend_lock.release()
def run(self):
    while True:
        if self._terminate:
            break
        self._suspend_lock.acquire()
        self._suspend_lock.release()
문장들
...

```

이 방법을 신뢰성 있게 구현하려면 스레드에서 블로킹 I/O 연산을 수행하지 않게 주의를 기울여야 한다. 예를 들어, 스레드에서 데이터 도착을 기다리느라 멈추어 있다면 해당 연산에서 깨어나기 전까지 종료하지 못한다. 이 때문에 타임아웃, 넌블로킹 I/O 또는 기타 고급 기능을 사용해 가끔씩 종료 검사를 할 수 있게 해야 한다.

유ти리티 함수

이 모듈에는 다음 유ти리티 함수들이 있다.

active_count()

현재 활성 Thread 객체의 수를 반환한다.

current_thread()

호출자의 제어 스레드에 해당하는 Thread 객체를 반환한다.

enumerate()

현재 모든 활성 Thread 객체들을 담은 리스트를 반환한다.

local()

스레드 지역 데이터를 저장할 수 있는 local 객체를 반환한다. 이 객체는 각 스레드에서 고유하다.

setprofile(func)

생성된 모든 스레드에 사용할 프로파일 함수를 설정한다. 각 스레드가 실행되기 전에 func가 sys.setprofile()로 전달된다.

settrace(func)

생성된 모든 스레드에 사용할 추적 함수를 설정한다. func는 각 스레드가 실행되기 전에 sys.settrace()로 전달된다.

stack_size([size])

새로운 스레드를 생성할 때 사용할 스택 크기를 반환한다. 옵션인 정수 size가 주어지면 새로운 스레드 생성에 사용할 스택 크기로 설정된다. size는 최대한 이식성을 보장하기 위해 32768(32KB) 이상이면서 4096(4KB)의 배수인 값을 사용하는 것이 좋다. 시스템에서 이 연산을 지원하지 않으면 ThreadError 예외가 발생한다.

전역 인터프리터 락

파이썬 인터프리터는 락으로 보호되기 때문에 여러 프로세서를 사용할 수 있는 경우에도 한 번에 한 스레드만 실행한다. 이는 계산 집중적 프로그램에서 스레드의 유용성을 심각하게 제약한다. 사실 CPU 위주 프로그램에서 스레드를 사용하면 같은 작업을 단순히 순차적으로 수행하는 경우보다 훨씬 느릴 수 있다. 따라서, 스레드는 네트워크 서버 같이 주로 I/O와 관련된 프로그램에만 사용해야 한다. 계산 집중적 작업에는 C 확장 모듈이나 multiprocessing 모듈의 사용을 고려하도록 한다. C 확장 모듈은 락이 해제되었을 때 인터프리터와 상호작용하지 않는다는 가정 아래에 인터프리터 락을 해제하고 병렬로 실행하는 옵션을 제공한다. multiprocessing 모듈은 작업을 락의 제한을 받지 않는 독립 하위 프로세스들로 나눈다.

스레드 프로그래밍

파이썬에서 락과 기타 동기화 기본 연산을 다양하게 조합하여 전통적인 다중 스레드 프로그램을 작성하는 것이 가능하지만 이것 대신 추천하는 한 가지 프로그래밍 스타일이 있다. 그것은 다중 스레드 프로그램을 서로 메시지 큐를 통해 통신하는 독립 작업들로 조직하는 것이다. 다음 절(queue 모듈)에서 예를 통해 여기에 대해 설명한다.

queue, Queue

queue 모듈(파이썬 2에서는 Queue)은 여러 실행 스레드 사이에 정보를 안전하게 교환하는 데 사용할 수 있는 여러 가지 다중 생성자, 다중 소비자 큐를 구현한다.

queue 모듈에는 세 가지 큐 클래스가 있다.

Queue([maxsize])

FiFO(first-in first-out: 선입 선출) 큐를 생성한다. maxsize는 큐에 들어갈 수 있는

최대 항목 수이다. maxsize를 생략하거나 0으로 설정하면 큐 크기는 무한대이다.

LifoQueue([maxsize])

LIFO(last-in first-out: 후입 선출) 큐를 생성한다(스택(stack)이라고도 함).

PriorityQueue([maxsize])

항목들이 가장 낮은 우선 순위에서 가장 높은 우선 순위로 정렬된 우선 순위 큐를 생성한다. 이 큐를 사용할 때 각 항목은 (priority, data) 형식의 튜플이어야 하며, 여기서 priority는 숫자이어야 한다.

queue 클래스의 인스턴스 q는 다음 메서드들을 가진다.

q.qsize()

큐의 대략적인 크기를 반환한다. 다른 스레드가 큐를 생성하고 있는 중일 수 있기 때문에 이 숫자를 전적으로 신뢰할 수는 없다.

q.empty()

큐가 비어 있으면 True를, 아니면 False를 반환한다.

q.full()

큐가 가득 차 있으면 True를, 아니면 False를 반환한다.

q.put(item [, block [, timeout]])

큐에 item을 추가한다. 옵션인 인수 block이 True(기본 값)이면 호출자는 빈 공간이 생길 때까지 기다린다. 그렇지 않으면(block이 False이면) 큐가 가득 차 있을 경우 Full 예외가 발생한다. timeout은 옵션인 타임아웃 값으로 초 단위로 지정한다. 타임아웃이 발생하면 Full 예외가 발생한다.

q.put_nowait(item)

q.put(item, False)와 같다.

q.get([block [, timeout]])

큐에서 항목을 하나 제거하여 반환한다. 옵션인 인수 block이 True(기본 값)이면 호출자는 새 항목이 나타날 때까지 기다린다. 그렇지 않으면(block이 False이면) 큐가 빈 경우 Empty 예외가 발생한다. timeout은 옵션인 타임아웃 값으로 초 단위로 지정한다. 타임아웃이 발생하면 Empty 예외가 발생한다.

q.get_nowait()

get(0)과 같다.

q.task_done()

항목 처리가 끝났음을 알리기 위해 소비자에서 호출한다. 이 메서드를 사용할 경우에는 큐에 있는 각 항목에 대해 한 번씩만 호출해야 한다.

q.join()

큐에 있는 모든 항목이 제거되고 처리될 때까지 기다린다. 큐에 있는 모든 각 항목에 대해 q.task_done()이 호출되어야 반환된다.

스레드에서 큐 사용 예

큐를 사용하면 다중 스레드 프로그램을 단순화할 수 있는 경우가 있다. 예를 들어, 락으로 보호해야 하는 공유 상태에 의존하는 대신 공유 큐를 사용해 스레드들을 서로 연결할 수 있다. 이런 모델에서 일꾼 스레드가 일반적으로 데이터 소비자 역할을 한다. 다음은 이 개념을 설명하는 예이다.

```
import threading
from queue import Queue # 파이썬 2에서는 from Queue를 사용한다.

class WorkerThread(threading.Thread):
    def __init__(self,*args,**kwargs):
        threading.Thread.__init__(self,*args,**kwargs)
        self.input_queue = Queue()
    def send(self,item):
        self.input_queue.put(item)
    def close(self):
        self.input_queue.put(None)
        self.input_queue.join()
    def run(self):
        while True:
            item = self.input_queue.get()
            if item is None:
                break
            # 항목을 처리한다(유용한 작업을 수행하도록 변경한다.)
            print(item)
            self.input_queue.task_done()
            # 완료. 표지 받음을 알리고 반환한다.
            self.input_queue.task_done()
        return

# 사용 예
w = WorkerThread()
w.start()
w.send("hello")      # 큐를 통해 항목을 일꾼 스레드에 보낸다.
w.send("world")
w.close()
```

이 클래스는 주의를 기울여 설계하였다. 먼저, 프로그래밍 API가 multiprocessing 모듈에 있는 파이프가 생성하는 Connection 객체의 하위 집합임을 볼 수 있다. 이 때문에 나중에 확장하기 쉽다. 예를 들어, 데이터를 보내는 코드를 그대로 두고서 나중에 일꾼 스레드를 별개 프로세스로 옮길 수 있다.

둘째, 스레드를 종료할 수 있다. close() 메서드는 나중에 만날 경우 스레드를 종료하게 만드는 표지를 큐에 넣는다.

마지막으로 이 프로그래밍 API는 코루틴과 거의 같다. 수행할 작업에 어떤 블로킹 연산도 없다면 run() 메서드를 코루틴으로 재구현할 수 있으며 스레드 사용을 아예 없앨 수 있다. 이렇게 되면 스레드 컨텍스트 전환에 드는 오버헤드가 없어지기 때문에 더 빠르게 실행된다.

코루틴과 마이크로스레딩

특정 종류의 응용에서는 작업 스케줄러와 함께 생성기나 코루틴들을 사용하여 협력적 사용자 공간 다중 스레딩을 구현하는 것이 나은 경우가 있다. 이를 마이크로스레딩(microthreading)이라고 부르지만 다양한 이름으로 불리기도 한다. 종종 이 개념은 태스크릿(tasklet), 그린 스레드(green thread), 그린렛(greenlet) 등의 문맥에서 등장한다. 이 기법은 프로그램에서 많은 수의 열린 파일이나 소켓을 다루어야 할 때 흔히 사용된다. 예를 들어, 1,000개의 클라이언트 접속을 동시에 처리하고자 하는 네트워크 서버를 생각할 수 있다. 이 경우 1,000개의 스레드를 생성하는 대신, 비동기 I/O나 폴링과 함께(select 모듈을 사용) I/O 이벤트를 처리하는 작업 스케줄러를 사용하는 것이다.

생성기나 코루틴 함수에 있는 yield문이 나중에 next() 또는 send() 연산을 통해 다시 시작될 때까지 함수 실행을 중지한다는 사실 때문에 이 프로그래밍 기법이 등장했다. 그렇기 때문에 스케줄러 루프를 사용하여 생성기 함수들 사이에 협력적으로 다중 작업을 수행할 수 있다. 다음은 이 아이디어를 보여주는 예이다.

```
def foo():
    for n in xrange(5):
        print("I'm foo %d" % n)
        yield

def bar():
    for n in xrange(10):
        print("I'm bar %d" % n)
        yield
```

```

def spam():
    for n in xrange(7):
        print("I'm spam %d" % n)
        yield

# 작업 큐를 생성하고 채운다.
from collections import deque
taskqueue = deque()
taskqueue.append(foo())      # 작업(생성기)들을 추가한다.
taskqueue.append(bar())
taskqueue.append(spam())

# 모든 작업을 수행한다.
while taskqueue:
    # 다음 작업을 가져온다.
    task = taskqueue.pop()
    try:
        # 다음 yield까지 실행하고 다시 큐에 넣는다.
        next(task)
        taskqueue.appendleft(task)
    except StopIteration:
        # 작업이 완료되었다.
        pass

```

여기에서 나온 것처럼 CPU 위주의 코루틴들을 정의하고 이들을 스케줄링하는 프로그램을 짜는 일은 드물다. 대신 보통은 I/O 위주 작업, 폴링 또는 이벤트 처리에 이 기법을 더 많이 사용한다. 이 기법을 활용한 고급 예는 21장의 ‘select’ 모듈 절에 나와 있다.

21장

Python Essential Reference

네트워크 프로그래밍과 소켓

이 장에서는 저수준 네트워크 서버와 클라이언트를 구현하는 데 사용할 수 있는 모듈을 설명한다. 파이썬에는 소켓으로 바로 프로그램을 작성하거나 HTTP 같은 고수준 응용 프로그램 프로토콜을 사용하는 등 광범위한 네트워크 지원 기능이 있다. 이곳에서는 일단 아주 간단히(약간 불친절하다고 느낄 수도 있을 것이다) 네트워크 프로그래밍에 대해 소개한다. 고급 네트워크 프로그래밍에 관한 자세한 정보는 리차드 스티븐스(W. Richard Stevens)가 쓴 〈유닉스 네트워크 프로그래밍〉 제1권 《네트워크 API: 소켓과 XTI(UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI)》(Prentice Hall, 1997) 같은 책을 참고하도록 한다. 22장에서는 응용 프로그램 수준 프로토콜과 관련된 모듈을 설명한다.

네트워크 프로그래밍 기본

파이썬의 네트워크 프로그래밍 모듈은 크게 두 인터넷 프로토콜인 TCP와 UDP를 지원한다. TCP 프로토콜은 연결 지향 프로토콜로서 머신 사이에 양방향 통신 스트림을 생성하는 데 사용한다. UDP는 저수준 패킷 기반 프로토콜(비연결형)이고 정식으로 연결을 생성하지 않고 정보를 개별 패킷으로 전송하거나 수신한다. TCP와 달리, UDP 통신은 신뢰성을 제공하지 않기 때문에 신뢰성 있는 통신을 요구하는 응용 프로그램에서 사용하려면 더 복잡할 수 있다. 이 때문에 인터넷 응용 프로그램에서는 대부분 TCP를 사용한다.

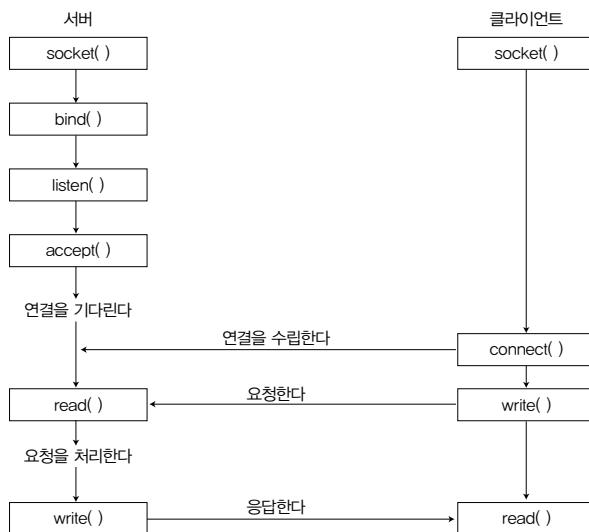
두 네트워크 프로토콜 모두 소켓(socket)^o라고 부르는 프로그래밍 추상 계층을

통해 다루어진다. 소켓은 파일과 비슷한 객체로서 들어오는 연결을 받아들이고 나가는 연결을 만들며 데이터를 주고받는 데 쓰인다. 두 머신이 통신하기 전에 둘 모두 소켓 객체를 만들어야 한다.

연결을 받는 머신(서버)은 소켓 객체를 알려진 포트 번호에 묶어야 한다. 포트(port)는 0-65535 범위에 있는 16비트 숫자이고 운영체제가 관리하고 클라이언트가 서버를 고유하게 식별하기 위해서 사용한다. 포트 0-1023은 시스템이 예약해놓은 것으로 공통 네트워크 프로토콜을 위해 쓰인다. 다음 표는 흔히 사용되는 프로토콜 몇 가지에 할당된 포트 번호를 보여준다(더 완전한 목록은 <http://www.iana.org/assignments/port-numbers>에서 찾을 수 있다).

서비스	포트 번호
FTP-데이터	20
FTP-제어	21
SSH	22
Telnet	23
SMTP(메일)	25
HTTP(WWW)	80
POP3	110
IMAP	143
HTTPS(보안 WWW)	443

그림 21.1 TCP 연결 프로토콜



TCP 연결을 수립하는 과정은 그림 21.1에 나와 있듯이 서버와 클라이언트 모두에서 일련의 정해진 과정에 따라 진행된다.

TCP 서버에서 연결을 받는 데 사용한 소켓 객체는 나중에 클라이언트와 통신을 하는 데 사용하는 소켓과 다르다. 특히 accept() 시스템 호출은 해당 연결에 실제로 사용할 새로운 소켓 객체를 반환한다. 이렇게 함으로써 서버에서 많은 수의 클라이언트를 동시에 처리할 수 있다.

UDP 통신도 비슷하게 진행이 되지만 그림 21.2에서 보듯이 클라이언트와 서버 사이에 연결을 수립하는 과정이 없다.

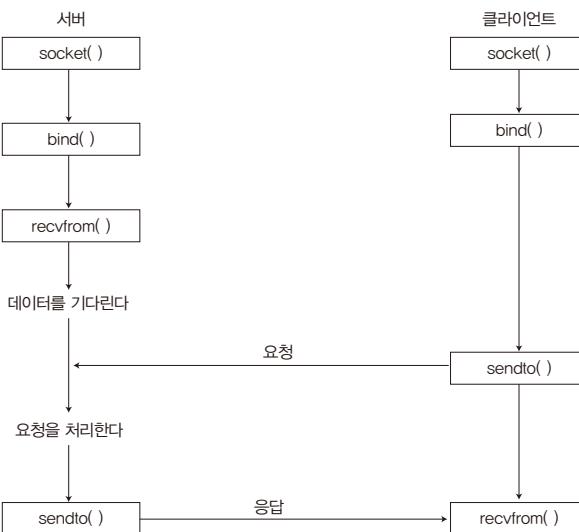
다음은 socket 모듈을 사용하여 클라이언트와 서버 사이에 TCP 프로토콜을 사용하는 예를 보여준다. 여기서 서버는 간단히 클라이언트에게 현재 시간을 문자열로 반환한다.

```
# 시간 서버 프로그램
from socket import *
import time

s = socket(AF_INET, SOCK_STREAM)      # TCP 소켓을 만든다.
s.bind(("",8888))                   # 포트 8888에 묶는다
s.listen(5)                          # 듣는다. 기다리는 연결이
                                    # 5개가 넘지 않게 제한한다.

while True:
```

그림 21.2 UDP 연결 프로토콜



```

client,addr = s.accept()    # 연결을 받는다.
print("Got a connection from %s" % str(addr))
timestr = time.ctime(time.time()) + "\r\n"
client.send(timestr.encode('ascii'))
client.close()

```

다음은 클라이언트 프로그램이다.

```

# 시간 클라이언트 프로그램
from socket import *
s = socket(AF_INET,SOCK_STREAM)    # TCP 소켓을 만든다.
s.connect(('localhost', 8888))      # 서버에 연결한다.
tm = s.recv(1024)                 # 1024 바이트까지만 받는다.
s.close()
print("The time is %s" % tm.decode('ascii'))

```

UCP 연결을 수립하는 예는 이 장의 ‘socket’ 모듈 절에 나온다.

네트워크 프로토콜을 통해 텍스트 형태로 데이터를 주고받는 일이 자주 있다. 이 경우 텍스트 인코딩에 주의를 기울여야 한다. 파이썬 3에서 모든 문자열은 유니코드 문자열이다. 따라서 네트워크로 텍스트 문자열을 보내려면 인코딩을 해야 한다. 그래서 앞에 나온 서버 코드에서 전송할 데이터에 encode('ascii') 메서드를 호출한 부분이 있었다. 비슷하게 클라이언트도 네트워크를 통해 데이터를 받을 때 먼저 인코딩 안 된 무가공 데이터를 받는다. 이것을 출력하거나 텍스트로 처리하려고 하면 기대했던 대로 작동하지 않는다. 먼저 디코딩을 해야 한다. 그래서 앞에 나온 클라이언트 코드에서 결과에 대해 decode('ascii')를 사용했다.

이 장의 나머지 부분에서는 소켓 프로그래밍과 관련된 모듈을 설명한다. 22장에서는 이메일이나 웹 같은 다양한 인터넷 응용 프로그래밍을 지원하는 고수준 모듈을 설명한다.

asynchat

asynchat 모듈은 asyncore 모듈을 사용하여 비동기 네트워킹을 지원하려는 응용 프로그램 구현을 간단하게 만들어준다. 이 모듈은 asyncore의 저수준 I/O 기능을 간단한 요청/응답 메커니즘에 기반한 네트워크 프로토콜(HTTP 같은)을 위해 설계된 고수준 프로그래밍 인터페이스로 감싼다.

이 모듈을 사용하려면 먼저 `async_chat`를 상속받아서 클래스를 정의해야 한다. 이 클래스 안에서 두 메서드 `collection_incoming_data()`와 `found_terminator()`를 정의해야 한다. 첫 번째 메서드는 네트워크 연결을 통해 데이터를 받을 때마다 호출

된다. 이 메서드에서는 보통 받은 데이터를 어딘가에 저장한다. `found_terminator()` 메서드는 요청 끝을 감지했을 때 호출된다. 예를 들어, HTTP 요청은 빈 줄로 끝난다.

데이터 출력을 위해서 `async_chat`은 생산자 FIFO 큐를 유지한다. 여러분이 데이터를 내보내면 간단히 이 큐에 추가된다. 그 다음 네트워크 연결을 통해 쓰기가 가능할 때마다 데이터가 이 큐에서 알아서 추출된다.

a.async_chat([sock])

새로운 처리기를 정의하는 데 사용하는 기반 클래스. `async_chat`는 `asyncore.dispatcher`에서 상속받았고 동일한 메서드를 제공한다. `sock`은 연결에 사용할 소켓 객체이다.

`async_chat`의 인스턴스 `a`는 `asyncore.dispatcher` 기반 클래스에 있는 것 이외에 추가로 다음 메서드들을 갖는다.

a.close_when_done()

생산자 FIFO 큐에 `None`을 넣음으로써 나가는 데이터 스트림에 파일 끝임을 알린다. 쓰기 객체가 여기에 도달하면 채널이 닫힌다.

a.collect_incoming_data(data)

채널을 통해 데이터를 받을 때마다 호출된다. `data`는 받은 데이터이고 보통 나중에 처리하기 위해 저장한다. 이 메서드는 사용자가 반드시 구현해야 한다.

a.discard_buffers()

입력, 출력 버퍼와 생산자 FIFO 큐에 있는 데이터를 모두 버린다.

a.found_terminator()

`set_terminator()`로 설정한 종료 조건이 만족되면 호출된다. 이 메서드는 사용자가 반드시 구현해야 한다. 보통 이 메서드에서는 이전에 `collect_incoming_data()` 메서드로 모은 데이터를 처리한다.

a.get_terminator()

채널의 종료기를 반환한다.

a.push(data)

채널의 나가는 생산자 FIFO 큐에 데이터를 집어넣는다. `data`는 보낼 데이터를 담은 문자열이다.

a.push_with_producer(producer)

생산자 객체 producer를 생산자 FiFO 큐에 밀어넣는다. producer에는 간단한 메서드인 more()를 갖는 아무 객체나 올 수 있다. more() 메서드는 호출될 때마다 문자열을 생성해야 한다. 빈 문자열은 데이터의 끝을 가리킨다. 내부적으로 `async_char` 클래스는 나가는 채널에 쓸 데이터를 얻기 위해서 `more()`를 반복적으로 호출한다. `push_with_producer()`를 반복적으로 호출해서 두 개 이상의 생산자 객체를 FiFO에 밀어넣을 수 있다.

s.set_terminator(term)

채널에 종료 조건을 설정한다. term은 문자열, 정수, None 중 하나일 수 있다. term이 문자열이면 입력 스트림에 해당 문자열이 나타날 때마다 `found_terminator()`가 호출된다. term이 정수이면 바이트 카운트를 의미한다. 해당하는 수의 바이트만큼 읽은 후 `found_terminator()`를 호출한다. term이 None이면 데이터는 끝없이 모인다.

이 모듈에는 `a.push_with_producer()` 메서드에서 사용하게 될 데이터를 생성하는 클래스 하나가 있다.

simple_producer(data [, buffer_size])

바이트 문자열 data에서 덩어리들을 생성하는 간단한 생산자 객체를 생성한다. `buffer_size`는 덩어리 크기를 나타내고 기본으로 512다.

asynchat 모듈은 항상 `asyncore` 모듈과 함께 쓰인다. 예를 들어, `asyncore`는 들어오는 연결을 받는 고수준 서버 구현에 쓰인다. `asynchat`은 이어서 각 연결에 대한 처리기를 구현하는 데 사용한다. 다음 예는 GET 요청을 처리하는 최소한의 웹 서버 구현을 통해 이 두 모듈을 어떻게 사용하는지를 보여준다. 이 예에서는 많은 여러 검사 부분과 상세한 부분을 생략하였지만 사용하기에는 문제없을 것이다. 이 예와 다음에 다룰 `asyncore` 모듈을 설명할 때 나오는 예를 비교해보기 바란다.

```
# asynchat을 사용한 비동기 HTTP 서버
import asynchat, asyncore, socket
import os
import mimetypes
try:
    from http.client import responses    # 파이썬 3
except ImportError:
    from httplib import responses        # 파이썬 2

# 이 클래스는 asyncore 모듈에 결합되어 단순히 연결을 받는 역할을 한다.
```

```

class async_http(asyncore.dispatcher):
    def __init__(self, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind(("", port))
        self.listen(5)

    def handle_accept(self):
        client, addr = self.accept()
        return async_http_handler(client)

# 비동기 HTTP 요청을 처리하는 클래스
class async_http_handler(asynchat.async_chat):
    def __init__(self, conn=None):
        asynchat.async_chat.__init__(self, conn)
        self.data = []
        self.got_header = False
        self.set_terminator(b"\r\n\r\n")

    # 들어오는 데이터를 받아서 데이터 버퍼에 추가한다.
    def collect_incoming_data(self, data):
        if not self.got_header:
            self.data.append(data)

    # 종료기를 받았다(빈 줄)
    def found_terminator(self):
        self.got_header = True
        header_data = b"".join(self.data)
        # 나중 처리를 위해 헤더 데이터(이진)를 텍스트로 디코딩한다.
        header_text = header_data.decode('latin-1')
        header_lines = header_text.splitlines()
        request = header_lines[0].split()
        op = request[0]
        url = request[1][1:]
        self.process_request(op, url)

    # 텍스트를 나가는 스트림에 밀어넣는다. 먼저 인코딩을 한다.
    def push_text(self, text):
        self.push(text.encode('latin-1'))

    # 요청을 처리한다.
    def process_request(self, op, url):
        if op == "GET":
            if not os.path.exists(url):
                self.send_error(404, "File %s not found\r\n")
            else:
                type, encoding = mimetypes.guess_type(url)
                size = os.path.getsize(url)
                self.push_text("HTTP/1.0 200 OK\r\n")
                self.push_text("Content-length: %s\r\n" % size)
                self.push_text("Content-type: %s\r\n" % type)
                self.push_text("\r\n")
                self.push_with_producer(file_producer(url))
        else:

```

```

        self.send_error(501,"%s method not implemented" % op)
        self.close_when_done()
    # 에러 처리
    def send_error(self,code,message):
        self.push_text("HTTP/1.0 %s %s\r\n" %
                      (code, responses[code]))
        self.push_text("Content-type: text/plain\r\n")
        self.push_text("\r\n")
        self.push_text(message)

    class file_producer(object):
        def __init__(self,filename,buffer_size=512):
            self.f = open(filename,"rb")
            self.buffer_size = buffer_size
        def more(self):
            data = self.f.read(self.buffer_size)
            if not data:
                self.f.close()
            return data

    a = async_http(8080)
    asyncore.loop()

```

이 예를 테스트하려면 서버가 실행되는 디렉터리에 있는 한 파일을 가리키는 URL을 입력하면 된다.

asyncore

asyncore 모듈은 select() 시스템 호출을 사용하여 네트워크 활동을 이벤트 루프를 통해 일련의 이벤트로서 비동기적으로 처리하는 네트워크 응용 프로그램을 만드는데 사용한다. 이 접근법은 동시성을 원하지만 스레드나 프로세스는 사용하지 않고 네트워크 프로그램을 작성하고자 할 때 유용하다. 또한 짧은 트랜잭션들을 빠르게 처리하고자 할 때도 좋다. 이 모듈에 있는 모든 기능은 보통의 소켓 객체를 감싸는 래퍼인 dispatcher 클래스가 구현한다.

dispatcher([sock])

이벤트 주도의 넌블록킹 소켓 객체를 정의하는 기반 클래스. sock은 기존 소켓 객체이다. sock을 생략했다면 소켓을 create_socket() 메서드로 생성해야 한다(곧 더 설명한다). 일단 생성되면 네트워크 이벤트가 특수 처리기 메서드에 의해 처리된다. 또한 모든 열린 디스패처 객체가 여러 폴링 함수에 의해서 사용되는 내부 리스트에 저장된다.

dispatcher 클래스의 다음 메서드들은 네트워크 이벤트를 처리하기 위해 호출된다. 이들은 하위 클래스에서 정의해야 한다.

d.handle_accept()

새로운 연결이 들어왔을 때 듣고 있는 소켓들에 대해서 호출된다.

d.handle_close()

소켓이 닫힐 때 호출된다.

d.handle_connect()

연결이 성립되었을 때 호출된다.

d.handle_error()

잡히지 않은 파이썬 예외가 발생했을 때 호출된다.

d.handle_expt()

소켓에 아웃오브밴드(out-of-band) 데이터가 도착했을 때 호출된다.

d.handle_read()

소켓에서 새로운 데이터를 읽을 수 있을 때 호출된다.

d.handle_write()

데이터 쓰기 시도가 이루어졌을 때 호출된다.

d.readable()

이 메서드는 객체가 데이터를 읽으려고 하는지를 알아보기 위해서 select() 루프에 의해서 호출된다. 그럴 경우 True를 반환하고 아닐 경우 False를 반환한다. 이 메서드는 새로운 데이터에 대해서 handle_read() 메서드가 호출되어야 하는지를 알아보기 위해서 호출된다.

d.writable()

객체가 데이터를 쓰기 원하는지를 알아보려고 select() 루프가 호출한다. 그럴 경우 True를, 아닐 경우 False를 반환한다. 이 메서드는 출력력을 생성하기 위해서 handle_write() 메서드가 호출되어야 하는지를 알아보기 위해서 항상 호출된다.

앞에 나온 메서드 말고도 다음 메서드들이 저수준 소켓 연산을 수행하는 데 사용된다. 이들은 소켓 객체에 있는 것과 비슷하다.

d.accept()

연결을 받는다. (client, addr)를 반환하며 여기서 client는 연결을 통해 데이터를 보내거나 받는 데 사용하는 소켓 객체이고 addr는 클라이언트 주소이다.

d.bind(address)

소켓을 address에 묶는다. address는 보통 (host, port) 튜플이지만 사용 중인 주소 체계에 따라 다를 수 있다.

d.close()

소켓을 닫는다.

d.connect(address)

연결을 생성한다. address는 (host, port) 튜플이다.

d.create_socket(family, type)

새 소켓을 만든다. 인수들은 socket.socket()의 것과 같다.

d.listen([backlog])

들어오는 연결을 듣는다. backlog는 내부 socket.listen() 함수에 전달될 정수 값이다.

d.recv(size)

최대 size 바이트만큼 받는다. 빈 문자열은 클라이언트가 채널을 닫았다는 것을 의미한다.

d.send(data)

data를 보낸다. data는 바이트 문자열이다.

다음 함수는 이벤트 처리를 위해 이벤트 루프를 시작한다.

loop([timeout [, use_poll [, map [, count]]]])

이벤트를 영원히 폴링한다. select() 함수가 폴링을 위해 사용된다. use_poll 매개 변수가 True인 경우 poll()이 대신 사용된다. timeout은 타임아웃 시간이고 기본으로 30초로 설정된다. map은 감시할 모든 채널을 담은 사전이다. count는 반복 전에 몇 번의 폴링 연산을 수행할지를 지정한다. count가 None(기본 값)이면 loop()는 모든 채널이 닫힐 때까지 영원히 폴링을 수행한다. count가 1이면 이 함수는 한번 폴링하고 반환한다.

예

다음 예는 `asyncore`를 사용하여 최소한의 웹 서버를 구현한 것이다. 두 클래스가 사용되었다. `asynhttp`는 연결을 받는 데 사용하고 `asynclient`는 클라이언트의 요청을 처리한다. 이 예제를 `asynchat` 모듈에서 나온 예와 비교해보기 바란다. 주요한 차이점은 이 예가 다소 저수준에서 작동한다는 점이다. 입력 스트림을 여러 줄로 나누고 넘치는 데이터를 버퍼링하고 요청 헤더를 종료하는 빈 줄을 찾는 등 더 신경 썼다.

```
# 비동기 HTTP 서버
import asyncore, socket
import os
import mimetypes
import collections
try:
    from http.client import responses    # 파이썬 3
except ImportError:
    from httplib import responses       # 파이썬 2

# 이 클래스는 단순히 연결을 받는다.
class async_http(asyncore.dispatcher):
    def __init__(self, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind(("", port))
        self.listen(5)

    def handle_accept(self):
        client, addr = self.accept()
        return async_http_handler(client)

# 클라이언트를 처리한다.
class async_http_handler(asyncore.dispatcher):
    def __init__(self, sock = None):
        asyncore.dispatcher.__init__(self, sock)
        self.got_request = False    # HTTP 요청을 읽은 것인가?
        self.request_data = b""
        self.write_queue = collections.deque()
        self.responding = False

    # 요청 헤더가 읽히지 않았을 때만 읽기가 가능하다.
    def readable(self):
        return not self.got_request

    # 들어오는 요청을 읽는다.
    def handle_read(self):
        chunk = self.recv(8192)
        self.request_data += chunk
        if b'\r\n\r\n' in self.request_data:
```

```

        self.handle_request()

# 들어오는 요청을 처리한다.
def handle_request(self):
    self.got_request = True
    header_data = self.request_data[:self.request_data.find(
        b'\r\n\r\n')]
    header_text = header_data.decode('latin-1')
    header_lines = header_text.splitlines()
    request = header_lines[0].split()
    op = request[0]
    url = request[1][1:]
    self.process_request(op,url)

# 요청을 처리한다.
def process_request(self,op,url):
    self.responding = True
    if op == "GET":
        if not os.path.exists(url):
            self.send_error(404,"File %s not found\r\n" % url)
        else:
            type, encoding = mimetypes.guess_type(url)
            size = os.path.getsize(url)
            self.push_text('HTTP/1.0 200 OK\r\n')
            self.push_text('Content-length: %d\r\n' % size)
            self.push_text('Content-type: %s\r\n' % type)
            self.push_text('\r\n')
            self.push(open(url,"rb").read())
    else:
        self.send_error(501,"%s method not implemented" % self.op)

# 에러 처리
def send_error(self,code,message):
    self.push_text('HTTP/1.0 %s %s\r\n' % (code, responses[code]))
    self.push_text('Content-type: text/plain\r\n')
    self.push_text('\r\n')
    self.push_text(message)

# 이진 데이터를 출력 큐에 추가한다.
def push(self,data):
    self.write_queue.append(data)

# 텍스트 데이터를 출력 큐에 추가한다.
def push_text(self,text):
    self.push(text.encode('latin-1'))

# 응답이 준비되었을 때만 쓰기가 가능하다.
def writable(self):
    return self.responding and self.write_queue

# 응답 데이터를 쓴다.
def handle_write(self):
    chunk = self.write_queue.popleft()
    bytes_sent = self.send(chunk)

```

```

if bytes_sent != len(chunk):
    self.write_queue.appendleft(chunk[bytes_sent:])
if not self.write_queue:
    self.close()

# 서버를 생성한다.
a = async_http(8080)
# 영원히 폴링한다.
asyncore.loop()

```

참고

socket(578페이지), select(565페이지), http 패키지(615페이지), SocketServer(605페이지)

select

select 모듈은 select()와 poll() 시스템 호출에 접근할 수 있게 한다. select()는 스크리드나 하위 프로세스를 사용하지 않고 여러 입력/출력 스트림에 걸쳐서 폴링을 구현하거나 다중 처리를 구현하는 데 보통 사용한다. 유닉스에서는 파일, 소켓, 파일, 기타 대부분의 파일 타입에 대해 사용할 수 있다. 윈도에서는 소켓에 대해서만 작동한다.

select(iwtd, owtd, etwd [, timeout])

파일 기술자 그룹의 입력, 출력, 예외 상태를 질의한다. 처음 세 개의 인수는 정수 파일 기술자나 파일 기술자를 반환하는 fileno() 메서드를 갖는 객체를 담은 리스트들이다. iwtd 매개변수는 입력을 기다리는 객체들을 나타내고 owtd는 출력을 기다리는 객체들을 나타내며 etwd는 예외 조건을 기다리는 객체들을 나타낸다. 각 리스트는 비어 있을 수 있다. timeout은 타임아웃 시간을 초로 나타낸 부동 소수점 수이다. timeout을 생략하면 이 함수는 적어도 하나 이상의 파일 기술자가 준비될 때까지 기다린다. 0이면 이 함수는 단순히 폴링을 수행하고 바로 반환한다. 반환되는 값은 준비된 객체들을 담은 리스트들의 튜플이다. 이것은 처음 세 인수의 하위 집합이다. 타임아웃이 되기 전까지 아무 객체도 준비가 되어 있지 않으면 세 개의 빈 리스트가 반환된다. 예러가 발생하면 select.error 예외가 발생한다. 이 예외의 값은 IOError나 OSError가 반환하는 것과 같다.

poll()

poll() 시스템 호출을 활용하는 폴링 객체를 생성한다. poll()을 지원하는 시스템에서만 사용할 수 있다.

`poll()`이 반환하는 폴링 객체 `p`는 다음 메서드들을 갖는다.

`p.register(fd [, eventmask])`

새로운 파일 기술자 `fd`를 등록한다. `fd`는 정수 파일 기술자이거나 기술자를 얻는데 사용되는 `fileno()` 메서드를 제공하는 객체일 수 있다. `eventmask`는 다음 플래그들을 비트 OR한 것으로 관심 있는 이벤트를 나타낸다.

상수	설명
<code>POLLIN</code>	데이터를 읽을 수 있다.
<code>POLLPRI</code>	급한 데이터를 읽을 수 있다.
<code>POLLOUT</code>	쓸 준비가 되었다.
<code>POLLERR</code>	에러 조건
<code>POLLHUP</code>	끊겼다.
<code>POLLNVAL</code>	유효하지 않은 요청

`eventmask`를 생략하면 `POLLIN`, `POLLPRI`, `POLLOUT` 이벤트가 켜진다.

`p.unregister(fd)`

폴링 객체에서 파일 기술자 `fd`를 제거한다. 해당 파일이 등록되어 있지 않으면 `KeyError`가 발생한다.

`p.poll([timeout])`

모든 등록된 파일 기술자에 대해서 이벤트를 폴링한다. `timeout`은 밀리세컨드로 지정하는 옵션인 타임아웃 시간을 나타낸다. (`fd`, `event`) 튜플들의 리스트를 반환하며, 여기서 `fd`는 파일 기술자를 나타내고 `event`는 이벤트를 나타내는 비트마스크이다. 이 비트마스크의 필드들은 `POLLIN`, `POLLOUT` 등의 상수에 대응한다. 예를 들어, `POLLIN` 이벤트를 검사하려면 간단히 `event & POLLIN`으로 값을 검사하면 된다. 빈 리스트가 반환되면 타임아웃이 발생한 것이며 아무런 이벤트가 발생하지 않았다는 것을 나타낸다.

고급 모듈 기능

이 모듈에서 `select()`와 `poll()` 함수가 가장 이식성 있는 함수이다. 리눅스에서 `select` 모듈은 훨씬 좋은 성능을 보이는 애지와 레벨 트리거 폴링(edge and level trigger polling: epoll)에 대한 인터페이스도 제공한다. BSD 시스템에서는 커널 큐와 이벤트 객체에 대한 접근을 제공한다. 이러한 프로그래밍 인터페이스에 대해서는 <http://docs.python.org/library/select>에 있는 `select`를 설명하는 온라인 문서에 나와 있다.

고급 비동기 I/O 예

select 모듈은 스레드나 프로세스를 사용하지 않고 동시 실행 기능을 구현하는 데 사용하는 기법인 태스크лет(tasklet)이나 코루틴을 통해 서버를 구현하는 데 사용되곤 한다. 다음 고급 예는 코루틴으로 I/O 기반 작업 스케줄러를 구현하여 이 개념을 보여준다. 이는 이 책에 나오는 예 중에서 가장 고급 예이고 이해하는 데 얼마간 공부를 해야 할 것이다. 추가 참고 자료로 필자가 진행했던 PyCON'09 퓨토리얼인 ‘코루틴과 동시성에 관한 별난 강좌(A Curious Course on Coroutines and Concurrency)’(<http://www.dabeaz.com/coroutines>)를 참고하기 바란다.

```

import select
import types
import collections

# 실행 중인 작업을 나타내는 객체
class Task(object):
    def __init__(self,target):
        self.target = target # 코루틴
        self.sendval = None # 다시 시작했을 때 전송할 값
        self.stack = [] # 호출 스택
    def run(self):
        try:
            result = self.target.send(self.sendval)
            if isinstance(result,SystemCall):
                return result
            if isinstance(result,types.GeneratorType):
                self.stack.append(self.target)
                self.sendval = None
                self.target = result
            else:
                if not self.stack: return
                self.sendval = result
                self.target = self.stack.pop()
        except StopIteration:
            if not self.stack: raise
            self.sendval = None
            self.target = self.stack.pop()

# 시스템 호출을 나타내는 객체
class SystemCall(object):
    def handle(self,sched,task):
        pass

# 스케줄러 객체
class Scheduler(object):
    def __init__(self):
        self.task_queue = collections.deque()
        self.read_waiting = {}
        self.write_waiting = {}
        self.numtasks = 0

```

```

# 코루틴을 가지고 새 작업을 생성한다.
def new(self,target):
    newtask = Task(target)
    self.schedule(newtask)
    self.numtasks += 1

# 작업을 작업 큐에 넣는다.
def schedule(self,task):
    self.task_queue.append(task)

# 작업이 파일 기술자에서 데이터를 기다리게 한다.
def readwait(self,task,fd):
    self.read_waiting[fd] = task

# 작업이 파일 기술자에서 쓰기를 기다리도록 한다.
def writewait(self,task,fd):
    self.write_waiting[fd] = task

# 주 스케줄러 루프
def mainloop(self,count=-1,timeout=None):
    while self.numtasks:
        # 처리할 I/O 이벤트를 검사한다.
        if self.read_waiting or self.write_waiting:
            wait = 0 if self.task_queue else timeout
            r,w,e = select.select(self.read_waiting,
                                  self.write_waiting, [], wait)
            for fileno in r:
                self.schedule(self.read_waiting.pop(fileno))
            for fileno in w:
                self.schedule(self.write_waiting.pop(fileno))

        # 큐에서 실행 준비가 된 모든 작업을 실행한다.
        while self.task_queue:
            task = self.task_queue.popleft()
            try:
                result = task.run()
                if isinstance(result, SystemCall):
                    result.handle(self,task)
                else:
                    self.schedule(task)
            except StopIteration:
                self.numtasks -= 1

        # 실행할 작업이 없으면 기다릴 것인지 반환할 것인지 결정한다.
        else:
            if count > 0: count -= 1
            if count == 0:
                return

    # 여러 시스템 호출 구현
    class ReadWait(SystemCall):
        def __init__(self,f):
            self.f = f
        def handle(self,sched,task):
            fileno = self.f.fileno()

```

```

        sched.readwait(task,fileno)

class WriteWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self,sched,task):
        fileno = self.f.fileno()
        sched.writewait(task,fileno)

class NewTask(SystemCall):
    def __init__(self,target):
        self.target = target
    def handle(self,sched,task):
        sched.new(self.target)
        sched.schedule(task)

```

이 예에서는 아주 작은 운영체제를 구현하고 있다. 이 코드가 작동하는 방식에 관해서 몇 가지 이야기를 하자면 다음과 같다.

- 모든 일은 코루틴 함수가 수행한다. 코루틴은 생성기처럼 yield문을 사용하지만 코루틴에 대해서는 반복을 수행하는 것이 아니라 send(value) 메서드로 값을 보낸다.
- 실행 중인 작업을 나타내는 Task 클래스는 단순히 코루틴 위에 얹은 얇은 층일 뿐이다. Task 객체 task는 task.run() 연산 하나만을 지원한다. 이 메서드는 작업을 다시 시작하고 다음 yield문을 만날 때까지 실행한 후 다시 멈춘다. 작업을 실행할 때 task.sendval 속성은 yield 표현식에 보내질 값을 담는다. 작업은 다음 yield문을 만날 때까지 실행된다. yield가 생성하는 값은 다음에 무슨 일이 벌어질지를 제어한다.
 - 이 값이 또 다른 코루틴이면(type.GeneratorType) 잠시 그 코루틴으로 제어를 넘기겠다는 의미이다. Task 객체의 stack 속성은 이런 경우 만들어지는 코루틴 호출 스택을 나타낸다. 작업이 다음에 실행되면 이 새 코루틴으로 제어가 넘어간다.
 - 이 값이 SystemCall 인스턴스이면 스케줄러가 대신 무언가 하기를 원한다는 의미이다(새 작업을 시작시키거나 I/O를 기다리는 등). 이 객체를 사용하는 목적은 곧 설명한다.
 - 이 값이 기타 다른 값이면 둘 중 하나이다. 현재 실행 중인 코루틴이 하위 코루틴으로 실행되고 있었다면 작업 호출 스택에서 꺼내지고 호출자에게 전달할 수 있도록 이 값을 저장한다. 호출자는 다음에 작업이 실행될 때 이 값을

받게 된다. 코루틴이 실행 중인 유일한 코루틴이라면 반환되는 값은 그냥 버려진다.

- StopIteration을 처리하는 부분은 종료된 코루틴을 위한 것이다. 이런 일이 벌어지면 제어가 이전 코루틴으로 넘어가거나(하나라도 있으면) 예외를 스케줄러에게 전달하여 작업이 종료되었다는 것을 알린다.

- SystemCall 클래스는 스케줄러에서 시스템 호출을 나타낸다. 실행 중인 작업에서 스케줄러가 자신을 위해서 어떤 일을 해주기를 원할 때 SystemCall 인스턴스를 만든다. 이 객체는 시스템 호출이라고 불리는데 그 이유는 유닉스나 윈도 같은 실제 다중처리 운영체제에서 프로그램이 서비스를 요청하는 방식을 흉내내기 때문이다. 프로그램에서 운영체제의 서비스를 원할 경우 운영체제에 제어를 넘겨주고 시스템이 무엇을 해야 하는지를 알 수 있게 정보를 제공한다. 이런 측면에서 SystemCall 인스턴스는 일종의 시스템 트랩(trap)을 실행하는 것과 비슷하다.
- Scheduler 클래스는 관리되고 있는 Task 객체들의 모음을 나타낸다. 그 핵심에는 실행할 준비가 된 작업들을 추적하는 작업 큐가 있다(task_queue 속성). 작업 큐와 관련해서 네 가지 기본 연산이 존재한다. new()는 새 코루틴을 만들고 Task 객체로 감싼 다음 작업 큐에 넣는다. schedule()은 기존 Task를 받아서 다시 작업 큐에 넣는다. mainloop()는 루프를 돌면서 스케줄러를 실행하고 작업이 더 없을 때까지 하나씩 작업을 실행한다. readwait()와 writewait() 메서드는 Task 객체를 I/O 이벤트를 기다릴 수 있게 임시 집합지에 둔다. 이렇게 되면 작업은 실행되지 않지만 죽은 것도 아니다. 단지 다시 실행될 순간을 기다리게 된다.
- mainloop() 메서드는 스케줄러에서 핵심부이다. 이 메서드는 먼저 I/O 이벤트를 기다리는 작업이 있는지를 검사한다. 그런 경우 I/O 활동을 폴링하기 위해서 select()를 호출한다. 관심 이벤트가 있는 경우 관련 작업들을 작업 큐로 이동 시켜 실행될 수 있게 한다. 다음으로 mainloop() 메서드는 작업 큐에서 작업을 꺼내어 run() 메서드를 실행한다. 어떤 작업이 종료되면(StopIteration) 그 작업을 버린다. 어떤 작업이 단순히 yield를 수행한 경우에는 다시 실행될 수 있게 작업 큐에 다시 넣는다. 이 과정은 실행할 작업이 하나도 없거나 모든 작업이 I/O 이벤트를 기다린다고 멈추어 있을 때까지 계속된다. 옵션으로 mainloop() 함수는 I/O 폴링 연산을 몇 번까지 한 후 반환할 것인지를 나타내는 count 매개변

수를 받는다. 스케줄러를 다른 이벤트 루프에 통합할 계획이라면 유용하게 쓸 수 있는 매개변수다.

- 스케줄러에서 가장 미묘한 부분은 mainloop() 메서드에서 SystemCall 인스턴스를 처리하는 것이다. 어떤 작업이 SystemCall 인스턴스를 생성하면 스케줄러는 관련 Scheduler와 Task 객체를 매개변수로 넘기면서 이 인스턴스의 handle() 메서드를 호출한다. 이 시스템 호출의 목적은 작업이나 스케줄러와 관련된 내부 연산을 수행하기 위함이다. ReadWait(), WriteWait(), NewTask() 클래스는 I/O를 위해 작업을 잠시 멈추거나 새 작업을 만드는 시스템 호출의 예이다. 예를 들어, ReadWait()는 작업을 받아 스케줄러에 readwait() 메서드를 호출한다. 그러면 스케줄러는 해당 작업을 받아 적절한 집합지에 둔다. 여기서 객체들 사이에 분리(decoupling)가 이루어지는 것을 볼 수 있다. 작업은 서비스를 호출하기 위해서 SystemCall 객체를 만들지만 직접 스케줄러와 상호 작용하지는 않는다. SystemCall 객체는 작업과 스케줄러에 연산을 수행하지만 특정 스케줄러나 작업 구현에 묶이지 않는다. 따라서 이론적으로 여러분은 이 프레임워크에 끼워 넣을 수 있는 완전 다른 스케줄러 구현(스레드를 사용한다든지 하는)을 작성 할 수 있고 그래도 문제 없이 작동한다.

다음은 이 I/O 작업 스케줄러를 사용해 간단한 네트워크 시간 서버를 구현한 예이다. 앞에서 설명한 내용을 이해하는 데 도움이 될 것이다.

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
    import time
    s = socket(AF_INET,SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        yield ReadWait(s)
        conn,addr = s.accept()
        print("Connection from %s" % str(addr))
        yield WriteWait(conn)
        resp = time.ctime() + "\r\n"
        conn.send(resp.encode('latin-1'))
        conn.close()

    sched = Scheduler()
    sched.new(time_server(("," ,10000))) # 포트 10000에 서버를 생성
    sched.new(time_server(("," ,11000))) # 포트 11000에 서버를 생성
    sched.run()
```

이 예에서 두 서버가 동시에 실행된다. 두 서버는 각각 다른 포트에서 기다린

다(telnet을 사용해서 연결하고 검사해 보도록 한다). yield ReadWait()와 yield WriteWait() 문은 해당 소켓이 I/O를 수행할 수 있을 때까지 각 서버를 실행하는 코루틴을 기다리게 한다. 이 두 문장이 반환되면 accept()나 send()로 I/O 연산을 바로 수행할 수 있다.

ReadWait와 WriteWait를 사용하는 것이 약간 저수준의 연산인 것처럼 느껴질 수도 있다. 운좋게도 우리의 설계는 이런 연산들을 라이브러리 함수나 메서드(코루틴인 경우에 한해서) 뒤로 숨길 수 있게 한다. 소켓 객체를 감싸서 관련 인터페이스를 흉내내는 다음 객체를 보자.

```
class CoSocket(object):
    def __init__(self,sock):
        self.sock = sock
    def close(self):
        yield self.sock.close()
    def bind(self,addr):
        yield self.sock.bind(addr)
    def listen(self,backlog):
        yield self.sock.listen(backlog)
    def connect(self,addr):
        yield WriteWait(self.sock)
        yield self.sock.connect(addr)
    def accept(self):
        yield ReadWait(self.sock)
        conn, addr = self.sock.accept()
        yield CoSocket(conn), addr
    def send(self, data):
        while data:
            evt = yield WriteWait(self.sock)
            nsent = self.sock.send(data)
            data = data[nsent:]
    def recv(self,maxsize):
        yield ReadWait(self.sock)
        yield self.sock.recv(maxsize)
```

다음은 CoSocket 클래스로 다시 구현한 시간 서버이다.

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
    import time
    s = CoSocket(socket(AF_INET,SOCK_STREAM))
    yield s.bind(address)
    yield s.listen(5)
    while True:
        conn,addr = yield s.accept()
        print(conn)
        print("Connection from %s" % str(addr))
        resp = time.ctime()+"\r\n"
        yield conn.send(resp.encode('latin-1'))
        yield conn.close()
```

```

sched = Scheduler()
sched.new(time_server(("," ,10000))) # 포트 10000에 서버를 생성
sched.new(time_server(("," ,11000))) # 포트 11000에 서버를 생성
sched.run()

```

이 예에서 CoSocket 객체의 프로그래밍 인터페이스는 보통 소켓과 거의 같다. 유일한 차이점은 각 연산 앞에 yield가 붙어 있다는 점이다(모든 메서드가 코루틴으로 정의되었기 때문이다). 처음에는 이 모든 것이 너무 복잡하고 왜 이렇게 하는지 궁금할 것이다. 하지만 서버를 실행해보면 스레드나 하위 프로세스를 사용하지 않고서도 동시에 실행되는 것을 볼 수 있을 것이다. 그뿐만 아니라 yield 키워드를 무시하고 보면 보통의 제어 흐름을 갖는다.

다음은 다중 클라이언트 연결을 동시에 처리하지만 역호출 함수나 스레드나 프로세스를 사용하지 않는 비동기 웹 서버를 보여준다. 이것을 asynchat과 asyncore 모듈을 설명할 때 살펴보았던 예와 비교해보기 바란다.

```

import os
import mimetypes
try:
    from http.client import responses    # 파일 3
except ImportError:
    from httplib import responses        # 파일 2
from socket import *

def http_server(address):
    s = CoSocket(socket(AF_INET, SOCK_STREAM))
    yield s.bind(address)
    yield s.listen(50)

    while True:
        conn,addr = yield s.accept()
        yield NewTask(http_request(conn,addr))
        del conn, addr

def http_request(conn,addr):
    request = b""
    while True:
        data = yield conn.recv(8192)
        request += data
        if b'\r\n\r\n' in request: break

    header_data   = request[:request.find(b'\r\n\r\n')]
    header_text   = header_data.decode('latin-1')
    header_lines  = header_text.splitlines()
    method, url, proto = header_lines[0].split()
    if method == 'GET':
        if os.path.exists(url[1:]):
            yield serve_file(conn,url[1:])
        else:

```

```

        yield error_response(conn,404,"File %s not found" % url)
    else:
        yield error_response(conn,501,"%s method not implemented" % method)
    yield conn.close()

def serve_file(conn,filename):
    content,encoding = mimetypes.guess_type(filename)
    yield conn.send(b"HTTP/1.0 200 OK\r\n")
    yield conn.send(("Content-type: %s\r\n" %
                    content).encode('latin-1'))
    yield conn.send(("Content-length: %d\r\n" %
                    os.path.getsize(filename)).encode('latin-1'))
    yield conn.send(b"\r\n")
    f = open(filename,"rb")
    while True:
        data = f.read(8192)
        if not data: break
        yield conn.send(data)

def error_response(conn,code,message):
    yield conn.send(("HTTP/1.0 %d %s\r\n" %
                    (code, responses[code])).encode('latin-1'))
    yield conn.send(b"Content-type: text/plain\r\n")
    yield conn.send(b"\r\n")
    yield conn.send(message.encode('latin-1'))

sched = Scheduler()
sched.new(http_server(("0.0.0.0",8080)))
sched.mainloop()

```

이 예를 유심히 살펴보면 고급 써드 파티 모듈에서 사용하는 코루틴과 병행 프로그래밍 기법에 대한 큰 통찰력을 얻을 수 있을 것이다. 그래도 이 기법을 너무 과도하게 사용하면 다음 코드 리뷰 이후에 해고될 수도 있으니 조심하기 바란다.

언제 비동기 네트워킹을 사용할 것인가

앞에 나온 예에서 본 비동기 I/O(asyncore와 asynchat), 폴링, 코루틴 등은 파이썬 개발에서 가장 난해한 부분 중에 하나이다. 그렇지만 이 기법들은 여러분이 생각하는 것보다 더 자주 쓰인다. 비동기 I/O를 사용하는 이유로 자주 언급되는 것이 많은 수의 스레드를 사용해서 프로그래밍을 할 때 나타나는 오버헤드를 최소화하기 위한 것이다. 특히, 많은 수의 클라이언트를 처리해야 할 때 전역 인터프리터 락과 관련된 제약을 생각해보면 그렇다(20장 참고).

역사적으로 asyncore 모듈은 비동기 I/O를 지원했던 첫 라이브러리 모듈이었다. asynchat 모듈은 코딩을 더 간단하게 하려고 나중에 나온 것이다. 그렇지만 두 모듈 모두 I/O를 이벤트로서 처리하는 점에서는 같다. 예를 들어, I/O 이벤트가 발생

했을 때 역호출 함수가 실행된다. 역호출 함수는 I/O 이벤트에 반응하여 필요한 처리를 수행한다. 이런 식으로 큰 규모의 응용 프로그램을 작성하게 되면 이벤트 처리가 응용 프로그램의 모든 부분에 영향을 주는 것을 볼 수 있다(예를 들어, I/O 이벤트가 역호출을 일으키고 이것이 다시 역호출을 더 일으키게 하고). 인기 있는 네트워킹 패키지 중 하나인 Twisted(<http://twistedmatrix.com>)는 이 접근법을 기반으로 기능을 크게 확장한 것이다.

코루틴은 좀 더 현대적인 것으로서 파이썬 2.5에서 도입된 후 상대적으로 덜 사용되고 이해가 부족한 것 같다. 코루틴의 중요한 특징 중 하나가 전체적인 제어 흐름이 스레드 프로그램처럼 보이는 프로그램을 작성할 수 있다는 점이다. 예를 들어, 앞의 예에서 웹 서버는 역호출 함수를 사용하지 않았고 스레드를 사용했을 때의 모습과 거의 유사한 모습이다. 단지 yield문 사용에 익숙해지기만 하면 된다. Stackless Python(<http://www.stackless.com>)은 이 아이디어를 더 확장한다.

일반적으로 여러분은 대부분의 네트워크 응용 프로그램에 대해서 비동기 I/O 기법을 사용하는 것에 거부감을 느낄 것이다. 예를 들어, 수백, 수천 개의 동시 네트워크 연결 위에 데이터를 지속적으로 전송해야 하는 서버를 작성한다면 스레드를 사용하는 것이 훨씬 나은 성능을 보일 수 있다. 그 이유는 select()는 감시해야 하는 연결 개수가 많아질수록 성능이 급격하게 저하되기 때문이다. 리눅스에서는 epoll() 같은 특수 함수를 사용해서 이런 단점을 줄일 수 있지만 그렇게 하면 코드 이식성이 떨어지게 된다. 비동기 I/O를 사용하는 것이 가장 좋을 때는 네트워킹이 다른 이벤트 루프(GUI 같은)에 통합되어야 하는 경우라든지 아니면 상당한 양의 CPU 처리가 필요한 곳에 네트워킹이 결합되어야 하는 경우이다. 이런 경우에는 비동기 네트워킹을 사용하는 것이 훨씬 빠른 응답 속도를 보인다.

예를 통해 살펴보자는 의미에서 ‘벽장 속 맥주 천만 병(10million bottles of beer on the wall)’이라는 노래에 나오는 일을 하는 다음 프로그램을 보자.

```
bottles = 10000000

def drink_beer():
    remaining = 12.0
    while remaining > 0.0:
        remaining -= 0.1

def drink_bottles():
    global bottles
    while bottles > 0:
        drink_beer()
```

```
bottles -= 1
```

이제 클라이언트가 접속해서 맥주가 몇 병이나 남았는지 볼 수 있도록 이 코드에 원격 감시 기능을 추가하고 싶다고 하자. 한 가지 방법으로 다음과 같이 각 서버를 자신만의 스레드로 띄워서 주 응용 프로그램과 함께 실행되게 할 수 있다.

```
def server(port):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.bind(("",port))
    s.listen(5)
    while True:
        client,addr = s.accept()
        client.send(( "%d bottles\r\n" % bottles).encode('latin-1'))
        client.close()
    # 감시 서버를 구동한다.
    thr = threading.Thread(target=server,args=(10000,))
    thr.daemon=True
    thr.start()
    drink_bottles()
```

다른 방법으로는 I/O 폴링에 기반한 서버를 작성하고 주 계산 루프에 폴링 연산을 바로 집어넣는 방법이 있다. 다음 예는 앞에서 개발한 코루틴 스케줄러를 사용하는 예이다.

```
def drink_bottles():
    global bottles
    while bottles > 0:
        drink_beer()
        bottles -= 1
    scheduler.mainloop(count=1,timeout=0) # 연결을 폴링한다.

# 코루틴에 기반한 비동기 서버
def server(port):
    s = CoSocket(socket.socket(socket.AF_INET,socket.SOCK_STREAM))
    yield s.bind(("",port))
    yield s.listen(5)
    while True:
        client,addr = yield s.accept()
        yield client.send(( "%d bottles\r\n" % bottles).
                           encode('latin-1'))
        yield client.close()

scheduler = Scheduler()
scheduler.new(server(10000))
drink_bottles()
```

별도의 프로그램을 작성해서 맥주병 프로그램에 주기적으로 연결하게 한 다음 상태 메시지를 받는 데 걸리는 응답 시간을 재보면 놀라운 결과를 얻게 된다. 필자의 머신에서(듀얼 코어 2GHz 맥북) 코루틴 기반 서버의 평균 응답 시간(1,000개의

요청에 대해)은 약 1ms였고 스레드 버전은 약 5ms였다. 이 차이는 코루틴 기반 코드의 경우 연결을 감지하자마자 응답할 수 있는 데 반해서 스레드 서버는 운영체제가 스케줄을 잡아줄 때까지 실행되기 못하기 때문이라는 것으로 설명할 수 있다. CPU 위주 스레드와 파이썬 전역 인터프리터 락이 사용되는 경우라면 서버는 CPU 위주 스레드가 자신에게 할당된 시간 조각을 다 쓸 때까지 기다려야 할지도 모른다. 많은 시스템에서 이 시간은 약 10ms이며 따라서 앞에서 스레드 응답 시간을 대략 쪘던 부분은 정확히 CPU 위주 작업이 운영체제에 의해서 선점될 때까지 기다려야 할 것으로 기대되는 평균 시간이 된다.

폴링의 단점으로는 너무 자주 사용될 경우 심대한 오버헤드를 가져온다는 점이 있다. 예를 들어, 이 예에서는 응답 시간이 더 짧기는 했지만 폴링이 들어간 프로그램은 종료되는 데까지 50% 정도 시간이 더 걸린다. 여섯 개짜리 맥주 팩을 마실 때마다 폴링을 하도록 코드를 수정하면 응답 시간이 1.2ms로 약간 증가하지만 프로그램의 실행 시간은 폴링 없는 프로그램보다 3% 정도만 더 길어진다. 불행히도 직접 실행해보고 재보는 것 말고는 얼마나 자주 폴링을 해야 하는지 알 방법이 없다.

응답 시간이 개선된 것 때문에 좋다고 생각할 수도 있지만 자신만의 동시 실행 기능을 구현하려고 할 때 부딪칠 수 있는 끔찍한 문제가 존재한다. 웹 서버 예에서 파일을 열어서 데이터를 읽는 부분이 있었다. 이 연산이 수행될 때 전체 프로그램이 멈춘다. 파일 접근에 디스크 탐색이 필요할 때는 잠재적으로 긴 시간 동안 멈출 수도 있다. 이 문제를 해결하는 유일한 방법은 추가로 비동기 파일 접근 기능을 구현하고 그것을 스케줄러에 추가하는 방법밖에 없다. 데이터베이스 질의를 수행하는 것 같은 더 복잡한 연산에 대해서 이것을 비동기 방식으로 어떻게 구현할 것인지를 생각하는 일은 복잡하다. 한 가지 방법으로 해당 작업을 별개의 스레드에서 실행하고 결과가 나오면 작업 스케줄러에 전달하는 방법이 있다. 메시지 큐 같은 것을 잘 사용하면 가능하다. 어떤 시스템에서는 비동기 I/O(유닉스에서 aio_* 함수 집합 등)를 위한 저수준 시스템 호출 기능을 제공한다. 이 책을 쓰는 동안 파이썬 라이브러리는 이런 함수에 대한 접근을 제공하지 않고 있지만 그런 기능을 제공하는 써드 파티 모듈을 찾을 수 있을 것이다. 필자가 경험한 바로는 그런 기능을 사용하면 보기보다 까다롭고 프로그램이 복잡해지기 때문에 그리 가치가 있는 일이 아닌 것 같다. 그런 문제는 스레드 관련 라이브러리에서 처리하도록 하는 것이 더 낫다.

socket

socket 모듈은 표준 BSD 소켓 인터페이스에 대한 접근을 제공한다. 유닉스 기반이긴 하지만 이 모듈은 모든 플랫폼에서 사용할 수 있다. 소켓 인터페이스는 범용으로 설계되었고 다양한 네트워크 프로토콜을 지원한다(인터넷, TIPC, 블루투스 등). 그렇지만 가장 널리 사용되는 프로토콜은 인터넷 프로토콜(IP: Internet Protocol)이다. 여기에는 TCP와 UDP가 모두 포함된다. 파이썬은 IPv4와 IPv6를 모두 지원하지만 IPv4가 훨씬 더 널리 사용된다.

이 모듈은 상대적으로 저수준 모듈로서 운영체제가 제공하는 네트워크 함수에 바로 접근할 수 있는 기능을 제공한다. 네트워크 응용 프로그램을 작성하려고 한다면 22장에서 설명할 모듈을 사용하거나 이 장의 마지막에 설명할 SocketServer 모듈을 사용하는 것이 더 쉽다.

주소 체계

socket 함수 중에서 몇 가지는 주소 체계(address family)를 기술해주어야 한다. 주소 체계는 사용할 네트워크 프로토콜을 기술한다. 다음 상수들이 이 목적에 쓰인다.

상수	설명
AF_BLUETOOTH	블루투스 프로토콜
AF_INET	IPv4 프로토콜들(TCP, UDP)
AF_INET6	IPv6 프로토콜들(TCP, UDP)
AF_NETLINK	Netlink 프로세스 사이 통신(Interprocess Communication)
AF_PACKET	링크 수준 패킷
AF_TIPC	TIPC(Transparent Inter-Process Communication) 프로토콜
AF_UNIX	유닉스 도메인 프로토콜

이 중에서 AF_INET과 AF_INET6가 표준 인터넷 연결을 나타내기 때문에 가장 널리 쓰인다. AF_BLUETOOTH는 블루투스를 지원하는 시스템에서만 사용할 수 있다(보통 임베디드 시스템). AF_NETLINK, AF_PACKET, AF_TIPC는 리눅스에서만 지원된다. AF_NETLINK는 사용자 응용 프로그램과 리눅스 커널 사이에 빠른 프로세스 사이 통신을 위해 쓰인다. AF_PACKET은 데이터 링크 계층에서 직접 작업할 때 사용한다(예를 들어, 무가공 이더넷 패킷). AF_TIPC는 리눅스 클러스터에서 고성능 IPC를 수행하는 데 사용하는 프로토콜이다(<http://tipc.sourceforge.net/>).

소켓 종류

몇몇 socket 함수에는 소켓 종류를 지정해주어야 한다. 소켓 종류는 주어진 프로토콜 체계 안에서 사용할 통신의 종류를 지정한다(스트림 또는 패킷). 다음 상수들이 이 목적에 쓰인다.

상수	설명
SOCK_STREAM	신뢰성 있는 연결 지향 바이트 스트림(TCP)
SOCK_DGRAM	데이터그램(UDP)
SOCK_RAW	무가공 소켓
SOCK_RDM	신뢰성 있는 데이터그램
SOCK_SEQPACKET	순서 있는 연결 모드로 레코드 전송

가장 널리 사용되는 소켓 종류는 인터넷 프로토콜 모음에 있는 TCP와 UDP에 대응하는 SOCK_STREAM과 SOCK_DGRAM이다. SOCK_RDM은 순서를 보장하지 않지만 신뢰성 있는 데이터그램 전송을 보장하는 UDP의 신뢰성 있는 형태이다(보낸 순서와 다르게 받을 수 있다). SOCK_SEQPACKET은 순서와 패킷 경계를 유지하면서 스트림 지향 연결을 통해 패킷을 보내는 데 사용한다. SOCK_RDM이나 SOCK_SEQPACKET은 둘 다 지원이 잘 안 되기 때문에 이식성을 생각한다면 사용하지 않는 것이 좋다. SOCK_RAW는 무가공 프로토콜에 대한 저수준 접근 기능을 제공하고 제어 메시지(ICMP 메시지 같은)를 보내는 등 특별한 목적의 연산을 수행할 때 사용할 수 있다. SOCK_RAW는 슈퍼유저나 관리자 권한으로 프로그램을 실행할 때만 사용할 수 있다.

모든 프로토콜 체계에서 모든 소켓 종류를 지원하는 것은 아니다. 예를 들어, 리눅스에서 패킷 스니핑을 하려고 AF_PACKET을 사용한다면 SOCK_STREAM으로 스트림 지향 연결을 맺을 수 없다. 대신 SOCK_DGRAM이나 SOCK_RAW를 사용해야 한다. AF_NETLINK 소켓에 대해서는 SOCK_RAW만 지원된다.

주소

소켓에 대고 통신을 하려면 목적지 주소를 적어야 한다. 주소의 형태는 소켓 주소 체계에 따라 다르다.

AF_INET(IPv4)

IPv4를 사용하는 인터넷 응용 프로그램에서는 주소는 튜플 (host, port)로 적어야

한다. 다음을 보자.

```
('www.python.org', 80)
('66.113.130.182', 25)
```

host가 빈 문자열이면 아무 주소나 된다는 뜻을 나타내는 INADDR_ANY와 동일한 의미다. 보통 아무 클라이언트나 연결할 수 있는 서버에서 소켓을 생성할 때 사용한다. host가 ‘<broadcast>’로 설정되면 소켓 API에서 INADDR_BROADCAST 상수와 같은 의미를 갖는다.

호스트 이름으로 ‘www.python.org’ 같은 것이 사용되면 파이썬은 호스트 이름을 IP 주소로 변환하려고 DNS를 사용한다. DNS가 어떻게 설정되어 있는지에 따라서 매번 다른 IP 주소를 얻을 수도 있다. 이런 일을 피하려면 ‘66.113.130.182’ 같은 무가공 IP 주소를 사용하도록 한다.

AF_INET6(IPv6)

IPv6에 대해서 주소는 4개짜리 항목을 갖는 튜플 (host, port, flowinfo, scopeid)로 지정해야 한다. IPv6에서 host와 port는 IPv4와 동일한 의미를 갖지만 숫자 형태로 IPv6 호스트 주소를 쓸 때에는 ‘FEDC:BA98:7654:3210:FEDC:BA98:7654:3210’나 ‘080A::4:1’(콜론 두 개는 해당 범위의 주소 부분을 0으로 채운다)처럼 보통 여덟 개의 콜론으로 분리된 16진수 숫자로 쓴다.

flowinfo 매개변수는 24비트 흐름 라벨(flow label), (아래쪽 24비트), 4비트 우선 순위(다음 4비트), 그리고 예약된 4비트(위쪽 4비트)로 구성되는 32비트 숫자이다. 흐름 라벨은 송신자가 라우터 쪽에서 특별한 처리를 해주기를 바랄 때만 사용한다. 나머지 경우에는 flowinfo는 0으로 설정된다.

scopeid 매개변수는 링크 지역 주소나 사이트 지역 주소로 작업할 때만 필요한 32비트 숫자이다. 링크 지역(link-local) 주소는 ‘FE80:...’로 항상 시작하며 같은 LAN 안에 있는 머신 사이에 쓰인다(라우터는 링크 지역 패킷을 포워딩하지 않는다). 이 경우 scopeid는 호스트에서 특정 네트워크 인터페이스를 식별하는 인터페이스 색인이다. 이 정보는 유닉스에서는 ‘ifconfig’ 명령으로, 윈도에서는 ‘ipv6 if’ 명령으로 볼 수 있다. 사이트 지역 주소는 항상 ‘FEC0:...’로 시작하며 같은 사이트에 있는 머신 사이에 쓰인다(예를 들어, 주어진 서브넷에 있는 모든 머신). 이 경우 scopeid는 사이트 식별 숫자가 된다.

flowinfo나 scopeid에 아무 것도 주지 않을 것이라면 IPv6 주소를 IPv4처럼 (host,

port) 튜플로 주어도 된다.

AP_UNIX

유닉스 도메인 소켓에 대해서 주소는 경로 이름을 담은 문자열이다. 예를 들어, '/tmp/myserver'처럼.

AP_PACKET

리눅스 패킷 프로토콜에 대해서 주소는 튜플 (device, protonum [, pkttype [, hatype [, addr]]])로 표현된다. 여기서 device는 “eth0” 같은 장치 이름을 담은 문자열이고 protonum은 <linux/if_ether.h> 헤더 파일에 정의된 이더넷 프로토콜 번호를 나타내는 정수이다(예를 들어, IP 패킷은 0x0800). packet_type은 패킷 종류를 나타내는 정수이고 다음 상수 중에 하나가 될 수 있다.

상수	설명
PACKET_HOST	지역 호스트 패킷 주소
PACKET_BROADCAST	물리 계층 방송 패킷
PACKET_MULTICAST	물리 계층 멀티캐스트
PACKET_OTHERHOST	다른 호스트로 가는 패킷이지만 무차별 모드(promiscuous mode)에서 장치 드라이버가 잡아낸 패킷
PACKET_OUTGOING	머신에서 만든 패킷이지만 패킷 소켓으로 루프백(loopback) 된 것

hatype은 ARP 프로토콜에서 사용하는 <linux/if_arp.h> 헤더 파일에 정의되어 있는 하드웨어 주소 종류를 나타내는 정수이다. addr는 hatype에 따라서 다른 구조를 갖는 하드웨어 주소를 담은 바이트 문자열이다. 이더넷에 대해서 addr는 하드웨어 주소를 담은 6바이트 문자열이다.

AF_NETLINK

리눅스 Netlink 프로토콜에 대해서 주소는 튜플 (pid, groups)이다. 여기서 pid와 groups는 모두 부호 없는 정수이다. pid는 소켓의 유니캐스트 주소이고 보통 소켓을 생성한 프로세스의 프로세스 ID와 같거나 커널일 경우 0이다. groups는 참여 할 멀티캐스트 그룹들을 지정하는 데 사용하는 비트 마스크이다. 더 많은 정보는 Netlink 문서를 참고하기 바란다.

AF_BLUETOOTH

블루투스 주소는 사용되는 프로토콜에 따라서 다르다. L2CAP에 대해서 주소는 튜플 (addr, psm)이고 여기서 addr는 ‘01:23:45:67:89:ab’ 같은 문자열이고 psm은 부호 없는 정수이다. RFCOMM에 대해서 주소는 튜플 (addr, channel)이고 addr는 주소 문자열이고 channel은 정수이다. HCI에 대해서 주소는 항목 하나짜리 튜플인 (deviceno,)이고 deviceno는 정수 장치 번호이다. SCO에 대해서 주소는 문자열 host이다.

상수 BDADDR_ANY는 아무 주소나 나타내고 문자열 ‘00:00:00:00:00:00’이다. 상수 BDADDR_LOCAL은 문자열 ‘00:00:00:ff:ff:ff’이다.

AF_TIPC

TIPC 소켓에 대해서 주소는 튜플 (addr_type, v1, v2, v3 [, scope])이고 모든 필드는 부호 없는 정수이다. addr_type은 다음 값들 중에 하나가 될 수 있고 v1, v2, v3 값은 결정한다.

주소 종류	설명
TIPC_ADDR_NAMESEQ	v1은 서버 종류, v2는 포트 식별자, v3는 0
TIPC_ADDR_NAME	v1은 서버 종류, v2는 아래 포트 번호, v3는 위 포트 번호
TIPC_ADDR_ID	v1은 노드, v2는 참조, v3는 0

옵션인 scope 필드는 TIPC_ZONE_SCOPE, TIPC_CLUSTER_SCOPE, TIPC_NODE_SCOPE 중 하나가 될 수 있다.

함수

socket 모듈에는 다음 함수들이 들어 있다.

`create_connection(address [, timeout])`

address로 SOCK_STREAM 연결을 생성하고 연결된 소켓 객체를 반환한다. address는 (host, port) 형태의 튜플이고 timeout은 옵션인 타임아웃 시간을 지정한다. 이 함수는 먼저 getaddrinfo()를 호출한 다음 반환되는 각 튜플에 연결하려고 시도한다.

`fromfd(fd, family, socktype [, proto])`

정수 파일 기술자 fd에서 소켓 객체를 생성한다. 주소 체계(family), 소켓 종류

(socktype), 프로토콜 번호(proto)는 socket()의 것과 같다. 파일 기술자는 이전에 생성된 소켓을 가리켜야 한다. SocketType의 인스턴스를 반환한다.

```
getaddrinfo(host, port [, family [, socktype [, proto [, flags]]]])
```

어떤 호스트의 host와 port 정보가 주어졌을 때 이 함수는 소켓 연결을 여는 데 필요한 정보를 담은 튜플들의 리스트를 반환한다. host는 호스트 이름이나 숫자 IP 주소를 담은 문자열이다. port는 서비스 이름을 나타내는 숫자나 문자열이다(예를 들어, “http”, “ftp”, “smtp”). 반환되는 각 튜플은 다섯 개 항목짜리 튜플 (family, socktype, proto, canonname, sockaddr)이다. family, socktype, proto는 socket() 함수에 전달하는 것과 의미가 같다. canonname은 호스트의 정식 이름(canonical name)을 나타내는 문자열이다. sockaddr는 인터넷 주소에 관해서 다룬 앞 절에서 설명한 대로 소켓 주소를 담은 튜플이다. 다음 예를 보자.

```
>>> getaddrinfo("www.python.org", 80)
[(2, 2, 17, "('194.109.137.226', 80)), (2, 1, 6, "('194.109.137.226', 80))]
```

이 예에서 getaddrinfo()는 두 가지 소켓 연결 정보를 반환했다. 첫 번째 것 (proto=17)은 UDP 연결이고 두 번째 것(proto=6)은 TCP 연결이다. getaddrinfo()에 추가 매개변수를 주어 선택을 더 좁힐 수 있다. 예를 들어, 다음 예는 IPv4 TCP 연결 생성과 관련된 정보를 반환한다.

```
>>> getaddrinfo("www.python.org", 80, AF_INET, SOCK_STREAM)
[(2, 1, 6, "('194.109.137.226', 80))]
```

특별한 상수인 AF_UNSPEC을 주소 체계로 지정하면 아무 연결이나 찾는다. 예를 들어, 다음 코드는 아무 TCP 같은 연결에 대한 정보를 얻는 것으로서 IPv4 또는 IPv6에 대한 정보를 반환할 수 있다.

```
>>> getaddrinfo("www.python.org", "http", AF_UNSPEC, SOCK_STREAM)
[(2, 1, 6, "('194.109.137.226', 80))]
```

getaddrinfo()는 아주 범용적으로 사용되도록 만들어졌고 모든 지원되는 네트워크 프로토콜에 대해서 사용할 수 있다(IPv4, IPv6 등). 호환성에 신경을 쓰고 있고 미래에 나올 프로토콜을 지원할 예정이라면 이 함수를 사용하도록 한다. 특히 IPv6 을 지원할 것이라면 더욱 그렇다.

getdefaulttimeout()

기본 소켓 타임아웃 시간을 초 단위로 반환한다. None 값은 타임아웃이 설정되지 않았다는 것을 나타낸다.

getfqdn([name])

name의 완전히 한정된 도메인 이름을 반환한다. name을 생략하면 지역 머신을 가정한다. 예를 들어, getfqdn("foo")는 "foo.quasievil.org" 같은 것을 반환할 수 있다.

gethostbyname(hostname)

'www.python.org' 같은 호스트 이름을 IPv4 주소로 변환한다. IP 주소는 '132.151.1.90' 같은 문자열로 반환된다. IPv6는 지원하지 않는다.

gethostbyname_ex(hostname)

호스트 이름을 IPv4 주소로 변환하지만 튜플 (hostname, aliaslist, ipaddrlist)을 반환한다. 여기서 hostname은 주 호스트 이름을, aliaslist는 같은 주소의 다른 호스트 이름 목록을, ipaddrlist는 같은 호스트의 같은 인터페이스에 대한 IPv4 주소 목록이다. 예를 들어, gethostbyname_ex('www.python.org')는 ('fang.python.org', ['www.python.org'], ['194.109.137.226']) 같은 것을 반환할 수 있다. 이 함수는 IPv6를 지원하지 않는다.

gethostname()

지역 머신의 호스트 이름을 반환한다.

gethostbyaddr(ip_address)

'132.151.1.90' 같은 IP 주소가 주어질 때 gethostbyname_ex()와 같은 정보를 반환한다. ip_address가 'FEDC:BA98:7654:3210:FEDC:BA98:7654:3210'처럼 IPv6 주소이면 IPv6에 관한 정보를 반환한다.

getnameinfo(address, flags)

소켓 주소 address가 주어질 때 이 함수는 이 주소를 flags 값에 따라서 (host, port) 튜플로 변환한다. address 매개변수는 주소를 기술하는 튜플이다. 예를 들어, ('www.python.org', 80). flags는 다음 상수들을 비트 OR한 것이다.

상수

NI_NODNAME

설명

지역 호스트에 대해서 완전히 한정된 이름을 사용하지 않는다.

NI_NUMERICHOST	주소를 숫자 형식으로 반환한다.
NI_NAMEREQD	호스트 이름을 주어야 한다. 주소에 대응하는 DNS 항목이 없으면 에러를 반환한다.
NI_NUMERICSERV	포트 번호를 담은 문자열로 포트를 반환한다.
NI_DGRAM	검색할 서비스가 TCP(기본)가 아니라 데이터그램(UDP)이라는 것을 기술한다.

이 함수는 주로 어떤 주소에 대해서 추가 정보를 얻는 데 사용한다. 다음 예를 보자.

```
>>> getnameinfo(('194.109.137.226', 80), 0)
('fang.python.org', 'http')
>>> getnameinfo(('194.109.137.226', 80), NI_NUMERICSERV)
('fang.python.org', '80')
```

getprotobyname(protocolname)

socket() 함수의 세 번째 인수로 전달할 수 있도록 인터넷 프로토콜 이름('icmp' 같은)을 프로토콜 번호(IPPROTO_ICMP 같은 값)로 변환한다. 프로토콜 이름을 식별할 수 없으면 socket.error를 발생시킨다. 보통 이 함수는 무가공 소켓에 사용한다.

getservbyname(servicename [, protocolname])

인터넷 서비스 이름과 프로토콜 이름을 해당 서비스의 포트 번호로 변환한다. 예를 들어, getservbyname('ftp', 'tcp')은 21를 반환한다. 프로토콜 이름을 줄 경우 'tcp'나 'udp' 중 하나여야 한다. servicename이 알려진 서비스에 매칭이 안 되면 socket.error를 발생시킨다.

getservbyport(port [, protocolname])

getservbyname()의 반대. 포트 번호 port가 주어질 때 이 함수는 있을 경우 서비스 이름을 담은 문자열을 반환한다. 예를 들어, getservbyport(21, 'tcp')는 'ftp'를 반환한다. 프로토콜 이름을 줄 때는 'tcp'나 'udp' 중 하나여야 한다. port에 대응하는 서비스 이름이 없으면 socket.error를 발생시킨다.

has_ipv6

IPv6를 지원하면 True가 되는 불리언 상수.

htonl(x)

32비트 정수를 호스트 바이트 순서에서 네트워크 바이트 순서(빅 엔디안)로 변환한다.

htons(x)

16비트 정수를 호스트 바이트 순서에서 네트워크 바이트 순서(빅 엔디안)로 변환 한다.

inet_aton(ip_string)

문자열로 주어진 IPv4 주소(예를 들어, ‘135.128.11.209’)를 무가공 인코딩 주소로 사용하기 위한 목적으로 32비트 채운 이진 형식으로 변환한다. 반환되는 값은 이진 인코딩 값을 담은 네 글자 문자열이다. 주소를 C에 전달하거나 다른 프로그램에 전달하기 위해 데이터 구조에 채워넣을 때 유용하게 쓸 수 있다. IPv6는 지원하지 않는다.

inet_ntoa(packedip)

채워진 이진 IPv4 주소를 표준 점 형식 문자열(예를 들어, ‘135.128.11.209’)로 변환한다. packedip은 IP 주소의 무가공 32비트 인코딩 값을 담은 네 글자 문자열이다. 이 함수는 C에서 주소를 받았거나 데이터 구조에서 주소 값을 추출할 때 유용하게 쓸 수 있다. IPv6는 지원하지 않는다.

inet_ntop(address_family, packed_ip)

IP 네트워크 주소를 나타내는 채워진 이진 문자열 packed_ip을 ‘123.45.67.89’ 같은 문자열로 변환한다. address_family는 주소 체계를 나타내고 보통 AF_INET이나 AF_INET6이다. 이 함수는 무가공 바이트 버퍼에서 네트워크 주소 문자열을 얻는 데 사용할 수 있다(예를 들어, 저수준 네트워크 패킷의 내용으로부터).

inet_pton(address_family, ip_string)

‘123.45.67.89’ 같은 IP 주소를 채워진 바이트 문자열로 변환한다. address_family는 주소 체계를 나타내고 보통 AF_INET이나 AF_INET6이다. 네트워크 주소를 무가공 이진 데이터 패킷으로 인코딩하려고 할 때 사용할 수 있다.

ntohl(x)

32비트 정수를 네트워크 바이트 순서(빅 엔디안)에서 호스트 바이트 순서로 변환 한다.

ntohs(x)

16비트 정수를 네트워크 바이트 순서(딕 엔디안)에서 호스트 바이트 순서로 변환 한다.

setdefaulttimeout(timeout)

새로 생성되는 소켓 객체의 기본 타임아웃 값을 설정한다. timeout은 초 단위인 부동 소수점 수이다. None 값은 타임아웃을 설정하지 않는다는 것을 의미한다(기본 값이기도 하다).

socket(family, type [, proto])

주어진 주소 체계, 소켓 종류, 프로토콜 번호로 새 소켓을 생성한다. family는 주소 체계를 나타내고 type은 이 절의 시작 부분에서 논의한 소켓 종류를 나타낸다. TCP 연결을 열고 싶으면 socket(AF_INET, SOCK_STREAM)을 사용한다. UDP 연결을 열고 싶으면 socket(AF_INET, SOCK_DGRAM)을 사용한다. 이 함수는 SocketType 인스턴스(곧 설명한다)를 반환한다.

프로토콜 번호는 보통 생략한다(기본은 0). 보통 무가공 소켓(SOCK_RAW)에 대해서 사용하고 주소 체계에 따라서 특정 상수 값으로 설정한다. 다음 목록은 호스트 시스템에서 사용할 수 있는지 여부에 따라서 AF_INET과 AF_INET6에 대해 파일에서 정의하고 있을 만한 모든 프로토콜 번호를 보여준다.

상수	설명
IPPROTO_AH	IPv6 인증 헤더
IPPROTO_BIP	Banyan VINES
IPPROTO_DSTOPTS	IPv6 목적지 옵션
IPPROTO_EGP	외부 게이트웨이 프로토콜
IPPROTO_EON	ISO CNLP(Connectionless Network Protocol)
IPPROTO_ESP	IPv6 보안 페이로드 캡슐화(encapsulating security payload)
IPPROTO_FRAGMENT	IPv6 단편화(fragmentation) 헤더
IPPROTO_GGP	게이트웨이 대 게이트웨이 프로토콜(Gateway to Gateway Protocol)(RFC823)
IPPROTO_GRE	일반 라우팅 캡슐화(Generic Routing Encapsulation)(RFC1701)
IPPROTO_HELLO	Fuzzball HELLO 프로토콜
IPPROTO_HOPOPTS	IPv6 흡별(hop-by-hop) 옵션
IPPROTO_ICMP	IPv4 ICMP
IPPROTO_ICMPV6	IPv6 ICMP
IPPROTO_IDP	XNS IDP
IPPROTO_IGMP	그룹 관리 프로토콜
IPPROTO_IP	IPv4
IPPROTO_IPCOMP	IP 페이로드 압축 프로토콜
IPPROTO_IPIP	IP 내부(inside) IP
IPPROTO_IPV4	IPv4 헤더
IPPROTO_IPV6	IPv6 헤더

IPPROTO_MOBILE	IP 이동성(Mobility)
IPPROTO_ND	Netdisk <u>프로토콜</u>
IPPROTO_NONE	IPv6 다음 헤더 없음
IPPROTO_PIM	프로토콜 독립 멀티캐스트(Protocol Independent Multicast)
IPPROTO_PUP	제록스 PARC 보편 패킷(Xerox PARC Universal Packet)(PUP)
IPPROTO_RAW	무가공 IP 패킷
IPPROTO_ROUTING	IPv6 라우팅 헤더
IPPROTO_RSVP	자원 예약
IPPROTO_TCP	TCP
IPPROTO_TP	OSI 전송 <u>프로토콜</u> (Transport Protocol)(TP-4)
IPPROTO_UDP	UDP
IPPROTO_VRRP	가상 라우터 중복 <u>프로토콜</u> (Virtual Router Redundancy Protocol)
IPPROTO_XTP	고속 전송 <u>프로토콜</u> (eXpress Transfer Protocol)

다음 프로토콜 번호들은 AF_BLUETOOTH에 대해서 사용한다.

상수	설명
BTPROTO_L2CAP	논리 링크 제어와 적응 <u>프로토콜</u> (Logical Link Control and Adaption Protocol)
BTPROTO_HCI	호스트/제어기 인터페이스(Host/Controller Interface)
BTPROTO_RFCOMM	케이블 교체 <u>프로토콜</u> (Cable replacement protocol)
BTPROTO_SCO	동기 접속 지향 링크(Synchronous Connection Oriented Link)

socketpair([family [, type [, proto]]])

socket() 함수의 것과 같은 의미를 갖는 family, type, proto 옵션이 주어질 때 이를 사용하여 연결된 소켓 객체 한 쌍을 생성한다. 이 함수는 유닉스 도메인 소켓에만 사용할 수 있다(family=AF_UNIX). type은 SOCK_DGRAM이나 SOCK_STREAM이 될 수 있다. type이 SOCK_STREAM이면 스트림 파이프(stream pipe)라고 부르는 객체가 생성된다. proto는 보통 0(기본)이다. 이 함수는 주로 os.fork()로 생성한 프로세스 사이에 통신 채널을 설정할 때 사용된다. 예를 들어, 부모 프로세스에서 socketpair()를 호출해서 소켓 한 쌍을 만든 다음에 os.fork()를 호출한다. 그 다음에 부모와 자식 프로세스는 이 소켓을 사용하여 서로 통신한다.

소켓은 SocketType 타입 인스턴스로 표현된다. 소켓 s에는 다음 메서드들이 있다.

s.accept()

연결을 받고 튜플 (conn, address)를 반환한다. conn은 연결을 통해 데이터를 주고받는 데 사용할 새로운 소켓 객체이고 address는 상대방의 주소이다.

s.bind(address)

소켓을 특정 주소에 묶는다. address의 형식은 주소 체계에 따라 다르다. 대부분의 경우 (hostname, port) 형식의 튜플이 된다. IP 주소에 대해서 빈 문자열은 INADDR_ANY를 나타내고 문자열 ‘<broadcast>’는 INADDR_BROADCAST를 나타낸다. INADDR_ANY 호스트 이름(빈 문자열)은 서버가 시스템에 있는 아무 인터넷 인터페이스로 연결이 들어오는 것을 허용하겠다는 의미이다. 종종 서버가 멀티홈(multihomed) 형태로 작동할 때 쓰인다. INADDR_BROADCAST 호스트 이름(‘<broadcast>’)은 방송 메시지를 보내기 위해서 소켓을 사용할 때 쓰인다.

s.close()

소켓을 닫는다. 소켓은 쓰레기 수집될 때에도 닫힌다.

s.connect(address)

address에 있는 원격 소켓에 연결한다. address의 형식은 주소 체계에 따라 다를 수 있지만 보통은 (hostname, port) 튜플로 표현한다. 에러가 발생하면 socket.error를 발생시킨다. 같은 컴퓨터에 있는 서버에 연결할 때는 hostname에 ‘localhost’라고 써도 된다.

s.connect_ex(address)

connect(address)와 같지만 성공하면 0을 반환하고 실패하면 errno의 값을 반환한다.

s.fileno()

소켓 파일 기술자를 반환한다.

s.getpeername()

소켓이 연결된 원격 주소를 반환한다. 보통 (ipaddr, port) 튜플이지만 주소 체계에 따라 달라질 수 있다. 모든 시스템에서 지원하는 것은 아니다.

s.getsockname()

소켓 자체의 주소를 반환한다. 보통 (ipaddr, port) 튜플로 반환된다.

s.getsockopt(level, optname [, buflen])

소켓 옵션 값을 반환한다. level은 옵션의 수준을 정의하며, 소켓 수준 옵션에 대해서는 SOL_SOCKET이고 프로토콜과 연관된 옵션에 대해서는 IPPROTO_IP처럼

프로토콜 번호이다. optname은 특정 옵션을 선택한다. buflen을 생략하면 옵션을 정수 값으로 가정하고 정수 값을 반환한다. buflen가 주어질 경우 옵션을 받아올 버퍼의 최대 길이를 나타낸다. 이 버퍼는 바이트 문자열을 담는다. 이 내용을 struct 모듈이나 기타 다른 방법으로 디코딩하는 일은 호출자 쪽에서 해야 한다.

다음 표는 파이썬에서 정의되어 있는 소켓 옵션 목록을 보여준다. 이 옵션들은 대부분 고급 소켓 API의 일부로 취급되는 것으로 네트워크의 저수준 세부 사항을 제어한다. 더 자세한 설명을 원하면 다른 문서를 찾아보아야 할 것이다. 값 열에 타입 이름이 써 있으면 그 이름은 값에 연결된 표준 C 데이터 구조의 이름과 같으며 표준 소켓 프로그래밍 인터페이스에서 사용되는 것이다. 모든 옵션을 모든 머신에서 사용할 수 있는 것은 아니다.

다음은 SOL_SOCKET 수준에 대해서 자주 사용되는 옵션 이름을 나열한 것이다.

옵션 이름	값	설명
SO_ACCEPTCONN	0, 1	소켓이 연결을 받을 수 있는지 여부를 결정
SO_BROADCAST	0, 1	방송 데이터그램 전송을 허용
SO_DEBUG	0, 1	디버깅 정보를 기록할 것인지를 결정
SO_DONTROUTE	0, 1	라우팅 테이블(routing table) 검색을 생략함
SO_ERROR	int	에러 정보를 얻음
SO_EXCLUSIVEADDRUSE	0, 1	다른 소켓이 같은 주소와 포트에 묶이는 것을 막음. 이 옵션은 SO_REUSEADDR 옵션을 비활성화한다.
SO_KEEPALIVE	0, 1	주기적으로 연결의 다른 쪽 끝을 살펴보고 반열림(half-open) 상태이면 연결을 종료한다.
SO_LINGER	linger	송신 버퍼에 데이터가 있으면 close()하기 전에 기다린다. linger는 두 개의 32비트 정수(onoff, seconds)를 담은 채워진 이진 문자열이다.
SO_OOBINLINE	0, 1	아웃오브밴드(out-of-band) 데이터를 입력 큐에 넣는다.
SO_RCVBUF	int	수신 버퍼의 크기(바이트로)
SO_RCVLOWAT	int	select()가 소켓을 읽을 수 있는 것으로 반환하기 전에 읽을 바이트 수
SO_RCVTIMEO	timeval	수신 호출에서 초 단위 타임아웃. timeval은 두 개의 32 비트 정수(seconds, microseconds)를 담은 채워진 이진 문자열
SO_REUSEADDR	0, 1	지역 주소 재사용을 허용
SO_REUSEPORT	0, 1	이 소켓 옵션이 모든 프로세스에서 설정되어 있다면 여러 프로세스가 같은 주소에 묶일 수 있다.
SO_SNDBUF	int	송신 버퍼 크기(바이트로)
SO_SNDLOWAT	int	select()가 소켓을 쓸 수 있다고 반환하기 전에 송

SO_SNDFTIMEO	timeval	신 버퍼에 들어 있어야 하는 바이트 수. 송신 호출에서 초 단위 타임아웃. timeval 설명은 SO_RCVTIMEO 참고
SO_TYPE	int	소켓 타입을 얻음
SO_USELOOPBACK	0, 1	라우팅 소켓이 자신이 보내는 것의 복사본을 얻음

다음 옵션들은 IPPROTO_IP 수준에서 사용할 수 있다.

옵션 이름	값	설명
IP_ADD_MEMBERSHIP	ip_mreg	멀티캐스트 그룹(설정만 가능)에 참여. ip_mreg는 두 개의 32비트 IP 주소(multiaddr, localaddr)를 담은 채워진 이진 문자열. multiaddr는 멀티캐스트 주소이고 localaddr는 사용될 지역 인터페이스의 IP이다.
IP_DROP_MEMBERSHIP	ip_mreg	멀티캐스트 그룹을 떠남(설정만 가능). ip_mreg는 이전과 동일.
IP_HDRINCL	int	데이터에 IP 헤더도 포함
IP_MAX_MEMBERSHIPS	int	최대 멀티캐스트 그룹 수
IP_MULTICAST_IF	in_addr	나가는 인터페이스. in_addr는 32비트 IP 주소를 담은 채워진 이진 문자열.
IP_MULTICAST_LOOP	0, 1	루프백
IP_MULTICAST_TTL	uint8	생존 시간(time to live). unit8은 1바이트 부호 없는 char를 담은 채워진 이진 문자열.
IP_OPTIONS	ipopts	IP 헤더 옵션들. ipopts는 44바이트를 넘지 않는 채워진 이진 문자열. 이 문자열의 내용은 RFC 791에 설명되어 있다.
IP_RECVDSTADDR	0, 1	IP 목적지 주소를 데이터그램으로 받음
IP_RECVOPTS	0, 1	모든 IP 옵션을 데이터그램으로 받음
IP_RECVRETOPTS	0, 1	IP 옵션들을 응답으로 받음
IP_RETOPTS	0, 1	IP_RECVOPTS와 같지만, 타임스탬프나 경로 기록 옵션(route record option)을 채우지 않은 채로 옵션을 처리하지 않고 놔둔다.
IP_TOS	int	서비스 종류
IP_TTL	int	생존 시간

다음 옵션들은 IPPROTO_IPV6 수준에서 사용할 수 있다.

옵션 이름	값	설명
IPV6_CHECKSUM	0, 1	시스템이 체크섬(checksum)을 계산하게 한다.
IPV6_DONTFRAG	0, 1	MTU 크기를 넘길 경우 패킷을 조각내지 않는다.
IPV6_DSTOPTS	ip6_dest	목적지 옵션. ip6_dest는 (next, len, options) 형태인 채워진 이진 문자열이다. next는 다음 헤더 옵션

IPV6_HOPLIMIT	int	타입을 주는 8비트 정수이다. len은 헤더의 크기를 첫 8바이트를 제외하고 8바이트 단위로 기술하는 8비트 정수이다. options는 인코딩된 옵션들이다.
IPV6_HOPTS	ip6_hbh	홉(hop) 제한 홉별(hop-by-hop) 옵션. ip6_hbh는 ip6_dest와 같은 인코딩을 갖는다.
IPV6_JOIN_GROUP	ip6_mreq	멀티캐스트 그룹에 참여한다. ip6_mreg는 (multiaddr, index)를 담은 채워진 이진 문자열이다. multiaddr는 128비트 IPv6 멀티캐스트 주소이고 index는 지역 인터페이스에 대한 32비트 부호 없는 인터페이스 색인이다.
IPV6_LEAVE_GROUP	ip6_mreg	멀티캐스트 그룹을 떠난다
IPV6_MULTICAST_HOPS	int	멀티캐스트 패킷의 흡 제한
IPV6_MULTICAST_IF	int	나가는 멀티캐스트 패킷에 대한 인터페이스 색인
IPV6_MULTICAST_LOOP	0, 1	나가는 멀티캐스트 패킷을 지역 응용 프로그램에 다시 전달한다.
IPV6_NEXTHOP	sockaddr_in6	나가는 패킷의 다음 흡 주소를 설정한다. sockaddr_in6는 보통 <netinet/in.h>에 정의되어 있는 C의 sockaddr_in6를 담은 채워진 이진 문자열이다.
IPV6_PKTINFO	ip6_pktinfo	패킷 정보 구조. ip6_pktinfo는 (addr, index)를 담은 채워진 이진 문자열이다. addr는 128비트 IPv6 주소이고 index는 인터페이스 색인에 대한 32비트 부호 없는 정수이다.
IPV6_RECVDSTOPTS	0, 1	목적지 옵션들을 받는다.
IPV6_RECVHOPLIMIT	0, 1	홉 제한을 받는다.
IPV6_RECVHOPOPTS	0, 1	홉별 옵션들을 받는다.
IPV6_RECVPKTINFO	0, 1	패킷 정보를 받는다.
IPV6_RECVRTHDR	0, 1	라우팅 헤더를 받는다.
IPV6_RECVTCLASS	0, 1	트래픽 클래스(traffic class)를 받는다.
IPV6_RTHDR	ip6_rthdr	라우팅 헤더. ip6_rthdr는 (next, len, type, segleft, data)를 담은 채워진 이진 문자열이다. next, len, type, segleft는 모두 8비트 부호 없는 정수이고 data는 라우팅 데이터이다. RFC 2460을 참고.
IPV6_RTHDRDSTOPTS	ip6_dest	라우팅 옵션 헤더 앞에 있는 목적지 옵션 헤더 IPV6_PATHMTU 보조 데이터 항목 받기를 활성화한다.
IPV6_RECVPATHMTU	0, 1	
IPV6_TCLASS	int	트래픽 클래스
IPV6_UNICAST_HOPS	int	유니캐스트 패킷의 흡 제한
IPV6_USE_MIN_MTU	-1, 0, 1	경로 MTU 발견. 1은 모든 목적지에 대해서 비활성화시킨다. -1은 멀티캐스트 목적지에 대해서만

IPV6_V6ONLY	0, 1	비활성화시킨다. IPv6 노드에만 연결한다.
--------------------	------	-----------------------------

다음 옵션들은 SOL_TCP 수준에 대해서 사용할 수 있다.

옵션 이름	값	설명
TCP_CORK	0, 1	설정될 경우 불완전한 프레임을 내보내지 않는다.
TCP_DEFER_ACCEPT	0, 1	데이터가 소켓에 도달했을 때만 리스너를 깨운다.
TCP_INFO	tcp_info	소켓에 대한 정보를 담은 구조를 반환한다. tcp_info는 구현마다 다를 수 있다.
TCP_KEEPCNT	int	연결을 포기하기 전에 TCP가 보낼 최대 살아있음(keepalive) 검사 메시지 수
TCP_KEEPIDLE	int	TCP_KEEPALIVE 옵션이 설정된 경우 TCP가 살아있음 검사 메시지를 보내기 시작하기 전에 연결을 유휴 상태로 둘 시간(초)
TCP_KEEPINTVL	int	살아있음 검사 사이에 시간(초)
TCP_LINGER2	int	고아가 된 FIN_WAIT2 상태인 소켓의 수명
TCP_MAXSEG	int	나가는 TCP 패킷의 최대 세그먼트 크기
TCP_NODELAY	0, 1	설정할 경우 네이글(Nagle) 알고리즘을 비활성화한다.
TCP_QUICKACK	0, 1	설정할 경우 ACK가 즉시 보내진다. TCP 지연 ACK 알고리즘을 비활성화한다.
TCP_SYNCNT	int	연결 요청을 취소하기 전까지 SYN 재전송 수
TCP_WINDOW_CLAMP	int	광고할 TCP 윈도 크기에 상한을 설정한다.

s.gettimeout()

있을 경우 현재 타임아웃 값을 반환한다. 초 단위인 부동 소수점 수를 반환하거나 타임아웃이 설정 안 된 경우 None을 반환한다.

s.ioctl(control, option)

윈도에서 WSAIoctl 인터페이스에 대한 제한된 접근을 제공한다. control에 지원되는 유일한 값은 네트워크에서 받은 모든 IP 패킷을 수집하는 SIO_RCVALL이다. 관리자 접근 권한이 필요하다. option으로는 다음 값을 사용할 수 있다

옵션	설명
RCVALL_OFF	소켓이 모든 IPv4 패킷 및 IPv6 패킷을 받는 것을 막는다.
RCVALL_ON	무차별 모드를 활성화한다. 소켓이 네트워크에 있는 모든 IPv4와 IPv6 패킷을 받을 수 있게 한다. ARP 같은 다른 네트워크 프로토콜에 연관된 패킷은 수집하지 않는다.
RCVALL_IPELEVEL	네트워크에 있는 모든 IP 패킷을 받지만 무차별 모드를 활성화하지는 않는다. 설정된 아무 IP 주소에 대해서 이 호스트로 보내진 모든

IP 패킷을 수집한다.

s.listen(backlog)

들어오는 연결을 듣기 시작한다. backlog는 연결을 거부하기 전에 운영체제에서 최대 기다리고 있을 연결 수를 지정한다. 이 값은 적어도 1 이상이어야 하고 대부분의 경우 5면 충분하다.

s.makefile([mode [, bufsize]])

소켓에 연결된 파일 객체를 만든다. mode와 bufsize는 내장 open() 함수에서의 미와 같다. 파일 객체는 os.dup()를 통해 생성된 소켓 파일 기술자의 복제본을 사용하기 때문에 파일 객체와 소켓 객체를 서로 독립적으로 닫거나 쓰레기 수집할 수 있다. 소켓 s에는 타입아웃이나 넌블로킹 모드를 설정해서는 안 된다.

s.recv(bufsize [, flags])

소켓에서 데이터를 받는다. 데이터는 문자열로 반환된다. bufsize는 받을 최대 데이터 양을 지정한다. flags는 메시지에 관한 정보를 제공하고 보통은 생략한다(기본 값은 0). 지정할 경우 보통 다음 상수 중 하나로 설정한다(시스템마다 다를 수 있다).

상수	설명
MSG_DONTROUTE	라우팅 표 검색을 건너뛴다(보낼 때만).
MSG_DONTWAIT	넌블로킹으로 작동한다.
MSG_EOR	메시지가 레코드의 마지막 메시지라는 것을 가리킨다. 보통 데이터를 SOCK_SEQPACKET 소켓으로 보낼 때만 사용한다.
MSG_PEEK	데이터를 살펴보지만 버리지는 않는다(받을 때만).
MSG_OOB	아웃오브밴드 데이터를 받거나 보낸다.
MSG_WAITALL	요청한 수만큼의 바이트를 받을 때까지 반환하지 않는다(받을 때만).

s.recv_into(buffer [, nbytes [, flags]])

데이터가 버퍼 인터페이스를 지원하는 객체 buffer에 써진다는 것 말고는 recv()와 같다. nbytes는 받을 최대 바이트 수를 나타낸다. 생략하면 버퍼 크기에서 최대 크기를 얻는다. flags는 recv()에서와 같은 의미를 갖는다.

s.recvfrom(bufsize [, flags])

반환되는 값이 (data, address)라는 것 말고는 recv()와 같다. data는 받은 데이터를

담은 문자열이고 address는 데이터를 보낸 소켓의 주소이다. 옵션인 flags는 recv()에서와 같은 의미를 갖는다. 이 함수는 주로 UDP 프로토콜을 사용할 때 쓴다.

s.recvfrom_info(buffer [, nbytes [, flags]])

받은 데이터를 버퍼 객체 buffer에 저장한다는 것 말고는 recvfrom()과 같다. nbytes는 받을 최대 바이트 수를 나타낸다. 생략하면 buffer의 크기에서 최대 크기를 얻는다. flags는 recv()에서와 의미가 같다.

s.send(string [, flags])

string에 있는 데이터를 연결된 소켓으로 보낸다. 옵션인 flags 인수는 앞에서 설명하였듯이 recv()에서 의미와 같다. 보낸 바이트 수를 반환하고 이 값은 string에 있는 바이트 수보다 작을 수 있다. 예러가 발생하면 예외를 발생시킨다.

s.sendall(string [, flags])

반환하기 전에 모든 데이터를 보내려고 시도하면서 string에 있는 데이터를 연결된 소켓으로 보낸다. 성공할 경우 None을 반환한다. 실패할 경우 예외를 발생시킨다. flags는 send()에서와 의미가 같다.

s.sendto(string [, flags], address)

데이터를 소켓으로 보낸다. flags는 recv()에서와 의미가 같다. address는 (host, port) 형태의 튜플이고 원격 주소를 나타낸다. 소켓이 이미 연결되어 있으면 안 된다. 보낸 바이트 수를 반환한다. 이 함수는 주로 UDP 프로토콜과 함께 쓰인다.

s.setblocking(flag)

flag가 0이면 소켓이 넌블로킹 모드로 설정된다. 아니면 소켓은 블로킹 모드로 설정된다(기본). 넌블로킹 모드에서 recv() 호출이 데이터를 찾지 못하거나 send() 호출이 데이터를 즉시 보내지 못할 경우 socket.error 예외가 발생한다. 블로킹 모드에서는 이 두 호출이 더 진행할 수 있을 때까지 멈추어 있다.

s.setsockopt(level, optname, value)

주어진 소켓 옵션 값을 설정한다. level과 optname은 getsockopt()에서와 의미가 같다. value는 정수이거나 버퍼의 내용을 나타내는 문자열일 수 있다. 후자의 경우 호출하는 쪽에서 문자열에 데이터를 적절히 만들어야 한다. 소켓 옵션의 이름과 값, 그리고 자세한 설명은 getsockopt()를 참고한다.

s.settimeout(timeout)

소켓 연산에 타임아웃을 설정한다. timeout은 초 단위인 부동 소수점 수다. None 값은 타임아웃을 설정하지 않겠다는 의미이다. 타임아웃이 발생하면 socket.timeout 예외가 발생한다. 일반적으로 타임아웃은 소켓을 생성하자마자 설정해야 한다. 그래야 연결을 생성하는 연산(connect()처럼) 같은 곳에 적용되기 때문이다.

s.shutdown(how)

연결의 한쪽 또는 양쪽을 종료한다. how가 0이면 더 이상 받을 수 없다. how가 1이면 더 이상 보낼 수 없다. how가 2면 보내기 받기 모두 안 된다.

지금까지 설명한 메서드들 이외에도 소켓 인스턴스 s는 socket() 함수에 넘기는 인수들에 대응하는 다음 읽기 전용 프로퍼티들이 있다.

프로퍼티	설명
s.family	소켓 주소 체계(예를 들어, AF_INET)
s.proto	소켓 프로토콜
s.type	소켓 종류(예를 들어, SOCK_STREAM)

예외

socket 모듈에는 다음 예외들이 정의되어 있다.

error

이 예외는 소켓이나 주소 관련 에러에 대해서 발생된다. 아래쪽 시스템 호출에서 반환한 에러를 담은 (errno, mesg) 쌍을 반환한다. IOError에서 상속받았다.

herror

주소 관련 에러에 대해서 발생하는 에러. 에러 번호와 에러 메시지를 담은 튜플 (herrno, hmesg)를 반환한다. error에서 상속받았다.

gaierror

getaddrinfo()와 getnameinfo() 함수에서 발생하는 주소 관련 에러에 대해서 발생한다. 에러 값은 튜플 (errno, mesg)이다. errno는 에러 번호를, mesg는 메시지를 담은 문자열을 나타낸다. errno는 socket 모듈에 정의된 다음 상수 중 한 값을 갖는다.

상수	설명
EAI_ADDRFAMILY	주소 체계가 지원되지 않음
EAI AGAIN	이름 해석을 일시적으로 실패
EAI_BADFLAGS	유효하지 않은 플래그
EAI_BADHINTS	좋지 않은 힌트
EAI_FAIL	회복할 수 없는 이름 해석 실패
EAI_FAMILY	호스트가 주소 체계를 지원하지 않음
EAI_MEMORY	메모리 할당 실패
EAI_NODATA	노드 이름과 연결된 주소가 없음
EAI_NONAME	노드 이름이나 서비스 이름이 제공되지 않음
EAI_PROTOCOL	프로토콜이 지원되지 않음
EAI_SERVICE	소켓 타입에 대해서 서비스 이름이 지원되지 않음
EAI_SOCKTYPE	소켓 타입이 지원되지 않음
EAI_SYSTEM	시스템 에러

timeout

소켓 연산이 타임아웃 되었을 때 발생할 예외. setdefaulttimeout() 함수나 소켓 객체의 settimeout() 메서드를 사용해서 타임아웃을 설정했을 때만 발생함. 예외 값은 문자열 'timeout'. error에서 상속받음.

예

TCP 연결 관련 예는 이 장의 도입부에서 살펴보았다. 다음 예는 간단한 UDP 에코 (echo) 서버를 보여준다.

```
# UDP 메시지 서버
# 아무 곳에서나 온 작은 패킷을 받아서 출력한다.
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(("","10000"))
while True:
    data, address = s.recvfrom(256)
    print("Received a connection from %s" % str(address))
    s.sendto(b"echo:" + data, address)
```

다음은 이 에코 서버에 메시지를 보내는 클라이언트이다.

```
# UDP 메시지 클라이언트
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b"Hello World", ("","10000"))
resp, addr = s.recvfrom(256)
print(resp)
s.sendto(b"Spam", ("", 10000))
```

```
resp, addr = s.recvfrom(256)
print(resp)
s.close()
```

Note

- 상수들과 소켓 옵션들을 모든 플랫폼에서 사용할 수 있는 것은 아니다. 이식성을 중요하게 생각한다면 이 절의 시작 부분에서 언급한 리차드 스티븐스가 쓴 유닉스 네트워크 프로그래밍 책 같은 주요 소스에 문서화되어 있는 옵션만을 사용하는 것이 좋다.
- socket 모듈에 포함되지 않은 것 중에서 눈에 띄는 것으로 recvmsg()와 sendmsg() 시스템 호출이 있다. 이 둘은 보통 보조적인 데이터나 패킷 헤더, 라우팅 같은 고급 네트워크 옵션을 다루는 데 사용된다. 이 기능을 사용하려면 PyXAPI(<http://pypi.python.org/pypi/PyXAPI>) 같은 써드 파티 모듈을 설치해야 한다.
- 넌블로킹 소켓 연산과 타임아웃이 걸린 연산에는 미묘한 차이가 있다. 넌블로킹 모드에서 소켓 함수를 호출했을 때 해당 연산이 막힐 경우에는 에러와 함께 즉시 반환이 이루어진다. 타임아웃이 설정된 경우에는 연산이 주어진 시간 안에 완료되지 못하면 함수가 에러를 반환하게 된다.

ssl

ssl 모듈은 데이터 암호화와 피어(peer) 인증 기능을 제공하는 SSL(Secure Sockets Layer)로 소켓을 감싸는 데 사용한다. 파이썬은 Open SSL 라이브러리(<http://www.openssl.org/>)를 사용해 이 모듈을 구현한다. SSL의 이론과 작동 방식에 관한 자세한 내용은 이곳에서 다루기 힘들다. SSL 설정, 키, 인증서 같은 논의를 할 때는 여러분이 이 내용을 이미 알고 있다고 가정할 것이며 여기서는 이 모듈을 사용할 때 알아야 하는 핵심적인 내용만 다룬다.

`wrap_socket(sock [, **opts])`

기존 소켓 `sock(socket 모듈로 생성한)`이 SSL을 지원하도록 감싼 `SSLSocket` 인스턴스를 반환한다. 이 함수는 `connect()`나 `accept()`를 호출하기 전에 사용해야 한다. `opts`는 추가 설정을 담은 키워드 인수들을 나타낸다.

키워드 인수`server_side`**설명**

소켓이 서버로 작동하는지(True) 클라이언트로 작동하는지(False)를 가리키는 불리언 플래그. 기본은 False.

`keyfile`

연결의 지역 쪽을 식별하는 데 사용할 키 파일. PEM 형

	식의 파일이어야 하고 보통 certfile로 지정한 파일이 키를 포함하고 있지 않은 경우에만 사용한다.
certfile	연결의 지역 쪽을 식별하는 데 사용할 인증서 파일. PEM 형식의 파일이어야 한다.
cert_reqs	연결의 다른 쪽에서도 인증서가 필요한지와 인증서를 검증해야 하는지를 지정한다. CERT_NONE은 인증서를 무시하고 CERT_OPTIONAL은 인증서가 필요하지 않지만 제공될 경우 검증한다는 것을 의미하고, CERT_REQUIRED는 인증서가 필수이고 검증한다는 것을 의미한다. 인증서를 검증할 때는 ca_certs 매개변수를 반드시 제공해야 한다.
ca_certs	검증을 위한 인증 기관 인증서를 담은 파일 이름
ssl_version	사용할 SSL 프로토콜 버전. PROTOCOL_TLSv1, PROTOCOL_SSLv2, PROTOCOL_SSLv3, PROTOCOL_SSLv23, PROTOCOL_SSLv3 중 하나. 기본은 PROTOCOL_SSLv3.
do_handshake_on_connect	연결할 때 자동으로 SSL 핸드셰이크(handshake)를 수행할 것인지 여부를 지정하는 불리언 플래그. 기본으로 True.
suppress_ragged_eofs	연결에 대해 예상하지 못한 EOF를 받았을 때 read()에서 어떻게 처리할지를 지정. True이면(기본 값) 보통의 EOF 신호를 보낸다. False이면 예외가 발생한다.

socket,socket에서 상속받은 SSLSocket의 인스턴스 s는 다음 연산들도 지원한다.

s.cipher()

튜플 (name, version, secretbits)를 반환한다. name은 사용되는 암호 이름, version은 SSL 프로토콜, secretbits는 사용되는 비밀 비트 수를 나타낸다.

s.do_handshake()

SSL 핸드셰이크를 수행한다. 보통 wrap_socket() 함수에서 do_handshake_on_connect 옵션이 False로 설정되어 있지 않는 한 자동으로 수행된다. 내부 소켓 s가 넌블로킹이면 해당 연산이 종료되지 못할 경우 SSLError 예외가 발생한다. SSLError 예외 e의 e.args[0] 속성은 수행되어야 하는 연산에 따라서 SSL_ERROR_WANT_READ나 SSL_ERROR_WANT_WRITE 값을 갖는다. 일단 읽기나 쓰기를 계속할 수 있게 되어 핸드셰이크 과정을 이어가려면 간단히 s.do_handshake()를 다시 호출하면 된다.

s.getpeercert([binary_form])

연결의 다른 쪽에 인증서가 있는 경우 이것을 반환한다. 인증서가 없으면 None을 반환한다. 인증서가 있더라도 검증이 되지 않으면 빈 사전이 반환된다. 검증된 인증서를 받으면 키로 ‘subject’와 ‘notAfter’를 갖는 사전이 반환된다. binary_form이 True로 설정되면 인증서가 DER로 인코딩된 바이트 순서열로 반환된다.

s.read([nbytes])

nbytes까지 데이터를 읽고 반환한다. nbytes를 생략하면 1,024 바이트까지 읽어서 반환한다.

s.write(data)

바이트 문자열 data를 쓴다. 써진 바이트 수를 반환한다.

s.unwrap()

SSL 연결을 닫고 이어서 암호화되지 않은 통신을 계속할 수 있는 내부 소켓 객체를 반환한다.

이 모듈에는 다음 유ти리티 함수들도 정의되어 있다.

cert_time_to_seconds(timestring)

인증서에서 사용하는 형식으로 된 문자열 timestring을 time.time() 함수와 호환되는 부동 소수점 수로 변환한다.

DER_cert_to_PEM_cert(derbytes)

DER로 인코딩된 인증서를 담은 바이트 문자열 derbytes가 주어지면 이를 PEM으로 인코딩된 문자열로 반환한다.

PEM_cert_to_DER_cert(pemstring)

인증서의 PEM으로 인코딩된 버전의 문자열인 pemstring이 주어지면 이를 DER로 인코딩된 바이트 문자열로 반환한다.

get_server_certificate(addr [, ssl_version [, ca_certs]])

SSL 서버의 인증서를 추출해서 PEM으로 인코딩한 문자열로 반환한다. addr는 (hostname, port) 형식을 갖는 주소이다. ssl_version은 SSL 버전 번호이고 ca_certs는 wrap_socket() 함수에서 설명한 것처럼 인증 기관 인증서를 담은 파일 이름이다.

RAND_status()

SSL 계층의 유사 무작위 번호 생성기가 충분한 무작위성을 갖는다고 생각하는지 여부에 따라 True나 False를 반환한다.

RAND_egd(path)

엔트로피 수집 데몬으로부터 256바이트의 무작위 값을 읽어서 이를 유사 무작위 수 생성기에 추가한다. path는 해당 데몬의 유닉스 도메인 소켓 이름을 나타낸다.

RAND_add(bytes, entropy)

바이트 문자열 bytes에 있는 바이트들을 유사 무작위 수 생성기에 추가한다. entropy는 엔트로피의 하한을 주는 음수 아닌 부동 소수점 수이다.

예

다음 예는 이 모듈을 사용해서 SSL 클라이언트 연결을 생성하는 방법을 보여준다.

```
import socket, ssl

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_s = ssl.wrap_socket(s)
ssl_s.connect(('gmail.google.com', 443))
print(ssl_s.cipher())
# 요청을 보낸다.
ssl_s.write(b"GET / HTTP/1.0\r\n\r\n")
# 응답을 받는다.
while True:
    data = ssl_s.read()
    if not data: break
    print(data)
ssl_s.close()
```

다음은 SSL로 보안이 강화된 시간 서버를 보여준다.

```
import socket, ssl, time

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("", 12345))
s.listen(5)
while True:
    client, addr = s.accept() # Get a connection
    print "Connection from", addr
    client_ssl = ssl.wrap_socket(client,
                                  server_side=True,
                                  certfile="timecert.pem")

    client_ssl.sendall(b"HTTP/1.0 200 OK\r\n")
```

```

client_ssl.sendall(b"Connection: Close\r\n")
client_ssl.sendall(b"Content-type: text/plain\r\n\r\n")
resp = time.ctime() + "\r\n"
client_ssl.sendall(resp.encode('latin-1'))
client_ssl.close()
client.close()

```

이 서버를 실행하려면 서명된 서버 인증서를 timecert.pem 파일에 넣어두어야 한다. 테스트를 위해 다음 유닉스 명령을 입력해서 하나를 만들 수 있다.

```
% openssl req -new -x509 -days 30 -nodes -out timecert.pem -keyout
timecert.pem
```

이 서버를 테스트하려면 브라우저에서 ‘<https://localhost:1234>’ 같은 주소로 연결을 시도한다. 성공할 경우 스스로 서명한 인증서를 사용하고 있다는 경고 메시지가 나타날 것이다. 동의하면 서버의 출력 결과를 볼 수 있다.

SocketServer

이 모듈은 파이썬 3에서는 socketserver로 불린다. SocketServer 모듈은 TCP, UDP, 유닉스 도메인 소켓 서버를 간단하게 구현할 수 있는 클래스들을 제공한다.

처리기

이 모듈을 사용하려면 기반 클래스인 BaseRequestHandler에서 상속을 받아서 처리기 클래스를 정의해야 한다. BaseRequestHandler의 인스턴스 h는 다음 메서드들 중 하나 이상을 구현한다.

h.finish()

handle() 메서드가 종료된 후 청소 작업을 수행하기 위해 호출된다. 기본으로 아무 일도 하지 않는다. setup()이나 handle() 메서드에서 예외를 발생시키면 이 메서드는 호출되지 않는다.

h.handle()

이 메서드는 요청을 실제로 처리하기 위해서 호출된다. 인수 없이 호출되지만 몇몇 인스턴스 변수에 유용한 정보가 담겨 있다. h.request는 요청 정보를 담고 h.client_address는 클라이언트 주소를 담으며 h.server는 처리기를 호출한 서버의 인스턴스를 담는다. TCP 같은 스트림 서비스에 대해서 h.request 속성은 소켓 객체를 담는다. 데이터그램 서비스에 대해서는 받은 데이터를 담은 바이트 문자열이 된다.

h.setup()

handle() 메서드를 호출하기 전에 초기화 작업을 수행하는 데 이 메서드가 호출된다. 기본으로 아무 일도 하지 않는다. 서버에서 SSL 연결을 수립한다거나 하는 등 추가 연결 설정을 구현하고 싶을 때 이 메서드를 구현하면 된다.

다음은 스트림이나 데이터그램을 처리할 수 있는 간단한 시간 서버를 구현한 처리기 클래스의 예를 보여준다.

```
try:
    from socketserver import BaseRequestHandler # Python 3
except ImportError:
    from SocketServer import BaseRequestHandler # Python 2
import socket
import time

class TimeServer(BaseRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        if isinstance(self.request,socket.socket):
            # 스트림 지향 연결
            self.request.sendall(resp.encode('latin-1'))
        else:
            # 데이터그램 지향 연결
            self.server.socket.sendto(resp.encode('latin-1'),
                                      self.client_address)
```

처리기에서 TCP 같은 스트림 지향 연결만 처리된다고 확실하면 BaseRequestHandler 대신 StreamRequestHandler에서 상속받으면 된다. 이 클래스에는 두 개의 속성이 있다. h.wfile은 클라이언트에 데이터를 쓰는 데 사용하는 파일 같은 객체이고 h.rfile은 클라이언트에서 데이터를 읽는 데 사용하는 파일 같은 객체이다. 다음 예를 보자.

```
try:
    from socketserver import StreamRequestHandler # 파이썬 3
except ImportError:
    from SocketServer import StreamRequestHandler # 파이썬 2
import time

class TimeServer(StreamRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        self.wfile.write(resp.encode('latin-1'))
```

처리기에서 패킷 단위로 데이터를 다룰 것이고 송신자 쪽으로 항상 응답을 보낼 것이라면 BaseRequestHandler 대신 DatagramRequestHandler에서 상속받으면 된다. 이 클래스도 StreamRequestHandler처럼 파일 같은 인터페이스를 제공한다. 다음 예를 보자.

```

try:
    from socketserver import DatagramRequestHandler # 파일 3
except ImportError:
    from SocketServer import DatagramRequestHandler # 파일 2
import time

class TimeServer(DatagramRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        self.wfile.write(resp.encode('latin-1'))

self.wfile에 쓴 데이터는 한 패킷에 모은 다음 handler( ) 메서드가 반환될 때 보낸다.

```

Servers

처리기를 작성했으면 이것을 서버 객체에 끼워넣어야 한다. 서버 객체는 네 가지가 있다.

TCPServer(address, handler)

IPv4를 사용하는 TCP 프로토콜을 지원하는 서버. address는 (host, port) 형식의 튜플이다. handler는 앞에서 설명한 BaseRequestHandler 클래스의 하위 클래스 인스턴스이다.

UDPServer(address, handler)

IPv4를 사용하는 인터넷 UDP 프로토콜을 지원하는 서버이다. address와 handler는 TCPServer()에서와 의미가 같다.

UnixStreamServer(address, handler)

유닉스 도메인 소켓을 사용하는 스트림 지향 프로토콜을 구현하는 서버. TCPServer에서 상속한다.

UnixDatagramServer(address, handler)

유닉스 도메인 소켓을 사용하는 데이터그램 프로토콜을 구현하는 서버. UDPServer에서 상속한다.

이 네 클래스의 인스턴스는 다음 기본 메서드들을 지원한다.

s.fileno()

서버 소켓의 정수 파일 기술자를 반환한다. 이 메서드가 있기 때문에 서버 인스턴

s를 select() 함수 같은 폴링 연산에 사용해도 된다.

s.serve_forever()

요청을 끝없이 처리한다.

s.shutdown()

serve_forever() 루프를 멈춘다.

다음 속성들은 실행 중인 서버의 설정에 관한 기본 정보를 제공한다.

s.RequestHandlerClass

서버 생성자에 넘긴 사용자가 제공한 요청 처리 클래스.

s.server_address

서버가 듣고 있는 튜플 ('127.0.0.1', 80) 같은 주소.

s.socket

들어오는 요청을 받는 데 사용하고 있는 소켓 객체.

다음 예는 TimeHandler를 TCP 서버로 실행한다.

```
from SocketServer import TCPServer

serv = TCPServer(("",10000,TimeHandler)
serv.serve_forever()
```

다음 예는 UDP 서버로 실행한다.

```
from SocketServer import UDPServer

serv = UDPServer(("",10000,TimeHandler)
serv.serve_forever()
```

SocketServer 모듈의 핵심적인 특징은 처리기와 서버가 분리되어 있다는 점이다.

즉, 처리기를 한번 작성하면 구현을 바꾸지 않고서도 여러 종류의 서버에 끼워넣어서 사용할 수 있다.

커스터마이즈된 서버 정의

서버마다 네트워크 주소 체계, 타임아웃, 동시성 등의 요구 사항이 다르기 때문에 특별한 설정이 필요한 경우가 종종 있다. 이러한 커스터마이즈는 앞 절에서 설명한 네 개의 기본 서버에서 상속을 받아서 수행한다. 다음 클래스 속성들은 내부 네트워크 소켓의 기본 설정을 변경하기 위해서 정의할 수 있는 것들이다.

Server.address_family

서버 소켓이 사용하는 주소 체계. 기본은 socket.AF_INET이다. IPv6를 원하면 socket.AF_INET6를 사용한다.

Server.allow_reuse_address

소켓이 주소를 재사용할 수 있는지를 가리키는 불리언 플래그. 프로그램이 종료한 후에 즉시 같은 포트에 서버를 재구동하고 싶을 때 유용하게 쓰인다(아니면 몇 분을 기다려야 한다). 기본은 False.

Server.request_queue_size

소켓의 listen() 메서드에 전달할 요청 큐 크기. 기본은 5.

Server.socket_type

서버가 사용할 소켓 타입. socket.SOCK_STREAM이나 socket.SOCK_DGRAM 등이 가능하다.

Server.timeout

서버가 새로운 요청을 기다릴 때 타임아웃 시간을 초로. 타임아웃이 되면 서버가 handle_timeout() 메서드(뒤에서 설명한다)를 호출하고 다시 기다린다. 이 타임아웃 시간은 소켓 타임아웃을 설정하는 데 사용하지 않는다. 소켓 타임아웃이 설정된 경우에는 그 값을 이 값 대신 사용한다.

다음은 포트 번호를 재사용할 수 있게 하여 서버를 생성하는 예를 보여준다.

```
from SocketServer import TCPServer

class TimeServer(TCPServer):
    allow_reuse_address = True

serv = TimeServer(("", 10000, TimeHandler))
serv.serve_forever()
```

원할 경우 다음 메서드들을 통해서 서버 중 하나에서 상속받은 클래스를 더 확장 할 수 있다. 여러분이 만든 서버에서 이 메서드들 중 하나를 정의할 경우에는 상위 클래스에 있는 동일한 메서드를 호출해주어야 한다.

Server.activate()

서버에서 listen() 연산을 수행하는 메서드. self.socket으로 서버 소켓에 접근할 수 있다.

Server.bind()

서버에서 bind() 연산을 수행하는 메서드.

Server.handle_error(request, client_address)

잡히지 않은 예외를 처리하는 메서드. 마지막으로 발생한 예외에 관한 정보를 얻으려면 sys.exc_info()나 traceback 모듈에 있는 함수를 사용하면 된다.

Server.handle_timeout()

서버 타임아웃이 일어났을 때 호출되는 메서드. 이 메서드를 재정의하여 타임아웃 설정을 조정함으로써 서버 이벤트 루프에 다른 처리 과정을 통합할 수 있다.

Server.verify_request(request, client_address)

추가 처리를 하기 전에 연결을 검증하고자 할 때 이 메서드를 재정의한다. 방화벽을 구현하거나 기타 검증 작업을 수행하고자 할 때 정의하는 메서드이다.

마지막으로 혼합 클래스를 통해 서버에 기능을 더 추가할 수 있다. 스레드나 포크 등을 통해 동시성을 추가하고자 할 때 혼합 클래스를 사용한다. 이 목적으로 다음 두 클래스가 정의된다.

ForkingMixIn

여러 클라이언트에 서비스를 제공하기 위해서 서버에 유닉스 프로세스 포크를 추가하는 혼합 클래스. 클래스 변수인 max_children은 최대 자식 프로세스 개수를 제어하고 timeout 변수는 좀비 프로세스를 수집하려는 시도 사이에 얼마나 기다릴지를 결정한다. 인스턴스 변수인 active_children은 현재 활발히 실행 중인 프로세스가 몇 개인지를 추적한다.

ThreadingMixIn

스레드로 여러 클라이언트에 서비스를 제공할 수 있게 서버를 수정하는 혼합 클래스. 생성할 수 있는 스레드 수에는 제한이 없다. 기본으로 daemon_threads를 True로 설정하지 않는 한 스레드는 데몬 모드로 실행되지 않는다.

서버에 추가 기능을 추가하려면 혼합 클래스를 가장 먼저 지정하면서 다중 상속을 사용하면 된다. 예를 들어, 다음은 포크 시간 서버를 보여준다.

```
from SocketServer import TCPServer, ForkingMixIn

class TimeServer(ForkingMixIn, TCPServer):
    allow_reuse_address = True
    max_children = 10
```

```
serv = TimeServer("", 10000, TimeHandler)
serv.serve_forever()
```

동시성을 지원하는 서버가 상대적으로 흔히 사용되기 때문에 SocketServer에는 다음과 같이 미리 정의된 서버 클래스들이 있다.

- ForkingUDPServer(address, handler)
- ForkingTCPServer(address, handler)
- ThreadingUDPServer(address, handler)
- ThreadingTCPServer(address, handler)

실제로 이 클래스들은 혼합 클래스와 서버 클래스를 사용해서 정의한 것들이다. 예를 들어, ForkingTCPServer의 정의는 다음과 같다.

```
class ForkingTCPServer(ForkingMixIn, TCPServer): pass
```

응용 프로그램 서버의 커스터마이즈

다른 라이브러리 모듈에서 HTTP나 XML-RPC 같은 응용 프로그램 수준 프로토콜을 지원하는 서버를 구현하기 위해 SocketServer 모듈을 종종 사용한다. 상속을 사용하거나 기본 서버 연산을 수행하는 메서드를 확장함으로써 이러한 서버를 커스터마이즈할 수 있다. 예를 들어, 다음은 루프백 인터페이스에 대한 연결만 받아들이는 포크 XML-RPC 서버를 보여준다.

```
try:
    from xmlrpclib import SimpleXMLRPCServer # 파이썬 3
    from socketserver import ForkingMixIn
except ImportError: # 파이썬 2
    from SimpleXMLRPCServer import SimpleXMLRPCServer
    from SocketServer import ForkingMixIn

class MyXMLRPCServer(ForkingMixIn, SimpleXMLRPCServer):
    def verify_request(self, request, client_address):
        host, port = client_address
        if host != '127.0.0.1':
            return False
        return SimpleXMLRPCServer.verify_request(self, request,
                                                client_address)

# 사용 예
def add(x,y):
    return x+y
server = MyXMLRPCServer(("","45000"))
server.register_function(add)
server.serve_forever()
```

이 코드를 테스트하려면 xmlrpclib 모듈을 사용할 필요가 있다. 앞에 나온 서버를 실행하고 별개의 파이썬 프로세스를 시작시킨다.

```
>>> import xmlrpclib  
>>> s = xmlrpclib.ServerProxy("http://localhost:45000")  
>>> s.add(3,4)  
7  
>>>
```

연결 거부를 테스트하기 위해서 같은 코드를 실행하되 네트워크에 있는 다른 머신에서 실행해본다. 이때 “localhost”를 서버를 실행하고 있는 머신의 호스트 이름으로 바꾸어주어야 한다.

22장

P y t h o n E s s e n t i a l R e f e r e n c e

인터넷 응용 프로그래밍

이 장에서는 HTTP, XML-RPC, FTP, SMTP 등 인터넷 응용 프로토콜과 관련 있는 모듈을 다룬다. CGI 스크립팅 같은 웹 프로그래밍과 관련된 주제는 23장에서 논의한다. 주로 사용되는 인터넷 데이터 형식에 관련된 모듈은 24장에서 다룬다.

네트워크 관련 모듈이 구성되어 있는 방식은 파이썬 2와 3에서 크게 다르다. 이 장에서는 나중을 위해 더 논리적으로 구성된 파이썬 3 라이브러리 구조를 다룬다. 그렇지만 이 책을 쓰고 있는 지금 파이썬 버전에 상관없이 제공하는 기능은 사실상 동일하다. 각 절에서는 가능할 경우 옆에 대응하는 파이썬 2 모듈의 이름을 표시할 것이다.

ftplib

ftplib 모듈은 FTP 프로토콜에서 클라이언트 쪽을 구현한다. urllib 패키지에서 관련 기능의 고수준 인터페이스를 제공하기 때문에 이 모듈을 직접 사용하는 일은 드물다. 그렇지만 저수준에서 FTP 연결을 제어하려는 경우에는 여전히 유용하게 사용할 수 있다. 인터넷 RFC 959에 설명된 FTP 프로토콜을 잘 이해하면 이 모듈을 사용하는 데 도움이 된다.

이 모듈에는 FTP 연결을 수립하는 데 사용할 수 있는 다음 클래스가 하나 정의되어 있다.

FTP([host [, user [, passwd [, acct [, timeout]]]]])

FTP 연결을 나타내는 객체를 생성한다. host는 호스트 이름을 지정하는 문자열

이다. user, passwd, acct는 추가로 사용자 이름, 암호, 계정을 지정한다. 아무것도 지정하지 않은 경우에는 연결을 직접 초기화하기 위해 connect()와 login() 메서드를 호출해주어야 한다. host가 주어지면 connect()가 자동으로 호출된다. user, passwd, acct가 주어지면 login()이 호출된다. timeout은 초 단위인 타임아웃 시간이다.

FTP 인스턴스 f는 다음 메서드들을 갖는다.

f.abort()

진행 중인 파일 전송을 취소하려고 시도한다. 원격 서버에 따라 성공 여부가 달라질 수 있다.

f.close()

FTP 연결을 닫는다. 이 메서드를 호출하고 나면 FTP 객체 f에 대해서 추가 연산을 수행할 수 없다.

f.connect(host [, port [, timeout]])

주어진 호스트와 포트로 FTP 연결을 한다. host는 호스트 이름을 지정하는 문자열이다. port는 FTP 서버의 정수 포트 번호이고 기본 포트는 21이다. timeout은 초 단위로 타임아웃 시간을 나타낸다. FTP()에 호스트 이름을 이미 준 경우라면 이 메서드를 호출할 필요가 없다.

f.getcwd(pathname)

서버에서 현재 작업 디렉터리를 pathname으로 변경한다.

f.delete(filename)

서버에서 filename 파일을 삭제한다.

f.dir([dirname [, ... [, callback]]])

‘LIST’ 명령이 출력하는 디렉터리 목록을 생성한다. dirname은 나열할 디렉터리 이름을 지정한다. 추가로 인수가 주어지면 간단히 ‘LIST’의 인수로 주어진다. 마지막 인수인 callback이 함수이면 이 함수는 반환되는 디렉터리 목록 데이터를 처리하는 역호출 함수로 사용된다. 이 역호출 함수는 retrlines() 메서드에서 사용하는 역호출 함수와 같은 방식으로 작동한다. 기본으로 이 메서드는 디렉터리 목록을 sys.stdout에 출력한다.

f.login([user, [passwd [, acct]]])

주어진 사용자 이름, 암호, 계정으로 서버에 로그인한다. user는 사용자 이름을 담은 문자열이고 기본으로 ‘anonymous’이다. passwd는 암호를 담은 문자열이고 기본은 빈 문자열 ‘ ’이다. acct는 문자열로서 기본은 빈 문자열이다. FTP()에서 관련 정보를 이미 지정했다면 이 메서드를 호출할 필요가 없다.

f.mkd(pathname)

서버에 새로운 디렉터리를 생성한다.

f.ntransfercmd(command [, rest])

튜플 (sock, size)를 반환하는 것 말고는 transfercmd()와 동일. sock은 데이터 연결을 나타내는 소켓 객체이고 size는 바이트로 예상 데이터 크기를 지정한다. 크기를 모를 경우에는 None이다.

f.getcwd()

서버의 현재 작업 디렉터리를 담은 문자열을 반환한다.

f.quit()

서버에 ‘QUIT’ 명령을 보냄으로써 FTP 연결을 닫는다.

f.rename(oldname, newname)

서버에서 파일 이름을 변경한다.

f.retrbinary(command, callback [, blocksizes [, rest]])

서버에서 명령을 실행한 결과를 이진 전송 모드로 반환한다. command는 적절한 파일 추출 명령을 지정하는 문자열이고 거의 대부분 ‘RETR filename’이다. callback은 데이터 블록을 받을 때마다 호출되는 역호출 함수이다. 이 역호출 함수는 문자열로 받은 데이터를 담은 인수 하나를 받는다. blocksizes는 사용할 최대 블록 크기이고 기본은 8192 바이트이다. rest는 옵션인 파일 오프셋이다. 제공될 경우 전송을 시작할 파일 위치를 지정한다. rest 옵션은 모든 FTP 서버가 지원하는 것이 아니며 그런 경우 error_reply 예외가 발생할 수 있다.

f.retrlines(command [, callback])

서버에서 명령을 실행한 결과를 텍스트 전송 모드로 반환한다. command는 명령을 지정하는 문자열이고 보통 ‘RETR filename’이다. callback은 데이터 한 줄을 받을

때마다 호출되는 역호출 함수이다. 이 역호출 함수는 받은 데이터를 담은 문자열 인수 하나를 받는다. callback을 생략하면 반환되는 데이터가 sys.stdout에 써진다.

f.rmd(pathname)

서버에서 디렉터리를 삭제한다.

f.sendcmd(command)

간단한 명령을 서버로 보내고 서버의 응답을 반환한다. command는 명령을 담은 문자열이다. 이 메서드는 데이터 전송을 일으키지 않는 명령에만 사용할 수 있다.

f.set_pasv(pasv)

수동 모드(passive mode)를 설정한다. pasv는 수동 모드를 나타내는 불리언 플래그이다. True이면 수동 모드를 켜고 False이면 끈다. 기본으로 수동 모드는 켜진다.

s.size(filename)

filename의 크기를 바이트로 반환한다. 어떤 이유로 크기를 알 수 없으면 None을 반환한다.

f.storbinary(command, file [, blocksize])

서버에서 명령을 실행하고 이진 전송 모드로 데이터를 전송한다. command는 저수준 명령을 담은 문자열이다. 대부분 ‘STOR filename’이 된다. filename은 서버에 올리고자 하는 파일 이름이다. file은 file.read(blocksize)로 데이터를 읽어서 서버로 전송하는 데 사용할 열린 파일 객체이다. blocksize는 전송에 사용할 블록 크기를 나타낸다. 기본으로 8192 바이트이다.

f.storlines(command, file)

서버에서 명령을 실행하고 텍스트 전송 모드로 데이터를 전송한다. command는 저수준 명령을 담은 문자열이다. 보통 ‘STOR filename’이 된다. file은 file.readline()로 데이터를 읽어서 서버로 보내는 데 사용할 열린 파일 객체이다.

f.transfercmd(command [, rest])

FTP 데이터 연결을 통해 전송을 시작한다. 능동 모드(active mode)가 사용 중이면 ‘PORT’나 ‘EPRT’ 명령을 보내고 서버의 연결을 받아들인다. 수동 모드가 사용 중이면 ‘EPSV’나 ‘PASV’ 명령을 보내고 서버에 연결한다. 어느 경우이든 데이터 연결이 일단 성립되면 FTP 명령 command를 보낸다. 이 함수는 열린 데이터 연결을

나타내는 소켓 객체를 반환한다. 옵션인 rest 매개변수는 서버에 요청한 파일의 시작 바이트 오프셋을 지정한다. 이 옵션은 모든 서버에서 지원하지 않을 수 있고 그런 경우 error_reply 예외가 발생한다.

예

다음 예는 이 모듈을 사용해 파일을 FTP 서버에 업로드하는 방법을 보여준다.

```
host = "ftp.foo.com"
username = "dave"
password = "1235"
filename = "somefile.dat"

import ftplib
ftp_serv = ftplib.FTP(host,username,password)
# 보낼 파일을 연다.
f = open(filename,"rb")
# FTP 서버에 보낸다.
resp = ftp_serv.storbinary("STOR "+filename, f)
# 연결을 닫는다.
ftp_serv.close()
```

FTP 서버에서 문서를 받을 때는 urllib 패키지를 사용한다. 다음 예를 보자.

```
try:
    from urllib.request import urlopen    # 파이썬 3
except ImportError:
    from urllib2 import urlopen        # 파이썬 2

u = urlopen("ftp://username:password@somehostname/somefile")
contents = u.read()
```

http 패키지

http 패키지는 HTTP 클라이언트와 서버를 작성하기 위한 모듈과 상태 관리(쿠키)를 지원하는 모듈로 구성된다. HTTP(Hypertext Transfer Protocol)은 다음과 같이 작동하는 간단한 텍스트 기반 프로토콜이다.

1. 클라이언트는 HTTP 서버에 연결을 맺고 다음 형태의 요청 헤더를 보낸다.

```
GET /document.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.61 [en] (X11; U; SunOS 5.6 sun4u)
Host: rustler.cs.uchicago.edu:8000
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, /*
Accept-Encoding: gzip
Accept-Language: en
```

```
Accept-Charset: iso-8859-1,* ,utf-8
```

옵션인 데이터

...

첫 번째 줄은 요청 종류, 문서(selector), 프로토콜 버전을 정의한다. 이어서 암호, 쿠키, 캐시 설정, 클라이언트 소프트웨어 등 클라이언트와 관련된 다양한 정보를 담은 헤더 줄들이 나온다. 그다음으로 빈 줄 하나가 나와서 헤더 줄의 끝을 알린다. 폼(form) 정보를 보낸다든지 파일을 업로드하는 경우처럼 요청을 통해 정보를 보내는 경우에는 헤더 다음에 추가로 데이터가 따라 나올 수 있다. 헤더에서 각 줄은 반드시 개리지 리턴과 줄바꿈 문자('r\n')로 끝나야 한다.

2. 서버가 다음 형태의 응답을 보낸다.

```
HTTP/1.0 200 OK
```

```
Content-type: text/html
```

```
Content-length: 72883 bytes
```

...

헤더: 데이터

데이터

...

서버 응답에서 첫 번째 줄은 HTTP 프로토콜 버전, 성공 코드, 반환 메시지를 담는다. 응답 줄 다음에는 반환되는 문서의 종류, 문서의 크기, 웹 서버 소프트웨어, 쿠키 등에 관한 정보를 담은 일련의 헤더 필드들이 나온다. 헤더는 빈 줄 하나로 끝나고 그다음에 요청된 문서인 무가공 데이터가 나온다.

다음은 가장 많이 사용되는 요청 방식이다.

방식	설명
GET	문서를 얻는다.
POST	폼 데이터를 올린다.
HEAD	헤더 정보만 반환한다.
PUT	데이터를 서버에 업로드한다.

표 22.1은 서버에서 가장 자주 사용되는 응답 코드를 정리한 것이다. 기호 상수 열은 가독성을 높이기 위해 사용할 수 있는 정수 응답 코드 값을 담은 http.client에 미리 정의된 변수의 이름을 나타낸다.

표 22.1. 서버에서 가장 자주 사용되는 응답 코드

코드	설명	기호 상수
성공 코드(2xx)		
200	OK	OK
201	생성함	CREATED
202	수락함	ACCEPTED
204	내용 없음	NO_CONTENT
전환(3xx)		
300	다중 선택	MULTIPLE_CHOICES
301	영원히 이동함	MOVED_PERMANENTLY
302	임시로 이동함	MOVED_TEMPORARILY
303	수정 안 됨	NOT_MODIFIED
클라이언트 에러(4xx)		
400	잘못된 요청	BAD_REQUEST
401	권한 없음	UNAUTHORIZED
403	금지됨	FORBIDDEN
404	찾을 수 없음	NOT_FOUND
서버 에러(5xx)		
500	내부 서버 에러	INTERNAL_SERVER_ERROR
501	구현 안 됨	NOT_IMPLEMENTED
502	게이트웨이 불량	BAD_GATEWAY
503	서비스를 사용할 수 없음	SERVICE_UNAVAILABLE

요청과 응답에 모두 나타날 수 있는 헤더들은 RFC-822라고 부르는 널리 알려진 형식으로 인코딩된다. 각 헤더의 일반적인 형식은 ‘헤더 이름: 데이터’이며 더 자세한 정보는 RFC를 참고하기 바란다. 보통은 파이썬이 알아서 이 헤더들을 파싱해주기 때문에 직접 파싱하는 일은 거의 없다.

http.client(httplib)

http.client 모듈은 HTTP의 클라이언트 쪽을 위한 저수준 기능을 제공한다. 파이썬 2에서 이 모듈의 이름은 `httplib`이었다. 보통은 이 모듈보다는 `urllib` 패키지에 있는 함수를 사용하는 것이 좋을 것이다. 이 모듈은 HTTP/1.0과 HTTP/1.1을 모두 지원

하고 파이썬을 OpenSSL을 지원하도록 컴파일했다면 SSL을 통한 연결 맺기도 지원 한다. 보통 이 패키지를 바로 사용하지는 않는다. 대신 urllib 패키지를 사용하는 것을 고려해보라. 그래도 HTTP가 매우 중요한 프로토콜이기 때문에 urllib로는 쉽게 처리할 수 없는 저수준 작업이 필요한 경우가 있다. 예를 들어, GET이나 POST 이외의 요청을 보내고자 할 때가 그런 경우다. HTTP 프로토콜에 대한 더 자세한 정보는 RFC 2616(HTTP/1.1)과 RFC 1945(HTTP/1.0)을 참고하도록 한다.

다음 클래스들은 서버로 HTTP 연결을 맺는 데 사용한다.

HTTPConnection(host [,port])

HTTP 연결을 맺는다. host는 호스트 이름이고 port는 원격 포트 번호이다. 기본 포트는 80이다. HTTPConnection 인스턴스를 반환한다.

HTTPSConnection(host [, port [, key_file=kfile [, cert_file=cfile]]])

보안 소켓을 사용해 HTTP 연결을 맺는다. 기본 포트는 443이다. key_file과 cert_file은 옵션인 키워드 인수로서 PEM 형식의 개인 키와 인증서 체인 파일을 나타내며 이들은 클라이언트 인증에 필요하다. 서버 인증서를 검증하지는 않는다. HTTPSConnection 인스턴스를 반환한다.

HTTPConnection이나 HTTPSConnection 인스턴스 h는 다음 메서드들을 지원한다.

h.connect()

HTTPConnection()나 HTTPSConnection()에 주어진 호스트와 포트로 연결을 초기화한다. 다른 메서드에서는 연결이 아직 성립되지 않았으면 이 메서드를 자동으로 호출한다.

h.close()

연결을 닫는다.

h.send(bytes)

바이트 문자열 bytes를 서버로 보낸다. 내부 응답 또는 요청 프로토콜이 진행되는 것을 방해할 수 있기 때문에 이 메서드를 바로 사용하는 것을 권장하지 않는다. h.endheaders()를 호출하고 난 뒤에 서버에 데이터를 보내는 데 주로 사용한다.

h.putrequest(method, selector [, skip_host [, skip_accept_encoding]])

서버에 요청을 보낸다. method는 ‘GET’이나 ‘POST’ 같은 HTTP 요청 방식이다.

selector는 '/index.html'처럼 반환할 객체를 지정한다. skip_host와 skip_accept_encoding 매개변수는 HTTP 요청에서 Host:와 Accept-Encoding: 헤더를 보내지 않게 하는 플래그이다. 기본으로 두 인수 모두 False이다. HTTP/1.1 프로토콜에서는 단일 연결로 여러 요청을 보내는 것을 허락하기 때문에 어떤 연결에서 새 요청을 보내는 것을 금지하는 상황일 경우 CannotSendRequest 예외가 발생할 수 있다.

h.putheader(header, value, ...)

RFC-822 형식의 헤더를 서버로 보낸다. 헤더, 콜론, 스페이스, 값으로 구성되는 한 줄을 서버로 보낸다. 추가 인수들은 헤더의 이어지는 줄들로서 인코딩한다. h가 헤더를 보낼 수 있는 상태가 아닌 경우에는 CannotSendHeader 예외가 발생한다.

h.endheaders()

헤더 줄의 끝을 알리는 빈 줄을 서버로 보낸다.

h.request(method, url [, body [, headers]])

서버에 완전한 HTTP 요청을 보낸다. method와 url은 h.putrequest()에서와 같은 의미를 갖는다. body는 옵션인 문자열로서 요청을 보낸 후에 서버에 업로드할 데이터를 담은 문자열이다. body를 지정하면 Content-length: 헤더가 자동으로 적절한 값으로 설정된다. headers는 h.putheader() 메서드에 주어질 헤더: 값 쌍들을 담은 사전이다.

h.getresponse()

서버에서 응답을 받은 후 데이터를 읽을 때 사용할 수 있는 HTTPResponse 인스턴스를 반환한다. h가 응답을 받을 수 있는 상태가 아니면 ResponseNotReady 예외를 발생시킨다.

getresponse() 메서드가 반환하는 HTTPResponse 인스턴스 r은 다음 메서드들을 지원한다.

r.read([size])

서버에서 size만큼 바이트를 읽는다. size를 생략하면 이 요청에 대한 전체 데이터를 반환한다.

r.getheader(name [,default])

응답 헤더를 얻는다. name은 헤더의 이름이다. default는 헤더를 찾을 수 없을 때 반환할 기본 값이다.

r.getheaders()

(헤더, 값) 튜플들의 리스트를 반환한다.

HTTPResponse 인스턴스 r에는 다음 속성들도 있다.

r.version

서버가 지원하는 HTTP 버전.

r.status

서버가 반환한 HTTP 상태 코드.

r.reason

서버가 반환한 HTTP 에러 메시지.

r.length

응답에 남아 있는 바이트 수.

예외

HTTP 연결을 다루는 동안 다음 예외들이 발생할 수 있다.

예외	설명
HTTPException	HTTP와 관련된 에러들의 기본 클래스
NotConnected	요청을 했지만 연결이 되지 않은 상태임
InvalidURL	불량 URL 또는 포트 번호
UnknownProtocol	알 수 없는 HTTP 프로토콜 번호
UnknownTransferEncoding	알 수 없는 전송 인코딩
Unimplemented FileMode	구현 안 된 파일 모드
IncompleteRead	불완전한 데이터를 받음
BadStatusLine	알 수 없는 상태 코드를 받음

다음 예외들은 HTTP/1.1 연결 상태와 관련된 것들이다. HTTP/1.1에서는 한 연결에 대해 여러 요청, 응답을 보낼 수 있기 때문에 요청과 응답을 언제 보내고 받을 수 있는지에 대한 추가 규칙이 존재한다. 엉뚱한 순서로 연산을 수행할 경우 예외가 발생할 수 있다.

예외	설명
ImproperConnectionState	모든 HTTP 연결 상태 에러의 기본 클래스
CannotSendRequest	요청을 보낼 수 없음
CannotSendHeader	헤더를 보낼 수 없음
ResponseNotReady	응답을 보낼 수 없음

예

다음 예는 POST 요청을 통해 메모리 효율적으로 서버에 파일을 보내기 위해 HTTPConnection 클래스를 사용하는 방법을 보여준다. 같은 일을 urllib 프레임워크로는 쉽게 할 수 없다.

```
import os
try:
    from httplib import HTTPConnection      # 파이썬 2
except ImportError:
    from http.client import HTTPConnection  # 파이썬 3

BOUNDARY = "$Python-Essential-Reference$"
CRLF = '\r\n'

def upload(addr, url, formfields, filefields):
    # 폼 필드를 위한 구역을 생성한다.
    formsections = []
    for name in formfields:
        section = [
            '--'+BOUNDARY,
            'Content-disposition: form-data; name="%s"' % name,
            '',
            formfields[name]
        ]
        formsections.append(CRLF.join(section)+CRLF)

    # 업로드할 모든 파일에 대한 정보를 모은다.
    fileinfo = [(os.path.getsize(filename), formname, filename)
                for formname, filename in filefields.items()]

    # 각 파일에 대해 HTTP 헤더를 생성한다.
    filebytes = 0
    fileheaders = []
    for filesize, formname,filename in fileinfo:
        headers = [
            '--'+BOUNDARY,
            'Content-Disposition: form-data; name="%s"; filename="%s"' % \
                (formname, filename),
            'Content-length: %d' % filesize,
            ''
        ]
        fileheaders.append(CRLF.join(headers)+CRLF)
        filebytes += filesize

    # 닫음 표시
    closing = "--"+BOUNDARY+"--\r\n"

    # 전체 요청 길이를 알아낸다.
    content_size = (sum(len(f) for f in formsections) +
                    sum(len(f) for f in fileheaders) +
                    filebytes+len(closing))
```

```

# 업로드한다.
conn = HTTPConnection(*addr)
conn.putrequest("POST",url)
conn.putheader("Content-type",
    'multipart/form-data; boundary=%s' % BOUNDARY)
conn.putheader("Content-length", str(content_size))
conn.endheaders()

# 모든 폼 구역을 보낸다.
for s in formsections:
    conn.send(s.encode('latin-1'))

# 모든 파일을 보낸다.
for head,filename in zip(fileheaders,filefields.values()):
    conn.send(head.encode('latin-1'))
    f = open(filename,"rb")
    while True:
        chunk = f.read(16384)
        if not chunk: break
        conn.send(chunk)
    f.close()
conn.send(closing.encode('latin-1'))
r = conn.getresponse()
responsedata = r.read()
conn.close()
return responsedata

# 실례: 파일을 몇 개 올린다. 원격 서버에서는 폼 필드 'name', 'email'
# 'file_1','file_2' 등을 기대한다(물론 다를 수 있다).
server      = ('localhost', 8080)
url         = '/cgi-bin/upload.py'
formfields = {
    'name' : 'Dave',
    'email' : 'dave@dabeaz.com'
}
filefields = {
    'file_1' : 'IMG_1008.JPG',
    'file_2' : 'IMG_1757.JPG'
}
resp = upload(server, url,formfields,filefields)
print(resp)

```

http.server(BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer)

http.server 모듈은 HTTP 서버를 구현하는 데 필요한 다양한 클래스를 제공한다. 파이썬 2에서 이 모듈의 내용은 BaseHTTPServer, CGIHTTPServer와 SimpleHTTPServer 세 개의 라이브러리 모듈에 나누어져 있다.

HTTPServer

다음 클래스는 기본 HTTP 서버를 구현한다. 파이썬 2에서는 BaseHTTPServer 모듈이 그 역할을 한다.

HTTPServer(server_address, request_handler)

새로운 HTTPServer 객체를 생성한다. server_address는 서버가 응답을 기다릴 (host, port) 형식의 튜플이다. request_handler는 나중에 설명할 BaseHTTPRequestHandler에서 상속받은 처리기 클래스이다.

HTTPServer는 socketserver 모듈에 정의된 TCPServer를 바로 상속한다. 따라서 여러분이 HTTP 서버의 작동 방식을 변경하고자 한다면 HTTPServer에서 상속해서 기능을 확장하면 된다. 다음은 특정 서브넷에서 오는 연결만 받아들이는 멀티스레드 HTTP 서버를 정의하는 방법을 보여준다.

```
try:
    from http.server import HTTPServer      # 파이썬 3
    from socketserver import ThreadingMixIn
except ImportError:
    from BaseHTTPServer import HTTPServer    # 파이썬 2
    from SocketServer import ThreadingMixIn

class MyHTTPServer(ThreadingMixIn, HTTPServer):
    def __init__(self, addr, handler, subnet):
        HTTPServer.__init__(self, addr, handler)
        self.subnet = subnet
    def verify_request(self, request, client_address):
        host, port = client_address
        if not host.startswith(subnet):
            return False
        return HTTPServer.verify_request(self, request, client_address)

# 서버를 실행하는 방법
serv = MyHTTPServer(("", 8080), SomeHandler, '192.168.69.')
serv.serve_forever()
```

HTTPServer 클래스는 저수준 HTTP 프로토콜만을 다룬다. 실제로 무언가를 수행하려면 처리기 클래스를 제공해야 한다. 두 개의 내장 처리기와 여러분만의 처리기를 정의하는 데 사용할 수 있는 하나의 기반 클래스가 있다. 이어서 이들을 설명한다.

SimpleHTTPRequestHandler와 CGIHTTPRequestHandler

간단한 독립 웹 서버를 빠르게 구축하고 싶다면 두 개의 미리 만들어진 웹 서버 처

리기 클래스를 사용하면 된다. 이 클래스들은 아파치 같은 써드 파티 웹 서버와는 별개로 작동한다.

CGIHTTPRequestHandler(request, client_address, server)

현재 디렉터리와 그 아래에 있는 모든 하위 디렉터리에 있는 파일을 제공한다. 그리고 이 처리기는 특수한 CGI 디렉터리(cgi_directories 클래스 변수로 정의되고 기본으로 ['/cgi-bin', '/htbin']로 설정됨)에 있는 파일을 CGI 스크립트로서 실행한다. 이 처리기는 GET, HEAD, POST 메서드를 지원한다. HTTP 전환(HTTP 코드 302)은 지원하지 않기 때문에 간단한 CGI 응용 프로그램을 위해서만 사용할 수 있다. 보안 문제로 CGI 스크립트는 UID가 nobody인 상태로 실행된다. 파이썬 2에서 이 클래스는 CGIHTTPServer 모듈에 정의되어 있다.

SimpleHTTPRequestHandler(request, client_address, server)

현재 디렉터리와 그 아래에 있는 모든 하위 디렉터리에 있는 파일을 제공한다. 이 클래스는 HEAD와 GET 메서드를 지원한다. 모든 IOError 예외는 “404 File not found” 에러를 발생시킨다. 디렉터리에 접근하려는 시도는 “403 Directory listing not supported” 에러를 발생시킨다. 파이썬 2에서 이 클래스는 SimpleHTTPServer 모듈에 정의되어 있다.

원활 경우 상속을 통해서 변경이 가능하도록 두 처리기 모두에서 다음 클래스 변수들을 정의한다.

handler.server_version

클라이언트에 반환할 서버 버전 문자열. 기본으로 ‘SimpleHTTP/0.6’ 같은 문자열로 설정된다.

handler.extensions_map

파일 확장자를 MIME 타입에 매핑하는 사전. 인식할 수 없는 파일 타입은 ‘application/octet-stream’ 타입인 것으로 간주된다.

다음은 이 처리기 클래스들을 사용하여 CGI 스크립트를 실행할 수 있는 독립 웹 서버를 생성하고 실행하는 예를 보여준다.

```
try:
    from http.server import HTTPServer, CGIHTTPRequestHandler # 파이썬 3
except ImportError:
    from BaseHTTPServer import HTTPServer # 파이썬 2
    from CGIHTTPServer import CGIHTTPRequestHandler
```

```

import os

# 문서 루트 디렉터리로 이동한다.
os.chdir("/home/httpd/html")
# CGIHTTP 서버를 포트 8080에서 시작한다.
serv = HTTPServer(("",8080),CGIHTTPRequestHandler)
serv.serve_forever()

```

BaseHTTPRequestHandler

BaseHTTPRequestHandler 클래스는 자신만의 HTTP 서버 처리 기능을 구현하고 싶을 때 사용할 수 있는 기반 클래스이다. 미리 만들어진 SimpleHTTPRequestHandler와 CGIHTTPRequestHandler 클래스가 이 클래스에서 상속받는다. 파일 2에서 이 클래스는 BaseHTTPServer 모듈에 정의되어 있다.

BaseHTTPRequestHandler(request, client_address, server)

HTTP 요청을 처리하는 데 사용할 수 있는 기반 처리기 클래스. 연결을 받으면 요청과 HTTP 헤더를 파싱한다. 그리고 나서 요청 방식(REQUEST)에 따라서 do_REQUEST() 형태의 메서드가 실행된다. 예를 들어, ‘GET’ 방식은 do_GET() 메서드를 호출하고 ‘POST’ 메서드는 do_POST()를 호출한다. 기본으로 이 클래스는 아무 일도 하지 않는다. 하위 클래스에서 이 메서드들을 정의해야 한다.

BaseHTTPRequestHandler에는 다음 클래스 변수들이 정의되어 있고 하위 클래스에서 재정의할 수 있다.

BaseHTTPRequestHandler.server_version

서버가 클라이언트에 보고할 서버의 소프트웨어 버전 문자열을 담는다. 예를 들어, ‘ServerName/1.2’.

BaseHTTPRequestHandler.sys_version

‘Python/2.6’ 같은 파일 시스템 버전.

BaseHTTPRequestHandler.error_message_format

클라이언트에 보낼 여러 메시지를 만드는 데 사용할 수 있는 포맷 문자열. 포맷 문자열은 code, message, explain 속성을 담은 사전에 적용된다. 다음 예를 보자.

```

'''<head>
<title>Error response</title>
</head>
<body>
<h1>Error response</h1>

```

```
<p>Error code %(code)d.
<p>Message: %(message)s.
<p>Error code explanation: %(code)s = %(explain)s.
</body>'''
```

BaseHTTPRequestHandler.protocol_version

응답에 사용할 HTTP 프로토콜 버전. 기본은 ‘HTTP/1.0’.

BaseHTTPRequestHandler.responses

정수 HTTP 에러 코드를 이 문제를 설명하는 두 개 항목을 가진 튜플 (message, explain)으로 매핑. 예를 들어, 정수 코드 404는 (“Not Found”, “Nothing matches the given URI”)로 매핑된다. 이 매핑에서 정수 코드와 문자열은 앞에서 설명한 error_message_format 속성의 에러 메시지를 만들 때 사용된다.

연결을 처리하기 위해 만들어진 BaseHTTPRequestHandler 인스턴스 b에는 다음 속성들이 정의되어 있다.

속성	설명
b.client_address	클라이언트 주소를 튜플 (host, port)로
b.command	‘GET’, ‘POST’, ‘HEAD’ 같은 요청 방식
b.path	‘/index.html’ 같은 요청 경로
b.request_version	‘HTTP/1.0’ 같은 요청에서 HTTP 버전
b.headers	매핑 객체에 저장된 HTTP 헤더들. 헤더의 내용을 검사하거나 추출하고 싶으면 headername in b.headers나 headerval = b.headers[headername]처럼 사전 연산을 사용하면 된다.
b.rfile	추가 입력 데이터를 읽기 위한 입력 스트림. 클라이언트가 데이터를 업로드할 때 사용한다(예를 들어, POST 요청 동안).
b.wfile	클라이언트로 응답을 쓰기 위한 출력 스트림

다음 메서드들이 하위 클래스에서 주로 사용되거나 재정의된다.

b.send_error(code [, message])

실패한 요청에 대한 응답을 보낸다. code는 숫자 HTTP 응답 코드이다. message는 옵션인 에러 메시지이다. 에러를 기록하기 위해 log_error()가 호출된다. 이 메서드는 error_message_format 클래스 변수를 사용해서 완전한 에러 응답을 생성하고 이것을 클라이언트로 전송한 다음 연결을 닫는다. 그리고 나서는 어떠한 연산도 수행할 수 없다.

b.send_response(code [, message])

성공한 요청에 대해 응답을 보낸다. HTTP 응답 줄을 보내고 그다음으로 Server 와 Date 헤더를 전송한다. code는 HTTP 응답 코드이고 message는 옵션인 메시지이다. 응답을 기록하기 위해 log_request()를 호출한다.

b.send_header(keyword, value)

출력 스트림으로 MIME 헤더를 쓴다. keyword는 헤더 키워드이고 value는 값이다. send_response()를 호출한 다음에만 호출해야 한다.

b.end_headers()

MIME 헤더의 끝을 알리는 빈 줄을 보낸다.

b.log_request([code [, size]])

성공한 요청을 기록한다. code는 HTTP 코드이고 size는 응답의 크기를 바이트 단위로 나타낸 것이다(응답이 있을 경우). 기본으로 로깅을 위해서 log_message()를 호출한다.

b.log_error(format, ...)

에러 메시지를 기록한다. 기본으로 로깅을 위해서 log_message()를 호출한다.

b.log_message(format, ...)

임의의 메시지를 sys.stderr에 기록한다. format은 추가로 넘어온 인수들에 적용할 포맷 문자열이다. 각 메시지에는 클라이언트 주소와 현재 시간이 앞에 붙는다.

다음 예는 별개의 스레드로 실행되어 요청 경로를 키로 해석해서 사전의 내용을 모니터링할 수 있게 하는 커스텀 HTTP 서버를 생성하는 예이다.

```
try:
    from http.server import BaseHTTPRequestHandler, HTTPServer # 파이썬 3
except ImportError:
    from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # 파이썬 2

class DictRequestHandler(BaseHTTPRequestHandler):
    def __init__(self, thedict, *args, **kwargs):
        self.thedict = thedict
        BaseHTTPRequestHandler.__init__(self, *args, **kwargs)

    def do_GET(self):
        key = self.path[1:] # Strip the leading '/'
        if not key in self.thedict:
            self.send_error(404, "No such key")
        else:
            self.send_response(200)
```

```

        self.send_header('content-type','text/plain')
        self.end_headers()
        resp = "Key : %s\n" % key
        resp += "Value: %s\n" % self.thedict[key]
        self.wfile.write(resp.encode('latin-1'))

# 사용 예
d = {
    'name' : 'Dave',
    'values' : [1,2,3,4,5],
    'email' : 'dave@dbeaz.com'
}
from functools import partial
serv = HTTPServer(("",9000), partial(DictRequestHandler,d))

import threading
d_mon = threading.Thread(target=serv.serve_forever)
d_mon.start()

```

이 예를 테스트하려면 서버를 실행하고 브라우저 주소창에 `http://localhost:9000/name` 또는 `http://localhost:9000/values`를 입력한다. 제대로 작동하면 사전의 내용이 출력될 것이다.

이 예는 서버에서 추가 매개변수를 사용해 처리기 클래스를 초기화하는 데 사용할 수 있는 기법도 보여준다. 보통 서버는 `__init__()`에 전달되는 미리 정의된 인수들을 사용해 처리기를 생성한다. 매개변수를 추가하고 싶다면 앞에 나온 것처럼 `functools.partial()` 함수를 사용하면 된다. 이렇게 하면 서버에서 기대하는 호출 시 그녀처를 유지하면서도 여러분의 추가 매개변수를 담아서 호출 가능한 객체를 만들 수 있다.

http.cookies(Cookie)

`http.cookies` 모듈은 서버 측에서 HTTP 쿠키를 처리하는 기능을 제공한다. 파이썬 2에서 이 모듈의 이름은 `Cookie`이다.

쿠키는 서버에서 세션, 사용자 로그인 및 기타 기능을 구현하기 위해 상태를 관리하는 데 사용한다. 사용자 브라우저에 쿠키를 떨어뜨리기 위해 HTTP 서버에서는 보통 HTTP 응답에 다음과 비슷한 HTTP 헤더를 추가한다.

```
Set-Cookie: session=8273612; expires=Sun, 18-Feb-2001 15:00:00 GMT; \
path=/; domain=foo.bar.com
```

아니면 HTML 문서에 `<head>` 절에 자바스크립트를 포함시켜서 쿠키를 설정할 수도 있다.

```
<SCRIPT LANGUAGE="JavaScript">
document.cookie = "session=8273612; expires=Sun, 18-Feb-2001
15:00:00 GMT; \
Feb 17; Path=/; Domain=foo.bar.com;"
</SCRIPT>
```

http.cookies 모듈은 모르셀(morsel)이라고 부르는 쿠키 값들을 저장하고 관리하는 데 사용할 수 있는 특수한 사전 같은 객체를 제공하는데 이를 이용하면 손쉽게 쿠키를 생성할 수 있다. 각 모르셀은 이름, 값, 그리고 옵션인 브라우저에 제공할 메타데이터를 담은 속성들(expires, path, comment, domain, max-age, secure, version, httponly 등)을 가질 수 있다. 이름으로는 보통 “name” 같은 간단한 식별자를 사용하며 “expires”나 “path”처럼 메타데이터 이름들은 쓸 수 없다. 값은 보통 짧은 문자열이다. 쿠키를 생성하려면 다음과 같이 간단히 쿠키 객체를 하나 만들면 된다.

```
c = SimpleCookie()
```

그리고 나서 보통의 사전 할당 연산을 사용해 쿠키 값(모르셀)들을 설정한다.

```
c["session"] = 8273612
c["user"] = "beazley"
```

다음과 같이 특정 모르셀에 추가로 속성을 설정할 수도 있다.

```
c["session"]["path"] = "/"
c["session"]["domain"] = "foo.bar.com"
c["session"]["expires"] = "18-Feb-2001 15:00:00 GMT"
```

쿠키 데이터를 HTTP 헤더로 출력하려면 c.output() 메서드를 사용하면 된다. 다음 예를 보자.

```
print(c.output())
# 두 줄짜리 출력을 생성한다.
# Set-Cookie: session=8273612; expires=...; path=/; domain=...
# Set-Cookie: user=beazley
```

브라우저에서 HTTP 서버로 쿠키를 되돌려 보낼 때는 “session=8273612; user=beazley”처럼 키=값 쌍들을 담은 문자열로 인코딩한다. expires, path, domain 같은 옵션인 속성들은 되돌려 보내지 않는다. 쿠키 문자열은 보통 HTTP_COOKIE 환경 변수에 저장되고 CGI 응용 프로그램에서 이를 읽는다. 쿠키 값을 다시 얻으려면 다음과 비슷한 코드를 사용하면 된다.

```
c = SimpleCookie(os.environ["HTTP_COOKIE"])
session = c["session"].value
```

```
user = c["user"].value
```

이제 SimpleCookie 객체를 더 자세히 설명한다.

SimpleCookie([input])

쿠키 값을 단순 문자열로 저장하는 쿠키 객체를 정의한다.

쿠키 인스턴스 c는 다음 메서드들을 가진다.

c.output([attrs [,header [,sep]]])

HTTP 헤더에 쿠키 값을 설정하기 적합한 형태로 문자열을 생성한다. attrs는 포함시킬 옵션인 속성들을 나타낸다(“expires”, “path”, “domain” 등). 기본으로 모든 쿠키 속성이 포함된다. header는 사용할 HTTP 헤더이다(기본으로 ‘Set-Cookie:’). sep는 헤더들을 연결하는 데 사용할 문자이며 기본으로 줄바꿈 문자가 사용된다.

c.js_output([attrs])

자바스크립트를 지원하는 브라우저에서 실행할 경우 쿠키를 설정하는 자바스크립트 코드를 담은 문자열을 생성한다.

c.load(rawdata)

rawdata에 있는 데이터를 쿠키에 저장한다. rawdata가 문자열이면 CGI 프로그램에서 사용하는 HTTP_COOKIE 환경 변수와 같은 형식이어야 한다. rawdata가 사전이면 각 key/value 쌍을 `c[key]=value`로 저장한다.

내부적으로 쿠키 값을 나타내는 키/값 쌍은 Morsel 클래스 인스턴스이다. Morsel 인스턴스 m은 사전처럼 작동하며 옵션 키 “expires”, “path”, “comment”, “domain”, “max-age”, “secure”, “version”, “httponly”를 설정할 수도 있다. 또한 모르셀 m은 다음 메서드와 속성도 지원한다.

m.value

쿠키의 무가공 값을 담은 문자열.

m.coded_value

브라우저에 보내거나 브라우저로부터 받을 쿠키를 인코딩한 값을 담은 문자열.

m.key

쿠키 이름.

m.set(key, value, coded_value)

앞에 나온 m.key, m.value, m.coded_value 값을 설정한다.

m.isReservedKey(k)

k가 “expires”, “path”, “domain” 같은 예약된 키워드인지를 검사한다.

m.output([attrs [,header]])

이 모듈을 나타내는 HTTP 헤더를 출력한다. attrs는 포함시킬 추가 속성들을 나타낸다(“expires”, “path” 등). header는 사용할 헤더 문자열이다(기본으로 ‘Set-Cookie:’).

m.js_output([attrs])

실행되면 쿠키를 설정하는 자바스크립트 코드를 출력한다.

m.OutputString([attrs])

쿠키 문자열을 HTTP 헤더나 자바스크립트 코드 없이 반환한다.

예외

쿠키 값을 파싱하거나 생성할 때 에러가 발생하면 CookieError 예외가 발생한다.

http.cookiejar(cookielib)

http.cookiejar 모듈은 클라이언트 쪽에서 HTTP 쿠키를 저장하고 관리할 수 있는 기능을 제공한다. 파이썬 2에서 이 모듈의 이름은 cookielib이다.

이 모듈은 주로 인터넷에 있는 문서에 접근하는 데 사용하는 urllib 패키지와 함께 사용되며 HTTP 쿠키를 저장하는 데 사용할 수 있는 객체들을 제공한다. 예를 들어, http.cookiejar 모듈은 쿠키를 받았다가 이어지는 요청에서 다시 보내는 데 사용할 수 있다. 또한 다양한 브라우저에서 쿠키를 저장하기 위해 생성하는 파일들을 다루는 데 사용할 수도 있다.

이 모듈에는 다음 객체들이 정의되어 있다.

CookieJar()

HTTP 쿠키 값을 관리하고 HTTP 요청의 결과로 받은 쿠키를 저장하며 나가는 HTTP 요청에 쿠키를 추가하는 데 사용할 수 있는 객체. 쿠키는 전부 메모리에 저

장되며 CookieJar 인스턴스가 쓰레기 수집되면 없어진다.

FileCookieJar(filename [, delayload])

쿠키 정보를 파일에서 추출하거나 파일에 저장하는 데 사용할 수 있는 FileCookieJar 인스턴스를 생성한다. filename은 파일의 이름이다. delayload는 True로 설정하면 파일에 게으른(lazy) 접근을 활성화한다. 즉, 요청이 있을 경우에만 파일에 접근한다.

MozillaCookieJar(filename [, delayload])

모질라의 cookies.txt 파일과 호환되는 FileCookieJar 인스턴스를 생성한다.

LWPCookieJar(filename [, delayload])

libwww-perl의 Set-Cookie3 파일 형식과 호환되는 FileCookieJar 인스턴스를 생성한다.

이 객체들의 메서드나 속성에 직접 접근하는 일은 드물다. 저수준 프로그래밍 인터페이스를 알아야 할 필요가 있다면 온라인 문서를 살펴보기 바란다. 보통은 이 중 하나의 인스턴스를 생성한 다음에 이것을 쿠키를 사용하고자 하는 다른 부분에서 쓰는 경우가 더 흔하다. 이 장의 urllib.request 절에 한 예가 나와 있다.

smtplib

smtplib 모듈은 RFC 821과 RFC 1869에 설명되어 있는 SMTP 프로토콜을 사용해 메일을 보내는 데 사용할 수 있는 저수준 SMTP 클라이언트 인터페이스를 제공한다. 이 모듈은 온라인 문서에 더 자세하게 설명되어 있는 몇 가지 함수와 메서드를 담고 있다. 여기서는 핵심적인 내용만을 다룬다.

SMTP([host [, port]])

SMTP 서버로의 연결을 나타내는 객체를 생성한다. host를 주면 SMTP 서버의 이름을 가리킨다. port는 옵션인 포트 번호이다. 기본 포트는 25이다. host를 주면 connect() 메서드가 자동으로 호출된다. 아니면 연결을 맺기 위해 반환되는 객체에 직접 connect()를 호출해주어야 한다.

SMTP의 인스턴스 s는 다음 메서드들을 가진다.

s.connect([host [, port]])

host에 있는 SMTP 서버에 연결한다. host를 생략하면 지역 호스트 ('127.0.0.1')로

연결한다. port는 옵션인 포트 번호이며 생략할 경우 25로 설정된다. SMTP()에 호스트 이름을 주었으면 connect()를 호출할 필요 없다.

s.login(user, password)

인증이 필요할 경우 서버에 로그인한다. user는 사용자 이름, password는 암호다.

s.quit()

서버에 ‘QUIT’ 명령을 보내 세션을 종료한다.

s.sendmail(fromaddr, toaddrs, message)

서버에 메일 메시지를 보낸다. fromaddr는 보내는 사람의 이메일 주소를 담은 문자열이다. toaddrs는 받는 사람의 이메일 주소를 담은 문자열들의 리스트이다. message는 RFC-822 형식을 제대로 갖춘 메시지를 담은 문자열이다. 보통 메시지를 생성하는 데 email 패키지를 사용한다. message를 텍스트 문자열로 줄 수 있지만 오직 0에서 127 범위에 있는 값을 가지는 유효한 ASCII 문자만 담을 수 있다. 아니면 인코딩 에러가 발생한다. UTF-8 같은 다른 인코딩으로 메시지를 보내려면 먼저 메시지를 바이트 문자열로 인코딩하고 그것을 message에 지정하면 된다.

예

다음 예는 이 모듈을 사용해 메시지를 보내는 방법을 보여준다.

```
import smtplib
fromaddr = "someone@some.com"
toaddrs = ["recipient@other.com"]
msg = "From: %s\r\nTo: %s\r\n\r\n" % (fromaddr, ",".join(toaddrs))
msg += """
Refinance your mortgage to buy stocks and Viagra!
"""
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

urllib 패키지

urllib 패키지는 HTTP 서버, FTP 서버 및 지역 파일과 상호작용할 필요가 있는 클라이언트를 작성하는 데 사용할 수 있는 고수준 인터페이스를 제공한다. 이 모듈은 웹 페이지, 자동화 도구, 프록시, 웹 크롤러 등에서 데이터를 긁어오는 데 주로 사용한다. 이 모듈은 상당히 많은 설정 기능을 제공하는데 이곳에서 모든 내용을 다루

진 않겠다. 대신 가장 흔히 사용되는 기능을 설명한다.

파이썬 2에서는 urllib의 기능이 urllib, urllib2, urlparse, robotparser 등 몇 개의 라이브러리 모듈에 흩어져 있다. 파이썬 3에서는 모든 것이 urllib 패키지로 통합되고 재조직되었다.

urllib.request(urllib2)

urllib.request 모듈은 URL을 열고 데이터를 얻는 데 사용할 수 있는 함수와 클래스를 제공한다. 파이썬 2에서는 같은 기능이 urllib2 모듈에 있다.

이 모듈은 HTTP를 통해 웹 서버에서 데이터를 얻는 데 가장 많이 사용한다. 예를 들어, 다음 코드는 웹 페이지를 얻어오는 가장 간단한 방법을 보여준다.

```
try:
    from urllib.request import urlopen # 파이썬 3
except ImportError:
    from urllib2 import urlopen       # 파이썬 2

u = urlopen("http://docs.python.org/3.0/library/urllib.request.html")
data = u.read()
```

물론 실전에서는 서버와 상호작용할 때 복잡한 일이 벌어질 수 있다. 예를 들어, 프록시 서버, 인증, 쿠키, 사용자 에이전트 등에 관해서 신경 써야 할지도 모른다. 이 모든 기능이 지원되지만 코드가 복잡해진다. 일단 계속 읽어보기 바란다.

urlopen()과 요청

요청을 보내는 가장 쉬운 방법은 urlopen() 함수를 사용하는 것이다.

urlopen(url [, data [, timeout]])

URL url을 열고 반환된 데이터를 읽는 데 사용할 수 있는 파일 같은 객체를 반환 한다. url은 URL을 담은 문자열이거나 곧 설명할 Request 클래스의 인스턴스일 수 있다. data는 서버로 업로드할 폼 데이터를 담은 URL 인코딩된 문자열이다. 주어질 경우 HTTP ‘GET’ 방식(기본) 대신 HTTP ‘POST’ 방식이 사용된다. 보통 데이터는 urllib.parse.urlencode() 같은 함수를 사용해 생성한다. 옵션인 timeout은 내부에서 사용하는 모든 블로킹 연산에 사용할 타임아웃 시간을 초로 나타낸다.

urlopen()이 반환하는 파일 같은 객체 u는 다음 메서드들을 지원한다.

메서드	설명
u.read([nbytes])	nbytes의 데이터를 바이트 문자열로 읽는다.
u.readline()	한 줄의 텍스트를 바이트 문자열로 읽는다.
u.readlines()	모든 줄을 읽어서 리스트로 반환한다.
u.fileno()	정수 파일 기술자를 반환한다.
u.close()	연결을 닫는다.
u.info()	URL에 연관된 메타 정보를 담은 매팽 객체를 반환한다. HTTP에 대해서는 서버 응답에 담긴 HTTP 헤더들이 반환된다. FTP에 대해서는 헤더에 'content-length'도 포함된다. 지역 파일에 대해서는 헤더에 날짜, 'content-length', 'content-type' 필드가 포함된다.
u.getcode()	HTTP 응답 코드를 정수로 반환한다. 예를 들어, 성공은 200, 파일을 찾을 수 없음은 404이다.
u.geturl()	반환된 데이터의 실제 URL을 반환한다. 페이지 전환(redirection)이 일어난 것을 고려한다.

파일 같은 객체 u는 이진 모드에서 작동한다. 응답 데이터를 텍스트로 처리하려면 codecs 모듈이나 기타 다른 방법으로 디코딩을 해야 한다.

다운로드 과정에서 에러가 발생하면 URLError 예외가 발생한다. 발생할 수 있는 에러에는 접근 금지 또는 인증 요청 같은 HTTP 프로토콜 자체와 관련된 에러가 포함된다. 이러한 종류의 에러에 대해서는 보통 서버에서 더 구체적인 정보를 담은 내용을 보낸다. 이 내용을 얻으려면 예외 자체를 파일 같은 객체로 다루면 된다. 다음 예를 보자.

```
try:
    u = urlopen("http://www.python.org/perl.html")
    resp = u.read()
except HTTPError as e:
    resp = e.read()
```

urlopen()을 사용할 때 프록시 서버를 통해 웹 페이지에 접근하려고 하는 경우 예외가 흔히 발생한다. 예를 들어, 여러분이 몸담고 있는 조직에서 모든 웹 트래픽을 프록시를 통하게 해놓았다면 요청이 실패할 수 있다. 프록시 서버에서 인증을 요구하지 않는 경우에는 단순히 os.environ 사전에 있는 HTTP_PROXY 환경 변수를 설정하는 것만으로 이 문제를 해결할 수 있다. 예를 들어, os.environ['HTTP_PROXY'] = 'http://example.com:12345'.

간단한 요청인 경우 urlopen()의 url 매개변수는 'http://www.python.org' 같은 문자열을 담게 된다. HTTP 요청 헤더를 변경하는 것 같은 더 복잡한 작업을 수행하려면 Request 인스턴스를 생성하고 url 매개변수로 전달하면 된다.

Request(url [, data [, headers [, origin_req_host [, unverifiable]]]])

새로운 Request 인스턴스를 생성한다. url은 URL을 지정한다(예를 들어, ‘http://www.foo.bar/spam.html’). data는 HTTP 요청으로 서버에 업로드할 URL 인코딩된 데이터이다. 이것을 제공하면 HTTP 요청 타입이 ‘GET’에서 ‘POST’로 바뀐다. headers는 HTTP 헤더의 내용을 표현하는 키, 값 쌍들을 담은 사전이다. origin_req_host는 트랜잭션 요청 호스트로 설정한다. 보통 처음 요청을 보내온 호스트 이름이다. unverifiable은 검증할 수 없는 URL에 대한 요청일 경우 True로 설정한다. 검증할 수 없는 URL이란 비공식적으로 말하자면 사용자가 직접 입력하지 않은 URL이다. 예를 들어, 페이지 안에 들어 있는 이미지를 불러오는 URL 같은 것을 말한다. unverifiable의 기본 값은 False이다.

Request 인스턴스 r은 다음 메서드들을 가진다.

r.add_data(data)

요청에 데이터를 추가한다. 요청이 HTTP 요청이면 요청 방식이 ‘POST’로 바뀐다. data는 Request()에서 설명한 것처럼 URL 인코딩된 데이터이다. 데이터를 이전에 설정한 데이터에 덧붙이지 않고 간단히 이전 데이터를 data로 대체한다.

r.add_header(key, val)

요청에 헤더 정보를 추가한다. key는 헤더 이름이고 val는 헤더 값이다. 두 인수 모두 문자열이다.

r.add_unredirected_header(key, val)

페이지 전환이 이루어진 요청에는 추가되지 않을 헤더 정보를 요청에 추가한다. key와 val는 add_header()에서 의미와 같다.

r.get_data()

있을 경우 요청 데이터를 반환한다.

r.get_full_url()

요청의 전체 URL을 반환한다.

r.get_host()

요청을 보낼 호스트를 반환한다.

r.get_method()

‘GET’과 ‘POST’ 중 하나인 HTTP 요청 방식을 반환한다.

r.get_origin_req_host()

트랜잭션을 시작시킨 요청 호스트를 반환한다.

r.get_selector()

URL에서 선택기 부분을 반환한다(예를 들어, ‘/index.html’).

r.get_type()

URL 타입을 반환한다(예를 들어, ‘http’).

r.has_data()

요청에 데이터가 포함된 경우 True를 반환한다.

r.is_unverifiable()

요청을 검증할 수 없는 경우 True를 반환한다.

r.has_header(header)

요청이 헤더 header를 가지고 있는 경우 True를 반환한다.

r.set_proxy(host, type)

요청을 프록시 서버에 연결할 수 있게 준비한다. 초기 호스트를 host로 바꾸고 초기 요청 타입을 type으로 바꾼다. URL에서 선택기 부분은 초기 URL로 설정된다.

다음은 Request 객체를 사용해 urlopen()이 사용하는 ‘User-Agent’ 헤더를 변경하는 예를 보여준다. 서버가 여러분이 현재 인터넷 익스플로러, 파이어폭스 또는 기타 다른 브라우저를 사용하고 있다고 생각하게 만들고자 할 때 활용할 수 있는 예이다.

```
headers = {
    'User-Agent':
        'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
        .NET CLR 2.0.50727)'
}

r = Request("http://somedomain.com/", headers=headers)
u = urlopen(r)
```

커스텀 오프너

기본 `urlopen()` 함수는 인증, 쿠키 또는 기타 HTTP의 고급 기능을 지원하지 않는다. 이런 기능을 원한다면 `build_opener()` 함수를 사용해 여러분만의 오프너 객체를 만들어야 한다.

```
build_opener([handler1 [, handler2, ... ]])
```

URL을 여는 데 사용할 수 있는 커스텀 오프너 객체를 생성한다. `handler1`, `handler2` 등의 인수들은 모두 특수한 처리기 객체 인스턴스이다. 처리기는 생성되는 오프너 객체에 다양한 기능을 추가하는 데 사용한다. 다음 목록은 사용 가능한 모든 처리기 객체를 보여준다.

처리기	설명
<code>CacheFTPHandler</code>	지속되는 FTP 연결을 제공하는 FTP 처리기
<code>FileHandler</code>	지역 파일을 연다.
<code>FTPHandler</code>	URL을 FTP를 통해 연다.
<code>HTTPBasicAuthHandler</code>	기본 HTTP 인증 처리
<code>HTTPCookieProcessor</code>	HTTP 쿠키를 처리한다.
<code>HTTPDefaultErrorHandler</code>	HTTPError 예외를 발생시킴으로써 HTTP 에러를 처리한다.
<code>HTTPDigestAuthHandler</code>	HTTP 요약(digest) 인증 처리
<code>HTTPHandler</code>	URL을 HTTP를 통해 연다.
<code>HTTPRedirectHandler</code>	HTTP 페이지 전환을 처리한다.
<code>HTTPSHandler</code>	URL을 보안 HTTP를 통해 연다.
<code>ProxyHandler</code>	요청이 프록시를 거치게 만든다.
<code>ProxyBasicAuthHandler</code>	기본 프록시 인증
<code>ProxyDigestAuthHandler</code>	요약 프록시 인증
<code>UnknownHandler</code>	모든 알 수 없는 URL을 처리하는 처리기

기본으로 오프너는 처리기 `ProxyHandler`, `UnknownHandler`, `HTTPHandler`, `HTTPSHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, 그리고 `HTTPErrorProcessor`를 가진 채 생성된다. 이 처리기들은 기본 기능을 제공한다. 인수로 추가 처리기를 제공하면 여기에 추가된다. 추가로 제공한 처리기가 기존 처리기와 같은 종류일 경우 새로운 처리기가 우선한다. 예를 들어, `HTTPHandler` 또는 `HTTPHandler`에서 상속받은 클래스의 인스턴스를 추가하면 기본 값 대신 사용된다.

`build_opener()`가 반환하는 객체는 다양한 처리기가 제공하는 규칙에 따라 URL

을 열고자 할 때 사용하는 `open(url [, data [, timeout]])` 메서드 하나를 가진다. `open()`에 전달하는 인수는 `urlopen()` 함수에 전달하는 것과 같다.

install_opener(opener)

`urlopen()`이 사용하는 전역 URL 오프너로서 사용할 오프너 객체를 설치한다. `opener`는 보통 `build_opener()`로 생성한 오프너 객체이다.

이어지는 몇 개의 절에서는 `urllib.request` 모듈을 사용할 때 흔히 접하게 되는 시나리오에 대해 커스텀 오프너를 정의하는 방법을 설명한다.

암호 인증

암호 인증이 필요한 요청을 처리할 때는 오프너에 `HTTPBasicAuthHandler`, `HTTPDigestAuthHandler`, `ProxyBasicAuthHandler`나 `ProxyDigestAuthHandler` 처리기를 적절히 조합해 사용하면 된다. 각 처리기는 암호를 설정하는 데 사용하는 다음 메서드를 가진다.

h.add_password(realm, uri, user, passwd)

주어진 영역(`realm`)과 URL에 대해 사용자와 암호 정보를 추가한다. 모든 매개변수는 문자열이다. `uri`는 URI들의 순서열일 수 있다. 이 경우 순서열에 들어 있는 모든 URI에 대해 사용자와 암호 정보가 적용된다. `realm`은 인증과 연관된 이름이나 설명이다. 보통은 관련 웹 페이지들의 공통 이름을 나타낸다. `uri`는 인증과 연관된 기반 URL이다. `realm`과 `uri`는 보통 ‘Administrator’, ‘`http://www.somesite.com`’ 같은 값을 가진다. `user`와 `password`는 사용자 이름과 암호를 각각 지정한다.

다음 예는 기본 인증을 수행하는 오프너를 설정하는 방법을 보여준다.

```
auth = HTTPBasicAuthHandler()
auth.add_password("Administrator", "http://www.secretlair.com",
                  "drevil", "12345")

# 인증 기능이 추가된 오프너를 생성한다.
opener = build_opener(auth)

# URL을 연다.
u = opener.open("http://www.secretlair.com/evilplan.html")
```

HTTP 쿠키

HTTP 쿠키를 관리하려면 오프너 객체에 `HTTPCookieProcessor` 처리기를 추가해서

열면 된다. 다음 예를 보자.

```
cookiehand = HTTPCookieProcessor()
opener = build_opener(cookiehand)
u = opener.open("http://www.example.com/")
```

기본으로 `HTTPCookieProcessor`는 `http.cookiejar` 모듈에 있는 `CookieJar` 객체를 사용한다. `HTTPCookieProcessor`에 다른 `CookieJar` 객체를 제공하여 다른 방식으로 쿠키를 처리할 수도 있다. 다음 예를 보자.

```
cookiehand = HTTPCookieProcessor(
    http.cookiejar.MozillaCookieJar("cookies.txt")
)
opener = build_opener(cookiehand)
u = opener.open("http://www.example.com/")
```

프록시

요청을 프록시를 통하여도록 전환하려면 `ProxyHandler` 인스턴스를 생성하면 된다.

`ProxyHandler([proxies])`

요청이 프록시를 통하여도록 만드는 프록시 처리기를 생성한다. 인수 `proxies`는 프로토콜 이름(‘`http`’, ‘`ftp`’ 등)을 해당 프록시 서버의 URL로 매핑하는 사전이다.

다음은 이 처리기를 어떻게 사용하는지 보여준다.

```
proxy = ProxyHandler({'http': 'http://someproxy.com:8080/'})
auth = HTTPBasicAuthHandler()
auth.add_password("realm", "host", "username", "password")
opener = build_opener(proxy, auth)

u = opener.open("http://www.example.com/doc.html")
```

`urllib.response`

`urllib.request` 모듈에 있는 함수가 반환하는 파일 같은 객체를 구현하는 내부 모듈이다. 공개 API는 없다.

`urllib.parse`

`urllib.parse` 모듈은 “`http://www.python.org`” 같은 URL 문자열을 조작하는 데 사용한다.

URL 파싱(파이썬 2에서는 urlparse 모듈)

URL의 일반적인 형식은 “scheme://netloc/path;parameters?query#fragment”이다. URL의 netloc 부분은 “hostname:port”처럼 포트 번호를 포함하거나 “user:pass@hostname”처럼 사용자 인증 정보를 담을 수 있다. 다음 함수는 URL을 파싱하는데 사용한다.

urlparse(urlstring [, default_scheme [, allow_fragments]])

urlstring에 있는 URL을 파싱하여 ParseResult 인스턴스를 반환한다. default_scheme은 URL에 지정하지 않은 경우 사용할 체계(“http”, “ftp” 등)를 지정한다. allow_fragments가 0이면 조각 식별자(fragment identifier)를 허용하지 않는다. ParseResult 인스턴스 r은 이름 있는 튜플 (scheme, netloc, path, parameters, query, fragment)이다. 이 인스턴스에는 다음 읽기 전용 속성들도 정의되어 있다.

속성	설명
r.scheme	URL 체계 지정자(예를 들어 ‘http’)
r.netloc	네트워크 위치(netloc) 지정자(예를 들어, ‘www.python.org’)
r.path	계층적 경로(예를 들어, ‘/index.html’)
r.params	경로 다음에 나오는 매개변수들(parameters 부분)
r.query	질의 문자열(예를 들어 ‘name=Dave&id=42’)
r.fragment	앞쪽 '#'를 뺀 조각 식별자
r.username	netloc 지정자가 ‘username:password@hostname’ 형식인 경우 사용자 이름 부분
r.password	netloc 지정자에서 암호 부분
r.hostname	netloc 지정자에서 호스트 이름 부분
r.port	netloc 지정자가 ‘hostname:port’ 형식이면 포트 번호

ParseResult 인스턴스는 r.geturl()로 다시 URL 문자열로 되돌릴 수 있다.

urlunparse(parts)

urlparse()가 반환하는 튜플로 표현된 URL에서 URL 문자열을 생성한다. parts는 여섯 부분으로 이루어진 튜플이나 반복 가능한 객체이어야 한다.

urlsplit(url [, default_scheme [, allow_fragments]])

URL의 매개변수(parameters) 부분을 그대로 둔다는 점을 제외하고 urlparse()와 같다. ‘scheme://netloc/path1;param1/path2;param2/path3?query#fragment’처럼 매개변수가 개별 경로 요소에 붙은 URL을 파싱할 수 있게 한다. 결과로서 이름 있는 튜플 (scheme, netloc, path, query, fragment)인 SplitResult 인스턴스를 반환한다.

이 인스턴스에는 다음 읽기 전용 속성들도 있다.

속성	설명
r.scheme	URL 체계 지정자(예를 들어, 'http')
r.netloc	네트워크 위치(netloc) 지정자(예를 들어, 'www.python.org')
r.path	계층적 경로(예를 들어, '/index.html')
r.query	질의 문자열(예를 들어, 'name=Dave&id=42')
r.fragment	앞쪽 '#'를 뺀 조각 식별자
r.username	netloc 지정자가 'username:password@hostname' 형식인 경우 사용자 이름 부분
r.password	netloc 지정자에서 암호 부분
r.hostname	netloc 지정자에서 호스트 이름 부분
r.port	netloc 지정자가 'hostname:port' 형식이면 포트 번호

SplitResult 인스턴스는 r.geturl()로 다시 URL 문자열로 되돌릴 수 있다.

urlunsplit(parts)

urlsplit()의 생성하는 튜플 표현에서 URL을 생성한다. parts는 다섯 부분으로 구성된 튜플이나 반복 가능한 객체일 수 있다.

urldefrag(url)

튜플 (newurl, fragment)를 반환한다. newurl은 url에서 조각들을 제거한 것이고 fragment는 조각 부분을 담은 문자열이다(있을 경우). url에 조각이 없으면 newurl은 url과 같고 fragment는 빈 문자열이 된다.

urljoin(base, url [, allow_fragments])

기반 URL base와 상대 URL url을 합쳐서 절대 URL을 생성한다. allow_fragments는 urlparse()에서와 같은 의미이다. 기반 URL의 마지막 부분이 디렉터리가 아니면 제거된다.

parse_qs(qs [, keep_blank_values [, strict_parsing]])

URL 인코딩된 질의 문자열 qs를 파싱하여 키가 질의 변수 이름이고 값이 해당 이름에 대해 정의된 값들의 리스트인 사전을 반환한다. keep_blank_values는 빈 값을 어떻게 처리할 것인지를 제어하는 불리언 플래그이다. True이면 빈 문자열로 설정해서 사전에 포함시킨다. False(기본 값)이면 버린다. strict_parsing은 True일 경우 파싱 에러를 ValueError 예외로 바꾸는 불리언 플래그이다. 기본으로 에러는 조용히 무시된다.

parse_qs [, keep_blank_values [, strict_parsing]]

결과가 (name, value) 쌍들의 리스트라는 점을 제외하고 parse_qs()와 같다. 여기서 name은 질의 변수의 이름을 value는 값을 나타낸다.

URL 인코딩(파이썬 2에서는 urllib 모듈)

다음 함수들은 URL을 구성하는 데이터를 인코딩하거나 디코딩하는 데 사용한다.

quote(string [, safe [, encoding [, errors]]])

string에 있는 특수 문자를 URL에 포함시키기 적당한 탈출 순서열로 대체한다. 문자, 숫자, 밑줄(underscore), 콤마(,), 마침표(.)와 하이픈(-) 문자는 그대로 둔다. 나머지 문자는 모두 '%xx' 형식의 탈출 순서열로 변환한다. safe는 변환하지 않을 추가 문자들을 담은 문자열이며 기본으로 '/'이다. encoding은 ASCII 아닌 문자에 사용할 인코딩을 지정한다. 기본은 'utf-8'이다. errors는 인코딩 에러가 발생했을 때 어떻게 할 것인지를 지정하며 기본으로 'strict'이다. encoding과 errors 매개변수는 파이썬 3에만 있다.

quote_plus(string [, safe [, encoding [, errors]]])

quote()를 호출하고 추가로 스페이스를 플러스 기호로 바꾼다. string과 safe는 quote()에서와 의미가 같다. encoding과 errors도 마찬가지이다.

quote_from_bytes(bytes [, safe])

quote()와 같지만 바이트 문자열을 받고 인코딩을 수행하지 않는다. 반환되는 결과는 텍스트 문자열이다. 파이썬 3에만 있다.

unquote(string [, encoding [, errors]])

'%xx' 형식의 탈출 순서열을 대응하는 단일 문자로 바꾼다. encoding과 errors는 '%xx' 탈출 순서열에 있는 데이터를 디코딩할 때 인코딩과 에러 처리를 어떻게 할 것인지를 지정한다. 기본 인코딩은 'utf-8'이고 기본 errors 정책은 'replace'이다. encoding과 errors는 파이썬 3에만 있다.

unquote_plus(string [, encoding [, errors]])

unquote()와 같지만 플러스 기호를 스페이스로 바꾸는 작업도 수행한다.

unquote_to_bytes(string)

unquote()와 같지만 디코딩을 수행하지 않고 바이트 문자열을 반환한다.

urlencode(query [, doseq])

query에 있는 질의 값들을 URL의 query 매개변수로 사용하거나 POST 요청의 일부로 업로드하기 적합하도록 URL 인코딩된 문자열로 변환한다. query는 사전 이거나 (key, value) 쌍들의 순서열이어야 한다. 결과 문자열은 ‘&’로 구분된 일련의 ‘key=value’ 쌍들이며 key와 value는 모두 quote_plus()로 처리된다. doseq 매개변수는 query에 있는 어떤 값이 같은 키에 대한 여러 값을 나타내기 위해 순서열로 되어 있는 경우 True로 설정해야 한다. 이 경우 value에 있는 각 v에 대해 별개의 ‘key=v’ 문자열이 생성된다.

예

다음 예는 질의 변수들을 담은 사전을 HTTP GET 요청을 통해 보내기 적합하도록 변환하는 방법과 URL을 파싱하는 방법을 보여준다.

```
try:
    from urllib.parse import urlparse, urlencode, parse_qs # 파일 3
except ImportError:
    from urlparse import urlparse, parse_qs # 파일 2
from urllib import urlencode

# 적절히 인코딩된 질의 변수들을 담은 URL을 생성하는 예
form_fields = {
    'name' : 'Dave',
    'email' : 'dave@dabeaz.com',
    'uid' : '12345'
}
form_data = urlencode(form_fields)
url = "http://www.somehost.com/cgi-bin/view.py?" + form_data

# URL을 파싱하여 여러 부분으로 나누는 예
r = urlparse(url)
print(r.scheme) # 'http'
print(r.netloc) # 'www.somehost.com'
print(r.path) # '/cgi-bin/view.py'
print(r.params) # ''
print(r.query) # 'uid=12345&name=Dave&email=dave%40dabeaz.com'
print(r.fragment) # ''

# 질의 데이터를 추출한다.
parsed_fields = dict(parse_qs(r.query))
assert form_fields == parsed_fields
```

urllib.error

urllib.error 모듈은 urllib 패키지에서 사용하는 예외들을 정의한다.

ContentTooShort

다운로드된 데이터의 양이 예상한 것('Content-Length' 헤더로 정의됨)보다 적은 경우 발생한다. 파이썬 2에서는 urllib 모듈에 정의되어 있다.

HTTPError

HTTP 프로토콜과 관련된 문제를 나타낸다. 이 에러는 인증 요구 같은 이벤트를 알리는 데도 사용될 수 있다. 이 예외는 에러와 관련해 서버가 반환한 데이터를 읽기 위해 파일 객체로 사용할 수도 있다. URLError의 하위 클래스이다. 파이썬 2에서는 urllib2 모듈에 정의되어 있다.

URLError

문제가 발생했을 때 처리기가 발생시키는 에러. IOError의 하위 클래스이다. 예외 인스턴스의 reason 속성은 발생된 문제에 대한 정보를 담는다. 파이썬 2에서는 urllib2 모듈에 정의되어 있다.

urllib.robotparser(robotparser)

urllib.robotparser 모듈(파이썬 2에서는 robotparser)은 웹 크롤러에게 알릴 지시 사항을 담은 'robots.txt' 파일의 내용을 얻어오거나 파싱하는 데 사용한다. 사용법은 온라인 문서를 참고하기 바란다.

Note

- 고급 사용자는 상상할 수 있는 모든 방향으로 urllib 패키지를 커스터마이즈할 수 있다. 예를 들어, 새로운 오피너, 처리기, 요청, 프로토콜 등을 생성할 수 있다. 이러한 내용을 이 책에서 모두 다루기는 힘들다. 자세한 내용은 온라인 문서를 찾아보기 바란다.
- 파이썬 2 사용자는 이미 많은 곳에서 사용되고 있는 urllib.urlopen() 함수가 파이썬 2.6에서는 공식적으로 사용이 권장되지 않고 파이썬 3에서는 제거되었다는 점을 유념하기 바란다. urllib.urlopen() 대신 이곳에서 설명한 urllib.request.urlopen()와 동일한 기능을 제공하는 urllib2.urlopen()을 사용해야 한다.

xmlrpc 패키지

xmlrpc 패키지는 XML-RPC 서버와 클라이언트를 구현하기 위한 모듈을 담고 있다. XML-RPC는 데이터 인코딩에 XML을 사용하고 전송 메커니즘으로 HTTP를 사용하는 원격 프로시저 호출 메커니즘이다. 내부 프로토콜이 파이썬에 국한된 것이 아니기 때문에 이곳에서 설명할 모듈로 작성한 프로그램은 다른 언어로 작성한 프로그램과 상호작용할 수 있다. XML-RPC에 관한 더 자세한 내용은 <http://www.xmlrpc.com>에서 찾을 수 있다.

`xmlrpc.client(xmlrpclib)`

`xmlrpc.client` 모듈은 XML-RPC 클라이언트를 작성하는 데 사용한다. 파이썬 2에서는 이름이 `xmlrpclib`이다. 클라이언트 역할을 수행하려면 `ServerProxy` 인스턴스를 생성하면 된다.

```
ServerProxy(uri [, transport [, encoding [, verbose [, allow_none [, use_datetime]]]])
```

`uri`는 원격 XML-RPC 서버의 위치를 나타낸다. 예를 들어, “`http://www.foo.com/RPC2`”. 필요한 경우 기본 인증 정보를 “`http://user:pass@host:port/path`”의 형식으로 URI에 붙일 수 있다. 여기서 `user:pass`는 사용자 이름과 암호를 나타낸다. 이 인증 정보는 base 64로 인코딩되어 ‘Authorization:’ 헤더로 전송된다. 파이썬을 OpenSSL을 지원하도록 설정했다면 HTTPS도 사용할 수 있다. `transport`는 저수준 통신에 사용할 내부 전송 객체를 생성하는 데 사용할 팩토리 함수를 지정한다. 이 인수는 XML-RPC를 HTTP나 HTTPS가 아닌 다른 종류의 연결 위에서 사용하고자 할 때 지정한다. 보통은 이 인수를 지정할 일이 거의 없다(자세한 내용은 온라인 문서를 참고한다). `encoding`은 인코딩을 지정하며 기본으로 UTF-8이다. `verbose`는 `True`로 설정하면 디버깅 정보를 출력한다. `allow_none`은 `True`로 설정하면 `None` 값을 원격 서버로 보낼 수 있게 허용한다. 어디서나 지원하는 기능이 아니므로 기본으로 비활성화된다. `use_datetime`은 `True`로 설정할 경우 날짜와 시간을 표현하는데 `datetime` 모듈을 사용하게 하는 블리언 플래그이다. 기본으로 `False`이다.

`ServerProxy` 인스턴스 `s`는 원격 서버에 있는 모든 메서드를 그대로 노출시킨다. `s`의 속성을 통해 메서드에 접근할 수 있다. 예를 들어, 다음 코드는 서비스를 제공하는 원격 서버에서 현재 시간을 얻는다.

```
>>> s = ServerProxy("http://www.xmlrpc.com/RPC2")
>>> s.currentTime.getCurrentTime()
<DateTime u'20051102T20:08:24' at 2c77d8>
>>>
```

대개 RPC 호출은 평범한 파이썬 함수처럼 작동한다. 그렇지만 XML-RPC 프로토콜은 제한된 수의 인수 타입과 반환 값만을 지원한다.

XML-RPC 타입	파이썬 대응
boolean	True와 False
integer	int
float	float
string	string이나 unicode(XML에서 유효한 문자만 담아야 한다)
array	유효한 XML-RPC 타입을 담은 순서열
structure	문자열 키와 유효한 타입의 값을 담은 사전
dates	날짜와 시간(xmlrpc.client.DateTime)
binary	이진 데이터(xmlrpc.client.Binary)

날짜 값은 xmlrpc.client.DateTime 인스턴스 d로 저장한다. d.value 속성은 날짜를 ISO 8601 시간/날짜 문자열로 저장한다. 이것을 time 모듈과 호환되는 시간 튜플로 변환하려면 d.timetuple()를 사용하면 된다. 이진 데이터는 xmlrpc.client. Binary 인스턴스 b로 저장한다. b.data 속성은 데이터를 이진 문자열로 저장한다. 문자열은 유니코드인 것으로 가정하므로 적절한 인코딩을 사용하도록 주의를 기울여야 한다. 무가공 파이썬 2 바이트 문자열을 보낼 경우 ASCII 문자를 담고 있는 경우에만 제대로 작동하고 다른 경우에는 실패한다. 문제가 없게 하려면 먼저 유니코드 문자열로 변환해야 한다.

유효하지 않은 타입의 인수로 RPC 호출을 시도하면 TypeError나 xmlrpclib.Fault 예외가 발생할 수 있다.

원격 XML-RPC 서버가 관찰(introspection) 기능을 지원하면 다음 메서드들을 사용할 수 있다.

s.system.listMethods()

XML-RPC 서버가 제공하는 모든 메서드를 나열하는 문자열들의 리스트를 반환한다.

s.methodSignatures(name)

메서드 이름 name이 주어질 때 해당 메서드의 모든 호출 시그너처 리스트를 반환한다. 각 시그너처는 콤마로 구분된 문자열 형식으로 표현된 타입들의 리스트이

다(예를 들어, ‘string, int, int’). 여기서 첫 번째 항목은 반환 타입이고 나머지는 인수 타입이다. 오버로딩 때문에 여러 시그너처가 반환될 수 있다. 파일로 구현된 XML-RPC 서버의 경우 함수와 메서드가 동적으로 타입이 결정되기 때문에 시그너처는 보통 빈 값이다.

s.methodHelp(name)

메서드 이름 name이 주어질 때 해당 메서드의 사용법을 설명하는 문서화 문자열을 반환한다. 문서화 문자열은 HTML 마크업을 담을 수도 있다. 문서화 정보가 없으면 빈 문자열이 반환된다.

xmlrpclib 모듈에는 다음 유ти리티 함수들이 있다.

boolean(value)

value에서 XML-RPC 불리언 객체를 생성한다. 이 함수는 파일에 불리언 타입이 없을 때부터 있었고 오래된 코드에서 볼 수 있다.

Binary(data)

이진 데이터를 담은 XML-RPC 객체를 생성한다. data는 무가공 데이터를 담은 문자열이다. Binary 인스턴스를 반환한다. 반환되는 Binary 인스턴스는 전송 과정에서 base 64로 알아서 인코딩되고 디코딩된다. Binary 인스턴스 b에서 데이터를 추출하려면 b.data를 사용한다.

DateTime(daytime)

날짜를 담은 XML-RPC 객체를 생성한다. daytime은 ISO 8601 형식의 날짜 문자열, 시간 튜플, time.localtime()이 반환하는 구조, 또는 datetime 모듈에 있는 datetime 인스턴스 중 하나가 될 수 있다.

dumps(params [, methodname [, methodresponse [, encoding [, allow_none]]]])

params를 XML-RPC 요청 또는 응답으로 변환한다. params는 인수들을 담은 튜플이거나 Fault 예외의 인스턴스이다. methodname은 문자열인 메서드 이름이다. methodresponse는 불리언 플래그이다. True이면 결과가 XML-RPC 응답이다. 이 경우 params에 값 하나만 지정해야 한다. encoding은 생성된 XML의 텍스트 인코딩을 지정하며 기본으로 UTF-8이다. allow_none은 매개변수 타입으로 None을 지원할지 여부를 지정하는 불리언 플래그이다. None은 XML-RPC 명세서에 직접 언급되

어 있지는 않지만 많은 서버에서 지원한다. 기본으로 allow_none은 False이다.

Loads(data)

XML-RPC 요청이나 응답을 담은 data를 튜플 (params, methodname)로 변환한다. params는 매개변수들의 튜플이고 methodname은 메서드 이름을 담은 문자열이다. 요청이 실제 값이 아니라 실패 조건을 나타내는 경우 Fault 예외가 발생한다.

MultiCall(server)

여러 XML-RPC 요청을 묶어 하나의 요청으로 보내는 MultiCall 객체를 생성한다. 같은 서버에 여러 RPC 요청을 보내야 하는 경우 성능 최적화에 도움이 된다. server는 원격 서버로 연결을 나타내는 ServerProxy 인스턴스이다. 반환되는 MultiCall 객체는 ServerProxy와 같은 방식으로 사용한다. 이 객체는 원격 메서드를 바로 실행하지 않고 자신을 함수로서 호출하기 전까지 메서드 호출들을 큐에 저장한다. 그리고 나서 RPC 요청들을 전송한다. 이 연산의 반환 값은 각 RPC 연산의 반환 결과를 차례대로 생성하는 생성기이다. 원격 서버가 system.multicall() 메서드를 제공할 때만 MultiCall()을 사용할 수 있다.

다음 예는 MultiCall을 사용하는 방법을 보여준다.

```
multi = MultiCall(server)
multi.foo(4,6,7)      # 원격 메서드 foo
multi.bar("hello world") # 원격 메서드 bar
multi.spam()          # 원격 메서드 spam
# 이제 실제로 RPC 요청을 보내고 반환되는 결과들을 얻는다
foo_result, bar_result, spam_result = multi()
```

예외

xmlrpc.client에는 다음 예외들이 정의되어 있다.

Fault

XML-RPC 실패를 나타낸다. faultCode 속성은 실패 타입을 담은 문자열이다. faultString은 실패와 관련된 설명을 담은 메시지이다.

ProtocolError

내부 네트워크에서 발생한 문제를 나타낸다. 예를 들어, URL이 잘못되었거나 연결에 문제가 있는 경우 등이 있다. url 속성은 에러를 일으킨 URI를 담는다. errcode 속성은 에러 코드를 담는다. errmsg 속성은 설명을 담은 문자열이다. headers 속성은 에러를 일으킨 요청의 모든 헤더를 담는다.

xmlrpc.server(SimpleXMLRPCServer, DocXMLRPCServer)

xmlrpc.server 모듈은 다양한 XML-RPC 서버를 구현하는 데 사용할 수 있는 클래스들을 제공한다. 파이썬 2에서는 두 개의 모듈 SimpleXMLRPCServer와 DocXMLRPCServer에 나누어져 있다.

SimpleXMLRPCServer(addr [, requestHandler [, logRequests]])

소켓 주소 addr에서 듣는 XML-RPC 서버를 생성한다(예를 들어, ('localhost', 8080)). requestHandler는 연결이 들어올 때 요청 처리기 객체를 생성하는 팩토리 함수이다. 사용할 수 있는 유일한 처리기인 SimpleXMLRPCRequestHandler를 기본으로 사용한다. logRequests는 들어오는 요청을 기록할 것인지를 나타내는 불리언 플래그이다. 기본으로 True이다.

DocXMLRPCServer(addr [, requestHandler [, logRequest]])

추가로 HTTP GET 요청에 응답하는 문서화 XML-RPC를 생성한다(보통 브라우저가 보낸다). 이런 요청을 받으면 서버는 등록된 모든 메서드와 객체의 문서화 문자열로부터 문서를 생성한다. 인수들은 SimpleXMLRPCServer에서 의미와 같다.

SimpleXMLRPCServer나 DocXMLRPCServer의 인스턴스 s는 다음 메서드들을 가진다.

s.register_function(func [, name])

XML-RPC 서버에 새 함수 func를 등록한다. name은 옵션인 함수의 이름이다. name을 제공하면 클라이언트에서 함수에 접근하는 데 이 이름을 사용하게 된다. 이 이름은 마침표(.) 같은 유효한 파이썬 식별자가 아닌 문자도 담을 수 있다. name을 제공하지 않으면 func의 실제 함수 이름을 대신 사용한다.

s.register_instance(instance [, allow_dotted_names])

register_function() 메서드로 등록하지 않은 메서드 이름을 해석하는 데 사용되는 객체를 등록한다. 인스턴스 instance가 _dispatch(self, methodname, params) 메서드를 정의하면 요청을 처리하기 위해 이 메서드가 호출된다. 여기서 methodname은 메서드 이름이고 params는 인수들을 담은 튜플이다. _dispatch()의 반환 값은 클라이언트에 전달된다. _dispatch() 메서드가 정의되어 있지 않으면 주어진 메서드 이름이 instance에 정의된 메서드 이름에 매칭되는지 확인한다. 그렇다면 해당 메서드가 바로 호출된다. allow_dotted_names 매개변수는 메서드 이름

을 검색할 때 계층적 검색을 수행할 것인지를 지정하는 플래그이다. 예를 들어, 메서드 ‘foo.bar.spam’에 대한 요청이 들어오면 instance.foo.bar.spam을 검색할 것인지를 결정한다. 기본으로 False이다. 검증된 클라이언트가 아니라면 True로 설정해서는 안 된다. 아니면 침입자가 임의의 파이썬 코드를 실행할 수 있는 보안 구멍이 생기게 된다. 한 번에 인스턴스 하나만 등록할 수 있다.

s.register_introspection_functions()

XML-RPC 서버에 XML-RPC 관찰 함수 system.listMethods(), system.methodHelp(), system.methodSignature()를 추가한다. system.methodHelp()는 메서드의 문서화 문자열을 반환한다(있는 경우). system.methodSignature() 함수는 간단히 해당 연산이 지원되지 않는다는 것을 알리는 메시지를 반환한다(파이썬이 동적인 타입 언어이기 때문에 타입 정보가 없다).

s.register_multicall_functions()

서버에 system.multicall() 함수를 추가하여 XML-RPC 다중 호출 함수 지원 기능을 추가한다.

DocXMLRPCServer 인스턴스는 다음 메서스들을 추가로 지원한다.

s.set_server_title(server_title)

서버 제목을 HTML 문서에 설정한다. 이 문자열은 HTML <title> 태그에 사용된다.

s.set_server_name(server_name)

서버 이름을 HTML 문서에 설정한다. 이 문자열은 페이지 최상단에 <h1> 태그에 나타난다.

s.set_server_documentation(server_documentation)

설명을 담은 문장을 생성되는 HTML 출력에 추가한다. 이 문자열은 서버 이름 바로 다음, 그리고 XML-RPC 함수들에 대한 설명 앞에 추가된다.

XML-RPC 서버가 독립 프로세스로 작동하는 것이 일반적이지만 CGI 스크립트 안에서 실행될 수도 있다. 이 목적을 위해 다음 클래스들이 정의된다.

CGIXMLRPCRequestHandler([allow_none [, encoding]])

SimpleXMLRPCServer와 같은 방식으로 작동하는 CGI 요청 처리기. 인수들은 SimpleXMLRPCServer에서와 의미가 같다.

DocCGIXMLRPCRequestHandler()

DocXMLRPCServer와 같은 방식으로 작동하는 CGI 요청 처리기. 이 책을 쓰고 있는 지금 이 처리기의 호출 인수가 CGIXMLRPCRequestHandler()와 다르다. 버그일 수 있으니 나중 버전의 온라인 문서를 확인해보기 바란다.

CGI 처리기 인스턴스 c는 함수와 인스턴스를 등록하기 위한 보통의 XML-RPC 서버와 동일한 메서드들을 가진다. 또한 추가로 다음 메서드를 가진다.

c.handle_request([request_text])

XML-RPC 요청을 처리한다. 기본으로 표준 입력에서 요청을 읽는다. request_text 가 있으면 HTTP POST 요청 형식으로 온 요청 데이터를 담아야 한다.

예

다음은 독립 서버를 작성하는 아주 간단한 예를 보여준다. add 함수 하나를 추가한다. 그리고 math 모듈의 전체 내용을 인스턴스로 추가하여 모든 함수를 공개한다.

```
try:
    from xmlrpclib import SimpleXMLRPCServer # 파일 3
except ImportError:
    from SimpleXMLRPCServer import SimpleXMLRPCServer # 파일 2
import math

def add(x,y):
    "Adds two numbers"
    return x+y

s = SimpleXMLRPCServer(("," ,8080))
s.register_function(add)
s.register_instance(math)
s.register_introspection_functions()
s.serve_forever()
```

다음은 같은 기능을 CGI 스크립트로 구현한 것이다.

```
try:
    from xmlrpclib import CGIXMLRPCRequestHandler # 파일 3
except ImportError:
    from SimpleXMLRPCServer import CGIXMLRPCRequestHandler # 파일 2
import math

def add(x,y):
    "Adds two numbers"
    return x+y
```

```
s = CGIXMLRPCRequestHandler()
s.register_function(add)
s.register_instance(math)
s.register_introspection_functions()
s.handle_request()
```

파이썬 프로그램에서 XML-RPC 함수에 접근하려면 xmlrpclib 모듈을 사용하면 된다. 다음은 이를 보여주는 짧은 대화식 세션이다.

```
>>> from xmlrpclib import ServerProxy
>>> s = ServerProxy("http://localhost:8080")
>>> s.add(3,5)
8
>>> s.system.listMethods()
['acos', 'add', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh',
'degrees', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp',
'log', 'log10', 'modf', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'system.listMethods', 'system.methodHelp', 'system.methodSignature',
'tan', 'tanh']
>>> s.tan(4.5)
4.6373320545511847
>>>
```

고급 서버 커스터마이즈

XML-RPC 서버 모듈을 사용하면 기본적인 분산 컴퓨팅을 쉽게 수행할 수 있다. 예를 들어, 모든 시스템이 적절한 XML-RPC 서버를 실행하고 있다면 네트워크를 통해 다른 서버를 고수준에서 제어하는 프로토콜로서 XML-RPC를 사용할 수 있다. pickle 모듈을 사용하면 더 흥미로운 객체들을 시스템 사이에 주고받을 수 있다.

XML-RPC와 관련해서 한 가지 걱정되는 점은 보안 문제이다. 기본으로 XML-RPC 서버는 네트워크에서 오픈 서버로서 작동하기 때문에 서버의 주소와 포트 번호만 알면 누구나 연결할 수 있다(방화벽에 막혀 있지만 않으면). 그리고 XML-RPC 서버는 요청에 담을 수 있는 데이터량에 제한을 두지 않는다. 이 때문에 공격자가 메모리를 다 차지할 만큼의 데이터를 담은 HTTP POST 요청을 보내서 서버를 다운시킬 수 있는 여지가 있다.

이러한 문제를 해결하고 싶다면 XML-RPC 서버 클래스나 요청 처리기를 커스터마이즈해야 한다. 모든 서버 클래스가 socketserver 모듈에 있는 TCPServer에서 상속받았다. 따라서 이 클래스를 커스터마이즈하듯이 서버들을 커스터마이즈할 수 있다(예를 들어 스레드, 포크, 클라이언트 주소 검증 등). SimpleXMLRPCRequestHandler나 DocXMLRPCRequestHandler에서 상속받아서 do_POST() 메서드를 확장하면 요청 처리기에 검증 래퍼를 추가할 수 있다. 다음 예

는 들어오는 요청의 크기를 제한하는 예이다.

```
try:
    from xmlrpclib import (SimpleXMLRPCServer,
                           SimpleXMLRPCRequestHandler)
except ImportError:
    from SimpleXMLRPCServer import (SimpleXMLRPCServer,
                                      SimpleXMLRPCRequestHandler)
class MaxSizeXMLRPCHandler(SimpleXMLRPCRequestHandler):
    MAXSIZE = 1024*1024 # 1MB
    def do_POST(self):
        size = int(self.headers.get('content-length',0))
        if size >= self.MAXSIZE:
            self.send_error(400,"Bad request")
        else:
            SimpleXMLRPCRequestHandler.do_POST(self)

s = SimpleXMLRPCServer(("8080"),MaxSizeXMLRPCHandler)
```

HTTP 기반 인증 기능을 추가할 때도 비슷하게 하면 된다.

23장

P y t h o n E s s e n t i a l R e f e r e n c e

웹 프로그래밍

파이썬은 웹 사이트를 만드는 데 널리 사용되며 여러 가지 역할을 수행한다. 첫 번째로 파이썬 스크립트는 웹 서버에서 제공할 정적 HTML 페이지를 생성할 때 유용하게 쓰인다. 예를 들어, 파이썬 스크립트로 무가공 콘텐츠를 가져와서 웹 사이트에서 흔히 볼 수 있는 추가 기능(탐색바, 사이드바, 광고, 스타일 시트 등)을 붙일 수 있다. 이 부분은 대부분 파일 처리와 텍스트 처리와 관련 있으며 여기에 대해서는 이 책의 다른 곳에서 다룬다.

두 번째로 동적 콘텐츠를 생성하는 데 파이썬 스크립트를 사용한다. 예를 들어, 웹 서버를 아파치(Apache) 같은 표준 웹 서버에서 돌리면서 특정한 종류의 요청을 동적으로 처리하는 데 파이썬 스크립트를 사용할 수 있다. 여기서는 주로 폼(form) 처리를 수행하게 된다. 예를 들어, 다음과 같은 폼을 담은 HTML 페이지를 생각해 볼 수 있다.

```
<FORM ACTION='/cgi-bin/subscribe.py' METHOD='GET'>
Your name : <INPUT type='Text' name='name' size='30'>
Your email address: <INPUT type='Text' name='email' size='30'>
<INPUT type='Submit' name='submit-button' value='Subscribe'>
</FORM>
```

이 폼에서 ACTION 속성으로 폼을 제출할 때 서버에서 실행할 파이썬 스크립트 이름으로 ‘subscribe.py’를 지정하였다.

동적 콘텐츠 생성과 관련하여 AJAX(Aynchronous Javascript and XML)를 사용하는 일이 흔히 있다. AJAX를 사용할 때 웹 페이지에서 특정 HTML 원소(element)에 자바스크립트 이벤트 처리기가 연결된다. 예를 들어, 특정 문서 원소에 마우스를

갖다 대면 자바스크립트 함수가 실행되어 웹 서버에 HTTP 요청을 보내고 이 요청은 서버에서 처리한다(파이썬 스크립트로 처리할 수도 있다). 관련 응답이 오면 다른 자바스크립트 함수가 실행되고 응답 데이터를 처리하고 결과를 출력한다. 결과를 반환하는 데는 여러 가지 방법을 사용할 수 있다. 예를 들어, 데이터를 서버에서 일반 텍스트, XML, JSON 또는 기타 형식으로 반환할 수 있다. 다음은 지정된 원소 위에 마우스를 올리면 팝업 창을 띄우는 올림 팝업(hover popup)을 구현한 HTML 문서이다.

```

<html>
  <head>
    <title>ACME Officials Quiet After Corruption Probe</title>
    <style type="text/css">
      .popup { border-bottom:1px dashed green; }
      .popup:hover { background-color: #c0c0ff; }
    </style>
  </head>
  <body>
    <span id="popupbox" style="visibility:hidden; position:absolute;
      background-color:#ffffff;">
      <span id="popupcontent"></span>
    </span>
    <script>
      /* 팝업 박스 원소에 대한 참조를 가져온다.*/
      var popup = document.getElementById("popupbox");
      var popupcontent = document.getElementById("popupcontent");

      /* 서버에서 팝업 데이터를 가져와서 보여준다.*/
      function ShowPopup(hoveritem,name) {
        var request = new XMLHttpRequest();
        request.open("GET","cgi-bin/popupdata.py?name="+name, true);
        request.onreadystatechange = function() {
          var done = 4, ok = 200;
          if (request.readyState == done && request.status == ok) {
            if (request.responseText) {
              popupcontent.innerHTML = request.responseText;
              popup.style.left = hoveritem.offsetLeft+10;
              popup.style.top = hoveritem.offsetTop+20;
              popup.style.visibility = "Visible";
            }
          }
        };
        request.send();
      }

      /* 팝업 박스를 숨긴다.*/
      function HidePopup() {
        popup.style.visibility = "Hidden";
      }
    </script>
  </body>
</html>
```

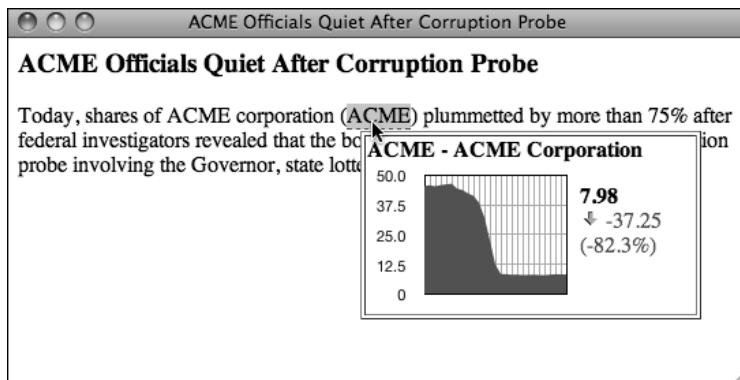
```

<h3>ACME Officials Quiet After Corruption Probe</h3>
<p>
Today, shares of ACME corporation
(<span class="popup" onMouseOver="ShowPopup(this,'ACME');" 
onMouseOut="HidePopup();">ACME</span>)
plummetted by more than 75% after federal investigators
revealed that the board of directors is the target of a
corruption probe involving the Governor, state lottery
officials, and the archbishop.
</p>
</body>
</html>

```

이 예에서 자바스크립트 함수 ShowPopup()은 서버에 있는 파이썬 스크립트 popupdata.py로 요청을 보낸다. 결과로 HTML 조각이 넘어오고 팝업 윈도에 출력된다. 그림 23.1은 팝업 윈도가 브라우저에서 어떤 모습으로 나타나는지를 보여준다.

그림 23.1 배경 텍스트가 보통의 HTML 문서이고 popupdata.py 스크립트로 팝업 윈도를 동적으로 생성하였을 때 브라우저에서 보이는 모습



마지막으로 파이썬으로 작성된 프레임워크의 컨텍스트에서 전체 웹사이트를 제어할 수도 있다. 재미있는 말로 “파이썬 언어의 키워드 수보다 웹 프로그래밍 프레임워크 수가 더 많다”라는 말이 있다. 웹 프레임워크는 이 책의 범위를 벗어나는 주제이고 더 많은 정보를 얻기 원하면 <http://wiki.python.org/moin/WebFrameworks>를 먼저 살펴보기 바란다.

이 장의 나머지 부분에서는 파이썬이 웹 서버 및 프레임워크와 상호작용하는 데 사용하는 저수준 인터페이스와 관련된 내장 모듈을 설명한다. 써드 파티 웹 서버에서 파이썬에 접근하는 데 사용하는 CGI 스크립팅, 그리고 다양한 파이썬 웹 프레임워크에 통합될 컴포넌트를 작성하는 데 사용하는 미들웨어 계층인 WSGI에 대해서 다룬다.

cgi

cgi 모듈은 폼으로 들어온 사용자의 입력을 처리하거나 보통 웹 서버가 특정 종류의 동적인 콘텐츠를 생성하기 위해서 실행하는 프로그램인 CGI 스크립트를 구현하는 데 사용한다.

CGI 스크립트와 관련된 요청이 오면 웹 서버는 CGI 프로그램을 하위 프로세서로 실행한다. CGI 프로그램에서는 두 개의 소스(sys.stdin과 서버에서 설정한 환경 변수)에서 입력을 받는다. 다음 목록은 웹 서버에서 흔히 설정하는 환경 변수들을 보여준다.

변수	설명
AUTH_TYPE	인증 방법
CONTENT_LENGTH	sys.stdin으로 전달된 데이터 길이
CONTENT_TYPE	질의 데이터 타입
DOCUMENT_ROOT	문서 루트 디렉터리
GATEWAY_INTERFACE	CGI 리비전 문자열
HTTP_ACCEPT	클라이언트가 수락한 MIME 타입
HTTP_COOKIE	네스케이프 지속 쿠키 값
HTTP_FROM	클라이언트 이메일 주소(보통 비활성화됨)
HTTP_REFERER	참조하는 URL
HTTP_USER_AGENT	클라이언트 브라우저
PATH_INFO	전달된 추가 경로 정보
PATH_TRANSLATED	PATH_INFO의 변환 버전
QUERY_STRING	질의 문자열
REMOTE_ADDR	클라이언트 원격 IP 주소
REMOTE_HOST	클라이언트 호스트 이름
REMOTE_IDENT	요청을 보낸 사용자
REMOTE_USER	인증된 사용자 이름
REQUEST_METHOD	요청 메서드('GET' 또는 'POST')
SCRIPT_NAME	프로그램 이름
SERVER_NAME	서버 호스트 이름
SERVER_PORT	서버 포트 번호

SERVER_PROTOCOL	서버 프로토콜
SERVER_SOFTWARE	서버 소프트웨어 이름과 버전

CGI 프로그램에서는 출력을 표준 출력 sys.stdout로 쓴다. CGI 프로그래밍과 관련된 자세한 정보는 쉬서 건다바람(Shishir Gundavaram)이 쓴 『Perl CGI 프로그래밍하기 제2판(CGI Programming with Perl, 2nd Edition)』(O'Reilly & Associates, 2000)를 참고하기 바란다. 여기에서는 두 가지만 알면 된다. 첫째, HTML 폼의 내용은 질의 문자열(query string)이라고 부르는 텍스트 순서열로 CGI 프로그램에 전달된다. 파이썬에서 FieldStorage 클래스를 사용하여 질의 문자열에 접근할 수 있다. 다음 예를 보자.

```
import cgi
form = cgi.FieldStorage()
name = form.getvalue('name')      # 폼에서 'name' 필드 값을 가져옴
email = form.getvalue('email')    # 폼에서 'email' 필드 값을 가져옴
```

둘째, CGI 프로그램의 출력은 HTTP 헤더와 무가공 데이터(일반적으로 HTML) 두 부분으로 구성된다. 항상 빈 줄로 이 두 부분을 나눈다. 다음은 간단한 HTTP 헤더를 보여준다.

```
print 'Content-type: text/html\r'  # HTML 출력
print '\r'                      # 빈 줄(필수!)
```

남은 것은 무가공 데이터 부분이다. 다음을 보자.

```
print '<TITLE>My CGI Script</TITLE>'
print '<H1>Hello World!</H1>'
print 'You are %s (%s)' % (name, email)
```

표준적인 관행으로서 HTTP 헤더는 원도 줄 마침 규약인 '\r\n'으로 끝낸다. 앞 예에서 '\r'를 보았던 이유기도 하다. 에러를 표시하려면 특수 헤더인 'Status:'를 출력에 포함시키면 된다. 다음을 보자.

```
print 'Status: 401 Forbidden\r'    # HTTP 에러 코드
print 'Content-type: text/plain\r'
print '\r'                      # 빈 줄(필수)
print 'You're not worthy of accessing this page!'
```

클라이언트를 다른 페이지로 돌리려면 다음처럼 하면 된다.

```
print 'Status: 302 Moved\r'
print 'Location: http://www.foo.com/orderconfirm.html\r'
print '\r'
```

cgi 모듈에서 대부분의 작업은 FieldStorage 클래스의 인스턴스를 생성하는 것으

로 시작한다.

```
FieldStorage([input [, headers [, outerboundary [, environ [, keep_
blank_values [, strict_parsing]]]]])
```

환경 변수 또는 표준 입력으로 전달된 질의 문자열을 읽어서 파싱하여 폼의 내용을 읽어온다. input은 파일과 유사한 객체를 나타내고 폼 데이터는 input에서 POST 요청 형식에 따라 읽는다. 기본 값으로 sys.stdin이 사용된다. headers와 outerboundary는 내부적으로 사용하는 것으로서 값을 지정하지 않아야 한다. environ은 CGI 환경 변수를 읽어올 사전이다. keep_blank_values은 빈 값을 유지할 것인지를 지정하는 불리언 플래그이다. 기본 값으로는 False이다. strict_parsing은 파싱 문제가 있을 경우 예외를 발생시킬 것인지를 지정하는 불리언 플래그다. 기본 값은 False이다.

FieldStorage 인스턴스 form은 사전과 유사하게 작동한다. 예를 들어, f = form[key]는 주어진 매개변수 key의 값을 추출한다. 이렇게 추출한 인스턴스 f는 또 다른 FieldStorage의 인스턴스이거나 MiniFieldStorage의 인스턴스이다. 다음 속성들이 f에서 정의된다.

속성	설명
f.name	주어진 경우 필드 이름
f.filename	업로드에 사용할 클라이언트 쪽 파일 이름
f.value	문자열로 표현된 값
f.file	데이터를 읽을 파일과 유사한 객체
f.type	콘텐츠 타입
f.type_options	HTTP 요청의 content-type 줄에서 지정된 옵션들을 담은 사전
f.disposition	'content-disposition' 필드. 지정하지 않았으면 None.
f.disposition_options	배열(disposition) 옵션들을 담은 사전
f.headers	HTTP 헤더의 모든 내용을 담은 사전과 유사한 객체

form에 기록된 값은 다음 메서드로 얻을 수 있다.

```
form.getvalue(fieldname [, default])
```

fieldname이라는 이름을 갖는 필드의 값을 반환한다. 어떤 필드가 두 번 이상 정의되면 이 함수는 정의된 모든 값의 리스트를 반환한다. default가 주어지면 필드가 존재하지 않을 경우 반환할 값으로 사용된다. 한 가지 주의할 점은 요청에 같은 폼 이름이 두 번 나타날 경우 두 값을 담은 리스트가 반환된다는 점이다. 프로그래밍을 간단하게 만들려면 첫 번째 값을 반환하는 form.getFirst()를 사용하도록 한다.

form.getFirst(fieldname [, default])

fieldname이라는 이름을 갖는 필드의 첫 번째 값을 반환한다. default가 주어지면 필드가 존재하지 않을 경우 사용된다.

form.getList(fieldname)

fieldname라는 이름을 갖는 모든 값의 리스트를 반환한다. 값이 하나만 정의되어 있어도 항상 리스트를 반환하고 값이 존재하지 않을 경우 빈 리스트를 반환한다.

cgi 모듈에는 어떤 속성의 이름과 값만 담는 MiniFieldStorage라는 클래스도 있다. 이 클래스는 질의 문자열로 전네진 폼의 개별 필드를 나타내는 데 사용하고 FieldStorage는 여러 필드와 여러 부분(multipart)으로 구성된 데이터를 담는 데 사용한다.

FieldStorage의 인스턴스는 파이썬 사전처럼 접근한다. 키는 폼에서 필드 이름을 나타낸다. 이렇게 접근할 때 반환되는 객체는 그 자체가 여러 부분 데이터(content-type이 'multipart/form-data') 또는 파일 업로드를 나타내는 FieldStorage의 인스턴스거나 간단한 필드(content-type이 'application/x-www-form-urlencoded')를 나타내는 MiniFieldStorage의 인스턴스이거나 폼에 같은 이름을 갖는 여러 필드가 있을 경우 인스턴스 리스트일 수 있다. 다음 예를 보자.

```
form = cgi.FieldStorage()
if "name" not in form:
    error("Name is missing")
    return
name = form['name'].value      # 폼에서 'name' 필드의 값을 가져옴.
email = form['email'].value    # 폼에서 'email' 필드의 값을 가져옴.
```

필드가 업로드된 파일을 나타내는 경우에는 속성 값에 접근하면 전체 파일을 바이트 문자열로 메모리에 읽어들이게 된다. 이는 서버에서 메모리를 많이 소비할 수 있기 때문에 file 속성을 통해 좀 더 작은 단위로 읽어 오는 것이 더 낫다. 예를 들어, 다음 예는 업로드된 데이터를 한 줄씩 읽는 것을 보여준다.

```
fileitem = form['userfile']
if fileitem.file:
    # 업로드된 파일이다. 줄 수를 셉.
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

다음은 CGI 스크립트에서 종종 사용되는 유ти리티 함수들이다.

escape(s [, quote])

문자열 s에 있는 ‘&’, ‘<’, ‘>’ 문자를 ‘&’, ‘<’, ‘>’ 같이 HTML에서 특별한 의미를 갖지 않는 안전한 순서열로 바꾼다. 옵션인 플래그 quote가 참이면 큰따옴표(“)도 ‘"’으로 바꾼다.

parse_header(string)

‘content-type’ 등과 같은 HTTP 헤더 필드 다음에 나오는 데이터를 파싱한다. 데이터는 주 값과 부차적인 매개변수들을 담은 사전으로 나누어지고 튜플로서 반환된다. 예를 들어,

```
parse_header('text/html; a=hello; b="world")
```

는 다음 결과를 반환한다.

```
('text/html', {'a': 'hello', 'b': 'world'})
```

parse_multipart(fp,pdict)

파일 업로드에 주로 사용되는 타입 ‘multipart/form-data’ 입력을 파싱한다. fp는 입력 파일이고 pdict는 content-type 헤더의 매개변수들을 담은 사전이다. 필드 이름을 값 리스트로 맵핑하는 사전을 반환한다. 이 함수는 중첩된 여러 부분 데이터에 대해서는 제대로 작동하지 않는다. 그런 경우에는 FieldStorage 클래스를 대신 사용해야 한다.

print_directory()

현재 작업 디렉터리의 이름을 HTML 형식으로 출력한다. 출력 결과는 브라우저에 전달되고 이 정보는 디버깅에 유용하게 사용할 수 있다.

print_environ()

모든 환경 변수 목록을 HTML 형식으로 출력하며 이 정보는 디버깅할 때 유용하게 사용할 수 있다.

print_environ_usage()

유용하다고 생각되는 몇몇 환경 변수를 선택적으로 출력하며 이 정보는 디버깅 할 때 유용하게 사용할 수 있다.

print_form(form)

폼으로 제출된 데이터를 HTML 형식으로 만든다. form은 FieldStorage의 인스턴

스이어야 한다. 디버깅에 사용한다.

test()

최소한의 HTTP 헤더를 쓰고 스크립트에 제공된 모든 정보를 HTML 형식으로 출력한다. 주로 CGI 환경이 적절히 설정되었는지를 확인하기 위해 사용된다.

CGI 프로그래밍 조언

지금 같은 웹 프레임워크 시대에 CGI 스크립트는 시대에 뒤떨어진 것처럼 보인다. 하지만, 여러분이 CGI를 계속 사용해야 하는 상황에 처해 있다면 다음 팁들이 여러분의 인생을 좀 더 편하게 만들어줄 것이다.

첫째, 하드코딩된 HTML 출력을 생성하기 위해서 엄청난 양의 print문을 사용하는 CGI 스크립트를 작성하지 않도록 한다. 그렇게 하면 파이썬과 HTML 코드가 서로 엉켜서 프로그램을 해석하기가 불가능하고 유지 관리할 수도 없게 된다. 이런 경우에는 템플릿을 사용하는 것이 더 낫다. 가장 간단하게는 string.Template 객체를 사용할 수 있다. 다음은 이 개념을 대략적으로 보여주는 예이다.

```
import cgi
from string import Template

def error(message):
    temp = Template(open("errmsg.html").read())
    print 'Content-type: text/html\r'
    print '\r'
    print temp.substitute({'message' : message})

form = cgi.FieldStorage()
name = form.getFirst('name')
email = form.getFirst('email')
if not name:
    error("name not specified")
    raise SystemExit
elif not email:
    error("email not specified")
    raise SystemExit

# 다양한 처리를 수행한다.
confirmation = subscribe(name, email)

# 결과 페이지 출력
values = {
    'name' : name,
    'email' : email,
    'confirmation' : confirmation,
    # 여기에 다른 값을 추가...
}
```

```

    }
temp = Template(open("success.html").read())
print temp.substitute(values)

```

이 예에서 ‘error.html’과 ‘success.html’ 파일은 그 안에 출력한 모든 내용을 담고 있는 HTML 페이지이지만 CGI 스크립트에서 동적으로 생성하는 값에 대응하는 \$variable 치환부를 담고 있다. 예를 들어, ‘success.html’ 파일은 다음과 같은 페이지일 수 있다.

```

<HTML>
<HEAD>
  <TITLE>Success</TITLE>
</HEAD>
<BODY>
Welcome $name. You have successfully subscribed to our newsletter.
Your confirmation code is $confirmation.
</BODY>
</HTML>

```

Note

- CGI 프로그램을 설치하는 과정은 사용하는 웹 서버의 종류에 따라 다르다. 일반적으로는 프로그램을 특별한 cgi-bin 디렉터리에 둔다. 서버에 추가로 설정을 해주어야 할 수도 있다. 자세한 것은 서버 관련 자료를 참고하거나 서버 관리자에게 물어보도록 한다.
 - 유닉스에서는 파이썬 CGI 프로그램에 적절히 실행 권한을 주고 다음과 같이 프로그램의 첫 줄에 한 줄을 추가해야 하는 경우가 있다.
- ```

#!/usr/bin/env python
import cgi
...

```
- 디버깅을 단순화하기 위해서 cgitb 모듈을 임포트하도록 한다. 예를 들어, import cgitb; cgitb.enable()처럼 한다. 이렇게 하면 예외 처리 방식이 변경되어 에러가 웹 브라우저에 출력된다.
  - os.system() 또는 os.popen() 같은 함수로 외부 프로그램을 호출할 때는 클라이언트로부터 받은 임의의 문자열을 셀로 전달하지 않도록 주의한다. 해커가 서버에서 임의의 셀 명령어를 실행할 수 있는 널리 알려진 보안 구멍을 만들 수 있다(그 이유는 이런 함수에 전달된 명령은 직접 실행되는 것이 아니라 먼저 유닉스 셀로 실행하기 때문이다). 특히, 문자열이 알파벳과 숫자, 대시, 밑줄, 마침표만 담고 있는지를 확실히 검사하기 전에는 URL의 일부나 폼 데이터를 절대로 셀로 넘기지 않도록 하라.
  - 유닉스에서 CGI 프로그램에 setuid 모드를 부여하지 않도록 한다. 이렇게 하면 보안에 문제가 생길 수 있고 모든 머신에서 지원되는 것도 아니다.
  - ‘from cgi import \*’로 이 모듈을 사용하지 않도록 한다. cgi 모듈은 여러분의 네임스페이스에 들어가기를 원하지 않을 만한 많은 이름과 기호를 정의한다.

앞의 스크립트에서 `temp.substitute()`는 이 파일에 있는 변수들을 채운다. 이 방법의 확실한 장점은 출력 형태를 변경하고자 할 때 CGI 스크립트를 수정하는 것이 아니라 단순히 템플릿 파일만 수정하면 된다는 점이다. 파이썬을 위한 써드파티 템플릿 엔진들이 많이 있다(어쩌면 웹 프레임워크 수보다 훨씬 많을 수도 있다). 이들은 템플릿 개념을 바탕으로 하여 기능을 상당히 확장한 것들이다. 자세한 것은 <http://wiki.python.org/moin/Templating>를 참고하도록 한다.

둘째, CGI 스크립트에서 데이터를 저장하려면 데이터베이스를 사용해본다. 비록 데이터를 파일에 직접 쓰는 것이 쉬울지라도 웹 서버가 동시성 있게 작동하기 때문에 자원에 락을 걸고 동기화하기 위한 단계들을 적절히 수행하지 않으면 파일이 깨질 수 있다. 데이터베이스 서버나 관련 파이썬 인터페이스를 사용할 때는 보통 이러한 문제가 발생하지 않는다. 따라서, 데이터를 저장해야 한다면 sqlite3을 사용하거나 MySQL 같은 써드파티 모듈을 사용해보도록 한다.

마지막으로, 많은 수의 CGI 스크립트와 쿠키, 인증, 인코딩 같은 HTTP의 저수준 기능을 조작하는 코드를 작성하고 있는 경우라면 웹 프레임워크를 사용하는 것을 고려해보도록 한다. 프레임워크라는 것을 사용한다는 것 자체가 앞에서 언급한 상세한 내용에 대해서 신경 쓰지 않겠다는 것이다. 적어도 생각보다 많지는 않다. 그러니까 시간과 노력을 낭비하지 않도록 한다.

## cgitb

이 모듈은 잡하지 않은 예외가 발생할 때마다 상세한 보고서를 출력하기 위해서 사용할 수 있는 예외 처리기를 제공한다. 보고되는 내용에는 소스 코드, 매개변수들의 값, 지역 변수 등이 포함된다. 원래 이 모듈은 CGI 스크립트의 디버깅을 돋기 위해 개발되었지만 어느 응용 프로그램에서든지 사용할 수 있다.

```
enable([display [, logdir [, context [, format]]]])
```

특별 예외 처리를 활성화한다. `display`는 에러가 발생했을 때 정보를 출력할 것인지를 결정하는 플래그이다. 기본 값은 1이다. `logdir`는 에러 보고를 표준 출력으로 출력하지 않고 파일에 출력할 때 파일을 쓸 디렉터리를 지정한다. `logdir`이 주어지면 각 에러 보고서는 `tempfile.mkstemp()` 함수가 생성하는 고유한 파일에 쓰여진다. `context`는 예외가 발생한 곳 주위의 몇 줄에 해당하는 소스 코드를 출력할 것인지를 지정하는 정수이다. `format`은 출력 포맷을 지정하는 문자열이다. ‘html’은

HTML을 나타낸다(기본 값). 기타 다른 값은 보통 텍스트 형식을 나타낸다.

#### **handle([info])**

enable( ) 함수의 기본 설정을 사용하여 예외를 처리한다. info는 튜플 (exctype, excvalue, tb)이며 exctype은 예외 타입을, excvalue는 예외 값을, tb는 traceback 객체를 나타낸다. 이 튜플은 보통 sys.exc\_info( )를 사용하여 얻는다. info가 생략되면 현재 예외가 사용된다.

#### **Note**

CGI 스크립트에서 특별 예외 처리 기능을 활성화하려면 스크립트 시작 부분에 import cgitb; enable( )를 포함시키도록 한다.

## wsgiref

WSGI (Python Web Server Gateway Interface)는 여러 웹 서버와 프레임워크 사이에 이식성을 항상시키기 위해 고안된 웹 서버와 웹 응용 프로그램 사이의 표준 인터페이스이다. 공식 표준 문서는 PEP 333(<http://www.python.org/dev/peps/pep-0333>)에 있다. 표준과 표준의 활용법에 대한 자세한 정보는 <http://www.wsgi.org>에서 찾을 수 있다. wsgiref 패키지는 테스트, 검증, 간단한 배포를 위해 사용할 수 있는 참조 구현을 제공한다.

## WSGI 명세서

WSGI에서 웹 응용 프로그램은 두 개의 인자를 받는 함수나 호출 가능한 객체 webapp(environ, start\_response)로 구현한다. environ은 환경 설정 정보를 담은 사전으로 적어도 다음에 나오는 CGI 스크립트에서 사용하는 것과 동일한 의미를 갖는 값들을 담고 있어야 한다.

| environ 변수      | 설명                  |
|-----------------|---------------------|
| CONTENT_LENGTH  | 전달된 데이터의 길이         |
| CONTENT_TYPE    | 질의 데이터 종류           |
| HTTP_ACCEPT     | 클라이언트가 받아들인 MIME 타입 |
| HTTP_COOKIE     | 네스케이프 지속 쿠키 값       |
| HTTP_REFERER    | 참조하는 URL            |
| HTTP_USER_AGENT | 클라이언트 브라우저          |

|                 |                         |
|-----------------|-------------------------|
| PATH_INFO       | 전달된 추가 경로 정보            |
| QUERY_STRING    | 질의 문자열                  |
| REQUEST_METHOD  | 요청 메서드('GET' 또는 'POST') |
| SCRIPT_NAME     | 프로그램 이름                 |
| SERVER_NAME     | 서버 호스트 이름               |
| SERVER_PORT     | 서버 포트 번호                |
| SERVER_PROTOCOL | 서버 프로토콜                 |

추가적으로 environ 사전은 다음에 나오는 WSGI만의 값들을 담고 있어야 한다.

| environ 변수        | 설명                                                           |
|-------------------|--------------------------------------------------------------|
| wsgi.version      | WSGI 버전을 나타내는 튜플(예를 들어, WSGI 1.0이면 (1,0))                    |
| wsgi.url_scheme   | URL의 형식 부분을 나타내는 문자열. 예를 들어 'http' 또는 'https'.               |
| wsgi.input        | 입력 스트림을 나타내는 파일과 유사한 객체. 양식 데이터나 업로드 데이터 같은 추가 데이터를 여기서 읽는다. |
| wsgi.errors       | 에러 출력을 쓰기 위한 텍스트 모드로 열린 파일과 유사한 객체                           |
| wsgi.multithread  | 같은 프로세스의 다른 스레드가 응용 프로그램을 동시에 실행하면 True인 불리언 플래그             |
| wsgi.multiprocess | 다른 프로세스가 응용 프로그램을 동시에 실행하면 True인 불리언 플래그                     |
| wsgi.run_once     | 프로세스를 실행하는 동안 응용 프로그램을 한 번만 실행하면 True인 불리언 플래그               |

start\_response 매개변수는 start\_response(status, headers) 형식을 갖는 호출 가능한 객체로서 응용 프로그램에서 응답하기 시작할 때 사용한다. status는 '200 OK' 또는 '404 Not Found' 같은 문자열이다. headers는 튜플 리스트로, 각각 (name, value) 형태이고 응답에 포함할 HTTP 헤더에 대응한다. 예를 들어 ('Content-type', 'text/html').

웹 응용 프로그램이 반환하는 응답 데이터 또는 몸체는 바이트 문자열들 또는 한 바이트로 인코딩할 수 있는 문자(ISO-8859-1 또는 Latin-1 문자 집합과 호환 가능한)들을 담은 텍스트 문자열들의 순서열을 생성하는 반복 가능한 객체이다. 예를 들어, 바이트 문자열들의 리스트나 바이트 문자열들을 생성하는 생성기 함수가 가능하다. 응용 프로그램에서 UTF-8 같은 문자 인코딩을 수행하려면 직접 해야 한다.

다음은 앞서 cgi 모듈을 설명할 때 나온 예와 비슷하게 폼 필드를 읽고 출력을 생성하는 간단한 WSGI 응용 프로그램을 보여준다.

```
import cgi
def subscribe_app(environ, start_response):
 fields = cgi.FieldStorage(environ['wsgi.input'], environ=environ)
```

```

name = fields.getvalue("name")
email = fields.getvalue("email")

다양한 처리를 수행

status = "200 OK"
headers = [('Content-type','text/plain')]
start_response(status, headers)

response = [
 'Hi %s. Thank you for subscribing.' % name,
 'You should expect a response soon.'
]
return (line.encode('utf-8') for line in response)

```

이 예에서 중요한 점이 몇 가지 있다. 첫 번째로 WSGI 응용 프로그램 컴포넌트는 특정 프레임워크, 웹 서버, 라이브러리 모듈 집합에 종속적이지 않다. 앞 예에서는 라이브러리 모듈 cgi 하나만 사용했고 편리하게 질의 변수들을 파싱하는 데 사용했다. 이 예는 start\_response() 함수로 응답을 시작하고 헤더를 만드는 방법을 보여 준다. 응답 자체는 문자열들의 리스트로 구축된다. 이 응용 프로그램에서 마지막 문장은 모든 문자열을 바이트 문자열로 바꾸는 생성기 표현식이다. 모든 WSGI 응용 프로그램에서는 인코딩이 되지 않은 유니코드 데이터가 아니라, 인코딩된 바이트 문자열을 반환해야 하기 때문에 파이썬 3에서는 이 부분이 중요하다.

WSGI 응용 프로그램을 배치(deploy)하면서 사용하고 있는 웹 프로그래밍 프레임워크에 등록해야 한다. 어떻게 하는지는 관련 메뉴얼을 참고하기 바란다.

## wsgiref 패키지

wsgiref 패키지는 응용 프로그램을 독립 서버에서 테스트하거나 보통의 GCI 스크립트로서 테스트할 수 있도록 WSGI 표준의 참조 구현을 제공한다.

### wsgiref.simple\_server

wsgiref.simple\_server 모듈은 간단히 하나의 WSGI 응용 프로그램을 실행할 수 있는 간단한 독립 HTTP 서버를 구현한다. 우리가 관심 있는 함수는 다음 두 가지이다.

#### **make\_server(host, port, app)**

주어진 호스트 이름 host와 포트 번호 port에 대해 연결을 받아들이는 HTTP 서버를 생성한다. app는 WSGI 응용 프로그램을 구현한 함수나 호출 가능한 객체이다. 서버를 구동하려면 s가 이 함수가 반환한 인스턴스라고 할 때 s.serve\_forever()를

실행하면 된다.

### **demo\_app(environ, start\_response)**

“Hello World” 메시지를 출력하는 페이지를 반환하는 완성된 WSGI 응용 프로그램. 서버가 제대로 돌아가는지를 확인하기 위해서 make\_server()에 전달할 app 인수로 사용할 수 있다.

다음은 간단히 WSGI 서버를 작동시키는 예를 보여준다.

```
def my_app(environ, start_response):
 # 응용 프로그램
 start_response("200 OK", [('Content-type', 'text/plain')])
 return ['Hello World']

if __name__ == '__main__':
 from wsgiref.simple_server import make_server
 serv = make_server("", 8080, my_app)
 serv.serve_forever()
```

### wsgiref.handlers

wsgiref.handlers 모듈은 응용 프로그램을 다른 웹 서버(아파치 서버에서 CGI 스크립트처럼)에서 실행할 수 있게 WSGI 실행 환경을 설정하는 처리기 객체를 담고 있다.

이 모듈에는 다음에 나오는 몇 가지 객체가 들어 있다.

### **CGIHandler()**

표준 CGI 환경에서 작동하는 WSGI 처리기 객체를 생성한다. 이 처리기는 cgi 라이브러리 모듈 부분에서 설명한 표준 환경 변수와 I/O 스트림에서 정보를 모운다.

### **BaseCGIHandler(stdin, stdout, stderr, environ [, multithread [, multiprocess]])**

CGI 환경에서 작동하는 WSGI 처리기를 생성하지만 표준 I/O 스트림과 환경 변수를 다르게 설정할 수 있다. stdin, stdout, stderr는 표준 I/O 스트림에 대한 파일과 유사한 객체를 나타낸다. environ은 환경 변수들을 담은 사전으로서 표준 CGI 환경 변수들을 미리 담고 있어야 한다. multithread와 multiprocess는 블리언 플래그로서 wsgi.multithread와 wsgi.multiprocess 환경 변수를 설정하기 위해 사용한다. multithread는 기본 값이 True, multiprocess는 False이다.

```
SimpleHandler(stdin, stdout, stderr, environ [, multithread [, multiprocess]])
```

BaseCGIHandler와 유사한 WSGI 처리기를 생성하지만 내부 응용 프로그램에서 stdin, stdout, stderr와 environ에 직접 접근할 수 있다. 특정 기능을 제대로 처리하기 위해 추가 로직을 포함하는 BaseCGIHandler와 약간 다르다(예를 들어, BaseCGIHandler에서는 응답 코드가 헤더 Status로 변환된다).

모든 처리기는 처리기 안에서 WSGI 응용 프로그램을 실행하는 데 사용하는 run(app) 메서드를 가지고 있다. 다음은 전통적인 CGI 스크립트로 작동하는 WSGI 응용 프로그램의 예이다.

```
#!/usr/bin/env python
def my_app(environ, start_response):
 # 응용 프로그램
 start_response("200 OK",[('Content-type','text/plain')])
 return ['Hello World']

from wsgiref.handlers import CGIHandler
hand = CGIHandler()
hand.run(my_app)
```

wsgiref.validate

wsgiref.validate 모듈은 WSGI 응용 프로그램과 서버가 모두 표준에 따라 작동하는 것을 검증하기 위해서 WSGI 응용 프로그램을 감싸는 함수를 담고 있다.

#### **validator(app)**

WSGI 응용 프로그램 app을 감싸는 새로운 WSGI 응용 프로그램을 생성한다. 새로운 응용 프로그램은 응용 프로그램과 서버가 WSGI 표준에 따라서 작동하는 것을 보장하기 위해서 광범위하게 여러 검사하는 부분이 추가된 것을 제외하고는 app과 같은 방식으로 작동한다. 위반 사항이 있으면 AssertionError 예외가 발생한다. 다음은 validator를 사용하는 예를 보여준다.

```
def my_app(environ, start_response):
 # 응용 프로그램
 start_response("200 OK",[('Content-type','text/plain')])
 return ['Hello World']

if __name__ == '__main__':
 from wsgiref.simple_server import make_server
 from wsgiref.validate import validator
 serv = make_server("",8080, validator(my_app))
 serv.serve_forever()
```

**Note**

이 절에서 다룬 내용은 주로 응용 프로그램 객체를 생성하기를 원하는 WSGI 사용자를 위한 것이다. 여러분이 파이썬 웹 프레임워크를 구현하고 있는 중이라면 WSGI를 지원하는 프레임워크를 만들기 위해 필요한 것을 자세하게 설명한 PEP 333을 참고하도록 한다. 써드파티 웹 프레임워크를 사용하는 중이라면 WSGI 객체의 지원을 상세하게 설명하는 프레임워크 문서를 참고하도록 한다. WSGI가 표준 라이브러리에서 참조 구현이 제공되는 공식 명세서이기 때문에 이 표준을 어떤 식으로든 지원하는 프레임워크가 점점 늘어날 것이다.

## webbrowser

webbrowser 모듈은 플랫폼에 독립적인 방법으로 웹 브라우저에서 문서를 열기 위한 유ти리티 함수를 제공한다. 이 모듈은 주로 개발 및 테스트 환경에서 사용한다. 예를 들어, HTML 결과를 생성하는 스크립트를 작성했을 때 결과를 보기 위해서 이 모듈에 있는 함수를 사용하여 브라우저를 바로 열 수 있다.

### `open(url [, new [, autoraise]])`

시스템의 기본 브라우저에서 url을 보여준다. new가 0이면 가능한 경우 현재 실행되고 있는 브라우저의 같은 창 안에서 url이 열린다. new가 1이면 새로운 브라우저 창에서 열린다. new가 2이면 브라우저에서 새로운 탭에 열린다. autoraise가 True이면 브라우저 창이 화면 앞쪽으로 나온다.

### `open_new(url)`

기본 브라우저의 새로운 창에서 url을 보여준다. `open(url, 1)`과 동일하다.

### `open_new_tab(url)`

기본 브라우저의 새로운 탭에서 url을 보여준다. `open(url, 2)`와 동일하다.

### `get([name])`

브라우저를 조작하기 위한 제어 객체를 반환한다. name은 브라우저 타입 이름이고, 보통 ‘netscape’, ‘mozilla’, ‘kfm’, ‘grail’, ‘windows-default’, ‘internet-config’, ‘command-line’ 등과 같이 문자열로 지정한다. 반환된 컨트롤 객체는 `open()`과 `open_new()` 메서드를 가지고 있다. 이 두 메서드는 앞의 두 함수와 동일한 인수를 받고 동일한 작업을 수행한다. name을 생략하면 기본 브라우저의 제어 객체가 반환된다.

**register(name, constructor[, controller])**

get( ) 함수에서 사용할 수 있게 새로운 브라우저 타입을 등록한다. name은 브라우저 이름이다. constructor는 브라우저에서 페이지를 여는 데 사용하는 제어 객체를 생성하려고 할 때 인수 없이 호출된다. controller는 대신 사용할 제어기 인스턴스이다. 이 값이 주어지면 constructor는 무시되고 None 값을 가져도 된다.

get( ) 함수가 반환하는 제어기 인스턴스 c는 다음 메서드를 갖는다.

**c.open(url[, new])**

open( ) 함수와 동일하다.

**c.open\_new(url)**

open\_new( ) 함수와 동일하다.

# 24장

P y t h o n   E s s e n t i a l   R e f e r e n c e

## 인터넷 데이터 처리와 인코딩

이 장에서는 base 64, HTML, XML, JSON 같이 널리 사용되는 인터넷 데이터 포맷과 인코딩 처리와 관련된 모듈을 설명한다.

### base64

base64 모듈은 base 64, base 32, base 16 인코딩을 사용하여 이진 데이터를 텍스트로 인코딩하거나 디코딩을 하는 데 사용한다. base 64는 메일 첨부나 HTTP 프로토콜에서 이진 데이터를 추가하는 데 널리 사용된다. 공식 문서는 RFC-3548과 RFC-1421에서 살펴볼 수 있다.

base 64 인코딩은 인코딩될 데이터를 24비트(3바이트) 단위로 그룹 짓는다. 각 24비트 그룹은 4개의 6비트 컴포넌트들로 나눠진다. 각 6비트 값은 다음 알파벳 중 하나를 사용하여 출력 가능한 ASCII 문자로 표현된다.

| 값     | 인코딩                        |
|-------|----------------------------|
| 0-25  | ABCDEFGHIJKLMNOPQRSTUVWXYZ |
| 26-51 | abcdefghijklmnopqrstuvwxyz |
| 52-61 | 0123456789                 |
| 62    | +                          |
| 63    | /                          |
| pad   | =                          |

입력 스트림에서 바이트 개수가 3의 배수(24 비트)가 아닐 경우 완전한 24비트 그룹을 만들기 위해 데이터를 채운다. 특수문자 '='가 채우기 문자로 사용되고 데이

터의 남은 부분을 채운다. 예를 들어, 16바이트 문자 순서열을 인코딩하면 5개의 3바이트 그룹을 형성하고 1바이트가 남게 된다. 3바이트 그룹을 형성하기 위해 남은 바이트에 채우기 문자들이 더해진다. 이 마지막 그룹은 2개의 base 64 알파벳 문자로 먼저 채워진다(실제 데이터 8비트를 담은 첫 번째 12비트). 이어서 채우기 비트들을 담은 순서열 ‘==’가 나온다. 유효한 base 64 인코딩은 인코딩 끝 부분에 0개, 1개 (=), 2개(==)의 채우기 문자를 갖게 된다.

base 32 인코딩은 이진 데이터를 40비트(5바이트) 단위로 그룹 짓는다. 각 40비트 그룹은 8개의 5비트 컴포넌트들로 나눠진다. 각 5비트 값은 다음 알파벳을 사용하여 인코딩된다.

| 값     | 인코딩                        |
|-------|----------------------------|
| 0-25  | ABCDEFGHIJKLMNOPQRSTUVWXYZ |
| 26-51 | 2-7                        |

base 64처럼 입력 스트림이 40비트 그룹으로 구성되지 않으면 40비트가 되게 데이터가 채워지며 '=' 문자가 채우기 문자로 사용된다. 마지막 그룹에 한 바이트만 있으면 최대 6개의 채우기 문자('=====')가 있을 수 있다.

base 16 인코딩은 데이터를 16진수로 인코딩하는 표준 인코딩 방식이다. 각 4비트 그룹은 '0'-'9'의 숫자와 'A'-'F'의 문자로 표현된다. base 16 인코딩에는 채우기나 채우기 문자가 존재하지 않는다.

#### **b64encode(s [, altchars])**

바이트 문자열 s를 base 64 인코딩을 사용하여 인코딩한다. 주어질 경우 altchars는 base 64 출력에서 일반적으로 나타나는 '+'와 '/' 문자의 대체 문자를 지정하는 두 문자로 구성된 문자열이다. base 64 인코딩을 파일 이름이나 URL에 적용할 때 유용하게 쓰인다.

#### **b64decode(s [, altchars])**

base 64로 인코딩된 문자열 s를 디코딩하여 디코딩된 바이트 문자열을 반환한다. 주어질 경우 altchars는 base 64 출력에서 일반적으로 나타나는 '+'와 '/' 문자의 대체 문자를 지정하는 두 문자로 구성된 문자열이다. 입력 s에 에러인 문자가 있거나 채우기가 잘못된 경우 TypeError를 발생시킨다.

#### **standard\_b64encode(s)**

표준 base 64 인코딩을 사용하여 바이트 문자열 s를 인코딩한다.

**standard\_b64decode(s)**

표준 base 64 인코딩을 사용하여 문자열 s를 디코딩한다.

**urlsafe\_b64encode(s)**

base 64를 사용하여 바이트 문자열 s를 인코딩하지만 '+' , '/' 대신에 '-' , '\_'를 사용한다. b64encode(s, '-\_')와 동일하다.

**urlsafe\_b64decode(s)**

URL을 문제 없이 인코딩할 수 있는 base 64 인코딩으로 인코딩된 문자열 s를 디코딩한다.

**b32encode(s)**

base 32 인코딩을 사용하여 바이트 문자열 s를 인코딩한다.

**b32decode(s [, casfold [, map01]])**

base 32 인코딩을 사용하여 문자열 s를 디코딩한다. casfold가 True이면 대문자나 소문자 모두 나타날 수 있다. 그렇지 않으면 대문자만 나타나야 한다(기본 값). map01이 주어지면 숫자 1이 매핑될 문자를 지정한다(예를 들어, 문자 'I' 또는 'L'). 이 인수가 주어지면 숫자 '0'이 'O'로 매핑된다. 입력 문자열에 에러인 문자가 있거나 채우기가 잘못된 경우 TypeError를 발생시킨다.

**b16encode(s)**

base 16(16진수) 인코딩을 사용하여 바이트 문자열 s를 인코딩한다.

**b16decode(s [, casfold])**

base 16(16진수) 인코딩을 사용하여 문자열 s를 디코딩한다. casfold가 True이면 대문자나 소문자 다 나타날 수 있다. 아니면 16진수 문자 'A'-'F'는 반드시 대문자여야 한다(기본 값). 입력 문자열에 에러인 문자가 있거나 어떤 식으로든 문제가 있으면 TypeError가 발생한다.

다음 함수들은 기존 파이썬 코드에서 볼 수 있는 오래된 base 64 모듈 인터페이스에 있는 것들 중 일부이다.

**decode(input, output)**

base 64로 인코딩된 데이터를 디코딩한다. input은 읽을 파일 이름 또는 파일 객체이다. output은 이진 모드로 쓸 파일 이름 또는 파일 객체이다.

### **decodestring(s)**

base 64로 인코딩된 문자열 s를 디코딩한다. 디코딩된 이진 데이터를 담은 문자열을 반환한다.

### **encode(input, output)**

base 64를 사용하여 데이터를 인코딩한다. input은 이진 모드로 읽을 파일 이름 또는 파일 객체이다. output은 쓸 파일 이름 또는 파일 객체이다.

### **encodestring(s)**

base 64를 사용하여 바이트 문자열 s를 인코딩한다.

## **binascii**

binascii 모듈은 이진 데이터와 base 64, BinHex, UUencoding 같은 다양한 ASCII 인코딩 사이에 변환을 하기 위한 저수준 함수를 담고 있다.

### **a2b\_uu(s)**

uu로 인코딩된 텍스트 s의 한 줄을 이진 형식으로 변환한 바이트 문자열을 반환한다. 각 줄은 45바이트보다 적을 수 있는 마지막 줄을 제외하고 일반적으로 45(이진)바이트를 담는다. 줄 데이터 끝에 공백이 있을 수도 있다.

### **b2a\_uu(data)**

이진 데이터 문자열을 uu로 인코딩된 ASCII 문자들을 담은 줄로 변환한다. data의 길이는 반드시 45바이트 이하여야 한다. 아니면 Error 예외가 발생한다.

### **a2b\_base64(string)**

base 64로 인코딩된 텍스트 문자열을 이진 데이터로 변환한 바이트 문자열을 반환한다.

### **b2a\_base64(data)**

이진 데이터 문자열을 base 64로 인코딩된 ASCII 문자들을 담은 줄로 변환한다. 생성되는 출력을 이메일로 전달하려면 data의 길이는 57바이트 이하여야 한다(아니면 잘릴 수도 있다).

### **a2b\_hex(string)**

16진수 숫자를 이진 데이터로 변환한다. 이 함수는 unhexlify(string)로 호출해도

된다.

#### **b2a\_hex(data)**

이진 데이터 문자열을 16진수 인코딩으로 변환한다. 이 함수는 hexlify(data)로 호출해도 된다.

#### **a2b\_hqx(string)**

BinHex 4로 인코딩된 데이터 문자열을 RLE(Run-Length Encoding) 압축 해제를 수행하지 않고 이진 데이터로 변환한다.

#### **rledecode\_hqx(data)**

data에 있는 이진 데이터를 RLE 압축 해제한다. 데이터 입력이 올바를 경우 압축 해제된 데이터를 반환한다. 아니면 Incomplete 예외가 발생한다.

#### **rlecode\_hqx(data)**

데이터를 BinHex 4 RLE로 압축한다.

#### **b2a\_hqx(data)**

이진 데이터를 BinHex 4로 인코딩된 ASCII 문자들을 담은 문자열로 변환한다. data는 이미 RLE 코딩이 되어 있어야 한다. 또한 data가 마지막 데이터 조각이 아니면 그 길이가 반드시 3으로 나눌 수 있어야 한다.

#### **crc\_hqx(data, crc)**

바이트 문자열 data의 BinHex 4 CRC 체크섬을 계산한다. crc는 체크섬의 시작 값이다.

#### **crc32(data [, crc])**

바이트 문자열 data의 CRC-32 체크섬을 계산한다. crc는 추가 초기 CRC 값이다. 이 값을 생략하면 crc는 기본으로 0이다.

## **CSV**

csv 모듈은 콤마로 구분된 값을 담은 파일(CSV)을 읽거나 쓸 때 사용한다. CSV 파일은 텍스트 행들을 담고 있으며 각 행은 보통 콤마(,)나 탭 구분자로 분리된 값을 담는다. 다음은 한 예를 보여준다.

```
Blues,Elwood,"1060 W Addison","Chicago, IL 60613",B263-1655-
```

2187", 116, 56

이 형식은 데이터베이스나 스프레드시트로 작업할 때 흔히 볼 수 있다. 예를 들어, 다른 프로그램에서 데이터베이스 표를 읽을 수 있도록 CSV 형식으로 표를 내보내기도 한다. 필드에 구분자가 들어 있을 때는 약간 복잡한 문제가 발생할 수 있다. 예를 들어, 앞의 예에서 콤마를 포함한 필드는 반드시 인용문 형식으로 변환해야 한다. 이 때문에 이런 파일을 처리할 때는 `split(',')` 같은 기본 문자열 연산자를 사용하는 것만으로 충분하지 않다.

```
reader(csvfile [, dialect [, **fmparms]])
```

입력 파일 csvfile의 각 줄에 대한 값들을 생성하는 읽기 객체를 반환한다. csvfile에는 매 반복마다 온전한 텍스트 한 줄을 생성하는 아무 반복 가능한 객체나 올 수 있다. 반환된 읽기 객체는 매 반복마다 문자열 리스트를 생성하는 반복자이다. dialect 매개변수는 방언 이름을 담은 문자열이거나 Dialect 객체일 수 있다. dialect 매개변수는 서로 다른 CSV 인코딩 사이의 차이를 설명한다. 이 모듈이 지원하는 내장 방언에는 ‘excel’(기본 값)과 ‘excel-tab’만 있지만 나중에 설명하듯이 사용자가 다른 방언을 정의할 수도 있다. fmparms는 방언의 다양한 측면을 설정하는 데 사용되는 키워드 인수들을 담는다. 다음은 키워드 인수를 사용할 수 있다.

| 키워드 인수                        | 설명                                                                                                                                     |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>delimiter</code>        | 필드를 분리하기 위해 사용할 문자(기본 값은 ‘,’)                                                                                                          |
| <code>doublequote</code>      | 필드에 인용 문자(quotchar)가 들어 있을 때 이를 어떻게 처리할 것인지를 지정하는 불리언 플래그. True이면 인용 문자가 하나 더 추가된다. False인 경우 탈출 문자(escapechar)가 앞에 붙는다. 기본 값은 True이다. |
| <code>escapechar</code>       | 필드에 구분자가 들어 있고 quoting이 QUOTE_NONE인 경우 탈출 문자로 사용할 문자. 기본 값은 None이다.                                                                    |
| <code>lineterminator</code>   | 줄 종료 순서열(기본 값은 ‘\r\n’)                                                                                                                 |
| <code>quotchar</code>         | 구분자를 담은 필드를 둘러싸는 데 사용할 인용 문자(기본 값은 “”)                                                                                                 |
| <code>skipinitialspace</code> | 이 값이 True이면 구분자 바로 다음의 빈칸은 무시된다(기본 값은 False).                                                                                          |

```
writer(csvfile [, dialect [, **fmparms]])
```

CSV 파일을 생성하는 데 사용할 수 있는 쓰기 객체를 반환한다. csvfile은 `write()` 메서드를 지원하는 파일과 유사한 객체라면 어떤 것이든 될 수 있다. dialect는 `reader()`

에서의 의미와 동일하며 다양한 CSV 인코딩 사이의 차이를 설명한다. `fmtparams`는 `reader()`에서의 의미와 동일하지만 다음에 나오는 추가 키워드 인수 하나를 더 지원한다.

| 키워드 인수               | 설명                                                                                                                                                                                                                                                                                     |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>quoting</code> | 출력 데이터의 인용 방식을 조절한다. <code>QUOTE_ALL</code> (모든 필드를 인용문으로 바꿈), <code>QUOTE_MINIMAL</code> (구분자를 포함하거나 인용 문자로 시작하는 필드를 인용문으로 바꿈), <code>QUOTE_NONNUMERIC</code> (모든 숫자가 아닌 필드를 인용문으로 바꿈), <code>QUOTE_NONE</code> (필드를 인용문으로 바꾸지 않음) 중 하나가 될 수 있다. 기본 값은 <code>QUOTE_MINIMAL</code> 이다. |

쓰기 인스턴스 `w`는 다음 메서드들을 지원한다.

#### `w.writerow(row)`

데이터 한 행을 파일에 쓴다. `row`는 반드시 문자열이나 숫자들의 순서열이어야 한다.

#### `w.writerows(rows)`

여러 데이터 행을 쓴다. `rows`는 반드시 `writerow()` 메서드에 전달되는 것과 동일한 행들의 순서열이어야 한다.

```
DictReader(csvfile [, fieldnames [, restkey [, restval [, dialect
[,**fmtparams]]]]])
```

일반적인 읽기 객체와 유사하게 작동하지만 파일을 읽을 때 문자열 리스트 대신 사전 객체를 반환하는 읽기 객체를 반환한다. `fieldnames`는 반환되는 사전에서 키로 사용되는 필드 이름 목록을 제공한다. 생략하면 입력 파일의 첫 번째 행에서 사전의 키 이름을 가져온다. `restkey`는 초과 데이터를 저장하는 데 사용되는 사전 키의 이름을 나타낸다. 예를 들어, 한 행이 필드 이름 개수보다 더 많은 필드를 가지고 있을 때 사용된다. `restval`은 입력에 존재하지 않는 필드에 대한 기본 값이다. 예를 들어, 한 행에서 필드 개수가 충분하지 않을 때 사용된다. `restkey`와 `restval`의 기본 값은 `None`이다. `dialect`와 `fmtparams`는 `reader()`에서와 동일한 의미를 가진다.

```
DictWriter(csvfile, fieldnames [, restval [, extrasaction [, dialect
[,**fmtparams]]]])
```

기존의 쓰기 객체와 유사하게 작동하지만 출력 행들을 사전에 쓰는 쓰기 객체를 반환한다. `fieldnames`는 파일에 쓸 속성의 이름과 순서를 나타낸다. `restval`은 사전

에서 `filednames`에 있는 필드 이름이 빠진 경우 사용할 값이다. `extrasaction`은 사전이 `fieldnames`에 존재하지 않는 키를 가지고 있을 때 수행할 일을 명시하는 문자열이다. `extrascation`의 기본 값은 ‘`raise`’이고 이 경우 `ValueError` 예외를 발생시킨다. ‘`ignore`’ 값은 사전에 있는 추가 값을 무시한다. `dialect`와 `fmtparams`는 `writer()`에서와 동일한 의미를 가진다.

`DictWriter` 인스턴스 `w`는 다음 메서드들을 지원한다.

#### `w.writerow(row)`

데이터 한 행을 파일에 쓴다. `row`는 필드 이름과 값을 매핑하는 사전이어야 한다.

#### `w.writerows(rows)`

여러 데이터 행을 파일에 쓴다. `rows`는 `writerow()` 메서드로 전달되는 것과 동일한 행들의 순서열이어야 한다.

#### `sniffer()`

CSV 파일 형식을 자동으로 감지하는 데 사용할 수 있는 `Sniffer` 객체를 생성한다. `Sniffer` 인스턴스 `s`는 다음 메서드들을 가진다.

#### `s.sniff(sample [, delimiters])`

`sample`에 있는 데이터를 살펴보고 데이터 형식을 나타내는 적절한 `Dialect` 객체를 반환한다. `sample`은 적어도 한 행 이상의 데이터를 담은 CSV 파일의 일부여야 한다. `delimiters`가 주어지면 이는 가능한 필드 구분자들을 담은 문자열이어야 한다.

#### `s.has_header(sample)`

`sample`에 있는 CSV 데이터를 살펴보고 첫 번째 행이 열 헤더들의 모음인 것처럼 보이면 `True`를 반환한다.

## 방언

`csv` 모듈에 있는 많은 함수와 메서드에서 특별한 `dialect` 매개변수를 사용한다. 이 매개변수는 서로 다른 형식으로 된 CSV 파일들을 처리하기 위해서 사용한다(아직 공식 “표준” 형식이란 것이 없다). 예를 들어 콤마로 구분된 값들, 탭으로 구분된 값들, 인용문 만들기 규칙 등 차이가 있는 부분들이 있다.

클래스 Dialect를 상속받은 다음에 reader( )와 writer( ) 함수에 주어지는 포맷 지정을 위한 매개변수들(delimiter, doublequote, escapechar, lineterminator, quotechar, quoting, skipinitialspace)과 동일한 속성들을 정의함으로써 방언을 생성할 수 있다.

다음 유ти리티 함수들은 방언을 관리하는 데 사용한다.

#### **register\_dialect(name, dialect)**

새로운 Dialect 객체 dialect를 name이란 이름으로 등록한다.

#### **unregister\_dialect(name)**

name이라는 이름을 갖는 Dialect 객체를 제거한다.

#### **get\_dialect(name)**

name이란 이름을 갖는 Dialect 객체를 반환한다.

#### **list\_dialects()**

모든 등록된 방언 이름을 반환한다. 현재 내장 방언으로 ‘excel’과 ‘excel-tab’ 두 가지가 있다.

## 예

```
import csv
간단한 CSV 파일을 읽는다.
f = open("scmods.csv","r")
for r in csv.reader(f):
 lastname, firstname, street, city, zip = r
 print("{0} {1} {2} {3} {4}".format(*r))

DictReader를 사용한다.
f = open("address.csv")
r = csv.DictReader(f,['lastname','firstname','street','city','zip'])
for a in r:
 print("{firstname} {lastname} {street} {city} {zip}".format(**a))

간단한 CSV 파일을 쓴다.
data = [
 ['Blues','Elwood','1060 W Addison','Chicago','IL','60613'],
 ['McGurn','Jack','4802 N Broadway','Chicago','IL','60640'],
]
f = open("address.csv","w")
w = csv.writer(f)
w.writerows(data)
f.close()
```

## email 패키지

email 패키지는 MIME 표준에 따라 인코딩된 이메일 메시지를 파싱하거나 조작하는 데 사용할 수 있는 다양한 함수와 객체를 제공한다.

여기서 email 패키지와 관련된 모든 내용을 다루기는 어렵고 여러분들도 원하지 않을 것이다. 이곳에서는 실전에서 흔히 접하게 되는 두 가지 문제, 즉 이메일 메시지를 파싱해서 유용한 정보를 추출하는 방법과 이메일을 보낼 수 있도록 `smtplib` 모듈을 사용하여 이메일 메시지를 생성하는 방법을 설명한다.

### 이메일 파싱

email 모듈은 메시지를 파싱하는 데 사용할 수 있는 두 개의 최상위 수준 함수를 제공한다.

#### `message_from_file(f)`

파일과 유사한 텍스트 모드로 열린 객체 `f`로부터 이메일 메시지를 파싱한다. 입력 메시지는 헤더, 텍스트, 첨부 파일을 모두 담은 MIME로 인코딩된 완전한 이메일 메시지여야 한다. `Message` 인스턴스를 반환한다.

#### `message_from_string(str)`

텍스트 문자열 `str`로부터 이메일 메시지를 읽어 파싱한다. `Message` 인스턴스를 반환한다.

앞의 함수에서 반환되는 `Message` 인스턴스 `m`은 사진을 흉내 내며 메시지 데이터를 살펴보기 위해 사용할 수 있는 다음의 연산들을 지원한다.

| 연산                               | 설명                                                                                |
|----------------------------------|-----------------------------------------------------------------------------------|
| <code>m[name]</code>             | 헤더 <code>name</code> 의 값을 반환한다.                                                   |
| <code>m.keys()</code>            | 모든 메시지 헤더의 이름을 담은 리스트를 반환한다.                                                      |
| <code>m.values()</code>          | 메시지 헤더 값들을 담은 리스트를 반환한다.                                                          |
| <code>m.items()</code>           | 메시지 헤더 이름과 값을 담은 튜플들의 리스트를 반환한다.                                                  |
| <code>m.get(name [, def])</code> | 헤더 <code>name</code> 의 값을 반환한다. <code>def</code> 는 헤더를 찾지 못했을 경우 반환 할 기본 값을 나타낸다. |
| <code>len(m)</code>              | 메시지 헤더 개수를 반환한다.                                                                  |
| <code>str(m)</code>              | 메시지를 문자열로 바꾼다. <code>as_string()</code> 메서드와 동일하다.                                |
| <code>name in m</code>           | <code>name</code> 이 메시지 헤더의 이름인 경우 <code>True</code> 를 반환한다.                      |

이 연산들과 더불어 m은 정보를 추출하는 데 사용할 수 있는 다음 메서드들도 가지고 있다.

#### **m.get\_all(name [, default])**

이름이 name인 헤더의 모든 값을 담은 리스트를 반환한다. 찾는 헤더가 없는 경우 default를 반환한다.

#### **m.get\_boundary([default])**

이메일 메시지의 ‘Content-type’ 헤더에서 찾은 경계 매개변수를 반환한다. 일반적으로 이메일 메시지에서 경계는 여러 부분을 분리할 때 사용하는 ‘=====0995017162==’ 같은 문자열이다. 경계 매개변수가 발견되지 않으면 default를 반환한다.

#### **m.get\_charset()**

메시지 페이로드의 문자 집합을 반환한다(예를 들어, ‘iso-8859-1’).

#### **m.getCharsets([default])**

메시지에서 나타난 모든 문자 집합(character set)의 리스트를 반환한다. 반환되는 리스트는 여러 부분으로 구성되는 메시지에서 각 부분의 문자 집합을 나타낸다. 각 부분의 문자 집합은 메시지의 ‘Content-type’ 헤더에서 얻는다. 문자 집합이 명시되어 있지 않거나 content-type 헤더가 없는 경우 각 부분의 문자 집합이 default로 설정된다(기본 값은 None).

#### **m.get\_content\_charset([default])**

메시지의 첫 번째 ‘Content-type’으로부터 문자 집합을 반환한다. 헤더가 존재하지 않거나 문자 집합이 명시되어 있지 않으면 default가 반환된다.

#### **m.get\_content\_mainType()**

주 content-type을 반환한다(예를 들어, ‘text’ 또는 ‘multipart’)

#### **m.get\_content\_subtype()**

부 content-type을 반환한다(예를 들어, ‘plain’ 또는 ‘mixed’).

#### **m.get\_content\_type()**

메시지의 content-type을 담은 문자열을 반환한다(예를 들어, ‘multipart/mixed’ 또는 ‘text/plain’).

**m.get\_default\_type()**

기본 content-type을 반환한다(예를 들어, 단순한 메시지에 대해서는 ‘text/plain’)

**m.get\_filename([default])**

‘Content-Disposition’ 헤더에 filename 매개변수가 존재하면 이를 반환한다. 헤더가 생략되었거나 filename 매개변수가 없으면 default를 반환한다.

**m.get\_param(param [, default [, header [, unquote]]])**

종종 헤더 ‘Content-Type: text/plain; charset=“utf-8”; format=flowed’에는 ‘charset’나 ‘format’ 같이 헤더에 붙어 있는 매개변수들이 있다. 이 메서드는 지정된 헤더 매개변수의 값을 반환한다. param은 매개변수의 이름이고 default는 매개변수가 없을 경우 반환되는 기본 값이다. header는 헤더의 이름이고 unquote는 매개변수를 인용문으로 만들 것인지 여부를 나타낸다. header 값이 주어지지 않으면 ‘Content-type’ 헤더에서 매개변수들이 추출된다. unquote의 기본 값은 True이다. 반환되는 값은 문자이거나 RFC-2231에 따라 인코딩된 경우에는 튜플 (charset, language, value)가 된다. 여기서 charset은 ‘iso-8859-1’ 같은 문자열이고 language는 ‘en’ 같은 언어 코드를 담은 문자열이며 value는 매개변수 값이다.

**m.get\_params([default [, header [, unquote]]])**

헤더의 모든 매개변수를 리스트로 반환한다. default는 헤더를 찾을 수 없는 경우 반환하는 값을 나타낸다. header가 생략된 경우 ‘Content-type’ 헤더가 사용된다. unquote는 값을 인용문으로 만들 것인지를 지정하는 플래그이다(기본 값은 True). 반환된 리스트는 튜플 (name, value)들을 담으며 여기서 name은 매개변수의 이름을, value는 get\_param() 메서드에서 반환되는 값을 나타낸다.

**m.get\_payload([i [, decode]])**

메시지의 페이로드를 반환한다. 메시지가 단순 메시지인 경우 메시지 몸체를 담은 바이트 문자열을 반환한다. 메시지가 여러 부분으로 구성된 메시지인 경우 모든 부분을 담은 리스트를 반환한다. 여러 부분으로 구성된 메시지인 경우 i는 리스트에 대한 추가 색인을 나타낸다. 이 값을 주면 해당 부분의 내용이 반환된다. decode가 True인 경우 페이로드는 헤더 ‘Content-Transfer-Encoding’에 명시된(예를 들어 ‘quoted-printable’, ‘base64’ 등) 설정에 따라 디코딩된다. 여러 부분으로 구성되지 않은 단순 메시지의 페이로드를 디코딩하려면 i를 None으로 설정하고 decode를

True로 설정하거나, decode를 키워드 인수를 통해 설정해야 한다. 페이로드는 무가공 콘텐츠를 담은 바이트 문자열로 반환된다. 페이로드가 UTF-8 또는 다른 인코딩을 통해 인코딩된 텍스트인 경우 decode() 메서드를 사용해서 적절히 변환해야 한다.

#### **m.get\_unixfrom()**

있을 경우 유닉스 스타일의 ‘From ...’ 줄을 반환한다.

#### **m.is\_multipart()**

m이 여러 부분으로 구성된 메시지이면 True를 반환한다.

#### **m.walk()**

메시지의 모든 부분을 반복하는 생성기를 만든다. 각 부분은 다시 Message 인스턴스로 표현된다. 반복은 깊이 우선 방법에 따라 이루어진다. 일반적으로 이 함수는 여러 부분으로 구성된 메시지의 모든 부분을 처리하는 데 사용한다.

마지막으로 Message 인스턴스는 저수준 파싱 과정과 관련된 속성들을 가진다.

#### **m.preamble**

여러 부분 메시지에서 헤더의 끝을 나타내는 빈 줄과 메시지의 첫 번째 부분을 표시하는 경계 문자열 사이에 나타나는 텍스트.

#### **m.epilogue**

마지막 경계 문자열과 메시지 끝 다음에 나타나는 텍스트.

#### **m.defects**

메시지를 파싱할 때 발견된 모든 메시지 결함을 담은 리스트. 자세한 내용은 온라인 문서에서 email.errors 모듈을 참고한다.

다음은 이메일 메시지를 파싱하기 위해 Message 클래스를 어떻게 사용하는지를 보여주는 예이다. 다음 코드는 이메일 메시지를 읽고 유용한 정보를 담은 헤더들의 내용을 간단히 요약한 것을 출력하고 메시지에서 일반 텍스트 부분을 출력한 다음 첨부 파일을 저장한다.

```
import email
import sys
f = open(sys.argv[1],"r") # 메시지 파일 열기
m = email.message_from_file(f) # 메시지 파싱

송신자와 수신자에 대한 간단한 요약 정보 출력
```

```

print("From : %s" % m["from"])
print("To : %s" % m["to"])
print("Subject : %s" % m["subject"])
print("")

if not m.is_multipart():
 # 단순 메시지. 간단히 페이로드를 출력.
 payload = m.get_payload(decode=True)
 charset = m.get_content_charset('iso-8859-1')
 print(payload.decode(charset))
else:
 # 여러 부분 메시지. 모든 부분을 순회하고
 # 1. text/plain 조각들을 출력
 # 2. 첨부 파일들을 저장
 for s in m.walk():
 filename = s.get_filename()
 if filename:
 print("Saving attachment: %s" % filename)
 data = s.get_payload(decode=True)
 open(filename,"wb").write(data)
 else:
 if s.get_content_type() == 'text/plain':
 payload = s.get_payload(decode=True)
 charset = s.get_content_charset('iso-8859-1')
 print(payload.decode(charset))

```

이 예에서 주목할 점은 메시지의 페이로드를 추출하는 연산은 항상 바이트 문자열을 반환한다는 점이다. 페이로드가 텍스트를 담고 있으면 특정 문자 집합에 따라 디코딩할 필요가 있다. 이 예에서는 `m.get_content_charset()`와 `payload.decode()`로 이러한 변환을 수행하였다.

## 이메일 작성

이메일 메시지를 작성하려면 `email.message` 모듈에 정의된 `Message` 객체의 빈 인스턴스를 생성하거나 이메일 메시지를 파싱하여 생성한 `Message` 객체(앞 절 참고)를 사용하여야 한다.

### `Message()`

기본적으로 비어 있는 새로운 메시지를 생성한다.

`Message` 인스턴스 `m`은 메시지에 내용, 헤더 및 기타 정보를 채우는 데 사용할 수 있는 다음 메서드들을 지원한다.

#### `m.add_header(name, value, **params)`

새로운 메시지 헤더를 추가한다. `name`은 헤더의 이름이고 `value`는 헤더의 값을

나타내며 param은 추가 매개변수를 제공하는 키워드 인수 집합이다. 예를 들어, add\_header('Foo', 'Bar', spam='major')는 메시지의 헤더 줄에 'Foo: Bar; spam= "major"'를 추가한다.

#### **m.as\_string([unixfrom])**

전체 메시지를 문자열로 변환한다. unixfrom은 불리언 플래그이다. True로 설정되면 유닉스 스타일의 'From ...' 줄이 첫 번째 줄이 된다. 기본적으로 unixfrom은 False이다.

#### **m.attach(payload)**

여러 부분 메시지에 첨부 파일을 추가한다. payload는 또 다른 Message 객체(예를 들어, email.mime.text.MIMEText)여야 한다. 내부적으로 payload는 메시지의 여러 부분을 추적하는 리스트에 덧붙여진다. 메시지가 여러 부분 메시지가 아니면 set\_payload()를 사용해서 메시지 몸체를 단순한 문자열로 설정하도록 한다.

#### **m.del\_param(param [, header [, requote]])**

헤더 header의 매개변수 param을 삭제한다. 예를 들어, message가 'Foo: Bar; spam= "major"' 헤더를 가지고 있다면 del\_param('spam', 'Foo')는 헤더의 'spam= "major"' 부분을 삭제한다. requote가 True이면(기본 값) 헤더가 다시 쓰여질 때 남아 있는 모든 값이 인용 형식이 된다. header를 생략하면 이 연산이 'Content-type' 헤더에 적용된다.

#### **m.replace\_header(name, value)**

첫 번째로 나타난 헤더 name의 값을 value로 바꾼다. header를 찾지 못한 경우 KeyError가 발생한다.

#### **m.set\_boundary(boundary)**

메시지의 경계 매개변수를 문자열 boundary로 설정한다. 이 문자열은 메시지의 'Content-type' 헤더의 boundary 매개변수로서 추가된다. 메시지에 content-type 헤더가 없는 경우 HeaderParseError 예외가 발생한다.

#### **m.set\_charset(charset)**

메시지에서 사용되는 기본 문자 집합을 설정한다. charset은 'iso-8859-1' 또는 'euc-jp' 같은 문자열일 수 있다. 문자 집합을 설정하면 보통 메시지의 'Content-type' 헤더에 매개변수가 추가된다(예를 들어, 'Content-type: text/html; charset=

“iso-8859-1”).

#### **m.set\_default\_type(ctype)**

메시지의 기본 콘텐트 타입을 ctype으로 설정한다. ctype은 ‘text/plain’ 또는 ‘message/rfc822’ 같은 MIME 타입을 담은 문자열이어야 한다. 지정한 타입이 메시지의 ‘Content-type’ 헤더에 추가되지는 않는다.

#### **m.set\_param(param, value [, header [, requote [, charset [, language]]]])**

헤더 매개변수의 값을 설정한다. param은 매개변수 이름을, value는 매개변수 값을 나타낸다. header는 헤더 이름을 나타내며 기본 값은 ‘Content-type’이다. requote는 매개변수를 추가한 후 헤더의 모든 값에 따옴표를 붙일 것인지를 지정한다. 기본으로 이 값은 True이다. charset과 language는 추가로 문자 집합과 언어 정보를 지정한다. 이 두 값을 제공하면 해당 매개변수는 RFC-2231에 따라 인코딩된다. 이 경우 param=“‘iso-8859-1’en-us’some%20value” 같은 매개변수 텍스트가 생성된다.

#### **m.set\_payload(payload [, charset])**

전체 메시지 페이로드를 payload로 설정한다. 단순 메시지의 경우 payload는 메시지의 몸체를 담은 바이트 문자열이다. 여러 부분 메시지의 경우 payload는 Message 객체들을 담은 리스트이다. charset은 추가로 텍스트를 인코딩하는 데 사용한 문자 집합을 지정한다(set\_charset 참고).

#### **m.set\_type(type [, header [, requote]])**

‘Content-type’ 헤더에 사용할 타입을 설정한다. type은 ‘text/plain’나 ‘multipart/mixed’ 같은 타입을 지정하는 문자열이다. header는 기본 헤더인 ‘Content-type’ 헤더가 아닌 다른 헤더를 지정할 때 쓴다. requote는 이미 헤더에 추가되어 있는 매개변수의 값을 따옴표로 둘러쌀지 여부를 지정한다. 기본으로 이 값은 True이다.

#### **m.set\_unixfrom(unixfrom)**

유닉스 스타일인 ‘From ...’ 줄의 텍스트를 설정한다. unixfrom은 ‘From’ 텍스트를 포함한 완전한 텍스트를 담은 문자열이다. 여기서 설정한 텍스트는 m.as\_string()의 매개변수 unixfrom이 True일 경우에만 출력된다.

새로운 무가공 Message 객체를 생성하고 매번 새로 내용을 채워넣는 대신 서로

다른 콘텐츠에 대응하는 미리 구축된 메시지 객체를 사용할 수 있다. 이러한 메시지 객체는 특히 여러 부분으로 구성된 MIME 메시지를 생성할 때 유용하게 쓰인다. 예를 들어, 새로운 메시지를 생성하고 Message의 attach() 메서드를 사용하여 각 부분을 추가할 수 있다. 각 객체는 서로 다른 하위 모듈에 정의되어 있고 어디에 정의되어 있는지는 각 객체를 설명할 때 언급한다.

#### **MIMEApplication(data [, subtype [, encoder [, \*\*params]]])**

email.mime.application에 정의되어 있다. 응용 프로그램 데이터를 담은 메시지를 생성한다. data는 무가공 데이터를 담은 바이트 문자열이다. subtype은 데이터의 하위 타입을 나타내며 기본 값은 ‘octet-stream’이다. encoder는 email.encoders 하위 패키지에 있는 인코딩 함수이다. 기본으로 데이터는 base 64로 인코딩된다. params는 메시지의 ‘Content-type’ 헤더에 추가될 키워드 인수와 값을 나타낸다.

#### **MIMEAudio(data [, subtype [, encoder [, \*\*params]]])**

email.mime.audio에 정의되어 있다. 오디오 데이터를 담은 메시지를 생성한다. data는 무가공 이진 오디오 데이터를 담은 바이트 문자열이다. subtype은 데이터의 타입을 나타내며 ‘mpeg’ 또는 ‘wav’ 같은 문자열이다. subtype이 주어지지 않으면 sndhdr 모듈을 사용하여 데이터를 살펴봄으로써 오디오 타입을 추측한다. encoder와 params는 MIMEApplication에서와 동일한 의미를 가진다.

#### **MIMEImage(data [, subtype [, encoder [, \*\*params]]])**

email.mime.image에 정의되어 있다. 이미지 데이터를 담은 메시지를 생성한다. data는 무가공 이미지 데이터를 담은 바이트 문자열이다. subtype은 이미지의 타입을 나타내며 ‘jpg’ 또는 ‘png’ 같은 문자열이다. subtype이 주어지지 않으면 imghdr 모듈의 함수를 사용하여 이미지 타입을 추측한다. encoder와 params는 MIMEApplication에서와 동일한 의미를 가진다.

#### **MIMEMessage(msg [, subtype])**

email.mime.message에 정의되어 있다. 여러 부분으로 구성되지 않는 새로운 MIME 메시지를 생성한다. msg는 메시지의 기본 페이로드를 담은 메시지 객체이다. subtype은 메시지의 타입으로 기본 값은 ‘rfc822’이다.

#### **MIMEMultipart([subtype [, boundary [, subparts [, \*\*params]]]])**

email.mime.multipart에 정의되어 있다. 새로운 MIME 여러 부분 메시지를 생성한

다. subtype은 ‘Content-type: multipart/subtype’ 헤더에 추가할 하위 타입을 나타낸다. 기본 값으로 subtype은 ‘mixed’이다. boundary는 메시지의 각 하위 부분을 만드는 경계 구분자를 나타낸다. None으로 설정하거나 생략하면 자동으로 적당한 경계가 결정된다. subparts는 메시지의 내용을 구성하는 Message 객체들의 순서열이다. params는 메시지의 ‘Content-type’ 헤더에 추가될 키워드 인수와 값을 나타낸다. 일단 여러 부분 메시지가 생성되면 추가 하위 부분은 Message.attach() 메서드를 사용하여 추가할 수 있다.

#### **MIMEText(data [, subtype [, charset]])**

email.mime.text에 정의되어 있다. 텍스트 데이터를 담은 메시지를 생성한다. data는 메시지 페이로드를 담은 문자열이다. subtype은 ‘plain’(기본 값) 또는 ‘html’과 같이 텍스트 타입을 나타내는 문자열이다. charset은 문자 집합으로 기본 값은 ‘us-ascii’이다. 메시지의 내용에 따라 메시지를 인코딩할 수도 있다.

다음은 이 절에서 살펴본 클래스들을 사용하여 이메일 메시지를 작성하고 보내는 방법을 보여주는 예이다.

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.audio import MIMEAudio

sender = "jon@nogodiggydie.net"
receiver= "dave@dabeaz.com"
subject = "Faders up!"
body = "I never should have moved out of Texas. -J.\n"
audio = "TexasFuneral.mp3"

m = MIMEMultipart()
m["to"] = receiver
m["from"] = sender
m["subject"] = subject

m.attach(MIMEText(body))
apart = MIMEAudio(open(audio,"rb").read(),"mpeg")
apart.add_header("Content-Disposition","attachment",filename=audio)
m.attach(apart)

이메일 메시지를 보낸다.
s = smtplib.SMTP()
s.connect()
s.sendmail(sender, [receiver],m.as_string())
s.close()
```

**Note**

- 몇 가지 고급 커스터마이즈 기능과 설정 옵션에 관해서는 다루지 않았다. 이 모듈의 고급 사용법을 알고 싶으면 온라인 문서를 참고한다.
- email 패키지는 적어도 네 번 정도 버전이 바뀌었고 이에 따라 기본 프로그래밍 인터페이스도 변경되었다(하위 모듈의 이름이 바뀌고 클래스가 다른 위치로 이동하는 등). 이 장에서는 파이썬 2.6과 파이썬 3.0에서 사용되는 인터페이스의 4.0 버전을 다루었다. 만약 레거시 코드를 다루고 있는 중이라면 기본 개념은 그대로 적용되지만 클래스나 하위 모듈이 조금 다를 수 있다는 것을 염두에 두기 바란다.

**hashlib**

hashlib 모듈은 다양한 보안 해시와 MD5나 SHA1 같은 메시지 요약 알고리즘(message digest algorithm)을 구현한다. 해시 값을 계산하려면 다음 함수 중 하나를 호출하면 된다. 각 함수 이름은 알고리즘 이름과 같다.

| 함수       | 설명                 |
|----------|--------------------|
| md5()    | MD5 해시(128 비트)     |
| sha1()   | SHA1 해시 (160 비트)   |
| sha224() | SHA224 해시 (224 비트) |
| sha256() | SHA256 해시 (256 비트) |
| sha384() | SHA384 해시 (384 비트) |
| sha512() | SHA512 해시 (512 비트) |

이 함수들이 반환하는 요약 객체의 인스턴스 d는 다음 인터페이스를 갖는다.

| 메서드 또는 속성      | 설명                                                                              |
|----------------|---------------------------------------------------------------------------------|
| d.update(data) | 새로운 데이터로 해시를 업데이트한다. data는 바이트 문자열이어야 한다. 여러 번 호출하는 것은 연결된 데이터에 한 번 호출하는 것과 같다. |
| d.digest()     | 무가공 바이트 문자열로 요약 값을 반환한다.                                                        |
| d.hexdigest()  | 일련의 16진수 숫자들로 인코딩된 요약 값을 담은 텍스트 문자열을 반환한다.                                      |
| d.copy()       | 요약 값의 복사본을 반환한다. 이 복사본은 기존의 요약 값의 내부 상태를 그대로 담는다.                               |
| d.digest_size  | 결과로 생성되는 해시의 크기를 바이트로                                                           |
| d.block_size   | 해시 알고리즘의 내부 구조 크기를 바이트로                                                         |

이 모듈에는 다음에 나오는 생성 인터페이스도 있다.

#### **new(hashname)**

새로운 요약 객체를 생성한다. hashname은 ‘md5’나 ‘sha256’ 같이 사용할 해싱 알고리즘의 이름을 담은 문자열이다. 해시의 이름은 적어도 앞에 나온 해싱 알고리즘 중 하나이거나 OpenSSL 라이브러리에 있는 해시 알고리즘(설치된 버전에 따라 다를 수 있다)의 이름 중 하나여야 한다.

## **hmac**

hman 모듈은 RFC-2104에 나와 있는 HMAC(Keyed-Hashing for Message Authentication)를 지원한다. HMAC는 MD5나 SHA-1 같은 암호화 해싱 함수를 기반으로 하는 메시지 인증 메커니즘이다.

#### **new(key [, msg [, digest]])**

새로운 HMAC 객체를 생성한다. 여기서 key는 해시를 위한 시작 키를 담은 바이트 문자열이고 msg는 처리할 초기 데이터를 담으며 digest는 암호화 해싱에 사용할 요약 생성기이다. digest의 기본 값은 hashlib.md5이다.

일반적으로 초기 키 값은 강력한 암호화 능력을 갖는 난수 생성기를 사용하여 무작위로 선정된다.

HMAC 객체 h는 다음 메서드들을 가진다.

#### **h.update(msg)**

HMAC 객체에 문자열 msg을 추가한다.

#### **h.digest()**

지금까지 처리된 모든 데이터의 요약을 담은 바이트 문자열을 반환한다. 반환되는 문자열의 길이는 사용된 해싱 함수에 따라 다르다. MD5의 경우 16개 문자이고 SHA-1인 경우 20개 문자이다.

#### **h.hexdigest()**

16진수 숫자들을 담은 문자열로 요약 내용을 반환한다.

#### **h.copy()**

HMAC 객체의 복사본을 만든다.

## 예

hmac 모듈은 메시지 송신자를 인증해야 할 필요가 있는 응용 프로그램에서 주로 사용된다. new()의 key 매개변수는 송신자와 수신자 모두가 알고 있는 보안 키를 나타내는 바이트 문자열이어야 한다. 송신자는 메시지를 보낼 때 주어진 키로 새로운 HMAC 객체를 생성하고 보낼 메시지 데이터로 이 객체를 업데이트한 후 메시지 데이터를 HMAC 요약 값과 함께 수신자에게 보낸다. 수신자는 자신의 HMAC 요약 값을 계산하고 받은 요약 값과 비교함으로써 메시지를 검증한다. 다음 예를 살펴보자.

```
import hmac secret_key = b"peekaboo" # 나만 알고 있는 바이트 문자열.
 # 보통 os.urandom() 같은 함수로 생성한
 # 무작위 바이트 문자열을 사용한다.

data = b"Hello World" # 송신할 메시지

어디론가 메시지를 보낸다.
out은 데이터를 보낼 소켓 또는 기타 I/O 채널이다.
h = hmac.new(secret_key)
h.update(data)
out.send(data) # 데이터 송신
out.send(h.digest()) # 요약 송신

메시지를 받는다.
in은 데이터를 받을 소켓 또는 기타 I/O 채널을 나타낸다.
h = hmac.new(secret_key)
data = in.receive() # 메시지 데이터를 수신
h.update(data)
digest = in.receive() # 송신자에서 보낸 요약을 수신
if digest != h.digest():
 raise AuthenticationError('Message not authenticated')
```

## HTMLParser

파이썬 3에서 이 모듈의 이름은 `html.parser`이다. HTMLParser 모듈은 HTML과 XHTML 문서를 파싱하는 데 사용할 수 있는 HTMLParser 클래스를 정의한다. 이 모듈을 사용하려면 HTMLParser에서 상속을 받은 후 적절하게 메서드들을 재정의하면 된다.

### `HTMLParser()`

HTML 파서를 생성하는 데 사용하는 기반 클래스이다. 인수 없이 초기화된다.

HTMLParser의 인스턴스 h는 다음 메서드들을 가진다.

#### **h.close()**

파서를 닫고 아직 처리되지 않은 데이터를 처리하게 한다. 이 메서드는 모든 HTML 데이터가 파서에 입력되고 나면 호출된다.

#### **h.feed(data)**

파서에 새로운 데이터를 제공한다. 데이터는 즉시 파싱된다. 데이터가 불완전한 경우(예를 들어, HTML 엘리먼트가 불완전하게 끝난 경우) 불완전한 부분은 보관 했다가 다음에 추가 데이터로 feed()가 호출되었을 때 파싱된다.

#### **h.getpos()**

현재 줄 번호와 현재 줄 안에서 문자 오프셋을 튜플 (line, offset)로 반환한다.

#### **h.get\_starttag\_text()**

가장 최근에 열린 시작 태그에 해당하는 텍스트를 반환한다.

#### **h.handle\_charref(name)**

'&#ref;' 같은 문자 참조를 만났을 때 이 처리기 메서드가 호출된다. name은 참조 이름을 담은 문자열이다. 예를 들어, '&#229;'을 파싱할 때 name은 '229'로 설정 된다.

#### **h.handle\_comment(data)**

주석을 만났을 때 이 처리기 메서드가 호출된다. data는 주석 텍스트를 담은 문자열이다. 예를 들어, 주석 '<!-comment->'을 파싱할 때 data는 'comment' 텍스트를 담는다.

#### **h.handle\_data(data)**

태그 사이에 나타난 데이터를 처리하기 위해 이 처리기가 호출된다. data는 텍스트를 담은 문자열이다.

#### **h.handle\_decl(decl)**

이 처리기는 '<!DOCTYPE HTML ...>' 같은 선언을 처리하기 위해서 호출된다. decl은 앞쪽 '<!'와 뒤쪽 '>'를 제외한 텍스트를 담은 문자열이다.

#### **h.handle\_endtag(tag)**

이 처리기는 끝 태그를 만났을 때 호출된다. tag는 소문자로 변환된 태그의 이름

이다. 예를 들어, 끝 태그가 ‘</BODY>’이면 tag는 문자열 ‘body’가 된다.

#### **h.handle\_entityref(name)**

이 처리기는 ‘&name;’ 같은 개체 참조(entity reference)를 처리하기 위해 호출된다. name은 참조 이름을 담은 문자열이다. 예를 들어, ‘&lt;’을 파싱할 때 name은 ‘lt’로 설정된다.

#### **h.handle\_pi(data)**

이 처리기는 ‘<?processing instruction>’ 같은 처리 명령에 대해서 호출된다. data는 앞쪽 ‘!와 뒤쪽 ’’를 제외한 처리 명령의 텍스트를 담은 문자열이다. ‘<?...?>’ 형식의 XHTML 스타일 명령에 대해 호출될 때는 마지막 ‘?’가 data에 들어간다.

#### **h.handle\_startendtag(tag, attrs)**

이 처리기는 ‘<tag name=“value” ... />’ 같은 XHTML 스타일의 빈 태그를 처리한다. tag는 태그의 이름을 담은 문자열이다. attrs는 어트리뷰트(attribute) 정보를 담으며 (name, value) 형식의 튜플들을 담은 리스트이다. 여기서 name은 소문자로 변환된 어트리뷰트의 이름이고 value는 어트리뷰트의 값이다. 값을 추출할 때 따옴표와 문자 개체는 대체된다. 예를 들어, ‘< a href=“http://www.foo.com” />’ 파싱하면 tag는 ‘a’가 되고 attrs는 [('href', 'http://www.foo.com')]가 된다. 파생 클래스에서 이 메서드를 정의하지 않으면 기본 구현으로 handle\_starttag()와 handle\_endtag()가 호출된다.

#### **h.handle\_starttag(tag, attrs)**

이 처리기는 ‘< tag name=“value” ...>’ 같은 시작 태그를 처리한다. tag와 attrs는 handle\_startendtag( )의 것과 동일하다.

#### **h.reset()**

파서를 재설정한다. 처리되지 않은 데이터는 버린다.

이 모듈에는 다음 예외가 있다.

#### **HTMLParserError**

파싱 에러로 인해 발생된 예외. 이 예외는 세 가지 속성을 가진다. 속성 msg는 에러를 설명하는 메시지를 담고 속성 lineno은 파싱 에러가 발생한 곳의 줄 번호이며 속성 offset은 해당 줄에서 문자 오프셋을 나타낸다.

## 예

다음은 `urllib` 패키지를 사용하여 HTML 문서를 추출한 다음 ‘`<a href="...">`’ 선언으로 지정한 모든 링크를 출력하는 예이다.

```
printlinks.py
try:
 from HTMLParser import HTMLParser
 from urllib2 import urlopen
except ImportError:
 from html.parser import HTMLParser
 from urllib.request import urlopen
import sys

class PrintLinks(HTMLParser):
 def handle_starttag(self, tag, attrs):
 if tag == 'a':
 for name, value in attrs:
 if name == 'href': print(value)

p = PrintLinks()
u = urlopen(sys.argv[1])
data = u.read()
charset = u.info().getparam('charset') # 파이썬 2
#charset = u.info().get_content_charset() # 파이썬 3
p.feed(data.decode(charset))
p.close()
```

이 예에서 `urllib`을 사용하여 추출된 HTML은 바이트 문자열로 반환된다는 점을 주의한다. 이를 적절히 파싱하려면 문서의 문자 집합에 따라 텍스트를 적절히 디코딩하여야 한다. 이 예에서 파이썬 2와 파이썬 3에서 문자 집합 정보를 어떻게 얻는지 보였다.

### Note

`HTMLParser`의 파싱 기능은 좀 부족하다. 아주 복잡하거나 비정형 HTML 문서에 대해서 파서가 제대로 작동하지 않을 수 있다. 이 모듈이 저수준의 기능을 제공하기 때문에 유용성이 좀 떨어진다는 점을 사용자들도 알고 있다. HTML 페이지에서 데이터를 긁어모으는 프로그램을 작성하려고 한다면 `Beautiful Soup` 패키지를 사용해보도록 한다(<http://pypi.python.org/pypi/BeautifulSoup>).

## json

`json` 모듈은 JSON(JavaScript Object Notation)으로 표현된 객체를 직렬화 및 역직

렬화하는 데 사용한다. JSON에 대한 더 많은 정보는 <http://json.org>에서 찾을 수 있다. JSON 문법은 JavaScript 문법의 부분집합이다. 우연히도 리스트와 사전을 표현하기 위한 문법은 파이썬 문법과 거의 같다. 예를 들어, JSON 배열은 [value1, value2, ...]처럼 쓰고 JSON 객체는 {name : value, name : value, ....}처럼 쓴다.

다음 목록은 JSON 값과 파이썬 값이 어떻게 매핑되는지를 보여준다. 괄호 안에 쓴 파이썬 타입은 인코딩할 때는 쓸 수 있지만 디코딩할 때는 쓰이지 않는다(대신 괄호 밖에 있는 타입이 반환된다).

| JSON 타입 | 파이썬 타입             |
|---------|--------------------|
| object  | dict               |
| array   | list(tuple)        |
| string  | unicode(str,bytes) |
| number  | int, float         |
| true    | True               |
| false   | False              |
| null    | None               |

문자열 데이터는 유니코드로 작성되었다고 가정해야 한다. 인코딩 도중에 바이트 문자열이 나타나면 기본으로 ‘utf-8’을 사용하여(다른 것으로 설정 가능) 유니코드 문자열로 디코딩한다. 디코딩 과정에서 JSON 문자열은 항상 유니코드로 반환된다.

다음 함수들은 JSON 문서를 인코딩하거나 디코딩하는 데 사용한다.

### **dump(obj, f, \*\*opts)**

obj를 파일과 유사한 객체 f로 직렬화한다. opts는 키워드 인수들을 나타내며 직렬화 과정을 제어한다.

| 키워드 인수         | 설명                                                                                                                                            |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| skipkeys       | 사전의 키(값이 아닌)가 문자열이나 숫자 같은 기본 타입이 아닐 때 어떻게 할 것인지를 제어하는 불리언 플래그. True이면 키를 건너뛴다. False이면(기본 값) TypeError가 발생한다.                                 |
| ensure_ascii   | 유니코드 문자열을 파일 f에 쓸 수 있는지 여부를 결정하는 불리언 플래그. 기본 값은 False다. codecs 모듈로 생성했거나 특정 인코딩 집합을 통해 연 경우처럼 f가 유니코드를 올바르게 처리할 수 있는 파일인 경우에만 True로 설정하도록 한다. |
| check_circular | 컨테이너에 대해 순환 참조를 검사할 것인지를 결정하는 불리언 플래그. 기본 값은 True이다. False로 설정했는데 순환 참조가 있으면 OverflowError 예외가 발생한다.                                          |
| allow_nan      | 범위를 벗어난 부동 소수점 값을 직렬화할 것인지를 결정하는 불리                                                                                                           |

|            |                                                                                                                                                                         |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|            | 언 플래그, 기본 값은 True이다.                                                                                                                                                    |
| cls        | 사용할 JSONEncoder의 하위 클래스. JSONEncoder를 상속하여 자신의 인코더를 생성한 경우 명시하여야 한다. dump( )에 주어진 키워드 인수가 더 있으면 이 클래스의 생성자의 인수로 전달된다.                                                   |
| indent     | 배열이나 객체의 멤버를 출력할 때 사용할 들여쓰기 양을 설정하는 음이 아닌 정수. 이를 설정하면 깔끔한 출력 결과를 얻게 된다. 기본 값은 None이고 가장 간결하게 결과를 표현한다.                                                                  |
| separators | (item_separator, dict_separator) 형태의 튜플로서 item_separator는 배열 항목 사이를 구분하는 데 사용되는 구분자를 담은 문자열이고 dict_separator는 사전의 키와 값을 구분하는 데 사용되는 구분자를 담은 문자열이다. 기본 값은 (', ', ':')이다. |
| encoding   | 유니코드 문자열에 대해 사용할 인코딩. 기본 값은 'utf-8'                                                                                                                                     |
| default    | 기본 지원 타입이 아닌 객체를 직렬화하는 데 사용되는 함수. 이 함수는 직렬화가 가능한 값(예를 들어, 문자열)을 반환하거나 TypeError를 발생시켜야 한다. 기본으로 지원되지 않는 타입에 대해 TypeError가 발생한다.                                         |

**dumps(obj, \*\*opts)**

결과를 담은 문자열을 반환하는 것을 제외하고는 dump( )와 동일하다.

**load(f, \*\*opts)**

파일과 유사한 객체 f에서 JSON 객체를 역직렬화하여 반환한다. opts는 디코딩 과정을 제어하는 키워드 인수들을 나타내며 잠시 후 설명한다. 이 함수는 f의 전체 내용을 소비하기 위해 f.read( )를 호출한다. 따라서 소켓을 통해 더 큰 데이터 스트림 또는 계속되는 데이터 스트림의 일부로 JSON 데이터를 받는 것처럼 스트리밍되는 파일에 대해서는 사용하지 않아야 한다.

| 키워드 인수      | 설명                                                                                                               |
|-------------|------------------------------------------------------------------------------------------------------------------|
| encoding    | 디코딩될 문자열 값을 해석하는 데 사용할 인코딩. 기본 값은 'utf-8'이다.                                                                     |
| strict      | 상수(탈출시키지 않은) 줄바꿈 문자가 JSON 문자열에 나타나는 것을 허락할지를 결정하는 불리언 플래그. 기본 값은 True이며 이런 상황에서 예외를 발생시킨다.                       |
| cls         | 디코딩에 사용할 JSONDecoder의 하위 클래스. JSONDecoder를 상속하여 나름의 디코더를 만들었다면 명시해야 한다. load()에 주어진 키워드 인수가 더 있으면 클래스 생성자에 전달된다. |
| object_hook | 디코딩되는 각 결과 객체를 인수로 주며 호출할 함수. 기본으로 내장 dict( ) 함수가 사용된다.                                                          |
| parse_float | JSON 부동 소수점 값을 디코딩하기 위해 호출하는 함수. 기본 값은 내장 float( )함수이다.                                                          |

|                             |                                                                |
|-----------------------------|----------------------------------------------------------------|
| <code>parse_int</code>      | JSON 정수 값을 디코딩하기 위해 호출하는 함수. 기본 값은 내장 <code>int()</code> 함수이다. |
| <code>parse_constant</code> | 'NaN', 'true', 'false' 같은 JSON 상수를 디코딩하기 위해 호출하는 함수            |

**Loads(s, \*\*opts)**

문자열 s에서 객체를 역직렬화하는 점을 제외하고 `load()`와 동일하다.

비록 이 함수들이 `pickle`이나 `marshal` 모듈에 있는 함수들과 이름이 같고 또 데이터를 직렬화하는 데 이 함수들을 사용하기는 하지만 사용법이 같지는 않다. 구체적으로 말하자면 같은 파일에 둘 이상의 JSON으로 인코딩한 객체를 쓰는 데 `dump()`를 사용하면 안 된다. 비슷하게 같은 파일에서 둘 이상의 JSON으로 인코딩한 객체를 읽는 데 `load()`를 사용할 수 없다(입력 파일에 둘 이상의 객체가 있으면 에러가 발생한다). JSON으로 인코딩한 객체는 HTML이나 XML처럼 다루어야 한다. 예를 들어, 별개의 두 XML 문서를 가져와서 둘을 단순히 합치는 일은 없다.

인코딩 또는 디코딩 과정을 커스터마이즈하려면 다음 기반 클래스에서 상속받으면 된다.

**JSONDecoder(\*\*opts)**

JSON 데이터를 디코딩하는 클래스. opts는 키워드 인수들을 나타내며 `load()` 함수에서 사용되는 것과 같다. `JSONDecoder` 인스턴스 `d`는 다음 두 메서드를 가진다.

**d.decode(s)**

s에 있는 JSON 객체의 파이썬 표현을 반환한다. s는 문자열이다.

**d.raw\_decode(s)**

튜플 (`pyobj, index`)를 반환한다. 여기서 `pyobj`는 s에 있는 JSON 객체의 파이썬 표현이고 `index`는 s에서 JSON 객체의 끝 위치를 나타낸다. 끝에 데이터가 더 있는 입력 스트림에서 객체를 파싱하려고 할 때 사용할 수 있다.

**JSONEncoder(\*\*opts)**

파이썬 객체를 JSON으로 인코딩하는 클래스. opts는 키워드 인수들을 나타내며 `dump()` 함수에서 사용되는 것과 같다. `JSONEncoder` 인스턴스 `e`는 다음 두 메서드를 가진다.

**e.default(obj)**

파이썬 객체 obj를 일반 인코딩 규칙에 따라 인코딩할 수 없을 때 이 메서드가 호출된다. 이 메서드는 인코딩할 수 있는 타입 중 하나(예를 들어 문자열, 리스트나 사전)인 결과를 반환해야 한다.

**e.encode(obj)**

파이썬 객체 obj의 JSON 표현을 생성하기 위해 호출되는 메서드.

**e.iterencode(obj)**

파이썬 객체 obj의 JSON 표현을 나타내는 문자열들을 생성하는 반복자를 만든다. JSON 문자열을 생성하는 과정은 그 특성상 매우 재귀적이다. 예를 들어, 사전의 키들을 반복하게 되는데 그동안 발견하게 되는 다른 사전이나 리스트를 다시 방문하는 식이다. 이 메서드를 사용하면 모든 것을 메모리에서 큰 공간을 차지하는 문자열에 담을 필요 없이 출력을 조금씩 처리할 수 있게 된다.

JSONDecoder 또는 JSONEncoder에서 상속받아 하위 클래스를 정의할 때 `__init__( )`도 정의하는 경우라면 주의가 필요하다. 다음 코드는 모든 키워드 인수를 잘 처리하기 위해서 `__init__( )`를 어떻게 정의해야 하는지를 보여준다.

```
class MyJSONDecoder(JSONDecoder):
 def __init__(self, **kwargs):
 # 나만의 인수들을 가져온다.
 foo = kwargs.pop('foo', None)
 bar = kwargs.pop('bar', None)
 # 남은 것으로 부모를 초기화한다.
 JSONDecoder.__init__(self, **kwargs)
```

## mimetypes

mimetypes 모듈은 파일 확장자를 바탕으로 파일의 MIME 타입을 알아내는 데 사용한다. 또한 MIME 타입을 대응하는 표준 파일 확장자로 변환하는 데 사용할 수도 있다. MIME 타입은 ‘text/html’, ‘image/png’, ‘audio/mpeg’처럼 타입/하위 타입 쌍으로 구성된다.

**guess\_type(filename [, strict])**

파일 이름 또는 URL을 바탕으로 파일의 MIME 타입을 추측한다. 튜플 (type, encoding)을 반환하며, type은 “타입/하위 타입” 형태의 문자열이고 encoding은 전송을 위해 데이터를 인코딩하는 데 사용되는 프로그램이다(예를 들어, compress

또는 gzip). 타입을 추측할 수 없는 경우 (None, None)을 반환한다. strict가 True(기본 값)이면 IANA에 등록된 공식 MIME 타입만 인식한다(<http://www.iana.org/assignments/media-types>를 참고한다). 그렇지 않을 경우 비공식적인 MIME 타입이지만 흔히 사용되는 몇몇 타입도 인식한다.

#### **guess\_extension(type [, strict])**

MIME 타입을 바탕으로 표준 파일 확장자를 추측한다. 앞쪽 점(.)까지 포함한 파일 확장자를 담은 문자열을 반환한다. 모르는 타입인 경우 None을 반환한다. strict 가 True(기본 값)이면 공식 MIME 타입만 인식한다.

#### **guess\_all\_extensions(type [, strict])**

guess\_extension()과 동일하지만 가능한 모든 파일 확장자의 목록을 반환한다.

#### **init([files])**

모듈을 초기화한다. files은 타입 정보가 들어 있는 파일 이름들의 순서열이다. 이 파일들은 다음과 같이 MIME 타입을 허용 파일 확장자 목록으로 매핑하는 줄들을 담는다.

```
image/jpeg: jpe jpeg jpg
text/html: htm html
...
```

#### **read\_mime\_types(filename)**

주어진 filename에서 타입 매핑 정보를 읽는다. 파일 확장자를 MIME 타입 문자열로 매핑하는 사전을 반환한다. filename이 존재하지 않거나 읽을 수 없는 경우 None을 반환한다.

#### **add\_type(type, ext [, strict])**

새로운 MIME 타입을 맵핑에 추가한다. type은 ‘text/plain’ 같은 MIME 타입이고 ext는 ‘.txt’ 같은 파일 확장자이며 strict는 타입이 공식적으로 등록된 MIME 타입인지를 나타내는 불리언 값이다. 기본 값으로 strict은 True이다.

## **quopri**

quopri 모듈은 바이트 문자열에 대해 따옴표 처리된 출력 가능한(quoted-printable) 전송 인코딩 및 디코딩을 수행한다. 이 형식은 대부분 ASCII로서 읽을 수 있지만 소

수의 비출력 문자나 특수문자(예를 들어, 제어 문자나 128-255 범위에 있는 ASCII 아닌 문자)를 담은 8비트 텍스트 파일을 인코딩하는 데 주로 사용한다. 다음 규칙은 따옴표 처리된 출력 가능한 인코딩의 작동 방식을 보여준다.

- ‘=’를 제외한 공백 아닌 출력 가능한 ASCII 문자는 그대로 나타난다.
- ‘=’ 문자는 탈출 문자로 사용한다. 두 개의 16진수가 뒤에 나오면 해당 값을 갖는 문자를 나타낸다(예를 들어, ‘=0C’). 등호는 ‘=3D’로 표현한다. ‘=’가 줄 끝에 나타나면 소프트 줄바꿈(soft line break)을 표시한다. 긴 줄의 입력 텍스트가 여러 줄로 나누어져야 할 때 사용한다.
- 스페이스와 탭은 그대로 남지만 줄 끝에서는 없어질 수도 있다.

문서에서 확장된 ASCII 문자 집합에 있는 특수 문자를 사용하는 경우에 이 형식을 자주 사용한다. 예를 들어, 문서에 텍스트 “Copyright © 2009”가 들어 있을 때 파일 바이트 문자열 b‘Copyright \xa9 2009’로 나타낼 수 있다. 문자열을 따옴표 처리한 버전은 ‘\xa9’를 탈출 순서열 ‘=A9’로 바꾼 b‘Copyright =A9 2009’가 된다.

#### **decode(input, output [, header])**

바이트들을 quopri 형식으로 디코딩한다. input과 output은 이진 모드로 열린 파일 객체이다. header가 True이면 밑줄(\_)을 스페이스로 해석한다. 아니면 그대로 둔다. 이것은 인코딩된 MIME 헤더들을 디코딩할 때 사용한다. 기본 값으로 header는 False이다.

#### **decodestring(s [, header])**

문자열 s를 디코딩한다. s는 유니코드 또는 바이트 문자열이고 결과는 항상 바이트 문자열이다. header는 decode()에서와 의미가 같다.

#### **encode(input, output, quotetabs [, header])**

바이트들을 quopri 형식으로 인코딩한다. input과 output은 이진 모드로 열린 파일 객체이다. quotetabs를 True로 설정하면 보통 따옴표 처리 규칙에 더해 탭 문자를 따옴표 처리한다. 아니면 탭을 그대로 둔다. quotetabs의 기본 값은 False이다. header는 decode()에서와 의미가 같다.

#### **encodestring(s [, quotetabs [, header]])**

바이트 문자열 s를 인코딩한다. 결과 또한 바이트 문자열이다. quotetabs과

header는 encode( )에서와 의미가 같다.

### Note

따옴표 처리된 출력 가능한 데이터 인코딩은 유니코드 전에 나온 것으로 8비트 데이터에만 사용할 수 있다. 보통 텍스트에 대해서 사용하지만 실제로는 단일 바이트로 표현할 수 있는 ASCII와 확장 ASCII 문자 집합에 대해서만 쓸 수 있다. 이 모듈을 사용할 때는 모든 파일이 이진 모드로 열린다는 점과 바이트 문자열을 사용해야 한다는 점을 잊지 말아야 한다.

## xml 패키지

파이썬에는 XML 데이터 처리를 위한 다양한 모듈이 있다. XML 처리라는 주제는 매우 방대한 것이라서 이 책에서 모두 자세히 다루기는 어렵다. 이 절에서는 여러분이 XML 기본 개념에 익숙하다고 가정하고 설명을 해나갈 것이다. 스티브 홀즈너(Steve Holzner)가 쓴 《Inside XML》(New Riders 출판사)과 엘리엇 헤롤드(Elliott Harld)와 W. 스콧 민즈(W. Scott Means)가 쓴 《XML in a Nutshell》(O'Reilly and Associates 출판사) 같은 책을 보면 기본 XML 개념을 이해하는 데 도움이 될 것이다. 크리스토퍼 존스(Christopher Jones)가 쓴 《Python & XML》(O'Reilly and Associates 출판사)와 선 맥그래스(Sean McGrath)가 쓴 《XML Processing with Python》(Prentice Hall 출판사) 같은 몇몇 책에서는 파이썬으로 XML을 처리하는 방법을 다룬다.

파이썬은 XML을 두 가지 형태로 지원한다. 첫 번째로 XML 파싱을 위한 업계 표준 방식인 SAX와 DOM을 기본으로 지원한다. SAX(Simple API for XML)는 이벤트 처리를 기반으로 하는 것으로서 XML 문서를 순차적으로 읽으며 XML 엘리먼트를 만날 때마다 처리기 함수를 구동한다. DOM(Document Object Model)은 전체 XML 문서를 표현하는 트리 구조를 생성한다. 일단 트리가 생성되고 나면 DOM 인터페이스로 트리를 탐색하고 원하는 데이터를 추출할 수 있다. SAX나 DOM API 모두 파이썬에서 나온 것이 아니다. 파이썬은 간단히 자바와 자바스크립트용으로 개발된 표준 프로그래밍 인터페이스를 따르고 있다.

SAX와 DOM 인터페이스를 사용하여 XML을 처리할 수 있지만 표준 라이브러리에 있는 가장 편한 프로그래밍 인터페이스는 ElementTree 인터페이스이다. 이것 은 XML을 파싱할 때 파이썬 언어의 장점을 최대한 활용하는 파이썬에 특화된 접

근 방식으로서 대부분의 사용자들이 SAX나 DOM보다 훨씬 쉽고 빠르다고 느끼는 방식이다. 이 절에서는 세 가지 XML 파싱 방식을 모두 다룰 테지만 그중에서도 ElementTree 방식을 가장 자세하게 다룬다.

이 장에서는 기본적인 XML 데이터 파싱 방법에 관해서만 초점을 맞출 것이다. 파이썬에는 새로운 종류의 파서를 구현한다거나 아무것도 없는 것에서 XML 문서를 생성하는 것과 관련된 XML 모듈도 있다. 또한 XSLT이나 XPATH 지원 같은 추가 XML 기능을 지원하는 다양한 써드 파티 확장 모듈이 존재한다. 여기에 관한 더 많은 정보는 <http://wiki.python.org/moin/PythonXml>에서 찾을 수 있다.

## XML 문서 예

다음 예는 요리법을 담은 전형적인 XML 문서를 보여준다.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe>
 <title>
 Famous Guacamole
 </title>
 <description>
 A southwest favorite!
 </description>
 <ingredients>
 <item num="4"> Large avocados, chopped </item>
 <item num="1"> Tomato, chopped </item>
 <item num="1/2" units="C"> White onion, chopped </item>
 <item num="2" units="tbl"> Fresh squeezed lemon juice </item>
 <item num="1"> Jalapeno pepper, diced </item>
 <item num="1" units="tbl"> Fresh cilantro, minced </item>
 <item num="1" units="tbl"> Garlic, minced </item>
 <item num="3" units="tsp"> Salt </item>
 <item num="12" units="bottles"> Ice-cold beer </item>
 </ingredients>
 <directions>
 Combine all ingredients and hand whisk to desired consistency.
 Serve and enjoy with ice-cold beers.
 </directions>
</recipe>
```

문서는 <tilte>...</title> 같이 태그로 시작하고 태그로 끝나는 엘리먼트(element)들로 구성된다. 엘리먼트는 보통 중첩되고 조직화되어 계층 구조를 이룬다. 예를 들어, <item> 엘리먼트는 <ingredients> 아래에 나타난다. 각 문서에서 문서의 루트인 엘리먼트가 하나 있다. 앞의 예에서 <recipe> 엘리먼트가 문서 루트이다. item 엘리먼트 <item num="4">Large avocados, chopped</item>에서 볼 수 있듯이 엘리먼

트는 추가로 어트리뷰트(attribute)를 가질 수 있다.

XML 문서를 다룰 때는 이러한 기본 특성을 잘 이해해야 한다. 예를 들어, 특정 엘리먼트에서 텍스트나 어트리뷰트를 추출하려고 한다고 하자. 해당 엘리먼트의 위치를 알아내려면 루트 엘리먼트에서 시작해서 문서 계층 구조를 순회해야 한다.

## **xml.dom.minidom**

xml.dom.minidom 모듈은 XML 문서를 파싱하여 DOM 규약에 따라 메모리에 트리 구조로 저장하는 기능을 제공한다. 이 모듈에는 두 개의 파싱 함수가 있다.

### **parse(file [, parser])**

file의 내용을 파싱하여 문서 트리의 제일 위를 나타내는 노드를 반환한다. file은 파일 이름이거나 이미 열린 파일 객체일 수 있다. parser는 트리를 생성하는 데 사용할 추가 SAX2 호환 파서 객체이다. 생략하면 기본 파서를 사용한다.

### **parseString(string [, parser])**

파일 대신 문자열로 입력 데이터를 주는 것 말고는 parse()와 동일하다.

## Nodes

파싱 함수가 반환하는 문서 트리는 서로 연결된 노드 집합으로 구성된다. 각 노드 n은 정보를 얻거나 트리 구조를 탐색하는 데 사용할 수 있는 다음 속성들을 가진다.

노드 어트리뷰트	설명
n.attributes	어트리뷰트 값들을 담은 매핑 객체(있다면)
n.childNodes	n의 모든 자식 노드 목록
n.firstChild	노드 n의 첫 번째 자식
n.lastChild	노드 n의 마지막 자식
n.localName	엘리먼트의 지역 태그 이름. 태그에 콜론이 있으면(예를 들어, '<foobar ...>') 콜론 이후 부분만 담는다.
n.namespaceURI	있는 경우 n에 연결된 네임스페이스
n.nextSibling	트리에서 n 다음에 나타나는 부모가 같은 노드. n이 마지막 자식이면 None.
n.nodeName	노드 이름. 노드 타입에 따라 의미가 다르다.
n.nodeType	노드 타입을 나타내는 정수. Node 클래스의 클래스 변수인 다음 값 중 하나로 설정된다. ATTRIBUTE_NODE, CDATA_SECTION_NODE, COMMENT_NODE, DOCUMENT_FRAGMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, ELEMENT_NODE, ENTITY_NODE,

	ENTITY_REFERENCE_NODE, NOTATION_NODE, PROCESSING_INSTRUCTION_NODE, TEXT_NODE.
n.nodeValue	노드 값. 의미는 노드 타입에 따라 다르다.
n.parentNode	부모 노드에 대한 참조
n.prefix	태그 이름의 콜론 앞쪽 부분. 예를 들어, 엘리먼트 ‘<foobar ...>’는 접두사 ‘foo’를 갖는다.
n.previousSibling	트리에서 n 앞에 나타나는 부모가 같은 노드.

이러한 속성들에 대해 모든 노드는 다음 메서드들을 갖는다. 보통 이들은 트리 구조를 다루는 데 사용한다.

#### **n.appendChild(child)**

새로운 자식 노드 child를 n에 추가한다. 새로운 자식은 다른 자식들 끝에 추가된다.

#### **n.cloneNode(deep)**

노드 n의 복사본을 생성한다. deep이 True이면 모든 자식 노드가 복제된다.

#### **n.hasAttributes()**

노드가 어트리뷰트를 가지고 있으면 True를 반환한다.

#### **n.hasChildNodes()**

노드가 자식을 가지고 있으면 True를 반환한다.

#### **n.insertBefore(newchild, ichild)**

새로운 자식 newchild를 자식 ichild 앞에 추가한다. ichild는 반드시 n의 자식이어야 한다.

#### **n.isSameNode(other)**

노드 other가 n과 동일한 DOM 노드를 참조할 때 True를 반환한다.

#### **n.normalize()**

이웃 텍스트 노드들을 하나의 텍스트 노드로 합친다.

#### **n.removeChild(child)**

n에서 자식 child를 제거한다.

#### **n.replaceChild(newchild, oldchild)**

oldchild를 newchild로 대체한다. oldchild는 반드시 n의 자식이어야 한다.

트리에 나타날 수 있는 노드 타입은 매우 다양하지만 Document, Element, Text 노드가 주로 나타난다. 지금부터 각각에 대해서 간단하게 설명한다.

### Document 노드

Document 노드는 전체 문서 트리의 가장 위에 나타나며 전체 문서를 총제적으로 표현한다. 다음 메서드와 속성들을 갖는다.

#### **d.documentElement**

전체 문서의 루트 엘리먼트.

#### **d.getElementsByTagName(tagname)**

모든 자식 노드를 검색하여 주어진 태그 이름 tagname을 갖는 엘리먼트들의 리스트를 반환한다.

#### **d.getElementsByTagNameNS(namespaceuri, localname)**

모든 자식 노드를 검색하여 주어진 네임스페이스 URI와 지역 이름을 갖는 엘리먼트들의 목록을 반환한다. 반환되는 리스트는 타입이 NodeList인 객체이다.

### Element 노드

Element 노드 e는 ‘<foo>...</foo>’ 같은 하나의 엘리먼트를 나타낸다. 엘리먼트에서 텍스트를 가져오려면 자식 Text 노드를 찾아야 한다. 기타 다른 정보를 가져오기 위해 다음 속성과 메서드들은 사용할 수 있다.

#### **e.tagName**

엘리먼트의 태그 이름. 예를 들어, 엘리먼트가 ‘<foo ...>’로 정의되면 태그 이름은 ‘foo’이다.

#### **e.getElementsByTagName(tagname)**

주어진 태그 이름을 갖는 모든 자식들의 리스트를 반환한다.

#### **e.getElementsByTagNameNS(namespaceuri, localname)**

주어진 네임스페이스에 있는 주어진 태그 이름을 갖는 모든 자식들의 리스트를 반환한다. namespaceuri와 localname은 네임스페이스와 태그 이름을 지정하는 문자열이다. 네임스페이스가 ‘<foo xmlns:foo=“http://www.spam.com/foo”>’처럼

선언되어 있다면 namespaceuri를 ‘<http://www.spam.com/foo>’로 설정하면 된다. 엘리먼트 ‘`<foo:bar>`’를 찾는다면 localname을 ‘bar’로 설정하면 된다. 반환되는 객체는 NodeList 타입이다.

#### **e.hasAttribute(name)**

엘리먼트가 이름이 name인 어트리뷰트를 가지고 있다면 True를 반환한다.

#### **e.getAttributeNS(namespaceuri, localname)**

엘리먼트가 namespaceuri와 localname인 이름을 갖는 어트리뷰트를 가지고 있으면 True를 반환한다. 인수들의 의미는 `getElementsByTagNameNS()`에서 의미와 같다.

#### **e.getAttribute(name)**

어트리뷰트 name의 값을 반환한다. 반환되는 값은 문자열이다. 어트리뷰트가 존재하지 않으면 빈 문자열을 반환한다.

#### **e.getAttributeNS(namespaceuri, localname)**

namespaceuri와 localname인 이름을 가지는 속성의 값을 반환한다. 반환되는 값은 문자열이다. 어트리뷰트가 존재하지 않으면 빈 문자열을 반환한다. 인수들은 `getElementsByTagName()`에서 설명한 의미를 가진다.

## Text 노드

Text 노드는 텍스트 데이터를 표현하는 데 사용한다. 텍스트 데이터는 Text 객체 t의 어트리뷰트 t.data에 저장된다. 주어진 문서 엘리먼트에 연결된 텍스트는 항상 해당 엘리먼트의 자식 Text 노드들로 저장된다.

## 유ти리티 함수

다음 유ти리티 메서드들이 노드에 정의되어 있다. DOM 표준의 일부는 아니지만 사용상의 편의와 디버깅을 위해 파이썬에서 제공한다.

#### **n.toprettyxml([indent [, newl]])**

노드 n과 자식들로 표현되는 XML 문서를 담은 깔끔하게 포맷이 지정된 문자열을 생성한다. indent는 들여쓰기 문자열을 지정하며 기본 값은 탭(‘\t’)이다. newl은 줄바꿈 문자를 나타내며 기본 값은 ‘\n’이다.

**n.toxml([encoding])**

노드 n과 자식들로 표현되는 XML 문서를 담은 문자열을 생성한다. encoding은 인코딩을 지정한다(예를 들어, ‘utf-8’). 인코딩이 주어지지 않으면 출력 텍스트에 아무것도 표시되지 않는다.

**n.writexml(writer [, indent [, addindent [, newl]]])**

XML을 writer에 쓴다. writer는 파일 인터페이스와 호환되는 write() 메서드를 제공하는 아무 객체나 될 수 있다. indent는 n의 들여쓰기를 지정한다. 즉, 출력할 때 노드 n의 시작 부분에 붙는 문자열이다. addindent는 n의 자식 노드에 적용되는 점증적인 들여쓰기를 지정하는 문자열이다. newl은 줄바꿈 문자를 지정한다.

**DOM 예**

다음은 xml.dom.minidom 모듈을 사용하여 XML 파일을 파싱하여 정보를 추출하는 방법을 보여주는 예이다.

```
from xml.dom import minidom
doc = minidom.parse("recipe.xml")

ingredients = doc.getElementsByTagName("ingredients")[0]
items = ingredients.getElementsByTagName("item")

for item in items:
 num = item.getAttribute("num")
 units = item.getAttribute("units")
 text = item.firstChild.data.strip()
 quantity = "%s %s" % (num,units)
 print("%-10s %s" % (quantity, text))
```

**Note**

xml.dom.minidom 모듈은 파스 트리(parse tree)를 변경하거나 다른 종류의 XML 노드를 다루는 데 사용할 수 있는 많은 추가 기능을 제공한다. 여기에 관한 자세한 정보는 온라인 문서에서 찾을 수 있다.

**xml.etree.ElementTree**

xml.etree.ElementTree 모듈은 계층적인 데이터를 저장하고 조작하는 데 사용할 수 있는 유연성 있는 컨테이너인 ElementTree 객체를 정의한다. 이 객체는 XML 처

리에 주로 사용하지만 실제로는 리스트와 사전을 아우르는 범용적인 목적으로 사용할 수 있다.

### ElementTree 객체

다음 클래스는 새로운 ElementTree 객체를 정의하고 계층 구조의 가장 위 수준을 표현하는 데 사용된다.

#### **ElementTree([element [, file]])**

새로운 ElementTree 객체를 생성한다. element는 트리의 루트 노드를 나타내는 인스턴스이다. 이 인스턴스는 곧 설명할 엘리먼트 인터페이스를 지원한다. file은 트리를 생성하기 위해 XML 데이터를 읽을 파일 이름 또는 파일과 유사한 객체이다.

ElementTree의 인스턴스 tree는 다음 메서드들을 가진다.

#### **tree.\_setroot(element)**

element를 루트 엘리먼트로 설정한다.

#### **tree.find(path)**

주어진 경로에 맞는 태입을 갖는 트리에서 가장 높은 수준에 있는 엘리먼트를 찾아서 반환한다. path는 엘리먼트 태입과 다른 엘리먼트에 상대적인 해당 엘리먼트의 위치를 나타내는 문자열이다. 다음 목록은 경로 문법을 설명한다.

경로	설명
'tag'	주어진 태그를 갖는 최상위 수준 엘리먼트에만 매칭된다. 예를 들어, <tag>...</tag>. 더 하위 수준에서 정의된 엘리먼트와는 매칭되지 않는다. <foo><tag>...</tag></foo>처럼 다른 엘리먼트에 포함되는 태입이 tag인 엘리먼트에도 매칭되지 않는다.
'parent/tag'	태그가 'parent'인 엘리먼트의 자식일 경우에만 태그가 'tag'인 엘리먼트에 매칭된다. 원하는 만큼 경로를 길게 써도 된다.
'*'	모든 자식 엘리먼트를 선택한다. 예를 들어, '*/*'는 'tag'라는 태그 이름을 가진 모든 손자 엘리먼트에 매칭된다.
'.'	현재 노드에서 검색을 시작한다.
'//'	엘리먼트 아래에 있는 모든 수준에 있는 하위 엘리먼트를 선택한다. 예를 들어, '//tag'는 어느 하위 수준에서든 'tag' 태그를 가진 모든 엘리먼트에 매칭된다.

XML 네임스페이스를 사용하는 문서를 다룰 때는 경로에 나타나는 tag 문자열이 '{uri}tag' 형식이어야 한다. 여기서 uri은 'http://www.w3.org/TR/html4/' 같은 문

자열이다.

#### **tree.findall(path)**

주어진 경로에 일치하는 모든 최상위 수준 엘리먼트를 트리에서 찾아서 문서에 나타난 순서에 따라 리스트나 반복자 형태로 반환한다.

#### **tree.findtext(path [, default])**

트리에서 주어진 경로에 일치하는 첫 번째 최상위 수준 엘리먼트의 텍스트를 반환한다. default는 매칭된 엘리먼트를 발견하지 못하였을 경우 반환할 문자열이다.

#### **tree.getiterator([tag])**

태그가 tag인 트리의 모든 엘리먼트를 구역(section) 순서대로 생성하는 반복자를 반환한다. tag를 생략하면 트리의 모든 엘리먼트를 차례대로 반환한다.

#### **tree.getroot()**

트리의 루트 엘리먼트를 반환한다.

#### **tree.parse(source [, parser])**

외부 XML 데이터를 파싱한 결과로 루트 엘리먼트를 대체한다. source는 XML 파일 이름 또는 파일과 유사한 객체이다. parser는 TreeBuilder 인스턴스로 나중에 설명한다.

#### **tree.write(file [, encoding])**

트리의 모든 내용을 파일에 쓴다. file은 파일 이름이거나 쓰기용으로 열 파일과 유사한 객체일 수 있다. encoding은 사용할 출력 인코딩이며 지정하지 않을 경우 인터프리터의 기본 인코딩을 사용한다(대부분 ‘utf-8’ 또는 ‘ascii’이다).

### 엘리먼트 생성

ElementTree에 있는 엘리먼트의 타입은 다양한 인스턴스로 표현되는데 이들은 파일을 파싱하여 생성하거나 다음 생성 함수들을 사용해서 생성할 수 있다.

#### **Comment([text])**

새로운 주석 엘리먼트를 생성한다. text는 엘리먼트 텍스트를 담은 문자열 또는 바이트 문자열이다. 이 엘리먼트는 파싱하거나 출력을 내보낼 때 XML 주석에 매핑된다.

**Element(tag [, attrib [, \*\*extra]])**

새로운 엘리먼트를 생성한다. tag는 엘리먼트 이름이다. 예를 들어, 엘리먼트 '`<foo>....</foo>`'를 생성하였다면 tag는 'foo'가 된다. attrib는 문자열 또는 바이트 문자열로 표현되는 어트리뷰트들을 담은 사전이다. extra로 제공하는 추가 키워드 인수들도 엘리먼트의 어트리뷰트들을 설정하는 데 사용된다.

**fromstring(text)**

text에 있는 XML 텍스트 조각에서 엘리먼트를 생성한다. 곧 설명할 XML()과 동일하다.

**ProcessingInstruction(target [, text])**

처리 명령에 대응하는 새로운 엘리먼트를 생성한다. target과 text는 모두 문자열 또는 바이트 문자열이다. XML에서는 '`<? target text? >`'에 해당한다.

**SubElement(parent, tag [, attrib [, \*\*extra]])**

Element()와 동일하지만 새 엘리먼트를 자동으로 parent로 지정한 엘리먼트의 자식으로 추가한다.

**XML(text)**

text에 있는 XML 코드 조각을 파싱하여 엘리먼트를 생성한다. 예를 들어, text를 '`<foo>....</foo>`'로 설정하면 'foo' 태그를 가진 표준 엘리먼트를 생성한다.

**XMLID(text)**

'id' 어트리뷰트들을 수집하여 ID 값을 엘리먼트로 매핑하는 사전을 구축하는 점을 제외하고는 XML(text)와 동일하다. 튜플 (elem, idmap)을 반환한다. 여기서 elem은 새로운 엘리먼트이고 idmap은 ID 매핑 사전이다. 예를 들어, XMLID('foo id="123"<bar id="456">Hello</bar></foo>')는 ((Element foo), {'123': Element foo, '456': Element bar})을 반환한다.

## 엘리먼트 인터페이스

ElementTree에 저장된 엘리먼트는 다양한 타입을 가질 수 있지만 모두 공통 인터페이스를 지원한다. elem이 엘리먼트일 때 다음 파이썬 연산자들이 정의된다.

연산자	설명
-----	----

elem[n]	elem의 n번째 자식 엘리먼트를 반환한다.
---------	--------------------------

<code>elem[n] = newelem</code>	elem의 n번째 자식 엘리먼트를 엘리먼트 newelem로 변경한다.
<code>del elem[n]</code>	elem의 n번째 엘리먼트를 삭제한다.
<code>len(elem)</code>	elem의 자식 엘리먼트 개수

모든 엘리먼트는 다음의 기본 데이터 속성들을 가진다.

속성	설명
<code>elem.tag</code>	엘리먼트 태입을 나타내는 문자열. 예를 들어, <foo>...</foo>는 ‘foo’ 태그를 가진다.
<code>elem.text</code>	엘리먼트에 연결된 데이터. 보통 XML 엘리먼트의 시작과 끝 태그 사이에 있는 텍스트를 담은 문자열이다.
<code>elem.tail</code>	엘리먼트와 함께 저장된 추가 데이터. XML에서는 일반적으로 엘리먼트 끝 태그 다음부터 다음 태그가 시작되기 전 사이에 있는 공백을 담은 문자열이 된다.
<code>elem.attrib</code>	엘리먼트의 어트리뷰트들을 담은 사전

엘리먼트는 다음 메서드들을 지원한다. 이 중에 몇 가지는 사전의 메서드를 흡내낸다.

#### **elem.append(subelement)**

엘리먼트 subelement를 자식 리스트에 추가한다.

#### **elem.clear()**

엘리먼트의 어트리뷰트, 텍스트, 자식 등 모든 데이터를 지운다.

#### **elem.find(path)**

path와 일치하는 태입을 갖는 첫 번째 하위 엘리먼트를 찾는다.

#### **elem.findall(path)**

path와 일치하는 태입을 갖는 모든 하위 엘리먼트를 찾는다. 매칭된 엘리먼트들을 문서 순서대로 리스트나 반복 가능한 객체로 반환한다.

#### **elem.findtext(path [, default])**

path와 일치하는 태입을 갖는 첫 번째 엘리먼트의 텍스트를 찾는다. default는 찾지 못한 경우 반환할 문자열이다.

#### **elem.get(key [, default])**

어트리뷰트 key의 값을 가져온다. default는 어트리뷰트가 존재하지 않는 경우 반환할 기본 값이다. XML 네임스페이스가 있는 경우 key는 ‘uri:key’ 형식의 문자열이 되어야 한다. 여기서 uri는 ‘http://www.w3.org/TR/html4/’ 같은 문자열이다.

**elem.getchildren()**

모든 하위 엘리먼트를 문서 순서대로 반환한다.

**elem.getiterator([tag])**

tag에 매칭되는 태입을 갖는 모든 하위 엘리먼트를 생성하는 반복자를 반환한다.

**elem.insert(index, subelement)**

자식 리스트에서 index 위치에 하위 엘리먼트를 삽입한다.

**elem.items()**

모든 어트리뷰트를 (name, value) 쌍들의 리스트로 반환한다.

**elem.keys()**

모든 어트리뷰트 이름을 담은 리스트를 반환한다.

**elem.remove(subelement)**

자식 리스트에서 엘리먼트 subelement를 제거한다.

**elem.set(key, value)**

어트리뷰트 key에 값 value를 설정한다.

## 트리 구축

ElementTree 객체는 트리와 유사한 구조로부터 쉽게 생성할 수 있다. 다음 객체를 사용하면 된다.

**TreeBuilder([element\_factory])**

파일을 파싱하거나 트리 구조를 순회하는 도중 일련의 start( ), end( ), data( ) 호출을 통해 ElementTree 구조를 생성하는 클래스. element\_factory는 새로운 엘리먼트 인스턴스를 생성하기 위해 호출되는 함수다.

TreeBuilder 인스턴스 t는 다음 메서드들을 가진다.

**t.close()**

트리 생성을 종료하고 생성된 최상위 수준의 ElementTree 객체를 반환한다.

**t.data(data)**

처리 중인 현재 엘리먼트에 텍스트 데이터를 추가한다.

**t.end(tag)**

처리 중인 현재 엘리먼트를 닫고 최종 엘리먼트 객체를 반환한다.

**t.start(tag, attrs)**

새로운 엘리먼트를 생성한다. tag는 엘리먼트 이름이고 attrs은 어트리뷰트 값들을 담은 사전이다.

## 유ти리티 함수

이 모듈에는 다음 유ти리티 함수들이 있다.

**dump(elem)**

elem의 구조를 디버깅을 위해 sys.stdout에 쓴다. 보통 XML로 출력된다.

**iselement(elem)**

elem이 유효한 엘리먼트 객체인지를 확인한다.

**iterparse(source [, events])**

source에서 점진적으로 XML을 파싱한다. source는 XML 데이터를 가리키는 파일 이름이거나 파일과 유사한 객체이다. events는 생성할 이벤트 타입 리스트이다. 가능한 이벤트 타입은 ‘start’, ‘end’, ‘start-ns’, ‘end-ns’이다. 생략하면 ‘end’ 이벤트만 생성된다. 이 함수는 튜플 (event, elem)을 생성하는 반복자를 반환한다. 여기서 event는 ‘start’ 또는 ‘end’ 같은 문자열이고 elem은 처리할 엘리먼트이다. ‘start’ 이벤트에서는 엘리먼트가 새롭게 생성되고 어트리뷰트만 채워진다. ‘end’ 이벤트에서는 엘리먼트가 완전히 채워져서 모든 하위 엘리먼트를 담게 된다.

**parse(source)**

XML 소스를 ElementTree 객체로 파싱한다. source는 XML 데이터를 담은 파일 이름 또는 파일과 유사한 객체이다.

**tostring(elem)**

elem와 모든 하위 엘리먼트를 표현하는 XML 문자열을 생성한다.

## XML 예

다음은 ElementTree로 견본 요리법 파일을 파싱하여 필요한 재료 목록을 출력하

는 예를 보여준다. DOM 관련 예와 비슷하다.

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
ingredients = doc.findall('ingredients')

for item in ingredients.findall('item'):
 num = item.get('num')
 units = item.get('units', '')
 text = item.text.strip()
 quantity = "%s %s" % (num, units)
 print("%-10s %s" % (quantity, text))
```

ElementTree의 경로 문법을 사용하면 특정 작업을 간단하게 수행할 수 있고 필요할 때 원하는 정보를 손쉽게 얻을 수 있다. 예를 들어, 다음은 앞에 나온 코드를 경로 문법을 사용해 간단히 모든 <item>...</item> 엘리먼트를 추출하는 코드로 변환한 것이다.

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
for item in doc.findall("./item"):
 num = item.get('num')
 units = item.get('units', '')
 text = item.text.strip()
 quantity = "%s %s" % (num, units)
 print("%-10s %s" % (quantity, text))
```

이제 네임스페이스가 있는 다음 XML 파일 ‘recipens.xml’을 살펴보자.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe xmlns:r="http://www.dabeaz.com/namespaces/recipe">
 <r:title>
 Famous Guacamole
 </r:title>
 <r:description>
 A southwest favorite!
 </r:description>
 <r:ingredients>
 <r:item num="4"> Large avocados, chopped </r:item>
 ...
 </r:ingredients>
 <r:directions>
 Combine all ingredients and hand whisk to desired consistency.
 Serve and enjoy with ice-cold beers.
 </r:directions>
</recipe>
```

네임스페이스를 처리할 때는 네임스페이스의 접두사를 연결된 네임스페이스 URI에 매핑하는 사전을 사용하는 것이 편리하다. 그리고 나서 다음처럼 문자열 포맷

지정 연산자를 URI를 채우는 데 사용하면 된다.

```
from xml.etree.ElementTree import ElementTree
doc = ElementTree(file="recipens.xml")
ns = {
 'r' : 'http://www.dabeaz.com/namespaces/recipe'
}
ingredients = doc.findall('{%(r)s}ingredients' % ns)
for item in ingredients.findall('{%(r)s}item' % ns):
 num = item.get('num')
 units = item.get('units')
 text = item.text.strip()
 quantity = "%s %s" % (num, units)
 print("%-10s %s" % (quantity, text))
```

작은 XML 파일은 ElementTree 모듈을 사용해서 빠르게 메모리로 읽어서 작업해도 된다. 하지만, 다음과 같은 구조를 가진 큰 XML 파일을 다루어야 한다고 하자.

```
<?xml version="1.0" encoding="utf-8"?>
<music>
 <album>
 <title>A Texas Funeral</title>
 <artist>Jon Wayne</artist>
 ...
 </album>
 <album>
 <title>Metaphysical Graffiti</title>
 <artist>The Dead Milkmen</artist>
 ...
 </album>
 ...
</music>
```

큰 XML 파일을 메모리로 읽으면 엄청난 양의 메모리를 소비할 수도 있다. 예를 들어, 10MB인 XML 파일을 메모리로 읽으면 100MB가 넘는 메모리 자료 구조가 필요할 수 있다. 이러한 파일에서 정보를 추출할 때는 ElementTree.iterparse( ) 함수를 사용하는 것이 가장 좋다. 다음은 앞에 나온 파일에서 <album> 노드를 반복적으로 처리하는 예를 보여준다.

```
from xml.etree.ElementTree import iterparse

iparse = iterparse("music.xml", ['start','end'])
최상위 music 엘리먼트를 찾는다.
for event, elem in iparse:
 if event == 'start' and elem.tag == 'music':
 musicNode = elem
 break

모든 앨범을 가져온다.
```

```

albums = (elem for event, elem in iparse
 if event == 'end' and elem.tag == 'album')

for album in albums:
 # 필요한 처리를 한다.
 ...
 musicNode.remove(album) # 끝났다면 앨범을 버린다.

```

`iterparse( )`를 효율적으로 사용하려면 더 이상 사용하지 않는 데이터를 지워야 한다. 마지막 문장 `musicNode.remove(album)`은 각 `<album>` 엘리먼트에 대한 처리가 끝나면 바로 버린다(부모 노드에서 제거한다). 이 코드의 메모리 사용량을 추적해보면 입력 파일이 매우 큰 경우에도 메모리를 적게 사용하는 것을 볼 수 있을 것이다.

### Note

- 파이썬에서 간단한 XML 문서를 다룰 때는 `ElementTree` 모듈을 사용하는 것이 가장 쉽고 유연성 있다. 하지만 이 모듈은 충분한 기능을 제공하지 않는다. 예를 들어, 유효성 검사 기능을 제공하지 않으며 DTD 같은 XML 문서의 복잡한 측면을 처리하는 방법을 제공하지 않는다. 따라서 써드 파티 패키지를 설치해야 할 것이다. 한 예로 `lxml.etree( http://codespeak.net/xml/에 있음)`는 인기 있는 `libxml2`와 `libxslt` 라이브러리에 대한 `ElementTree` API를 제공하고 XPATH, XSLT 및 기타 다양한 기능을 지원한다.
- `ElementTree` 모듈도 프레드리크 런드(Fredrik Lundh)가 <http://effbot.org/zone/element-index.htm>를 통해 관리하고 있는 써드 파티 패키지이다. 이 사이트에 가면 표준 라이브러리에 포함된 것보다 더 많은 기능을 제공하는 최신 버전을 받을 수 있다.

## xml.sax

`xml.sax` 모듈은 SAX2 API를 사용하여 XML 문서를 파싱할 수 있게 한다.

**parse(file, handler [, error\_handler])**

XML 문서 `file`을 파싱한다. `file`은 파일 이름 또는 열린 파일 객체이다. `handler`는 콘텐츠 처리기 객체이다. `error_handler`는 옵션인 SAX 에러 처리기 객체이며 자세한 내용은 온라인 문서를 찾아보기 바란다.

**parseString(string, handler [, error\_handler])**

`string`에 담긴 XML 데이터를 파싱한다는 점을 제외하고 `parse( )`와 동일하다.

## 처리기 객체

데이터를 처리하려면 parse( ) 또는 parseString( ) 함수에 콘텐츠 처리기 객체를 제공해야 한다. 처리기를 정의하려면 ContentHandler에서 상속받아서 클래스를 정의하면 된다. ContentHandler 인스턴스 c는 다음 메서드들을 가지며 사용자 처리기 클래스에서 필요에 따라 재정의할 수 있다.

### c.characters(content)

무가공 문자 데이터를 제공하기 위해 파서가 호출한다. content는 문자들을 담은 문자열이다.

### c.endDocument()

문서의 끝에 도달하였을 때 파서가 호출한다.

### c.endElement(name)

엘리먼트 name의 끝에 도달하였을 때 호출된다. 예를 들어, ‘</foo>’가 파싱되면 name을 ‘foo’로 설정해서 이 메서드가 호출된다.

### c.endElementNS(name, qname)

XML 네임스페이스가 있는 엘리먼트의 끝에 도달하였을 때 호출된다. name은 문자열들의 튜플 (uri, localname)이고 qname은 완전히 한정된(fully qualified) 이름이다. 보통 qname은 SAX namespace-prefixes 기능이 활성화되지 않은 한 None이다. 예를 들어, 엘리먼트가 ‘<foo:bar xmlns:foo="http://spam.com">’로 정의되어 있으면 name 튜플은 (u'http://spam.com', u'bar')가 된다.

### c.endPrefixMapping(prefix)

XML 네임스페이스의 끝에 도달하였을 때 호출된다. prefix는 네임스페이스 이름이다.

### c.ignorableWhitespace whitespace)

문서에서 무시할 수 있는 공백을 만났을 때 호출된다. whitespace는 공백을 담은 문자열이다.

### c.processingInstruction(target, data)

‘? ... ?’로 둘러싸인 XML 처리 명령을 만났을 때 호출된다. target은 명령 타입이고 data는 명령 데이터이다. 예를 들어, 명령이 ‘<?xmlstylesheet href="mystyle.css”

`type="text/css"?>`이면 target은 ‘xmlstylesheet’로 설정되고 data는 명령 텍스트의 남은 부분 ‘`href="mystyle.css" type="text/css"`’이 된다.

#### **c.setDocumentLocator(locator)**

줄 번호, 열 및 기타 정보를 추적하는 데 사용할 수 있는 locator 객체를 제공하기 위해 파서가 호출한다. 이 메서드에서는 주로 에러 메시지를 출력한다든지 하는 용도로 나중에 사용할 수 있게 locator를 어디가에 저장해둔다. locator에는 위치 정보를 얻는 데 사용할 수 있는 네 가지 메서드 `getColumnName()`, `getLineNumber()`, `getPublicId()`, `getSystemId()`가 있다.

#### **c.skippedEntity(name)**

파서가 엔티티(entity)를 건너뛸 때마다 호출된다. name은 건너뛴 엔티티의 이름이다.

#### **c.startDocument()**

문서가 시작될 때 호출된다.

#### **c.startElement(name, attrs)**

새로운 XML 엘리먼트를 만날 때마다 호출된다. name은 엘리먼트 이름이고 attrs는 어트리뷰트 정보를 담은 객체이다. 예를 들어, XML 엘리먼트가 ‘`<foo bar="whatever" spam="yes">`’이면 name은 ‘foo’로 설정되고 attrs는 bar와 spam 어트리뷰트에 대한 정보를 담는다. attrs 객체에는 어트리뷰트 정보를 얻는 데 사용할 수 있는 메서드 몇 개가 있다.

메서드	설명
<code>attrs.getLength()</code>	어트리뷰트 개수를 반환한다.
<code>attrs.getNames()</code>	어트리뷰트 이름 리스트를 반환한다.
<code>attrs.getType(name)</code>	어트리뷰트 name의 타입을 얻는다.
<code>attrs.getValue(name)</code>	어트리뷰트 name의 값을 얻는다.

#### **c.startElementNS(name, qname, attrs)**

XML 네임스페이스를 사용하는 새로운 XML 엘리먼트를 만났을 때 호출된다. name은 튜플 (`uri, localname`)이고 qname은 완전히 한정된 이름이다(보통 SAX2 namespace-prefixes 기능이 활성화되지 않은 경우 `None`으로 설정된다). attrs은 어트리뷰트 정보를 담은 객체이다. 예를 들어, XML 엘리먼트가 ‘`<foo:bar xmlns:foo="http://spam.com" blah="whatever">`’이면 name은 (`u'http://spam.com'`, `u'bar'`)

이고 qname은 None<sup>o</sup>이며 attrs는 어트리뷰트 blah에 대한 정보를 담는다. attrs 객체는 앞에서 본 startElement( )에서 어트리뷰트에 접근하는 데 사용하는 메서드들을 갖는다. 더불어 네임스페이스를 다루는 데 사용하는 다음 추가 메서드들도 있다.

메서드	설명
attrs.getValueByQName(qname)	한정된 이름에 대한 값을 반환한다.
attrs.getNameByQName(qname)	주어진 이름에 대해 투플 (namespace, localname)을 반환한다.
attrs.getQNameByName(name)	튜플 (namespace, localname)로 지정한 name에 대해서 한정된 이름을 반환한다.
attrs.getQNames()	모든 어트리뷰트의 한정된 이름을 반환한다.

### c.startPrefixMapping(prefix, uri)

XML 네임스페이스 선언 시작 부분에서 호출된다. 예를 들어, 엘리먼트가 ‘`<foo:bar xmlns:foo="http://spam.com">`’로 정의되어 있으면 prefix는 ‘foo’로 설정되고 uri은 ‘`http://spam.com`’로 설정된다.

예

다음은 SAX 기반 파서를 사용해서 앞에서 살펴본 요리법 파일에 있는 재료 목록을 출력하는 예이다. xml,dom,minidom 절에 있는 예와 비교해보기 바란다.

```
from xml.sax import ContentHandler, parse

class RecipeHandler(ContentHandler):
 def startDocument(self):
 self.inititem = False

 def startElement(self, name, attrs):
 if name == 'item':
 self.num = attrs.get('num', '1')
 self.units = attrs.get('units', 'none')
 self.text = []
 self.inititem = True
 def endElement(self, name):
 if name == 'item':
 text = "\n".join(self.text)
 if self.units == 'none': self.units = ""
 unitstr = "%s %s" % (self.num, self.units)
 print("%-10s %s" % (unitstr, text.strip()))
 self.inititem = False
 def characters(self, data):
 if self.inititem:
 self.text.append(data)

parse("recipe.xml", RecipeHandler())
```

**Note**

`xml.sax` 모듈은 다양한 종류의 XML 데이터를 처리하고 나름의 파서를 만드는 등 많은 기능을 제공한다. 예를 들어, DTD 데이터나 기타 문서의 다른 부분을 파싱하는 데 사용할 수 있는 처리기 객체들이 있다. 자세한 정보는 온라인 문서를 참고하기 바란다.

**xml.sax.saxutils**

`xml.sax.saxutils` 모듈은 보통 SAX 파서와 함께 사용되지만 어디서든 일반적으로 유용하게 쓸 수 있는 유ти리티 함수와 객체를 정의한다.

**`escape(data [, entities])`**

문자열 `data`가 주어질 때 이 함수는 특정 문자를 탈출 순서열로 대체한다. 예를 들어, ‘<’은 ‘&lt;’로 대체된다. `entities`는 문자를 탈출 순서열에 매핑하는 옵션인 사전이다. 예를 들어, `entities`를 { u'\xf1' : '&ntilde;' }로 설정하면 ñ을 ‘&ntilde;’로 대체한다.

**`unescape(data [, entities])`**

`data`에 있는 특수 탈출 순서열을 되돌린다. 예를 들어, ‘&lt;’은 ‘<’로 대체된다. `entities`는 탈출 순서열을 원래 문자 값으로 매핑하는 옵션인 사전이다. `entities`는 `escape()`에 사용되는 사전을 반대로 한 것이다. 예를 들어, { ‘&ntilde;’ : u'\xf1' }.

**`quoteattr(data [, entities])`**

문자열 `data`를 탈출시키지만 추가로 처리해서 결과를 XML 어트리뷰트 값으로 사용할 수 있게 한다. 예를 들어, `print "<element attr=%s>" % quoteattr(somevalue)`. `entities`는 `escape()` 함수에서 사용한 것과 호환되는 사전이다.

**`XMLGenerator([out [, encoding]])`**

파싱된 XML 데이터를 출력 스트림으로 XML 문서로서 단순히 다시 내보내는 ContentHandler 객체. 원래 XML 문서를 다시 생성한다. `out`은 출력 문서를 나타내며 기본으로 `sys.stdout`이다. `encoding`은 사용할 문자 인코딩이며 기본 값은 ‘iso-8859-1’이다. 파싱을 수행하는 코드를 디버깅할 때나 일단 제대로 작동하는 처리기를 사용하고 싶을 때 쓸 수 있다.

# 25장

Python Essential Reference

## 기타 라이브러리 모듈

이곳에서 다루는 모듈들은 이 책에서 상세하게 다루지는 않을 것이지만 그래도 표준 라이브러리에 들어 있는 것들이다. 이들은 아주 저수준 기능을 제공하거나, 사용할 때 한계점이 있거나, 특정 플랫폼에서만 사용이 제한되거나, 이제 쓰지 않게 되었거나, 제대로 다루기 위해서는 책 한 권이 필요할 정도로 복잡하기 때문에 이전 장들에서 다루지 않은 것들이다. 이 책에서 자세히 설명하지는 않지만 각 모듈에 대해서 온라인 문서가 있으니 참고하기 바란다(<http://docs.python.org/library/모듈이름>). 모든 모듈에 대한 색인은 <http://docs.python.org/library/modindex.html>에 가면 있다.

이곳에 나열된 모듈들은 파이썬 2와 3에 공통적인 것들이다. 여기에 나와 있지 않은 모듈을 사용하는 중이라면 공식적으로 사용이 권장되지 않는 모듈일 가능성이 크다. 어떤 모듈은 파이썬 3에서 이름이 바뀌었다. 새로운 이름이 있는 경우 괄호 안에 표시했다.

### 파이썬 서비스

다음 모듈들은 파이썬 언어와 파이썬 인터프리터 실행과 관련된 추가 서비스를 제공한다. 여기에 나온 많은 모듈이 파이썬 소스 코드를 파싱하고 컴파일하는 데 쓰인다.

모듈	설명
bdb	디버거 프레임워크에 접근

code	인터프리터 기반 클래스
codeop	파이썬 코드를 컴파일한다.
compileall	특정 디렉토리에 있는 파이썬 파일들을 바이트 컴파일한다.
copy_reg(copyreg)	pickle 모듈에서 사용할 내장 타입을 등록한다.
dis	역어셈블러
distutils	파이썬 모듈의 배포
fpectl	부동 소수점 예외 제어
imp	import문의 구현에 접근
keyword	문자열이 파이썬 키워드인지 검사한다.
linecache	소스 파일에서 특정 줄을 추출
modulefinder	스크립트에서 사용되는 모듈을 찾는다.
parser	파이썬 소스 코드의 파스 트리(parse tree)에 접근
pickletools	pickle 개발자를 위한 도구
pkgutil	패키지 확장 유ти리티
pprint	객체를 깔끔하게 출력하는 출력기
pyclbr	클래스 브라우저를 위한 정보 추출
py_compile	파이썬 소스를 바이트코드 파일로 컴파일
repr(reprlib)	repr() 함수의 다른 구현
symbol	파스 트리의 내부 노드를 표현하는 데 사용하는 상수들
tabnanny	애매한 들여쓰기 감지
test	회귀 검사 패키지
token	파스 트리의 종말 노드
tokenize	파이썬 소스 코드 스캐너
user	사용자 설정 파일 파싱
zipimport	zip 아카이브에서 import를 수행하는 모듈

## 문자열 처리

다음은 지금은 쓰지 않는 문자열 처리를 위해 사용되었던 오래된 모듈들이다.

모듈	설명
difflib	문자열 사이에 차이를 계산한다.
fpformat	부동 소수점 숫자 형식 지정
stringprep	인터넷을 위해 문자열을 준비한다.
textwrap	텍스트 래퍼

## 운영체제 모듈

다음 모듈들은 추가적인 운영체제 시스템 서비스를 제공한다. 여기에 나와 있는 모듈 중에는 19장에서 다루었던 모듈에 이미 그 기능이 통합된 것도 있다.

모듈	설명
crypt	유닉스 crypt 함수에 접근
curses	curses 라이브러리 인터페이스
grp	그룹 데이터베이스에 접근
pty	가상 터미널(pseudo terminal) 처리
pipes	셸 파이프라인 인터페이스
nis	선(Sun)의 NIS 인터페이스
platform	플랫폼 종속적인 정보에 접근
pwd	암호 데이터베이스에 접근
readline	GNU readline 라이브러리에 접근
rlcompleter	GNU readline 종료 함수
resource	자원 사용 정보
sched	이벤트 스케줄러
spwd	섀도(shadow) 암호 데이터베이스에 접근
stat	os,stat()의 결과를 해석하는 기능 제공
syslog	유닉스 syslog 대몬 인터페이스
termios	유닉스 TTY 제어
tty	터미널 제어 함수

## 네트워크

다음 모듈들은 덜 자주 사용되는 네트워크 프로토콜에 대한 지원 기능을 제공한다.

모듈	설명
imaplib	IMAP 프로토콜
nntplib	NNTP 프로토콜
poplib	POP3 프로토콜
smtpd	SMTP 서버
telnetlib	Telnet 프로토콜

## 인터넷 데이터 처리

다음 모듈들은 24장에서 다루지 않은 인터넷 데이터 처리 기능을 제공한다.

모듈	설명
binhex	BinHex4 파일 형식 지원
formatter	범용 출력 포맷 지정
mailcap	mailcap 파일 처리
mailbox	다양한 메일함 형식을 읽음
netrc	netrc 파일 처리
plistlib	매킨토시 plist 파일 처리
uu	UUencode 파일 지원

xdrlib	선 XDR 데이터 인코딩과 디코딩
--------	--------------------

## 국제화

다음 모듈들은 다국어 응용 프로그램을 작성하는 데 사용한다.

모듈	설명
gettext	다국어 텍스트 처리 서비스
locale	시스템이 제공하는 국제화 함수

## 멀티미디어 서비스

다음 모듈들은 다양한 종류의 멀티미디어 파일을 처리하는 기능을 제공한다.

모듈	설명
audioop	미가공 오디오 데이터를 조작
aifc	AIFF와 AIFF 파일 읽기와 쓰기
sunau	Sun AU 파일 읽기와 쓰기
wave	WAV 파일 읽기와 쓰기
chunk	IFF 청크 데이터 읽기
colorsys	색상 시스템 사이에 변환
imghdr	이미지 종류 결정
sndhdr	사운드 파일 종류 결정
ossaudiodev	OSS 호환 오디오 장치에 접근

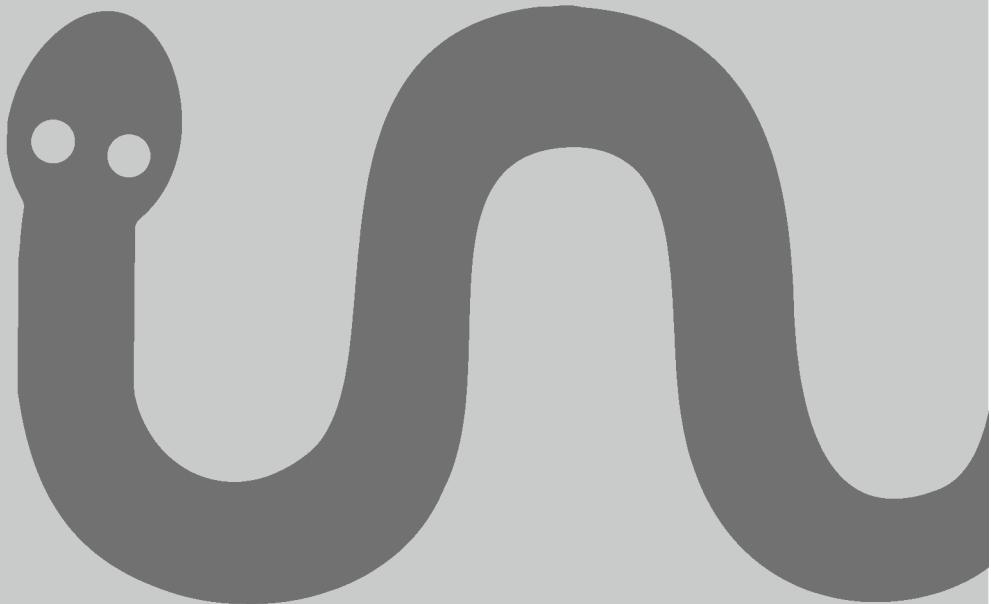
## 기타

다음 모듈들은 딱히 분류하기 어려운 모듈들이다.

모듈	설명
cmd	줄 기반 명령 인터프리터
calendar	달력 생성 함수
shlex	간단한 어휘 분석 모듈
Tkinter(tkinter)	파이썬 Tcl/Tk 인터페이스
winsound	윈도에서 소리 재생

# 3부

## 확장과 임베딩



P y t h o n   E s s e n t i a l   R e f e r e n c e



---

# 26장

Python Essential Reference

---

## 파이썬 확장과 임베딩

---

파이썬의 강력한 기능 중 하나는 C로 작성한 소프트웨어와 상호작용할 수 있다는 점이다. 파이썬을 외부 코드와 통합하는 데는 두 가지 방식이 흔히 사용된다. 첫째, 외부 코드를 import문으로 사용할 수 있게 파이썬 라이브러리 모듈로 패키징 하는 방법이 있다. 인터프리터를 파이썬으로 작성하지 않은 기능으로 확장하는 것 이기 때문에 이런 모듈을 확장 모듈(extension module)이라고 부른다. 파이썬에서 고성능 프로그래밍 라이브러리에 접근할 수 있기 때문에 파이썬과 C를 통합할 때 이 방식을 가장 많이 사용한다. 파이썬과 C를 통합하는 다른 방식으로는 임베딩(embedding)이란 것이 있다. 임베딩에서는 C에서 라이브러리 형태로 파이썬 프로그램과 인터프리터에 접근한다. 이 방식은 파이썬을 스크립팅 엔진으로 사용한다든지 어떤 이유에서든 기존 C 응용 프레임워크에 파이썬 인터프리터를 내장하고 싶은 경우 사용한다.

이 장에서는 파이썬-C 프로그래밍 인터페이스와 관련해 가장 기본적인 내용을 다룬다. 먼저, 확장 모듈을 생성하는 방법과 C에 파이썬 인터프리터를 임베딩하기 위한 C API에서 핵심적인 부분을 다룬다. 이 절은 튜토리얼 형식으로 되어 있지 않다. 이 주제에 생소한 독자들은 <http://docs.python.org/extending>에 있는 〈파이썬 인터프리터 임베딩과 확장(Embedding and Extending the Python Interpreter)〉이나 <http://docs.python.org/c-api>에 있는 〈파이썬/C API 참조 메뉴얼(Python/C API Reference Manual)〉 문서를 참고하기 바란다. 이어서 ctypes 라이브러리 모듈을 다룬다. 이 모듈은 추가로 C 코드를 작성하지 않으면 C 컴파일러를 사용하지 않고서

파이썬에서 C 라이브러리에 있는 함수에 바로 접근할 수 있게 해주는 매우 유용한 라이브러리 모듈이다.

고급 확장과 임베딩 응용에서는 대부분의 프로그래머가 고급 코드 생성기와 프로그래밍 라이브러리에 의존한다. 예를 들어, SWIG(<http://www.swig.org>)는 C 헤더 파일을 파싱해서 파이썬 확장 모듈을 생성하는 컴파일러이다. 이 모듈과 기타 다른 확장 기능 생성 도구에 관해서는 <http://wiki.python.org/moin/IntegratingPythonWithOtherLanguages>를 참고하기 바란다.

## 확장 모듈

이 절에서는 파이썬에서 쓸 수 있도록 C 확장 모듈을 손으로 직접 생성하는 기본적인 과정을 대략적으로 설명한다. 확장 모듈을 작성하는 일은 C로 작성한 기존 기능과 파이썬 사이에 인터페이스를 만드는 일과 같다. C 라이브러리를 작성할 때는 보통 다음과 같이 헤더 파일을 가지고 시작한다.

```
/* 파일: example.h */
#include <stdio.h>
#include <string.h>
#include <math.h>

typedef struct Point {
 double x;
 double y;
} Point;

/* 두 정수 x와 y의 GCD(최대공약수)를 계산한다. */
extern int gcd(int x, int y);

/* s에서 och를 nch로 바꾸고 바꾼 횟수를 반환한다. */
extern int replace(char *s, char och, char nch);
}

/* 두 점 사이의 거리를 계산한다. */
extern double distance(Point *a, Point *b);

/* 전처리 상수 */
#define MAGIC 0x31337
```

이 함수 프로토타입들의 구현 내용은 별개의 파일에 들어 있다. 다음 예를 보자.

```
/* example.c */
#include "example.h"
/* 두 정수 x와 y의 GCD(최대공약수)를 계산한다. */
int gcd(int x, int y) {
 int g;
```

```

g = y;
while (x > 0) {
 g = x;
 x = y % x;
 y = g;
}
return g;
}

/* 문자열에서 문자를 바꾼다. */
int replace(char *s, char oldch, char newch) {
 int nrep = 0;
 while (*s == oldch) {
 *s++ = newch;
 nrep++;
 }
 return nrep;
}

/* 두 점 사이의 거리 */
double distance(Point *a, Point *b) {
 double dx,dy;
 dx = a->x - b->x;
 dy = a->y - b->y;
 return sqrt(dx*dx + dy*dy);
}

```

다음은 이 함수들을 사용하는 모습을 보여준다.

```

/* main.c */
#include "example.h"
int main() {
 /* gcd() 함수를 테스트한다. */
 {
 printf("%d\n", gcd(128,72));
 printf("%d\n", gcd(37,42));
 }
 /* replace() 함수를 테스트한다. */
 {
 char s[] = "Skipping along unaware of the unspeakable peril.";
 int nrep;
 nrep = replace(s, ',', '-');
 printf("%d\n", nrep);
 printf("%s\n", s);
 }
 /* distance() 함수를 테스트한다. */
 {
 Point a = { 10.0, 15.0 };
 Point b = { 13.0, 11.0 };
 printf("%0.2f\n", distance(&a,&b));
 }
}

```

다음은 앞의 프로그램을 실행한 결과다.

```
% a.out
8
1
6
Skipping-along-unaware-of-the-unspeakable-peril.
5.00
```

## 확장 모듈 프로토타입

확장 모듈은 파이썬 인터프리터와 C 코드 사이에 접착제 역할을 하는 일련의 래퍼 함수들을 담은 별개의 C 소스 파일로 작성한다. 다음은 `_example.o`라고 부르는 간단한 확장 모듈의 예를 보여준다.

```
/* pyexample.c */

#include "Python.h"
#include "example.h"

static char py_gcd_doc[] = "Computes the GCD of two integers";
static PyObject *
py_gcd(PyObject *self, PyObject *args) {
 int x,y,r;
 if (!PyArg_ParseTuple(args,"ii:gcd",&x,&y)) {
 return NULL;
 }
 r = gcd(x,y);
 return Py_BuildValue("i",r);
}

static char py_replace_doc[] = "Replaces all characters in a string";
static PyObject *
py_replace(PyObject *self, PyObject *args, PyObject *kwargs) {
 static char *argnames[] = {"s","och","nch",NULL};
 char *s,*sdup;
 char och, nch;
 int nrep;
 PyObject *result;
 if (!PyArg_ParseTupleAndKeywords(args,kwds, "scc:replace",
 argnames, &s, &och, &nch)) {
 return NULL;
 }
 sdup = (char *) malloc(strlen(s)+1);
 strcpy(sdup,s);
 nrep = replace(sdup,och,nch);
 result = Py_BuildValue("(is)",nrep, sdup);
 free(sdup);
 return result;
}
```

```

static char py_distance_doc[] = "Computes the distance between two points";
static PyObject *
py_distance(PyObject *self, PyObject *args) {
 PyErr_SetString(PyExc_NotImplementedError,
 "distance() not implemented.");
 return NULL;
}

static PyMethodDef _examplemethods[] = {
 {"gcd", py_gcd, METH_VARARGS, py_gcd_doc},
 {"replace", py_replace, METH_VARARGS | METH_KEYWORDS,
 py_replace_doc},
 {"distance", py_distance, METH_VARARGS, py_distance_doc},
 {NULL, NULL, 0, NULL}
};

#endif PY_MAJOR_VERSION < 3
/* 파이썬 2 모듈 초기화 */
void init_example(void) {
 PyObject *mod;
 mod = Py_InitModule("_example", _examplemethods);
 PyModule_AddIntMacro(mod, MAGIC);
}
#else
/* 파이썬 3 모듈 초기화 */
static struct PyModuleDef _examplemodule = {
 PyModuleDef_HEAD_INIT,
 "_example", /* 모듈 이름 */
 NULL, /* 모듈 문서화. NULL일 수 있다. */
 -1,
 _examplemethods
};
PyMODINIT_FUNC
PyInit__example(void) {
 PyObject *mod;
 mod = PyModule_Create(&_examplemodule);
 PyModule_AddIntMacro(mod, MAGIC);
 return mod;
}
#endif

```

확장 모듈에는 항상 “Python.h”를 포함시켜야 한다. 그리고 접근할 각 C 함수에 대해 래퍼 함수를 작성한다. 래퍼 함수는 두 개의 인수를(PyObject \* 타입인 self와 args) 받거나 세 개의 인수(PyObject\* 타입인 self, args, kwargs)를 받아야 한다. self 매개변수는 래퍼 함수에서 객체 인스턴스의 내장 메서드를 구현할 때 사용한다. 이 경우 해당 인스턴스가 self 매개변수에 저장된다. 아니면 self는 NULL로 설정된다. args는 인터프리터가 전달한 함수 인수들을 담은 튜플이다. kwargs는 키워드 인수들을 담은 사전이다.

인수들은 PyArg\_ParseTuple( )이나 PyArg\_ParseTupleAndKeywords( ) 함수

를 사용해 파이썬에서 C로 변환한다. 이와 비슷하게, 반환 값을 생성하는 데 Py\_BuildValue( ) 함수를 사용한다. 이 함수들에 대해서는 나중에 다시 설명한다.

확장 함수용 문서화 문자열은 앞에서 보듯이 py\_gcd\_doc나 py\_replace\_doc 같은 별개의 문자열 변수에 저장해야 한다. 이 변수들은 모듈 초기화 과정에 쓰인다 (곧 설명한다).

래퍼 함수는 절대 인터프리터로부터 참조를 통해 받은 데이터를 변경해서는 안 된다. 앞에서 이 때문에 받은 문자열을 py\_replace( ) 래퍼에서 C 함수(문자열을 바로 수정한다)로 전달하기 전에 복사본을 생성했다. 이 단계가 없으면 래퍼 함수에서 파이썬의 문자열 불변성을 깰 수도 있다.

예외를 던지고 싶으면 py\_distance( ) 래퍼에서 보듯이 PyExc\_SetString( ) 함수를 사용하면 된다. 그리고 이어서 에러가 발생했다는 것을 알리기 위해 NULL을 반환하였다.

파이썬 이름을 C 래퍼 함수에 묶는 데 메서드 표 \_examplemethods를 사용했다. 이 이름들은 인터프리터에서 함수를 호출하는 데 사용한다. METH\_VARARGS 플래그는 래퍼에 대한 호출 규약을 나타낸다. 이 플래그만 지정하면 튜플 형태인 위치 인수만 허용한다. METH\_VARARGS | METH\_KEYWORDS로 설정해서 래퍼 함수에서 키워드 인수를 받게 할 수도 있다. 메서드 표에는 각 래퍼 함수의 문서화 문자열도 설정한다.

마지막으로 확장 모듈에서 파이썬 2와 3에서 서로 다른 초기화 과정을 수행한다. 파이썬 2에서는 모듈의 내용을 초기화하는 데 모듈 초기화 함수인 init\_example를 사용한다. 여기서 Py\_InitModule("example", \_examplemethods) 함수는 모듈 \_example을 생성하고 이것을 메서드 표에 나열된 함수들에 대응하는 내장 함수 객체들로 채운다. 파이썬 3에서는 모듈을 설명하는 PyModuleDef 객체인 \_examplemodule을 생성해야 한다. 그다음에 앞에서 보듯이 모듈을 초기화하는 PyInit\_\_example( ) 함수를 작성한다. 모듈 초기화 함수는 필요할 경우 상수라든지 기타 모듈의 다른 부분을 설정하는 곳이기도 하다. 예를 들어, PyModule\_AddIntMacro( )는 전처리 값을 모듈에 추가한다.

모듈을 초기화할 때 이름 짓기가 매우 중요하다. modname이라는 모듈을 생성하는 것이라면 파이썬 2에서는 모듈 초기화 함수의 이름이 initmodname( )이어야 하고 파이썬 3에서는 PyInit\_modname( )이어야 한다. 이렇게 하지 않으면 인터프리

터가 모듈을 제대로 로드하지 못한다.

## 확장 모듈 이름

C 확장 모듈의 이름은 '\_example'처럼 밑줄로 시작하는 것이 관례이다. 파이썬 표준 라이브러리도 이 관례를 따른다. 예를 들어, C 프로그래밍 컴포넌트인 socket, threading, re, io 모듈에 대응하는 \_socket, \_thread, \_sre, \_fileno라는 이름의 모듈이 있다. 보통 C 확장 모듈을 직접 사용하지는 않는다. 대신 다음과 같이 고수준 파이썬 모듈을 생성해서 사용한다.

```
example.py
from _example import *
추가 지원 코드를 아래에 추가한다.
...
```

이렇게 파이썬 래퍼를 사용하는 이유는 추가 지원 코드를 제공하거나 고수준 인터페이스 제공하기 위해서이다. 많은 경우 확장 모듈의 일부를 C보다 파이썬으로 구현하는 것이 수월하다. 파이썬 래퍼를 사용하면 이를 쉽게 할 수 있다. 표준 라이브러리 모듈을 살펴보면 많은 모듈이 위와 같은 방식으로 C와 파이썬을 섞어 구현한 것을 확인할 수 있을 것이다.

## 확장 기능 컴파일과 패키징

확장 모듈을 컴파일하고 패키징하는 메커니즘 중 distutils를 사용하는 방법이 널리 사용된다. 먼저 다음처럼 생긴 setup.py 파일을 생성한다.

```
setup.py
from distutils.core import setup, Extension

setup(name="example",
 version="1.0",
 py_modules = ['example.py'],
 ext_modules = [
 Extension("_example",
 ["pyexample.c","example.c"])
]
)
```

이 파일에 고수준 파이썬 파일(example.py)과 확장 모듈을 구성하는 소스 파일들(pyexample.c, example.c)을 적어주어야 한다. 테스트를 위해 모듈을 빌드하려면 다음과 같이 한다.

```
% python setup.py build_ext --inplace
```

이렇게 하면 확장 코드가 공유 라이브러리로 컴파일되고 현재 작업 디렉터리에

저장된다. 라이브러리의 이름은 \_examplemodule.so, \_examplemodule.pyd나 기타 비슷한 형태가 될 것이다.

컴파일이 제대로 되었으면 모듈을 사용하는 일은 간단하다. 다음 예를 보자.

```
% python3.0
Python 3.0 (r30:67503, Dec 4 2008, 09:40:15)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import example
>>> example.gcd(78,120)
6
>>> example.replace("Hello World", ' ', '-')
(1, 'Hello-World')
>>> example.distance()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NotImplementedError: distance() not implemented.
>>>
```

더 복잡한 확장 모듈이라면 추가로 디렉터리, 라이브러리, 전처리 매크로 같은 빌드 정보를 담을 수 있다. 이런 정보도 다음과 같이 setup.py에 포함시킬 수 있다.

```
setup.py
from distutils.core import setup, Extension

setup(name="example",
 version="1.0",
 py_modules = ['example.py'],
 ext_modules = [
 Extension("_example",
 ["pyexample.c","example.c"],
 include_dirs = ["/usr/include/X11","/opt/include"],
 define_macros = [('DEBUG',1),
 ('MONDO_FLAG',1)],
 undef_macros = ['HAVE_FOO','HAVE_NOT'],
 library_dirs= ["/usr/lib/X11", "/opt/lib"],
 libraries = ["X11", "Xt", "blah"])
])
```

확장 모듈을 범용적으로 사용할 목적으로 설치하고 싶다면 간단히 python setup.py install을 입력하면 된다. 여기에 관한 더 자세한 설명은 8장에서 찾을 수 있다.

어떤 경우에는 확장 모듈을 직접 빌드하고 싶은 경우도 있다. 이럴 경우에는 대부분 다양한 컴파일러와 링커 옵션에 관한 고급 지식이 필요하다. 다음은 리눅스의 예를 보여준다.

```
linux % gcc -c -fPIC -I/usr/local/include/python2.6 example.c pyexample.c
```

```
linux % gcc -shared example.o pyexample.o -o _examplemodule.so
```

## 파이썬에서 C로 타입 변환

다음은 확장 모듈에서 파이썬 인수들을 C로 변환하기 위해 사용하는 함수들이다. 이 함수들의 프로토타입이 정의되어 있는 Python.h 헤더 파일을 포함시키면 이 함수들을 사용할 수 있다.

```
int PyArg_ParseTuple(PyObject *args, char *format, ...);
```

args에 있는 위치 인수들의 튜플을 일련의 C 변수들로 파싱한다. format은 표 26.1에서 26.3까지 나와 있는 영 개 또는 하나 이상의 지정자 문자열을 담은 포맷 지정 문자열이며 args가 담고 있기를 기대하는 내용을 기술한다. 나머지 인수들은 결과를 담을 C 변수 주소들을 담는다. 이 인수들의 순서와 타입은 format에 사용된 지정자들과 일치해야 한다. 인수를 파싱할 수 없으면 영이 반환된다.

```
int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kwargs,
 char *format, char **kwlist, ...);
```

위치 인수들의 튜플과 kwargs에 있는 키워드 인수들의 사전 둘 모두를 파싱한다.

**표 26.1** PyArg\_Parse\*의 숫자 변환 및 연관 C 데이터 타입

포맷	파이썬 타입	C 인수 타입
“b”	정수	signed char *r
“B”	정수	unsigned char *r
“h”	정수	short *r
“H”	정수	unsigned short *r
“r”	정수	int *r
“R”	정수	unsigned int *r
“l”	정수	long int *r
“K”	정수	unsigned long *r
“L”	정수	long long *r
“K”	정수	unsigned long long *r
“n”	정수	Py_ssize_t *r
“f”	실수(float)	float *r
“d”	실수(float)	double *r
“D”	복소수	Py_complex *r

format은 PyArg\_ParseTuple( )에서 의미와 같다. 유일한 차이점은 kwlist가 모든 인수의 이름을 담은 널 문자로 종료되는 문자열들의 리스트라는 점이다. 성공하면 1을 에러인 경우 0을 반환한다.

표 26.1은 format 인수에서 숫자를 변환하기 위해 사용할 수 있는 포맷 코드를 나열한 것이다. C 인수 타입 열은 PyArg\_Parse\*( ) 함수에 전달해야 하는 C 데이터 타입을 나타낸다. 숫자에 대해서 C 인수 타입은 항상 결과가 저장되어야 하는 위치를 가리키는 포인터이다.

부호 있는 정수 값을 변환할 때, 파이썬 정수가 요청된 C 데이터 타입으로 변환하기에 너무 클 경우에는 OverflowError 예외가 발생한다. 부호 없는 값('I', 'H', 'K' 등)을 받는 변환에서는 오버플로우를 검사하지 않고 지원하는 범위를 넘을 경우 조용히 값을 자른다. 부동 소수점 변환에 대해서 파이썬 int나 float가 입력으로 주어질 수 있다. 이 경우 정수는 float로 강제 타입 변환된다. 사용자 정의 클래스에서 \_\_int\_\_( )나 \_\_float\_\_( ) 같은 적절한 변환 메서드를 제공하는 경우 해당 클래스를 숫자로 인식한다. 예를 들어, 사용자 정의 클래스에서 \_\_int\_\_( )를 구현하면 앞서 나온 아무 정수 변환이나 입력으로 사용할 수 있다(변환을 위해 \_\_int\_\_( )을 알아서 호출한다).

표 26.2는 문자열과 바이트에 적용되는 변환을 보여준다. 문자열 변환 중 많은 것이 포인터와 길이 둘다 결과로 반환한다.

**표 26.2. PyArg\_Parse\*의 문자열 변환 및 연관 C 데이터 타입**

포맷	파이썬 타입	C 인수 타입
"c"	문자열 또는 길이 1인 바이트 문자열	char *r
"s"	문자열	char **r
"s#"	문자열, 바이트들 또는 버퍼	char **r, int *len
"o**"	문자열, 바이트들 또는 버퍼	Py_buffer *r
"z"	문자열 또는 None	char **r
"z#"	문자열, 바이트들 또는 None	char **r, int *len
"z**"	문자열, 바이트들, 버퍼 또는 None	Py_buffer *r
"y"	바이트들(널로 종료됨)	char **r
"y#"	바이트들	char **r, int *len
"y**"	바이트들 또는 버퍼	Py_buffer *r
"u"	문자열(유니코드)	Py_UNICODE **r

“u#”	문자열(유니코드)	Py_UNICODE **r, int *len
“es”	문자열	const char *enc, char **r
“es#”	문자열 또는 바이트들	const char *enc, char **r, int *len
“et”	문자열 또는 널로 종료되는 바이트들	const char *enc, char **r, int *len
“et#”	문자열 또는 바이트들	const char *enc, char **r, int *len
“t#”	읽기 전용 버퍼	char **r, int *len
“w”	읽기 쓰기 버퍼	char **r
“w#”	읽기 쓰기 버퍼	char **r, int *len
“w*”	읽기 쓰기 버퍼	Py_buffer *r

char \* 데이터 타입은 여러 가지 용도로 사용되기 때문에 C 확장 기능에서 문자열을 처리해야 할 때 문제가 생길 수 있다. 예를 들어, 이 데이터 타입은 텍스트, 단일 문자 또는 무가공 이진 데이터를 담은 버퍼 등을 가리킬 수 있다. 또한 C에서 텍스트 문자열의 끝을 알리는 데 사용하는 NULL 문자('x00')을 어떻게 처리할 것인지에 관한 문제도 있다.

텍스트를 전달하려면 표 26.2에 있는 변환 코드 중에 “s”, “z”, “u”, “es” 그리고 “et”를 사용해야 한다. 이들 코드에 대해서 파이썬은 입력 텍스트에 NULL이 포함되어 있지 않다고 가정한다. NULL이 들어 있는 경우에는 TypeError 예외가 발생한다. 그렇지만 C에서는 변환 결과 문자열이 NULL로 종료된다고 가정해도 안전하다. 파이썬 2에서는 8비트 문자열이나 유니코드 문자열 모두 전달할 수 있었지만 파이썬 3에서는 “et”를 제외한 모든 변환에서 파이썬 str 타입이 필요하고 bytes는 제대로 지원하지 않는다. C에 유니코드 문자열이 전달될 때는 항상 인터프리터가 기본으로 사용하는 유니코드 인코딩에 따라 인코딩된다(기본으로 UTF-8). 한 가지 예외는 파이썬의 내부 유니코드 표현으로 되어 있는 문자열을 반환하는 “u” 변환 코드를 사용하는 경우이다. 이 경우 반환되는 문자열은 보통 C에서 wchar\_t 타입으로 표현되는 Py\_UNICODE 타입인 값들의 배열로 표현된다.

“es”와 “et” 코드는 텍스트에 대해 다른 인코딩을 지정할 수 있게 한다. 이 경우 ‘utf-8’이나 ‘iso-8859-1’ 같은 인코딩 이름을 지정하면 텍스트는 인코딩되어 버퍼에 담기고 해당 형식으로 반환된다. “et” 코드는 파이썬 바이트 문자열이 주어질 때 이미 인코딩되어 있다고 가정하여 수정하지 않고 전달한다는 점에서 “es”와 다르다. “es”나 “et” 변환을 사용할 때 주의할 점으로는 이들은 결과를 저장하기 위해 메모

리를 동적으로 할당하기 때문에 사용자가 PyMem\_Free()를 사용해 직접 메모리를 해제해야 한다는 점이다. 따라서 이들 변환을 사용하는 코드는 보통 다음과 같은 모습을 보인다.

```
PyObject *py_wrapper(PyObject *self, PyObject *args) {
 char *buffer;
 if (!PyArg_ParseTuple(args, "es", "utf-8", &buffer)) {
 return NULL;
 }
 /* 뭔가를 한다. */
 ...
 /* 정리한 후 결과를 반환한다. */
 PyMem_Free(buffer);
 return result;
}
```

텍스트나 이진 데이터를 다루어야 할 때는 “s#”, “z#”, “u#”, “es#”나 “et#” 코드를 사용한다. 이들 변환은 앞서 나온 것과 정확히 같지만 길이를 추가로 반환하는 점이 다르다. 이 점 때문에 내부에 NULL 문자를 포함할 수 없는 제약에서 자유로울 수 있다. 또한 버퍼 인터페이스라고 부르는 것을 지원하는 바이트 문자열이나 기타 어떤 객체라도 처리할 수 있다. 버퍼 인터페이스(buffer interface)는 파이썬 객체가 자신의 내용을 표현하는 무가공 이진 버퍼를 공개할 수 있는 방법을 제공한다. 보통 문자열, 바이트들, 배열에서 볼 수 있다(array 모듈로 생성한 배열이 버퍼 인터페이스를 지원한다). 이 경우 어떤 객체가 읽기용 버퍼 인터페이스를 제공하면 버퍼를 가리키는 포인터와 버퍼 크기가 반환된다. 마지막으로, “es#”나 “et#”에 NULL이 아닌 포인터와 길이가 주어지면 인코딩 결과를 저장할 미리 할당된 버퍼를 나타낸다고 가정한다. 이 경우 인터프리터에서 결과를 담을 메모리를 할당하지 않으며 PyMem\_Free()를 호출하지 않아도 된다.

변환 코드 “s\*”와 “z\*”는 “s#”과 “z#”과 비슷하지만 받은 데이터에 대한 정보로 Py\_buffer 구조체를 채운다는 점이 다르다. 여기에 관한 더 자세한 정보는 PEP-3118에 나와 있으며, 이 구조체는 최소한으로 버퍼를 가리키는 포인터, 버퍼 길이(바이트로), 버퍼에 담긴 항목들의 크기를 나타내는 속성 char \*buf, int len, int itemsize를 가진다. 또한 확장 모듈이 사용하는 동안 다른 스레드에서 그 내용을 바꿀 수 없도록 인터프리터는 버퍼에 락을 건다. 이 때문에 확장 모듈에서는 버퍼의 내용을 독립적으로 사용할 수 있으며 인터프리터와 다른 스레드에서도 그렇다. 모든 처리가 끝난 후 버퍼에 대고 PyBuffer\_Release()를 호출할지 여부는 사용자가 결정한다.

변환 코드 “t#”, “w”, “w#”, “w\*”는 “s” 계열 코드와 비슷하지만 버퍼 인터페이스를 구현하는 객체만 허용한다. “t#”는 버퍼가 읽기 가능할 것을 요구한다. “w” 코드는 버퍼가 읽기 쓰기 모두 가능할 것을 요구한다. 쓰기 가능한 버퍼를 지원하는 파이썬 객체는 변경 가능한 것으로 가정된다. 따라서 C 확장 기능에서 버퍼의 내용을 덮어쓰거나 수정할 수 있다.

변환 코드 “y”, “y#”, “y\*”는 “s” 계열 코드와 비슷하지만 바이트 문자열만 허용한다. 유니코드 문자열이 아니고 바이트들을 받아야 하는 함수를 작성할 때 사용하도록 한다. “y” 코드는 NULL 문자를 담지 않은 바이트 문자열만 받는다.

표 26.3은 임의의 파이썬 객체를 입력으로 받아서 결과를 타입 PyObject \*로 저장하는 데 사용하는 변환 코드들을 보여준다. 이들은 C 확장 모듈에서 단순 숫자나 문자열보다 더 복잡한 파이썬 객체를 다룰 때 사용한다. 예를 들어, C 확장 함수에서 파이썬 클래스의 인스턴스나 사전을 받고 싶을 때 사용할 수 있다.

**표 26.3** 파이썬 객체 변환과 PyArg\_Parse\* 관련 C 데이터 타입

포맷	파이썬 타입	C 타입
“O”	아무거나	PyObject **r
“O!”	아무거나	PyTypeObject *type, PyObject **r
“O&”	아무거나	int (*converter)(PyObject *, void *), void *r
“S”	문자열	PyObject **r
“U”	유니코드	PyObject **r

“O!” 변환 코드는 두 개의 C 인수를 필요로 한다. 파이썬 타입 객체를 가리키는 포인터와 해당 객체를 가리키는 포인터를 저장할 PyObject \*에 대한 포인터이다. 타입 객체와 객체의 타입이 일치하지 않으면 TypeError 예외가 발생한다. 다음 예를 보자.

```
/* 리스트 인수를 파싱한다. */
PyObject *listobj;
PyArg_ParseTuple(args, "O!", &PyList_Type, &listobj);
```

다음 목록은 이 변환에 주로 사용되는 몇 가지 파이썬 컨테이너 타입에 대응하는 C 타입 이름을 보여준다.

C 이름	파이썬 타입
PyList_Type	list
PyDict_Type	dict

PySet_Type	set
PyFrozenSet_Type	frozen_set
PyTuple_Type	tuple
PySlice_Type	slice
PyByteArray_Type	bytearray

“O&”는 두 인수 (converter, addr)를 받으며 PyObject \*를 C 데이터 타입으로 바꾸기 위해 함수를 사용한다. converter는 프로토타입이 int converter(PyObject \*obj, void \*addr)인 함수를 가리키는 포인터이다. 여기서 obj는 전달된 파이썬 객체이고 addr는 PyArg\_ParseTuple()에 두 번째 인수로 전달된 주소이다. converter()는 성공하면 1을, 실패하면 0을 반환해야 한다. 예러가 있을 때 converter()는 반드시 예외를 던져야 한다. 이런 종류의 변환을 통해서 파이썬 객체를 리스트나 튜플 같은 C 데이터 구조에 매핑할 수 있다. 예를 들어, 다음은 앞에 나왔던 코드에서 distance() 래퍼를 구현하는 예를 보여준다.

```
/* 튜플을 Point 구조체로 변환한다. */
int convert_point(PyObject *obj, void *addr) {
 Point *p = (Point *) addr;
 return PyArg_ParseTuple(obj, "ii", &p->x, &p->y);
}
PyObject *py_distance(PyObject *self, PyObject *args) {
 Point p1, p2;
 double result;
 if (!PyArg_ParseTuple(args, "O&O&",
 convert_point, &p1, convert_point, &p2)) {
 return NULL;
 }
 result = distance(&p1, &p2);
 return Py_BuildValue("d", result);
}
```

마지막으로 인수 포맷 문자열은 튜플 풀어헤치기, 문서화, 예러 메시지, 기본 인수 등에 관련된 몇 가지 추가적인 변경자를 담을 수 있다. 다음은 이러한 변경자 목록을 보여준다.

포맷 문자열	설명
“(items)”	객체들의 튜플을 풀어헤친다. items는 포맷 변환 코드들을 나타낸다.
“ ”	옵션인 인수의 시작
“:”	인수 끝. 나머지 텍스트는 함수 이름이다.
“;”	인수 끝. 나머지 텍스트는 예러 메시지이다.

“(items)”는 파이썬 튜플을 값들로 풀어헤친다. 튜플을 간단한 C 구조체에 매핑하고자 할 때 유용하게 쓸 수 있다. 예를 들어, 다음은 py\_distance() 래퍼 함수를

구현하는 다른 예를 보여준다.

```
PyObject *py_distance(PyObject *self, PyObject *args) {
 Point p1, p2;
 double result;
 if (!PyArg_ParseTuple(args, "(dd)(dd)",
 &p1.x, &p1.y, &p2.x, &p2.y)) {
 return NULL;
 }
 result = distance(&p1, &p2);
 return Py_BuildValue("d", result);
}
```

변경자 “!”는 나머지 인수들이 옵션임을 지정한다. 이 변경자는 포맷 지정자에서 한 번만 나타날 수 있고 중첩될 수 없다. 변경자 “:”는 인수 끝을 가리킨다. 이어서 나오는 텍스트는 에러 메시지에서 함수 이름으로 사용된다. 변경자 “;”는 인수 끝을 알린다. 이어서 나오는 텍스트는 에러 메시지로 사용된다. :와 ; 중에 하나만 사용할 수 있다. 다음 예를 보자.

```
PyArg_ParseTuple(args, "ii:gcd", &x, &y);
PyArg_ParseTuple(args, "ii; gcd requires 2 integers", &x, &y);

/* 옵션인 인수를 파싱한다 */
PyArg_ParseTuple(args, "s|s", &buffer, &delimiter);
```

## C에서 파이썬으로 타입 변환

다음 C 함수는 C 변수에 담긴 값을 파이썬 객체로 변환하는 데 사용한다.

```
PyObject *Py_BuildValue(char *format, ...)
```

이 함수는 일련의 C 변수들로부터 파이썬 객체를 구축한다. format은 원하는 변환을 기술하는 문자열이다. 나머지 인수들은 변환에 사용할 C 변수들을 지정한다. format은 PyArg\_ParseTuple\* 함수에서 사용하는 것과 비슷하며 표 26.4에 설명이 나와 있다.

**표 26.4** Py\_BuildValue( )용 포맷 지정자

포맷	파이썬 타입	C 타입	설명
“”	None	void	아무것도 없음.
“s”	문자열	char *	널로 종료되는 문자열. C 문자열 포인터가 NULL이면 None을 반환한다.
“s#”	문자열	char *, int	문자열과 길이. 널 바이트를 담을 수도 있음. C 문자열 포인터가 NULL이면 None을 반환한다.

“y”	바이트들	char *	“s” 와 같지만 바이트 문자열을 반환한다.
“y#”	바이트들	char *, int	“s#” 과 같지만 바이트 문자열을 반환한다.
“z”	문자열 또는 None	char *	“s” 와 같다.
“z#”	문자열 또는 None	char *, int	“s#” 과 같다.
“u”	유니코드	Py_UNICODE *	널로 종료되는 유니코드 문자열. 문자열 포인터가 NULL이면 None을 반환한다.
“u#”	유니코드	Py_UNICODE *, int	유니코드 문자열과 길이
“U”	유니코드	char *	널로 종료되는 C 문자열을 유니코드 문자열로 변환한다.
“U#”	유니코드	char *, int	C 문자열을 유니코드로 변환한다.
“p”	정수	char	8비트 정수
“B”	정수	unsigned char	8비트 부호 없는 정수
“h”	정수	short	짧은 16비트 정수
“H”	정수	unsigned short	부호 없는 짧은 16비트 정수
“t”	정수	int	정수
“I”	정수	unsigned int	부호 없는 정수
“T”	정수	long	긴 정수
“L”	정수	unsigned long	부호 없는 긴 정수
“K”	정수	long long	긴 긴 정수
“K”	정수	unsigned long long	부호 없는 긴 긴 정수
“n”	정수	Py_ssize_t	파이썬 크기 타입
“c”	문자열	char	단일 문자. 길이 1인 파이썬 문자열을 생성한다.
“f”	실수(float)	float	단일 정밀도 부동 소수점
“d”	실수(float)	double	배정밀도 부동 소수점
“D”	복소수	Py_complex	복소수
“O”	아무거나	PyObject *	아무 파이썬 객체. 해당 객체는 참조 횟수가 1 증가하는 것 말고는 변하지 않는다. NULL 포인터가 주어지면 NULL 포인터를 반환한다. 이 포맷은 다른 곳에서 발생한 에러를 단순히 전파하고자 할 때 유용하게 쓰인다.
“O&”	아무거나	converter, any	converter 함수로 처리되는 C 데이터
“S”	문자열	PyObject *	“O” 와 같다.
“N”	아무거나	PyObject *	“O” 와 같지만 참조 횟수가 증가하지 않는다.

“(items)”	튜플	vars	항목들의 튜플을 생성한다. items는 이 표에 있는 포맷 지정자들을 담은 문자열이다. vars는 items에 있는 항목들에 해당하는 C 변수들의 리스트이다.
“[items]”	리스트	vars	항목들의 리스트를 생성한다. items는 포맷 지정자들을 담은 문자열이다. vars는 items에 있는 항목들에 대응하는 C 변수들의 리스트이다.
“{items}”	사전	vars	항목들을 담은 사전을 생성한다.

다음은 다양한 값들을 생성하는 예를 보여준다.

```
Py_BuildValue("") None
Py_BuildValue("i",37) 37
Py_BuildValue("ids",37,3.4,"hello") (37, 3.5, "hello")
Py_BuildValue("s#", "hello", 4) "hell"
Py_BuildValue("()") ()
Py_BuildValue("(i)",37) (37,)
Py_BuildValue("[ii]",1,2) [1,2]
Py_BuildValue("[i,i]",1,2) [1,2]
Py_BuildValue("{s:i,s:i}","x",1,"y",2) {'x':1, 'y':2}
```

char \*와 관련된 유니코드 문자열 변환에서 데이터는 기본 유니코드 인코딩(보통 UTF-8)으로 인코딩된 일련의 바이트들로 구성된다고 가정한다. 해당 데이터는 파이썬에 전달할 때 알아서 유니코드로 디코딩된다. 예외적으로 “y”나 “y#” 변환은 무가공 바이트 문자열을 반환한다.

## 모듈에 값 추가

확장 모듈을 초기화하는 함수에서는 보통 상수나 기타 필요한 값을 모듈에 추가하는 작업을 수행한다. 이 일을 하는 데 다음 함수들을 사용할 수 있다.

```
int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)
```

모듈에 새 값을 추가한다. name은 값의 이름이고 value는 값을 담은 파이썬 객체이다. 같은 Py\_BuildValue()로 생성할 수 있다.

```
int PyModule_AddIntConstant(PyObject *module, const char *name, long value)
```

모듈에 정수 값을 추가한다.

```
void PyModule_AddStringConstant(PyObject *module, const char *name,
const char *value)
```

모듈에 문자열 값을 추가한다. value는 널로 종료되는 문자열이어야 한다.

```
void PyModule_AddIntMacro(PyObject *module, macro)
```

모듈에 매크로 값을 정수로 추가한다. macro는 전처리 매크로의 이름이어야 한다.

```
void PyModule_AddStringMacro(PyObject *module, macro)
```

모듈에 매크로 값을 문자열로 추가한다.

## 에러 처리

확장 모듈은 인터프리터에 NULL을 반환함으로써 에러를 알린다. NULL을 반환하기 전에 다음 함수 중에 하나를 사용해서 예외를 설정해야 한다.

```
void PyErr_NoMemory()
```

MemoryException 예외를 발생시킨다.

```
void PyErr_SetFromErrno(PyObject *exc)
```

예외를 발생시킨다. exc는 예외 객체이다. 예외 값은 C 라이브러리의 errno 변수에서 얻는다.

```
void PyErr_SetFromErrnoWithFilename(PyObject *exc, char *filename)
```

PyErr\_SetFromErrno()와 비슷하지만 예외 값에 파일 이름도 포함시킨다.

```
void PyErr_SetObject(PyObject *exc, PyObject *val)
```

예외를 발생시킨다. exc는 예외 객체이고 val은 예외 값을 담은 객체이다.

```
void PyErr_SetString(PyObject *exc, char *msg)
```

예외를 발생시킨다. exc는 예외 객체이고 msg는 무엇이 잘못되었는지를 설명하는 메시지이다.

앞의 함수들에서 exc 인수는 다음 중 하나로 설정할 수 있다.

C 이름

파이썬 예외

PyExc_ArithmeticError	ArithmetError
-----------------------	---------------

PyExc_AssertionError	AssertionError
----------------------	----------------

PyExc_AttributeError	AttributeError
----------------------	----------------

PyExc_EnvironmentError	EnvironmentError
------------------------	------------------

PyExc_EOFError	EOFError
----------------	----------

PyExc_Exception	Exception
PyExc_FloatingPointError	FloatingPointError
PyExc_ImportError	ImportError
PyExc_IndexError	IndexError
PyExc_IOError	IOError
PyExc_KeyError	KeyError
PyExc_KeyboardInterrupt	KeyboardInterrupt
PyExc_LookupError	LookupError
PyExc_MemoryError	MemoryError
PyExc_NameError	NameError
PyExc_NotImplementedError	NotImplementedError
PyExc_OSError	OSError
PyExc_OverflowError	OverflowError
PyExc_ReferenceError	ReferenceError
PyExc_RuntimeError	RuntimeError
PyExc_StandardError	StandardError
PyExc_StopIteration	StopIteration
PyExc_SyntaxError	SyntaxError
PyExc_SystemError	SystemError
PyExc_SystemExit	SystemExit
PyExc_TypeError	TypeError
PyExc_UnicodeError	UnicodeError
PyExc_UnicodeEncodeError	UnicodeEncodeError
PyExc_UnicodeDecodeError	UnicodeDecodeError
PyExc_UnicodeTranslateError	UnicodeTranslateError
PyExc_ValueError	ValueError
PyExc_WindowsError	WindowsError
PyExc_ZeroDivisionError	ZeroDivisionError

다음 함수들은 인터프리터의 예외 발생 상황을 질의하는 데 사용한다.

**void PyErr\_Clear()**

이전에 발생한 예외를 청소한다.

**PyObject \*PyErr\_Occurred()**

예외가 발생하였는지를 검사한다. 예외가 발생하였다면 현재 예외 값을 반환한다. 아니면 NULL을 반환한다.

**int PyErr\_ExceptionMatches(PyObject \*exc)**

현재 예외가 예외 exc에 매칭하는지를 검사한다. 맞으면 1을 아니면 0을 반환한다. 이 함수는 파이썬 코드에서와 동일한 예외 매칭 규칙을 따른다. 즉, exc는 현재

예외의 상위 클래스이거나 예외 클래스들의 튜플일 수 있다.

다음 코드는 C에서 try-except 블록을 구현하는 방법을 보여준다.

```
/* 파이썬 객체와 관련된 몇 가지 연산을 수행한다. */
if (PyErr_Occurred()) {
 if (PyErr_ExceptionMatches(PyExc_ValueError)) {
 /* 복구 작업을 수행한다. */
 ...
 PyErr_Clear();
 return result; /* 유효한 PyObject * */
 } else {
 return NULL; /* 예외를 인터프리터로 전파한다. */
 }
}
```

## 참조 횟수 세기

파이썬으로 작성한 프로그램과는 달리 C 확장 모듈에서는 파이썬 객체의 참조 횟수 관리를 해야 하는 경우가 있다. 다음은 참조 횟수를 관리하는 데 사용할 수 있는 전부 타입 PyObject \*인 객체를 받는 매크로들이다.

매크로	설명
Py_INCREF(obj)	널이 아니어야 하는 obj의 참조 횟수를 증가시킨다.
Py_DECREF(obj)	널이 아니어야 하는 obj의 참조 횟수를 감소시킨다.
Py_XINCREF(obj)	널일 수 있는 obj의 참조 횟수를 증가시킨다.
Py_XDECREF(obj)	널일 수 있는 obj의 참조 횟수를 감소시킨다.

C에서 파이썬 객체의 참조 횟수를 관리하는 일은 까다로운 주제 중 하나이며, 더 읽기 전에 <http://docs.python.org/extending>에 있는 ‘파이썬 인터프리터 확장과 임베딩(Extending and Embedding the Python Interpreter)’을 꼭 읽어보기 바란다. 일 반적으로 다음에 나오는 경우를 제외하고는 C 확장 함수에서 참조 횟수 세기에 관해 걱정할 필요가 없다.

- 파이썬 객체를 가리키는 참조를 나중에 사용하려고 저장하거나 C 구조체에 저장하는 경우 참조 횟수를 하나 증가시켜야 한다.
- 비슷하게, 이전에 저장한 객체를 버리기 위해서는 참조 횟수를 하나 감소시켜야 한다.
- C에서 파이썬 컨테이너(리스트, 사전 등)를 조작하는 경우에 개별 항목에 대한 참조 횟수를 직접 관리해야 할 수도 있다. 예를 들어, 컨테이너에서 항목을 얻거나 설정하는 고수준 연산에서는 보통 참조 횟수를 증가시켜야 한다.

확장 코드로 인해 인터프리터가 다운되거나(참조 횟수를 증가시키는 일은 잊어버린 것이다) 확장 함수를 사용하는 동안 인터프리터에서 메모리 누수가 발생하는 경우(참조 횟수를 감소시키는 일을 잊어버린 것이다) 참조 횟수 관련 문제가 발생하였다라는 것을 알아차릴 수 있다.

## 스레드

전역 인터프리터 락은 인터프리터에서 한 번에 하나 이상의 스레드가 실행되는 것을 막는다. 확장 모듈에 있는 함수가 오래 실행될 경우 이 함수가 종료될 때까지 다른 스레드의 실행을 막는다. 그 이유는 확장 함수가 호출될 때마다 락을 획득하기 때문이다. 확장 모듈이 스레드 안전을 보장하는 경우 다음 매크로들을 사용하면 전역 인터프리터 락을 해제하거나 다시 얻을 수 있다.

### **Py\_BEGIN\_ALLOW\_THREADS**

전역 인터프리터 락을 해제하여 다른 스레드가 인터프리터에서 실행될 수 있게 한다. C 확장 모듈은 락을 해제한 동안 파이썬 C API에 있는 함수를 호출해서는 안 된다.

### **Py\_END\_ALLOW\_THREADS**

전역 인터프리터 락을 다시 얻는다. 이 경우 확장 모듈은 락을 성공적으로 얻을 때까지 기다린다.

다음은 이 매크로들을 사용하는 예를 보여준다.

```
PyObject *py_wrapper(PyObject *self, PyObject *args) {
 ...
 PyArg_ParseTuple(args, ...)
 Py_BEGIN_ALLOW_THREADS
 result = run_long_calculation(args);
 Py_END_ALLOW_THREADS
 ...
 return Py_BuildValue(fmt, result);
}
```

## 파이썬 인터프리터 임베딩

C 응용 프로그램에 파이썬 인터프리터를 임베딩할 수 있다. 임베딩에서는 파이썬 인터프리터가 프로그래밍 라이브러리처럼 쓰여서 C 프로그램에서 인터프리터를 초기화하고 인터프리터로 스크립트나 코드 조각을 실행하고 라이브러리 모듈을 로드하며 파이썬에서 구현한 함수나 객체를 조작할 수 있다.

### 임베딩 템플릿

임베딩을 사용할 경우 C 프로그램에서 인터프리터를 관리하게 된다. 다음은 임베딩을 가능하게 하는 최소한의 코드를 보여준다.

```
#include <Python.h>

int main(int argc, char **argv) {
 Py_Initialize();
 PyRun_SimpleString("print('Hello World')");
 Py_Finalize();
 return 0;
}
```

이 예에서는 인터프리터를 초기화하고 짧은 스크립트를 문자열로 실행하고 인터프리터를 종료하였다. 더 설명하기 전에 앞에 나온 코드를 실행하는 방법을 먼저 알아보자.

### 컴파일과 링크

유닉스에서 임베딩된 인터프리터를 컴파일하려면 코드에 “Python.h” 헤더 파일을 포함시키고 libpython2.6.a 같은 인터프리터 라이브러리에 대해 링크를 수행해야 한다. 헤더 파일은 보통 /usr/local/include/python2.6에 있고 라이브러리는 /usr/local/lib/python2.6/config에 있다. 윈도에서는 파이썬 설치 디렉터리에서 이 파일들을 찾을 수 있다. 링크를 수행할 때는 인터프리터가 의존하는 다른 라이브러리를 함께 포함시켜야 할 수도 있다. 불행히도 이 부분은 플랫폼에 따라 다를 수 있고 파이썬이 어떻게 설정되어 있는지에 따라 다를 수 있다. 제대로 하려면 어느 정도 시행착오가 필요할 것이다.

### 기본 인터프리터 연산과 설정

다음 함수들은 인터프리터를 설정하고 스크립트를 실행하는 데 사용한다.

**int PyRun\_AnyFile(FILE \*fp, char \*filename)**

fp가 유닉스에서 tty 같은 대화식 기기이면 이 함수는 PyRun\_InteractiveLoop( )를 호출한다. 아니면 PyRun\_Simplefile( )을 호출한다. filename은 입력 스트림의 이름을 나타내는 문자열이다. 이 이름은 인터프리터가 에러를 출력할 때 사용하는 이름이다. filename이 NULL이면 파일 이름으로 문자열 “???”가 기본으로 사용된다.

**int PyRun\_SimpleFile(FILE \*fp, char \*filename)**

PyRun\_SimpleString( )과 비슷하지만 프로그램을 파일 fp에서 읽는다.

**int PyRun\_SimpleString(char \*command)**

인터프리터의 \_\_main\_\_ 모듈에서 command를 실행한다. 성공하면 0을, 예외가 발생하면 -1을 반환한다.

**int PyRun\_InteractiveOne(FILE \*fp, char \*filename)**

대화식 명령 하나를 실행한다.

**int PyRun\_InteractiveLoop(FILE \*fp, char \*filename)**

대화식 모드에서 인터프리터를 실행한다.

**void Py\_Initialize()**

파이썬 인터프리터를 초기화한다. 이 함수는 C API에 있는 Py\_SetProgramName( ), PyEval\_InitThreads( ), PyEval\_ReleaseLock( ) 및 PyEval\_AcquireLock( )를 제외한 다른 함수를 호출하기 전에 호출해야 한다.

**int Py\_IsInitialized()**

인터프리터가 초기화되었으면 1을, 아니면 0을 반환한다.

**void Py\_Finalize()**

Py\_Initialize( )를 호출한 이후로 생성된 모든 하위 인터프리터와 객체를 파괴하여 인터프리터를 청소한다. 보통 이 함수는 인터프리터가 할당한 모든 메모리를 해제 한다. 하지만, 순환 참조가 있다거나 확장 모듈에서 무언가를 잘못했다면 이 함수로 복구할 수 없는 메모리 누수가 생길 수 있다.

**void Py\_SetProgramName(char \*name)**

보통 sys 모듈의 argv[0] 인수에서 얻을 수 있는 프로그램 이름을 설정한다. 반드시 Py\_Initialize( )를 호출하기 전에 호출해야 한다.

**char \*Py\_GetPrefix()**

설치된 플랫폼 종속적인 파일들의 접두사를 반환한다. sys.prefix와 동일한 값을 반환한다.

**char \*Py\_GetExecPrefix()**

설치된 플랫폼 종속적인 파일들의 실행 파일 접두사를 반환한다. sys.exec\_prefix와 동일한 값을 반환한다.

**char \*Py\_GetProgramFullPath()**

파이썬 실행 파일의 전체 경로를 반환한다.

**char \*Py\_GetPath()**

기본 모듈 검색 경로를 반환한다. 이 경로는 플랫폼 종속적인 구분자(유닉스에서는 :, DOS/윈도에서는 ;)로 구분되는 디렉터리 이름들을 담은 문자열로 반환된다.

**int PySys\_SetArgv(int argc, char \*\*argv)**

sys.argv를 채우는 데 사용하는 명령줄 옵션들을 설정한다. Py\_Initialize() 전에 호출해야 한다.

## C에서 파이썬에 접근

C에서 인터프리터에 접근하는 데는 많은 방법이 있지만 임베딩과 관련해서 다음 네 가지 주요 작업이 가장 많이 이루어진다.

- 파이썬 모듈을 임포트한다(import문을 흉내 내어).
- 모듈에 정의된 객체를 가리키는 참조를 얻는다.
- 파이썬 함수, 클래스, 메서드를 호출한다.
- 객체의 속성에 접근한다(데이터, 메서드 등).

이 모든 연산들은 파이썬 C API에 정의된 다음 기본 연산들을 통해 수행된다.

**PyObject \*PyImport\_ImportModule(const char \*modname)**

모듈 modname을 임포트하고 해당 모듈 객체에 대한 참조를 반환한다.

**PyObject \*PyObject\_GetAttrString(PyObject \*obj, const char \*name)**

객체의 속성을 얻는다. obj.name과 같다.

```
int PyObject_SetAttrString(PyObject *obj, const char *name, PyObject *value)
```

객체에 속성을 설정한다. obj.name = value와 같다.

```
PyObject *PyEval_CallObject(PyObject *func, PyObject *args)
```

인수 args로 func를 호출한다. func는 호출 가능한 객체(함수, 메서드, 클래스 등)이다. args는 인수들의 튜플이다.

```
PyObject *
PyEval_CallObjectWithKeywords(PyObject *func, PyObject *args,
PyObject *kwargs)
```

위치 인수 args와 키워드 인수 kwargs로 func를 호출한다. func는 호출 가능한 객체이고 args는 튜플이며 kwargs는 사전이다.

다음은 C에서 re에 있는 다양한 부분에 접근하는 예를 통해 이 함수들을 어떻게 사용하는지를 보여준다. 이 프로그램은 stdin으로 줄들을 읽어서 사용자가 입력한 파일 정규 표현식에 매칭되는 모든 줄을 출력한다.

```
#include "Python.h"

int main(int argc, char **argv) {
 PyObject *re;
 PyObject *re_compile;
 PyObject *pat;
 PyObject *pat_search;
 PyObject *args;
 char buffer[256];

 if (argc != 2) {
 fprintf(stderr,"Usage: %s pattern\n",argv[0]);
 exit(1);
 }

 Py_Initialize();

 /* re를 임포트한다. */
 re = PyImport_ImportModule("re");

 /* pat = re.compile(pat,flags) */
 re_compile = PyObject_GetAttrString(re,"compile");
 args = Py_BuildValue("(s)", argv[1]);
 pat = PyEval_CallObject(re_compile, args);
 Py_DECREF(args);

 /* pat_search = pat.search - 메서드를 끝낸다. */
 pat_search = PyObject_GetAttrString(pat,"search");
 /* 줄들을 읽어서 매칭을 수행한다. */
 while (fgets(buffer,255,stdin)) {
 PyObject *match;
```

```

args = Py_BuildValue("(s)", buffer);

/* match = pat.search(buffer) */
match = PyEval_CallObject(pat_search,args);
Py_DECREF(args);
if (match != Py_None) {
 printf("%s",buffer);
}
Py_XDECREF(match);

}

Py_DECREF(pat);
Py_DECREF(re_compile);
Py_DECREF(re);
Py_Finalize();
return 0;
}

```

임베딩 관련 코드에서는 참조 횟수를 적절히 관리하는 일이 아주 중요하다. 특히 C에서 생성되었거나, 함수 평가 결과로 C로 반환된 객체는 참조 횟수를 감소시켜 주어야 한다.

## 파이썬 객체를 C로 변환

인터프리터를 임베딩해서 쓸 때 주요 문제는 파이썬 함수나 메서드 호출의 결과를 적절한 C 표현으로 변환하는 일이다. 일반적으로 주어진 연산이 정확히 어떤 종류의 데이터를 반환할지를 미리 알아야 한다. 불행히도 단일 객체 값은 변환하는 PyArg\_ParseTupe( ) 같은 고수준의 편리한 함수는 없다. 그렇지만 다음 목록은 여러분이 어떤 파이썬 객체로 작업을 하고 있는지를 정확하게 알 때 몇 가지 기본 파이썬 데이터 타입을 적절한 C 표현으로 변환하는 데 사용할 수 있는 저수준 변환 함수들을 보여준다.

파이썬	C 변환 함수들
long	PyInt_AsLong(PyObject *)
long	PyLong_AsLong(PyObject *)
double	PyFloat_AsDouble(PyObject *)
char	*PyString_AsString(PyObject *) (파이썬 2에만 있음)
char	*PyBytes_AsString(PyObject *) (파이썬 3에만 있음)

이보다 더 복잡한 타입을 변환하려는 경우에는 C API 문서를 참고하도록 한다 (<http://docs.python.org/c-api>).

## ctypes

ctypes 모듈은 파이썬에서 DLL과 공유 라이브러리에 정의된 함수에 접근하는 데 사용한다. 내부 C 라이브러리의 내용(이름, 호출 인수, 타입 등)을 어느 정도 알아야 하지만 ctypes를 사용하면 C 확장 래퍼 코드를 작성하거나 C 컴파일러로 컴파일하지 않고서도 C 코드에 접근할 수 있다. ctypes는 많은 고급 기능을 제공하는 꽤 규모가 큰 모듈이다. 이곳에서는 일단 시작하는 데 필요한 핵심 부분만을 다룬다.

### 공유 라이브러리 로드

다음 클래스들은 C 공유 라이브러리를 로드하여 그 내용을 나타내는 인스턴스를 반환한다.

```
CDLL(name [, mode [, handle [, use_errno [, use_last_error]]]])
```

표준 C 공유 라이브러리를 표현하는 클래스. name은 ‘libc.so.6’ 또는 ‘msvcrt.dll’ 같은 라이브러리 이름이다. mode는 라이브러리를 어떻게 로드할 것인지를 결정하는 플래그이며 유닉스에서는 내부 dlopen() 함수에 전달된다. RTLD\_LOCAL, RTLD\_GLOBAL, RTLD\_DEFAULT(기본 값)를 비트 OR한 값으로 설정한다. 윈도에서 mode는 무시된다. handle은 이미 로드된 라이브러리에 대한 핸들을 지정한다 (있는 경우). 기본으로 None이다. use\_errno는 로드된 라이브러리에서 C errno 변수를 처리할 때 보안 층을 추가하는 불리언 플래그이다. 이 층을 추가하면 외부 함수(foreign function)를 호출하기 전에 errno의 스레드 지역 복사본을 저장하고 나중에 그 값을 복구한다. 기본으로 use\_errno는 False이다. use\_last\_error는 시스템 에러 코드를 다루는 데 사용할 수 있는 두 함수 get\_last\_error()와 set\_last\_error()를 활성화하는 불리언 플래그이다. 보통 윈도에서 주로 사용된다. 기본으로 use\_last\_error는 False이다.

```
WinDLL(name [, mode [, handle [, use_errno [, use_last_error]]]])
```

CDLL()과 같지만 라이브러리에 있는 함수가 윈도의 stdcall 호출 규약을 따른다고 가정한다(윈도에서만 쓸 수 있다).

다음 유ти리티 함수는 시스템에 있는 공유 라이브러리위 위치를 파악하여 앞에서 살펴본 클래스들에서 사용하기 적절한 name 매개변수의 이름을 생성한다. ctypes, util 하위 모듈에 정의되어 있다.

**find\_library(name)**

ctypes.util에 정의되어 있다. 라이브러리 name의 경로를 반환한다. name은 'libc', 'libm'처럼 파일 접미사 없는 라이브러리 이름이다. 이 함수가 반환하는 문자열은 '/usr/lib/libc.so.6' 같은 전체 경로이다. 이 함수의 작동 방식은 시스템 종속적이고 공유 라이브러리와 환경 설정 여부(예를 들어, LD\_LIBRARY\_PATH나 기타 매개변수)에 따라 다를 수 있다. 라이브러리를 찾을 수 없으면 None을 반환한다.

**외부 함수**

CDLL( ) 클래스로 생성한 공유 라이브러리 인스턴스는 내부 C 라이브러리에 대한 대리자로서 작동한다. 라이브러리 내용에 접근하려면 간단히 속성 검색을 사용하면 된다. 다음 예를 보자.

```
>>> import ctypes
>>> libc = ctypes.CDLL("/usr/lib/libc.dylib")
>>> libc.rand()
16807
>>> libc.atoi("12345")
12345
>>>
```

이 예에서 libc.rand( )나 libc.atoi( ) 같은 연산은 로드된 C 라이브러리에 있는 함수를 직접 호출한다.

ctypes는 모든 함수가 타입이 int나 char \*인 매개변수를 받고 타입이 int인 결과를 반환한다고 가정한다. 따라서, 앞선 예에서 함수 호출이 제대로 이루어졌음에도 불구하고 다른 라이브러리 함수를 호출하면 기대했던 대로 작동하지 않을 수 있다. 다음 예를 보자.

```
>>> libc.atof("34.5")
-1073746168
>>>
```

이 문제를 해결하기 위해 다음 속성을 변경하면 외부 함수 func의 타입 시그너처와 처리 방식을 설정할 수 있다.

**func.argtypes**

func에 대한 입력 인수들을 설명하는 ctypes 데이터 타입(곧 설명한다)들의 튜플.

**func.restype**

func의 반환 값을 설명하는 ctypes 데이터 타입. void를 반환하는 함수에 대해서

는 None.

### **func.errcheck**

세 가지 매개변수 (result, func, args)를 받는 파이썬의 호출 가능한 객체. result는 외부 함수가 반환하는 값을 나타내고 func는 외부 함수 자체를 가리키는 참조이며 args는 입력 인수들의 튜플이다. 이 함수는 외부 함수 호출 이후에 호출되며 에러 검사나 기타 다른 작업을 수행하는 데 사용할 수 있다.

다음 예는 이전 예에서 보았던 atof() 함수 인터페이스와 관련된 문제를 해결한다.

```
>>> libc.atof.restype=ctypes.c_double
>>> libc.atof("34.5")
34.5
>>>
```

ctypes.c\_double은 미리 정의된 데이터 타입을 가리키는 참조이다. 다음 절에서 데이터 타입에 관해 설명한다.

## **데이터 타입**

표 26.5는 외부 함수의 argtypes와 restype 속성에 사용할 수 있는 데이터 타입을 보여준다. “파이썬 값” 열은 주어진 데이터 타입에 대해서 허용되는 파이썬 데이터의 타입을 기술한다.

ctypes 타입 이름	C 데이터 타입	파이썬 값
c_bool	bool	True나 False
c_bytes	signed char	작은 정수
c_char	char	단일 문자
c_char_p	char *	널로 종료되는 문자열이나 바이트들
c_double	double	부동 소수점
c_longdouble	long double	부동 소수점
c_float	float	부동 소수점
c_int	int	정수
c_int8	signed char	8비트 정수
c_int16	short	16비트 정수
c_int32	int	32비트 정수
c_int64	long long	64비트 정수
c_long	long	정수
c_longlong	long long	정수
c_short	short	정수
c_size_t	size_t	정수

c_ubyte	unsigned char	부호 없는 정수
c_uint	unsigned int	부호 없는 정수
c_uint8	unsigned char	8비트 부호 없는 정수
c_uint16	unsigned short	16비트 부호 없는 정수
c_uint32	unsigned int	32비트 부호 없는 정수
c_uint64	unsigned long long	64비트 부호 없는 정수
c_ulong	unsigned long	부호 없는 정수
c_ulonglong	unsigned long long	부호 없는 정수
c_ushort	unsigned short	부호 없는 정수
c_void_p	void *	정수
c_wchar	wchar_t	단일 유니코드 문자
c_wchar_p	wchar_t *	널로 종료되는 유니코드

C 포인터를 나타내는 타입을 생성하려면 타입에 다음 함수를 적용하면 된다.

### POINTER(type)

타입 type을 가리키는 포인터인 타입을 정의한다. 예를 들어, POINTER(c\_int)는 C 타입 int \*를 나타낸다.

고정 크기 C 배열을 나타내는 타입을 정의하려면 기존 타입에 배열 차원 수를 곱하면 된다. 예를 들어, c\_int\*4는 C 데이터 타입 int[4]를 나타낸다.

C 구조체(struct)나 유니온(union)을 나타내는 타입을 정의하려면 기반 클래스 Structure나 Union에서 상속하면 된다. 그리고 나서 파생 클래스에서 내용을 설명하는 클래스 변수 \_fields\_를 정의한다. \_fields\_는 2 개짜리 항목 튜플 (name, ctype) 또는 3 개짜리 항목 튜플 (name, ctype, width)들의 리스트이다. 여기서 name은 구조체의 필드 식별자이고 ctype는 타입을 설명하는 ctypes 클래스이며 width는 정수 비트 필드 폭이다. 예를 들어, 다음 C 구조체를 보자.

```
struct Point {
 double x, y;
};
```

이 구조체를 ctypes로 설명하면 다음과 같다.

```
class Point(Structure):
 fields = [("x", c_double),
 ("y", c_double)]
```

### 외부 함수 호출

라이브러리에 있는 함수를 호출할 때는 단순히 해당 함수를 타입 시그너처에 맞는 인수들로 호출하면 된다. c\_int, c\_double 같은 간단한 데이터 타입에 대해서는 간

단히 호환되는 파이썬 타입을 입력으로 넘기면 된다(정수, 실수 등). 입력으로 c\_int, c\_double 또는 비슷한 타입의 인스턴스를 넘겨도 된다. 배열에 대해서는 호환되는 타입들의 파이썬 순서열을 넘기면 된다.

외부 함수에 포인터를 넘기려면 먼저 포인터가 가리킬 값을 나타내는 ctypes 인스턴스를 생성하고 그 다음에 다음 함수 중 하나를 사용해서 포인터 객체를 생성해야 한다.

#### **byref(cvalue [, offset])**

cvalue를 가리키는 경량 포인터를 나타낸다. cvalue는 반드시 ctypes 데이터 타입의 인스턴스이어야 한다. offset은 포인터 값에 더할 바이트 오프셋이다. 이 함수가 반환하는 값은 함수를 호출할 때만 사용할 수 있다.

#### **pointer(cvalue)**

cvalue를 가리키는 포인터 인스턴스를 생성한다. cvalue는 반드시 ctypes 데이터 타입의 인스턴스이어야 한다. 이 함수는 앞에서 설명한 POINTER 타입의 인스턴스를 생성한다.

다음 예는 타입이 double \*인 매개변수를 C 함수에 전달하는 방법을 보여준다.

```
dval = c_double(0.0) # double 인스턴스를 생성한다.
r = foo(byref(dval)) # foo(&dval)을 호출한다.

p_dval = pointer(dval) # 포인터 변수를 생성한다.
r = foo(p_dval) # foo(p_dval)을 호출한다.

dval의 값을 확인한다.
print (dval.value)
```

int나 float 같은 내장 타입을 가리키는 포인터는 생성할 수 없다. 그런 타입을 가리키는 포인터를 전달하면 내부 C 함수가 값을 수정할 경우 불변성을 위반할 수 있기 때문이다.

ctypes 인스턴스 cobj의 cobj.value 속성은 내부 데이터를 담는다. 예를 들어, 앞에서 dval.value는 ctypes c\_double 인스턴스 dval에 저장된 부동 소수점 값을 반환한다.

C 함수에 구조체를 전달하려면 구조체나 유니온 인스턴스를 먼저 생성해야 한다. 이를 위해 미리 정의한 구조체나 유니온 타입 StructureType을 다음과 같이 호출하면 된다.

**StructureType(\*args, \*\*kwargs)**

StructureType 인스턴스를 생성한다. StructureType은 Structure나 Union에서 상속한 클래스이다. \*args에 있는 위치 인수들은 \_fields\_에 나열된 순서로 구조체 멤버들을 초기화하는 데 사용한다. \*\*kwargs에 있는 키워드 인수들은 이름 있는 구조체 멤버들을 초기화한다.

**기타 타입 생성 메서드들**

c\_int, POINTER 등 모든 ctypes 타입의 인스턴스는 메모리 위치 또는 다른 객체로부터 ctypes 타입의 인스턴스를 생성하는 클래스 메서드들을 갖는다.

**ty.from\_buffer(source [,offset])**

source와 동일한 메모리 베퍼를 공유하는 ctype 타입 ty의 인스턴스를 생성한다. source는 쓰기 가능한 베퍼 인터페이스(예를 들어, bytearray, array 모듈에 있는 array 객체, mmap 등)를 지원하는 객체이어야 한다. offset은 베퍼 시작 부분에서부터 사용할 바이트 수를 나타낸다.

**ty.from\_buffer\_copy(source [, offset])**

ty.from\_buffer()와 같지만 메모리 복사본이 생성되며 source는 읽기 전용일 수 있다.

**ty.from\_address(address)**

정수로 지정하는 무가공 메모리 주소 address로부터 ctypes 타입 ty의 인스턴스를 생성한다.

**ty.from\_param(obj)**

파이썬 객체 obj에서 ctypes 타입 ty의 인스턴스를 생성한다. 전달된 객체 obj를 적절한 타입으로 변환할 수 있는 경우에만 제대로 작동한다. 예를 들어, 파이썬 정수는 c\_int 인스턴스로 변환할 수 있다.

**ty.in\_dll(library, name)**

공유 라이브러리에 있는 기호에서 ctypes 타입인 ty의 인스턴스를 생성한다. library는 CDLL 클래스로 생성한 객체처럼 로드된 라이브러리 인스턴스이다. name은 기호 이름이다. 이 메서드는 라이브러리에 정의된 전역 변수들을 감싸는 ctypes 래퍼를 생성하는 데 사용할 수 있다.

다음 예는 라이브러리 libexample.so에 정의된 변수 int status를 가리키는 참조를 생성하는 방법을 보여준다.

```
libexample = ctypes.CDLL("libexample.so")
status = ctypes.c_int.in_dll(libexample, "status")
```

## 유ти리티 함수

다음은 ctypes에 정의된 유ти리티 함수들이다.

### **addressof(cobj)**

cobj의 메모리 주소를 정수로 반환한다. cobj는 ctypes 타입의 인스턴스이어야 한다.

### **alignment(ctype\_or\_obj)**

ctypes 타입이나 객체의 정수 정렬 요구 사항들을 반환한다. ctype\_or\_obj는 ctypes의 타입이거나 타입의 인스턴스이어야 한다.

### **cast(cobj, ctype)**

ctypes 객체 cobj를 ctype에 있는 새로운 타입으로 변환한다. 포인터에 대해서만 작동하기 때문에 cobj는 포인터이거나 배열이어야 하고 ctype은 포인터 타입이어야 한다.

### **create\_string\_buffer(init [, size])**

변경 가능한 문자 버퍼를 타입 c\_char 배열로 반환한다. init는 정수 크기 또는 초기 내용을 담은 문자열이다. size는 init가 문자열일 때 사용할 크기를 지정하는 옵션인 매개변수이다. 기본으로 크기는 init에 있는 문자 개수보다 하나 크게 설정된다. 유니코드 문자열은 기본 인코딩을 사용해서 바이트들로 인코딩된다.

### **create\_unicode\_buffer(init [, size])**

create\_string\_buffer()와 같지만 타입 c\_wchar 배열이 생성된다.

### **get\_errno()**

errno의 ctypes 개인 복사본의 현재 값을 반환한다.

### **get\_last\_error()**

윈도에서 GetLastError의 ctypes 개인 복사본의 현재 값을 반환한다.

### **memmove(dst, src, count)**

count 바이트만큼을 src에서 dst로 복사한다. src와 dst는 메모리 주소를 나

타내는 정수이거나 포인터로 변환할 수 있는 ctypes 타입의 인스턴스이다. C의 memmove( ) 라이브러리 함수와 같다.

**memset(dst, c, count)**

dst에서 시작하는 메모리에서 count 바이트만큼을 바이트 값 c로 채운다. dst는 정수이거나 ctypes 인스턴스이다. c는 범위 0-255 안에 있는 바이트를 나타내는 정수이다.

**resize(cobj, size)**

ctypes 객체 cobj를 나타내는 데 사용되는 내부 메모리 크기를 조정한다. size는 새로운 크기를 바이트로 나타낸 것이다.

**set\_conversion\_mode(encoding, errors)**

유니코드 문자열을 8비트 문자열로 변환하는 데 사용할 유니코드 인코딩을 설정한다. encoding은 ‘utf-8’ 같은 인코딩 이름이고 errors는 ‘strict’나 ‘ignore’ 같은 에러 처리 정책이다. 이전 설정을 담은 튜플 (encoding, errors)를 반환한다.

**set\_errno(value)**

시스템 errno 변수의 ctypes 개인 복사본을 설정한다. 이전 값을 반환한다.

**set\_last\_error(value)**

윈도의 LastError 변수를 설정하고 이전 값을 반환한다.

**sizeof(type\_or\_cobj)**

ctypes 타입 또는 객체의 크기를 바이트로 반환한다.

**string\_at(address [, size])**

주소 address에서 시작하는 메모리에 있는 size 바이트만큼을 나타내는 바이트 문자열을 반환한다. size를 생략하면 바이트 문자열이 NULL로 종료된다고 가정한다.

**wstring\_at(address [, size])**

주소 address에서 시작하는 size만큼의 넓은 문자(wide character)들을 나타내는 유니코드 문자열을 반환한다. size를 생략하면 문자열이 NULL로 종료된다고 가정한다.

## 예

다음 예는 이 장의 시작 부분에서 파이썬 확장 모듈을 직접 생성하는 방법을 다룰 때 사용했던 C 함수들에 대한 인터페이스를 구축하는 데 ctypes 모듈을 사용하는 예를 보여준다.

```
example.py

import ctypes
_example = ctypes.CDLL("./libexample.so")

int gcd(int, int)
gcd = _example.gcd
gcd.argtypes = (ctypes.c_int,
 ctypes.c_int)
gcd.restype = ctypes.c_int

int replace(char *s, char olcdh, char newch)
_example.replace.argtypes = (ctypes.c_char_p,
 ctypes.c_char,
 ctypes.c_char)
_example.replace.restype = ctypes.c_int

def replace(s, oldch, newch):
 sbuffer = ctypes.create_string_buffer(s)
 nrep = _example.replace(sbuffer,oldch,newch)
 return (nrep,sbuffer.value)

double distance(Point *p1, Point *p2)
class Point(ctypes.Structure):
 fields = [("x", ctypes.c_double),
 ("y", ctypes.c_double)]

_example.distance.argtypes = (ctypes.POINTER(Point),
 ctypes.POINTER(Point))
_example.distance.restype = ctypes.c_double

def distance(a,b):
 p1 = Point(*a)
 p2 = Point(*b)
 return _example.distance(byref(p1),byref(p2))
```

보통 ctypes을 사용할 때는 다양한 수준의 복잡도를 가지는 파이썬 래퍼 층을 작성하게 된다. 예를 들어, C 함수를 직접 호출할 수도 있다. 하지만 내부 C 코드의 특별한 부분을 다루기 위해서 작은 래퍼 층을 구현해야 하는 경우도 있다. 앞의 예에서는 C 라이브러리가 입력 버퍼를 수정한다는 사실을 고려하여 replace() 함수에서 추가 단계를 거치게 하였다. distance() 함수는 튜플에서 Point 인스턴스를 생성하고 포인터를 전달하기 위해 추가 단계를 수행하였다.

**Note**

ctypes 모듈은 이곳에서 다루지 않은 많은 고급 기능을 제공한다. 예를 들어, 윈도에서 다양한 종류의 라이브러리에 접근할 수 있으며 역호출 함수, 불완전 타입 등도 지원한다. 온라인 문서에 많은 예가 있으므로 고급 기능을 활용하고자 할 때 참고하면 도움이 될 것이다.

## 고급 확장과 임베딩

파이썬을 간단한 C 코드로 확장하는 경우에는 직접 확장 모듈을 손으로 작성하거나 ctypes을 사용하는 일이 어렵지 않다. 하지만 더 복잡한 코드에 대해서는 꽤 번거로울 수 있다. 이런 경우 적절한 확장 모듈 생성 도구가 있으면 좋을 것이다. 이런 도구들은 확장 모듈 생성 과정의 많은 부분을 자동화하고 훨씬 고수준에서 작업할 수 있는 프로그래밍 인터페이스를 제공한다. <http://wiki.python.org/moin/IntegratingPythonWithOtherLanguages>에 가면 다양한 도구들에 대한 링크를 찾을 수 있을 것이다. 이곳에서는 맛보기로 SWIG(<http://www.swig.org>)를 사용하는 간단한 예를 살펴본다. 솔직히 말하면 SWIG는 원래 필자가 처음 제작한 것이다.

자동화된 도구를 사용할 때는 보통 간단히 확장 모듈의 내용을 고수준에서 기술한다. 예를 들어, SWIG를 사용한다면 다음과 같은 짧은 인터페이스 명세서를 작성한다.

```
/* example.i : SWIG 명세서 샘플 */
%module example
%{
/* 도입부. 필요한 모든 헤더 파일을 이곳에 둔다. */
#include "example.h"
%}

/* 모듈 내용. 모든 C 선언을 이곳에 둔다. */
typedef struct Point {
 double x;
 double y;
} Point;
extern int gcd(int, int);
extern int replace(char *, char oldch, char newch);
extern double distance(Point *a, Point *b);
```

이 명세서를 바탕으로 하여 SWIG는 파이썬 확장 모듈을 생성하는 데 필요한 모든 것을 만들어낸다. SWIG를 실행하기 위해 다음과 같이 SWIG를 컴파일러처럼 호출한다.

```
% swig -python example.i
%
```

SWIG는 출력으로 .c와 .py 파일들을 생성한다. 보통은 이런 부분에 관해 신경쓸 필요가 없다. distutils를 사용하고 있다면 setup.py 파일에 .i 파일을 포함시키면 확장 모듈을 생성할 때 자동으로 SWIG가 실행된다. 예를 들어, 다음 setup.py 파일은 example.i 파일을 입력으로 SWIG를 자동으로 실행한다.

```
setup.py
from distutils.core import setup, Extension
setup(name="example",
 version="1.0",
 py_modules = ['example.py'],
 ext_modules = [
 Extension("_example",
 ["example.i","example.c"])
]
)
```

이 예에서 example.i 파일과 setup.py 파일이 제대로 작동하는 확장 모듈을 만드는 데 필요한 전부다. python setup.py build\_ext -inplace를 입력하면 완전히 작동하는 확장 모듈이 생성된다.

## Jython과 IronPython

확장과 임베딩은 C 프로그램에만 국한되지 않는다. 자바로 작업하고 있다면 파이썬 인터프리터를 자바로 완전히 새로 구현한 Jython을 사용하는 것을 고려해보도록 한다(<http://www.jython.org>). jython을 사용하면 간단히 import문으로 자바라이브러리를 임포트할 수 있다. 다음 예를 보자.

```
bash-3.2$ jython
Jython 2.2.1 on java1.5.0_16
Type "copyright", "credits" or "license" for more information.
>>> from java.lang import System
>>> System.out.println("Hello World")
Hello World
>>>
```

윈도에서 .NET 프레임워크를 사용하고 있다면 파이썬 인터프리터를 C#으로 완전히 새로 구현한 IronPython의 사용을 고려해보라(<http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython>). IronPython을 사용하면 비슷한 방식으로 파이썬에서 모든 .NET 라이브러리에 쉽게 접근할 수 있다. 다음 예를 보자.

```
% ipy
IronPython 1.1.2 (1.1.2) on .NET 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
>>> import System.Math
>>> dir(System.Math)
['Abs', 'Acos', 'Asin', 'Atan', 'Atan2', 'BigMul', 'Ceiling', 'Cos', 'Cosh', ...]
>>> System.Math.Cos(3)
-0.9899924966
>>>
```

jython과 IronPython에 관해 자세하게 다루는 일은 이 책의 범위를 벗어난다. 그렇지만 둘 다 파이썬이라는 것을 기억하기 바란다. 가장 큰 차이점은 지원하는 라이브러리에 있다.

---

## 부록

P y t h o n   E s s e n t i a l   R e f e r e n c e

---

# 파이썬 3

---

2008년 12월에 파이썬 3.0이 릴리스되었다. 파이썬 3는 여러 가지 주요 영역에서 파이썬 2와 하위 호환성이 없는 대규모 업데이트 버전이다. <http://docs.python.org/3.0/whatsnew/3.0.html>에 있는 ‘파이썬 3.0에서 새로운 내용(What’s New in Python 3.0)’ 문서에 파이썬 3에서 변화된 부분이 잘 요약되어 있다. 어떤 면에서 보자면 이 책은 ‘새로운 내용’ 문서와 극을 달린다고 볼 수 있다. 즉, 지금까지 다루었던 모든 내용은 파이썬 2와 파이썬 3에서 함께 공유하는 기능에 초점을 맞추었다. 여기에는 표준 라이브러리 모듈, 주요 언어 기능, 예 등이 포함된다. 몇몇 사소한 이름 변화와 `print()`가 함수라는 점을 제외하고는 지금까지 파이썬 3에만 있는 기능은 다루지 않았다.

부록에서는 파이썬 3에만 있는 파이썬 언어의 새로운 기능과 기존 코드를 옮길 때 명심해야 하는 두 버전 사이의 몇 가지 중요한 차이점을 중점적으로 다룬다. 끝 부분에서는 포팅 전략과 2 to 3 코드 변환 도구를 사용하는 방법을 설명한다.

### 누가 파이썬 3를 사용해야 하나?

더 나가기 전에, 먼저 누가 파이썬 3.0 릴리스를 사용하게 될 것인가라는 질문에 대답할 필요가 있다. 파이썬 커뮤니티에서는 파이썬 3로의 전환이 하루아침에 이루어 지지 않을 것이고 파이썬 2도 앞으로 어느 정도(몇 년 동안)는 계속 사용될 것이라는 생각을 해왔다. 이 책을 쓰고 있는 지금, 파이썬 2 코드를 버려야 할 시급한 이유는 없다. 필자는 이 책의 5판이 쓰여질 몇 년 후에도 여전히 많은 코드가 파이썬 2로

개발되고 있을 것이라고 생각한다.

파이썬 3가 직면하고 있는 주요 문제는 써드 파티 라이브러리와의 호환성과 관련 있다. 파이썬의 강력함은 대부분 다양한 프레임워크와 라이브러리에서 나온다. 하지만 이 라이브러리들은 파이썬 3로 확실히 포팅되지 않을 경우 제대로 작동하지 않을 것이 거의 확실하다. 이 문제는 많은 라이브러리가 다른 라이브러리, 그리고 또 다른 라이브러리에 의존하다는 점 때문에 더 심각하다. 이 책을 쓰고 있는 지금(2009년), 파이썬 2.6 또는 3.0은 제쳐두고서라도 아직 파이썬 2.4로도 제대로 포팅되지 않은 주요 라이브러리와 프레임워크들이 있다. 따라서, 여러분이 써드 파티 코드를 사용하려는 생각으로 파이썬을 사용하고 있다면 당분간 파이썬 2를 계속 쓰는 것이 좋을 것이다. 여러분이 이 책을 집어 들었고 2012년이 되었다면 상황이 더 나아졌기를 바랄 뿐이다.

파이썬 3에서 언어의 불필요한 부분이 많이 정리되기는 했지만 여전히 기본을 배우려는 새로운 사용자에게 적합한지는 불명확하다. 대부분의 기존 문서, 튜토리얼, 쿡북, 예 등이 파이썬 2를 가정하여 쓰여 있고 코딩 규약도 서로 호환되지 않는다. 말할 것도 없이, 모든 것이 제대로 작동하지 않는다면 긍정적인 교육을 경험하기 어려울 것이다. 심지어 공식 문서조차도 파이썬 3의 코딩 규약에 맞게 업데이트되지 않은 부분이 있을 정도이다. 이 책을 쓰면서 필자는 파이썬 3 문서에 있는 오류와 빠뜨린 부분 등 수많은 버그를 발견하여 신고하였다.

마지막으로 파이썬 3가 가장 최신의 것이고 가장 멋진 것이라고 설명하고 있지만 여전히 성능과 작동 방식에 문제가 산재해 있다. 예를 들어, 초기 릴리스에서는 I/O 시스템 쪽이 정말 경악할 만큼 받아들일 수 없는 런타임 성능을 보였다. 바이트들과 유니코드를 분리한 것도 문제를 일으킬 수 있다. 심지어 몇몇 내장 라이브러리 모듈은 I/O 및 문자열 처리와 관련한 변화 때문에 제대로 작동하지 않는다. 물론 이런 문제들은 시간이 지나면서 더 많은 프로그래머들이 릴리스에 대해 테스트를 많이 수행함에 따라서 나아질 것이다. 하지만 필자의 의견으로는 현재 파이썬 3는 파이썬 전문가들이나 실험적으로 사용하기에 적합한 상태라고 생각한다. 안정적이고 제품 품질의 코드를 원한다면 파이썬 3에서 문제들이 해결될 수 있는 시간 동안은 파이썬 2를 계속 사용할 것을 권한다.

## 새로운 언어 기능

여기에서는 파이썬 2에서는 지원되지 않는 파이썬 3에 있는 몇 가지 기능을 간단히 살펴본다.

### 소스 코드 인코딩과 식별자

파이썬 3에서는 소스 코드가 UTF-8로 인코딩된 것을 가정한다. 또한 식별자에 나타날 수 있는 문자에 대한 규칙도 완화되었다. 더 구체적으로는, 식별자는 코드 포인트 U+0080 이상 되는 어떠한 유효한 유니코드 문자도 담을 수 있다. 다음 예를 보자.

```
π = 3.141592654
r = 4.0
print(2*π**r)
```

코드에 이런 문자를 사용할 수 있다고 해서 실제로 사용하는 것은 좋은 생각이 아니다. 모든 편집기, 터미널, 개발 도구에서 똑같이 유니코드를 잘 처리하는 것은 아니기 때문이다. 또한 표준 키보드에서 볼 수 없는 문자를 입력하느라 불편하게 키들을 입력해야 한다면 프로그래머들은 짜증을 낼 것이다(사무실에서 흔 수염을 기른 몇몇 해커가 모든 이에게 API이라는 언어에 대한 이야기를 즐겁게 늘어놓게 만들지도 모른다). 이런 이유 때문에 주석이나 문자열 상수에는 유니코드 문자를 사용하지 않는 것이 좋다.

### 집합 상수

항목들의 집합을 이제 값들을 대괄호로 둘러싸으로써 정의할 수 있다. 다음 예를 보자.

```
days = { 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun' }
```

이 문법은 다음에 나오는 set( ) 함수를 사용하는 것과 같다.

```
days = set(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
```

### 집합과 사전 내포

문법 { expr for x in s if condition }은 집합 내포를 나타낸다. 이렇게 하면 집합 s에 있는 모든 원소에 대해 연산을 수행하며 리스트 내포와 비슷하게 사용할 수 있다.

다음 예를 보자.

```
>>> values = { 1, 2, 3, 4 }
>>> squares = {x*x for x in values}
>>> squares
{16, 1, 4, 9}
>>>
```

문법 { kexpr:vexpr for k,v in s if condition }은 사전 내포를 나타낸다. 이렇게 하면 (key, value) 튜플들의 순서열 s에 있는 모든 키워드 값에 대해 연산을 수행하고 사전을 반환한다. 새로운 사전의 키는 kexpr 표현식으로 기술하고 값은 vexpr 표현식으로 기술한다. 이 문법을 dict() 함수의 더 강력한 버전쯤으로 볼 수도 있다.

설명을 위해 다음과 같이 주식 가격을 담은 파일 ‘prices.dat’이 있다고 하자.

```
GOOG 509.71
YHOO 28.34
IBM 106.11
MSFT 30.47
AAPL 122.13
```

다음은 사전 내포를 사용해서 이 파일을 읽고 주식 이름을 가격으로 매핑하는 사전을 만드는 코드이다.

```
fields = (line.split() for line in open("prices.dat"))
prices = {sym:float(val) for sym,val in fields}
```

다음은 사전에 있는 모든 키를 소문자로 바꾸는 예이다.

```
d = {sym.lower():price for sym,price in prices.items()}
```

다음은 가격이 100달러가 넘는 주식을 담은 사전을 만드는 예이다.

```
d = {sym:price for sym,price in prices.items() if price >= 100.0}
```

## 확장된 반복 가능한 풀어헤치기

파이썬 2에서 반복 가능한 객체에 있는 항목들은 다음과 같은 문법을 사용해서 변수들로 풀어헤칠 수 있었다.

```
items = [1,2,3,4]
a,b,c,d = items # 항목들을 변수들로 풀어헤친다.
```

이 풀어헤치기가 성공하려면 변수와 항목 개수가 정확히 일치해야 한다.

파이썬 3에서는 와일드카드 변수를 사용해서 순서열에 있는 항목들 중 몇 개만 풀어헤치고 나머지는 리스트에 넣을 수 있다. 다음 예를 보자.

```
a,*rest = items # a = 1, rest = [2,3,4]
a,*rest,d = items # a = 1, rest = [2,3], d = 4
*rest, d = items # rest = [1,2,3], d = 4
```

이 예에서 앞에 \*가 붙은 변수는 나머지 값들을 받아서 리스트에 담는다. 남은 항목이 없으면 리스트가 빌 수도 있다. 튜플(또는 순서열)들의 리스트에 대해 루프를 도는데 튜플들의 크기가 다를 수 있는 경우 이 기능을 사용할 수 있다. 다음 예를 보자.

```
points = [(1,2), (3,4,"red"), (4,5,"blue"), (6,7)]
for x,y, *opt in points:
 if opt:
 # 필드가 추가로 발견되었다.
문장들
```

풀어헤치기를 할 때 별표가 붙은 변수는 하나만 나올 수 있다.

## nonlocal 변수

nonlocal 선언을 사용해서 내부 함수에서 외부 함수에 있는 변수를 수정할 수 있다. 다음 예를 보자.

```
def countdown(n):
 def decrement():
 nonlocal n
 n -= 1
 while n > 0:
 print("T-minus", n)
 decrement()
```

파이썬 2에서는 내부 함수에서 외부 함수의 변수를 읽을 수 있지만 수정할 수는 없었다. nonlocal 선언은 이것을 가능하게 한다.

## 함수 주석

함수의 인수와 반환 값에 아무 값으로나 주석을 달 수 있다. 다음 예를 보자.

```
def foo(x:1,y:2) -> 3:
 pass
```

함수 속성인 `__annotations__`는 인수 이름을 주석 값으로 매핑하는 사전이다. 특수한 ‘return’ 키가 반환 값 주석에 대한 매핑을 담는다. 다음 예를 보자.

```
>>> foo.__annotations__
{'y': 4, 'x': 3, 'return': 5}
>>>
```

인터프리터는 주석에 특별한 의미를 부여하지 않는다. 사실 주석으로 어떤 값이 든 사용할 수 있다. 그래도 나중을 위해서 타입 정보를 붙이는 것이 가장 유용할 것이다. 다음 예를 보자.

```
def foo(x:int, y:int) -> str:
 문장들
```

주석으로 반드시 값 하나만 사용할 수 있는 것은 아니다. 어떤 유효한 파이썬 표현식이라도 주석으로 사용할 수 있다. 가변 개수 위치 인수와 키워드 인수에도 같은 문법을 사용한다. 다음 예를 보자.

```
def bar(x, *args:"additional", **kwargs:"options"):
 문장들
```

다시 한 번 강조하지만 파이썬은 주석에 아무런 의미도 부여하지 않는다. 주석은 써드 파티 라이브러리나 프레임워크에서 메타프로그래밍과 관련한 다양한 응용에 사용할 수 있게 제공되는 기능이다. 주석은 정적 분석 도구, 문서화, 테스트, 함수 오버로딩, 마샬링, 원격 프로시저 호출, IDE, 계약 등 다양한 용도로 쓸 수 있지만 꼭 이런 곳에만 쓸 수 있는 것은 아니다. 다음은 함수 인수와 반환 값을 검사하는 장식자 함수의 예를 보여준다.

```
def ensure(func):
 # 주석 데이터를 추출한다.
 return_check = func.__annotations__.get('return',None)
 arg_checks = [(name,func.__annotations__.get(name))
 for name in func.__code__.co_varnames]

 # 주석으로 지정한 함수를 사용해서 인수 값과 반환 값을 검사하는
 # 래퍼를 생성한다.

 def assert_call(*args,**kwargs):
 for (name,check),value in zip(arg_checks,args):
 if check: assert check(value), "%s %s" % (name, check.__doc__)
 for name,check in arg_checks[len(args):]:
 if check: assert check(kwargs[name]), "%s %s" % (name, check.__doc__)
 result = func(*args,**kwargs)
 assert return_check(result), "return %s" % return_check.__doc__
 return result

 return assert_call
```

다음은 앞에 나온 장식자를 사용하는 코드를 보여준다.

```

def positive(x):
 "must be positive"
 return x > 0

def negative(x):
 "must be negative"
 return x < 0

@ensure
def foo(a:positive, b:negative) -> positive:
 return a - b

```

다음은 이 함수를 사용했을 때 나올 수 있는 출력 결과의 예를 보여준다.

```

>>> foo(3,-2)
5
>>> foo(-5,2)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "meta.py", line 19, in call
 def assert_call(*args,**kwargs):
AssertionError: a must be positive
>>>

```

## 키워드 전용 인수

함수는 키워드 전용 인수(keyword-only argument)를 가질 수 있다. 첫 번째 별표 표시된 매개변수 다음에 추가 매개변수를 정의하면 된다. 다음 예를 보자.

```

def foo(x, *args, strict=False):
 문장들

```

이 함수를 호출할 때 strict 매개변수는 키워드로만 지정할 수 있다. 다음 예를 보자.

```
a = foo(1, strict=True)
```

추가 위치 인수는 strict의 값을 설정하는 데 사용되지 않고 args에 추가된다. 가변 개수 인수를 받지 않으면서 키워드 전용 인수를 받고 싶다면 매개변수 목록에 \*를 홀로 사용하면 된다. 다음 예를 보자.

```

def foo(x, *, strict=False):
 문장들

```

다음은 사용법을 보여준다.

```

foo(1,True) # 실패한다. TypeError: foo()는 위치 인수 하나만 받는다.
foo(1,strict=True) # 문제 없다.

```

## 표현식인 Ellipsis

`Ellipsis` 객체(...)는 이제 표현식으로서 나타날 수 있다. 이 때문에 이 객체를 컨테이너에 넣거나 변수에 대입하는 일이 가능하다. 다음 예를 보자.

```
>>> x = ... # Ellipsis 대입
>>> x
Ellipsis
>>> a = [1, 2, ...]
>>> a
[1, 2, Ellipsis]
>>> ... in a
True
>>> x is ...
True
>>>
```

`Ellipsis` 객체를 해석하는 일은 이 객체를 사용하는 응용에 따라 달라질 수 있다. 라이브러리나 프레임워크를 제작할 때는 이 문법을 흥미로운 방식으로 사용할 수 있다(예를 들면 와일드카드, 계속됨 또는 기타 개념을 나타내는 데 사용할 수 있을 것이다).

## 연쇄 예외

예외는 이제 서로 묶일 수 있다. 이렇게 하면 현재 예외에 이전 예외에 대한 정보를 실을 수 있다. 명시적으로 예외를 묶으려면 `raise`문에서 `from` 한정어를 사용하면 된다. 다음 예를 보자.

```
try:
 문장들
except ValueError as e:
 raise SyntaxError("Couldn't parse configuration") from e
```

`SyntaxError` 예외가 발생할 때 다음과 같이 두 예외를 다 가진 역추적 메시지를 보게 된다.

```
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'nine'

The above exception was the direct cause of the following exception:
```

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
SyntaxError: Couldn't parse configuration
```

예외 객체에는 이전 예외를 저장하는 `__cause__` 속성이 있다. `raise`와 `from` 한정

어를 함께 사용하면 이 속성에 값이 설정된다.

조금 더 복잡한 예외 연결 예로 다른 예외 처리기에서 발생한 예외와 관련한 예가 있다. 다음 예를 보자.

```
def error(msg):
 print(m) # 참고: 일부러 오자를 입력(m이 정의되지 않음)

try:
 문장들
except ValueError as e:
 error("Couldn't parse configuration")
```

파이썬 2에서 이 코드를 실행하면 error()에서 NameError와 관련된 예외만 보게 된다. 파이썬 3에서는 처리 중인 이전 예외가 함께 전달된다. 예를 들어, 다음과 같은 메시지를 보게 된다.

```
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'nine'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 File "<stdin>", line 2, in error
NameError: global name 'm' is not defined
```

임묵적인 예외 연결이 일어났을 때는 예외 인스턴스 e의 `__context__` 속성이 이전 예외에 대한 참조를 담게 된다.

## 향상된 `super()`

기반 클래스에서 메서드를 검색하는 `super()` 함수는 파이썬 3에서 이제 인수 없이 사용할 수 있게 되었다. 다음 예를 보자.

```
class C(A,B):
 def bar(self):
 return super().bar() # 기반 클래스에 있는 bar()를 호출
```

파이썬 2에서는 `super(C, self).bar()`를 사용해야 한다. 여전히 예전 문법을 사용할 수 있지만 상당히 번거롭다.

## 고급 메타클래스

파이썬 2에서는 메타클래스를 정의해서 클래스의 작동 방식을 변경할 수 있다. 메

타클래스 구현과 관련해서 까다로운 측면은 클래스 몸체가 실행된 후에 비로소 메타클래스에서 작업이 수행된다는 점이다. 즉, 인터프리터는 먼저 클래스의 몸체 전체를 실행하고 사전의 내용을 채운다. 채워진 사전은 메타클래스 생성자로 전달된다(클래스 몸체가 실행된 후).

파이썬 3에서는 메타클래스에서 클래스 몸체가 실행되기 전에 추가로 작업을 수행할 수 있다. 메타클래스에 `__prepare__(cls, name, bases, **kwargs)`라는 특수한 클래스 메서드를 정의하면 된다. 이 메서드는 결과로 사전을 반환한다. 이 사전은 클래스 정의 몸체가 실행되는 도중 채워지게 된다. 다음은 기본 과정을 보여주는 예이다.

```
class MyMeta(type):
 @classmethod
 def __prepare__(cls, name, bases, **kwargs):
 print("preparing", name, bases, kwargs)
 return {}
 def __new__(cls, name, bases, classdict):
 print("creating", name, bases, classdict)
 return type.__new__(cls, name, bases, classdict)
```

파이썬 3에서는 메타클래스를 기술할 때 다른 문법을 사용한다. 예를 들어, `MyMeta`라는 클래스를 정의할 때 다음과 같이 한다.

```
class Foo(metaclass=MyMeta):
 print("About to define methods")
 def __init__(self):
 pass
 def bar(self):
 pass
 print("Done defining methods")
```

이 코드를 실행하면 제어 흐름을 보여주는 다음과 같은 출력을 얻는다.

```
preparing Foo () {}
About to define methods
Done defining methods
creating Foo () {'__module__': '__main__',
 'bar': <function bar at 0x3845d0>,
 '__init__': <function __init__ at 0x384588>}
```

메타클래스의 `__prepare__( )` 메서드에서 추가 키워드 인수는 `class`문의 기본 클래스 목록에서 사용한 키워드 인수를 나타낸다. 예를 들어, `class Foo(metaclass=MyMeta, spam=42, blah="Hello")` 문은 키워드 인수인 `spam`과 `blah`를 `MyMeta.__prepare__( )`에 전달한다. 이 규약은 임의의 설정 정보를 메타클래스에 넘기는 데 사용할 수 있다.

메타클래스의 새 메서드인 `__prepare__( )`에서 유용한 작업을 수행하고자 할 때 보통 이 메서드에서 커스터マイ즈된 사전 객체를 반환하면 된다. 예를 들어, 클래스가 정의되는 과정에 특수한 처리를 하고 싶다면 dict에서 상속한 클래스를 정의하고 클래스 사전에 할당되는 값들을 포착하기 위해 `__setitem__( )` 메서드를 재구현하면 된다. 다음은 이 방법을 사용하여 메서드나 클래스 변수가 여러 번 정의된 경우 예러를 보고하도록 메타클래스를 정의하는 예를 보여준다.

```
class MultipleDef(dict):
 def __init__(self):
 self.multiple = set()
 def __setitem__(self, name, value):
 if name in self:
 self.multiple.add(name)
 dict.__setitem__(self, name, value)

class MultiMeta(type):
 @classmethod
 def __prepare__(cls, name, bases, **kwargs):
 return MultipleDef()
 def __new__(cls, name, bases, classdict):
 for name in classdict.multiple:
 print(name, "multiply defined")

 if classdict.multiple:
 raise TypeError("Multiple definitions exist")
 return type.__new__(cls, name, bases, classdict)
```

이 메타클래스는 클래스 정의에 적용하면 메서드가 중복 정의될 경우 예러를 보고한다. 다음 예를 보자.

```
class Foo(metaclass=MultiMeta):
 def __init__(self):
 pass
 def __init__(self, x): # Error. __init__가 여러 번 정의되었다.
 pass
```

## 흔한 위험

파이썬 2에서 3로 옮겨가는 중이라면 파이썬 3가 단순히 새로운 문법과 언어 기능을 제공하는 것에 그치지 않는다는 점을 알아야 한다. 핵심 언어와 라이브러리의 주요 부분들에 때때로 까다로울 정도로 재작업이 이루어졌다. 파이썬 2 프로그래머에게는 버그인 것처럼 보이는 파이썬 3의 측면도 있다. 다른 경우에는 파이썬 2에서는 쉬워 보였던 것이 금지된 것도 있다.

여기에서는 파이썬 2 프로그래머가 파이썬 3로 전환할 때 흔히 겪게 되는 주요 위험 요소들을 개관한다.

## **텍스트 대 바이트들**

파이썬 3는 텍스트 문자열(문자들)과 이진 데이터(바이트들) 사이를 엄격히 구분한다. “hello” 같은 상수는 유니코드로 저장되는 텍스트 문자열이고 b“hello”는 바이트들로 구성되는 문자열을 나타낸다(이 경우 ASCII 문자들을 담는다).

파이썬 3에서는 어떠한 경우에도 str과 bytes 타입을 섞으면 안된다. 예를 들어, 문자열과 바이트들을 연결하려고 하면 TypeError 예외가 발생한다. 이 점은 필요할 때 바이트 문자열이 유니코드로 자동 변환하는 파이썬 2와 다르다.

텍스트 문자열 s를 바이트들로 변환하려면 s.encode(encoding)를 사용해야 한다. 예를 들어, s.encode('utf-8')는 s를 UTF-8로 인코딩된 바이트 문자열로 바꾼다. 바이트 문자열 t를 텍스트로 다시 바꾸려면 t.decode(encoding)를 사용해야 한다. encode()와 decode()를 일종의 문자열과 바이트들 사이에 타입 변환으로 볼 수 있다.

텍스트와 바이트들을 구별하는 것은 결국 좋은 일이다. 파이썬 2에서 문자열 타입들을 섞어 사용할 때 규칙이 애매모호하고 이해하기도 어렵기 때문이다. 하지만, 파이썬 3의 접근법으로 인해 결과적으로 바이트 문자열은 텍스트로서 작동하기에는 제약이 생기게 되었다. 바이트 문자열에도 split()이나 replace() 같은 표준 문자열 메서드가 있긴 하지만 바이트 문자열의 다른 부분들은 파이썬 2에서와 같지 않다. 예를 들어, 바이트 문자열을 출력하고 싶다면 b‘contents’처럼 따옴표 처리를 한 다음 repr()로 출력력을 얻어야 한다. 비슷하게 %나 .format() 같은 문자열 포맷 연산자도 작동하지 않는다. 다음 예를 보자.

```
x = b'Hello World'
print(x) # b'Hello World'를 생성
print(b"You said %s" % x) # TypeError: % 연산자가 지원되지 않는다.
```

바이트들이 텍스트처럼 작동하지 않게 된 것은 시스템 프로그래머에게 잠재적인 어려움을 안겨준다. 유니코드의 공습에도 불구하고 ASCII 같은 바이트 지향 데이터로 작업하고 싶은 경우가 많다. 유니코드를 사용할 때의 부담과 복잡성을 피하기 위해서 bytes 타입을 사용할 수도 있다. 하지만 이렇게 하면 실제로는 바이트 지향 텍스트 처리를 더욱 어렵게 만드는 꼴이 된다. 다음 예를 통해 잠재적인 문제를 살

펴보자.

```
>>> # 문자열(유니코드)을 사용해 응답 메시지를 생성한다.
>>> status = 200
>>> msg = "OK"
>>> proto = "HTTP/1.0"
>>> response = "%s %d %s" % (proto, status, msg)
>>> print(response)
HTTP/1.0 200 OK

>>> # 바이트들(ASCII)을 사용해 응답 메시지를 생성한다.
>>> status = 200
>>> msg = b"OK"
>>> proto = b"HTTP/1.0"
>>> response = b"%s %d %s" % (proto, status, msg)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'

>>> response = proto + b" " + str(status) + b" " + msg
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str

>>> bytes(status)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00....'

>>> bytes(str(status))
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding

>>> bytes(str(status), 'ascii')
b'200'

>>> response = proto + b" " + bytes(str(status), 'ascii') + b" " + msg
>>> print(response)
b'HTTP/1.0 200 OK'

>>> print(response.decode('ascii'))
HTTP/1.0 200 OK
>>>
```

이 예에서 파이썬 3에서 어떻게 텍스트와 바이트들을 엄격하게 구별하고 있는지 볼 수 있다. 정수를 ASCII 문자들로 변환하는 것 같이 간단하게 보이는 연산도 바이트들을 사용할 때는 훨씬 복잡해진다.

텍스트 기반 처리나 포맷 지정 같은 일을 수행하는 경우에는 항상 표준 텍스트 문자열을 사용하는 것이 좋다. 처리를 끝내고 나서 바이트 문자열을 얻어야 한다면 유니코드를 변환하기 위해 `s.encode('latin-1')`을 사용할 수 있다.

텍스트와 바이트들을 구별하는 것은 다양한 라이브러리 모듈을 사용할 때 번거로운 일을 만들 수 있다. 어떤 라이브러리는 텍스트와 바이트들 모두에 잘 작동하지만 바이트들의 사용을 금지하는 라이브러리도 있다. 다른 경우에는 입력이 무엇이냐에 따라 작동 방식이 달라지는 것도 있다. 예를 들어, `os.listdir(dirname)` 함수는 `dirname`이 문자열이면 성공적으로 디코딩할 수 있는 파일 이름들을 유니코드로 반환한다. `dirname`이 바이트 문자열이면 모든 파일 이름을 바이트 문자열로 반환한다.

## 새로운 I/O 시스템

파이썬 3는 완전히 새로운 I/O 시스템을 구현하였다. 자세한 내용은 19장 ‘운영체제 서비스’의 `io` 모듈 관련 절에서 설명하였다. 새로운 I/O 시스템은 문자열과 관련해 존재하는 텍스트와 이진 데이터 사이에 엄격한 구별도 반영한다.

텍스트에 대해 I/O를 수행하는 경우 파이썬 3에서는 파일을 텍스트 모드로 열고, 기본 인코딩(보통 UTF-8)이 아닌 다른 인코딩을 사용하고자 할 때는 옵션으로 인코딩을 입력해야 한다. 이진 데이터에 대해 I/O를 수행하는 것이라면 반드시 파일을 이진 모드에서 열고 바이트 문자열을 사용해야 한다. 흔히 발생하기 쉬운 에러는 잘못된 모드로 연 파일이나 I/O 스트림에 출력을 전달하는 것이다. 다음 예를 보자.

```
>>> f = open("foo.txt", "wb")
>>> f.write("Hello World\n")
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "/tmp/lib/python3.0/io.py", line 1035, in write
 raise TypeError("can't write str to binary stream")
TypeError: can't write str to binary stream
>>>
```

소켓, 파일, 기타 I/O 채널은 항상 이진 모드로 작동한다고 가정된다. 특별히 네트워크 코드와 관련해서 발생할 수 있는 잠재적인 문제로 많은 네트워크 프로토콜이 텍스트 기반 요청/응답 처리가 필요하다는 점 때문에 발생하는 문제가 있다 (HTTP, SMTP, FTP 등). 소켓이 이진 모드로 작동하기 때문에 이렇게 이진 I/O와 텍스트 처리가 섞이면 앞에서 살펴본 텍스트와 바이트들을 섞을 때 발생할 수 있는 문제들을 겪을 수 있으므로 조심해야 한다.

## print( )와 exec( ) 함수

파이썬에서 print와 exec문이 이제 함수이다. 다음 예는 print( ) 함수의 사용법을 파이썬 2의 것과 비교한다.

```
print(x,y,z) # print x, y, z와 같다.
print(x,y,z,end=' ') # print x, y, z,와 같다.
print(a,file=f) # print >>f, a와 같다.
```

print( )가 함수이기 때문에 원할 경우 정의를 변경할 수 있다.

exec( )도 이제 함수이다. 그렇지만 파이썬 3에서 작동 방식은 파이썬 2에서와는 상당히 다르다. 다음 코드를 보자.

```
def foo():
 exec("a = 42")
 print(a)
```

파이썬 2에서는 foo( )를 호출하면 숫자 '42'가 출력된다. 파이썬 3에서는 변수 a가 정의되지 않았다는 NameError가 발생한다. 무슨 일이 발생했나 하면, exec( )가 함수이기 때문에 globals( )나 locals( ) 함수가 반환하는 사전을 사용했다. locals( )가 반환하는 사전은 실제로는 지역 변수들의 복사본이다. exec( ) 함수에서 수행한 할당은 지역 변수 자체를 수정하는 것이 아니라 단순히 이 복사본을 수정한다. 다음은 이 문제를 피해가는 방법을 보여준다.

```
def foo():
 _locals = locals()
 exec('a = 42',globals(),_locals)
 a = _locals['a'] # 값을 설정한 변수를 가져온다.
 print(a)
```

이제 파이썬 2에서 exec( ), eval( ), execfile( ) 등으로 가능했던 어느 정도의 마법 같은 일이 파이썬 3에도 그대로 지원될 것이라고 기대하면 안 된다. 사실 execfile( )은 아예 없어졌다(exec( )에 열린 파일 같은 객체를 넘겨서 비슷한 기능을 흉내 낼 수는 있다).

## 반복자와 뷰의 사용

파이썬 3에서는 2에서보다 반복자와 생성기를 훨씬 많이 사용한다. 이전에 리스트를 반환했던 zip( ), map( ), range( ) 같은 내장 함수들은 이제 반복자를 반환한다. 결과로부터 리스트를 만들려면 list( ) 함수를 사용하도록 한다.

파이썬 3에서는 사전에서 키와 값을 추출할 때 약간 다른 접근법을 취한다. 파이

씬 2에서 키, 값, 키/값 쌍 등을 얻기 위해 `d.keys()`, `d.values()`, `d.items()` 같은 메서드를 사용했다. 파이썬 3에서는 이 메서드들이 뷰(view)라고 불리는 객체를 반환한다. 다음 예를 보자.

```
>>> s = { 'GOOG': 490.10, 'AAPL': 123.45, 'IBM': 91.10 }
>>> k = s.keys()
>>> k
<dict_keys object at 0x33d950>
>>> v = s.values()
>>> v
<dict_values object at 0x33d960>
>>>
```

이 객체들은 반복을 지원하므로 내용을 보는 데 for 루프를 사용할 수 있다. 다음 예를 보자.

```
>>> for x in k:
... print(x)
...
GOOG
AAPL
IBM
>>>
```

뷰 객체는 자신을 생성한 사전에 항상 묶여 있다. 이 때문에 내부 사전이 변하면 뷰가 생성하는 항목도 변한다. 다음 예를 보자.

```
>>> s['ACME'] = 5612.25
>>> for x in k:
... print(x)
...
GOOG
AAPL
IBM
ACME
>>>
```

사전 키나 값으로 리스트를 생성해야 한다면 간단히 `list()` 함수를 사용하도록 한다. 예를 들어, `list(s.keys())`.

## 정수와 정수 나누기

파이썬 3에서는 더 이상 32비트 정수에 대한 `int` 타입과 긴 정수에 대한 별개의 `long` 타입이 없다. `int` 타입이 이제 임의의 정밀도를 가진 정수를 표현한다(그 내부는 사용자에게 공개되지 않는다).

더불어 정수 나누기는 이제 항상 부동 소수점 결과를 생성한다. 예를 들어,  $3/5$ 은

0이 아니라 0.6이다. 심지어 결과가 정수일지라도 항상 실수로 변환된다. 8/2은 4가 아니라 4.0이다.

## 비교

파이썬 3에서는 값은 훨씬 엄격하게 비교한다. 파이썬 2에서는 말이 안 될지라도 어떤 객체든 비교할 수 있었다. 다음 예를 보자.

```
>>> 3 < "Hello"
True
>>>
```

파이썬 3에서는 TypeError가 발생한다. 다음 예를 보자.

```
>>> 3 < "Hello"
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
>>>
```

작은 변화이지만, 이제 파이썬 3에서는 더 조심스럽게 데이터가 적절한 타입으로 되어 있도록 해야 한다. 예를 들어, 리스트의 sort( ) 메서드를 사용할 때는 반드시 리스트에 들어 있는 모든 항목이 ‘<’ 연산자로 비교할 수 있어야 한다. 아니면 에러가 발생한다. 파이썬 2에서는 보통 의미 없는 결과를 내면서 해당 연산이 조용히 수행된다.

## 반복자와 생성기

파이썬 3에서는 반복자 프로토콜에 약간 변화가 있다. \_\_iter\_\_( )를 호출하고 반복을 수행하기 위해 호출하는 next( ) 메서드의 이름이 \_\_next\_\_( )로 변경되었다. 반복 가능한 객체에 대해 직접 반복을 수행하는 코드를 작성했거나 나름대로의 반복자 객체를 정의한 경우를 제외하고는 대부분 이 변화에 영향을 받지 않을 것이다. 그렇지 않은 경우라면 클래스에서 next( ) 메서드 이름을 바꾸는 것을 잊지 않도록 한다. 이식 가능하도록 반복자의 next( )나 \_\_next\_\_( ) 메서드를 적절히 호출해주는 내장 next( ) 함수를 사용하는 것이 좋다.

## 파일 이름, 인수와 환경 변수

파이썬 3에서는 파일 이름, sys.argv에 있는 명령줄 인수, 그리고 os.environ에 있는 환경 변수가 지역 설정에 따라 유니코드로 취급될 수도, 아닐 수도 있다. 문제가 될 수 있는 유일한 부분은 모든 운영체제에서 유니코드를 사용하지 않을 수도 있다는 점이다. 예를 들어, 많은 시스템에서는 기술적으로 사실상 파일 이름, 명령줄 인수, 환경 변수를 유효한 유니코드 인코딩에 대응하지 않는 무가공 바이트들의 순서열로 지정할 수 있다. 실전에서 이런 경우는 드물지만, 그래도 파이썬을 시스템 관리 작업과 관련된 곳에 사용하려는 프로그래머에게는 걱정거리가 될 수 있다. 이미 언급했듯이 파일 이름과 디렉터리 이름을 바이트 문자열로 지정하면 많은 문제를 해결할 수 있다. 예를 들어, os.listdir(b'foo').

## 라이브러리 재구성

파이썬 3에서는 표준 라이브러리의 몇몇 부분을 재구성하고 이름을 바꾸었다. 가장 눈에 띄는 변화는 네트워킹과 인터넷 데이터 형식 관련 모듈이다. 또한 다양한 레거시 모듈이 라이브러리에서 제거되었다(gopherlib, rfc822 등).

이제 모듈 이름에 소문자를 쓰는 것이 표준 관례가 되었다. ConfigParser, Queue, SocketServer 같은 모듈은 configparser, queue, socketserver로 이름이 바뀌었다. 여러분의 코드에서도 비슷한 관례를 따르도록 하라.

별개의 모듈로 있던 코드가 패키지를 사용해 재구성되었다. 예를 들면, http 패키지는 HTTP 서버를 작성하는 데 필요한 모듈을 담고, html 패키지는 HTML을 파싱하는 데 필요한 모듈을 담으며, xmlrpc 패키지는 XML-RPC를 위한 모듈을 담고 있다.

사용이 권장되지 않는 모듈과 관련해서 이 책에서는 파이썬 2.6과 3.0에서 현재 사용 중인 모듈만 설명하기 위해 주의를 기울였다. 기존 파이썬 2 코드를 보고 있는데 이 책에 나와 있지 않은 모듈을 사용하고 있다면 더 최신의 것이 나와서 현재 사용이 권장되지 않는 모듈일 가능성이 크다. 한 예로 파이썬 3에는 파이썬 2에서 하위 프로세스를 실행하는 데 사용하던 popen2 모듈이 없다. 대신 subprocess 모듈을 사용해야 한다.

## 절대적인 임포트

라이브러리 재구성과 관련해서 패키지의 하위 모듈에 나타나는 모든 import문은 절대적인 이름을 사용하도록 바뀌었다. 여기에 관해서는 8장에서 자세히 다루었다. 다음과 같이 패키지가 구성되어 있다고 하자.

```
foo/
 __init__.py
 spam.py
 bar.py
```

파일 spam.py에서 import bar문을 사용하면 bar.py가 같은 디렉터리에 있더라도 ImportError 예외가 발생한다. 해당 하위 모듈을 로드하려면 spam.py에서 import foo.bar문을 사용하거나 from . import bar처럼 패키지에 상대적인 임포트를 사용해야 한다.

이 부분은 항상 sys.path에 있는 다른 디렉터리를 검사하기 전에 현재 디렉터리를 검사하는 파이썬 2의 import 작동 방식과 다르다.

## 코드 이주와 2to3

코드를 파이썬 2에서 3으로 변환하는 일은 어렵다. 임의의 파이썬 2 프로그램을 파이썬 3에서 실행할 수 있게 하는 마법 같은 임포트, 플래그, 환경 변수나 도구 같은 것은 없다. 그렇지만 코드를 이주하는 데 취할 수 있는 구체적인 단계들은 나열할 수 있다. 지금부터 이에 관해 알아보도록 한다.

### 파이썬 2.6으로 코드 포팅

코드를 파이썬 3으로 포팅하고자 한다면 먼저 파이썬 2.6으로 포팅할 것을 권한다. 파이썬 2.6은 파이썬 2.5와 하위 호환성이 있으면서도 파이썬 3에 있는 새로운 기능의 일부를 지원한다. 그런 기능의 예로 고급 문자열 포맷 지정, 새로운 예외 문법, 바이트 상수, I/O 라이브러리, 추상 기반 클래스 등이 있다. 따라서 파이썬 2 프로그램에서는 완전한 이주 준비가 되어 있지 않더라도 지금 바로 파이썬 3에 있는 유용한 기능을 사용하기 시작할 수 있다.

파이썬 2.6으로 먼저 포팅해야 하는 다른 이유로는 파이썬 2.6에는 -3 명령줄 옵션으로 실행하면 사용이 권장되지 않는 기능이 있는 경우 경고 메시지를 출력하는 기능이 있다는 점을 들 수 있다. 다음 예를 보자.

```

bash-3.2$ python -3
Python 2.6 (trunk:66714:66715M, Oct 1 2008, 18:36:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5370)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = {}
>>> a.has_key('foo')
__main__:1: DeprecationWarning: dict.has_key() not supported in 3.x;
use the in operator
False
>>

```

이 경고 메시지를 참고해서 파이썬 3로 포팅하기 전에 프로그램이 파이썬 2.6에서 경고 없이 잘 작동하게 만들도록 한다.

## **테스트 커버리지 제공**

파이썬에는 doctest와 unittest 같은 유용한 테스트 모듈이 있다. 응용 프로그램을 파이썬 3로 포팅하려고 시도하기 전에 충분한 테스트 도달 범위를 달성하도록 하는 것이 좋다. 여러분의 프로그램에 아직까지도 테스트를 작성하지 않았다면 바로 지금이 시작하기 좋은 때이다. 파이썬 2.6에서 실행했을 때 테스트가 가능한 한 많은 부분을 커버하고 모든 테스트가 경고 메시지 없이 통과하도록 만들도록 한다.

## **2 to 3 도구 사용**

파이썬 3에는 파이썬 2.6 코드를 파이썬 3으로 이주하는 작업을 돋는 2 to 3라는 도구가 있다. 이 도구는 보통 파이썬 소스 배포본에서 Tools/scripts 디렉터리에서 찾을 수 있고 대부분의 시스템에서 python3.0 프로그램이 설치된 곳과 같은 디렉터리에 설치된다. 이 도구는 보통 유닉스나 윈도 셸에서 실행하는 명령줄 도구이다.

예로서 사용이 권장되지 않는 몇몇 기능을 담은 다음 프로그램을 보자.

```

example.py
import ConfigParser

for i in xrange(10):
 print i, 2*i

def spam(d):
 if not d.has_key("spam"):
 d["spam"] = load_spam()
 return d["spam"]

```

이 프로그램에 대해 2 to 3를 실행하기 위해 “2 to 3 example.py”를 실행한다. 다음을 보자.

```
% 2to3 example.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- example.py (original)
+++ example.py (refactored)
@@ -1,10 +1,10 @@
example.py
-import ConfigParser
+import configparser

-for i in xrange(10):
- print i, 2*i
+for i in range(10):
+ print(i, 2*i)

def spam(d):
- if not d.has_key("spam"):
+ if "spam" not in d:
 d["spam"] = load_spam()
 return d["spam"]
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

2 to 3는 문제가 있을 것 같은 부분이나 수정이 필요하다고 생각되는 부분을 찾아서 출력한다. 이런 부분들은 문맥 비교(context-diff) 형태로 출력된다. 한 파일에 2 to 3를 사용해도 되지만 디렉터리 이름을 주면 해당 디렉터리 안에 있는 모든 파일을 재귀적으로 찾아서 모든 것에 대한 보고서를 출력한다.

기본으로 2 to 3는 스캔한 소스 코드를 직접 수정하지 않는다. 단순히 수정이 필요한 부분을 출력하기만 한다. 2 to 3는 불완전한 정보만 가지고 있다는 문제를 안고 있다. 앞의 예에 나온 spam() 함수를 보자. 이 함수는 d.has\_key() 메서드를 호출한다. 사전의 경우 in 연산자를 쓰는 것이 권장되어 has\_key()가 제거되었다. 2 to 3는 이 변화를 보고하지만 추가 정보 없이는 spam()에서 실제로 사전으로 작업하는지를 알 수 없다. d가 우연히 has\_key() 메서드를 가지게 된 다른 객체(아마 데이터베이스)일 수 있으며 그런 경우 in 연산자가 제대로 작동하지 않을 수 있다. 2 to 3에서 발생할 수 있는 다른 문제로 바이트 문자열과 유니코드와 관련한 문제가 있다. 파일 2에서 바이트 문자열을 자동으로 유니코드로 변환한다는 점 때문에 이 둘을 섞어 쓰는 코드를 흔히 볼 수 있다. 불행히도 2 to 3는 이 문제를 모두 해결하지 못한다. 단위 테스트 도달 범위가 넓어야 하는 이유이기도 하다. 물론 응용

에 따라 다를 수 있다.

2 to 3에는 선택된 비호환성을 고치게 하는 옵션이 있다. 먼저, 2 to 3 -l을 입력하면 고침 목록이 출력된다. 다음을 보자.

```
% 2to3 -l
Available transformations for the -f/--fix option:
apply
basestring
buffer
callable
...
...
xrange
xreadlines
zip
```

이 목록에 있는 이름으로 “2 to 3 -f fixname filename”을 입력하면 어떤 변화를 가하는지를 볼 수 있다. 여러 가지를 고치고 싶다면 각각을 별개의 -f 옵션으로 지정하면 된다. 소스 파일을 실제로 고치고 싶을 때는 2 to 3 -f fixname -w filename처럼 -w 옵션을 사용하면 된다. 다음 예를 보자.

```
% 2to3 -f xrange -w example.py
--- example.py (original)
+++ example.py (refactored)
@@ -1,7 +1,7 @@
 # example.py
 import ConfigParser

-for i in xrange(10):
+for i in range(10):
 print i, 2*i

def spam(d):
RefactoringTool: Files that were modified:
RefactoringTool: example.py
```

이렇게 한 다음에 example.py를 보면 xrange( )가 range( )로 수정된 것 말고는 다른 변화는 없는 것을 확인할 수 있다. 원래 example.py 파일은 example.py.bak에 백업된다.

-f 옵션과 반대되는 것은 -x 옵션이다. 2 to 3 -x fixname filename을 입력하면 -x 옵션으로 지정한 것을 제외한 모든 고침 목록이 적용된다.

2 to 3로 모든 것을 고친 다음 파일을 덮어쓰는 것도 가능하지만 실전에서는 그렇게 하지 않는 것이 좋다. 코드 변환은 완벽히 과학적인 것이 아니며 2 to 3는 항상 올바른 일만 수행하는 것도 아니다. 운에 맡기고 모든 것이 마법처럼 작동하기를 기대하기보다는 계획에 따라 단계적으로 코드 이주를 수행하는 것이 훨씬 낫다.

2 to 3는 추가로 몇 가지 유용한 옵션을 제공한다. -v 옵션은 상세 모드를 활성화하여 디버깅에 도움이 될 만한 추가 정보를 출력한다. -p 옵션은 여러분이 코드에서 이미 print문을 함수로 사용하고 있기 때문에 변환하지 않아야 한다는 것을 2 to 3에 알린다(from `__future__ import print_statement`문으로 활성화한다).

## 효과적 포팅 전략

다음은 파이썬 2 코드를 파이썬 3로 포팅하는 효과적인 전략을 설명한다. 다시 한번 말하지만 모든 것을 한 번에 하기보다는 조직적이고 단계적으로 코드를 이주하는 것이 더 좋다.

1. 코드에 대한 단위 테스트 뮤음을 작성하고 파이썬 2에서 모든 테스트가 성공하게 하라.
2. 코드와 테스트 뮤음을 파이썬 2.6으로 포팅하고 모든 테스트가 통과하는 것을 확인하라.
3. 파이썬 2.6에 -3 옵션을 켠다. 모든 경고 메시지를 처리하고 프로그램이 경고 메시지 없이 실행되고 모든 테스트를 통과하게 만들어라. 여기까지 제대로 했으면 여러분의 코드가 파이썬 2.5나 이전 버전에서도 여전히 잘 돌아갈 확률이 높다. 이제 여러분은 프로그램에 쌓인 깔끔하지 못한 부분들을 정리하였다.
4. 코드의 백업본을 만든다(더 설명할 필요가 없을 것이다).
5. 단위 테스트 뮤음을 파이썬 3로 포팅하고 테스팅 환경 자체가 제대로 작동하는지를 확실히 한다. 개별 단위 테스트 자체는 실패할 수 있다(아직 아무 코드도 포팅하지 않았기 때문이다). 하지만 적절히 작성된 테스트 뮤음이라면 테스트 소프트웨어 자체는 제대로 돌아가서 테스트 실패를 적절히 처리할 수 있어야 한다.
6. 2 to 3를 사용해 프로그램 자체를 파이썬 3로 포팅한다. 결과 코드에 단위 테스트 뮤음을 실행하고 발생하는 모든 문제를 해결한다. 이렇게 하는 데 몇 가지 전략을 쓸 수 있다. 운이 좋다고 느낀다면 2 to 3에 모든 것을 고치라고 하고 어떻게 되는지 본다. 조심스러운 편이라면 일단 2 to 3가 확실한 것만 고치게 하고(`print`, `except`문, `xrange()`, 라이브러리 모듈 이름 등) 그런 다음 남아 있는 문제를 하나씩 처리해 나갈 수 있다.

이 과정의 끝에 도달하면 여러분의 코드는 모든 단위 테스트를 통과하고 이전과 같은 방식으로 작동할 것이다.

이론적으로는 파이썬 2.6에서 돌아가면서도 사용자 개입 없이 자동으로 파이썬 3로 변환될 수 있는 코드를 작성하는 것이 가능하다. 그렇지만 이렇게 하려면 최신 파이썬 코딩 규약을 주의 깊게 따를 필요가 있다. 적어도 파이썬 2.6에서 경고가 발생하지 않게 해야 한다. 자동 변환 과정에서 2 to 3를 특정한 방식으로 사용해야 한다면(고침 목록 중에서 특정 몇 개만 실행한다든지) 사용자가 직접 2 to 3를 실행하게 하지 말고 셸 스크립트를 작성해서 자동으로 필요한 작업을 수행하도록 하는 것이 좋을 것이다.

## 파이썬 2와 3 동시 지원

파이썬 3 이주와 관련해서 마지막 질문은 과연 파이썬 2와 3에서 수정하지 않고 모두 작동하는 단일 코드 기반을 구축할 수 있느냐는 것이다. 어떤 경우에는 가능하기도 하지만 그렇게 작성한 코드는 엉망이 될 수 있다. 예를 들어, print문은 사용하지 말아야 하고 except절에는 예외 값을 포함시키면 안 된다(대신 sys.exc\_info()에서 얻어야 한다). 다른 가능성은 아예 작동하지 않을 수도 있다. 예를 들어, 문법이 다르기 때문에 파이썬 2와 3에서 포팅 가능한 형태로 메타클래스를 사용할 수 있는 방법은 없다. 따라서 여러분이 파이썬 2와 3에서 모두 작동해야 하는 코드를 관리할 경우 가장 좋은 것은 먼저 코드를 가능한 한 깔끔하게 정리하고 파이썬 2.6에서 돌아가게 하고 단위 테스트 뮤음을 확보하며 자동 변환이 가능하도록 2 to 3 고침 목록을 찾는 것이다.

단일 코드 기반을 확보하는 것이 좋은 경우는 단위 테스트를 작성할 때이다. 파이썬 2.6과 3에서 수정 없이 작동하는 테스트 뮤음이 있다면 2 to 3로 변환을 수행한 후에 응용 프로그램이 제대로 작동하는지를 검사하는 데 유용하게 쓸 수 있다.

## 참여하라

오픈 소스 프로젝트로서 파이썬은 사용자들의 공헌에 힘입어 계속 개발되어 왔다. 특히 파이썬 3에 대해서는 버그, 성능 문제나 기타 문제를 신고하는 것이 매우 중요하다. 버그를 신고하려면 <http://bugs.python.org>를 방문하면 된다. 망설이지 말기 바란다. 여러분의 피드백은 모든 이들을 위해 파이썬을 더 좋게 만드는 데 도움이 될 것이다.

# 찾아보기

## 기호

- 단항 마이너스 연산자 77  
 - 빼기 연산자 77  
 - 차집합 연산자 16, 90  
 - 하이픈 문자, 파일 이름으로 사용 214  
 ! 디버거 명령, pdb 모듈 230  
 != ~와 다른 연산자 79  
 lr 지정자 in 문자열 포맷 89  
 ls 지정자 in 문자열 포맷 89  
 # 주석 5, 30  
 #! 유닉스 셸 스크립트 5, 217  
 + 단항 플러스 연산자 77  
 + 더하기 연산자 77  
 + 리스트 연결 연산자 13  
 + 문자열 연결 연산자 11  
 + 순서열 연결 연산자 80  
 == ~와 같은 연산자 79, 94  
 설치 때 패키지 재작성 188  
 \$ 변수 문자열 86  
 % 나머지 연산자 77  
 % 문자열 포맷 지정 연산자 7, 84, 198  
 %= 연산자 90  
 & 교집합 연산자 16, 90  
 & 비트 연산자 78  
 &= 연산자 90  
 () 튜플 14, 35  
 () 함수 호출 연산자 91

\* 꼽하기 연산자 77  
 \* 순서열 복사 연산자 80  
 \* 순서열을 함수 인수로 전달 112  
 \* 와일드카드  
 모듈 임포트 27  
 파이썬 3에서 반복 가능한 객체 풀어헤치기 771  
 from 모듈 import 178  
 -\* 코드: 주석, 소스 코드 37  
 \* 키워드 전용 인수, 파이썬 3 773  
 \* 함수 정의에서 가변 길이 인수 112  
 \*\* 사전을 키워드 인수로 전달 113~114  
 \*\* 제곱 연산자 77  
 \*\* 함수 정의에서 가변 길이 키워드 인수 113~114  
 \*\*= 연산자 90  
 \*= 연산자 90  
 . 상대 import문에서 디렉터리 참조 185  
 . 속성 바인딩 연산자 40, 58, 91, 143  
 모듈 61  
 특수 메서드 69  
 ... Ellipsis 36, 65, 72  
 ... 인터프리터 프롬프트 216  
 / 나누기 연산자 77  
 // 끝수를 버리는 나누기 연산자 77  
 //= 연산자 90  
 /= 연산자 90  
 : 문자열 포맷 지정자에서 콜론 87  
 ; 세미콜론 6, 30  
 @ 장식자 36, 121  
 [:] 확장 분할 연산자 47~48, 80~81  
 [:] 분할 연산자 47~48, 80~81  
 [] 리스트 12, 35  
 [] 색인 연산자 47~48, 80  
 매핑 54  
 순서열 80  
 특수 메서드 70  
 ^ 대칭 차집합 연산자 16, 90  
 ^ 문자열 포맷 지정자에서 가운데 정렬 87  
 ^ 비트 xor 연산자 78  
 ^= 연산자 90  
 \_ 변수, 대화식 모드 4, 217  
 {} 사전 16, 35  
 {} 집합 상수, 파이썬 3 769  
 {} 포맷 문자열에서 자리 표시자 86  
 | 비트 or 연산자 78  
 | 합집합 연산자 16, 90  
 |= 연산자 90  
 ~ 비트 negation 연산자 78  
 ~ 파일 이름에서 사용자 홈 디렉터리 확장 488  
 ' 작은따옴표 10, 32  
 " 심중따옴표 10, 32  
 " 큰따옴표 10, 32  
 """ 심중따옴표 10, 32  
 \ 문자열 탈출 코드 32

- \ 줄 이음 문자 8, 29, 35  
 \N 탈출 코드, 문자열 33  
 \U 탈출 코드, 문자열 33  
 \u 탈출 코드, 문자열 33  
 \x 탈출 코드, 문자열 33  
 += 연산자 90  
 < ~보다 작은 연산자 79  
 < 문자열 포맷 지정자에서 왼쪽 정렬 87  
 << 왼쪽 이동 연산자 78  
 <<= 연산자 90  
 <= ~보다 작거나 같은 연산자 79  
 -= 연산자 90  
 > ~보다 큰 연산자 79  
 > 문자열 포맷 지정자에서 오른쪽 정렬 87  
 >= ~보다 크거나 같은 연산자 79  
 >> print 파일 전환 변경자 9, 199  
 >> 오른쪽 이동 연산자 78  
 >>= 연산자 90  
 >>> 인터프리터 프롬프트 3, 216  
 0b 이진 정수 상수 31  
 0o 8진수 정수 상수 31  
 0x 16진수 정수 상수 31  
 16진수  
     정수 상수 31  
     정수로부터 문자열 생성 93  
 1개 요소 튜플 14  
 1급 객체 44  
     사용 44  
 2to3 도구 785~788  
     한계 787  
 2의 보수 정수 78  
 -3 명령줄 옵션 213, 789  
 8진수 정수 상수 31
- 
- ㄱ
- 가변 개수 인수 함수 정의 112  
 가변 개수 키워드 인수 함수 정의  
     113~114  
 개인 메서드, 상속 156  
 개인 속성 155  
 이름 변형 155
- 프로퍼티 156  
 개인 클래스 멤버 31  
 객체 57, 24  
     1급 지위 44  
     계층 구조 169  
     널 객체 정의 455  
     반복 지원 98  
     복사 방법 43  
     비교 40  
     비교 메서드 68  
     속성 40  
     순서 정하기 요구사항 69  
     신원 40  
     약한 참조 296  
     영속화 209  
     이름 41  
     인스턴스 39  
     인터프리터에서 공유 42  
     정의 39  
     조사 273  
     참조 객체 목록 얻기 272  
     참조 횟수 세기 41  
     컨테이너 또는 컬렉션 39  
     큐로 프로세스 사이 전달 514  
     크기 얻기 237, 289  
     크기 얻기 289  
     클래스 39  
     타입 40  
     파이썬 3에서 비교 783  
     파이프로 프로세스 사이 전달 518  
     표현 160  
     dir()로 객체 검사 76  
     marshal로 직렬화 278  
     multiprocessing 모듈에서 대리자 530  
     pickle로 직렬화 279  
     객체 검사, dir() 76  
     객체 계층 구조 169  
     객체 공유 42  
     객체 동등 비교 40  
     객체 신원 40  
     객체 조사 273  
     객체 타입 40
- 객체 타입 비교 41  
 객체의 느슨한 결합 149  
 객체의 신원 비교 40  
 검색, 시작과 끝 색인과 문자열 50  
 검색표, 사전 17  
 게으른 평가 119  
 결과 기억 298  
 경고  
     무시 294  
     예외로 변경 295  
     예외와 차이점 267  
 경쟁 조건 239, 508  
 계산된 속성과 프로퍼티 151  
 계층적 로깅, logging 모듈 442  
 계층적 잠금 546  
 고급 문자열 포맷 지정 7, 51, 86  
 곱하기 연산자 \* 77  
 공유 라이브러리  
     확장 모듈 181  
     ctypes과 함께 로드 755  
 공유 메모리  
     리스트 전달 예 526~527  
     multiprocessing 모듈 525  
 공유 배열, multiprocessing 모듈 525  
 과학 표기, 부동 소수점 31  
 관계 데이터베이스, 파이썬에서 접근 365  
 관계 연산자 9, 68  
 관찰자 패턴 158, 296  
 팔호 생략, 튜플 14  
 교집합 연산자 &, 집합 16  
 교집합 연산자 &, 집합 16  
 구분자 35  
 구조체, 튜플 14  
 구형 클래스 170  
 국제화 문자  
     문자열 비교 84  
     소스 코드 37  
     균일 접근 원칙 152  
     그린 스크립트 551  
     그린릿 551  
     기반 클래스 23  
     기본 유니코드 에러 처리 방식 203

기본 유니코드 인코딩 203, 218

기본 인수 20, 111~112

값 할당 111~112

변경 가능 객체 112

기술자 70, 154

메타클래스 172

기호와 숫자

긴 문장을 여러 줄로 나누기 8

긴 정수 정수 46

정수에서 자동으로 변경 31

31

긴 정수 끝 L 31

깊은 복사 43

꼬리 재귀 최적화, 없음 135

끝 콤마

튜플 14

print문 9, 198

끝수를 버리는 나누기 연산자 // 77

끝수를 버리는, 정수 나누기 74

↳

나누기 연산자 / 77

나누기 연산자, 파이썬 2와 파이썬 3 75

나누기, 정수 버리기 74~75, 77

나머지 연산자 % 77

날짜 파싱 423, 501

날짜와 시간 조작 413

날짜와 시간을 표현 413

내부 타이머 492

내장 예외 26, 104

내장 타입 45

내장 함수, 파이썬 2에서 파이썬 3 함수 사용 268

내장 함수와 타입 247

널 값 46

널 객체 455

네임스페이스

클래스 142

함수 지역 변수 115

import문 27, 175

네트워크 프로그래밍

비동기 575

소개 553

유니코드 인코딩 556

이벤트 주도 프로그래밍 560

풀링 성능 577

호스트 이름 얻기 584

네트워크 프로그래밍 모듈, 파이썬 3 제조

직 611

네트워크 프로그래밍에서 인코딩 문제

556

□

다중 상속 146~147

다중 스레드 담 509

다중 코어, 프로그램 실행 509

다중화, I/O 565

다차원 리스트 13

다형성 148

단언 -O 옵션으로 벗김 109, 181

단위 테스트

예 227

파이썬 3 이주 789

unittest 모듈 226

단일 정밀도 부동 소수점 46

단축 평가, 불리언 표현식 94

단항 마이너스 연산자 - 77

단항 플러스 연산자 + 77

달러 변수 치환 199

대리자 75

대리자 쌍 34, 50

대리자, 속성 바인딩 메서드 161

대소문자 구별, 식별자 30~31

대소문자 변환, 문자열 50~51

대칭 차집합 연산자 ^ 90

대칭 차집합 연산자 ^, 집합 16

대화식 모드 5, 216

결과 출력 67, 217

빈 줄 30

대화식 모드에서 마지막 연산 결과 4, 217

대화식 인터프리터 종료 5

대화식 터미널 214

더하기 연산자 + 77

데드락, 잠금과 원인 546

데몬 스레드 538

데몬 프로세스 510

데이터 구조

리스트와 튜플 15

사전 16

데이터 캡슐화 155

데이터 흐름 처리, 코루틴 130~130

데이터그램 579

데이터베이스

영속 사전 210

CGI 스크립트 665

데이터베이스 API 365

데이터베이스 결과, 사전으로 변환 372

데이터베이스 인터페이스 스레드 372

365

데이터베이스, DBM 스타일 382

도움 얻기, help() 함수 27

동기화

병행 프로그램 508

생성기의 close() 메서드 124~125

생성기의 throw() 메서드 126

동기화 기본 기능

multiprocessing 모듈 526

threading 모듈 540

동등 비교를 위한 최소 요구사항 69

동적 바인딩, 객체 속성 148

동적 유효 범위, 없음 117

들여쓰기 6, 29

같은 줄에 여러 문장 30

문서화 문자열 36

선호 스타일 7

줄 이음 문자 \ 8

탭 30

디렉터리

복사 391

비교 387

셀 파일드카드로 파일 읽기 390

임시 398

재귀 순회 480

접근을 위해 시스템 호출 475

파일 이름 검사 489  
 디렉터리 복사 391  
 디렉터리 제거 392  
 디렉터리 트리 재귀 순회 480  
 디버깅  
     디버거 설정 234  
     메모리 누수 검사 272  
     명령줄에서 전체 프로그램 234  
     잡히지 않은 예외 229  
     중단점 설정 231  
     직접 중단점 설정 230  
     함수 실행 229  
     CGI 스크립트 665  
     logging 모듈 사용 436  
     pdb 모듈 229  
     파옴표, 스타일 차이 32

## ■

락  
     예외 26  
     적절한 관리 546  
     컨텍스트 관리자 107  
     래퍼 함수 114  
     예 121  
     클로저 120  
     확장 모듈 732~733  
     래퍼, 속성 바인딩 메서드 161  
     레거시 코드, exec문 138  
     로그 파일, 실시간 감시 예 21  
     로케일 설정, 문자열 비교 84  
     루비, 객체 시스템 다른점 151  
     루트 로거, logging 모듈 437  
     루프 18, 98  
         루프 카운터 유지 99  
         중간에 빠져나오기 100  
         while문 6  
     리눅스 407  
     리눅스 링크 수준 패킷 프로토콜 주소 형식 578, 581  
     리눅스, epoll 인터페이스 566  
     리스트 12, 48

검색 48  
 다중 프로세스 공유 528  
 동등 94  
 뒤집기 49  
 리스트 내포 130  
 무작위 섞기 313  
 배열 객체와 비교 321  
 분할 12  
 분할 삭제 82~83  
 분할 재할당 12  
 분할 할당 82~83  
 비교 83  
 빈 13  
 색인 연산자 12  
 순서열 47  
 얇은 복사본 생성 48  
 연결 12  
 정렬 49  
 정렬된 순서로 유지 322  
 중첩 13  
 추가 12, 48  
 튜플과 비교 15  
 항목 삭제 82~83  
 항목 세기 47  
 항목 제거 48  
 항목 추가 12, 48, 82~83  
 항목 할당 12, 82~83  
 deque와 비교 240, 323  
 insert()의 비효율성 240  
 리스트 내포 14  
     리스트 내포에서 튜플 생성 131  
     생성기 표현식과 차이점 132  
     선언형 프로그래밍 133  
     유효 범위 반복 변수 131  
     일반적인 문법 130  
     조건 표현식 96  
     awk 명령어와 유사점 134  
     SQL 질의문과 유사점 134  
 리스트 뒤집기 49  
 리스트 일부 재할당 12  
 리틀 엔디안 포맷 204  
 리틀 엔디안, 포장과 풀어헤치기 359

## □

마이크로스레딩 551  
 매핑 53  
     키 색인 연산자 54  
     특수 메서드 70  
     항목 삭제 54  
 메르세네 트위스터 312  
 메모리 관리 156  
     누수 검사 272  
     쓰레기 수집 42, 271  
     인스턴스 생성 157  
     참조 횟수 세기 157  
 메모리 매핑 파일 IPC 455, 507~508  
 메모리 사용  
     객체 크기 얻기 289  
     배열 객체 321  
     측정 237  
     튜플과 리스트 15  
 메모리 효율  
     생성기 128~129  
     생성기 표현식 132~133  
     \_\_slots\_\_ 162  
 메모리, 객체 위치 40  
 메서드 58, 141  
     내장 타입 60  
     묶인 60, 152  
     안 묶인 60  
     정의 142  
     정적 153  
     클래스 152  
     클래스에서 정의 24  
     타입 57  
     프로퍼티로 처리 152  
     하위 클래스에서 재정의 막기 156  
     호출 프로세스 58  
     @classmethod 장식자 58  
     @staticmethod 장식자 58  
     super() 함수 사용 146  
 메서드 분석  
     다중 상속 147  
     단일 상속 145

- `__mro__` 속성 147  
 메서드 분석 순서, `TypeError` 예외 147  
 메시지 요약 691  
 메시지 전달 507~508  
 동기화 510  
 바이트 베퍼 전달 518  
 정의 507~508  
 코루틴 510, 130  
 프로세스 사이 객체 전달 518  
 메시지 전파, 로그 메시지 443  
 메시지 큐 510  
 코루틴 130  
`multiprocessing` 모듈 513  
 메인 프로그램 실행 179  
 메인 프로그램, `pickle` 모듈 282  
 메인 프로그램으로 실행하는지 확인 179  
 메타클래스 169  
 기술자 172  
 사용 주의 173  
 상속 171  
 성능 향상 244  
 예 172  
 정의 방법 170  
 커스텀 사전 객체 사용 777  
`__new__()` 메서드 사용 66, 157  
`__prepare__()` 메서드 776~777  
 멤버 검사  
 사전 17, 89  
 순서열 80  
 명령줄 옵션 13, 191  
 인터프리터에서 사용 213  
 파이썬 3 784  
 프로그램에서 설정 감지 283  
`optparse`로 파싱 461  
 모듈 26, 175  
 객체 177  
 검색 경로 180  
 동적 로딩 176  
 모듈 객체 타입 57  
 속성 62  
 속성 접근 62  
 실행 가능한 프로그램 작성 176  
 여러 모듈 임포트 176  
 인식 파일 타입 181  
 일회성 실행 176  
 클래스 접근 176  
 타입 61  
 함수 전역 네임스페이스 116  
`doctest`로 스스로를 테스트 225  
`.pyc` 파일 182  
 모듈 검색 경로  
 변경 180  
 환경 변수 설정 215  
 site 모듈 219  
 zip 파일 180  
 모듈 검색 경로 변경 180  
 모듈 내리기 182, 182  
 모듈 로딩 181  
 모듈 재로딩 182, 182  
 모듈에서 검색 경로 180  
 모듈에서 선택적 기호 임포트 178  
 모든 예외 잡기 103  
 무작위 수 생성 312  
 무작위 수, 스레드 315  
 뮐린 메서드 59, 152  
 문서화 문자열 28, 36, 58, 136  
 들여쓰기 36  
 장식자 123, 136~137  
 테스트 224  
 확장 모듈 733~734  
`doctest` 모듈 224  
 -OO 옵션으로 벗김 181  
 XML-RPC 650  
 문서화 문자열, 장식자로 복사 332  
 문자  
 유니코드 33  
 탈출 코드 33  
 문자 치환 51  
 문자열 11  
 내부 표현 33  
 대소문자 변환 50~51  
 로그 메시지 포맷 지정 440  
 메모리 텍스트 파일 435  
 문자 반복 19  
 문자 치환 51  
 바이트 문자열 49, 248  
 바이트 문자열과 유니코드 혼합 84  
 바이트 상수 34  
 변경 가능한 바이트 배열 248  
 부분 문자열 검색 50  
 부분 문자열 대체 50  
 분할 50, 11, 53  
 불변성 50, 82  
 비교 84  
 사전 키 17  
 상수에서 탈출 코드 33  
 상수에서 탈출코드 사용 불가능 34  
 순서열 47  
 유니코드 49, 202  
 이어 붙이기 52  
 인접 상수 연결 32  
 정규 표현식 345  
 정렬과 국제화 84  
 제거 53  
 커스텀 포맷 생성 354  
 타입 확인을 위한 `basestring` 객체 248  
 파이썬 코드 실행 138  
 포맷 지정 7, 51  
 필드로 분할 15, 50  
`format()` 메서드 7, 86  
`format()` 메서드 지정자 86~87  
 HTML에 쓰려고 문자 탈출 662  
 URL-용 인코딩 643  
 XML 문자 참조 탈출 되돌림 722  
 XML에 사용될 탈출 문자 722  
 문자열 보간 86, 199  
 문자열 상수 32  
 문서화 문자열 36  
 바이트 문자열 34  
 소스 코드에 유니코드 문자 37  
 유니코드 문자 33  
 유니코드 인코딩 34  
 문자열 속성, `MatchObject` 객체 351  
 문자열 포맷 지정 84  
 사전 87  
 사전 검색 87

- 속성 검색 87  
 정렬 87  
 채움 문자 88  
 포맷 연산자 % 84  
 포맷 지정자 87  
 % 연산자 코드 84~85  
 format() 메서드 커스터마이즈 89  
 !r 지정자 89  
 !s 지정자 89  
 문자열 포맷 지정자에서 채움 문자 88  
 문자열에서 변수 보간 199  
 문자열에서 앞에 나오는 r 문자, 미가공 문자열 34  
 문자열에서 줄바꿈 탈출 코드 33  
 문자열에서 템 탈출 코드 33  
 문자열을 숫자로 변환 11  
 문자열을 코드로 실행 138  
 문장  
     디버거에서 실행 229  
     여러 문장을 같은 줄에 29~30  
     여러 줄로 나누기 8  
 문장 종료 세미콜론 6  
 미가공 문자열 34  
     백슬래시 규칙 34  
     유니코드 34  
     정규 표현식에서 사용 345  
 미가공 소켓 579  
 미래 기능, 사용 220  
 밑줄, 식별자로 사용 31
- 
- ㅂ
- 바닥 나누기 77  
 바이트 문자열 49  
     메모리 이진 파일 433  
     변경 가능한 바이트 배열 248  
     시스템 인터페이스에서 사용 780  
     유니코드 문자열과 같이 사용 84, 203  
     유니코드로 디코딩 202  
     파이썬 3에서 다른 작동 방식 778  
     파이썬 3에서 시스템 인터페이스 780  
     파이썬 3에서 포맷 지정 없음 778
- 파일 196  
 WSGI 667  
 바이트 문자열과 유니코드 섞기 203  
 바이트 문자열에서 앞에 나오는 b 문자, 문자열 상수 34  
 바이트 상수 34  
 바이트 코드로 컴파일 181  
 바이트, 문자열에서 탈출 코드 33  
 반복 9, 18, 72, 98  
     루프에서 빠져나오기 100  
     반복 변수 98  
     반복 변수의 유효 범위 99  
     사전 값 55  
     사전 키 19  
     순서열 47, 82  
     여러 순서열 100  
     지원 객체 19  
     튜플 풀어헤치기 99  
     파이썬 3에서 프로토콜 변화 783  
     프로토콜 72, 98  
     next() 호환 가능 함수 255  
     반복자, 파이썬 3에서 사용 782  
     반올림 방식 78  
     파이썬 3에서 바뀐 점 79  
     배열, 한 종류 타입으로 생성 319  
     배정밀도 부동 소수점 46  
     배포용 코드 조작화 186~187  
     백슬래시 규칙, 미가공 문자열 34  
     버리는, 정수 나누기 77  
     버전 정보, 인터프리터 285  
     비퍼 이진 I/O 431  
     비퍼, 원형 323  
     비퍼링 없는 파일 I/O 194  
     비퍼링, 생성기 201  
     변경 가능성  
         기본 함수 인수 112  
         사전 키 53  
         제자리 대입 연산자 91  
         참조 횟수 세기 42  
         함수 매개변수 114  
         변경 가능성, 정의 39  
         변경 불가능 타입, 상속 66
- 변경 불가능, 정의 39  
 변경 불가능, 튜플 15  
 변수 6  
     객체 이름 41  
     비인딩과 모듈 임포트 178  
     반복 99  
     유효 범위 115, 116~117  
     이름 규칙 30~31  
     중첩 함수 116  
     클래스 141~142  
     함수에서 전역 변수 바인딩 117  
 변수의 동적 타입 6  
 변환 연산 92  
 병행 프로그래밍 507~508  
 병행성 507  
     다중 처리에 관한 조언 536  
     다중 코어에서 제약 508  
     동기화 기본 기능 508  
     메시지 전달 507~508  
     부작용 114~115  
     생성기에서의 다중 작업 551~552  
     전역 인터프리터 락 509  
     코루틴 551  
     파이썬 프로그램 508  
     확장성 510  
     보간, 문자열에서 값 86  
     ~보다 작거나 같은 연산자 <= 79  
     ~보다 작은 연산자 79  
     ~보다 크거나 같은 연산자 >= 79  
     ~보다 큰 연산자 > 79  
 보안  
     데이터베이스 질의 369  
     marshal 모듈 279  
     pickle 모듈 211, 282  
     XML-RPC 서버 653  
 보안 소켓 계층 598  
 보편 줄바꿈 모드 194  
 복사  
     깊은 복사 43  
     변경 가능 객체 42  
     사전 54  
     얕은 복사 43

참조 횟수 세기 41  
 복사, 순서열 얇은 복사 80  
 복소수 31, 47  
 비교 79  
 cmath library 모듈 309  
 복소수 상수 끝 J 31  
 복합 문자열 포맷 지정 7, 51, 87  
 검색 51  
 \_\_format\_\_() 67  
 부동 소수점 31  
 무작위 수 분포 313  
 복소수와 혼합 47  
 부정확한 표현 12, 300  
 사전 키 17  
 십진수와 비교 301  
 유리수 변환 308  
 유리수로 변환 46  
 이진 표현 46  
 저수준 속성 284  
 정밀도 46  
 표현 46  
 NaN와 Inf 정의 263~264, 309  
 부동 소수점의 부정확한 표현 12  
 부모 클래스 144  
 부분 문자열  
 검색 50  
 존재 여부 검사 82  
 부분 문자열 바꿈 50  
 부작용  
 피해야 하는 이유 114~115  
 함수 114  
 분리자 문자, print() 함수 199  
 분산 컴퓨팅, multiprocessing 모듈 536  
 분할 47  
 다차원 72  
 삭제 48, 82~83  
 특수 메서드 72  
 할당 48, 82~83  
 xrange 객체 53  
 분할 객체 64  
 색인 메서드 72  
 속성 65

분할 삭제 48, 83  
 분할 연산자 [:] 80~81  
 리스트 12  
 문자열 11  
 분할 제거 48  
 분할 타입 64  
 분할 할당, 리스트 12  
 분할, 문자열 15, 53  
 분해 238  
 불리언 값 31, 46  
 불리언 연산자 79  
 불리언 표현식 8, 93  
 평가 규칙 94  
 뷰 객체, 파이썬 3 782  
 브라우저, 파이썬에서 구동 671  
 블로킹 연산, 비동기 네트워킹 577  
 블루투스 프로토콜 578  
 주소 형식 582  
 비교 79  
 객체 40  
 순서열 83  
 약한 참조 298  
 연결 79  
 파이썬 3 783  
 호환성 없는 객체 94  
 비교 연산자 68  
 비동기 I/O 510  
 비동기 네트워킹  
 블로킹 연산 577  
 언제 고려 574  
 비트 negation 연산자 ~ 78  
 비트 or 연산자 | 77  
 비트 xor 연산자 ^ 78  
 비트 연산자 & 78  
 비트 연산자와 하드웨어에 특화된 정수  
 78  
 빅 엔디안 포맷 204  
 빅 엔디안, 포장과 풀어헤치기 359  
 빙 리스트 13  
 빙 사전 17  
 빙 줄 30  
 빼기 연산자 - 77

## ㅅ

---

사용자 디렉터리, 패키지 설치 190  
 사용자 입력 읽기 10, 197  
 사용자 정의 인스턴스 생성 24  
 사용자별 사이트 디렉터리 190, 219  
 패키지 설치 219  
 사이트 설정 파일 218  
 사전 16, 53  
 값 얻기 55  
 갱신 54  
 검색표로 사용 17  
 기본 값 자동 생성 325  
 기본 값으로 검색 17  
 대중 프로세스 공유 528  
 데이터 구조로 사용 17  
 데이터베이스 결과에서 생성 372  
 동등 94  
 리스트로 변환 17  
 문자열 포맷 지정 84, 87  
 복사 55  
 복합 문자열 포맷 지정에서 검색 51, 87  
 빙 사전 정의 17  
 색인 연산자 89  
 성능 18  
 제거 54  
 키 값 53, 89  
 키 반복 17  
 키 얻기 55  
 키워드 인수를 함수에 전달하는 데 사용 113  
 튜플을 키로 89  
 파이썬 3 주의 55  
 파이썬 3에서 뷰 객체 782  
 함수를 값으로 사용 44  
 항목 리스트 55  
 항목 삭제 17, 54, 89  
 항목 접근 17  
 항목 제거 17  
 항목 추가 17  
 항목 할당 89  
 허용되는 키 타입 17

defaultdict 객체와 비교 325  
 dict() 함수와 생성 250  
 \_\_hash\_\_() 메서드 68  
 in 연산자 성능 244  
 shelve 모듈로 영속화 210  
 사전 개선 54  
 사전 내포, 파이썬 3 770  
 사전 순서  
     문자열 84  
     UTF-8 208  
 사전 제거 54  
 사전에서 항목 삭제 17  
 사전을 리스트로 변환 17  
 산술 연산자  
     제자리 74  
     혼합 타입 79  
 산술 특수 메서드 72  
 삼중따옴표 문자열 11  
     변수 보간 200  
 상대적인 패키지 임포트 184  
 상속 24, 144  
     개인 메서드 156  
     내부 최적화 287  
     내장 타입에서 24  
     다중 상속 146  
     메서드 분석 순서 147  
     메서드 재정의 막기 156  
     메타클래스 171  
     변경 불가능 타입 66  
     상위클래스 메서드 호출 146  
     상위클래스 초기화 145  
     속성 바인딩 145  
     예외 106  
     추상 기반 클래스 168  
     isinstance() 함수 41  
     issubclass() 함수 165  
     \_\_mro\_\_ 속성 클래스 147  
     \_\_slots\_\_ 과 상호작용 162  
 상위 클래스 144  
     메서드 호출 146  
     super() 함수 146  
     상호 배제 락 540  
 새로운 예외 정의 106  
 색인 연산자 [] 48, 80  
     리스트 12  
     문자열 11  
     순서열 81  
     튜플 15  
     색인, 0부터 11  
     생략 가능한 함수 인수 20, 112  
     None 46  
 생산자 소비자  
     스레드 세마포어 542  
     스레드 조건 변수 545  
     코루틴 23  
     큐 516  
     파이프 519~520  
 생성기 20, 123~124  
     다중 작업 예 552  
     단기 64  
     메모리 효율 128~129  
     반복에서 break문 124  
     병행 프로그래밍 551  
     실사용 예 128  
     실행 모델 21  
     예외 던지기 64  
     재귀 135  
     처리 파이프라인 21  
     GeneratorExit 메서드의 처리 264  
     I/O 201  
     WSGI 202  
 생성기 객체 62, 64  
     속성 64  
 생성기 표현식 132  
     리스트 내포와 차이점 132  
     리스트로 변환 132  
     조건 표현식 96  
 생성기 함수, 컨텍스트 관리자 109  
 서버 프로그램 553  
     코루틴 예 571  
     HTTP에서 접근 제약 예 623  
     SocketServer 모듈 예 603  
     TCP 예 555  
     UDP 예 597  
 선언형 프로그래밍 133  
 설정 파일 408  
     변수 치환 412  
     파이썬 스크립트와 다른점 411~412  
     logging 모듈 453  
 설정 파일 읽기 408  
 성능  
     생성기 표현식 132~133  
     이진 파일 I/O 433~434  
     타입 검사 41  
     logging 모듈 454  
 세기, 루프 99  
 세미콜론 ; 30  
 세미콜론으로 문장 종료 6  
 셸 명령  
     출력 모으기 408  
     파이썬에서 흉내 391  
 셸 파이프, 생성기와 유사점 128~129  
 소스 배포판 생성 188  
 소스 코드 인코딩 37  
     파이썬 3 769  
 소켓  
     네트워크 주소 지정 579  
     메서드 588  
     종류 579  
     주소 체계 578  
     select()로 플링 565  
     소켓, 정의 553  
     속도 향상, 정의 237  
     속성  
         개인 155  
         객체 40  
         기술자 70, 154  
         복합 문자열 포맷 지정에서 검색 51  
         캡슐화 155  
         프로퍼티로서 계산 141, 151  
         함수에 사용자 정의 137  
         \_\_init\_\_() 메서드에서 생성 143  
         \_\_slots\_\_으로 이름 제한 162  
         속성 검색 문자열 포맷 지정에서 87  
         속성 바인딩  
         과정 69

- 메서드 58  
 사용자 정의 객체 160  
 상속 144  
 인스턴스와 클래스 143  
 클래스에서 재정의 161  
 특수 메서드 69  
 속성 바인딩 연산자 . 40, 91  
 쇠적화 242  
 속성 삭제, 인스턴스 161  
 속성 이름 제한 162  
 속성 할당, 인스턴스 160  
 순서열 47  
     무작위 원소 고르기 313  
     무작위 표본 추출 313  
     문자열 포맷 지정에서 인덱싱 87  
     반복 47, 82  
     복사 80  
     복합 문자열 포맷 지정에서 검색 51  
     분할 연산자 81  
     분할 할당 48  
     비교 83  
     얕은 복사 80  
     연결 80  
     연산자 80  
     음수 인덱스 81  
     특수 메서드 70  
     풀어헤치기 80~81  
     항목 할당 48  
     확장 분할 81~82  
     in 연산자 80  
 순서열 동시 반복 99  
 순서열 항목 삭제 48  
 순서열을 리스트로 변환 93  
 순서열을 집합으로 변환 16  
 순서열을 튜플로 변환 93  
 순환 데이터 구조, \_\_del\_\_() 메서드 158  
 순환 참조  
     쓰레기 수집 273  
     약한 참조로 피하기 159, 296  
 순환 참조, 쓰레기 수집 42  
 숫자 데이터, 문자열 11  
 숫자 상수 31  
     숫자 타입 46  
     숫자 타입 강제 타입 변환 79  
     숫자 타입 계층 구조 169, 311  
     숫자 타입의 강제 타입 변환 79  
     숫자, 새로운 타입 정의 예 163  
 스크립트 이름 191  
 스택 크기, 스레드에서 548  
 스택 프레임 63  
     역추적 64  
 스트림 579  
     시간 조작 413  
     시간 측정 236, 236  
     시간 파싱 423  
     시간과 날짜 파싱 501  
 시그널  
     목록 493  
     생성기의 close() 메서드 124~125  
     생성기의 throw() 메서드 126  
     스레드와 섞기 494  
     시그널 처리 492  
     시그널, 세마포어와 541  
     시스템 명령 실행 408  
     popen() 함수 483  
     subprocess 모듈 495  
     system() 함수 484  
     시스템 에러 코드 423  
     시스템 호출, os 모듈 466  
     시스템의 CPU 개수 535  
     시작 스크립트, 대화식 모드 214  
     식별자 30  
         1급 데이터 44  
         대소문자 구별 30~31  
         밑줄 사용 31  
         예약어 30  
         파이썬 3에서 유니코드 사용 769  
     신뢰성 있는 데이터그램 579  
     신원 연산자 is 94  
     실수 307  
     실제 실행 시간, 얻기 236  
     실행 모델 97  
     심벌릭 링크, 파일 이름 검사 489  
     십진수 299  
         부동 소수점 12  
     쌍, 사전에서 리스트 생성 55  
     써드파티 라이브러리, 파이썬 3 768  
     써드파티 패키지  
     사용자별 사이트 디렉터리에 설치 190,

- 219  
설치 189~190  
C/C++ 코드 189~190  
sys.path 변수 190  
써드파티 패키지 설치 189~190  
사용자 디렉터리에 190  
쓰레기 수집 41~42, 271  
과정 설명 271  
관찰자 패턴 예 158  
순환 참조 42  
프로그램 종료 221  
\_\_del\_\_() 메서드 158  
\_\_del\_\_() 메서드 문제 271  
쓰레기 수집 강제 271  
쓰레기 수집 비활성화 271
- 
- 
- 안 묶인 메서드  
파이썬 3 60  
알람 491  
암묵적 타입 변환, 없음 75  
암호 해싱 함수 691  
압축  
파일 385, 390  
zlib 압축 404  
약한 참조 159, 296  
얕은 복사 43  
리스트 48  
사전 55  
순서열 복사 80  
어휘 유효 범위 116  
업로드  
CGI 스크립트에서 파일 662  
FTP 서버로 파일 업로드 614  
POST로 HTTP 서버에 파일 업로드 621  
pypi에 패키지 업로드 190  
에러 메시지 191  
에러 역추적 출력 제한 286  
에러 코드, 시스템 에러 목록 423  
여러 예외 잡기 103  
여러 인스턴스 생성 메서드 정의 149
- 역순 피연산자 메서드, 호출 164  
역추적  
출력량 제한 286  
traceback 모듈로 생성 290  
역추적 객체  
속성 64  
스택 프레임 64  
역추적 메시지 25  
역호출 함수 117~118  
lambda 134  
연결  
리스트 13  
문자열 11  
인접 문자열 상수 33  
연결된 비교 79  
연결된 예외, 파이썬 3 774  
연관 배열 16, 53  
연산자 35, 77  
산술 72  
우선순위 94  
연산자 결합 규칙 94  
연산자 오버로딩 65  
예 163  
타입 강제 변환 164  
피연산자 순서 163  
피연산자 순서 반대 73  
연산자, 불리언 표현식 8, 93  
영속 사전, shelve 모듈 209  
예약어 30  
예외 25, 101  
값 25, 101  
경고와 차이점 267  
계층 106  
내장 목록 104  
락 26  
모두 잡기 103  
모든 예외 잡을 때 주의 103  
무시 103  
부합 규칙 102  
새로운 정의 106  
성능 243  
속성 262~263
- 시스템 에러용 에러 코드 423  
여러 타입 잡기 102  
전파 102  
처리 25  
최적화 전략 243  
최종 예외 다시 발생 102  
최종 예외 지우기 288  
파이썬 3에서 연결 774  
finally문 104  
예외 계층 구조 104~105  
예외 무시 103  
예외 처리, 확장 모듈 746  
오른쪽 이동 연산자 >> 78  
오리 타입화 149  
오버플로우, 정수에서 없음 78  
~와 같은 연산자 == 79, 94  
~와 다른 연산자 != 79  
외부 함수 인터페이스, ctypes 모듈 756  
왼쪽 이동 연산자 <  
요청 636  
우선순위 큐 333  
운영체제, 스케줄링 508  
원격 프로시저 호출  
multiprocessing 모듈 520~521  
XML-RPC 646  
원자적 연산, 분해 238  
웹 브라우저 구동 671  
웹 서버  
커스텀 요청 처리 627~628  
파이썬에서 독립 실행 625  
웹 페이지에 유니코드 문자 포함 204  
웹 프레임워크 665  
템플릿 문자열 200  
웹 프로그래밍 655  
윈도우 407  
레지스트리 접근 502, 216  
레지스트리 접근 216  
메시지를 이벤트 로그로 보냄 446  
사용자별 사이트 디렉터리 219  
에러 코드 목록 425  
파이썬 프로그램을 더블 클릭 218  
파일 이름에 드라이브 문자 491

- 파일 잡금 459  
 프로그램 실행 5  
 distutils로 이진 배포판 생성 189  
 multiprocessing 모듈 메인 프로그램  
 513  
 multiprocessing 모듈로 프로세스 포크  
 537  
 원도우 설치 프로그램 생성 189  
**유니코드**  
 32 비트 문자 코드 포인트 사용 50  
 대리자 쌍 33, 50  
 문자 속성 데이터베이스 209  
 문자 인코딩과 디코딩 51  
 문자열 상수에서 문자 지정 33  
 문자열 상수에서 인코딩 34  
 문자열 정규화 209  
 바이트 문자열 혼합 84  
 바이트 순서 표시 344~345  
 소스 코드 인코딩 37  
 코드 포인트 33  
 파이썬 2과 파이썬 3 33  
 파일 I/O 204  
 흔히 사용되는 인코딩 206  
 BOM 문자 205  
 XML 205  
**유니코드 문자 데이터베이스** 360  
**유니코드 문자, 표현** 49  
**유니코드 문자열** 49  
 네트워크 프로그램에서 인코딩 556  
 바이트 문자열 혼합 203  
 분해 362  
 에러 처리 옵션 203  
 인코딩과 디코딩 202  
 정규 표현식 345  
 정규화 363  
 처리 202  
 흔히 사용되는 인코딩 203  
 WSGI 668  
**유니코드 문자열 상수** 33  
**유니코드 문자열 정규화** 209  
**유니코드 문자열에서 앞에 나오는 u 문자,**  
 문자열 상수 33
- 유니코드 타입 45  
**유닉스**  
 #! 실행 프로그램 5  
 사용자별 사이트 디렉터리 219  
 시간 애포크 정의 498  
**유닉스 도메인 프로토콜** 578  
 주소 형식 581  
**유닉스 시그널 이름 목록** 493  
**유닉스 시스템 로그, 메시지 보내기** 447  
 유연한 날짜와 시간 파싱 423  
**유효 범위 규칙**  
 리스트 내포에서 반복 변수 131  
 메서드에서 self 매개변수 143  
 모듈 임포트 178  
 반복 변수 99  
 어휘 유효 범위 함수 116  
 클래스 143  
 함수 변수 20, 115  
 음수 인덱스 81  
 응답 속도, 비동기 네트워킹 575  
 응용 프로그램 로깅 436  
 응용 프로그램, WSGI 666  
 이름 변형, 개인 속성 155  
 이름 있는 튜플  
 튜플로 사용 325  
 표준 라이브러리에서 사용 327  
**이메일 메시지**  
 작성 686  
 작성 및 전달 예 690~691  
 전달 예 633~634  
 파싱 682  
 이메일 메시지 작성 686  
 이메일 전달, 예 633~634, 690~691  
**이벤트 루프**  
 코루틴 130  
 asyncore 모듈 560  
 이벤트 주도 I/O 510  
 시그널 폴링 492  
 언제 고려 574  
 이식성 있는 파일 이름 조작 488  
 이중 밑줄로 시작하는 식별자 31  
 이진 배포판 생성 188~189
- 이진 배포판, distutils로 생성 188~189  
 이진 자료 구조, 포장과 풀어헤치기 357  
 이진 정수 상수 31  
 이진 파일 431  
 베패 I/O 431  
 줄 기반 함수 사용 주의 433~434  
 이진 파일 모드 193  
 이차원 리스트 13, 15  
 익명 함수 134  
**인스턴스** 141  
 생성 66, 142, 157  
 속성 삭제 161  
 속성 할당 160  
 정의 39  
 타입 60  
 피클링 281  
 호출 가능 객체 60  
**인스턴스 메서드** 58, 142  
**인스턴스 생성** 142  
 관여 단계 157  
**인스턴스 속성 삭제** 161  
**인용, URL에서 문자** 643  
 인접 문자열 상수, 연결 33  
 인증, URL 처리 639  
 인코딩, 소스 코드 37  
**인터프리터** 3  
 인터프리터 명령줄 옵션 213~214  
 인터프리터 명령줄 옵션 설정 감지 283  
 인터프리터 프롬프트 216  
 인터프리터 환경 변수 214  
 인터프리터, -t -tt 옵션 30  
 읽기 평가 루프 3  
 임계 구역, 락 508  
 임베딩  
 파이썬 타입을 C로 변환 754  
 C에서 함수 호출 752  
 임시 파일 398
- ㅈ
- 
- 자료 구조  
 이름 있는 튜플 325

- 자바 765  
 클래스 시스템에서 다른 점 143  
 자바스크립트, 웹 업 윈도우 예 657  
 자유 변수, 함수 118  
 작업 디렉터리 변경 467  
 작업 스케줄러, 코루틴과 select() 예 567  
 작업 풀, 프로세스 521  
 작업, 코루틴 22  
 잘 알려진 포트 번호 554  
 잠금  
     데드락 피하기 546  
     윈도우에서 파일 459  
     임계 구역 508  
     파일 428  
     multiprocessing 모듈 526  
     threading 모듈 540  
 잠자기 500  
     시그널 수신 때까지 492  
 장식자 25, 121  
     다중 37  
     문서화 문자열 123, 136  
     배치 36, 121  
     사용자 정의 함수 속성 123, 138  
     성능 향상 244  
     예 121  
     인수를 가지는 122  
     재귀 함수 123, 136  
     클래스 정의에 적용 122, 173  
     함수 속성 복사 332  
 재귀 135  
     생성기 함수 135  
     장식자 123, 136  
     재귀 한도, 변경 135, 289  
     재진입 락 540  
     저수준 파일 조작 470  
     적용 순 평가 92  
     전역 변수 116  
     모듈 178~179  
     스택 프레임에서 저장소 63  
     함수에서 수정 20  
     C와 Fortan 차이 179  
     eval() 139
- 전역 인터프리터 락 509, 548  
 확장 모듈에서 해제 748~749  
 multiprocessing 모듈 548  
 절대 값 78  
 절대 임포트 185  
     상대 임포트와 비교 185  
     파이썬 3 785  
 접근 제어 지정자, 없음 155  
 정규 표현식  
     미가공 문자열 사용 345  
     페턴 문법 346  
     re 모듈 345  
 정렬  
     객체 요구사항 69  
     리스트 제자리에서 49  
     역순 49  
     작동 방식 변경 49  
     operator 모듈 사용 340  
 정밀도, 부동 소수점 46  
 정수 31  
     16진수 문자열 생성 93  
     16진수, 8진수, 2진수로 지정 31  
     2의 보수 표현 78  
     진 정수로 변환 46  
     무작위 수 생성 313  
     범위 46  
     사전 키 17  
     오버플로우 78  
     자동으로 long으로 변경 31  
 정수 값 범위 46  
 정수 나누기, 파이썬 3 782, 782  
 정수에서 0b로 시작, 2진수 31  
 정수에서 0o로 시작, 8진수 31  
 정수에서 0x로 시작, 16진수 31  
 정수와 진 정수 타입 통합 46  
 정적 메서드 25, 58, 149, 153  
     실용적인 용도 150  
 정지, 스레드 546  
 제거  
     문자열 53  
     문자열에서 제어 문자 제거 50  
     제곱 연산자 \*\* 77
- 제어 문자, 문자열에서 제거 50  
 제자리 대입 연산자 91  
 제자리 산술 연산자 73  
 제자리 수정  
     리스트 49  
     집합 57  
 제자리 파일 수정 194  
 조건 변수 544  
 조건 표현식 95  
 조건문 8, 98  
 조건문으로 여러 경우 검사 8  
 종료  
     스레드 546~547  
     쓰레기 수집 없이 221  
     쓰레기 수집 없이 즉시 481  
     정리 함수 등록 269  
     프로그램 221  
     sys.exit() 함수 288  
 주 스레드 507  
 주석 5, 30  
 주소 체계, 소켓 578  
 주소, 네트워크 579  
     줄 이음 문자 \ 8, 29, 35  
     줄 이음, 괄호, 대괄호, 중괄호 29  
     줄 읽기, 파일 9  
     줄바꿈 문자, 유닉스와 윈도우 다른점 194  
     줄바꿈 변환 비활성화 194  
     줄바꿈 생략, print문 198  
     줄바꿈으로 문장 종료 6  
     중괄호, 사전 16  
 중단점  
     디버거에서 설정 231  
     직접 설정 230  
 중첩 리스트 13  
 중첩 클래스, pickle 문제 281  
 중첩 함수 116  
     클로저 119  
 지수, 부동 소수점 범위 46  
 지역 변수 정의하기 전에 사용 117  
 지연 실행, 스레드 사용 539  
 지연 평가 119  
 진리 값 9

진리 값 검사 94  
 질의, 데이터베이스에 안전하게 작성 369  
 집합 16  
     생성 16  
     교집합 연산자 16  
     대칭 차집합 연산자 16  
     동등 94  
     반복 가능 객체에서 생성 55  
     제자리 수정 57  
     차집합 연산자 16  
     합집합 연산자 16  
     항목 개수 90  
     항목 제거 16  
     항목 추가 16  
     집합 내포, 파이썬 3 769  
     집합 상수, 파이썬 3 769  
     집합 이론, 리스트 내포와 유사점 133  
     집합 타입 45, 55, 90

## ㅋ

차집합 연산자 - 90  
 차집합 연산자 -, 집합 16  
 참조 횟수 세기 41, 157  
     메모리 사용 237  
     변경 가능 객체 43  
     복사 41  
     얻기 42  
     확장 모듈 748~749  
     del문 42  
 최적화  
     계층 추가 효과 240  
     내부 타입 캐시 287  
     내장 타입 239  
     로그 메시지 포맷 지정 440  
     리스트와 배열 객체 321  
     메모리 사용 측정 237  
     반복 시간 측정 236  
     배열 객체 321  
     분해 238  
     사용자 정의 클래스 241  
     사전 검색 244

사전과 클래스 241  
 속도 향상 239  
 속도 향상 정의 237  
 속성 바인딩 242~243  
 시간 측정 236  
 역호출 함수 정렬 340  
 예외 243  
 인스턴스 생성 241  
 장식자와 메타클래스 244  
 클래스의 \_\_slots\_\_ 속성 242  
 튜닝 전략 239  
 함수형 프로그래밍 244  
 deque 객체 324  
 dict() 함수 241  
 I/O 폴링 영향 577  
 io 모듈 사용 436  
 logging 모듈 454  
 map()과 filter() 함수 244  
 marshal 과 pickle 278  
 multiprocessing 풀의 사용 524  
 operator 모듈 사용 339  
 select() 함수 575  
     \_\_slots\_\_ 속성 클래스 162  
 최적화 모드, 환경 변수로 활성화 214  
 최종 예외 다시 발생 102  
 최종 예외 삭제 288  
 추상 기반 클래스 41, 167, 317  
     검사 수행 168  
     기준 클래스 등록 168  
     숫자 타입 311  
     예 318  
     인스턴스 생성하면 에러 168  
     컨테이너 객체 327  
     특수 메서드 69  
     파일 I/O 435  
     하위 클래스 메서드 호출 167  
     추상 기반 클래스 인스턴스 생성 168  
 출력  
     날짜와 시간 500  
     커스텀 포맷 생성 354  
     포맷 지정 7  
     출력 결과 변경, 대화식 모드 217

출력할 수 없는 문자, 문자열 상수에서 지정 33

## ㅋ

캡슐화 155  
 커링, 부분 함수 평가 92  
 커스텀 문자열 포맷 지정기 생성 354  
 컨테이너 객체 35  
     정의 39  
     참조 횟수 세기 41  
 컨테이너 추상 기반 클래스 328  
 컨텍스트 관리 프로토콜 76  
 컨텍스트 관리자 75, 107, 546  
     생성기로 정의 330  
     중첩 331  
     decimal 모듈 305  
     컬렉션, 정의 39  
     컴파일러, 없음 223  
 코드 객체 62  
     속성 63  
     compile() 함수와 생성 139  
 코드 실행, 모듈 175~177  
 코드 이주  
     실용적인 전략 789, 789  
     파이썬 2에서 3으로 785, 785  
 코드 포인트, 유니코드 33  
 코드, 문자열 실행 138  
 코루틴 22, 125  
     값 전달 및 반환 127  
     고급 예 567  
     네트워크 프로그래밍에 사용 574  
     다른 코루틴으로 제어를 넘김 569  
     다중 작업 예 552  
     메시지 전달 130, 510  
     병행 프로그래밍 551  
     병행성 130  
     비동기 I/O 처리 567  
     실사용 예 129  
     예 22  
     작동 방식 126  
     재귀 135

- 
- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>호출 스택 구축 569<br/>next() 메서드 사용 125<br/>select()로 작업 스케줄러 567<br/>콘솔 윈도우, 윈도우 218<br/>쿠키<br/>  쿠키 지원으로 URL 추출 640<br/>  HTTP 628<br/>큐<br/>  다수의 소비자와 생산자 517<br/>  다중 프로세스 공유 528<br/>  메시지 전달 510<br/>  스레드 프로그래밍 549<br/>  스레드와 사용된 예 550~551<br/>  우선순위 333<br/>  코루틴 130<br/>  큐, 원형 323<br/>클라이언트 프로그램 553<br/>  TCP 예 556<br/>  UDP 예 597<br/>클래스 24<br/>  개인 멤버 31<br/>  구형 170<br/>  균일 접근 원칙 152<br/>  기술자 속성 70, 154<br/>  내장 타입에서 상속 24<br/>  네임스페이스 142<br/>  다중 상속 146~147<br/>  데이터 저장을 사전과 비교 241<br/>  메모리 관리 156<br/>  메서드 정의 24<br/>  메서드에서 self 매개변수 143<br/>  메서드에서 super() 함수 146<br/>  메타클래스 169<br/>  모듈에서 접근 176~177<br/>  상속 24, 144<br/>  상속 최적화 287<br/>  속성 바인딩 규칙 143<br/>  속성 바인딩 재정의 161<br/>  속성 접근 커스커마이즈 69<br/>  유효 범위 규칙 143<br/>  인스턴스 생성 24, 66, 142<br/>  작동 방식 변경 66</p> <p>장식자 적용 122, 173<br/>접근 제어 지정자, 없음 155<br/>정적 메서드 25<br/>최적화 241<br/>추상 기반 클래스 167, 317<br/>클래스 메서드 249<br/>타입 57<br/>특수 메서드 66<br/>피클링 279<br/>호출 가능 객체 61<br/>혼합 148<br/>C++와 자바 차이점 143<br/>  __del__() 메서드 쓰레기 수집 271<br/>  __init__() 메서드 142<br/>  __init__() 메서드와 상속 145<br/>object 기반 클래스 144<br/>pickle 모듈 지원 282<br/>  __slots__ 속성 162<br/>  __slots__의 성능 242<br/>클래스 메서드 58, 149<br/>  속성 바인딩 151<br/>  실용적인 용도 150<br/>클래스 몸체 실행 142, 169<br/>클래스 변수 141<br/>  모든 인스턴스 공유 142<br/>클래스 장식자 122, 173<br/>클로저 117~118<br/>  래퍼 120<br/>  장식자 121<br/>  중첩 함수 119<br/>  클래스 속도 향상 120</p> <p>키<br/>  사전 53<br/>  사전에 허용되는 타입 17</p> <p>키 색인 연산자 [] 54<br/>  사전 17</p> <p>키보드 인터럽트 198<br/>키워드 인수 20, 113<br/>  위치 인수와 혼합 113<br/>키워드 전용 인수, 파이썬 3 773</p> | <p>ㅌ</p> <hr/> <p>타임아웃, 알람 시그널로 구현 494<br/>타입 57<br/>  내장 45, 247<br/>  부동 소수점 46<br/>  불리언 46<br/>  사전 53<br/>  소켓 579<br/>  정수 46<br/>  집합 55<br/>  타입 57<br/>  호출 가능 57<br/>  frozenset 55<br/>타입 강제 변환, 연산자 오버로딩 164<br/>타입 객체 61<br/>타입 검사<br/>  객체 41<br/>  대리자 객체 문제 165<br/>  메타클래스 예 172<br/>  성능 영향 41<br/>타입 계층 구조 169<br/>타입 변환 92<br/>  데이터 파일에서 열 44<br/>  암묵적 타입 변환 없음 75<br/>  특수 메서드 75<br/>탈출 코드<br/>  문자열 상수 33<br/>  문자열 상수에서 사용 불가능 34<br/>  출력할 수 없는 문자 33<br/>테스클릿 551<br/>  비동기 I/O 567<br/>탭, 들여쓰기 30<br/>테스트<br/>  단위 테스트 226<br/>  문서화 문자열 224<br/>  doctest 모듈 224<br/>  doctest 한계 225<br/>  텍스트 I/O 434<br/>  텍스트 대체, 문자열의 replace() 메서드 50<br/>  텍스트 파일 모드 193</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

통계, 무작위 수 분포 313  
 튜닝 전략 239  
 템플 14  
 1개 요소 템플 14  
 팔호 생략 14  
 데이터 구조로 사용할 때 문제점 325  
 레코드 표현 15  
 리스트 15  
 리스트 내포에서 사용 131  
 메모리 절약 15  
 문자열 포맷 지정 84  
 반복에서 풀어헤치기 99  
 분할 14  
 불변성 14, 82  
 비교 83  
 사전 키 17, 90  
 사전에서 리스트 생성 55  
 색인 14  
 순서열 47  
 연결 14  
 이름 있는 속성으로 생성 325  
 파이썬 3에서 풀어헤치기 769  
 표준 라이브러리에서 사용 327  
 템플 풀어헤치기, for 루프 16  
 특수 메서드 24, 65  
 특수 문자 35

---

ㅍ

파생 클래스 144  
 파싱  
 명령줄 옵션 191, 461  
 CGI 스크립트에서 폼 필드 660  
 CSV 파일 677~678  
 ElementTree에서 큰 XML 문서 717~718  
 email 메세지 682  
 HTML 693  
 robots.txt 파일 645  
 URL 641  
 XML 703  
 파이썬 2,6에서 print() 함수 활성화 199  
 파이썬 2와 3 동시 지원 790

파이썬 3  
 2to3 도구 785~788  
 나누기 연산자 77  
 네트워크 프로그래밍 556  
 누가 사용하나 767  
 대화식 모드 인코딩 문제 216  
 메타클래스 170, 776~777  
 명령줄 옵션 784  
 바이트 문자열과 시스템 인터페이스 780  
 바이트 문자열의 다른 작동 방식 778  
 반복자 프로토콜 783  
 비교 783  
 사전 내포 770  
 사전 연산 55  
 사전에 뷔 객체 782  
 생성기 변경 123  
 생성기의 next() 메서드 64  
 식별자에 유니코드 문자 769  
 써드파티 라이브러리 768  
 안 뮤인 메서드 60  
 연결된 예외 774  
 이주 위험 777  
 재조직화된 네트워크 모듈 611  
 전환 767  
 절대 임포트 785  
 정수 나누기 782  
 집합 내포 769  
 집합 상수 769  
 추상 기반 클래스 167  
 키워드 전용 인수 773  
 파이썬 2에서 사용할 수 있는 내장 함수 268  
 파이썬 2와 3 동시 지원 790  
 파이썬 2와 비호환성 767  
 파일 196  
 파일 이름 784  
 표준 라이브러리 구성 784  
 표현식으로서 Ellipsis 774  
 함수 주석 771  
 확장 모듈에서 차이점 734  
 확장된 반복 가능한 객체 풀어헤치기 770  
 환경 변수 784  
 효과적인 포팅 전략 789  
 ASCII로 객체 출력 248  
 commands 모듈 408  
 encode()와 decode() 메서드 778  
 exception 속성 263  
 exec() 함수 781  
 filter() 함수 252  
 I/O 시스템 429, 780  
 import문 185  
 map() 함수 255  
 \_\_next\_\_() 메서드 783  
 nonlocal문 771  
 open() 함수 194, 256~257, 344  
 print 문법 오류 4  
 print() 함수 257, 781  
 raw\_input() 함수 258  
 round() 함수 258  
 socketserver 모듈 602  
 super() 함수 146, 259~260, 775  
 types 모듈 293  
 unicode() 함수 제거 261  
 WSGI 668  
 xrange() 함수 제거 53, 261  
 xrange()와 range() 함수 18  
 zip() 함수 100, 261  
 파이썬 3에서 텍스트와 바이트들 778  
 파이썬 어플리케이션 실행 217  
 파이썬 인터프리터 5  
 파이썬 타입을 C로 변환 754  
 파이썬으로부터 C로 타입 변환 737  
 파이썬을 계산기로 사용 4  
 파이프, subprocess 모듈로 생성 497  
 파이프라인, 생성기 128~129  
 파이프라인과 생성기 21  
 파일 9  
 메모리 매핑 455  
 메서드 195  
 미가공 이진 I/O 431  
 베티 이진 I/O 431  
 베티 크기 194

- 복사 391  
 비교 387  
 생성 시간 489  
 속성 197  
 쓰기 193  
 열기 9, 193~194  
 윈도우에서 잠금 459  
 유니코드 디코딩으로 열기 204  
 유니코드로 디코딩 204  
 임시 398  
 저수준 시스템 호출 470  
 저수준 제어 427  
 절대 경로 488  
 조작 함수 475  
 존재 여부 검사 488  
 줄 반복 19  
 최근 수정 시간 489  
 크기 489  
 타입 193  
 탐색 196  
 파이썬 3 196  
 파일 끝 감지 195  
 파일 모드 설명 193  
 파일 시스템에서 찾기 480  
 파일 포인터 196, 432  
 한 줄씩 읽기 9  
 bz2 압축 385  
 CSV 파싱 677~678  
 gzip 압축 390  
 io 라이브러리 모듈 문제점 436  
 softspace 속성과 print문 198  
 파일 I/O 9  
 파일 같은 객체 149  
 파일 기술자 427  
 조작 함수 470  
 파일 끝 감지 195  
 파일 모드, open() 함수 193  
 파일 복사 391  
 파일 업로드, CGI 스크립트 661  
 파일 이름  
     디렉터리 기반 이름으로 나누기 488  
     디렉터리 여부 검사 489  
 링크 여부 검사 489  
 셸 와일드카드에 부합 389  
 윈도우 드라이브 문자 491  
 이식성 있는 조작 488  
 절대 경로 488  
 존재 여부 검사 488  
 파이썬 3 784  
 파일 잠금 428  
 윈도우 459  
 sqlite3 모듈 373  
 파일 제거 478  
 파일 줄 구분 문자 466  
 파일 찾기 480  
 파일 포인터 이동 196  
 파일에 무가공 I/O 431  
 파일에 쓰기 196  
 패키지 183  
     상대적인 임포트 184  
 패키지 설치 188~189  
 패턴 문법, 정규 표현식 346  
 평가  
     연산자 결합 우선순위 94~95  
     우선순위 94~95  
     함수 인수 92  
     평가 순서 94~95  
     순서 재정의 96  
     포맷 지정 문자열 51, 85  
     포맷 지정 출력 7, 84~85, 198  
     포맷 지정, 로그 메시지 440, 449  
     포맷 지정자  
         정렬 문자 87  
         채움 문자 88  
         커스터마이즈 89  
         필드 중첩 89  
         format() 메서드 문자열 86~89  
 포맷 코드  
     날짜와 시간 500  
     문자열 포맷 지정 연산자 % 84~85  
 포장  
     이진 데이터 구조 357  
     튜플 14  
 포트 번호  
 네트워크 프로그램 554  
 잘 알려진 목록 554  
 포함 검사, in 연산자 9  
 폴링 565  
 성능 577  
 표준 I/O 스트림 197  
 기본 인코딩 설정 215  
 통합 개발 환경 198  
 표준 I/O의 기본 인코딩 설정 215  
 표준 라이브러리 구성, 파이썬 3 784  
 표준 예외 191  
 표준 입력과 출력 10  
 표지, 큐와 사용 517, 551  
 표현식 5  
 풀어헤치기  
     순서열 80~81  
     이진 데이터 구조 357  
     튜플 14  
 프레임 객체 63  
 속성 63  
 프로그래밍 예러, 컴파일러 검사 없음 223  
 프로그램 구조 97  
 프로그램 배포 186  
 프로그램 실행 5  
 프로그램 실행 모델 97  
 프로그램 실행, 메인 프로그램 179  
 프로그램 작성 4  
 프로그램 종료 5, 221, 288  
 쓰레기 수집 221  
 얹기로 221, 481  
 정리 함수 등록 269  
 NameError 예외 무시 221  
 프로그램 줄 구조 29  
 프로세스  
     결합 510  
     데몬 510  
     스케줄링 508  
     시그널 보내기 482  
     작업자 풀 521  
     정의 507  
     종료 481, 497, 510  
     파이프로 연결 518

- 프로세스 id, 얻기 468  
 프로세스 사이 통신 507~508  
 프로파일링 234  
     보고서 해석 235  
 프로퍼티 142  
     개인 속성 156  
     균일 접근 원칙 152  
     메서드에서 사용 152  
     설정 및 삭제 함수 153  
     정의 152  
     \_\_setattr\_\_() 메서드 160  
 프록시 객체  
     타입 검사 문제 165  
     multiprocessing 모듈 527, 530  
 프록시 함수 114  
 프롬프트  
     대화식 모드 216  
     변경 217  
 피연산자 반대, 연산자 오버로딩 73  
 피연산자 순서, 연산자 오버로딩 163
- 
- ㅎ
- 하위 클래스 144  
 하위 프로세스 실행 495  
     예 497  
 하위 프로세스, 정의 507  
 한 종류 타입 배열 319  
 한 줄에 여러 문장 작성 30  
 할당  
     변수 6  
     인스턴스 속성 160  
     제자리 연산자 74  
     중첩 함수 내 변수 할당 116  
     참조 횟수 세기 41  
     확장 74  
 함수 19  
     가변 길이 키워드 인수 113~114  
     객체 117~118  
     기본 값 할당 111~112  
     기본 인수 20, 111~112  
     내장 247
- 디버거에서 실행 229  
 래퍼 생성 120  
 매개변수 전달 114  
 문서화 문자열 58, 136  
 변경 가능한 매개변수 114  
 변수 인수 112  
 부분 평가 92, 331  
 부작용 114  
 사용자 정의 58  
 사용자 정의 속성 138  
 사용자 정의 타입 57  
 사전 값 44  
 생략 가능한 인수와 None 46  
 생성기 20  
 속성 58  
 속성과 장식자 122  
 속성을 장식자로 복사 332  
 스레드 지원 실행 539  
 여러 값 반환 20, 115  
 역호출 117~118  
 유효 범위 규칙 20, 115  
 익명 134  
 인수 평가 92  
 임의 개수 인수를 받는 예 113  
 자유 변수 118  
 장식자 121  
 장식자와 속성 137  
 재귀 135  
 재귀 한도 변경 135, 289  
 전역 변수 수정 20  
 정의 111  
 종료 함수 269  
 중첩 116, 119  
 코루틴 22  
 클로저 117~118  
 키워드 인수 20, 113  
 타입 내장 59  
 파이썬 3에서 주석 771  
 피클링 282  
 호출 19, 112  
 \_\_doc\_\_ 속성 28  
 func\_\* 속성 이름 변환 58
- lambda 연산자 134  
 함수 결과 캐싱 298  
 함수 정의 19  
 함수 호출 19, 112  
 함수 호출 연산자 () 91  
 함수기 75  
 함수에 전달되는 매개변수 114  
 함수에서 여러 값 반환 20, 115  
 함수에서 전역 변수 수정 20  
 합집합 연산자 | 90  
 합집합 연산자 |, 집합 16  
 항목 제거 48  
 항목을 리스트에 추가 49  
 항목을 사전에 추가 17  
 해시 테이블 16, 53  
 해시 테이블 기반 데이터베이스 383  
 현재 시간, 얻기 499  
 현재 작업 디렉터리 변경 467  
 호스트 이름, 호스트 머신 얻기 584  
 호출 가능 객체 60  
 클래스 60  
 타입 57  
     \_\_call\_\_() 메서드 75  
 호환성, marshal 모듈 279  
 혼합 클래스 148  
 혼합 타입 산술 연산 79  
 화면에 출력 10  
 확률, 무작위 수 분포 313  
 확장 가능한 코드, 모듈 176  
 확장 대입 연산자 74, 90  
 확장 모듈 730  
     래퍼 함수 732  
     문서화 문자열 733~734  
     수작업 730~731  
     스레드 749  
     예외 처리 746  
     이름 733~734  
     전역 인터프리터 락 748~749  
     직접 컴파일 736~737  
     참조 횟수 세기 748~749  
     파이썬 3에서 차이점 734  
     파이썬에서 C로 타입 변환 737

ctypes 모듈 755  
 C에서 파이썬으로 타입 변환 743  
 distutils에서 컴파일 735  
 확장 분할 48, 71  
 삭제 48, 83  
 순서열 81~82  
 할당 48, 83  
 확장 분할 연산자 [:] 80  
 확장된 반복 가능한 객체 풀어헤치기, 파이  
 션 3 770  
 확장성, 병행성 510  
 환경 변수 193, 466  
 삭제 470  
 인터프리터에서 사용 214  
 파이썬 3 784  
 파일 이름 확장 488  
 CGI 스크립트 658  
 WSGI 666  
 회전 로그 파일 446  
 흰 수염 해커 769  
 힙 333

**A**

a(args) 디버거 명령, pdb 모듈 231  
 'a' 모드, open() 함수 193  
 a2b\_base64() 함수, binascii 모듈 676  
 a2b\_hex() 함수, binascii 모듈 676  
 a2b\_hqx() 함수, binascii 모듈 677  
 a2b\_uu() 함수, binascii 모듈 676  
 abc 모듈 167, 317  
 ABCMeta 메타클래스 167, 317  
 abort() 메서드, FTP 객체 612  
 abort() 함수, os 모듈 480  
 \_\_abs\_\_() 메서드 74  
 abs() 함수 78, 247  
     operator 모듈 338  
 abspath() 함수, os.path 모듈 488  
 \_\_abstractmethods\_\_ 속성, 타입 61  
 @abstractmethod 장식자 167, 318  
 @abstractproperty 장식자 167, 318  
 accept() 메서드

dispatcher 객체 561  
 Listener 객체 534  
 socket 객체 589  
 accept2dayear 변수, time 모듈 498  
 access() 함수, os 모듈 475  
 acos() 함수, math 모듈 309  
 acosh() 함수, math 모듈 309  
 acquire() 메서드  
     Condition 객체 544  
     Lock 객체 540  
     RLock 객체 541  
     Semaphore 객체 541  
 activate() 메서드, SocketServer 클래스  
     607  
 active\_children() 함수, multiprocessing 모  
     듈 535  
 active\_count() 함수, threading 모듈 547  
 ActivePython 3  
 add() 메서드  
     집합 16, 56  
     TarFile 객체 394  
 \_\_add\_\_() 메서드 73  
 add() 함수, operator 모듈 338  
 add\_data() 메서드, Request 객체 636  
 add\_header() 메서드  
     Message 객체 687  
     Request 객체 636  
 add\_option() 메서드, OptionParser 객체  
     192, 462  
 add\_password() 메서드, AuthHandler 객체  
     639  
 add\_section() 메서드, ConfigParser 객체  
     409  
 add\_type() 함수, mimetypes 모듈 701  
 add\_unredirected\_header() 메서드,  
     Request 객체 636  
 addfile() 메서드, TarFile 객체 394  
 addFilter() 메서드  
     Handler 객체 448  
     Logger 객체 441  
 addHandler() 메서드, Logger 객체 444  
 addLevelName() 함수, logging 모듈 451  
 address 속성  
     BaseManager 객체 531  
     Listener 객체 534  
 address\_family 속성, SocketServer 클래스  
     606  
 addressof() 함수, ctypes 모듈 761  
 adler32() 함수, zlib 모듈 404  
 AF\_\* 상수, socket 모듈 578  
 aifc 모듈 726  
 aio\_\* 시스템 호출 집합, 없음 577  
 AJAX, 예 656  
 alarm() 함수, signal 모듈 491  
 alias 디버거 명령, pdb 모듈 231  
 alignment() 함수, ctypes 모듈 761  
 all() 함수 48, 80, 247  
 \_\_all\_\_ 변수  
     패키지 184  
     import문 178  
 allow\_reuse\_address 속성, SocketServer  
     클래스 606  
 altsep 변수, os 모듈 475  
 altzone 변수, time 모듈 498  
 and\_() 함수, operator 모듈 338  
 \_\_and\_\_() 메서드 73  
 \_\_annotations\_\_ 속성, 함수 771  
 any() 함수 48, 80, 247  
 anydbm 모듈 382  
 api\_version 변수, sys 모듈 283  
 %APPDATA% 환경 변수, 윈도우 218  
 append() 메서드  
     리스트 12, 48  
     배열 객체 320  
     deque 객체 324  
     Element 객체 713  
 appendChild() 메서드, DOM Node 객체  
     706  
 appendleft() 메서드, deque 객체 324  
 apply() 메서드, Pool 객체 522  
 apply\_async() 메서드, Pool 객체 522  
 args 속성  
     예외 106  
 Exception 객체 263

partial 객체 330  
 argtypes 속성, ctypes 함수 객체 756  
 argv 변수, sys 모듈 13, 191, 214, 283  
 ArithmeticError 예외 104, 262  
 array 모듈 319  
 Array() 메서드, Manager 객체 527  
 array() 함수, array 모듈 320  
 Array() 함수, multiprocessing 모듈 525  
 arraysize 속성, Cursor 객체 368  
 as 한정어  
     except문 25, 102  
     from-import문 177  
     import문 27, 176  
     with문 76, 107  
 as\_integer\_ratio() 메서드, 부동 소수점 47  
 as\_string() 메서드, Message 객체 687  
 ascii 인코딩, 설명 206  
 ascii() 함수 247  
     파이썬 3 248  
     future\_builtins 모듈 268  
 ASCII, UTF-8과 호환 208  
 ascii\_letters 변수, string 모듈 353  
 ascii\_lowercase 변수, string 모듈 353  
 ascii\_uppercase 변수, string 모듈 353  
 asctime() 함수, time 모듈 499  
 asin() 함수, math 모듈 309  
 asinh() 함수, math 모듈 309  
 assert문 109  
 assert\_() 메서드, TestCase 객체 227  
 assertAlmostEqual() 메서드, TestCase 객체 228  
 assertEquals() 메서드, TestCase 객체 228  
 AssertionError 예외 105, 109, 263  
 assertNotAlmostEqual() 메서드, TestCase 객체 228  
 assertNotEqual() 메서드, TestCase 객체 228  
 assertRaises() 메서드, TestCase 객체 228  
 astimezone() 메서드, datetime 객체 419  
 asynchat 모듈 556  
     사용 574  
 asynchat 클래스, asynchat 모듈 556

asyncore 모듈 510, 560  
     사용 574  
 AsyncResult 객체, multiprocessing 모듈 523  
 atan() 함수, math 모듈 309  
 atan2() 함수, math 모듈 309  
 atanh() 함수, math 모듈 309  
 atexit 모듈 221, 269  
 attach() 메서드, Message 객체 687  
 attrgetter() 함수, operator 모듈 339  
 attrib 속성, Element 객체 713  
 AttributeError 예외 105, 263  
     속성 바인딩 161  
 attributes 속성, DOM Node 객체 705  
 audioop 모듈 726  
 authkey 속성, Process 객체 512  
 awk 유닉스 명령어, 리스트 내포와 유사점 134

**B**

-B 명령줄 옵션 213  
 b 문자, 문자열 상수 앞 34  
 b(reak) 디버거 명령, pdb 모듈 231  
 b16decode() 함수, base64 모듈 675  
 b16encode() 함수, base64 모듈 675  
 b2a\_base64() 함수, binascii 모듈 676  
 b2a\_hex() 함수, binascii 모듈 677  
 b2a\_hqx() 함수, binascii 모듈 677  
 b2a\_uu() 함수, binascii 모듈 676  
 b32decode() 함수, base64 모듈 675  
 b32encode() 함수, base64 모듈 675  
 b64decode() 함수, base64 모듈 674  
 b64encode() 함수, base64 모듈 674  
 'backslashreplace' 에러 처리, 유니코드 인코딩 203  
 BadStatusLine 예외, http.client 모듈 620  
 base64 모듈 673  
 base64 인코딩, 설명 673  
 BaseCGIHandler() 함수, wsgiref.handlers 모듈 669  
 BaseException 예외 104  
 BaseException 클래스 262  
 BaseHTTPRequestHandler 클래스, http.server 모듈 625  
 BaseHTTPserver 모듈, http.server 참고 622  
 BaseManager() 함수, multiprocessing 모듈 530  
 basename() 함수, os.path 모듈 488, 488  
 BaseProxy 클래스, multiprocessing 모듈 532  
 BaseRequestHandler 클래스, SocketServer 모듈 602  
 \_\_bases\_\_ 속성  
     클래스 160  
     타입 61  
 basestring 변수 248  
 basicConfig() 함수, logging 모듈 437  
 BasicContext 변수, decimal 모듈 305  
 .bat 파일, 윈도우 218  
 bdb 모듈 723  
 BeautifulSoup 패키지 696  
 betavariate() 함수, random 모듈 314  
 bidirectional() 함수, unicodedata 모듈 360  
 bin() 함수 92, 248  
 Binary() 함수  
     데이터베이스 API 371  
     xmlrpc.client 모듈 648  
 binascii 모듈 676  
 bind() 메서드  
     dispatcher 객체 562  
     socket 객체 589  
     SocketServer 클래스 607  
 binhex 모듈 725  
 bisect 모듈 322  
 bisect() 함수, bisect 모듈 322  
 bisect\_left() 함수, bisect 모듈 322  
 bisect\_right() 함수, bisect 모듈 323  
 block\_size 속성, digest 객체 691  
 BOM 344  
     유니코드 205  
 BOM\_\* 상수, codecs 모듈 345

bool 타입 45  
 bool() 함수 248  
`_bool_()` 메서드 68  
 boolean() 함수, xmlrpclib.client 모듈 648  
 BoundedSemaphore 객체  
   multiprocessing 모듈 526  
   threading 모듈 542  
 BoundedSemaphore() 메서드, Manager 객체 528  
 break문 100~101  
   생성기 124  
 BSD, kqueue 인터페이스 566  
 BTPROTO\_\* 상수, socket 모듈 588  
 buffer\_info() 메서드, 배열 객체 320  
 BufferedIOBase 추상 기반 클래스 435  
 BufferedRandom 클래스, io 모듈 433  
 BufferedReader 클래스, io 모듈 432  
 BufferedRWPair 클래스, io 모듈 433  
 BufferedWriter 클래스, io 모듈 432  
 build\_opener() 함수, urllib.request 모듈 638  
`_builtin_` 모듈 247  
 builtin\_module\_names 변수, sys 모듈 283  
 BuiltinfunctionType 60  
 BuiltinfunctionType 타입 56, 292  
 builtins 모듈, 파이썬 3 247  
 byref() 함수, ctypes 모듈 759  
 bytearray() 함수 248  
 byteorder 변수, sys 모듈 283  
 bytes 데이터 타입, 파이썬 3 35  
 bytes() 함수 249  
 BytesIO 클래스, io 모듈 433  
 byteswap() 메서드, 배열 객체 320  
 bz2 모듈 385  
 BZ2Compressor() 함수, bz2 모듈 385  
 BZ2Decompressor() 함수, bz2 모듈 386  
 BZ2File() 함수, bz2 모듈 385

**C**


---

C  
   파이썬 변수와 비교 6  
   함수 구현 60  
   -c 명령줄 옵션 213~214  
   C 프로그램에서 인터프리터 임베딩 729, 750  
 C 확장 기능 729  
   모듈 재로딩 182  
   전역 인터프리터 락 해제 548  
   ctypes 예 763  
   distutils에서 컴파일 735  
   .egg 파일 180  
   SWIG에서 생성 764  
 C# 765  
 c(ont(inue)) 디버거 명령, pdb 모듈 232  
 C/C++ 코드, 씨드파티 패키지 189~190  
 c\_\* 데이터 타입, ctypes 모듈 757~758  
 C++, 클래스 시스템에서 다른 점 143  
 C3 선형화 알고리즘, 상속 147  
 CacheFTPHandler 클래스, urllib.request 모듈 638  
 calcsize() 함수, struct 모듈 358  
 calendar 모듈 726  
 call() 함수, subprocess 모듈 496  
`_call_()` 메서드 60, 75  
 Callable 추상 기반 클래스 328  
`_callmethod()` 메서드, BaseProxy 객체 533  
 callproc() 메서드, Cursor 객체 367  
 cancel() 메서드, Timer 객체 540  
 cancel\_join\_thread() 메서드, Queue 객체 514  
 CannotSendHeader 예외, http.client 모듈 621  
 CannotSendRequest 예외, http.client 모듈 621  
 capitalize() 메서드, 문자열 50~51  
 capitals 속성, Context 객체 301  
 capwords() 함수, string 모듈 356  
 case문, 없음 8  
 cast() 함수, ctypes 모듈 761  
 category() 함수, unicodedata 모듈 209, 360  
`_cause_` 속성, Exception 객체 263, 774  
 CDLL() 함수, ctypes 모듈 755  
 ceil() 함수, math 모듈 309  
 center() 메서드, 문자열 50~51  
 cert\_time\_to\_seconds() 함수, ssl 모듈 600  
 cgi 모듈 658  
 CGI 스크립트 658  
   데이터베이스 사용 665  
   웹 프레임워크 665  
   작성 조언 663  
   환경 변수 658  
   WSGI 응용 프로그램 실행 669  
   XML-RPC 서버 실행 652~653  
 CGIHandler() 함수, wsgiref.handlers 모듈 669  
 CGIHTTPRequestHandler 클래스, http.server 모듈 624  
 CGIHTTPServer 모듈, http.server 참고 622  
 cgitb 모듈 665  
 CGIXMLRPCRequestHandler 클래스, xmlrpc.server 모듈 651  
 chain() 함수, itertools 모듈 334  
 characters() 메서드, ContentHandler 객체 719  
 chdir() 함수, os 모듈 467  
 check\_call() 함수, subprocess 모듈 496  
 check\_unused\_args() 메서드, Formatter 객체 355  
 chflags() 함수, os 모듈 475  
 chicken, 다중 스레드 509  
 childNodes 속성, DOM Node 객체 705  
 chmod() 함수, os 모듈 476  
 choice() 함수, random 모듈 313  
 chown() 함수, os 모듈 476  
 chr() 함수 93, 249  
 chroot() 함수, os 모듈 467  
 chunk 모듈 726  
 cipher() 메서드, ssl 객체 599

cl(ear) 디버거 명령, pdb 모듈 231  
 class문 24, 141  
   상속 24, 144  
   클래스 몸체 실행 169  
 \_\_class\_\_ 속성  
   메서드 60  
   인스턴스 60, 160  
 @classmethod 장식자 58, 151, 152, 249  
 ClassType 타입, 구형 클래스 170  
 cleandoc() 함수, inspect 모듈 273  
 clear() 메서드  
   사전 54  
   집합 56  
   deque 객체 324  
   Element 객체 713  
   Event 객체 543  
 clear\_flags() 메서드, Context 객체 303  
 clear\_memo() 메서드, Pickler 객체 281  
 \_clear\_type\_cache() 함수, sys 모듈 287  
 Client 클래스, multiprocessing 모듈 533  
 client\_address 속성  
   BaseHTTPRequestHandler 객체 626  
   BaseRequestHandler 객체 603  
 clock() 함수, time 모듈 236, 499  
 cloneNode() 메서드, DOM Node 객체 706  
 close() 메서드  
   동기화 124  
   생성기 20, 64, 124, 126  
   파일 194  
   Connection 객체 366, 518  
   Cursor 객체 367  
   dbm 스타일 데이터베이스 객체 382  
   dispatcher 객체 562  
   FTP 객체 612  
   Handler 객체 449  
   HTMLParser 객체 694  
   HTTPConnection 객체 618  
   IOBase 객체 429  
   Listener 객체 534  
   mmap 객체 457  
   Pool 객체 522

Queue 객체 514  
 shelve 객체 383  
 socket 객체 589  
 TarFile 객체 394  
 TreeBuilder 객체 714  
 urlopen 객체 635  
 ZipFile 객체 401  
 close() 함수, os 모듈 470  
 close\_when\_done() 메서드, asynchat 객체 557  
 closed 속성  
   파일 197  
 IOBase 객체 429  
 closefd 매개변수, open() 함수 194  
 closefd 속성, FileIO 객체 431  
 CloseKey() 함수, winreg 모듈 502  
 \_\_closure\_\_ 속성, 함수 58, 120  
 closerange() 함수, os 모듈 470  
 closing() 함수, contextlib 모듈 331  
 cmath 모듈 309  
 cmd 모듈 726  
 cmp() 함수 249  
   filecmp 모듈 387  
 cmpfiles() 함수, 파일cmp 모듈 387  
 co\_\* 속성, 코드 객체 63  
 code 모듈 724  
   \_\_code\_\_ 속성, 함수 58  
 CodecInfo 클래스, codecs 모듈 341  
 codecs 모듈 204, 341  
   바이트 문자열 사용 345  
   압축 코덱 제거 345  
 coded\_value 속성, Morsel 객체 631  
 codeop 모듈 724  
 CodeType 타입 62, 293  
 \_\_coerce\_\_() 메서드, 사용 안됨 164  
 collect 함수, gc 모듈 221  
 collect() 함수, gc 모듈 271  
 collect\_incoming\_data() 메서드, asynchat 객체 557  
 collections 모듈 169, 323  
 colorsys 모듈 726  
 combinations() 함수, itertools 모듈 334  
 combine() 메서드, datetime 클래스 417  
 combining() 함수, unicodedata 모듈 361  
 command 속성,  
   BaseHTTPRequestHandler 객체 626  
 commands 디버거 명령, pdb 모듈 231  
 commands 모듈 408  
 comment 속성, ZipInfo 객체 403  
 Comment() 함수, xml.etree.ElementTree 모듈 711  
 commit() 메서드, Connection 객체 366  
 common 속성, difflib 객체 388  
 common\_dirs 속성, difflib 객체 388  
 common\_files 속성, difflib 객체 388  
 common\_funny 속성, difflib 객체 388  
 commonprefix() 함수, os.path 모듈 488  
 communicate() 메서드, Popen 객체 496  
 compile() 함수 139, 250  
   re 모듈 348  
 compileall 모듈 724  
 complete\_statement() 함수, sqlite3 모듈 375  
 Complex 추상 기반 클래스 311  
 complex 타입 45  
 complex() 함수 92, 250  
 \_\_complex\_\_() 메서드 70, 74~75  
   강제 타입 변환 164  
 compress() 메서드  
   BZ2Compressor 객체 386  
   compressobj 객체 404  
 compress() 함수  
   bz2 모듈 386  
   zlib 모듈 404  
 compress\_size 속성, ZipInfo 객체 404  
 compress\_type 속성, ZipInfo 객체 403  
 CompressionError 메서드, tarfile 모듈 397  
 compressobj() 함수, zlib 모듈 404  
 concat() 함수, operator 모듈 338  
 Condition 객체  
   multiprocessing 모듈 526  
   threading 모듈 544  
 condition 디버거 명령, pdb 모듈 232

Condition() 메서드, Manager 객체 528  
 configparser 모듈 408  
 ConfigParser 클래스, configparser 모듈 409  
 confstr() 함수, os 모듈 486  
 conjugate() 메서드  
 복소수 47  
 부동 소수점 47  
 connect() 메서드  
 BaseManager 객체 531  
 dispatcher 객체 562  
 FTP 객체 612  
 HTTPConnection 객체 618  
 SMTP 객체 633  
 socket 객체 589  
 connect() 함수  
 데이터베이스 API 366  
 sqlite3 모듈 374  
 connect\_ex() 메서드, socket 객체 589  
 connecting processes, multiprocessing 모듈 533  
 Connection 클래스  
 데이터베이스 API 366  
 sqlite3 모듈 375  
 ConnectRegistry() 함수, winreg 모듈 502  
 contains() 함수, operator 모듈 338  
 ContentHandler 클래스, xml.sax 모듈 719  
 ContentTooShort 메서드, urllib.error 모듈 645  
 Context 클래스, decimal 모듈 301  
 \_\_context\_\_ 속성, Exception 객체 263, 775  
 contextlib 모듈 109, 330  
 @contextmanager 장식자 109  
 continue문 100~101  
 convert\_field() 메서드, Formatter 객체 355  
 Cookie 모듈, http.cookies 참고 628  
 CookieError 예외, http.cookies 모듈 631  
 CookieJar 클래스, http.cookiejar 모듈 631  
 cookielib 모듈, http.cookiejar 참고 631  
 copy 모듈 43, 81, 269  
 한계 270  
 copy() 메서드  
 사전 54  
 집합 56  
 Context 객체 304  
 digest 객체 691  
 hmac 객체 692  
 copy() 함수  
 copy 모듈 270  
 shutil 모듈 391  
 \_\_copy\_\_() 메서드 270  
 copy\_reg 모듈 724  
 copy2() 함수, shutil 모듈 391  
 copyfile() 함수, shutil 모듈 392  
 copyfileobj() 함수, shutil 모듈 392  
 copymode() 함수, shutil 모듈 392  
 copyright 변수, sys 모듈 283  
 copysign() 함수, math 모듈 309  
 copystat() 함수, shutil 모듈 392  
 copytree() 함수, shutil 모듈 392  
 @coroutine 장식자 예 126  
 cos() 함수, math 모듈 309  
 cosh() 함수, math 모듈 309  
 count() 메서드  
 리스트 48  
 문자열 51  
 배열 객체 320  
 count() 함수, itertools 모듈 335  
 countOf() 함수, operator 모듈 338  
 cp1252 인코딩, 설명 207  
 cp437 인코딩, 설명 207  
 cPickle 모듈 282  
 cProfile 모듈 234  
 CPU 시간, 얻기 236, 499  
 CPU 위주 작업과 스레드 548  
 CPU, 시스템 CPU 개수 획득 535  
 cpu\_count() 함수, multiprocessing 모듈 535  
 CRC 속성, ZipInfo 객체 404  
 crc\_hqx() 함수, binascii 모듈 677  
 crc32() 함수  
 binascii 모듈 677  
 zlib 모듈 404  
 create\_aggregate() 메서드, Connection 객체 376  
 create\_collation() 메서드, Connection 객체 376  
 create\_connection() 함수, socket 모듈 582  
 create\_decimal() 메서드, Context 객체 304  
 create\_function() 메서드, Connection 객체 376  
 create\_socket() 메서드, dispatcher 객체 562  
 create\_string\_buffer() 함수, ctypes 모듈 761  
 create\_system 속성, ZipInfo 객체 403  
 create\_unicode\_buffer() 함수, ctypes 모듈 761  
 create\_version 속성, ZipInfo 객체 403  
 created 속성, Record 객체 442  
 CreateKey() 함수, winreg 모듈 502  
 critical() 메서드, Logger 객체 439  
 crypt 모듈 725  
 CSV 데이터 읽기, 예 15  
 CSV 데이터, 읽기 예 15  
 csv 모듈 677  
 CSV 파일  
 열 태입 변환 45  
 파싱 677~678  
 ctermid() 함수, os 모듈 467  
 ctime() 메서드, date 객체 414  
 ctime() 함수, time 모듈 499  
 Ctrl-C, 키보드 인터럽트 198  
 ctypes 모듈 755  
 가능한 데이터 타입 757~758  
 공유 라이브러리 로드 755  
 구조체 타입 758  
 디중 처리와 공유 메모리 525  
 데이터 타입 변환 761  
 라이브러리 모듈 찾기 755

메모리 복사 762  
 바이트 문자열 생성 761  
 배열 타입 758  
 베파로부터 객체 생성 760  
 예 763  
 포인터 타입 758  
 포인터와 참조 전달 758  
 함수 프로토 타입 설정 756  
 cunifvariate() 함수, random 모듈 314  
 curdir 변수, os 모듈 475  
 current\_process() 함수, multiprocessing 모듈 535  
 current\_thread() 함수, threading 모듈 547  
 currentframe() 함수, inspect 모듈 273  
 \_current\_frames() 함수, sys 모듈 287  
 curses 모듈 725  
 Cursor 클래스, 데이터베이스 API 366  
 cursor() 메서드, Connection 객체 366  
 cwd() 메서드, FTP 객체 612  
 cycle() 함수, itertools 모듈 335  
 C로 확장 729  
 C에서 파이썬 함수 호출 752  
 C에서 파이썬으로 타입 변환 737

**D**

d(own) 디버거 명령, pdb 모듈 232  
 daemon 속성  
 Process 객체 512  
 Thread 객체 538  
 data 속성, DOM Text 객체 708  
 data() 메서드, TreeBuilder 객체 715  
 DatabaseError 예외, 데이터베이스 API 372  
 DatagramHandler 클래스, logging 모듈 445  
 DatagramRequestHandler 클래스,  
 SocketServer 모듈 603  
 date 클래스, datetime 모듈 414  
 date() 메서드, datetime 객체 419  
 Date() 함수, 데이터베이스 API 371

date\_time 속성, ZipInfo 객체 403  
 DateFromTicks() 함수, 데이터베이스 API 371  
 datetime 모듈 413  
 datetime 클래스, datetime 모듈 417  
 DateTime() 함수, xmlrpclib.client 모듈 648  
 day 속성, date 객체 415  
 daylight 변수, time 모듈 498  
 dbhash 모듈 381  
 dbm 모듈 381  
 DBM 스타일 데이터베이스 382  
 debug 속성  
 sys.flags 284  
 TarFile 객체 394  
 ZipFile 객체 401  
 debug() 메서드, Logger 객체 439  
 \_debug\_ 변수 110, 454  
 Decimal 객체, 유리수로 변환 308  
 decimal 모듈 47, 299  
 반올림 방식 301  
 슬래드 307  
 sum() 함수 82  
 Decimal 클래스, decimal 모듈 299  
 decimal() 함수, unicodedata 모듈 362  
 decode() 메서드  
 문자열 35, 51, 202~203  
 적절한 사용 203  
 파이썬 3 778  
 CodecInfo 객체 342  
 IncrementalDecoder 객체 343  
 JSONDecoder 객체 699  
 decode() 함수  
 base64 모듈 675  
 quopri 모듈 702  
 decodestring() 함수  
 base64 모듈 676  
 quopri 모듈 702  
 decomposition() 함수, unicodedata 모듈 362  
 decompress() 메서드  
 BZ2Decompressor 객체 387  
 decompressobj 객체 405  
 decompress() 함수  
 bz2 모듈 386  
 zlib 모듈 405  
 decompressobj() 함수, zlib 모듈 405  
 deepcopy() 함수, copy 모듈 270  
 \_\_deepcopy\_\_() 메서드 270  
 def문 19, 57, 111  
 default() 메서드, JSONEncoder 객체 699  
 default\_factory 속성, defaultdict 객체 325  
 DefaultContext 변수, decimal 모듈 305  
 defaultdict() 함수, collections 모듈 325  
 defaults() 메서드, ConfigParser 객체 409  
 \_\_defaults\_\_ 속성, 함수 58  
 defects 속성, Message 객체 685  
 degrees() 함수, math 모듈 309  
 del문 42, 83  
 매팽 항목 제거 54  
 분할 48  
 \_\_del\_\_() 메서드 158  
 del 연산자, 사전 17, 89  
 \_\_del\_\_() 메서드 66, 158  
 사용 66  
 쓰레기 수집 158, 271  
 정의 위험 158  
 프로그램 종료 221  
 del\_param() 메서드, Message 객체 687  
 delattr() 함수 250  
 개인 속성 156  
 \_\_delattr\_\_() 메서드 69, 160  
 delete() 메서드, FTP 객체 612  
 \_\_delete\_\_() 메서드, 기술자 70, 154  
 @deleter 장식자 프로퍼티의 153  
 DeleteKey() 함수, winreg 모듈 503  
 DeleteValue() 함수, winreg 모듈 503  
 delitem() 함수, operator 모듈 338  
 \_\_delitem\_\_() 메서드 70  
 분할 70  
 delslice() 함수, operator 모듈 339  
 demo\_app() 함수, wsgiref.simple\_server 모듈 669  
 denominator 속성  
 정수 47

Fraction 객체 308  
 DeprecationWarning 경고 267, 294  
 deque 객체  
   리스트와 비교 240  
   collections 모듈 240  
 deque 객체로 원형 큐나 큐 323  
 deque() 함수, collections 모듈 323  
 DER\_cert\_to\_PEM\_cert() 함수, ssl 모듈 600  
 dereference 속성, TarFile 객체 394  
 description 속성, Cursor 객체 368  
 devnull 변수, os 모듈 475  
 Dialect 클래스, csv 모듈 680  
 dict 타입 45  
 dict() 메서드, Manager 객체 528  
 dict() 함수 17, 92, 250  
   성능 특성 241  
 \_\_dict\_\_ 속성  
   모듈 62, 177  
 사용자 정의 객체 76  
 인스턴스 61, 160  
 클래스 160  
 타입 61  
   함수 58, 138  
 \_\_dir\_\_() 메서드 76, 156, 251  
 DictReader() 함수, csv 모듈 679  
 dicts, 데이터 저장을 클래스와 비교 241  
 DictWriter() 함수, csv 모듈 679  
 diff\_files 속성, dircmp 객체 388  
 difference() 메서드, 집합 56  
 difference\_update() 메서드, 집합 56  
 difflib 모듈 724  
 dig 속성, sys.float\_info 284  
 digest() 메서드  
   digest 객체 691  
   hmac 객체 692  
 digest\_size 속성, digest 객체 691  
 digit() 함수, unicodedata 모듈 363  
 digits 변수, string 모듈 353  
 dir() 메서드, FTP 객체 612  
 dir() 함수 23, 27, 251  
   클래스에서 속성 이름 감추기 156, 251

dir() 함수에서 속성 이름 감추기 156  
 dircmp() 함수, 파일cmp 모듈 387  
 dirname() 함수, os.path 모듈 488  
 dis 모듈 724  
 dis(), dis 모듈 238  
 disable 디버거 명령, pdb 모듈 232  
 disable() 함수  
   gc 모듈 271  
   logging 모듈 451  
 disable\_interspersed\_args() 메서드,  
   OptionParser 객체 463  
 discard() 메서드, 집합 56  
 discard\_buffers() 메서드, asynchat 객체 557  
 dispatcher 클래스, asyncore 모듈 560  
 displayhook() 함수, sys 모듈 217, 287  
 \_\_displayhook\_\_ 변수, sys 모듈 283  
 disposition 속성, FieldStorage 객체 660  
 disposition\_options 속성, FieldStorage 객체 660  
 distutils 모듈 186~187, 724, 735  
   이진 배포판 생성 188~189  
   확장 모듈 735  
   SWIG에서 확장 기능 생성 765  
 div() 함수, operator 모듈 338  
 \_\_div\_\_() 메서드 73  
 division\_new 속성, sys.flags 284  
 division\_warning 속성, sys.flags 284  
 divmod() 함수 78, 251  
 \_\_divmod\_\_() 메서드 73  
 DLL  
   확장 모듈 181  
   ctypes와 함께 로드 755  
   distutils와 같이 생성 735  
 dllhandle 변수, sys 모듈 283  
 do\_handshake() 메서드, ssl 객체 599  
 \_\_doc\_\_ 속성  
   객체 36  
   내장 함수 60  
   메서드 60  
   모듈 62  
   타입 61

함수 28, 58, 136  
 DocCGIXMLRPCRequestHandler 클래스,  
   xmlrpc.server 모듈 652  
 doctest 모듈 223~224  
   verbose 옵션 225  
 Document 클래스, xml.dom.minidom 모듈 707  
 documentElement 속성, DOM Document 객체 707  
 DocXMLRPCServer 모듈 650  
 DocXMLRPCServer 클래스, xmlrpc.server 모듈 650  
 DOM 인터페이스  
   예 709  
   XML 파싱 703  
 dont\_write\_bytecode 변수, sys 모듈 283  
 dont\_write\_bytecode 속성, sys.flags 284  
 dropwhile() 함수, itertools 모듈 335  
 dst() 메서드  
   time 객체 416  
   tzinfo 객체 421  
 dumbdbm 모듈 381  
 dump() 메서드, Pickler 객체 281  
 dump() 함수  
   json 모듈 697  
   marshal 모듈 278  
   pickle 모듈 210, 279  
   xml.etree.ElementTree 모듈 715  
 dumps() 함수  
   json 모듈 697  
   marshal 모듈 278  
   pickle 모듈 280  
   xmlrpc.client 모듈 648  
 dup() 함수, os 모듈 470  
 dup2() 함수, os 모듈 470  
 dynamic loading, 모듈 176

**E**

-E 명령줄 옵션 213  
 e 변수, math 모듈 309  
 EAI\_\* 상수, socket 모듈 597

east\_asian\_width() 함수, unicodedata 모듈 363  
**easy\_install** 명령어, setuptools 패키지 190  
 .egg 파일 189  
 구조 180  
 모듈 180  
 사이트 구성 218  
**Element** 클래스, `xml.dom.minidom` 모듈 707  
**Element()** 함수, `xml.etree.ElementTree` 모듈 712  
**ElementTree** 인터페이스, XML 파싱 703~704  
**ElementTree** 클래스, `xml.etree.ElementTree` 모듈 710  
**ElementTree**, 예 716  
**elif**문 8, 98  
**Ellipsis** 36, 62, 65  
 색인 메서드에서 사용 65  
 타입 62  
 파일 3에서 표현식 774  
 확장 분할에서 사용 71  
**else**문 8, 98  
**else**절  
 try문 103  
 while과 for 루프 101  
**email** 패키지 682  
**Emax** 속성, Context 객체 303  
**Emin** 속성, Context 객체 304  
**Empty** 예외, Queue 모듈 514, 549  
**empty()** 메서드, Queue 객체 514, 549  
**enable** 디버거 명령, pdb 모듈 232  
**enable()** 함수  
 cgib 모듈 665  
 gc 모듈 271  
**enable\_callback\_tracebacks()** 함수, `sqlite3` 모듈 375  
**enable\_interspersed\_args()** 메서드  
 OptionParser 객체 463  
**encode()** 메서드  
 문자열 51, 202~203  
 적절한 사용 203  
 파일 3 778  
**CodecInfo** 객체 342  
**IncrementalEncoder** 객체 342  
**JSONEncoder** 객체 700  
**encode()** 함수  
 base64 모듈 676  
 quopri 모듈 702  
**EncodedFile** 객체, codecs 모듈 205  
**EncodedFile** 클래스, codecs 모듈 344  
**encodestring()** 함수  
 base64 모듈 676  
 quopri 모듈 702  
**encoding** 속성  
 파일 197  
 TextIOWrapper 객체 434  
**encoding** 인수 `open()` 함수 193  
**end** 속성, 분할 65  
**end** 키워드 인수, `print()` 함수 199  
**end()** 메서드  
 MatchObject 객체 351  
 TreeBuilder 객체 715  
**end\_headers()** 메서드,  
 BaseHTTPRequestHandler 객체 627  
**endDocument()** 메서드, ContentHandler 객체 719  
**endElement()** 메서드, ContentHandler 객체 719  
**endElementNS()** 메서드, ContentHandler 객체 719  
**endheaders()** 메서드, HTTPConnection 객체 619  
**endpos** 속성, MatchObject 객체 351  
**endPrefixMapping()** 메서드,  
 ContentHandler 객체 719  
**endswith()** 메서드, 문자열 52  
**\_enter\_()** 메서드, 컨텍스트 관리자 76, 108  
**enumerate()** 함수 99, 251  
 threading 모듈 547  
**EnumKey()** 함수, winreg 모듈 503  
**EnumValue()** 함수, winreg 모듈 503  
**environ** 변수, os 모듈 193, 466  
**EnvironmentError** 예외 105, 262  
**EOF** 문자, 대화식 모드 5  
**EOF** 표시, 파일 I/O 195  
**EOFError** 예외 105, 263  
**epilogue** 속성, Message 객체 685  
**epoll** 인터페이스, 리눅스 566  
**epsilon** 속성, `sys.float_info` 284  
**eq()** 함수, operator 모듈 338  
**\_eq\_()** 메서드 68  
**errcheck** 속성, ctypes 함수 객체 757  
**errno** 모듈 423  
**error** 예외 487  
 socket 모듈 596  
**error()** 메서드, Logger 객체 439  
**error\_message\_format** 속성,  
 BaseHTTPRequestHandler 클래스 626  
**errorcode** 변수, errno 모듈 423  
**errorlevel** 속성, TarFile 객체 395  
**errors** 매개변수  
 인코딩 함수 203  
**open()** 함수 194  
**errors** 속성, TextIOWrapper 객체 434  
**escape()** 함수  
 cgi 모듈 661  
 re 모듈 348  
 xml.sax.saxutils 모듈 722  
**eval()** 함수 67, 92, 139, 251  
**repr()** 67  
**Event** 객체  
 multiprocessing 모듈 526  
 threading 모듈 542  
**Event()** 메서드, Manager 객체 528  
**EX\_\*** 종료 코드 상수 481  
**exc\_clear()** 함수, sys 모듈 288  
**exc\_info** 속성, Record 객체 442  
**exc\_info()** 함수, sys 모듈 64, 107, 288  
**except**문 25, 102  
 문법 변경 102  
**excepthook()** 함수, sys 모듈 102, 288  
**\_excepthook\_** 변수, sys 모듈 283  
**Exception** 예외 104

Exception 클래스 262  
**exception()** 메서드, Logger 객체 440  
**.exe** 파일, distutils로 생성 189  
**exec문**, 레거시 코드와 사용할 때 주의점 138  
**exec()** 함수 138, 251  
 파일 3 781  
**exec\_prefix** 변수, sys 모듈 218, 283  
**exec()** 함수, os 모듈 480  
**execle()** 함수, os 모듈 481  
**execclp()** 함수, os 모듈 481  
**executable** 변수, sys 모듈 284  
**execute()** 메서드  
 Connection 객체 377  
 Cursor 객체 367  
**executemaney()** 메서드  
 Connection 객체 377  
 Cursor 객체 367  
**executescript()** 메서드, Connection 객체 377  
**execv()** 함수, os 모듈 481  
**execve()** 함수, os 모듈 481  
**execvp()** 함수, os 모듈 481  
**execvpe()** 함수, os 모듈 481  
**exists()** 함수, os.path 모듈 488  
**exit()** 함수, sys 모듈 221, 288  
**\_exit()** 함수, os 모듈 221, 481  
**\_exit\_()** 메서드 76  
 컨텍스트 관리자 76, 108  
**exitcode** 속성, Process 객체 512  
**exp()** 메서드, Decimal 객체 301  
**exp()** 함수, math 모듈 309  
**expand()** 메서드, MatchObject 객체 350  
**ExpandEnvironmentStrings()** 함수, winreg 모듈 503  
**expandtabs()** 메서드, 문자열 50~52  
**expanduser()** 함수, os.path 모듈 488  
**expandvars()** 함수, os.path 모듈 488  
**expovariate()** 함수, random 모듈 314  
**extend()** 메서드  
 리스트 49  
 배열 객체 320

**deque** 객체 324  
**ExtendedContext**, decimal 모듈 305  
**extendleft()** 메서드, deque 객체 324  
**Extension()** 함수, distutils 모듈 736  
**extensions\_map** 속성,  
 HTTPRequestHandler 클래스 624  
**external\_attr** 속성, ZipInfo 객체 404  
**extra** 속성, ZipInfo 객체 403  
**extract()** 메서드  
 TarFile 객체 395  
 ZipFile 객체 401  
**extract\_stack()** 함수, traceback 모듈 291  
**extract\_tb()** 함수, traceback 모듈 291  
**extract\_version** 속성, ZipInfo 객체 403  
**extractall()** 메서드, ZipFile 객체 401  
**ExtractError** 예외, tarfile 모듈 397  
**extractfile()** 메서드, TarFile 객체 395  
**extsep** 변수, os 모듈 475

**F**


---

**F\_\*** 상수, fcntl() 함수 427  
**f\_\*** 속성  
 프레임 객체 63  
 statvfs 객체 479  
**fabs()** 함수, math 모듈 309  
**factorial()** 함수, math 모듈 309  
**fail()** 메서드, TestCase 객체 228  
**failIf()** 메서드, TestCase 객체 228  
**failIfAlmostEqual()** 메서드, TestCase 객체 228  
**failIfEqual()** 메서드, TestCase 객체 228  
**failUnless()** 메서드, TestCase 객체 227  
**failUnlessAlmostEqual()** 메서드, TestCase 객체 228  
**failUnlessEqual()** 메서드, TestCase 객체 228  
**failUnlessRaises()** 메서드, TestCase 객체 228  
**failureException** 속성, TestCase 객체 228  
**False** 값 9, 31, 46  
**family** 속성, socket 객체 596  
**Fault** 예외, xmlrpclib.client 모듈 649  
**fchdir()** 함수, os 모듈 467  
**fchmod()** 함수, os 모듈 470  
**fchown()** 함수, os 모듈 470  
**fcntl** 모듈 427  
**fcntl()** 함수, fcntl 모듈 427  
**fdatasync()** 함수, os 모듈 470  
**fdopen()** 함수, os 모듈 471  
**feed()** 메서드, HTMLParser 객체 694  
**fetchall()** 메서드, Cursor 객체 367  
**fetchmany()** 메서드, Cursor 객체 367  
**fetchone()** 메서드, Cursor 객체 367  
**FieldStorage()** 함수, cgi 모듈 659  
**file** 속성, FieldStorage 객체 660  
**file** 키워드 인수, print() 함수 10, 199  
**\_file\_** 속성, 모듈 62  
**file\_offset** 속성, ZipInfo 객체 404  
**file\_size** 속성, ZipInfo 객체 404  
**filecmp** 모듈 387  
**fileConfig()** 함수, logging 모듈 453  
**FileCookieJar** 클래스, http.cookiejar 모듈 632  
**FileHandler** 클래스  
 logging 모듈 445  
 urllib.request 모듈 638  
**FileIO** 클래스, io 모듈 430  
**filename** 속성  
 FieldStorage 객체 660  
 Record 객체 442  
 ZipInfo 객체 403  
**fileno()** 메서드  
 파일 195~196  
 파일과 소켓 565  
 Connection 객체 518  
 IOBase 객체 429  
 socket 객체 589  
 SocketServer 객체 605  
 urlopen 객체 635  
**Filter** 클래스, logging 모듈 442  
**filter()** 함수 252  
 최적화 244  
 파일 3 252

fnmatch 모듈 389  
future\_builtins 모듈 268  
filterwarnings() 함수, warnings 모듈 295  
finally문 104  
    락 546  
find() 메서드  
    문자열 50~52  
    Element 객체 713  
    ElementTree 객체 710  
    mmap 객체 457  
find\_library() 함수, ctypes 모듈 756  
findall() 메서드  
    Element 객체 713  
    ElementTree 객체 711  
    Regex 객체 350  
findall() 함수, re 모듈 348  
findCaller() 메서드, Logger 객체 440  
finding all loaded 모듈 177  
finditer() 메서드, Regex 객체 350  
finditer() 함수, re 모듈 348  
findtext() 메서드  
    Element 객체 713  
    ElementTree 객체 711  
finish() 메서드, BaseRequestHandler 객체 602  
firstChild 속성, DOM Node 객체 705  
flag\_bits 속성, ZipInfo 객체 403  
flags 변수, sys 모듈 284  
flags 속성  
    Context 객체 304  
    Regex 객체 349  
flaming death, 확장 모듈 734  
float 타입 45  
float() 함수 13, 74, 92, 252  
\_\_float\_\_() 메서드 73  
    강제 타입 변환 164  
float\_info 변수, sys 모듈 284  
FloatingPointError 예외 104, 263~264  
flock() 함수, fcntl 모듈 428  
floor() 함수, math 모듈 309  
floordiv() 함수, operator 모듈 338  
\_\_floordiv\_\_() 메서드 73

flush() 메서드  
    파일 195  
BufferWriter 객체 432  
BZ2Compressor 객체 386  
compressobj 객체 405  
decompressobj 객체 405  
Handler 객체 449  
IOBase 객체 430  
mmap 객체 457  
FlushKey() 함수, winreg 모듈 504  
fma() 메서드, Decimal 객체 301  
fmod() 함수, math 모듈 309  
fnmatch 모듈 389  
fnmatch() 함수, fnmatch 모듈 389  
fnmatchcase() 함수, fnmatch 모듈 389  
for문 9, 18, 72, 82, 98  
    생성기 21  
    튜플 풀어헤치기 16  
    파일 9, 196  
fork() 함수, os 모듈 482  
ForkingMixIn 클래스, SocketServer 모듈 607  
ForkingTCPServer 클래스, SocketServer 모듈 608  
ForkingUDPServer 클래스, SocketServer 모듈 608  
forkpty() 함수, os 모듈 482  
format 속성, Struct 객체 358  
format() 메서드  
    문자열 7, 51, 67, 86~87, 198  
    문자열과 변수 보간 199  
    포맷 지정 코드 86  
    Formatter 객체 354  
format() 함수 7, 11~12, 67, 92, 252  
\_\_format\_\_() 메서드 67, 89  
format\_exc() 함수, traceback 모듈 291  
format\_exception() 함수, traceback 모듈 292  
format\_exception\_only() 함수, traceback 모듈 292  
format\_list() 함수, traceback 모듈 292  
format\_stack() 함수, traceback 모듈 292  
format\_tb() 함수, traceback 모듈 292  
format\_value() 메서드, Formatter 객체 355  
formatargspec() 함수, inspect 모듈 273  
formatargvalues() 함수, inspect 모듈 273  
formatter 모듈 725  
Formatter 클래스  
    logging 모듈 449  
    string 모듈 354  
formatwarning() 함수, warnings 모듈 295  
Fortran 공통 블럭, lack of 179  
found\_terminator() 메서드, asynchat 객체 557  
fpathconf() 함수, os 모듈 471  
fpectl 모듈 724  
fpformat 모듈 724  
Fraction 클래스, fractions 모듈 307  
fractions 모듈 47, 307  
fragment 속성  
    urlparse 객체 641  
    urllib.parse 객체 642  
FrameType 타입 62, 292  
freeze\_support() 함수, multiprocessing 모듈 535  
frexp() 함수, math 모듈 309  
from \_\_future\_\_ import 220  
from 모듈 import \* 27, 178  
    밑줄 포함 식별자 31  
    전역 변수 178~179  
    \_\_all\_\_ 변수 178  
from문  
    모듈 임포트 177  
    import문 27  
from\_address() 메서드, ctypes 타입 객체 760  
from\_buffer() 메서드, ctypes 타입 객체 760  
from\_buffer\_copy() 메서드, ctypes 타입 객체 760  
from\_decimal() 메서드, Fraction 클래스 308  
from\_float() 메서드, Fraction 클래스 308

from\_iterable() 메서드, 객체 334  
 from\_param() 메서드, ctypes 타입 객체 760  
 fromfd() 함수, socket 모듈 583  
 fromfile() 메서드, 배열 객체 321  
 fromhex() 메서드, 부동 소수점 47  
 fromkeys() 메서드, 사전 54  
 fromlist() 메서드, 배열 객체 321  
 fromordinal() 메서드  
     date 클래스 414  
     datetime 클래스 418  
 fromstring() 메서드, 배열 객체 321  
 fromstring() 함수, xml.etree.ElementTree 모듈 712  
 fromtimestamp() 메서드  
     date 클래스 414  
     datetime 클래스 418  
 fromutc() 메서드, tzinfo 객체 422  
 frozenset 타입 45, 55, 90  
 frozenset() 함수 93, 252  
 fstat() 함수, os 모듈 472  
 fstatvfs() 함수, os 모듈 472  
 fsum() 함수, math 모듈 309  
 fsync() 함수, os 모듈 472  
 FTP 서버, 파일 업로드 614  
 FTP() 함수, ftplib 모듈 611  
 FTPHandler 클래스, urllib.request 모듈 638  
 ftplib 모듈 611  
 ftruncate() 함수, os 모듈 472  
 Full 예외, Queue 모듈 514, 549  
 full() 메서드, Queue 객체 514, 549  
 func 속성, partial 객체 331  
 \_\_func\_\_ 속성, 메서드 60  
 funcName 속성, Record 객체 442  
 function call 연산자 () 57  
 functionType 타입 57, 293  
 functools 모듈 92, 137, 331  
 funny\_files 속성, dircmp 객체 388  
 \_\_future\_\_ 모듈 220  
 나누기 75  
 특징 목록 220

future\_builtins 모듈 268  
 FutureWarning 경고 267, 294

---

**G**

gaienter 예외, socket 모듈 596  
 gammavariate() 함수, random 모듈 314  
 garbage 변수, gc 모듈 271  
 gauss() 함수, random 모듈 314  
 gc 모듈 42, 221, 271  
 gcd() 함수, fractions 모듈 309  
 gdbm 모듈 381  
 ge() 함수, operator 모듈 338  
 \_\_ge\_\_() 메서드 68  
 GeneratorExit 예외 104, 124~125, 264  
 GeneratorType 타입 62, 292  
 get() 메서드  
     사전 17, 54  
     AsyncResult 객체 523  
     ConfigParser 객체 409  
     Element 객체 713  
     Message 객체 682  
     Queue 객체 514, 550  
 get() 함수, webbrowser 모듈 671  
 \_\_get\_\_() 메서드, 기술자 70, 154  
 get\_all() 메서드, Message 객체 683  
 get\_boundary() 메서드, Message 객체 683  
 get\_charset() 메서드, Message 객체 683  
 getCharsets() 메서드, Message 객체 683  
 get\_content\_charset() 메서드, Message 객체 683  
 get\_content\_maintype() 메서드, Message 객체 683  
 get\_content\_subtype() 메서드, Message 객체 683  
 get\_content\_type() 메서드, Message 객체 683  
 get\_count() 함수, gc 모듈 271  
 get\_data() 메서드, Request 객체 636  
 get\_debug() 함수, gc 모듈 271  
 get\_default\_type() 메서드, Message 객체 684  
 get\_dialect() 함수, csv 모듈 681  
 get\_errno() 함수, ctypes 모듈 761  
 get\_field() 메서드, Formatter 객체 355  
 get\_filename() 메서드, Message 객체 684  
 get\_full\_url() 메서드, Request 객체 637  
 get\_host() 메서드, Request 객체 637  
 get\_last\_error() 함수, ctypes 모듈 761  
 get\_logger() 함수, multiprocessing 모듈 535  
 get\_method() 메서드, Request 객체 637  
 get\_nowait() 메서드, Queue 객체 515, 550  
 get\_objects() 함수, gc 모듈 271  
 get\_origin\_req\_host() 메서드, Request 객체 637  
 get\_osfhandle() 함수, msvcrt 모듈 459  
 get\_param() 메서드, Message 객체 684  
 get\_params() 메서드, Message 객체 684  
 get\_payload() 메서드, Message 객체 684  
 get\_referents() 함수, gc 모듈 272  
 get\_referrers() 함수, gc 모듈 272  
 get\_selector() 메서드, Request 객체 637  
 get\_server\_certificate() 함수, ssl 모듈 600  
 get\_starttag\_text() 메서드, HTMLParser 객체 694  
 get\_terminator() 메서드, asynchat 객체 557  
 get\_threshold() 함수, gc 모듈 272  
 get\_type() 메서드, Request 객체 637  
 get\_unixfrom() 메서드, Message 객체 685  
 get\_value() 메서드, Formatter 객체 355  
 getaddrinfo() 함수, socket 모듈 583  
 getargspec() 함수, inspect 모듈 273  
 getargvalues() 함수, inspect 모듈 273  
 getatime() 함수, os.path 모듈 488  
 getattr() 함수 253  
 개인 속성 156  
 \_\_getattr\_\_() 메서드 69  
     \_\_slots\_\_ 162  
 getAttribute() 메서드, DOM Element 객체 707

`__getattribute__()` 메서드 69, 161  
`_slots__` 162  
`getAttributeNS()` 메서드, DOM Element 객체 707  
`getboolean()` 메서드, ConfigParser 객체 410  
`getch()` 함수, msvcrt 모듈 459  
`getche()` 함수, msvcrt 모듈 459  
`getcheckinterval()` 함수, sys 모듈 288  
`getchildren()` 메서드, Element 객체 714  
`getclasstree()` 함수, inspect 모듈 274  
`getcode()` 메서드, urlopen 객체 635  
`getcomments()` 함수, inspect 모듈 274  
`getcontext()` 함수, decimal 모듈 304  
`getctime()` 함수, os.path 모듈 489  
`getcwd()` 함수, os 모듈 467  
`getcwdu()` 함수, os 모듈 467  
`getdefaultencoding()` 함수, sys 모듈 203, 288  
`getdefaulttimeout()` 함수, socket 모듈 584  
`getdlopenflags()` 함수, sys 모듈 288  
`getdoc()` 함수, inspect 모듈 274  
`getEffectiveLevel()` 메서드, Logger 객체 443  
`getegid()` 함수, os 모듈 467  
`getElementsByTagName()` 메서드  
  DOM Document 객체 707  
  DOM Element 객체 708  
`getElementsByTagNameNS()` 메서드  
  DOM Document 객체 707  
  DOM Element 객체 708  
`geteuid()` 함수, os 모듈 467  
`getfile()` 함수, inspect 모듈 274  
`getfilesystemencoding()` 함수, sys 모듈 288  
`getfirst()` 메서드, FieldStorage 객체 660  
`getfloat()` 메서드, ConfigParser 객체 410  
`getfqdn()` 함수, socket 모듈 584  
`_getframe()` 함수, sys 모듈 289  
`getframeinfo()` 함수, inspect 모듈 274  
`getgid()` 함수, os 모듈 467  
`getgroups()` 함수, os 모듈 467

`getheader()` 메서드, HTTPResponse 객체 619  
`getheaders()` 메서드, HTTPResponse 객체 620  
`gethostbyaddr()` 함수, socket 모듈 584  
`gethostbyname()` 함수, socket 모듈 584  
`gethostbyname_ex()` 함수, socket 모듈 584  
`gethostname()` 함수, socket 모듈 584  
`getinfo()` 메서드, ZipFile 객체 402  
`getinnerframes()` 함수, inspect 모듈 275  
`getint()` 메서드, ConfigParser 객체 410  
`getitem()` 함수, operator 모듈 338  
`__getitem__()` 메서드 70  
  문할 70  
`getiterator()` 메서드  
  Element 객체 714  
  ElementTree 객체 711  
`getitimer()` 함수, signal 모듈 492  
`getLength()` 메서드, SAX 속성 객체 720  
`getLevelName()` 함수, logging 모듈 451  
`getlist()` 메서드, FieldStorage 객체 661  
`getloadavg()` 함수, os 모듈 486  
`getLogger()` 함수, logging 모듈 438  
`getlogin()` 함수, os 모듈 467  
`getmember()` 메서드, TarFile 객체 395  
`getmembers()` 메서드, TarFile 객체 395  
`getmembers()` 함수, inspect 모듈 275  
`getmodule()` 함수, inspect 모듈 275  
`getmoduleinfo()` 함수, inspect 모듈 275  
`getmodulename()` 함수, inspect 모듈 276  
`getmro()` 함수, inspect 모듈 276  
`getmtime()` 함수, os.path 모듈 489  
`getName()` 메서드, Thread 객체 538  
`getNameByQName()` 메서드, SAX 속성 객체 721  
`getnameinfo()` 함수, socket 모듈 584  
`getNames()` 메서드, SAX 속성 객체 720  
`getnames()` 메서드, TarFile 객체 395  
 `getopt` 모듈 466  
`getouterframes()` 함수, inspect 모듈 276  
`getoutput()` 함수, commands 모듈 408  
`getpeer cert()` 메서드, ssl 객체 600  
`getpeername()` 메서드, socket 객체 589  
`getpgid()` 함수, os 모듈 468  
`getpgrp()` 함수, os 모듈 468  
`getpid()` 함수, os 모듈 468  
`getpos()` 메서드, HTMLParser 객체 694  
`getppid()` 함수, os 모듈 468  
`getprofile()` 함수, sys 모듈 289  
`getprotobyname()` 함수, socket 모듈 585  
`get QNameByName()` 메서드, SAX 속성 객체 721  
`getQNames()` 메서드, SAX 속성 객체 721  
`getrandbits()` 함수, random 모듈 313  
`getrecursionlimit()` 함수, sys 모듈 135, 289  
`getrefcount()` 함수, sys 모듈 42, 289  
`getresponse()` 메서드, HTTPConnection 객체 619  
`getroot()` 메서드, ElementTree 객체 711  
`getservbyname()` 함수, socket 모듈 585  
`getservbyport()` 함수, socket 모듈 585  
`GetSetDescriptorType 타입` 292  
`getsid()` 함수, os 모듈 468  
`getsignal()` 함수, signal 모듈 491  
`getsize()` 함수, os.path 모듈 489  
`getsizeof()` 함수, sys 모듈 237, 289  
`getslice()` 함수, operator 모듈 339  
`getsockname()` 메서드, socket 객체 589  
`getsockopt()` 메서드, socket 객체 589  
`getsource()` 함수, inspect 모듈 276  
`getsourcefile()` 함수, inspect 모듈 276  
`getsourcelines()` 함수, inspect 모듈 276  
`getstate()` 함수, random 모듈 312  
`__getstate__()` 메서드 282  
`복사` 270  
`pickle` 모듈 211  
`getstatusoutput()` 함수, commands 모듈 408  
`gettarinfo()` 메서드, TarFile 객체 395  
`gettempdir()` 함수, tempfile 모듈 399  
`gettempprefix()` 함수, tempfile 모듈 399  
`gettext` 모듈 726

gettimeout() 메서드, socket 객체 593  
 gettrace() 함수, sys 모듈 289  
 getType() 메서드, SAX 속성 객체 720  
 getuid() 함수, os 모듈 468  
 geturl() 메서드, urlopen 객체 635  
 getvalue() 메서드  
     BytesIO 객체 433  
     FieldStorage 객체 660  
     StringIO 객체 435  
 getValue() 메서드, SAX 속성 객체 720  
 \_getvalue() 메서드, BaseProxy 객체 533  
 getValueByQName() 메서드, SAX 속성 객체 721  
 getwch() 함수, msvcr 모듈 459  
 getwche() 함수, msvcr 모듈 459  
 getweakrefcount() 함수, weakref 모듈 296  
 getweakrefs() 함수, weakref 모듈 296  
 getwindowsversion() 함수, sys 모듈 289  
 gi\_\* 속성, 생성기 객체 64  
 gid 속성, TarInfo 객체 396  
 glob 모듈 390  
 glob() 함수, glob 모듈 390  
 global문 20, 116  
     모듈 175  
 globals() 함수 253  
 \_\_globals\_\_ 속성, 함수 58, 118  
 gmtime() 함수, time 모듈 499  
 gname 속성, TarInfo 객체 396  
 goto문, 없음 101  
 group() 메서드, MatchObject 객체 351  
 groupby() 함수, itertools 모듈 335  
 groupdict() 메서드, MatchObject 객체 351  
 groupindex 속성, Regex 객체 349  
 groups() 메서드, MatchObject 객체 351  
 grp 모듈 725  
 gt() 함수, operator 모듈 338  
 \_\_gt\_\_() 메서드 68  
 guess\_all\_extensions() 함수, mimetypes 모듈 701  
 guess\_extension() 함수, mimetypes 모듈

701  
 guess\_type() 함수, mimetypes 모듈 700  
 GUI 프로그래밍, partial 함수 평가 사용 331  
 GUI, 네트워크 프로그래밍 575  
 gzip 모듈 390  
 GzipFile() 함수, gzip 모듈 391  
**H**  
 —h 명령줄 옵션 213  
 h(elp) 디버거 명령, pdb 모듈 232  
 handle() 메서드, BaseRequestHandler 객체 602  
 handle() 함수, cgiib 모듈 666  
 handle\_accept() 메서드, dispatcher 객체 561  
 handle\_charref() 메서드, HTMLParser 객체 694  
 handle\_close() 메서드, dispatcher 객체 561  
 handle\_comment() 메서드, HTMLParser 객체 694  
 handle\_connect() 메서드, dispatcher 객체 561  
 handle\_data() 메서드, HTMLParser 객체 694  
 handle\_decl() 메서드, HTMLParser 객체 694  
 handle\_endtag() 메서드, HTMLParser 객체 694  
 handle\_entityref() 메서드, HTMLParser 객체 695  
 handle\_error() 메서드  
     dispatcher 객체 561  
     SocketServer 클래스 607  
 handle\_expt() 메서드, dispatcher 객체 561  
 handle\_pi() 메서드, HTMLParser 객체 695  
 handle\_read() 메서드, dispatcher 객체 561

handle\_startendtag() 메서드, HTMLParser 객체 695  
 handle\_starttag() 메서드, HTMLParser 객체 695  
 handle\_timeout() 메서드, SocketServer 클래스 607  
 handle\_write() 메서드, dispatcher 객체 561  
 has\_data() 메서드, Request 객체 637  
 has\_header() 메서드  
     Request 객체 637  
     Sniffer 객체 680  
 has\_ip6 변수, socket 모듈 585  
 has\_key() 메서드, 사전 54  
 has\_option() 메서드, ConfigParser 객체 410  
 has\_section() 메서드, ConfigParser 객체 410  
 hasattr() 함수 253  
     개인 속성 156  
 hasAttribute() 메서드, DOM Element 객체 708  
 hasAttributeNS() 메서드, DOM Element 객체 708  
 hasAttributes() 메서드, DOM Node 객체 706  
 hasChildNodes() 메서드, DOM Node 객체 706  
 hash() 함수 253  
 \_\_hash\_\_() 메서드 68  
 Hashable 추상 기반 클래스 328  
 hashlib 모듈 691  
     예 523~524  
 header\_offset 속성, ZipInfo 객체 404  
 headers 속성  
     BaseHTTPRequestHandler 객체 626  
     FieldStorage 객체 660  
 heapify() 함수, heapq 모듈 333  
 heapmin() 함수, msvcr 모듈 459  
 heappop() 함수, heapq 모듈 333  
 heappush() 함수, heapq 모듈 333  
 heappushpop() 함수, heapq 모듈 333

heapq 모듈 333	asyncore 모듈 예 563	I
heapreplace() 함수, heapq 모듈 333	POST 요청에서 파일 업로드 621	
hello world 프로그램 4	HTTP 요청에서 user-agent 헤더, 변경	
help() 함수 27, 253	637	-i 명령줄 옵션 213~214
이상한 출력 장식자 137	HTTP 쿠키 628	I/O 다중화 565
herror 예외, socket 모듈 596	http 패키지 615	I/O 베파링, 생성기 201
hex() 메서드, 부동 소수점 47	HTTP 프로토콜	__iadd__() 메서드 74
hex() 함수 93, 253	설명 615~616	__iand__() 메서드 74
future_builtins 모듈 268	요청 방식 616	IBM 범용 십진수 계산 표준 299
hexdigest() 메서드	응답 코드 616	id() 함수 40, 253
digest 객체 691	http.client 모듈 617	ident 속성, Thread 객체 538
hmac 객체 692	http.cookiejar 모듈 631	IDLE 3~5
hexdigits 변수, string 모듈 353	http.cookies 모듈 628	표준 I/O 스트림 198
hexversion 변수, sys 모듈 285	http.server 모듈 622	__idiv__() 메서드 74
HIGHEST_PROTOCOL 상수, pickle 모듈 211	HTTPBasicAuthHandler 클래스, urllib.	__idivmod__() 메서드 73
HKEY_* 상수, winreg 모듈 502	request 모듈 638	IEEE 754 299
hmac 모듈 692	HTTPConnection() 함수, http.client 모듈 618	if문 8, 98
HMAC 인증 692	HTTPCookieProcessor 클래스, urllib.	__debug__ 변수 110
hostname 속성	request 모듈 638	ifilter() 함수, itertools 모듈 335
urllibparse 객체 641	HTTPDefaultErrorHandler 클래스, urllib.	ifilterfalse() 함수, itertools 모듈 336
urlsplit 객체 642	request 모듈 638	__floordiv__() 메서드 74
hour 속성, time 객체 416	HTTPDigestAuthHandler 클래스, urllib.	iglob() 함수, glob 모듈 390
HTML 파싱 693	request 모듈 638	ignoreableWhitespace() 메서드,
HTML 폼	HTTPError 예외, urllib.error 모듈 645	ContentHandler 객체 719
예 655	HTTPException 예외, http.client 모듈 620	ignore 디버거 명령, pdb 모듈 232
urllib 패키지에서 업로드 634	HTTPHandler 클래스	'ignore' 예러 처리, 유니코드 인코딩 203
html.parser 모듈 693	logging 모듈 445	ignore_environment 속성, sys.flags 284
HTMLParser 모듈, html.parser 참고 693	urllib.request 모듈 638	ignore_pattern() 함수, shutil 모듈 391
HTMLParser 클래스, html.parser 모듈 693	httplib 모듈, http.client 참고 617	ignore_zeros 속성, TarFile 객체 396
HTTP 서버	HTTPRedirectHandler 클래스, urllib.	__ilshift__() 메서드 73
독립 서버 예 625	request 모듈 638	imag 속성
방화벽 예 623	HTTPResponse 객체, http.client 모듈 619	복소수 47
커스텀 요청 처리 627~628	HTTPSConnection() 함수, http.client 모듈 618	부동 소수점 47
코루틴 예 573	HTTPServer 클래스, http.server 모듈 623	imap() 메서드, Pool 객체 522
asynchat 모듈 예 558	HTTPSHandler 클래스, urllib.request 모듈 638	imap() 함수, itertools 모듈 336
	HTTP에서 user-agent 헤더 변경 635	imap_unordered() 메서드, Pool 객체 522
	hypot() 함수, math 모듈 309	imaplib 모듈 725
		imghdr 모듈 726
		__imod__() 메서드 74
		imp 모듈 275, 724
		import 때 모듈 이름 변경 176
		import문 13, 27, 61, 175~177
		다중 모듈 176

대문자 구분 182  
 로드된 코드의 유효 범위 규칙 178  
 메인 프로그램 179  
 모듈 검색 경로 180  
 모듈 실행 175~176  
 상대적인 패키지 임포트 184  
 일회성 실행 모듈 176  
 타입 모듈 181  
 파이썬 3 185  
 패키지 184  
 패키지에서 절대 임포트 185  
 프로그램에서 배치 176  
 as 한정어 176  
 .pyc 파일 컴파일 182  
 sys.modules 177  
 sys.path 변수 180  
 ImportError 예외 105, 182, 264  
 ImproperConnectionState 예외, http.client 모듈 620  
 \_\_imul\_\_() 메서드 74  
 in 연산자 9  
 부분 문자열 적용 82  
 사전 17, 54, 89  
 순서열 80, 82  
 \_\_contains\_\_ 메서드 71  
 in\_dll() 메서드, ctypes 타입 객체 760  
 INADDR\_\* 상수, socket 모듈 589  
 IncompleteRead 예외, http.client 모듈 620  
 IncrementalDecoder 클래스, codecs 모듈 343  
 incrementaldecoder() 메서드, CodecInfo 객체 343  
 IncrementalEncoder 클래스, codecs 모듈 343  
 incrementalencoder() 메서드, CodecInfo 객체 343  
 IndentationError 예외 105, 264  
 index() 메서드  
 리스트 48  
 문자열 50, 52  
 배열 객체 321

IndexError 예외 83, 105, 264  
 indexOf() 함수, operator 모듈 338  
 indices() 메서드, 분할 65  
 inet\_aton() 함수, socket 모듈 586  
 inet\_ntoa() 함수, socket 모듈 586  
 inet\_ntop() 함수, socket 모듈 586  
 inet\_pton() 함수, socket 모듈 586  
 Inf  
 무한대 264  
 decimal 모듈 300  
 Inf 변수, decimal 모듈 305  
 info() 메서드  
 Logger 객체 439  
 urlopen 객체 635  
 infolist() 메서드, ZipFile 객체 402  
 .ini 파일  
 로깅 설정 453  
 파이썬에서 읽기 408  
 init() 함수, mimetypes 모듈 701  
 \_\_init\_\_() 메서드 60, 66  
 메타클래스 171  
 상속 145  
 여러 인스턴스 생성 메서드 정의 149  
 예외 106  
 인스턴스 생성 157  
 클래스 24, 142  
 pickle 281  
 \_\_init\_\_.py 파일 실행 184  
 \_\_init\_\_.py 파일 패키지에서 183  
 input() 함수 198, 254  
 파이썬 3 10  
 insert() 메서드  
 리스트 12, 49  
 배열 객체 321  
 Element 객체 714  
 insertBefore() 메서드, DOM Node 객체 706  
 insort() 함수, bisect 모듈 323  
 insort\_left() 함수, bisect 모듈 323  
 insort\_right() 함수, bisect 모듈 323  
 inspect 모듈 273  
 inspect 속성, sys.flags 284

install 명령어, setup.py 파일 188~189  
 install\_opener() 함수, urllib.request 모듈 638  
 \_\_instancecheck\_\_() 메서드 69, 166  
 int 타입 45  
 int() 함수 11, 74, 92, 254  
 \_\_int\_\_() 메서드 73  
 강제 타입 변환 164  
 Integral 추상 기반 클래스 311  
 IntegrityError 예외, 데이터베이스 API 372  
 interactive 속성, sys.flags 284  
 InterfaceError 예외, 데이터베이스 API 371  
 internal\_attr 속성, ZipInfo 객체 404  
 InternalError 예외, 데이터베이스 API 372  
 interrupt() 메서드, Connection 객체 377  
 intersection() 메서드, 집합 56  
 intersection\_update() 메서드, 집합 56  
 inv() 함수, operator 모듈 338  
 InvalidURL 예외, http.client 모듈 620  
 invert() 함수, operator 모듈 338  
 \_\_invert\_\_() 메서드 73  
 io 모듈 429  
 관련 문제점 436  
 파이썬 3 780  
 IOBase 추상 기반 클래스 435  
 IOBase 클래스, io 모듈 429  
 ioctl() 메서드, socket 객체 593  
 ioctl() 함수, fcntl 모듈 428  
 IOError 예외 105, 264  
 \_\_ior\_\_() 메서드 74  
 IP\_\* 소켓 옵션, socket 모듈 591  
 \_\_ipow\_\_() 메서드 74  
 IPPROTO\_\* 상수, socket 모듈 587  
 IPv4 프로토콜 578  
 주소 형식 579  
 IPv6 프로토콜 578  
 주소 형식 580  
 IPV6\_\* 소켓 옵션, socket 모듈 592  
 IronPython 3  
 예 766  
 \_\_irshift\_\_() 메서드 73

is 연산자, 객체 신원 40, 94  
 is\_() 함수, operator 모듈 339  
 is\_alive() 메서드  
     Process 객체 511  
     Thread 객체 538  
 is\_multipart() 메서드, Message 객체 685  
 is\_not() 함수, operator 모듈 339  
 is\_set() 메서드, Event 객체 542  
 is\_tarfile() 함수, tarfile 모듈 393  
 is\_unverifiable() 메서드, Request 객체 637  
 is\_zipfile() 함수, zipfile 모듈 400  
 isabs() 함수, os.path 모듈 489  
 isabstract() 함수, inspect 모듈 276  
 isAlive() 메서드, Thread 객체 538  
 isalnum() 메서드, 문자열 50, 52  
 isalpha() 메서드, 문자열 52  
 isatty() 메서드  
     파일 195  
     IOBase 객체 430  
 isatty() 함수, os 모듈 472  
 isblk() 메서드, TarInfo 객체 396  
 isbuiltin() 함수, inspect 모듈 276  
 ischr() 메서드, TarInfo 객체 396  
 isclass() 함수, inspect 모듈 276  
 iscode() 함수, inspect 모듈 276  
 isDaemon() 메서드, Thread 객체 538  
 isdatadescriptor() 함수, inspect 모듈 277  
 isdev() 메서드, TarInfo 객체 396  
 isdigit() 메서드, 문자열 52  
 isdir() 메서드, TarInfo 객체 396  
 isdir() 함수, os.path 모듈 489  
 isdisjoint() 메서드, 집합 56  
 iselement() 함수, xml.etree.ElementTree 모듈 715  
 isenabled() 함수, gc 모듈 272  
 isEnabledFor() 메서드, Logger 객체 441  
 isfifo() 메서드, TarInfo 객체 396  
 isfile() 메서드, TarInfo 객체 396  
 isfile() 함수, os.path 모듈 489  
 isframe() 함수, inspect 모듈 277  
 isfunction() 함수, inspect 모듈 277  
 isgenerator() 함수, inspect 모듈 277  
 isgeneratorfunction() 함수, inspect 모듈 277  
 isinf() 함수, math 모듈 309  
 isinstance() 함수 41, 44, 165, 254  
     대리자 객체 165  
     상속 165  
     작동 방식 재정의 166  
 islice() 함수, itertools 모듈 336  
 islink() 함수, os.path 모듈 489  
 islnk() 메서드, TarInfo 객체 396  
 islower() 메서드, 문자열 52  
 ismethod() 함수, inspect 모듈 277  
 ismethoddescriptor() 함수, inspect 모듈 277  
 ismodule() 함수, inspect 모듈 277  
 ismount() 함수, os.path 모듈 489  
 isnan() 함수, math 모듈 310  
 iso-8859-1 인코딩, 설명 206  
 isocalendar() 메서드, date 객체 415  
 isoformat() 메서드  
     date 객체 415  
     time 객체 417  
 isoweekday() 메서드, date 객체 415  
 isreg() 메서드, TarInfo 객체 396  
 isReservedKey() 메서드, Morsel 객체 630  
 isroutine() 함수, inspect 모듈 277  
 isSameNode() 메서드, DOM Node 객체 706  
 isspace() 메서드, 문자열 52  
 issubclass() 함수 165, 254  
     작동 방식 재정의 166  
 issubset() 메서드, 집합 56  
 issuperset() 메서드, 집합 56  
 issym() 메서드, TarInfo 객체 396  
 istitle() 메서드, 문자열 52  
 istraceback() 함수, inspect 모듈 277  
 \_\_isub\_\_() 메서드 74  
 isupper() 메서드, 문자열 50, 52  
 itemgetter() 함수, operator 모듈 339  
 items() 메서드  
     사전 54  
 파이썬 3에서 사전 782  
 ConfigParser 객체 410  
 Element 객체 714  
 Message 객체 682  
 itemsize 속성, 배열 객체 320  
 ItemsView 추상 기반 클래스 329  
 iter() 함수 254  
 \_\_iter\_\_() 메서드 72, 98  
 \_\_itruediv\_\_() 메서드 74  
 \_\_ixor\_\_() 메서드 74  
 \_\_le\_\_() 메서드 68  
 \_\_len\_\_() 메서드 68, 70  
     진리 값 검사 68  
 \_\_long\_\_() 메서드 73  
 \_\_lshift\_\_() 메서드 73  
 \_\_lt\_\_() 메서드 68  
 Iterable 추상 기반 클래스 328  
 Iterator 추상 기반 클래스 328  
 iterdecode() 함수, codecs 모듈 344  
 iterdump() 메서드, Connection 객체 377  
 iterencode() 메서드, JSONEncoder 객체 700  
 iterencode() 함수, codecs 모듈 344  
 iterkeyrefs() 메서드, WeakKeyDictionary 객체 297  
 iterparse() 함수, xml.etree.ElementTree 모듈 715  
 itertools 모듈 100, 334  
 itervalueref() 메서드,  
     WeakValueDictionary 객체 298  
 \_\_itruediv\_\_() 메서드 74  
 \_\_ixor\_\_() 메서드 74  
 izip() 함수, itertools 모듈 100, 261, 336  
 izip\_longest() 함수, itertools 모듈 336

**J**

J 문자, 복소수 상수 32  
 j(ump) 디버거 명령, pdb 모듈 232  
 join() 메서드  
     문자열 52  
 JoinableQueue 객체 516

Pool 객체 522  
 Process 객체 511  
 Queue 객체 550  
 Thread 객체 538  
 join() 함수, os.path 모듈 489  
 join\_thread() 메서드, Queue 객체 515  
 JoinableQueue() 함수, multiprocessing 모듈 515  
 js\_output() 메서드  
 Morsel 객체 630  
 SimpleCookie 객체 630  
 json 모듈 696  
 pickle과 marshal와의 차이점 699  
 JSON(JavaScript Object Notation) 696  
 JSONDecoder 클래스, json 모듈 699  
 JSONEncoder 클래스, json 모듈 699  
 jumpahead() 함수, random 모듈 312  
 Jython 3  
 예 765

**K**

kbhit() 함수, msvcr 모듈 459  
 key 속성, Morsel 객체 631  
 key 키워드 인수, sort() 49  
 KEY\_\* 상수, winreg 모듈 504  
 KeyboardInterrupt 예외 104, 198  
 KeyboardInterrupt 클래스 264  
 KeyError 예외 54, 105, 264  
 keyrefs() 메서드, WeakKeyDictionary 객체 297  
 keys() 메서드  
 사전 54  
 파이썬 3에서 사전 782  
 Element 객체 714  
 Message 객체 682  
 KeysView 추상 기반 클래스 329  
 keyword 모듈 724  
 keywords 속성, partial 객체 331  
 kill() 메서드, Popen 객체 496  
 kill() 함수, os 모듈 482  
 killpg() 함수, os 모듈 482

kqueue, BSD 566

**L**

L 문자, 긴 정수 31  
 l(list) 디버거 명령, pdb 모듈 232  
 lambda 연산자 57, 134  
 대안 339  
 LambdaType 타입 292  
 last\_accepted 속성, Listener 객체 534  
 last\_traceback 변수, sys 모듈 285  
 last\_type 변수, sys 모듈 285  
 last\_value 변수, sys 모듈 285  
 lastChild 속성, DOM Node 객체 705  
 lastgroup 속성, MatchObject 객체 352  
 lastindex 속성, MatchObject 객체 352  
 latin-1 인코딩, 설명 206  
 lchflags() 함수, os 모듈 476  
 lchmod() 함수, os 모듈 476  
 lchown() 함수, os 모듈 476  
 ldegp() 함수, math 모듈 310  
 le() 함수, operator 모듈 338  
 \_\_le\_\_() 메서드 68  
 left\_list 속성, dircmp 객체 388  
 left\_only 속성, dircmp 객체 388  
 len() 함수 71, 254  
 매핑 54  
 사전 89  
 순서열 47~48, 80, 82  
 집합 56, 90  
 \_\_len\_\_() 메서드 68, 70  
 진리 값 검사 68  
 length 속성, HTTPResponse 객체 620  
 letters 변수, string 모듈 353  
 levelname 속성, Record 객체 442  
 levelno 속성, Record 객체 442  
 lexists() 함수, os.path 모듈 489  
 LifoQueue() 함수, queue 모듈 549  
 limit\_denominator() 메서드, Fraction 객체 308  
 line\_buffering 속성, TextIOWrapper 객체 434

linecache 모듈 724

lineno 속성, Record 객체 442  
 linesep 변수, os 모듈 466  
 link() 함수, os 모듈 476  
 linkname 속성, TarInfo 객체 397  
 list 타입 45  
 list() 메서드  
 Manager 객체 528  
 TarFile 객체 396  
 list() 함수 13, 49, 92, 255  
 사전에 적용 17  
 list\_dialects() 함수, csv 모듈 681  
 listdir() 함수  
 파이썬 3 780, 784  
 os 모듈 476  
 listen() 메서드  
 dispatcher 객체 562  
 socket 객체 594  
 Listener 클래스, multiprocessing 모듈 534  
 ljust() 메서드, 문자열 52  
 ln() 메서드, Decimal 객체 301  
 load() 메서드  
 SimpleCookie 객체 630  
 Unpickler 객체 281  
 load() 함수  
 json 모듈 698  
 marshal 모듈 278  
 pickle 모듈 210, 280  
 loads() 함수  
 json 모듈 699  
 marshal 모듈 279  
 pickle 모듈 280  
 xmlrpc.client 모듈 649  
 local 변수 115~116  
 스택 프레임에서 저장소 63  
 eval() 139  
 local() 함수, threading 모듈 547  
 localcontext() 함수, decimal 모듈 304  
 locale 모듈 726  
 localName 속성, DOM Node 객체 705  
 locals() 함수 255  
 localtime() 함수, time 모듈 500

Lock 객체	lookup() 함수	future_builtins 모듈 268
multiprocessing 모듈 526	codecs 모듈 341	map_async() 메서드, Pool 객체 523
threading 모듈 540	unicodedata 모듈 363	Mapping 추상 기반 클래스 329
Lock() 메서드, Manager 객체 528	LookupError 예외 105, 262	MappingView 추상 기반 클래스 329
LOCK_* 상수, flock() 함수 428	loop() 함수, asyncore 모듈 562	map-reduce, multiprocessing 모듈 521
lockf() 함수, fcntl 모듈 428	lower() 메서드, 문자열 52	marshal 모듈 278
locking() 함수, msvcrt 모듈 459	lowercase 변수, string 모듈 353	match() 메서드, Regex 객체 350
log() 메서드, Logger 객체 441	lseek() 함수, os 모듈 472	match() 함수, re 모듈 349
log() 함수, math 모듈 310	lshift() 함수, operator 모듈 338	MatchObject 객체, re 모듈 350
log_error() 메서드,	__lshift__() 메서드 73	math 모듈 309
BaseHTTPRequestHandler 객체 627	lstat() 함수, os 모듈 477	max 속성
log_message() 메서드,	lstrip() 메서드, 문자열 52	date 클래스 414
BaseHTTPRequestHandler 객체 627	lt() 함수, operator 모듈 338	datetime 클래스 418
log_request() 메서드,	__lt__() 메서드 68	sys.float_info 284
BaseHTTPRequestHandler 객체 627	LWPCookieJar 클래스, http.cookiejar 모듈	time 클래스 416
log10() 메서드, Decimal 객체 301	632	timedelta 클래스 420
log10() 함수, math 모듈 310	<b>M</b>	max() 함수 14, 47~48, 80, 82, 255
log1p() 함수, math 모듈 310	-m pdb 옵션 인터프리터 234	사용자 정의 객체에서 필요 메서드 69
LogAdapter() 함수, logging 모듈 450	-m 명령줄 옵션 213~214	집합 90
logging 모듈 436	mailbox 모듈 725	max_10_exp 속성, sys.float_info 284
기본 설정 437	mailcap 모듈 725	max_exp 속성, sys.float_info 284
널 로거 사용 455	main() 함수, unittest 모듈 227	maxint 변수, sys 모듈 285
로거 계층 442	__main__ 모듈 179, 214	maxsize 변수, sys 모듈 285
로거 이름 고르기 439	__main__, multiprocessing 모듈을 위해 확인 필요 513	maxunicode 변수, sys 모듈 285
로그 메시지 생성 439	major() 함수, os 모듈 477	md5() 함수, hashlib 모듈 691
로그 메시지에 예외 포함 441	make_server() 함수, wsgiref.simple_server 모듈 668	MemberDescriptorType 타입 292
로그 메시지에 필드 추가 450	makedef() 함수, os 모듈 477	memmove() 함수, ctypes 모듈 761
메시지 전파 443	makedirs() 함수, os 모듈 477	MemoryError 예외 105, 264
메시지 처리 444	makefile() 메서드, socket 객체 594	MemoryHandler 클래스, logging 모듈 446
메시지 포맷 지정 450	maketrans() 함수, string 모듈 357	memset() 함수, ctypes 모듈 762
메시지 필터링 441	managed 객체, multiprocessing 모듈 527	merge() 함수, heapq 모듈 333
설정 방법 452	Manager() 함수, multiprocessing 모듈 527	message 속성, Exception 객체 263
handler 객체 445	mant_dig 속성, sys.float_info 284	Message 클래스, email 패키지 682, 686
.ini 파일로 설정 453	map() 메서드, Pool 객체 522	message_from_file() 함수, email package 682
multiprocessing 모듈 535	map() 함수 255	message_from_string() 함수, email package 682
login() 메서드	최적화 244	metaclass 키워드 인수, 클래스 정의 170
FTP 객체 612	파이썬 3 255	__metaclass__ 속성, 클래스 170
SMTP 객체 633		__metaclass__ 전역 변수 170
lognormvariate() 함수, random 모듈 314		methodcaller() 함수, operator 모듈 339
long 타입 45		
long() 함수 255		
__long__() 메서드 73		

methodHelp() 메서드, ServerProxy 객체 648  
 methodSignatures() 메서드, ServerProxy 객체 647  
 MethodType 타입 57, 293  
 microsecond 속성, time 객체 416  
 MIMEApplication 클래스, email 패키지 689  
 MIMEAudio 클래스, email 패키지 689  
 MIMEImage 클래스, email 패키지 689  
 MIMEMessage 클래스, email 패키지 689  
 MIMEMultipart 클래스, email 패키지 690  
 MIMEText 클래스, email 메세지 690  
 mimetypes 모듈 700  
 min 속성  
     date 클래스 414  
     datetime 클래스 418  
     sys.float\_info 284  
     time 클래스 416  
     timedelta 클래스 420  
 min() 함수 14, 47~48, 80, 82, 256  
     사용자 정의 객체에서 필요 메서드 69  
     집합 90  
 min\_10\_exp 속성, sys.float\_info 284  
 min\_exp 속성, sys.float\_info 284  
 minor() 함수, os 모듈 477  
 minute 속성, time 객체 416  
 mirrored() 함수, unicodedata 모듈 363  
 mkd() 메서드, FTP 객체 613  
 mkdir() 함수, os 모듈 477  
 mkdtemp() 함수, tempfile 모듈 398  
 mkfifo() 함수, os 모듈 477  
 mknod() 함수, os 모듈 477  
 mkstemp() 함수, tempfile 모듈 398  
 mktemp() 함수, tempfile 모듈 398  
 mktime() 함수, time 모듈 500  
 mmap 모듈 455  
 mmap() 함수, mmap 모듈 456  
 mod() 함수, operator 모듈 338  
     \_\_mod\_\_() 메서드 73  
 mode 속성  
     파일 197

FileIO 객체 430  
 TarInfo 객체 397  
 modf() 함수, math 모듈 310  
 module 속성, Record 객체 442  
     \_\_module\_\_ 속성, 타입 61  
 modulefinder 모듈 724  
 modules 변수, sys 모듈 182, 285  
 ModuleType 타입 57, 293  
 month 속성, date 객체 415  
 Morsel 클래스, http.cookies 모듈 629  
 move() 메서드, mmap 객체 457  
 move() 함수, shutil 모듈 393  
 MozillaCookieJar 클래스, http.cookiejar 모듈 632  
     \_\_mro\_\_ 속성, 클래스 147  
 MSG\_\* 상수, socket 모듈 594  
 msvcr 모듈 458  
 mtime 속성, TarInfo 객체 397  
 mul() 함수, operator 모듈 338  
     \_\_mul\_\_() 메서드 73  
 MultiCall() 함수, xmlrpc.client 모듈 649  
 multiprocessing 모듈 510  
     공유 메모리 525  
     공유 메모리를 통해 리스트 전달 526  
     동기화 기본 기능 526  
     로깅 535  
     별개 프로세스 연결 533  
     분산 컴퓨팅 536  
     전역 인터프리터 락 548  
     큐 514  
     파이프 518  
     프로세스 풀 521  
     피클링 536  
     \_\_main\_\_ 확인에 사용 513  
     managed 객체 527  
 MutableMapping 추상 기반 클래스 329  
 MutableSequence 추상 기반 클래스 329  
 MutableSet 추상 기반 클래스 329  
 MySQL, 파이썬에서 접근 365

N  
 n(ext) 디버거 명령, pdb 모듈 232  
 name 변수, os 모듈 466  
 name 속성  
     파일 197  
 FieldStorage 객체 660  
 FileIO 객체 430  
 Process 객체 512  
 Record 객체 442  
 TarInfo 객체 397  
 Thread 객체 538  
 name() 함수, unicodedata 모듈 363  
     \_\_name\_\_ 변수, 모듈 179  
     \_\_name\_\_ 속성  
         내장 함수 60  
         메서드 60  
         모듈 62  
         타입 61  
         함수 58  
 NamedTemporaryFile() 함수, tempfile 모듈 399  
 namedtuple() 함수, collections 모듈 325  
 NameError 예외 105, 264  
 NameError 예외 \_\_del\_\_에서 무시 221  
 NameError 예외, 변수 검색 115  
 namelist() 메서드, ZipFile 객체 402  
 Namespace() 메서드, Manager 객체 528  
 namespaceURI 속성, DOM Node 객체 705  
 NaN  
     숫자 아님 264  
     숫자 아님, decimal 모듈 300  
 NaN 변수, decimal 모듈 305  
 ne() 함수, operator 모듈 338  
     \_\_ne\_\_() 메서드 68  
 neg() 함수, operator 모듈 338  
     \_\_neg\_\_() 메서드 73  
 negInf 변수, decimal 모듈 305  
 nested() 함수, contextlib 모듈 331  
 Netlink 프로토콜 578  
     주소 형식 581

netloc 속성  
 urlparse 객체 641  
 urlsplit 객체 642  
 netrc 모듈 725  
 new() 함수  
   hashlib 모듈 692  
   hmac 모듈 692  
 \_\_new\_\_() 메서드 66  
   메타클래스 171  
   변경 불가능 타입 사용 157  
 사용 66  
 인스턴스 생성 157  
 코드 읽을 때 주의 157  
 newline 매개변수, open() 함수 194  
 newlines 속성  
   파일 197  
   TextIOWrapper 객체 434  
 next() 메서드 72  
   반복자 98  
   생성기 21, 64, 123  
   코루틴과 함께 사용 124  
   파일 195  
   TarFile 객체 396  
 next() 함수 256  
 \_\_next\_\_() 메서드 72  
   반복자 98  
   생성기 21, 123  
   파이썬 3 783  
 nextset() 메서드, Cursor 객체 368  
 nextSibling 속성, DOM Node 객체 705  
 NL\_\* 상수, socket 모듈 585  
 nice() 함수, os 모듈 482  
 nis 모듈 725  
 nlargest() 함수, heapq 모듈 334  
 nntplib 모듈 725  
 no\_site 속성, sys.flags 284  
 nodeName 속성, DOM Node 객체 705  
 nodeType 속성, DOM Node 객체 705  
 nodeValue 속성, DOM Node 객체 706  
 None 46  
 기본 인수 112  
 함수에서 return문 115

nonlocal문, 파이썬 3 117, 771  
 normalize() 메서드, DOM Node 객체 706  
 normalize() 함수, unicodedata 모듈 209,  
   363  
 normalvariate() 함수, random 모듈 314  
 normcase() 함수, os.path 모듈 489  
 normpath() 함수, os.path 모듈 490  
 not 연산자, 불리언 표현식 8, 93  
   93  
 not\_() 함수, operator 모듈 338  
 NotConnected 예외, http.client 모듈 620  
 notify() 메서드, Condition 객체 544  
 notify\_all() 메서드, Condition 객체 545  
 NotImplemented 예외 105, 264  
 NotSupportedError 예외, 데이터베이스  
   API 372  
 now() 메서드, datetime 클래스 418  
 nsmallest() 함수, heapq 모듈 334  
 NTEventLogHandler 클래스, logging 모듈  
   446  
 ntohl() 함수, socket 모듈 586  
 ntohs() 함수, socket 모듈 586  
 ntransfercmd() 메서드, FTP 객체 613  
 NULL로 끝나는 문자열, UTF-8 208  
 Number 추상 기반 클래스 311  
 numbers 모듈 169, 310  
 numerator 속성  
   정수 47  
   Fraction 객체 308  
 numeric() 함수, unicodedata 모듈 364  
 numpy 확장 기능 46, 322  
   322

**O**

-O 명령줄 옵션 110, 181, 213~214, 454  
 object 기반 클래스 24, 144  
 object() 함수 256  
 oct() 함수 93, 256  
   future\_builtins 모듈 268  
 octdigits 변수, string 모듈 353  
   -OO 명령줄 옵션 181, 213~214

open() 메서드  
   controller 객체 671  
   ZipFile 객체 402  
 open() 함수 9, 193, 256  
   파이썬 2와 3 차이점 256~257  
   파이썬 3 194  
   파일 모드 설명 193  
   codecs 모듈 204, 343  
   codecs 모듈 파이썬 3 344  
   dbm 모듈 382  
   gzip 모듈 391  
   io 모듈 435  
   os 모듈 472  
   shelve 모듈 209, 383  
   tarfile 모듈 393  
   webbrowser 모듈 671  
 open\_new() 메서드, controller 객체 671  
 open\_new() 함수, webbrowser 모듈 672  
 open\_new\_tab() 함수, webbrowser 모듈  
   672  
 open\_osfhandle() 함수, msrvct 모듈 460  
 OpenKey() 함수, winreg 모듈 504  
 OpenKeyEx() 함수, winreg 모듈 504  
 openpty() 함수, os 모듈 474  
 OpenSSL 598  
   인증서 생성 예 602  
 OperationalError 예외, 데이터베이스 API  
   372  
 operator 모듈 338  
   최적화에 사용 339  
   lambda 대안 339  
 optimize 속성, sys.flags 284  
 OptionParser() 함수, optparse 모듈 461  
 options() 메서드, ConfigParser 객체 410  
 optionxform() 메서드, ConfigParser 객체  
   410  
 optparse 모듈 461  
   예 192  
 or 연산자, 불리언 표현식 8, 93  
 or\_() 함수, operator 모듈 338  
 \_\_or\_\_() 메서드 73  
 ord() 함수 93, 257

OS X 407  
os 모듈 193, 466  
os.environ 변수 193  
os.path 모듈 488  
OSError 예외 105, 265  
osaudiodev 모듈 726  
output() 메서드  
  Morsel 객체 630  
  SimpleCookie 객체 630  
OutputString() 메서드, Morsel 객체 631  
OverflowError 예외 265

**P**

p 디버거 명령, pdb 모듈 233  
P\_\* 상수, spawnv() 함수 483  
pack() 메서드, Struct 객체 358  
pack() 함수, struct 모듈 357  
pack\_into() 메서드, Struct 객체 358  
pack\_into() 함수, struct 모듈 357  
PACKET\_\* 상수, socket 모듈 581  
params 속성, urlparse 객체 641  
paramstyle 변수, 데이터베이스 API 370  
pardir 변수, os 모듈 475  
parentNode 속성, DOM Node 객체 706  
paretovariate() 함수, random 모듈 314  
parse() 메서드  
  ElementTree 객체 711  
  Formatter 객체 354  
parse() 함수  
  xml.dom.minidom 모듈 705  
  xml.etree.ElementTree 모듈 715  
  xml.sax 모듈 718  
parse\_args() 메서드, OptionParser 객체 192, 463  
parse\_header() 함수, cgi 모듈 662  
parse\_multipart() 함수, cgi 모듈 662  
parse\_qs() 함수, urllib.parse 모듈 642  
parse\_qsl() 함수, urllib.parse 모듈 643  
parser 모듈 724  
parseString() 함수  
  xml.dom.minidom 모듈 705

xml.sax 모듈 718  
partial() 함수  
  네트워크 처리기 628  
  functools 모듈 92, 331  
partition() 메서드, 문자열 50, 52  
pass문 8, 30, 98  
password 속성  
  urlparse 객체 641  
  urlsplit 객체 642  
path 변수  
  os 모듈 466  
  sys 모듈 180, 218, 285  
path 속성  
  BaseHTTPRequestHandler 객체 626  
  urlparse 객체 641  
  urlsplit 객체 642  
  \_\_path\_\_ 변수, 패키지에서 186  
  \_\_path\_\_ 속성, 모듈 62  
pathconf() 함수, os 모듈 477  
pathname 속성, Record 객체 442  
pathsep 변수, os 모듈 475  
pattern 속성, Regex 객체 350  
pause() 함수, signal 모듈 492  
pdb 모듈 229  
  명령줄에서 프로그램 디버깅 234  
  .pdbrc 설정 파일 234  
.pdbrc 설정 파일 234  
peek() 메서드, BufferedReader 객체 432  
PEM\_cert\_to\_DER\_cert() 함수, ssl 모듈 600  
PEP 249, 파이썬 데이터베이스 API 명세서 365  
PEP 333(WSGI) 666  
Perl  
  동적 유효 범위 117  
  숫자 문자열 해석과 파이썬 11  
permutations() 함수, itertools 모듈 336  
PHP, 숫자 문자열 해석과 파이썬 11  
pi 변수, math 모듈 310  
pickle 모듈 209, 279  
  보안 문제 211  
  비호환 객체 210  
프로토콜 선택 210  
copy 모듈과 상호작용 270  
cPickle 282  
\_\_main\_\_ 모듈 282  
multiprocessing 모듈 536  
shelve에 의해 사용 383  
pickle 프로토콜, shelve 모듈에서 선택 210  
Pickler 클래스, pickle 모듈 281  
pickletools 모듈 724  
pid 속성  
  Popen 객체 497  
  Process 객체 513  
Pipe() 함수, multiprocessing 모듈 518  
pipe() 함수, os 모듈 474  
pipes 모듈 724  
pkgutil 모듈 724  
placement 장식자 121  
platform 모듈 725  
platform 변수, sys 모듈 286  
plistlib 모듈 726  
plock() 함수, os 모듈 483  
pm() 함수, pdb 모듈 230  
POINTER() 함수, ctypes 모듈 758, 759  
poll() 메서드  
  Connection 객체 518  
  Poll 객체 565  
  Popen 객체 496  
poll() 함수, select 모듈 565  
POLL\* 상수, select 모듈 566  
Pool() 함수, multiprocessing 모듈 521  
pop() 메서드  
  리스트 49  
  배열 객체 321  
  사전 54, 113  
  집합 56  
  deque 객체 324  
popen() 함수, os 모듈 483  
Popen() 함수, subprocess 모듈 495  
popitem() 메서드, 사전 54  
popleft() 메서드, deque 객체 324  
poplib 모듈 725

port 속성  
 urlparse 객체 641  
 urlsplit 객체 642  
 pos 속성, MatchObject 객체 351  
 pos() 함수, operator 모듈 338  
 \_\_pos\_\_() 메서드 73  
 \_\_pow\_\_() 메서드 73  
 posix 속성, TarFile 객체 396  
 POSIX 인터페이스 407  
 post\_mortem() 함수, pdb 모듈 230  
 pow() 함수 78, 257  
     math 모듈 310  
 pp 디버거 명령, pdb 모듈 233  
 pprint 모듈 724  
 preamble 속성, Message 객체 685  
 prec 속성, Context 객체 304  
 predicate() 함수, itertools 모듈 336  
 prefix 변수, sys 모듈 218, 286  
     286  
 prefix 속성, DOM Node 객체 706  
 --prefix 옵션 setup.py 190  
 \_\_prepare\_\_() 메서드, 파이썬 3 메타클래스 776~777  
 previousSibling 속성, DOM Node 객체  
     706  
 print문 4, 198  
     끝 콤마 9  
     줄바꿈 문자 생략 198  
     파이썬 3 문법 오류 4  
     파일 전환 10, 199  
     파일의 softspace 속성 198  
     포맷 지정 출력 7, 198  
     \_\_str\_\_() 67  
     sys.stdout 197  
 print() 함수 199, 257  
     분리자 문자 199  
     줄바꿈 문자 생략 199  
     파이썬 2.6에서 활성화 199  
     파이썬 3 781  
     파일 전환 199  
 print\_directory() 함수, cgi 모듈 662  
 print\_environ() 함수, cgi 모듈 662

print\_environ\_usage() 함수, cgi 모듈 662  
 print\_exc() 함수, traceback 모듈 291  
 print\_exception() 함수, traceback 모듈  
     291  
 print\_form() 함수, cgi 모듈 662  
 print\_last() 함수, traceback 모듈 291  
 print\_stack() 함수, traceback 모듈 291  
 print\_tb() 함수, traceback 모듈 291  
 printable 변수, string 모듈 353  
 printdir() 메서드, ZipFile 객체 402  
 printf() 함수와 동등 7  
 PriorityQueue() 함수, queue 모듈 549  
 private 지정자, 없음 155  
 process 속성, Record 객체 442  
 Process() 함수, multiprocessing 모듈 511  
 processingInstruction() 메서드,  
     ContentHandler 객체 719  
 ProcessingInstruction() 함수, xml.etree.  
     ElementTree 모듈 712  
 product() 함수, itertools 모듈 336  
 profile 모듈 234  
 ProgrammingError 예외, 데이터베이스  
     API 372  
 propagate 속성, Logger 객체 443  
 @property 장식자 152  
 property() 함수 153, 257  
 protected 지정자, 없음 155  
 proto 속성, socket 객체 596  
 protocol 매개변수, pickle 함수 210  
 protocol\_version 속성,  
     BaseHTTPRequestHandler 클래스 626  
 ProtocolError 예외, xmlrpc.client 모듈  
     649  
 proxy() 함수, weakref 모듈 297  
 ProxyBasicAuthHandler 클래스, urllib.  
     request 모듈 638  
 ProxyDigestAuthHandler 클래스, urllib.  
     request 모듈 638  
 ProxyHandler 클래스, urllib.request 모듈  
     638  
 ProxyTypes 클래스, weakref 모듈 297  
 ps1 변수, sys 모듈 286

ps2 변수, sys 모듈 286  
 .pth 파일, 사이트 구성 218  
 pty 모듈 725  
 punctuation 변수, string 모듈 353  
 push() 메서드, asynchat 객체 557  
 push\_with\_producer() 메서드, asynchat  
     객체 557  
 put() 메서드, Queue 객체 515, 549  
 put\_nowait() 메서드, Queue 객체 515,  
     549  
 putch() 함수, msvcr 모듈 460  
 putenv() 함수, os 모듈 468  
 putheader() 메서드, HTTPConnection 객체  
     619  
 putrequest() 메서드, HTTPConnection 객체  
     618  
 putwch() 함수, msvcr 모듈 460  
 pwd 모듈 724  
 pwd() 메서드, FTP 객체 613  
 .py 파일 5, 180  
     라이브러리 모듈 26  
 .py 파일 더블 클릭 5  
 Py\_BEGIN\_ALLOW\_THREADS 매크로  
     749  
 Py\_BuildValue() 함수 743  
 py\_compile 모듈 724  
 Py\_DECREF() 매크로 748  
 Py\_END\_ALLOW\_THREADS 매크로 749  
 Py\_Finalize() 함수 751  
 Py\_GetExecPrefix() 함수 752  
 Py\_GetPath() 함수 752  
 Py\_GetPrefix() 함수 752  
 Py\_GetProgramFullPath() 함수 752  
 Py\_INCRE() 매크로 748  
 Py\_Initialize() 함수 751  
 Py\_IsInitialized() 함수 751  
 Py\_SetProgramName() 함수 751  
 Py\_XDECREF() 매크로 748  
 Py\_XINCRE() 매크로 748  
 py2app 패키지 190  
 py2exe 패키지 190  
 py3k\_warning 속성, sys.flags 284

py3kwarning 변수, sys 모듈 286  
PyArg\_ParseTuple() 함수 737  
PyArg\_ParseTupleAndKeywords() 함수 737  
PyBytes\_AsString() 함수 754  
.pyc 파일 180  
  생성 방지 283  
  생성될 때 181  
  import 때 컴파일 181  
.pyc 파일 생성 막기 283  
.pyc와 .pyo 파일 생성 181  
pyclbr 모듈 724  
.pyd 파일, 컴파일된 확장 기능 181  
pydev 3  
pydoc 명령 28  
PyErr\_Clear() 함수 747  
PyErr\_ExceptionMatches() 함수 747  
PyErr\_NoMemory() 함수 746  
PyErr\_Occurred() 함수 747  
PyErr\_SetFromErrno() 함수 746  
PyErr\_SetFromErrnoWithFilename() 함수 746  
PyErr\_SetObject() 함수 746  
PyErr\_SetString() 함수 746  
PyEval\_CallObject() 함수 753  
PyEval\_CallObjectWithKeywords() 함수 753  
PyExc\_\* 예외, in extension 모듈 746~747  
PyFloat\_AsDouble() 함수 754  
PyImport\_ImportModule() 함수 752  
PyInt\_AsLong() 함수 754  
PyLong\_AsLong() 함수 754  
PyModule\_AddIntConstant() 함수 745  
PyModule\_AddIntMacro() 함수 746  
PyModule\_AddObject() 함수 745  
PyModule\_AddStringConstant() 함수 746  
PyModule\_AddStringMacro() 함수 746  
.pyo 파일 180  
  생성될 때 181  
PyObject\_GetAttrString() 함수 752  
PyObject\_SetAttrString() 함수 753  
pypi (파이썬 패키지 인덱스) 190

pyprocessing 라이브러리 536~537  
PyRun\_AnyFile() 함수 751  
PyRun\_InteractiveLoop() 함수 751  
PyRun\_InteractiveOne() 함수 751  
PyRun\_SimpleFile() 함수 751  
PyRun\_SimpleString() 함수 751  
PyString\_AsString() 함수 754  
PySys\_SetArgv() 함수 752  
PYTHON\* 환경 변수 214  
Python.h 헤더 파일, 확장 733  
.pyw 파일 180, 218  
PyZipFile() 함수, zipfile 모듈 401

**Q**

-Q 명령줄 옵션 214  
q(uit) 디버거 명령, pdb 모듈 233  
qsize() 메서드, Queue 객체 515, 549  
query 속성  
  urlparse 객체 641  
  urlsplit 객체 642  
QueryInfoKey() 함수, winreg 모듈 504  
QueryValue() 함수, winreg 모듈 505  
QueryValueEx() 함수, winreg 모듈 505  
queue 모듈 549  
Queue() 메서드, Manager 객체 528  
Queue() 함수  
  multiprocessing 모듈 514  
  queue 모듈 549  
quit() 메서드  
  FTP 객체 613  
  SMTP 객체 633  
quopri 모듈 701  
quote() 함수, urllib.parse 모듈 643  
quote\_from\_bytes() 함수, urllib.parse 모듈 643  
quote\_plus() 함수, urllib.parse 모듈 643  
quoteattr() 함수, xml.sax.saxutils 모듈 722  
RCVALL\_\* 상수, socket 모듈 593  
\_\_rdiv\_\_() 메서드 73  
\_\_rdivmod\_\_() 메서드 73  
re 모듈 50, 82, 345  
re 속성, MatchObject 객체 352  
read() 메서드  
  파일 195  
BufferedReader 객체 432

**R**

'r' 모드, open() 함수 193  
r 문자, 문자열 상수 앞 34  
return 디버거 명령, pdb 모듈 233  
\_\_radd\_\_() 메서드 73  
  \_\_add\_\_()에 대해 호출 164  
radians() 함수, math 모듈 309  
radix 속성, sys.float\_info 285  
raise문 26, 101, 106  
RAND\_add() 함수, ssl 모듈 601  
RAND\_egd() 함수, ssl 모듈 601  
RAND\_status() 함수, ssl 모듈 601  
\_\_rand\_\_() 메서드 74

randint() 함수, random 모듈 313

random 모듈 312  
random() 함수, random 모듈 313  
randrange() 함수, random 모듈 313  
range() 함수 18, 258

파이썬 3에서 제거 18

range() 함수 주의 18

Rational 추상 기반 클래스 311

raw\_decode() 메서드, JSONDecoder 객체 699

raw\_input() 함수 10, 198, 258

파이썬 3 10, 258

RawArray() 함수, multiprocessing 모듈 525

RawConfigParser 클래스, configparser 모듈 413

RawIOBase 추상 기반 클래스 435

raw-unicode-escape 인코딩, 설명 209

RawValue() 함수, multiprocessing 모듈 525

RCVALL\_\* 상수, socket 모듈 593

\_\_rdiv\_\_() 메서드 73

\_\_rdivmod\_\_() 메서드 73

re 모듈 50, 82, 345

re 속성, MatchObject 객체 352

read() 메서드

  파일 195

BufferedReader 객체 432

ConfigParser 객체 410  
 FileIO 객체 431  
 HTTPResponse 객체 619  
 mmap 객체 457  
 ssl 객체 600  
 StreamReder 객체 342  
 TextIOWrapper 객체 434  
 urlopen 객체 635  
 ZipFile 객체 402  
 read() 함수, os 모듈 474  
 read\_byte() 메서드, mmap 객체 457  
 read\_mime\_types() 함수, mimetypes 모듈 701  
 read1() 메서드, BufferedReader 객체 432  
 readable() 메서드  
     dispatcher 객체 561  
     IOBase 객체 430  
 readall() 메서드, FileIO 객체 431  
 reader() 함수, csv 모듈 678  
 ReadError 예외, tarfile 모듈 397  
 readfp() 메서드, ConfigParser 객체 411  
 readinto() 메서드, BufferedReader 객체 432  
 readline 라이브러리 217  
 readline 모듈 725  
 readline() 메서드  
     파일 9, 195~196  
     IOBase 객체 430  
     mmap 객체 457  
     StreamReder 객체 342  
     TextIOWrapper 객체 434  
     urlopen 객체 635  
 readlines() 메서드  
     파일 13, 195  
     IOBase 객체 430  
     StreamReder 객체 342  
     urlopen 객체 635  
 readlink() 함수, os 모듈 478  
 ready() 메서드, AsyncResult 객체 523  
 real 속성  
     복소수 47  
     부동 소수점 47  
 Real 추상 기반 클래스 311  
 realpath() 함수, os.path 모듈 490  
 reason 속성, HTTPResponse 객체 620  
 Record 객체, logging 모듈 442  
 recv() 메서드  
     Connection 객체 518  
     dispatcher 객체 562  
     socket 객체 594  
 recv\_bytes() 메서드, Connection 객체 519  
 recv\_bytes\_into() 메서드, Connection 객체 519  
 recv\_into() 메서드, socket 객체 594  
 recvfrom() 메서드, socket 객체 595  
 recvfrom\_info() 메서드, socket 객체 595  
 recvmsg() 시스템 호출, 없음 598  
 reduce() 함수, functools 모듈 332  
     \_\_reduce\_\_() 메서드 282  
     \_\_reduce\_ex\_\_() 메서드 282  
 ref() 함수, weakref 모듈 296  
 ReferenceError 예외 105, 265  
 REG\_\* 상수, winreg 모듈 503  
 Regex 객체, re 모듈 349  
 register() 메서드  
     추상 기반 클래스 168  
     BaseManager 클래스 530  
     Poll 객체 566  
 register() 함수  
     atexit 모듈 221, 269  
     webbrowser 모듈 672  
 register\_adapter() 함수, sqlite3 모듈 375  
 register\_converter() 함수, sqlite3 모듈 375  
 register\_dialect() 함수, csv 모듈 681  
 register\_function() 메서드, XMLRPCServer 객체 650  
 register\_instance() 메서드, XMLRPCServer 객체 650  
 register\_introspection\_functions() 메서드, XMLRPCServer 객체 651  
 register\_multicall\_functions() 메서드, XMLRPCServer 객체 651  
 RegLoadKey() 함수, winreg 모듈 504  
 release() 메서드  
     Condition 객체 544  
     Lock 객체 540  
     RLock 객체 541  
     Semaphore 객체 541  
 reload() 함수 183  
 relpath() 함수, os.path 모듈 490  
 remove() 메서드  
     리스트 48~49  
     배열 객체 321  
     집합 16, 56  
     deque 객체 324  
     Element 객체 714  
 remove() 함수, os 모듈 478  
 remove\_option() 메서드, ConfigParser 객체 411  
 remove\_section() 메서드, ConfigParser 객체 411  
 removeChild() 메서드, DOM Node 객체 706  
 removedirs() 함수, os 모듈 478  
 removeFilter() 메서드  
     Handler 객체 448  
     Logger 객체 441  
 removeHandler() 메서드, Logger 객체 444  
 rename() 메서드, FTP 객체 613  
 rename() 함수, os 모듈 478  
 renames() 함수, os 모듈 478  
 repeat() 함수  
     cProfile 모듈 236  
     itertools 모듈 347  
     operator 모듈 338  
     timeit 모듈 236  
 replace() 메서드  
     문자열 50, 52  
     date 객체 415  
     datetime 객체 419  
     time 객체 417  
 'replace' 에러 처리, 유니코드 인코딩 203  
 replace\_header() 메서드, Message 객체 687

replaceChild() 메서드, DOM Node 객체 706  
 report() 메서드, dircmp 객체 387  
 report\_full\_closure() 메서드, dircmp 객체 388  
 report\_partial\_closure() 메서드, dircmp 객체 387  
 repr() 함수 11, 67, 92, 217, 258  
 eval() 67  
 str()과 차이 12  
 repr(reprlib) 모듈 724  
 \_\_repr\_\_() 메서드 67  
 request 속성, BaseRequestHandler 객체 602  
 request() 메서드, HTTPConnection 객체 619  
 Request() 함수, urllib.request 모듈 636  
 request\_queue\_size 속성, SocketServer 클래스 606  
 request\_version 속성, BaseHTTPRequestHandler 객체 626  
 RequestHandlerClass 속성, SocketServer 객체 606  
 reserved 속성, ZipInfo 객체 403  
 reset() 메서드  
     HTMLParser 객체 695  
     IncrementalDecoder 객체 343  
     IncrementalEncoder 객체 343  
     StreamReeder 객체 342  
     StreamWriter 객체 343  
 resetwarnings() 함수, warnings 모듈 296  
 resize() 메서드, mmap 객체 458  
 resize() 함수, ctypes 모듈 762  
 resolution 속성  
     date 클래스 415  
     datetime 클래스 418  
     time 클래스 416  
     timedelta 클래스 420  
 resource 모듈 725  
 ResponseNotReady 예외, http.client 모듈 621  
 responses 속성,

BaseHTTPRequestHandler 클래스 626  
 restype 속성, ctypes 함수 객체 756  
 retrbinary() 메서드, FTP 객체 613  
 retrlines() 메서드, FTP 객체 613  
 return문 115  
 returncode 속성, Popen 객체 497  
 reverse 키워드 인수, sort() 49  
 reverse() 메서드  
     리스트 49  
     배열 객체 321  
 reversed() 함수 258  
 rfile 속성  
     BaseHTTPRequestHandler 객체 626  
     StreamRequestHandler 객체 603  
 rfind() 메서드, 문자열 50, 52  
 \_\_rfloordiv\_\_() 메서드 73  
 right\_list 속성, dircmp 객체 388  
 right\_only 속성, dircmp 객체 388  
 rindex() 메서드, 문자열 50, 52  
 rjust() 메서드, 문자열 52  
 rlcompleter 모듈 725  
 rlecode\_hqx() 함수, binascii 모듈 677  
 rledecode\_hqx() 함수, binascii 모듈 677  
 RLock 객체  
     multiprocessing 모듈 526  
     threading 모듈 541  
 RLock() 메서드, Manager 객체 529  
 \_\_rlshift\_\_() 메서드 73  
 rmdir() 메서드, FTP 객체 614  
 rmdir() 함수, os 모듈 478  
 \_\_rmod\_\_() 메서드 73  
 rmtree() 함수, shutil 모듈 393  
 \_\_rmul\_\_() 메서드 73  
 robotparser 모듈 645  
 robots.txt 파일 645  
 rollback() 메서드, Connection 객체 366  
 rollover() 메서드, SpoolTemporaryFile 객체 399  
 \_\_ror\_\_() 메서드 74  
 rotate() 메서드, deque 객체 324  
 RotatingFileHandler 클래스, logging 모듈 446  
 round() 함수 78, 258  
 파이썬 3 258  
 rounding 속성, Context 객체 304  
 rounding, decimal 모듈 301  
 rounds 속성, sys.float\_info 285  
 row\_factory 속성, Connection 객체 378  
 rowcount 속성, Cursor 객체 368  
 rpartition() 메서드, 문자열 50  
 \_\_rpow\_\_() 메서드 73  
 \_\_rrshift\_\_() 메서드 73  
 rshift() 함수, operator 모듈 338  
 \_\_rshift\_\_() 메서드 73  
 rsplit() 메서드, 문자열 50, 52  
 rstrip() 메서드, 문자열 52  
 \_\_rsub\_\_() 메서드 73  
 \_\_rtruediv\_\_() 메서드 73  
 run 디버거 명령, pdb 모듈 233  
 run() 메서드  
     Process 객체 512  
     Thread 객체 537  
 run() 함수  
     cProfile 모듈 234  
     pdb 모듈 229  
     profile 모듈 234  
 runcall() 함수, pdb 모듈 229  
 runeval() 함수, pdb 모듈 229  
 RuntimeError 예외 105, 265  
 RuntimeWarning 경고 267, 294  
 \_\_rxor\_\_() 메서드 74

**S**


---

-S 명령줄 옵션 214  
 s(tep) 디버거 명령, pdb 모듈 233  
 safe\_substitute() 메서드, Template 객체 356  
 SafeConfigParser 클래스, configparser 모듈 413  
 same\_files 속성, dircmp 객체 388  
 samefile() 함수, os.path 모듈 490  
 sameopenfile() 함수, os.path 모듈 490  
 samestat() 함수, os.path 모듈 490

sample() 함수, random 모듈 313  
 SaveKey() 함수, winreg 모듈 505  
**SAX 인터페이스**  
 예 721  
 XML 파싱 703  
 sched 모듈 725  
 scheme 속성  
      urlparse 객체 641  
      urlsplit 객체 642  
 search() 메서드, Regex 객체 350  
 search() 함수, re 모듈 349  
 second 속성, time 객체 416  
 sections() 메서드, ConfigParser 객체 411  
 seed() 함수, random 모듈 312  
 seek() 메서드  
      파일 195~196, 433  
 IOBase 객체 430  
      mmap 객체 458  
 seekable() 메서드, IOBase 객체 431  
 select 모듈 510, 565  
      시그널 처리 492  
 select() 함수  
      성능 문제 575  
      asyncore 모듈 560  
      select 모듈 565  
 self 매개변수 메서드 24, 143  
      왜 필요한가 143  
 \_\_self\_\_ 속성  
      내장 함수 60  
      메서드 60  
 Semaphore 객체  
      multiprocessing 모듈 526  
      threading 모듈 541  
 Semaphore 객체, 시그널 위해 541  
 Semaphore() 메서드, Manager 객체 529  
 send() 메서드  
      생성기 22, 64, 124  
 Connection 객체 519  
 dispatcher 객체 562  
 HTTPConnection 객체 618  
 socket 객체 595  
 send\_bytes() 메서드, Connection 객체 519  
 send\_error() 메서드,  
      BaseHTTPRequestHandler 객체 626  
 send\_header() 메서드,  
      BaseHTTPRequestHandler 객체 627  
 send\_response() 메서드,  
      BaseHTTPRequestHandler 객체 627  
 send\_signal() 메서드, Popen 객체 496  
 sendall() 메서드, socket 객체 595  
 sendcmd() 메서드, FTP 객체 614  
 sendmail() 메서드, SMTP 객체 633  
 sendmsg() 시스템 호출, 없음 598  
 sendto() 메서드, socket 객체 595  
 sep 변수, os 모듈 475  
 sep 키워드 인수, print() 함수 199  
 Sequence 추상 기반 클래스 328  
 serve\_forever() 메서드  
      BaseManager 객체 531  
      SocketServer 객체 605  
 server 속성, BaseRequestHandler 객체 602  
 server\_address 속성, SocketServer 객체 605  
 server\_version 속성  
      BaseHTTPRequestHandler 클래스 625  
      HTTPRequestHandler 클래스 624  
 ServerProxy() 함수, xmlrpc.client 모듈 646  
 Set 추상 기반 클래스 329  
 set() 메서드  
      ConfigParser 객체 411  
      Element 객체 714  
      Event 객체 543  
      Morsel 객체 631  
 set() 함수 16, 92, 259  
 \_\_set\_\_() 메서드, 기술자 70, 154  
 set\_authorizer() 메서드, Connection 객체 377  
 set\_boundary() 메서드, Message 객체 687  
 set\_charset() 메서드, Message 객체 687  
 set\_conversion\_mode() 함수, ctypes 모듈 762  
 set\_debug() 함수, gc 모듈 272  
 set\_default\_type() 메서드, Message 객체 688  
 set\_defaults() 메서드, OptionParser 객체 192, 464  
 set\_errno() 함수, ctypes 모듈 762  
 set\_executable() 함수, multiprocessing 모듈 536  
 set\_last\_error() 함수, ctypes 모듈 762  
 set\_param() 메서드, Message 객체 688  
 set\_pasv() 메서드, FTP 객체 614  
 set\_payload() 메서드, Message 객체 688  
 set\_progress\_handler() 메서드, Connection 객체 378  
 set\_proxy() 메서드, Request 객체 637  
 set\_server\_documentation() 메서드, XMLRPCServer 객체 651  
 set\_server\_name() 메서드, XMLRPCServer 객체 651  
 set\_server\_title() 메서드, XMLRPCServer 객체 651  
 set\_terminator() 메서드, asynchat 객체 558  
 set\_threshold() 함수, gc 모듈 272  
 set\_trace() 함수, pdb 모듈 230  
 set\_type() 메서드, Message 객체 688  
 set\_unixfrom() 메서드, Message 객체 688  
 set\_usage() 메서드, OptionParser 객체 464  
 set\_wakeup\_fd() 함수, signal 모듈 492  
 setattr() 함수 259  
      개인 속성 156  
 \_\_setattr\_\_() 메서드 69, 160  
      \_\_slots\_\_ 162  
 setblocking() 메서드, socket 객체 595  
 setcheckinterval() 함수, sys 모듈 290  
 setcontext() 함수, decimal 모듈 305  
 setDaemon() 메서드, Thread 객체 538  
 setdefault() 메서드  
      사전 54  
      사전과 defaultdict 객체 325

setdefaultencoding() 메서드, sys 모듈  
219  
setdefaultencoding() 함수, sys 모듈 290  
setdefaulttimeout() 함수, socket 모듈 587  
setdlopenflags() 함수, sys 모듈 290  
setDocumentLocator() 메서드,  
ContentHandler 객체 720  
setegid() 함수, os 모듈 468  
seteuid() 함수, os 모듈 468  
setFormatter() 메서드, Handler 객체 449  
setgid() 함수, os 모듈 468  
setgroups() 함수, os 모듈 469  
setinputsizes() 메서드, Cursor 객체 368  
setitem() 함수, operator 모듈 338  
\_\_setitem\_\_() 메서드 70  
    분할 72  
setitimer() 함수, signal 모듈 492  
setLevel() 메서드  
    Handler 객체 448  
    Logger 객체 441  
setmode() 함수, msvcr 모듈 460  
setName() 메서드, Thread 객체 538  
setoutputsize() 메서드, Cursor 객체 368  
setpassword() 메서드, ZipFile 객체 402  
setpgid() 함수, os 모듈 469  
setpgrp() 함수, os 모듈 469  
setprofile() 함수  
    sys 모듈 290  
    threading 모듈 547  
setrecursionlimit() 함수, sys 모듈 290  
setregid() 함수, os 모듈 469  
setreuid() 함수, os 모듈 469  
\_setroot() 메서드, ElementTree 객체 710  
setsid() 함수, os 모듈 469  
setslice() 함수, operator 모듈 339  
setsockopt() 메서드, socket 객체 595  
setstate() 함수, random 모듈 312  
\_\_setstate\_\_() 메서드 282  
    복사 270  
    pickle 모듈 211  
@setter 장식자 프로퍼티 153  
settimeout() 메서드, socket 객체 596

settrace() 함수  
    sys 모듈 290  
    threading 모듈 548  
setuid() 함수, os 모듈 469  
setUp() 메서드  
    TestCase 객체 227, 227  
setup() 메서드, BaseRequestHandler 객체  
    603  
setup() 함수, distutils 모듈 187~188, 736  
setup.py 파일  
    사용자별 사이트 디렉터리에 설치 219  
    생성 187~188  
    C 확장 기능 736  
    install 명령어 188~189  
    setuptools 190  
    SWIG 확장 기능 764  
setup.py 파일 명령어 등록 190  
setuptools 라이브러리 180, 190  
SetValue() 함수, winreg 모듈 505  
SetValueEx() 함수, winreg 모듈 505  
sha1() 함수, hashlib 모듈 691  
sha224() 함수, hashlib 모듈 691  
sha256() 함수, hashlib 모듈 691  
sha384() 함수, hashlib 모듈 691  
sha512() 함수, hashlib 모듈 691  
Shelf 클래스, shelve 모듈 384  
shelve 모듈 210, 383  
    dbhash 모듈 383  
    pickle 프로토콜 선택 211  
shlex 모듈 726  
showwarning() 함수, warnings 모듈 295  
shuffle() 함수, random 모듈 313  
shutdown() 메서드  
    BaseManager 객체 531  
    socket 객체 596  
    SocketServer 객체 605  
shutdown() 함수, logging 모듈 451  
shutil 모듈 391  
SIG\* 시그널 이름 493  
SIGHUP 시그널 221  
siginterrupt() 함수, signal 모듈 492  
signal 모듈 491  
signal() 함수, signal 모듈 492  
SIGTERM 시그널 221  
simple\_producer() 함수, asynchat 모듈  
    558  
SimpleCookie() 함수, http.cookies 모듈  
    630  
SimpleHandler() 함수, wsgiref.handlers  
    모듈 670  
SimpleHTTPRequestHandler 클래스, http.  
server 모듈 624  
SimpleHTTPServer 모듈, http.server 참고  
622  
SimpleXMLRPCServer 모듈 650  
SimpleXMLRPCServer 클래스, xmlrpcl  
    server 모듈 650  
sin() 함수, math 모듈 310  
sinh() 함수, math 모듈 310  
site 모듈 203, 214, 219  
sitecustomize 모듈 219  
site-packages 디렉터리 215  
size 속성  
    Struct 객체 358  
    TarInfo 객체 397  
size() 메서드  
    FTP 객체 614  
    mmap 객체 458  
Sized 추상 기반 클래스 328  
sizeof() 함수, ctypes 모듈 762  
skippedEntity() 메서드, ContentHandler  
    객체 720  
sleep() 함수, time 모듈 500  
slice() 함수 65, 259  
\_\_slots\_\_ 속성  
    다른 코드와 호환성 162  
    상속 162  
    인스턴스의 \_\_dict\_\_ 속성 61  
    최적화 242  
    클래스 정의 162  
SMTP 프로토콜, 메시지 전달 예 633~634  
SMTP() 함수, smtplib 모듈 632  
smtpd 모듈 725  
SMTPHandler 클래스, logging 모듈 446

smtplib 모듈 632  
 sndhdr 모듈 726  
 sniff() 메서드, Sniffer 객체 680  
 Sniffer() 함수, csv 모듈 680  
 SO\_\* 소켓 옵션, socket 모듈 590  
 SOCK\_\* 상수, socket 모듈 579  
 socket 모듈 578  
 socket 속성, SocketServer 객체 605  
 socket() 함수, socket 모듈 587  
 socket\_type 속성, SocketServer 클래스 606  
 SocketHandler 클래스, logging 모듈 447  
 socketpair() 함수, socket 모듈 588  
 SocketServer 모듈 602  
 서버 매개변수 변경 606  
 파이썬 3 602  
 softspace 속성, 파일 197  
 sort() 메서드, 리스트 49  
 sorted() 함수 259  
 span() 메서드, MatchObject 객체 351  
 spawnl() 함수, os 모듈 483  
 spawnle() 함수, os 모듈 484  
 spawnlp() 함수, os 모듈 484  
 spawnlpe() 함수, os 모듈 484  
 spawnnv() 함수, os 모듈 483  
 spawnnve() 함수, os 모듈 483  
 spawnnvp() 함수, os 모듈 484  
 spawnnvpe() 함수, os 모듈 484  
 split() 메서드  
 문자열 15, 50, 53  
 Regex 객체 350  
 split() 함수  
 os.path 모듈 490  
 re 모듈 349  
 splitdrive() 함수, os.path 모듈 490  
 splitext() 함수, os.path 모듈 490  
 splitlines() 메서드, 문자열 53  
 splitunc() 함수, os.path 모듈 491  
 SpooledTemporaryFile() 함수, tempfile 모듈 399  
 sprintf() 함수 동일 84  
 spwd 모듈 725

SQL 질의  
 SQL 주입 공격 369  
 데이터베이스에 실행 365  
 리스트 내포와 유사점 134  
 만들기 369  
 예 379  
 SQLite 데이터베이스 373  
 sqlite3 모듈 373  
 sqrt() 메서드, Decimal 객체 301  
 sqrt() 함수, math 모듈 310  
 ssl 모듈 598  
 SSL, 인증서 생성 예 602  
 st\_\* 속성, stat 객체 478  
 stack() 함수, inspect 모듈 278  
 stack\_size() 함수, threading 모듈 548  
 Stackless Python 575  
 standard\_b64decode() 함수, base64 모듈 675  
 standard\_b64encode() 함수, base64 모듈 674  
 StandardError 예외 104  
 starmap() 함수, itertools 모듈 347  
 start 속성, 분할 65  
 start() 메서드  
 BaseManager 객체 531  
 MatchObject 객체 351  
 Process 객체 512  
 Thread 객체 537  
 Timer 객체 539  
 TreeBuilder 객체 715  
 startDocument() 메서드, ContentHandler 객체 720  
 startElement() 메서드, ContentHandler 객체 720  
 startElementNS() 메서드, ContentHandler 객체 720  
 startfile() 함수, os 모듈 484  
 startPrefixMapping() 메서드, ContentHandler 객체 721  
 startswith() 메서드, 문자열 53  
 stat 모듈 476, 725  
 stat() 함수

os 모듈 478  
 os.path 모듈 490  
 stat\_float\_times() 함수, os 모듈 479  
 @staticmethod 장식자 25, 58, 149, 153, 259  
 status 속성, HTTPResponse 객체 620  
 statvfs() 함수, os 모듈 479  
 stderr 변수, sys 모듈 197, 286  
 stderr 속성, Popen 객체 497  
 stdin 변수, sys 모듈 10, 197, 286  
 stdin 속성, Popen 객체 497  
 \_\_stderr\_\_ 변수, sys 모듈 198, 286  
 \_\_stdin\_\_ 변수, sys 모듈 198, 286  
 stdout 변수, sys 모듈 10, 197, 286  
 stdout 속성, Popen 객체 497  
 \_\_stdout\_\_ 변수, sys 모듈 198, 286  
 step 속성, 분할 65  
 StopIteration 예외 72, 104, 265  
 생성기 124  
 storbinary() 메서드, FTP 객체 614  
 storlines() 메서드, FTP 객체 614  
 str 타입 45  
 str() 함수 11, 67, 92, 259  
 print 198  
 repr()와 차이 12  
 \_\_str\_\_() 메서드 67  
 StreamError 예외, tarfile 모듈 397  
 StreamHandler 클래스, logging 모듈 447  
 StreamReader 클래스, codecs 모듈 342  
 streamreader() 메서드, CodecInfo 객체 342  
 StreamRequestHandler 클래스, SocketServer 모듈 603  
 StreamWriter 클래스, codecs 모듈 343  
 streamwriter() 메서드, CodecInfo 객체 342  
 strerror() 함수, os 모듈 469  
 strftime() 메서드  
 date 객체 415  
 time 객체 417  
 strftime() 함수, time 모듈 500  
 'strict' 여러 처리, 유니코드 인코딩 203

string 모듈 353  
     Template 문자열 200  
     string\_at() 함수, ctypes 모듈 762  
     StringIO 클래스, io 모듈 435  
     stringprep 모듈 724  
     strip() 메서드, 문자열 53  
     strftime() 메서드, datetime 클래스 418  
     strptime() 함수, time 모듈 423, 501  
     struct 모듈 357  
     Struct 클래스, struct 모듈 358  
     Structure 클래스, ctypes 모듈 758  
     sub() 메서드, Regex 객체 350  
     sub() 함수  
         operator 모듈 338  
         re 모듈 349  
         \_\_sub\_\_() 메서드 73  
         \_\_subclasscheck\_\_() 메서드 69, 166  
     subdirs 속성, difflib 객체 389  
     SubElement() 함수, xml.etree.  
         ElementTree 모듈 712  
     subn() 메서드, Regex 객체 350  
     subn() 함수, re 모듈 349  
     subprocess 모듈 495  
     substitute() 메서드  
         Template 객체 356  
         Template 문자열 200  
     successful() 메서드,AsyncResult 객체 523  
     sum() 함수 47~48, 80, 259  
         숫자 데이터에만 사용 47  
     정확도 310  
     decimal 모듈 82  
     math.fsum() 함수와 차이점 310  
     sunau 모듈 726  
     super() 함수 146, 259~260  
         파이썬 3 259~260, 775  
     supports\_unicode\_filenames 변수,  
         os.path 모듈 491  
     swapcase() 메서드, 문자열 53  
     SWIG 730  
         예 764~765  
         인터페이스 파일 764  
     switch문, 없음 8

symbol 모듈 724  
     symlink() 함수, os 모듈 479  
     symmetric\_difference() 메서드, 집합 56  
     symmetric\_difference\_update() 메서드, 집  
         합 56  
     sync() 메서드  
         dbm 스타일 데이터베이스 객체 382  
         shelve 객체 383  
     SyntaxError 예외 105, 265  
         기본 인수 112  
         파이썬 3 print문 4  
         except문 102  
     SyntaxWarning 경고 267, 294  
     sys 모듈 13, 283  
         sys.argv 변수 13, 191, 214  
         sys.displayhook 변수 217  
         sys.exec\_prefix 변수 218  
         sys.exit() 함수 221  
         sys.modules 변수 177, 182  
         sys.path 변수 180  
             써드파티 모듈 190  
             site 모듈 218  
         sys.prefix 변수 218  
         sys.ps1 변수 217  
         sys.ps2 변수 217  
         sys.stderr 변수 191, 197  
         sys.stdin 변수 197  
         sys.stdout 변수 197  
         sys\_version 속성,  
             BaseHTTPRequestHandler 클래스 625  
     sysconf() 함수, os 모듈 487  
     syslog 모듈 725  
     SysLogHandler 클래스, logging 모듈 447  
     system() 함수, os 모듈 484  
     system.listMethods() 메서드, ServerProxy  
         객체 647  
     SystemError 예외 105, 265  
     SystemExit 예외 5, 104~106, 191, 221,  
         265

T  
     -t 명령줄 옵션 30, 214  
     tabcheck 속성, sys.flags 284  
     TabError 예외 30, 105, 265  
     tabnanny 모듈 724  
     tag 속성, Element 객체 713  
     tagName 속성, DOM Element 객체 707  
     tail 명령, 생성기 예 21  
     tail 속성, Element 객체 713  
     takewhile() 함수, itertools 모듈 347  
     tan() 함수, math 모듈 310  
     tanh() 함수, math 모듈 310  
     TarError 예외, tarfile 모듈 397  
     TarFile 객체, tarfile 모듈 394  
     tarfile 모듈 393  
     TarInfo 객체, tarfile 모듈 394  
     task\_done() 메서드  
         JoinableQueue 객체 515  
         Queue 객체 550  
     tb\_\* 속성, 역추적 객체 64  
     tb\_lineno() 함수, traceback 모듈 292  
     tbbreak 디버거 명령, pdb 모듈 233  
     tcgetpgrp() 함수, os 모듈 474  
     TCP 연결, 다이어그램 554  
     TCP 프로토콜 553  
         코드 예 555  
     TCP\_\* 소켓 옵션, socket 모듈 593  
     TCPServer 클래스, SocketServer 모듈 604  
     tcsetpgrp() 함수, os 모듈 474  
     tearDown() 메서드  
         TestCase 객체 227, 227  
     tee() 함수, itertools 모듈 337  
     tell() 메서드  
         IOBase 객체 430  
         mmap 객체 458  
         파일 195~196  
     telnetlib 모듈 725  
     tempdir 변수, tempfile 모듈 400  
     tempfile 모듈 398  
     Template 문자열  
         CGI 스크립트에서 사용 663

string 모듈 200  
 template 변수, tempfile 모듈 400  
 template 속성, Template 객체 356  
 Template 클래스, string 모듈 356  
 TemporaryFile() 함수, tempfile 모듈 399  
 terminate() 메서드  
   Pool 객체 523  
   Popen 객체 497  
   Process 객체 512  
 termios 모듈 725  
 test 모듈 724  
 test() 함수, cgi 모듈 662  
 TestCase 클래스, unittest 모듈 227  
 testmod() 함수  
   doctest 모듈 224  
   doctest() 모듈 224  
 testzip() 메서드, ZipFile 객체 402  
 text 속성, Element 객체 713  
 Text 클래스, xml.dom.minidom 모듈 708  
 text\_factory 속성, Connection 객체 379  
 TextIOBase 추상 기반 클래스 436  
 TextIOWrapper 클래스, io 모듈 434  
 textwrap 모듈 724  
 this 포인터, 메서드의 self 매개변수 143  
 thread 속성, Record 객체 442  
 Thread 클래스, threading 모듈 537  
 threading 모듈 537  
   동기화 기본 기능 540  
 ThreadingMixIn 클래스, SocketServer 모듈 608  
 ThreadingTCPServer 클래스, SocketServer 모듈 608  
 ThreadingUDPServer 클래스,  
   SocketServer 모듈 608  
 threadName 속성, Record 객체 442  
 threadsafety 변수, 데이터베이스 API 372  
 throw() 메서드, 생성기 64, 126~127  
 time 모듈 236, 498  
   시간 함수 정확도 501  
   현재 시간 499  
 time 클래스, datetime 모듈 416  
 time() 메서드, datetime 객체 419  
 time() 함수, time 모듈 236, 501  
 Time() 함수, 데이터베이스 API 371  
 timedelta 클래스, datetime 모듈 419  
 TimedRotatingFileHandler 클래스,  
   logging 모듈 447  
 TimeFromTicks() 함수, 데이터베이스 API 371  
 timeit 모듈 236  
 timeit() 함수  
   cProfile 모듈 236  
   timeit 모듈 236  
 timeout 속성, SocketServer 클래스 606  
 timeout 예외, socket 모듈 597  
 Timer() 함수, threading 모듈 539  
 times() 함수, os 모듈 484  
 Timestamp() 함수, 데이터베이스 API 371  
 TimestampFromTicks() 함수, 데이터베이스 API 371  
 timetuple() 메서드, date 객체 415  
 timetz() 메서드, datetime 객체 419  
 timezone 변수, time 모듈 499  
 TIPC 프로토콜 578  
   주소 형식 582  
 TIPC\_\* 상수, socket 모듈 582  
 title() 메서드, 문자열 53  
 Tkinter 모듈 726  
 today() 메서드, date 클래스 414  
 tofile() 메서드, 배열 객체 321  
 token 모듈 724  
 tokenize 모듈 724  
 tolist() 메서드, 배열 객체 321  
 toordinal() 메서드, date 객체 416  
 toprettyxml() 메서드, DOM Node 객체 708  
 tostring() 메서드, 배열 객체 321  
 tostring() 함수, xml.etree.ElementTree 모듈 715  
 total\_changes 속성, Connection 객체 379  
 tounicode() 메서드, 배열 객체 321  
 toxml() 메서드, DOM Node 객체 709  
 trace() 함수, inspect 모듈 278  
 traceback 모듈 290  
 \_\_traceback\_\_ 속성, Exception 객체 263  
 tracebacklimit 변수, sys 모듈 286  
 TracebackType 타입 62, 292  
 transfercmd() 메서드, FTP 객체 614  
 translate() 메서드, 문자열 50, 53  
 traps 속성, Context 객체 304  
 TreeBuilder() 함수, xml.etree.ElementTree 모듈 714  
 triangular() 함수, random 모듈 314  
 True 값 9, 31, 46  
 truediv() 함수, operator 모듈 338  
 \_\_truediv\_\_() 메서드 73  
 trunc() 함수, math 모듈 310  
 truncate() 메서드  
   파일 195  
 IOBase 객체 430  
 truth() 함수, operator 모듈 338  
 try문 25, 102  
 -tt 명령줄 옵션 30, 214  
 tty 모듈 725  
 ttyname() 함수, os 모듈 474  
 tuple 타입 45  
 tuple() 함수 92, 260  
 Twisted 라이브러리 510, 575  
 type 속성  
   FieldStorage 객체 660  
   socket 객체 596  
   TarInfo 객체 397  
 type() 메타클래스 169  
 type() 함수 40, 260  
   예외 107  
 type\_options 속성, FieldStorage 객체 660  
 typecode 속성, 배열 객체 320  
 TypeError 예외 105, 266  
   강제 타입 변환 75  
   메서드 분석 순서 147  
   함수 호출 113  
 types 모듈 57, 292  
   파이썬 3 293  
 tzinfo 속성, time 객체 416  
 tzname 변수, time 모듈 499  
 tzname() 메서드

time 객체 417  
tzinfo 객체 422  
tzset() 함수, time 모듈 501

**U**

-U 명령줄 옵션 33, 214  
'U' 모드, open() 함수 194  
u 문자, 문자열 상수 앞 33  
u(p) 디버거 명령, pdb 모듈 233  
UDP 서버 예 597  
UDP 클라이언트 예 597  
UDP 통신, 다이어그램 555  
UDP 프로토콜 553  
UDPServer 클래스, SocketServer 모듈 604  
uid 속성, TarInfo 객체 397  
umask() 함수, os 모듈 469  
unalias 디버거 명령, pdb 모듈 233  
uname 속성, TarInfo 객체 397  
uname() 함수, os 모듈 469  
UnboundLocalError 예외 105, 117, 266  
unconsumed\_tail 속성, decompressobj 객체 406  
unescape() 함수, xml.sax.saxutils 모듈 722  
ungetch() 함수, msvcrt 모듈 460  
ungetwch() 함수, msvcrt 모듈 460  
unhexlify() 함수, binascii 모듈 677  
unichr() 함수 93, 260  
unicode 속성, sys.flags 284  
unicode() 함수 261  
파이썬 3 261  
unicodedata 모듈 209, 360  
UnicodeDecodeError 예외 105, 266  
UnicodeEncodeError 예외 105, 266  
파이썬 3 대화식 모드 216  
UnicodeError 예외 105, 203, 266  
unicode-escape 인코딩, 설명 209  
UnicodeTranslateError 예외 105, 266  
unidata\_version 변수, unicodedata 모듈 364

uniform() 함수, random 모듈 314  
UnimplementedFileMode 예외, http.client 모듈 620  
Union 클래스, ctypes 모듈 758  
union() 메서드, 집합 56  
unittest 모듈 226  
예 227  
UnixDatagramServer 클래스, SocketServer 모듈 604  
UnixStreamServer 클래스, SocketServer 모듈 604  
UnknownHandler 클래스, urllib.request 모듈 638  
UnknownProtocol 예외, http.client 모듈 620  
UnknownTransferEncoding 예외, http.client 모듈 620  
unlink() 함수, os 모듈 480  
unpack() 메서드, Struct 객체 358  
unpack() 함수, struct 모듈 357  
unpack\_from() 메서드, Struct 객체 358  
unpack\_from() 함수, struct 모듈 358  
Unpickler 클래스, pickle 모듈 281  
unquote() 함수, urllib.parse 모듈 643  
unquote\_plus() 함수, urllib.parse 모듈 643  
unquote\_to\_bytes() 함수, urllib.parse 모듈 644  
unregister() 메서드, Poll 객체 566  
unregister\_dislect() 함수, csv 모듈 681  
unsetenv() 함수, os 모듈 470  
until 디버거 명령, pdb 모듈 233  
unused\_data 속성, decompressobj 객체 406  
unwrap() 메서드, ssl 객체 600  
update() 메서드  
    사전 54  
    집합 16, 56  
    digest 객체 691  
    hmac 객체 692  
update\_wrapper() 함수, functools 모듈 332  
upper() 메서드, 문자열 53  
uppercase 변수, string 모듈 353  
urandom() 함수, os 모듈 487  
URL 추출  
    예 633~634  
    인증 예 639  
    쿠키 예 640  
urldefrag() 함수, urllib.parse 모듈 642  
urlencode() 함수, urllib.parse 모듈 644  
URLError 예외 635  
    urllib.error 모듈 645  
urljoin() 함수, urllib.parse 모듈 642  
urllib 모듈 643  
    urllib.request 참고 634  
urllib 패키지 634  
urllib.error 모듈 645  
urllib.parse 모듈 640  
urllib.request 모듈 634  
urllib.response 모듈 640  
urllib.robotparser 모듈 645  
urllib2 모듈, urllib.request 참고 634  
urlopen() 함수, urllib.request 모듈 634  
urlparse 모듈 641  
urlparse() 함수, urllib.parse 모듈 641  
urlsafe\_b64decode() 함수, base64 모듈 675  
urlsafe\_b64encode() 함수, base64 모듈 675  
urlsplit() 함수, urllib.parse 모듈 641  
urlunparse() 함수, urllib.parse 모듈 641  
urlunsplit() 함수, urllib.parse 모듈 642  
user 모듈 724  
—user 옵션 setup.py 190  
username 속성  
    urlparse 객체 641  
    urlsplit 객체 642  
UserWarning 경고 267, 294  
utcfromtimestamp() 메서드, datetime 클래스 418  
utcnow() 메서드, datetime 클래스 418  
utcoffset() 메서드  
    time 객체 417

tzinfo 객체 422  
 utctimetuple() 메서드, datetime 객체 419  
 UTF-16 인코딩, 설명 208  
 UTF-8  
     문자열 상수에 포함 34  
     사전 순서 208  
     설명 207~208  
     인코딩과 디코딩 51  
     ASCII와 호환 208  
 utime() 함수, os 모듈 480  
 uu 모듈 726

**V**

-V 명령줄 옵션 214  
 validator() 함수, wsgiref.handlers 모듈 670  
 value 속성  
     FieldStorage 객체 660  
     Morsel 객체 631  
 Value() 메서드, Manager 객체 529  
 Value() 함수, multiprocessing 모듈 525  
 ValueError 예외 105, 266  
     리스트 48  
     문자열 50  
 valuerefs() 메서드, WeakValueDictionary 객체 297  
 values() 메서드  
     사전 54  
     파이썬 3에서 사전 782  
     Message 객체 682  
 ValuesView 추상 기반 클래스 329  
 vars() 함수 86, 261  
 verbose 속성, sys.flags 284  
 verify\_request() 메서드, SocketServer 클래스 607  
     version 변수, sys 모듈 286  
 version 속성, HTTPResponse 객체 620  
 version\_info 변수, sys 모듈 286  
 vformat() 메서드, Formatter 객체 354  
 volume 속성, ZipInfo 객체 404  
 vonmisesvariate() 함수, random 모듈 314

**W**

-W 명령줄 옵션 267, 294~295  
 'w' 모드, open() 함수 193  
 w(here) 디버거 명령, pdb 모듈 233  
 wait() 메서드  
     AsyncResult 객체 523  
     Event 객체 543  
     Popen 객체 497  
     락 544  
 wait() 함수, os 모듈 485  
 wait3() 함수, os 모듈 485  
 wait4() 함수, os 모듈 485  
 waitpid() 함수, os 모듈 485  
 walk() 메서드, Message 객체 685  
 walk() 함수, os 모듈 480  
 warn() 함수, warnings 모듈 267, 294  
 warn\_explicit() 함수, warnings 모듈 294  
 Warning 경고 267, 294  
 warning() 메서드, Logger 객체 439  
 warnings 모듈 293  
 warnoptions 변수, sys 모듈 287  
 WatchedFileHandler 클래스, logging 모듈 447  
 wave 모듈 726  
 WCOREDUMP() 함수, os 모듈 485  
 WeakKeyDictionary 클래스, weakref 모듈 297  
 weakref 모듈 159, 296  
 WeakValueDictionary 클래스, weakref 모듈 298  
 webbrowser 모듈 671  
 weekday() 메서드, date 객체 416  
 weibullvariate() 함수, random 모듈 315  
 WEXITSTATUS() 함수, os 모듈 486  
 wfile 속성  
     BaseHTTPRequestHandler 객체 626  
     StreamRequestHandler 객체 603  
 whichdb 모듈 382  
 whichdb() 함수, dbm 모듈 382  
 while문 6, 98  
 whitespace 변수, string 모듈 353

WIFCONTINUED() 함수, os 모듈 486  
 WIFEXITED() 함수, os 모듈 486  
 WIFSIGNALLED() 함수, os 모듈 486  
 WIFSTOPPED() 함수, os 모듈 486  
 WinDLL() 함수, ctypes 모듈 755  
 WindowsError 예외 266  
 Wing IDE 3  
 winreg 모듈 502  
 winsound 모듈 726  
 winver 변수, sys 모듈 287  
 with문 75, 107  
     락 기본 연산 546  
     예외 26  
     접급 107  
     decimal 모듈 305  
 wrap\_socket() 함수, ssl 모듈 598  
 @wraps 장식자, functools 모듈 137, 332  
 writable() 메서드  
     dispatcher 객체 561  
     IOBase 객체 430  
 write() 메서드  
     파일 10, 195~196  
     BufferWriter 객체 433  
     ConfigParser 객체 411  
     ElementTree 객체 711  
     FileIO 객체 431  
     mmap 객체 458  
     ssl 객체 600  
     StreamWriter 객체 343  
     TextIOWrapper 객체 435  
     ZipFile 객체 402  
 write() 함수, os 모듈 475  
 write\_byte() 메서드, mmap 객체 458  
 writelines() 메서드  
     파일 195~196  
     IOBase 객체 430  
     StreamWriter 객체 343  
 writeipy() 메서드, ZipFile 객체 403  
 writer() 함수, csv 모듈 678  
 writerow() 메서드  
     csv DictWriter 객체 680  
     csv writer 객체 679

writerows() 메서드  
  csv DictWriter 객체 680  
  csv writer 객체 679  
writestr() 메서드, ZipFile 객체 403  
writexml() 메서드, DOM Node 객체 709  
WSGI 666  
  독립 서버로 실행 668  
  예 667  
  웹 프레임워크와 통합 670  
  응용 프로그램 검증 670  
  응용 프로그램 명세서 666  
  폼 필드 처리 667  
  CGI 스크립트에서 실행 668  
  I/O에 생성기 사용 202  
wsgi.\* 환경 변수 667  
wsgiref 패키지 668  
wsgiref.handlers 모듈 669  
wsgiref.simple\_server 모듈 668  
WSTOPSIG() 함수, os 모듈 486  
wstring\_at() 함수, ctypes 모듈 762  
WTERMSIG() 함수, os 모듈 486

## X

---

-x 명령줄 옵션 214  
xdrlib 모듈 726  
XML  
  문서 예 704  
  문자 탈출과 텔출 되돌림 722  
  큰 파일 점진적인 파싱 717~718

파싱 703  
  ElementTree 모듈에서 네임스페이스  
  717  
xml 패키지 703  
XML() 함수, xml.etree.ElementTree 모듈  
  712  
xml.dom.minidom 모듈 705  
xml.etree.ElementTree 모듈 709  
xml.sax 모듈 718  
xml.sax.saxutils 모듈 722  
XMLGenerator() 함수, xml.sax.saxutils 모  
  듈 722  
XMLID() 함수, xml.etree.ElementTree 모  
  듈 712  
XML-RPC 646  
  서버 커스터마이즈 653~654  
  예 652~653  
XML-RPC 서버, 멀티스레드 예 608  
'xmlcharrefreplace' 에러 처리 203  
xmlrpc 패키지 646  
xmlrpc.client 모듈 646  
xmlrpc.server 모듈 650  
xmlrpclib 모듈 646

xrange() 함수 18, 53, 261  
파이썬 3 18, 53, 261  
XSLT 704

---

Y

Y2K 처리 501  
year 속성, date 객체 415  
yield문 20, 64, 123  
컨택스트 관리자 109  
I/O와 사용 201~202  
yield 표현식 22, 125

## Z

---

ZeroDivisionError 예외 105, 266  
zfill() 메서드, 문자열 53  
.zip 파일  
  디코딩과 인코딩 400  
모듈 180  
코드 아카이브로 사용 180  
zip() 함수 100, 261  
타입 변환 예 45  
파이썬 3 261  
future\_builtins 모듈 268  
zipfile 모듈 400  
ZipFile() 함수, zipfile 모듈 400  
zipimport 모듈 724  
ZipInfo() 함수, zipfile 모듈 401