

점프 투 파이썬

박웅용

2018년 12월 11일

차 례

※ 문서정보	2
00장 들어가기 전에	3
00-1 머리말	4
00-2 저자소개	5
00-3 주요변경이력	6
00-4 책구입안내	8
01장 파이썬이란 무엇인가?	10
01-1 파이썬이란?	11
01-2 파이썬의 특징	12
파이썬은 인간다운 언어이다	12
파이썬은 문법이 쉬워 빠르게 배울 수 있다	12
파이썬은 무료이지만 강력하다	13
파이썬은 간결하다	13
파이썬은 프로그래밍을 즐기게 해준다	14
파이썬은 개발 속도가 빠르다	14
01-3 파이썬으로 무엇을 할 수 있을까?	15
파이썬으로 할 수 있는 일	15
파이썬으로 할 수 없는 일	17
01-4 파이썬 설치하기	18
윈도우에서 파이썬 설치하기	18
리눅스에서 파이썬 설치	20
01-5 파이썬 둘러보기	21
파이썬 기초 실습 준비하기	21
파이썬 기초 문법 따라 해보기	22
01-6 파이썬과 에디터	26
IDLE	26
명령 프롬프트 창에서 파이썬 프로그램 실행하기	32
여러가지 에디터	32
02장 파이썬 프로그래밍의 기초, 자료형	37
02-1 숫자형	38

숫자형이란?	38
숫자형은 어떻게 만들고 사용할까?	38
숫자형을 활용하기 위한 연산자	39
연습문제	41
02-2 문자열 자료형	43
문자열이란?	43
문자열은 어떻게 만들고 사용할까?	43
문자열 연산하기	47
문자열 인덱싱과 슬라이싱	48
문자열 포매팅	54
문자열 관련 함수들	63
연습문제	67
02-3 리스트 자료형	69
리스트는 어떻게 만들고 사용할까?	69
리스트의 인덱싱과 슬라이싱	69
리스트 연산자	73
리스트의 수정, 변경과 삭제	75
리스트 관련 함수들	76
연습문제	82
02-4 튜플 자료형	84
튜플은 어떻게 만들까?	84
튜플의 인덱싱과 슬라이싱, 더하기(+)와 곱하기(*)	85
연습문제	86
02-5 덕셔너리 자료형	88
덕셔너리란?	88
덕셔너리는 어떻게 만들까?	88
덕셔너리 쌍 추가, 삭제하기	89
덕셔너리를 사용하는 방법	90
덕셔너리 관련 함수들	93
연습문제	96
02-6 집합 자료형	98
집합 자료형은 어떻게 만들까?	98
집합 자료형의 특징	98
집합 자료형 활용하는 방법	99
집합 자료형 관련 함수들	101
연습문제	102
02-7 불 자료형	104
불 자료형이란?	104

자료형의 참과 거짓	105
불연산	107
연습문제	108
02-8 자료형의 값을 저장하는 공간, 변수	109
변수는 어떻게 만들까?	109
변수란?	109
리스트를 변수에 넣고 복사하고자 할 때	110
변수를 만드는 여러 가지 방법	112
연습문제	113
03장 프로그램의 구조를 써는다! 제어문	116
03-1 if문	117
if문은 왜 필요할까?	117
if문의 기본 구조	117
조건문이란 무엇인가?	120
다양한 조건을 판단하는 elif	124
조건부 표현식	127
연습문제	127
03-2 while문	129
while문의 기본 구조	129
while문 직접 만들기	130
while문 강제로 빠져나가기	132
while문의 맨 처음으로 돌아가기	134
무한 루프	135
연습문제	136
03-3 for문	138
for문의 기본 구조	138
for문과 continue	140
for와 함께 자주 사용하는 range함수	141
리스트 안에 for문 포함하기	143
연습문제	144
04장 프로그램의 입력과 출력은 어떻게 해야 할까?	146
04-1 함수	147
함수란 무엇인가?	147
함수를 사용하는 이유는 무엇일까?	147
파이썬 함수의 구조	148
입력값과 결과값에 따른 함수의 형태	149
매개변수 지정하여 호출하기	152
입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?	153

함수의 결과값은 언제나 하나이다	157
매개변수에 초기값 미리 설정하기	159
함수 안에서 선언된 변수의 효력 범위	162
lambda	164
연습문제	165
04-2 사용자 입력과 출력	167
사용자 입력	167
print 자세히 알기	168
연습문제	169
04-3 파일 읽고 쓰기	171
파일 생성하기	171
파일을 쓰기 모드로 열어 출력값 적기	172
프로그램의 외부에 저장된 파일을 읽는 여러 가지 방법	174
파일에 새로운 내용 추가하기	176
with문과 함께 사용하기	177
연습문제	179
05장 파일 날개달기	182
05-1 클래스	183
클래스는 도대체 왜 필요한가?	183
클래스와 객체	187
사칙연산 클래스 만들기	188
생성자 (Constructor)	198
클래스의 상속	201
클래스 변수	204
클래스의 활용	206
연습문제	209
05-2 모듈	213
모듈 만들고 불러 보기	213
if __name__ == “__main__”: 의 의미	216
클래스나 변수 등을 포함한 모듈	218
새 파일 안에서 이전에 만든 모듈 불러오기	220
연습문제	222
05-3 패키지	224
패키지란 무엇인가?	224
패키지 만들기	224
__init__.py 의 용도	227
__all__의 용도	227
relative 패키지	228

연습문제	229
05-4 예외 처리	232
오류는 어떤 때 발생하는가?	232
오류 예외 처리 기법	233
오류 회피하기	237
오류 일부러 발생시키기	237
예외 만들기	239
연습문제	241
05-5 내장 함수	244
abs	244
all	244
any	245
chr	245
dir	246
divmod	246
enumerate	246
eval	247
filter	247
hex	248
id	249
input	249
int	250
isinstance	251
len	251
list	251
map	252
max	253
min	254
oct	254
open	254
ord	255
pow	255
range	256
round	256
sorted	257
str	258
tuple	258
type	259

zip	259
연습문제	259
05-6 외장 함수	262
sys	262
pickle	264
os	264
shutil	266
glob	266
tempfile	267
time	267
calendar	270
random	272
webbrowser	273
namedtuple	274
defaultdict	276
연습문제	281
06장 파일 프로그래밍, 어떻게 시작해야 할까?	284
06-1 내가 프로그램을 만들 수 있을까?	285
06-2 3과 5의 배수 합하기	289
06-3 게시판 페이징하기	292
06-4 간단한 메모장 만들기	294
06-5 템플릿을 4개의 공백으로 바꾸기	297
06-6 하위 디렉터리 검색하기	300
06-7 코딩도장	304
07장 유용한 파일 도구들	305
07-1 정규 표현식 살펴보기	306
정규 표현식은 왜 필요한가?	306
07-2 정규 표현식 시작하기	309
정규 표현식의 기초, 메타 문자	309
match 객체의 메서드	317
컴파일 옵션	319
백슬래시 문제	323
07-3 강력한 정규 표현식의 세계로	325
메타문자	325
그룹핑	328
전방 탐색	331
문자열 바꾸기	334
Greedy vs Non-Greedy	335

07-4 파일로 XML 처리하기	337
XML 문서 생성하기	337
XML문서 파싱하기	342
07-5 파일로 Json 처리하기	345
json 이란?	345
json dumps, loads	345
json file	346
json 송수신	347
07-6 소트	349
평범한 소트	349
key를 이용하여 소트하기	350
operator 모듈	352
순차 정렬과 역순 정렬	353
중첩 소트	354
08장 종합문제	355
09장 마치며	361
A. 풀이	362
A-1 연습문제 풀이	363
02-1 숫자형	363
02-2 문자열 자료형	364
02-3 리스트 자료형	367
02-4 튜플 자료형	369
02-5 딕셔너리 자료형	371
02-6 집합 자료형	373
02-7 불 자료형	375
02-8 자료형의 값을 저장하는 공간, 변수	375
03-1 if문	379
03-2 while문	381
03-3 for문	383
04-1 함수	385
04-2 사용자 입력과 출력	388
04-3 파일 읽고 쓰기	389
05-1 클래스	392
05-2 모듈	394
05-3 패키지	397
05-4 예외처리	397
05-5 내장 함수	399
05-6 외장 함수	400

A-2 종합문제 풀이	403
B. 부록	412
B-1 파이썬 2.7 vs 파이썬 3	413
print	413
자동 형 변환	414
input	414
<u>소스코드 인코딩</u>	414
에러처리	415
B-2 str 과 repr	416
str과 repr의 차이점	416
클래스의 <u>__str__</u> 과 <u>__repr__</u>	418
B-3 유니코드와 인코딩	421
유니코드의 유래	421
유니코드 vs utf-8	421
파이썬과 유니코드	422
입출력과 인코딩	424
<u>소스코드 인코딩</u>	425
B-4 파이썬 패키지 설치 (pip)	426
pip 이란?	426
pip 수동 설치	426
pip 사용	426
pip 따라해 보기	428
pip을 이용한 개발환경 구축하기	429

※ 문서정보

Copyright © 2015 박응용. All rights reserved.

이 책의 무단전재와 복제를 금합니다

이 책은 김준석(wnstjr8717@naver.com) 님이 구매하신 문서입니다.

(구매 : <http://wikidocs.net/book/1>)

00장 들어가기 전에

점프 투 파이썬이 세상에 나온지 벌써 15년이 지났다.

책을 처음 집필하던 15년 전의 초보 시절의 마음과 지나온 15년 프로그래밍 경험을 잘 조합하면 더 좋은 책을 만들 수 있을거라 생각한다. 시간이 날 때마다 틈틈이 보강해야 할 부분은 보강하고 삭제해야 할 부분은 삭제하며 점프 투 파이썬을 꾸준히 다듬어 나가고 있는 중이다.

점프 투 파이썬은 다음과 같은 모토로 온라인 상에서 계속 진화해 나갈 예정이다.

어제의 점프 투 파이썬과 오늘의 점프 투 파이썬은 다르다!

어제와 다른 오늘 -박응용

00-1 머리말

프로그래밍을 공부하고자 하는 사람들이 배울 수 있는 프로그래밍 언어는 상당히 많다. 하지만 프로그래밍을 위해 처음으로 배우게 될 언어를 선택할 때는 약간의 주의가 필요하다. 처음부터 너무 어려운 언어를 선택하거나 특정 기술에 특화된 언어를 선택할 경우 자칫 잘못하면 “우물 안 개구리”가 될 수도 있기 때문이다.

파이썬은 처음 배우기 좋은 언어로 많이들 추천하는 언어이다. 파이썬은 초보 프로그래머가 어려운 문법에 허우적거리게 만들지 않고 프로그래밍의 핵심적인 개념을 정말 쉽게 배울 수 있게 만드는 훌륭한 언어이다. 파이썬을 통해서 좋은 프로그래밍 스타일을 한번 배워두면 다른 언어를 습득하는 것이 무척 쉬울 뿐만 아니라 다른 언어로 만든 프로그램마저도 고급스러워지는 효과를 발휘하곤 한다.

아마도 이 책을 들고 있는 여러분은 이미 파이썬의 명성에 대해서 들어본 적이 있을 것이다. 파이썬은 이제 C, C++, Java 등과 어깨를 나란히 할 정도로 유명한 주류 언어가 되었다. 하지만 파이썬이 자바보다 더 오래된 언어라는 것을 아는 사람은 드물다. 파이썬은 혜성처럼 갑자기 등장해서 유명해진 스타 언어가 아닌 그 역사가 매우 오래되어 숙성된 언어이다.

그리고 여러분이 지금 보고 있는 이 “점프 투 파이썬” 역시 그 역사가 오래된 책이다. 이 책의 탄생은 2001년이니 무려 15살이 넘었다. 파이썬이 버전 업을 하며 진화해 가는 동안 “점프 투 파이썬” 역시 “위키독스(wikidocs.net)”라는 온라인 사이트에서 파이썬의 변화와 독자들의 요구에 발 맞추어 계속된 진화를 거듭해 왔다.

사실 “점프 투 파이썬”은 필자 한 사람에 의해서 쓰여진 책이 아니다. 그 이유는 위키독스의 “점프 투 파이썬”에 달린 무수한 댓글들을 보면 알 수 있다. 작게는 오타 수정부터 크게는 내용의 편집까지 오랜 시간 동안 독자 여러분들의 손길이 미치지 않은 곳이 없다.

이 책이 나올 수 있도록 도와주신 많은 분들께 감사의 말씀을 전하고 싶다.

책이 출간되더라도 “점프 투 파이썬”이 위키독스에 그대로 공개될 수 있도록 도움을 주신 이지스 퍼블리싱의 이지연 대표님께 위키독스의 독자를 대표하여 감사의 말씀을 전하고 싶다. 또한 책의 내용을 초보자의 입장에서 이해하기 쉽게 만들어 준 이지스 퍼블리싱의 홍연의씨에게 감사의 마음을 전하고 싶다.

이번 파이썬 원고를 면밀히 검토하면서 자기도 모르게 파이썬 고수가 되어버린 나의 아내 김선정, 그리고 아빠가 책을 쓸때면 조용히 옆에서 책을 읽으며 배려해 주었던 아들 박민규에게 고마운 마음을 전하고 싶다.

마지막으로 오랜 시간동안 이 책을 검토하고 읽어주신 “점프 투 파이썬”的 독자 여러분들 모두에게 무한한 감사의 말씀을 전하고 싶다.

2016년 박응용.

00-2 저자소개

박웅용 (pahkey@gmail.com)

국내 저자로는 최초로 파이썬 안내서인 “점프 투 파이썬”을 2001년에 출간하였다.

위키독스(<https://wikidocs.net>)라는 온라인 서비스를 제작하여 프로그래밍 및 IT관련 지식을 공유하는 데 힘쓰고 있으며 프로그래밍 문제풀이를 통해 코딩실력과 알고리즘을 수련하는 코딩도장(<http://codingdojang.com>) 서비스를 운영하고 있다.

00-3 주요변경이력

점프 투 파이썬의 주요 변경 이력

순서	주요 내용	날짜
1	점프 투 파이썬 오프라인 책 출간 (정보게이트)	2001.09
2	개인 위키에 “점프 투 파이썬” 공개	2006.03
3	위키독스에 “점프 투 파이썬” 무료 온라인 책 출간	2008.03
4	파이썬 버전 3으로 개정	2013.05
5	전자책(E-book) 판매 시작	2013.10
6	파이썬 버전 통합 (Python 2.7 + Python 3)	2014.02
7	Sets 챕터 추가	2014.06
8	패키지(Packages) 챕터 추가	2014.09
9	정규표현식과 XML 챕터 추가	2014.09
10	Do it 점프 투 파이썬 오프라인 책 출간 (이지스퍼블리싱)	2016.03
11	01-5 파이썬 들려보기 : IDLE 설명 추가	2016.07
12	05-1 클래스 : 개정	2016.10
13	02-07 불 자료형 : 개정	2018.02
14	07-05 파이썬으로 Json 처리하기 챕터 추가	2018.02
15	07-06 소트(Sort) 챕터 추가	2018.02
16	07-07 이터레이터 챕터 추가(비공개)	2018.02
17	07-08 제너레이터 챕터 추가 (비공개)	2018.02
18	07-09 클로저 챕터 추가 (비공개)	2018.02
19	07-10 데코레이터 챕터 추가 (비공개)	2018.02
20	05-06 외장함수에 namedtuple, defaultdict 추가	2018.02
21	A-2 str과 repr 챕터 추가	2018.02
22	A-3 유니코드와 인코딩 챕터 추가	2018.02
23	A-4 pip 챕터 추가	2018.02
24	02-2 문자열 자료형에 f 문자열 포매팅 내용 추가	2018.04
25	챕터별 연습문제 추가	2018.04

순서	주요 내용	날짜
26	종합문제 추가	2018.04

00-4 책구입안내

점프 투 파이썬이 오프라인 책으로 출간되었습니다. (이지스퍼블리싱. 2016년 3월)



- yes24에서 구매하기
- 전자책(ebook) 구매하기 - 전자책은 위키독스 버전입니다.

책 구입 FAQ

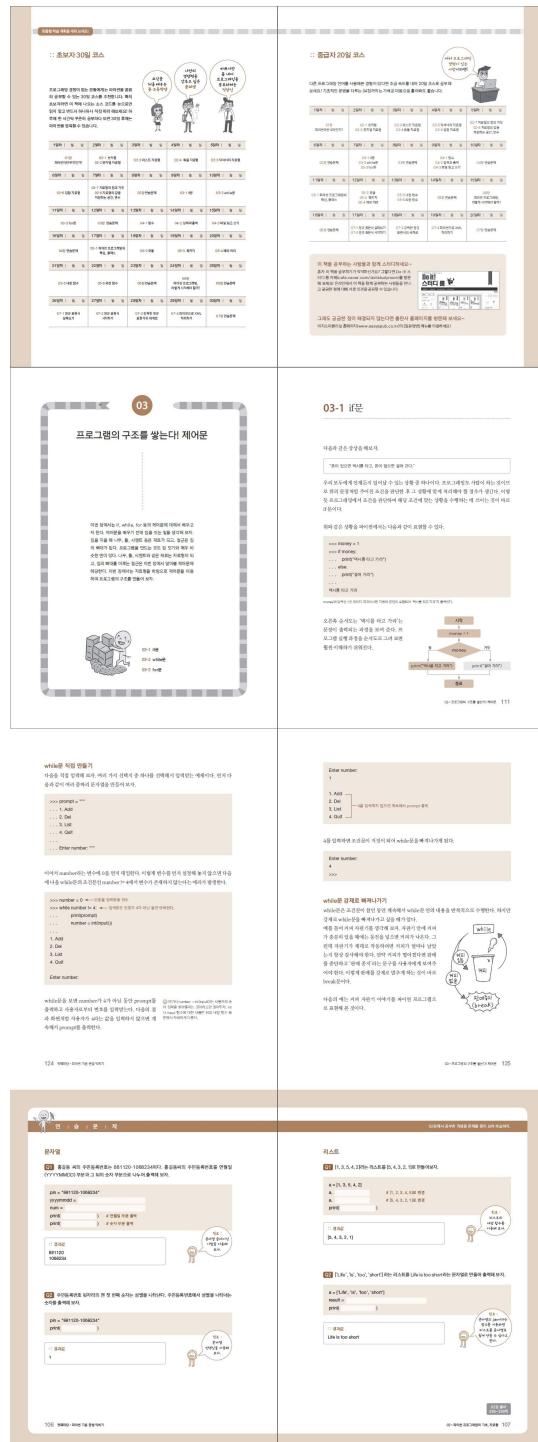
- 책의 내용은 위키독스의 “점프 투 파이썬”과 완전히 동일합니다.
- 전자책(ebook)은 위키독스에 공개된 버전으로 구매 요청 시 실시간 제작되어 발송됩니다.
- 오프라인 책과 전자책은 책 디자인(삽화 및 기본템플릿)이 다릅니다. 전자책은 디자인 없이 내용만 포함되어 있습니다.

인터넷서점에서 판매하는 이북은 오프라인 책과 동일한 내용과 디자인으로 되어 있는 출판사에서 제작한 전자책입니다. 위키독스에서 판매하는 이북은 사이트에 있는 내용이 그대로 전자책으로만 들어져서 발송됩니다.

- 위키독스 버전은 계속 수정되기 때문에 최신내용을 담고 있습니다.
- 위키독스 버전은 디자인적인 요소(그림이나 표등)가 전혀 없습니다.

점프 투 파이썬 오프라인 책 미리보기

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.



※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

01장 파이썬이란 무엇인가?

필자는 파이썬의 >>> 프롬프트를 처음 본 순간부터 지금까지 줄곧 파이썬과 함께 지내온 듯하다. “프로그래밍은 어렵고 지루한 것이다”라는 고정관념을 가지고 있던 필자에게 파이썬은 커다란 충격으로 다가왔다. 여러분도 이 책을 통해 파이썬의 매력에 흠뻑 빠져 보기를 바란다. 이 장에서는 파이썬의 특징과 장단점을 알아보고, 파이썬 프로그래밍을 위한 환경 구축 방법에 대해 배운다. 그리고 간단한 파이썬 프로그램도 작성해 본다.

01-1 파이썬이란?

파이썬(Python)은 1990년 암스테르담의 귀도 반 로섬(Guido Van Rossum)이 개발한 인터프리터¹ 언어이다. 귀도는 파이썬이라는 이름을 자신이 좋아하는 코미디 쇼인 “몬티 파이썬의 날아다니는 서커스(Monty Python's Flying Circus)”에서 따왔다고 한다. 파이썬의 사전적인 의미는 고대 신화에 나오는 파르나소스 산의 동굴에 살던 큰 뱀을 뜻하며, 아폴로 신이 텔파이에서 파이썬을 퇴치했다는 이야기가 전해지고 있다. 대부분의 파이썬 책 표지와 아이콘이 뱀 모양으로 그려져 있는 이유가 여기에 있다.



파이썬은 우리나라에서는 아직 대중적으로 사용되고 있지 않지만 외국에서는 교육 목적뿐 아니라 실무에서도 많이 사용되고 있다. 그 대표적인 예가 바로 구글이다. 필자는 구글에서 만들어진 소프트웨어의 50% 이상이 파이썬으로 만들어졌다는 이야기를 들은 적도 있다. 이외에도 많이 알려진 예를 몇 가지 들자면 파일 동기화 서비스인 드롭박스(Dropbox), 쉽고 빠르게 웹 개발을 할 수 있도록 도와주는 프레임워크인 장고(Django) 등이 있다.

또한 파이썬 프로그램은 공동 작업과 유지 보수가 매우 쉽고 편하다. 그 때문에 이미 다른 언어로 작성된 많은 프로그램과 모듈들이 파이썬으로 재구성되고 있다. 국내에서도 그 가치를 인정받아 사용자층이 더욱 넓어지고 있고, 파이썬을 이용해 프로그램을 개발하는 기업체들 또한 늘어 가고 있는 추세이다.

¹ 인터프리터 언어란 한 줄씩 소스 코드를 해석해서 그때그때 실행해 결과를 바로 확인할 수 있는 언어이다.

01-2 파이썬의 특징

필자는 파이썬을 무척 좋아한다. 모든 프로그래밍 언어에는 각기 장점이 있지만 파이썬에는 다른 언어들에서는 쉽게 찾아볼 수 없는 파이썬만의 독특한 매력이 있다. 파이썬의 특징을 알면 왜 파이썬을 공부해야 하는지, 과연 시간을 투자할 만한 가치가 있는지 분명하게 판단할 수 있을 것이다.

파이썬은 인간다운 언어이다

프로그래밍이란 인간이 생각하는 것을 컴퓨터에 지시하는 행위라고 할 수 있다. 앞으로 살펴볼 파이썬 문법에서도 보게 되겠지만 파이썬은 사람이 생각하는 방식을 그대로 표현할 수 있는 언어이다. 따라서 프로그래머는 굳이 컴퓨터의 사고 체계에 맞추어서 프로그래밍을 하려고 애쓸 필요가 없다. 이제 곧 어떤 프로그램을 구상하자마자 머릿속에서 생각한 대로 솔솔 써 내려가는 여러분의 모습에 놀라게 될 것이다.

아래 소스 코드를 보면 이 말이 쉽게 이해될 것이다.

```
if 4 in [1,2,3,4]: print("4가 있습니다")
```

위의 예제는 다음처럼 읽을 수 있다:

“만약 4가 1,2,3,4중에 있으면” 4가 있습니다“를 출력한다.”

프로그램을 모르더라도 직관적으로 무엇을 뜻하는지 알 수 있지 않는가? 마치 영어 문장을 읽는 듯한 착각에 빠져든다.

파이썬은 문법이 쉬워 빠르게 배울 수 있다

어려운 문법과 수많은 규칙에 둘러싸인 언어에서 탈피하고 싶지 않은가? 파이썬은 문법 자체가 아주 쉽고 간결하며 사람의 사고 체계와 매우 닮아 있다. 배우기 쉬운 언어, 활용하기 쉬운 언어가 가장 좋은 언어가 아닐까? 유명한 프로그래머인 에릭 레이먼드(Eric Raymond)는 파이썬을 공부한 지 단 하루 만에 자신이 원하는 프로그램을 작성할 수 있었다고 한다.

※ [프로그래밍](#) 경험이 조금이라도 있는 사람이라면 파이썬의 자료형, 함수, 클래스 만드는 법, 라이브러리 및 내장 함수 사용 방법 등을 익히는데 1주일이면 충분하리라 생각한다.

파이썬은 무료이지만 강력하다

오픈 소스²인 파이썬은 당연히 무료이다. 사용료 걱정없이 언제 어디서든 파이썬을 다운로드하여 사용할 수 있다.

또한 프로그래머는 만들고자 하는 프로그램의 대부분을 파이썬으로 만들 수 있다. 물론 시스템 프로그래밍이나 하드웨어 제어와 같은 매우 복잡하고 반복 연산이 많은 프로그램은 파이썬과 어울리지 않는다. 하지만 파이썬은 이러한 약점을 극복할 수 있게끔 다른 언어로 만든 프로그램을 파이썬 프로그램에 포함시킬 수 있다.

파이썬과 C는 찰떡궁합이란 말이 있다. 즉, 프로그램의 전반적인 뼈대는 파이썬으로 만들고, 빠른 실행 속도를 필요로 하는 부분은 C로 만들어서 파이썬 프로그램 안에 포함시키는 것이다(정말 놀라우리만치 영악한 언어가 아닌가). 사실 파이썬 라이브러리³들 중에는 순수 파이썬만으로 제작된 것도 많지만 C로 만들어진 것도 많다. C로 만들어진 것들은 대부분 속도가 빠르다.

파이썬은 간결하다

귀도는 파이썬을 의도적으로 간결하게 만들었다. 만약 펄(Perl)과 같은 프로그래밍 언어가 100가지 방법으로 하나의 일을 처리할 수 있다면 파이썬은 가장 좋은 방법 1가지만 이용하는 것을 선호한다. 이 간결함의 철학은 파이썬 문법에도 그대로 적용되어 파이썬 프로그래밍을 하는 사람들은 잘 정리되어 있는 소스 코드를 볼 수 있다. 다른 사람이 작업한 소스 코드도 한눈에 들어와 이해하기 쉽기 때문에 공동 작업과 유지 보수가 아주 쉽고 편하다.

다음은 파이썬 프로그램의 예제이다. 이 프로그램 소스 코드를 굳이 이해하려 하지 않아도 된다. 이것을 이해할 수 있다면 여러분은 이미 파이썬에 중독된 사람일 것이다. 그냥 한번 구경해 보도록 하자.

² 오픈 소스(Open Source)란 저작권자가 소스 코드를 공개하여 누구나 별다른 제한 없이 자유롭게 사용, 복제, 배포, 수정할 수 있는 소프트웨어이다.

³ 파이썬 라이브러리는 파이썬 프로그램 작성시 불러와 사용할 수 있는 미리 만들어진 파이썬파일들의 모음이다.

```
# simple.py
languages = ['python', 'perl', 'c', 'java']

for lang in languages:
    if lang in ['python', 'perl']:
        print("%6s need interpreter" % lang)
    elif lang in ['c', 'java']:
        print("%6s need compiler" % lang)
    else:
        print("should not reach here")
```

이 예제는 프로그래밍 언어를 판별하여 그에 맞는 문장을 출력하는 파이썬 프로그램 예제이다. 다른 언어들에서 늘 보게 되는 단락을 구분하는 괄호({ }) 문자가 보이지 않는 것을 확인할 수 있다. 또한 줄을 참 잘 맞춘 코드라는 것도 알 수 있다. 파이썬 프로그램은 줄을 맞추지 않으면 실행이 되지 않는다. 코드를 예쁘게 작성하려고 줄을 맞추는 것이 아니라 실행이 되게 하려면 꼭 줄을 맞추어야 하는 것이다. 이렇듯 줄을 맞추어 코드를 작성하는 행위⁴는 가독성에 크게 도움이 된다.

파이썬은 프로그래밍을 즐기게 해준다

이 부분이 가장 강조하고 싶은 부분이다. 파이썬만큼 필자에게 프로그래밍을 즐기게 해준 언어는 없었다. 파이썬은 다른 것에 신경 쓸 필요 없이 내가 하고자 하는 부분에만 집중할 수 있게 해준다. 파이썬을 배우고 나면 다른 언어로 프로그래밍하는 것에 지루함을 느끼게 될지도 모른다. 조심하자!

파이썬은 개발 속도가 빠르다

마지막으로, 재미있는 다음 문장으로 파이썬의 특징을 마무리하려 한다.

“Life is too short, You need python.” (인생은 너무 짧으니 파이썬이 필요해.)

파이썬의 엄청나게 빠른 개발 속도를 두고 유행처럼 퍼진 말이다. 이 위트 있는 문장은 이 책에서 계속 예제로 사용될 것이다.

⁴이렇게 코드의 줄을 맞추는 것을 “들여쓰기”라고 부른다. 파이썬에서 들여쓰기를 하지 않으면 프로그램이 실행되지 않는다.

01-3 파이썬으로 무엇을 할 수 있을까?

프로그래밍 언어를 좋은 언어와 나쁜 언어로 구분할 수 있을까? 사실 현실에서 이런 구분은 무의미하다. 어떤 언어든지 강점과 약점이 존재하기 때문이다. 그러므로 어떤 프로그래밍 언어가 어떤 일에 효율적인지를 안다는 것은 프로그래머의 생산성을 크게 높일수 있는 힘이 된다. 그렇다면 파이썬으로 하기에 적당한 일과 적당하지 않은 일은 무엇일까? 이에 대해서 알아보는 것은 매우 가치 있는 일이 될 것이다.

파이썬으로 할 수 있는 일

파이썬으로 할 수 있는 일은 아주 많다. 대부분의 프로그래밍 언어가 하는 일을 파이썬은 쉽고 깔끔하게 처리한다. 파이썬으로 할 수 있는 일들을 나열하자면 끝도 없겠지만 대표적인 몇 가지 예를 들어 보겠다.

시스템 유틸리티 제작

파이썬은 운영체제(윈도우, 리눅스 등)의 시스템 명령어들을 이용할 수 있는 각종 도구를 갖추고 있기 때문에 이를 바탕으로 갖가지 시스템 유틸리티⁵를 만드는 데 유리하다. 실제로 여러분은 시스템에서 사용 중인 서로 다른 유틸리티성 프로그램들을 하나로 뭉쳐서 큰 힘을 발휘하게 하는 프로그램들을 무수히 만들어낼 수 있다.

GUI 프로그래밍

GUI(Graphic User Interface) 프로그래밍이란 쉽게 말해 윈도우 창처럼 화면을 보면 마우스나 키보드로 조작할 수 있는 프로그램을 만드는 것이다. 파이썬으로 GUI 프로그램을 만드는 것은 다른 언어를 이용해 만드는 것보다 훨씬 쉽다. 대표적인 예로 파이썬 프로그램을 설치할때 함께 설치되는 기본 모듈인 Tkinter(티케이인터)를 이용해 만드는 GUI 프로그램을 들 수 있다. 실제로 Tkinter를 이용한 파이썬 GUI 프로그램의 소스 코드는 매우 간단하다. Tkinter를 이용하면 단 5 줄의 소스 코드만으로도 윈도우 창을 띠울 수 있다. 놀랍지 않은가!

※ 파이썬에는 wxPython, PyQT, PyGTK 등과 같이 Tkinter보다 빠른 속도와 보기 좋은 인터페이스를 자랑하는 것들도 있다.

⁵ 유틸리티란 컴퓨터 이용에 도움이 되는 여러소프트웨어를 말한다.

C/C++와의 결합

파이썬은 접착(glue) 언어라고도 부르는데, 그 이유는 다른 언어들과 잘 어울려 다른 언어와 결합해서 사용할 수 있기 때문이다. C나 C++로 만든 프로그램을 파이썬에서 사용할 수 있으며, 파이썬으로 만든 프로그램 역시 C나 C++에서 사용할 수 있다.

웹 프로그래밍

일반적으로 익스플로러나 크롬, 파이어폭스와 같은 브라우저를 이용해 인터넷을 사용하는데, 누구나 한 번쯤 웹 서핑을 하면서 게시판이나 방명록에 글을 남겨 본 적이 있을 것이다. 그러한 게시판이나 방명록을 바로 웹 프로그램이라고 한다. 파이썬은 웹 프로그램을 만들기에 매우 적합한 도구이며 실제로 파이썬으로 제작된 웹사이트는 셀 수 없을 정도로 많다.

수치 연산 프로그래밍

사실 파이썬은 수치 연산 프로그래밍에 적합한 언어는 아니다. 수치가 복잡하고 연산이 많다면 C 같은 언어로 하는 것이 더 빠르기 때문이다. 하지만 파이썬에는 NumPy이라는 수치 연산 모듈이 제공된다. 이 모듈은 C로 작성되었기 때문에 파이썬에서도 수치 연산을 빠르게 할 수 있다.

데이터베이스 프로그래밍

파이썬은 사이베이스(Sybase), 인포믹스(Infomix), 오라클(Oracle), 마이에스큐엘(MySQL), 포스트그레스큐엘(PostgreSQL) 등의 데이터베이스에 접근할 수 있게 해주는 도구들을 제공한다.

또한 이런 굵직한 데이터베이스를 직접 이용하는 것 외에도 파이썬에는 재미있는 모듈이 하나 더 있다. 바로 피클(pickle)이라는 모듈이다. 피클은 파이썬에서 사용되는 자료들을 변형없이 그대로 파일에 저장하고 불러오는 일들을 맡아 한다. 이 책에서는 외장 함수에서 피클을 어떻게 사용하고 활용하는지에 대해서 알아본다.

데이터 분석, 사물 인터넷

파이썬으로 만들어진 펜더스(Pandas)라는 모듈을 이용하면 데이터 분석을 더 쉽고 효과적으로 할 수 있다. 데이터 분석을 할 때 아직까지는 데이터 분석에 특화된 “R”이라는 언어를 많이 사용하고 있지만, 펜더스가 등장한 이후로 파이썬을 이용하는 경우가 점점 증가하고 있다. 사물 인터넷 분야에서도 파이썬은 활용도가 높다. 한 예로 라즈베리파이(Raspberry Pi)는 리눅스 기반의 아주 작은 컴퓨터이다. 라즈베리파이를 이용하면 홈시어터나 아주 작은 게임기 등 여러 가지 재미있는

것들을 만들 수 있는데 파이썬은 이 라즈베리파이를 제어하는 도구로 사용된다. 예를 들어 라즈베리파이에 연결된 모터를 작동시키거나 LED에 불이 들어오게 하는 일들을 파이썬으로 할 수 있다.

파이썬으로 할 수 없는 일

시스템과 밀접한 프로그래밍 영역

파이썬으로 도스나 리눅스 같은 운영체제, 엄청난 횟수의 반복과 연산을 필요로 하는 프로그램 또는 데이터 압축 알고리즘 개발 프로그램 등을 만드는 것은 어렵다. 즉, 대단히 빠른 속도를 요구하거나 하드웨어를 직접 건드려야 하는 프로그램에는 어울리지 않는다.

모바일 프로그래밍

파이썬은 구글이 가장 많이 애용하는 언어이지만 파이썬으로 안드로이드 앱(App)을 개발하는 것은 아직 어렵다. 안드로이드에서 파이썬으로 만든 프로그램들이 실행되도록 지원하긴 하지만 이 것만으로 앱을 만들기에는 아직 역부족이다. 아이폰 앱을 개발하는 것 역시 파이썬으로는 할 수 없다.

01-4 파이썬 설치하기

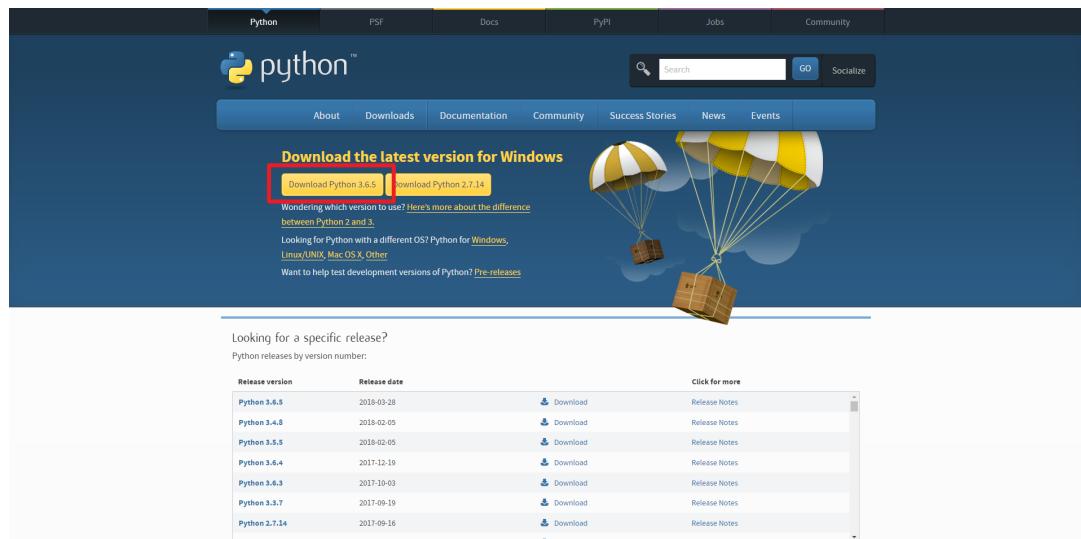
이제 실습을 위해 컴퓨터에 파이썬을 설치해 보자. 이 책에서는 윈도우와 리눅스에서 설치하는 방법만 다룬다. 다른 시스템을 이용할 경우 파이썬 홈페이지(<http://www.python.org>)의 설명을 참고하도록 하자.

윈도우에서 파이썬 설치하기

윈도우의 경우에는 설치가 정말 쉽다.

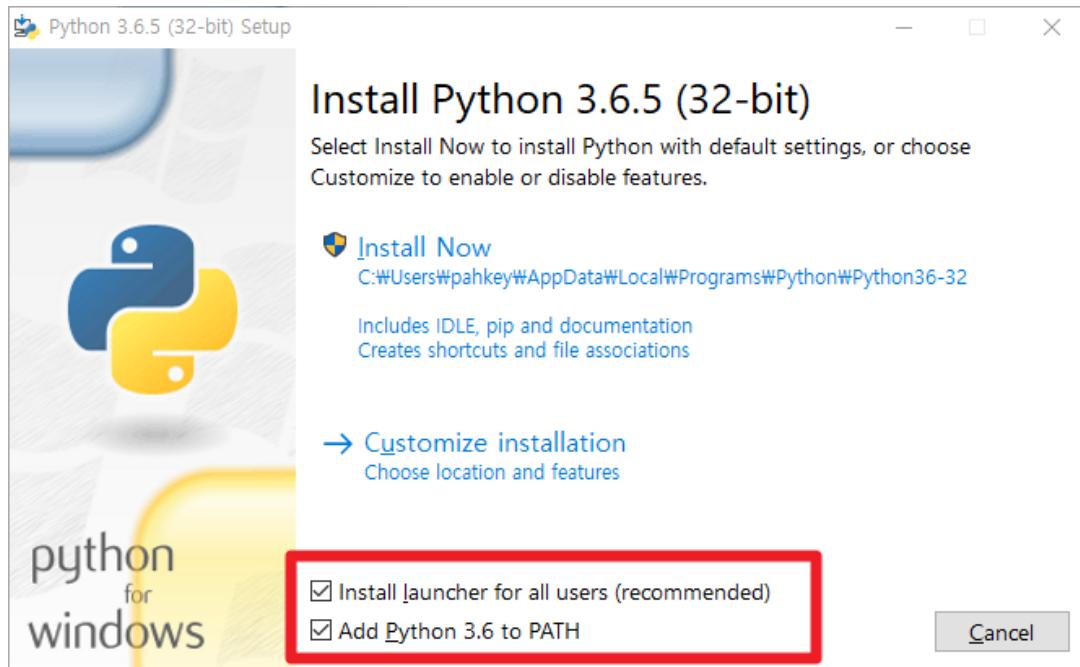
1. 우선 파이썬 공식 홈페이지의 다운로드 페이지(<http://www.python.org/downloads>)에서 윈도우용 파이썬 언어 패키지를 다운로드한다. 다음 화면에서 Python 3.x로 시작하는 버전 중 가장 최근의 윈도우 인스톨러를 다운로드하도록 하자(이 글을 작성하는 시점의 최신 버전은 3.6.5이다.).

※ 만약 파이썬 2.7 버전을 설치할 경우에는 Python 2.7용 인스톨러 파일을 받아서 설치하면 된다.



2. 인스톨러를 실행한 후에 “Install Now”를 선택하면 바로 설치가 진행된다.

파이썬이 어느 곳에서든지 실행될 수 있도록 “Add Python 3.6 to PATH” 옵션을 선택하도록 하자.



※ “Add Python 3.6 to PATH” 옵션을 누락할 경우 이후 설명되는 예제에서 오류가 발생할 수 있다. 만약 python이 설치되는 경로와 PATH에 대한 사전지식이 있는 사용자라면 이 옵션을 생략해도 된다.

3. 설치가 완료되면 [close]를 클릭하여 종료한다.

파이썬이 정상적으로 설치되었다면 오른쪽 그림과 같이 프로그램 메뉴에서 확인할 수 있을 것이다.

[시작 → 모든 프로그램(모든 앱) → Python 3.6]



리눅스에서 파이썬 설치

리눅스 사용자라면 기본적으로 파이썬이 설치되어 있을 것이다.

```
$ python -V
```

리눅스 셸에서 위의 명령어를 입력하면 파이썬 버전을 확인할 수 있다.

만약 파이썬이 기본으로 설치되어 있지 않다면 소스를 컴파일하여 설치한다. 리눅스의 경우 배포본별로 여러 가지 설치 방법이 있을 수 있는데, 소스를 컴파일하여 설치하는 것이 모든 배포본에서 사용할 수 있는 가장 일반적인 방법이다.

파이썬 공식 홈페이지의 다운로드 페이지(<http://www.python.org/download>)에 접속해 “Python-3.X.X.tgz”를 다운로드한다. 이 책에서는 Python-3.6.5 버전을 사용한다.

먼저 터미널에 다음 문장을 입력하여 다운로드한 파일의 압축을 푼다.

```
$ tar xvzf Python-3.6.5.tgz
```

그런 다음 해당 디렉터리로 이동한다.

```
$ cd Python-3.6.5
```

Makefile 파일을 만들기 위해서 configure를 실행한다.

```
$ ./configure
```

파이썬 소스를 컴파일한다.

```
$ make
```

루트 계정으로 설치한다.

```
$ su -
$ make install
```

파이썬 2.7 버전을 설치할 경우 Python-3.6.5.tgz가 아닌 Python-2.7.tgz 파일을 다운로드해 동일한 방법으로 설치하면 된다.

01-5 파이썬 둘러보기

도대체 파이썬이라는 언어는 어떻게 생겼는지, 간단한 코드를 작성하며 알아보자. 파이썬을 자세히 탐구하기 전에 전체적인 모습을 쭉 훑어보는 것은 매우 유익한 일이 될 것이다.

“백문이 불여일견, 백견이 불여일타”라고 했다. 직접 따라 해보자.

파이썬 기초 실습 준비하기

파이썬 프로그래밍 실습을 시작하기 전에 기초적인 것을 준비해 보자.

[시작] 메뉴에서 [프로그램 → Python 3.6 → Python 3.6(32-bit)]을 선택하자.



그리면 다음과 같은 화면이 나타난다.

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

위와 같은 것을 대화형 인터프리터라고 하는데, 앞으로 이 책에서는 이 인터프리터로 파이썬 프로그래밍의 기초적인 사항들에 대해 설명할 것이다.

※ 대화형 인터프리터는 파이썬 쉘(Python shell)이라고도 한다. 3개의 꺾은 괄호 (>>>)는 프롬프트(prompt)라고 한다.

대화형 인터프리터를 종료할 때는 Ctrl+Z 를 누른다 (유닉스 계열에서는 Ctrl+D). 또는 다음의 예와 같이 sys 모듈을 사용하여 종료할 수도 있다.

```
>>> import sys
>>> sys.exit()
```

파이썬 기초 문법 따라 해보기

여기서 소개하는 내용들은 나중에 다시 자세하게 다룰 것이니 이해가 되지 않는다고 절망하거나 너무 고심하지 말도록 하자.

파이썬 셸을 실행하여 다음을 직접 입력해 보자.

사칙연산

1 더하기(+) 2는 3이라는 값을 출력해 보자. 보통 계산기 사용하듯 더하기 기호만 넣어 주면 된다.

```
>>> 1 + 2
3
```

나눗셈(/)과 곱셈(*) 역시 예상한 대로 결과값을 보여준다.

```
>>> 3 / 2.4
1.25
>>> 3 * 9
27
```

우리가 일반적으로 알고 있는 \div 기호나 \times 기호가 아닌 것에 주의하자.

변수에 숫자 대입하고 계산하기

```
>>> a = 1
>>> b = 2
>>> a + b
3
```

a에 1을, b에 2를 대입한 다음 a와 b를 더하면 3이라는 결과값을 보여 준다.

변수에 문자 대입하고 출력하기

```
>>> a = "Python"
>>> print(a)
Python
```

a라는 변수에 Python이라는 값을 대입한 다음 print(a) 라고 작성하면 a의 값을 출력한다.

※ 파이썬은 대소문자를 구분한다. print를 PRINT로 쓰면 정의되지 않았다는 에러 메시지가 나온다.

또는 다음과 같이 print문을 생략하고 변수명 a만 입력하여 a의 값을 출력할 수도 있다.

```
>>> a = "Python"
>>> a
Python
```

조건문 if

다음은 간단한 조건문 if를 이용한 예제이다.

```
>>> a = 3
>>> if a > 1:
...     print("a is greater than 1")
...
a is greater than 1
```

※ print문 앞의 '...'은 아직 문장이 끝나지 않았음을 의미한다.

위 예제는 a가 1보다 크면 “a is greater than 1”이라는 문장을 출력(print)하라는 뜻이다. 위 예제에서 a는 3이므로 1보다 크다. 따라서 두 번째 “...” 이후에 Enter키를 입력하면 if문이 종료되고 “a is greater than 1”이라는 문장이 출력된다.

`if a > 1:` 다음 문장은 Tap 키 또는 Spacebar 키 4개를 이용해 반드시 들여쓰기 한 후에 `print("a is greater than 1")`이라고 작성해야 한다. 들여쓰기 규칙에 대해서는 03장 제어

문에서 자세하게 알아볼 것이다. 바로 뒤에 이어지는 반복문 for, while 예제도 마찬가지로 들여쓰기가 필요하다.

반복문 for

다음은 **for**를 이용해서 [1, 2, 3]안의 값들을 하나씩 출력해 주는 것을 보여주는 예이다.

```
>>> for a in [1, 2, 3]:
...     print(a)
...
1
2
3
```

for문을 이용하면 실행해야 할 문장을 여러 번 반복해서 실행시킬 수 있다. 위의 예는 대괄호([]) 사이에 있는 값들을 하나씩 출력한다. 위 코드의 의미는 “[1, 2, 3]이라는 리스트의 앞에서부터 하나씩 꺼내어 a라는 변수에 대입한 후 print(a)를 수행하라”이다. 당연히 a에 차례로 1, 2, 3이라는 값이 대입되며 print(a)에 의해서 그 값이 차례대로 출력된다.

반복문 while

다음은 **while**을 이용하는 예이다.

```
>>> i = 0
>>> while i < 3:
...     i=i+1
...     print(i)
...
1
2
3
```

while이라는 영어 단어는 “~인 동안”이란 뜻이다. for문과 마찬가지로 반복해서 문장을 수행할 수 있도록 해준다. 위의 예제는 i 값이 3보다 작은 동안 $i=i+1$ 과 `print(i)`를 수행하라는 말이다. $i=i+1$ 이라는 문장은 i의 값을 1씩 더하게 한다. i 값이 1씩 증가되어 3이 되면 while문을 빠져나가게 된다.

함수

파이썬의 **함수**는 다음과 같은 형태이다.

```
>>> def sum(a, b):
...     return a+b
...
>>> print(sum(3,4))
7
```

파이썬에서 **def**는 함수를 만들 때 사용하는 예약어이다. 위의 예제는 `sum`이라는 함수를 만들고 그 함수를 어떻게 사용하는지를 보여준다. `sum(a, b)`에서 `a, b`는 입력값이고, `a+b`는 결과값이다. 즉 3, 4가 입력으로 들어오면 $3+4$ 를 수행하고 그 결과값인 7을 돌려 준다.

이렇게 해서 기초적인 파이썬 문법에 대해서 간략하게 알아보았다.

01-6 파이썬과 에디터

에디터란 문서를 편집할 수 있는 프로그래밍 툴을 말한다. 대화형 인터프리터에서 만든 프로그램은 인터프리터를 종료함과 동시에 사라지지만 에디터로 만든 프로그램은 파일로 존재하기 때문에 언제든지 다시 사용할 수 있다.

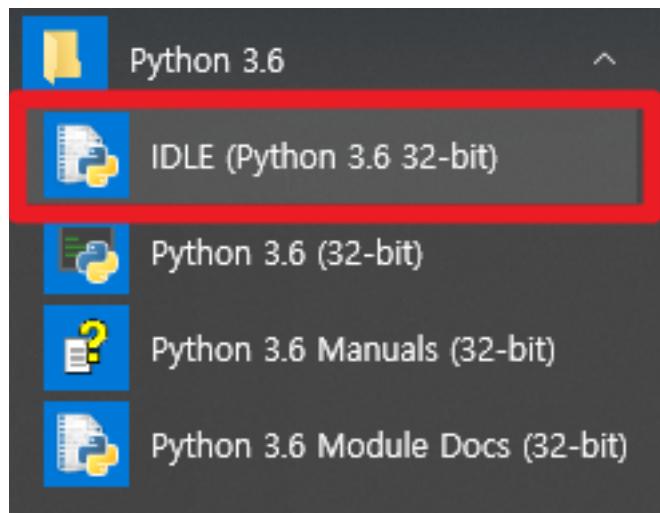
여기서는 파이썬 프로그래밍을 처음 시작하기에 좋은 IDLE를 이용하여 파이썬 프로그램을 작성해 보자.

IDLE

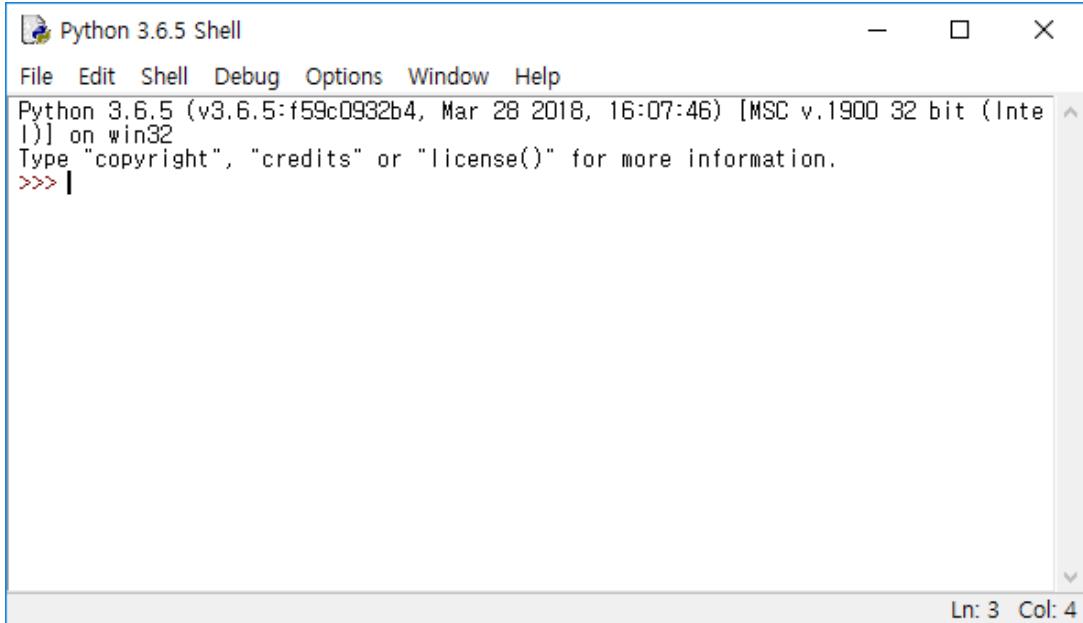
파이썬 IDLE(Integrated Development and Learning Environment)는 파이썬 프로그램 작성과 학습을 도와주는 통합 개발환경으로 파이썬 설치 시 기본으로 설치되는 프로그램이다.

파이썬 IDLE를 실행 해 보자.

[시작 -> 모든 프로그램 -> Python 3.6 -> IDLE 선택]



그러면 다음과 같은 IDLE 셸(Shell) 창이 나타난다.



IDLE는 크게 두가지 창으로 구성된다.

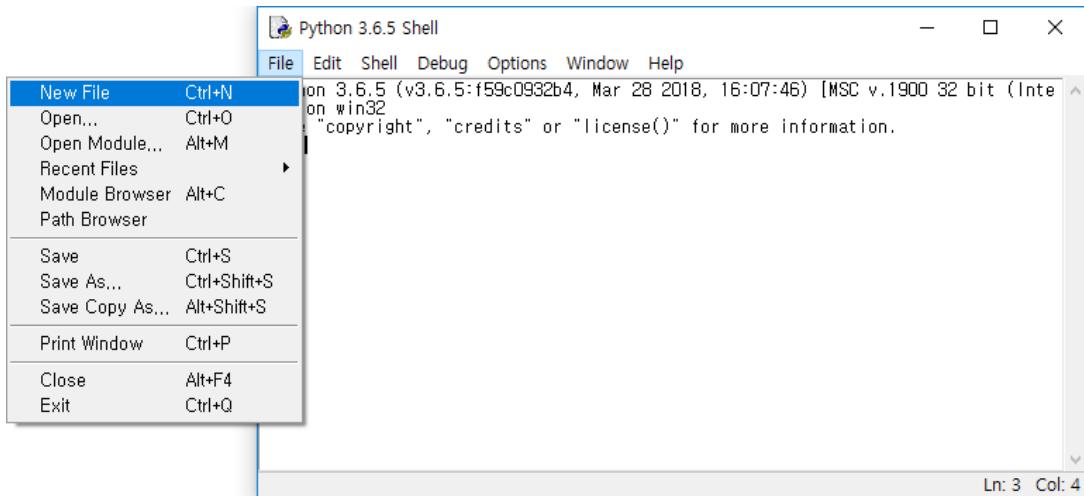
- IDLE 쉘 창(Shell Window) - IDLE 에디터에서 실행한 프로그램의 결과가 표시되는 창으로 파이썬 쉘과 동일한 기능을 수행한다.
- IDLE 에디터 창(Editor Window) - IDLE 에디터가 실행되는 창

IDLE 실행 시 가장 먼저 나타나는 창은 IDLE 쉘 창이다.

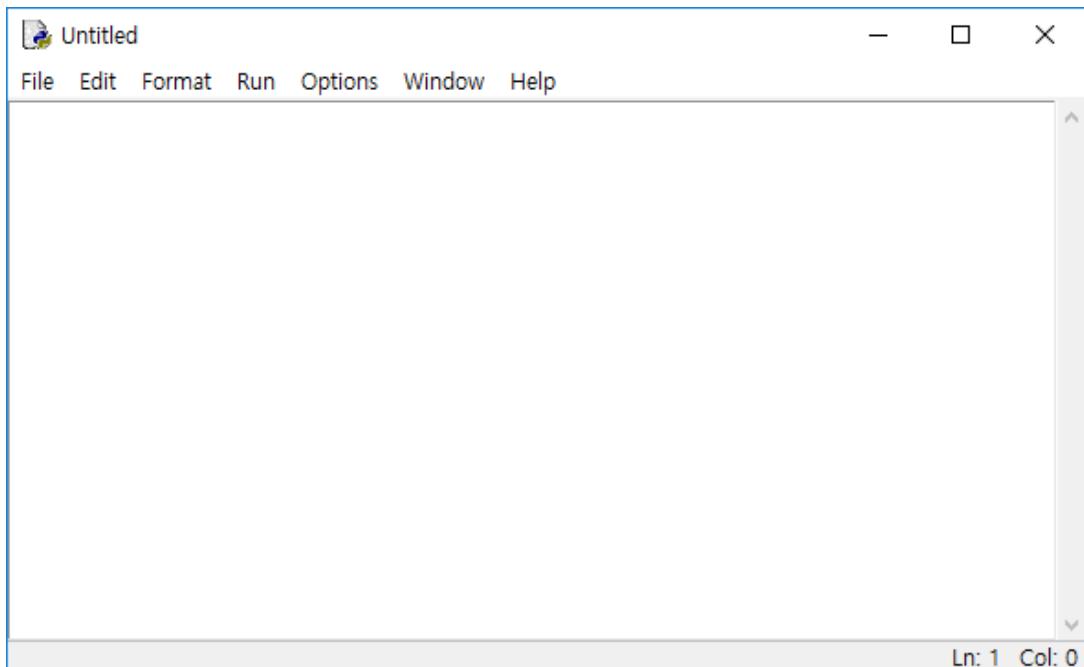
※ 이전 챕터에서 보았던 파이썬 쉘과 IDLE 쉘은 기능적으로 동일하다. 단, IDLE 쉘은 들여쓰기를 표시하는 방법등 기본 파이썬 쉘과 상이한 부분이 있기 때문에 이 책에서는 IDLE 쉘은 IDLE 에디터로 작성한 프로그램의 실행결과를 표시하는 용도로만 사용하기로 하자. 즉, 앞으로 >>> 프롬프트로 시작하는 예제는 IDLE 쉘이 아닌 파이썬 쉘로 실행해야 한다.

이제 IDLE 에디터(Editor)를 실행 해 보자.

다음과 같이 IDLE 쉘 창 메뉴에서 [File -> New File]을 선택하자.



그러면 다음 그림과 같은 IDLE 에디터가 나타날 것이다.



이제 IDLE 에디터에서 다음과 같은 파이썬 프로그램을 작성해 보자.

```
# hello.py
print("Hello World")
```

Ln: 2 Col: 20

위에서 `# hello.py`라는 문장은 주석이다. `#`으로 시작하는 문장은 `#`부터 시작해서 그 줄 끝까지 프로그램 수행에 전혀 영향을 주지 않는다. 주석은 프로그래머를 위한 것으로, 프로그램 소스에 설명문을 달 때 사용한다.

[여러줄짜리 주석문]

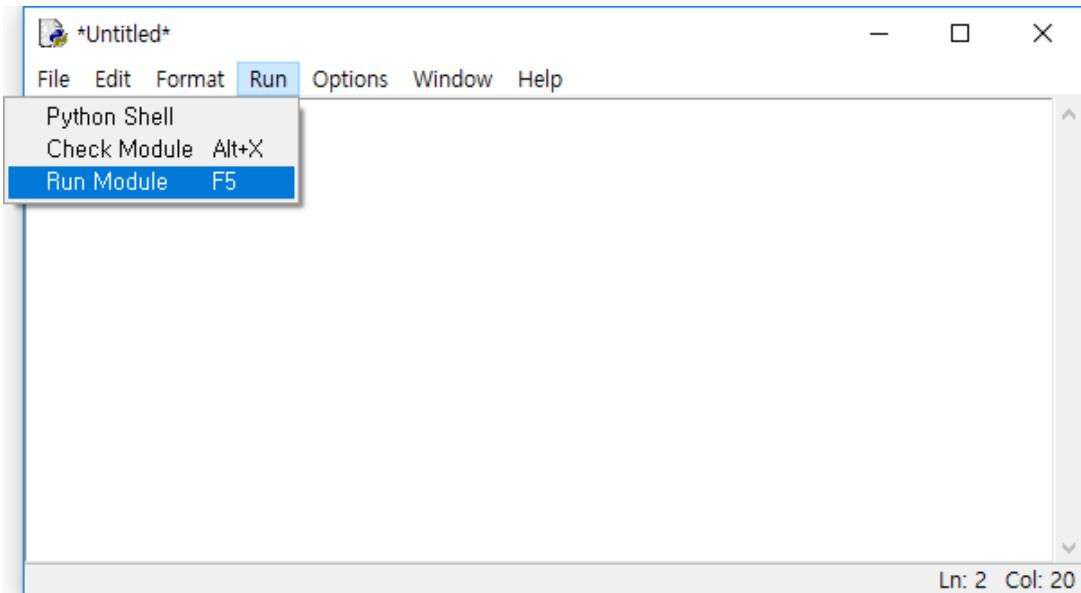
주석문이 여러 줄인 경우 다음의 방법을 사용하면 편리하다.

```
"""
Author: EungYong Park
Date : 2018-05-01
이 프로그램은 Hello World를 출력하는 프로그램이다.
"""
```

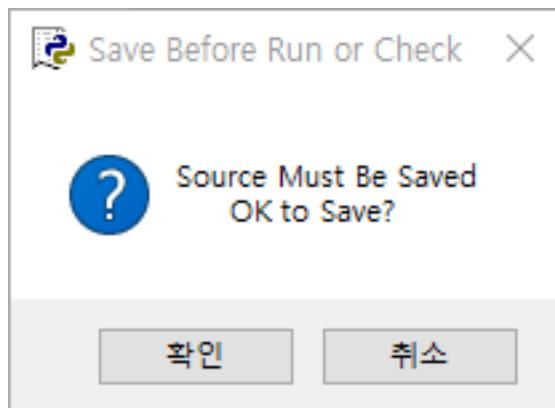
여러 줄로 이루어진 주석을 작성하려면 큰따옴표 세 개를 연속으로 사용한 “”” 기호 사이에 주석문을 작성하면 된다. 큰따옴표 대신 작은따옴표 세 개를(“”)를 사용해도 된다.

이제 작성한 프로그램을 실행 해 보자.

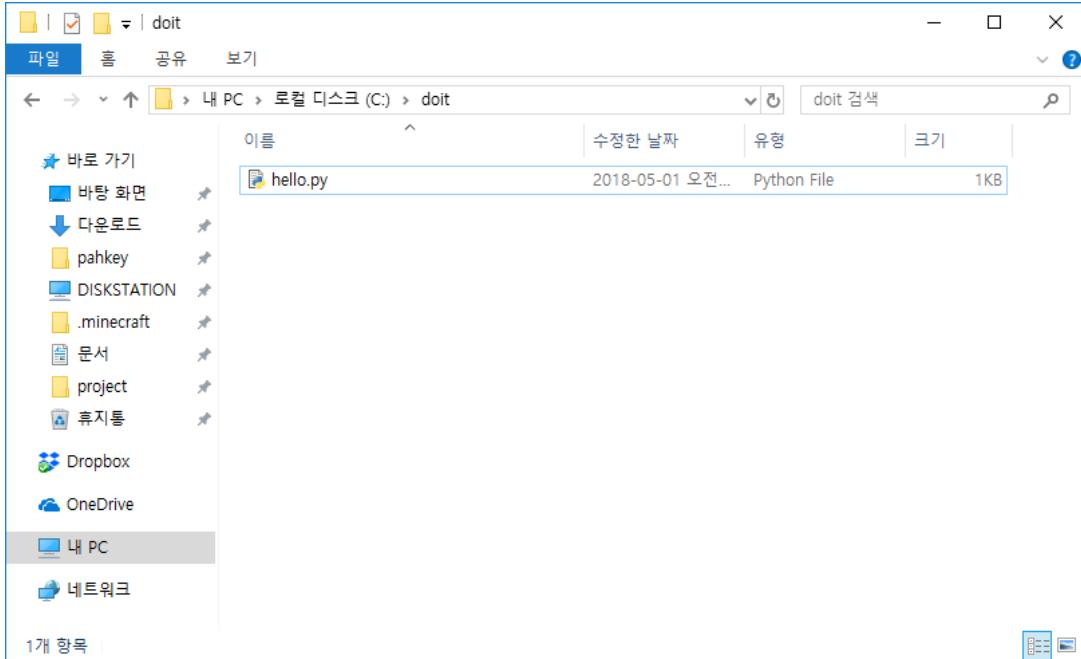
IDLE 창 메뉴에서 [Run -> Run Module]을 선택하자. (단축키: F5)



실행 해 보면 파일을 먼저 저장하라는 다음과 같은ダイ얼로그가 나올것이다.



“확인”을 선택하고 C:\doit이라는 디렉터리를 생성한 후 hello.py라는 이름으로 저장을 하도록 하자. 에디터로 파일 프로그램을 작성한 후 저장할 때는 파일 이름의 확장자명을 항상 py로 해야한다. py는 파일임을 알려주는 관례적인 확장자명이다.



파일을 저장하면 자동으로 파이썬 프로그램이 실행 된다. 실행 결과는 다음과 같이 IDLE 셸 창에 표시된다.

```

Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: C:/doit/hello.py =====
Hello World
>>> |

```

The screenshot shows the Python 3.6.5 Shell window. It displays the Python version and build information, followed by a prompt. The command `RESTART: C:/doit/hello.py` is entered, and the output `Hello World` is displayed. The status bar at the bottom right shows `Ln: 6 Col: 4`.

프로그램을 재 실행하려면 에디터 창에서 F5키로 다시한번 실행하면 된다.

명령 프롬프트 창에서 파이썬 프로그램 실행하기

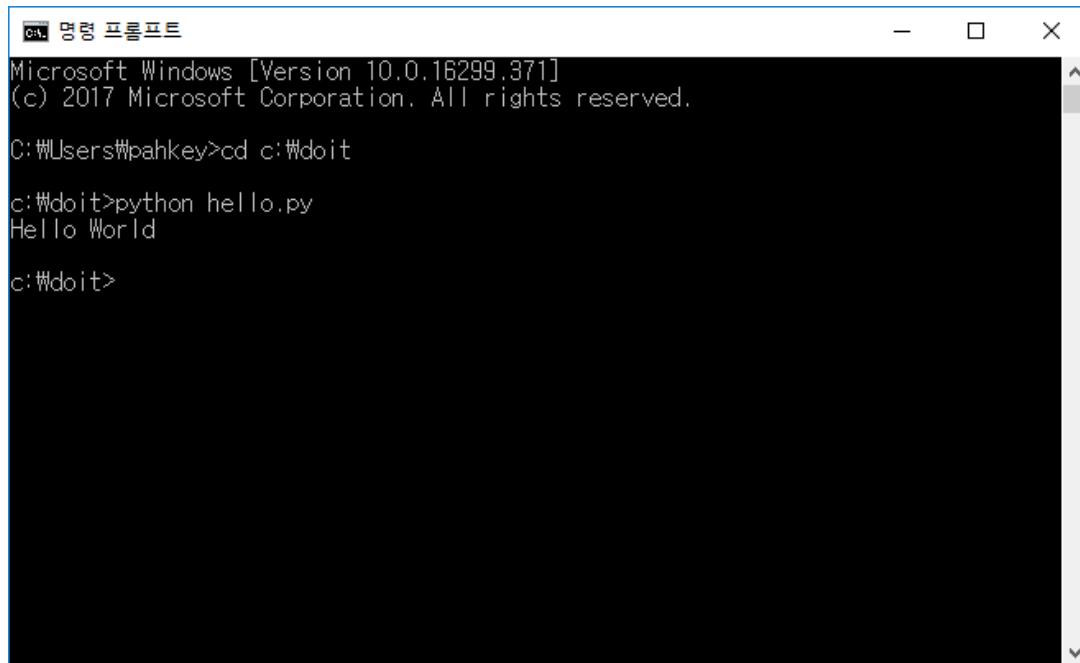
이번에는 앞서 IDLE로 작성한 hello.py 파일을 명령 프롬프트 창에서 실행해 보도록 하자.

우리는 이미 다음과 같은 프로그램을 c:\doit 디렉토리에 hello.py라는 이름으로 저장했다.

```
# hello.py
print("Hello world")
```

이제 이 hello.py라는 프로그램을 실행시키기 위해 [윈도우+R -> cmd 입력 -> Enter]를 눌러 명령 프롬프트 창을 연다.

그리고 다음과 같이 입력한다.



```
명령 프롬프트
Microsoft Windows [Version 10.0.16299.371]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\pahkey>cd c:\doit

c:\doit>python hello.py
Hello World

c:\doit>
```

위와 같은 결과값을 볼 수 있을 것이다. 결과값이 위와 같지 않다면 hello.py 파일이 C:\doit 디렉토리에 존재하는지 다시 한 번 살펴보도록 하자.

여러가지 에디터

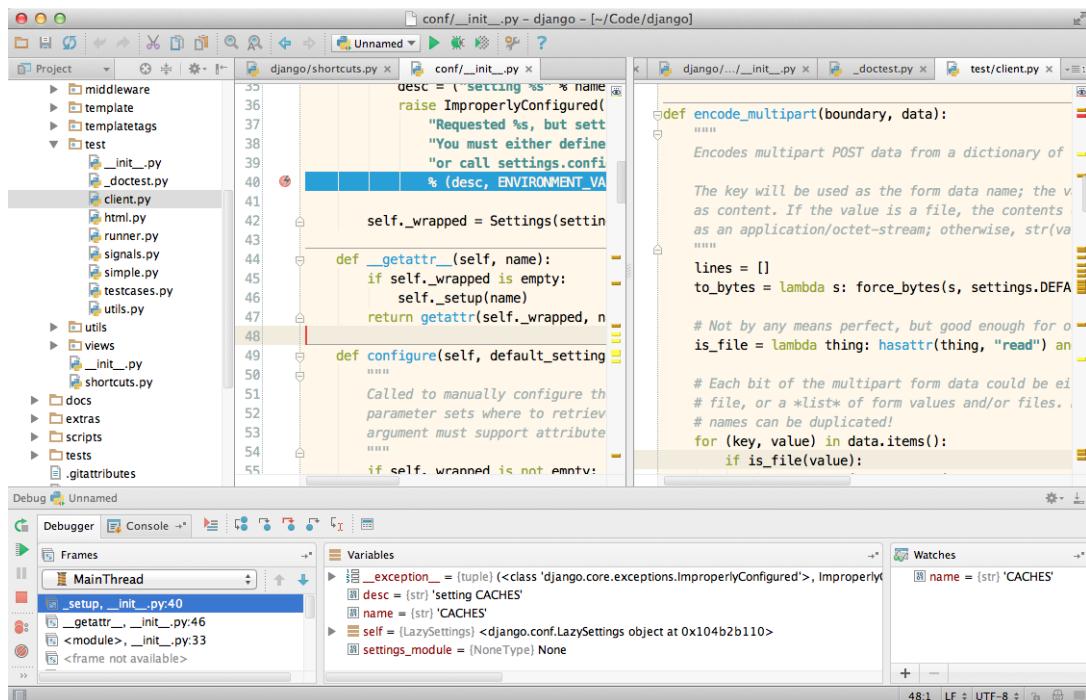
파이썬으로 실무 프로그램을 작성하기에는 위에 소개한 IDLE보다는 많은 기능을 갖춘 전문적인 에디터가 유리하다. 다음에 추천할 만한 에디터들을 소개해 두었으니 참고하도록 하자.

※ 이 책의 예제 중 에디터를 이용하여 작성해야 하는 경우에는 파일 공부에 최적화된 IDLE를 이용하여 작성하도록 하자. 다음의 에디터들은 파일 공부가 마무리 된 후에 사용해 보기를 권장한다.

파이참

파이썬에 어느 정도 익숙해졌다면 파이참(PyCharm)을 사용해 보기 좋다. 파이참은 가장 유명한 파이썬 에디터 중 하나로 코드 작성 시 자동 완성, 문법 체크 등 편리한 기능들을 많이 제공한다.

이 에디터는 파이참 공식 다운로드 사이트 (<http://www.jetbrains.com/pycharm/download>)에서 다운로드할 수 있다.



에디트 플러스

이 프로그램은 무료 소프트웨어가 아니기 때문에 평가판을 이용해야 한다. 다운로드 한 후부터 한 달간 사용할 수 있다. 에디트 플러스 공식 사이트 (<http://www.editplus.com/kr>)에서 파일을 다운로드해 설치하자.

The screenshot shows the EditPlus text editor interface. The title bar reads "C:\Python\sub_dir_search.py - EditPlus". The menu bar includes "파일(F)", "편집(E)", "보기(V)", "검색(S)", "문서(D)", "프로젝트(P)", "도구(I)", "브라우저(B)", "ZC", "창(W)", and "도움말(H)". The toolbar contains icons for file operations like Open, Save, Print, and Find. The main window displays the following Python script:

```

1 # C:/Python/sub_dir_search.py
2 import os
3
4 def search(dirname):
5     filenames = os.listdir(dirname)
6     for filename in filenames:
7         full_filename = os.path.join(dirname, filename)
8         print(full_filename)
9
10 search("c:/")
11

```

The left sidebar shows a file tree under [C:] with the following structure:

- C:\#
- Python
 - _pycache_
 - game
 - Mymodules

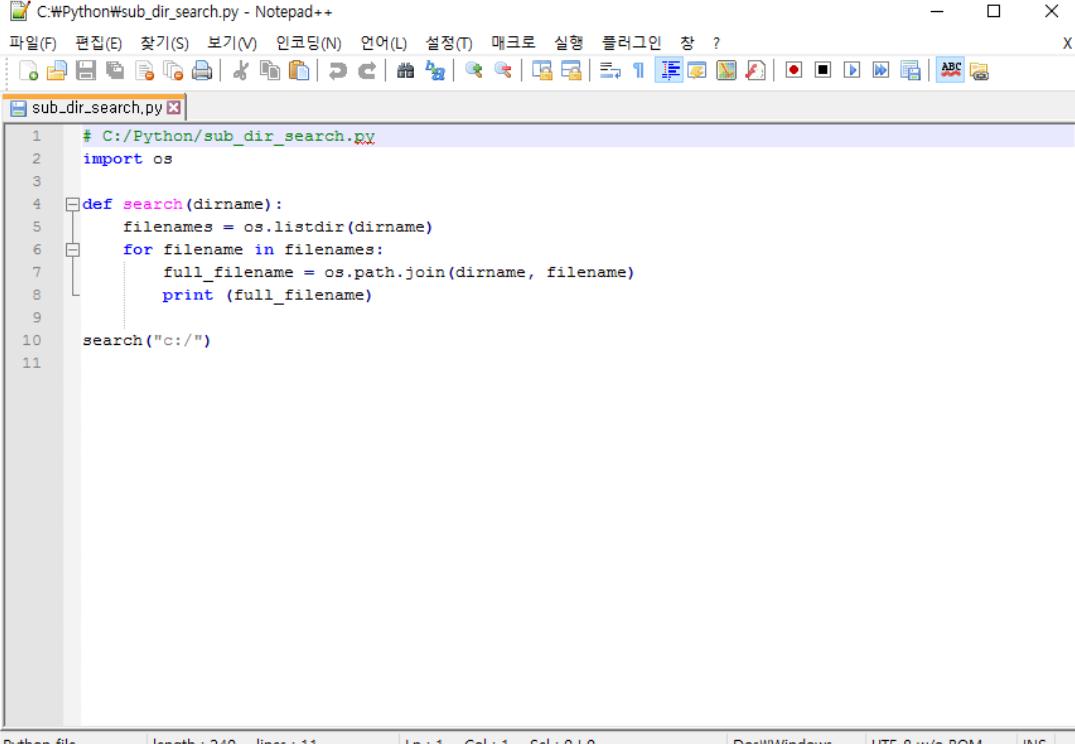
The bottom pane lists files in the current directory:

- a.txt
- addr.py
- addr2.py
- argv_test.py
- b.txt
- blog.xml
- fileopen.py
- gugu.py
- gugu1.py
- hello.py
- house.py
- marks1.py
- marks2.py
- marks3.py

The status bar at the bottom shows "All Files (*.*)", "줄 11", "칸 1", "11 00 PC ANSI", and "영".

노트패드++

노트패드++도 많은 사람들이 추천하는 윈도우용 파이썬 에디터 중의 하나이다. 이 에디터는 노트패드++ 공식 다운로드 사이트 (<https://notepad-plus-plus.org>)에서 다운로드할 수 있다.



The screenshot shows a Notepad++ window with the following Python script:

```

1  # C:/Python/sub_dir_search.py
2  import os
3
4  def search(dirname):
5      filenames = os.listdir(dirname)
6      for filename in filenames:
7          full_filename = os.path.join(dirname, filename)
8          print (full_filename)
9
10 search("c:/")
11

```

The status bar at the bottom indicates:

- Python file
- length : 240 lines : 11
- Ln : 1 Col : 1 Sel : 0 | 0
- Dos#Windows
- UTF-8 w/o BOM
- INS

서브라임 텍스트 3

서브라임 텍스트 3 역시 파이썬 사용자에게 사랑받는 에디터 중의 하나로 심플하면서 세련된 사용자 인터페이스를 자랑한다. 이 에디터는 서브라임 텍스트 3 공식 다운로드 사이트 (<http://www.sublimetext.com/3>)에서 다운로드할 수 있다.

The screenshot shows a Sublime Text window with the following details:

- File Path: C:\Python\sub_dir_search.py
- Editor Title: sub_dir_search.py
- Code Content:

```
1 # C:/Python/sub_dir_search.py
2 import os
3
4 def search(dirname):
5     filenames = os.listdir(dirname)
6     for filename in filenames:
7         full_filename = os.path.join(dirname, filename)
8         print (full_filename)
9
10 search("c:/")
11 |
```
- Status Bar: Line 11, Column 1
- Status Bar: Tab Size: 4
- Status Bar: Python

02장 파이썬 프로그래밍의 기초, 자료형

어떤 프로그래밍 언어든 “그 언어의 자료형을 알고 이해할 수 있다면 이미 그 언어의 절반을 터득한 것이나 다름없다”는 말이 있다.

자료형이란 프로그래밍을 할 때 쓰이는 숫자, 문자열 등 자료 형태로 사용하는 모든 것을 뜻한다. 프로그램의 기본이자 핵심 단위가 바로 자료형이다. 계산 프로그램을 만들려면 어떤 것을 계산 할지부터 알아야 하고, 데이터베이스 프로그램을 만들려면 어떤 자료를 저장할지부터 알아야하는 것처럼 기본 중의 기본이라고 할 수 있다. 따라서 자료형을 충분히 이해하지 않고 프로그래밍을 시작하려는 것은 기초 공사가 마무리되지 않은 상태에서 빌딩을 세우는 것과 같다.

자료형에는 어떤 것이 있는지 이 장에서 하나씩 자세하게 알아보자.

02-1 숫자형

숫자형이란?

숫자형(Number)이란 숫자 형태로 이루어진 자료형으로, 우리가 이미 잘 알고 있는 것들이다.

우리가 흔히 사용하는 것들을 생각해 보자. 123과 같은 정수, 12.34와 같은 실수, 드물게 사용하긴 하지만 8진수나 16진수 같은 것들도 있다.

아래 표는 숫자들이 파이썬에서 어떻게 사용되는지를 간략하게 보여 준다.

항목	사용 예
정수	123, -345, 0
실수	123.45, -1234.5, 3.4e10
8진수	0o34, 0o25
16진수	0x2A, 0xFF

이제 이런 숫자들을 파이썬에서는 어떻게 만들고 사용하는지 자세히 알아보자.

숫자형은 어떻게 만들고 사용할까?

정수형

정수형(Integer)이란 말 그대로 정수를 뜻하는 자료형을 말한다. 다음 예는 양의 정수와 음의 정수, 숫자 0을 변수 a에 대입하는 예이다.

```
>>> a = 123
>>> a = -178
>>> a = 0
```

실수형

파이썬에서 실수형(Floating-point)은 소수점이 포함된 숫자를 말한다. 다음 예는 실수를 변수 a에 대입하는 예이다.

```
>>> a = 1.2
>>> a = -3.45
```

위의 방식은 우리가 일반적으로 볼 수 있는 실수형의 소수점 표현 방식이다.

```
>>> a = 4.24E10
>>> a = 4.24e-10
```

위의 방식은 “컴퓨터식 지수 표현 방식”으로 파이썬에서는 4.24e10 또는 4.24E10처럼 표현한다(e와 E 둘 중 어느 것을 사용해도 무방하다). 여기서 4.24E10은 4.24×10^{10} , 4.24e-10은 4.24×10^{-10} 을 의미한다.

8진수와 16진수

8진수(Octal)를 만들기 위해서는 숫자가 0o 또는 0O(숫자 0 + 알파벳 소문자 o 또는 대문자 O)로 시작하면 된다.

```
>>> a = 0o177
```

16진수(Hexadecimal)를 만들기 위해서는 0x로 시작하면 된다.

```
>>> a = 0x8fff
>>> b = 0xABCD
```

8진수나 16진수는 파이썬에서 잘 사용하지 않는 형태의 숫자 자료형이니 간단히 눈으로 익히고 넘어가자.

숫자형을 활용하기 위한 연산자

사칙연산

프로그래밍을 한 번도 해본 적이 없는 독자라도 사칙연산(+, -, *, /)은 알고 있을 것이다. 파이썬 역시 계산기와 마찬가지로 아래의 연산자를 이용해 사칙연산을 수행한다.

```
>>> a = 3
>>> b = 4
>>> a + b
7
>>> a * b
12
>>> a / b
0.75
```

[파이썬 2.7에서 3/4를 실행하면 어떻게 될까?]

파이썬 2.7의 경우 위 사칙연산 예제의 a/b 를 실행하면 0.75가 아닌 0이 출력된다. 파이썬 2.7은 정수형끼리 나눌 경우 정수로만 결과값을 리턴하기 때문이다. 만약 위 예제와 동일한 결과값을 얻고 싶다면 $a/(b*1.0)$ 처럼 b를 강제로 실수형으로 변환해야 한다. 이 책에서 사용하는 파이썬 3은 위의 사칙연산 예제에서 볼 수 있듯이 실수형으로 따로 변환해 줄 필요가 없다.

x의 y제곱을 나타내는 ** 연산자

다음으로 알아야 할 연산자로 **라는 연산자가 있다. 이 연산자는 $x^{**} y$ 처럼 사용되었을 때 x의 y제곱(xy) 값을 리턴한다. 다음의 예를 통해 알아보자.

```
>>> a = 3
>>> b = 4
>>> a ** b
81
```

나눗셈 후 나머지를 반환하는 % 연산자

프로그래밍을 처음 접하는 독자라면 % 연산자는 본 적이 없을 것이다. %는 나눗셈의 나머지 값을 반환하는 연산자이다. 7을 3으로 나누면 나머지는 1이 될 것이고 3을 7로 나누면 나머지는 3이 될 것이다. 다음의 예로 확인해 보자.

```
>>> 7 % 3
1
>>> 3 % 7
3
```

나눗셈 후 몫을 반환하는 // 연산자

/ 연산자를 사용하여 7 나누기 4를 하면 그 결과는 예상대로 1.75가 된다.

```
>>> 7 / 4
1.75
```

이번에는 나눗셈 후 몫을 반환하는 // 연산자를 사용한 경우를 보자.

```
>>> 7 // 4
1
```

1.75에서 몫에 해당되는 정수값 1만 리턴되는 것을 확인할 수 있다.

// 연산자를 사용할 때는 한가지 주의해야 할 점이 있다. 그것은 다음처럼 음수에 // 연산자를 적용하는 경우이다.

```
>>> -7 / 4
-1.75
>>> -7 // 4
-2
```

-1.75라는 실수에서 정수 부분을 리턴하면 -1이 되어야 할 것 같지만 -7 // 4의 결과는 -2가 되었다. 이렇게 되는 이유는 // 연산자는 나눗셈의 결과값보다 작은 정수 중, 가장 큰 정수를 리턴하기 때문이다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-1>)

[문제1] 점수의 평균

홍길동씨의 과목별 점수는 각각 다음과 같다.

과목	점수
국어	80
영어	75
수학	55

과목 점수

홍길동씨의 평균점수를 구하시오.

[문제2] 나눗셈의 몫

17을 3으로 나누었을 때 그 몫을 구하시오.

[문제3] 나눗셈의 나머지

17을 3으로 나누었을 때 그 나머지 값을 구하시오.

[문제4] 자연수의 홀짝

주어진 자연수가 홀수인지 짝수인지 판별할 수 있는 방법에 대해서 말해보자.

02-2 문자열 자료형

문자열이란?

문자열(String)이란 문자, 단어 등으로 구성된 문자들의 집합을 의미한다. 예를 들어 다음과 같은 것들이 문자열이다.

```
"Life is too short, You need Python"
"a"
"123"
```

위의 문자열 예문을 보면 모두 큰따옴표(“ ”)로 둘러싸여 있다.”123은 숫자인데 왜 문자열이지?“라는 의문이 드는 독자도 있을 것이다. 따옴표로 둘러싸여 있으면 모두 문자열이라고 보면 된다.

문자열은 어떻게 만들고 사용할까?

위의 예에서는 문자열을 만들 때 큰따옴표(“ ”)만을 사용했지만 이 외에도 문자열을 만드는 방법은 3가지가 더 있다. 파이썬에서 문자열을 만드는 방법은 총 4가지이다.

1. 큰따옴표로 양쪽 둘러싸기

```
"Hello World"
```

2. 작은따옴표로 양쪽 둘러싸기

```
'Python is fun'
```

3. 큰따옴표 3개를 연속으로 써서 양쪽 둘러싸기

```
"""Life is too short, You need python"""
```

4. 작은따옴표 3개를 연속으로 써서 양쪽 둘러싸기

```
'''Life is too short, You need python'''
```

단순함이 자랑인 파이썬이 문자열을 만드는 방법은 왜 4가지나 가지게 되었을까? 그 이유에 대해서 알아보자.

문자열 안에 작은따옴표나 큰따옴표를 포함시키고 싶을 때

문자열을 만들어 주는 주인공은 작은따옴표(')와 큰따옴표(")이다. 그런데 문자열 안에도 작은따옴표와 큰따옴표가 들어 있어야 할 경우가 있다. 이때는 좀 더 특별한 기술이 필요하다. 예제를 하나씩 살펴보면서 원리를 익혀 보도록 하자.

1) 문자열에 작은따옴표 ('') 포함시키기

```
Python's favorite food is perl
```

위와 같은 문자열을 food라는 변수에 저장하고 싶다고 가정하자. 문자열 중 Python's에 작은따옴표(')가 포함되어 있다.

이럴 때는 다음과 같이 문자열을 큰따옴표("")로 둘러싸야 한다. 큰따옴표 안에 들어 있는 작은따옴표는 문자열을 나타내기 위한 기호로 인식되지 않는다.

```
>>> food = "Python's favorite food is perl"
```

프롬프트에 food를 입력해서 결과를 확인하자. 변수에 저장된 문자열이 그대로 출력되는 것을 볼 수 있다.

```
>>> food
"Python's favorite food is perl"
```

시험삼아 다음과 같이 큰따옴표("")가 아닌 작은따옴표(')로 문자열을 둘러싼 후 다시 실행해 보자. 'Python'이 문자열로 인식되어 구문 오류(SyntaxError)가 발생할 것이다.

```
>>> food = 'Python's favorite food is perl'
      File "<stdin>", line 1
          food = 'Python's favorite food is perl'
          ^
SyntaxError: invalid syntax
```

2) 문자열에 큰따옴표 ("") 포함시키기

```
"Python is very easy." he says.
```

위와 같이 큰따옴표("")가 포함된 문자열이라면 어떻게 해야 큰따옴표가 제대로 표현될까? 다음과

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

같이 문자열을 작은따옴표(')로 둘러싸면 된다.

```
>>> say = '"Python is very easy." he says.'
```

이렇게 작은따옴표(') 안에 사용된 큰따옴표(")는 문자열을 만드는 기호로 인식되지 않는다.

3) \(백슬래시)를 이용해서 작은따옴표(')와 큰따옴표("")를 문자열에 포함시키기

```
>>> food = 'Python\'s favorite food is perl'
>>> say = "\"Python is very easy.\" he says."
```

작은따옴표(')나 큰따옴표("")를 문자열에 포함시키는 또 다른 방법은 \(백슬래시)를 이용하는 것이다. 즉, 백슬래시(\)를 작은따옴표(')나 큰따옴표("") 앞에 삽입하면 \(백슬래시) 뒤의 작은따옴표(')나 큰따옴표("")는 문자열을 둘러싸는 기호의 의미가 아니라 문자 ('), (") 그 자체를 뜻하게 된다.

어떤 방법을 사용해서 문자열 안에 작은따옴표(')와 큰따옴표("")를 포함시킬지는 각자의 선택이다. 대화형 인터프리터를 실행시킨 후 위의 예문들을 꼭 직접 작성해 보도록 하자.

여러 줄인 문자열을 변수에 대입하고 싶을 때

문자열이 항상 한 줄짜리만 있는 것은 아니다. 다음과 같이 여러 줄의 문자열을 변수에 대입하려면 어떻게 처리해야 할까?

```
Life is too short
You need python
```

1) 줄을 바꾸기 위한 이스케이프 코드 \n 삽입하기

```
>>> multiline = "Life is too short\nYou need python"
```

위의 예처럼 줄바꿈 문자인 \n을 삽입하는 방법이 있지만 읽기에 불편하고 줄이 길어지는 단점이 있다.

2) 연속된 작은따옴표 3개(''') 또는 큰따옴표 3개(""""") 이용

위 1번의 단점을 극복하기 위해 파이썬에서는 다음과 같이 작은따옴표 3개("") 또는 큰따옴표 3개("""")를 이용한다.

```
>>> multiline='''  
... Life is too short  
... You need python  
... '''
```

작은따옴표 3개를 사용한 경우

```
>>> multiline="""  
... Life is too short  
... You need python  
... """
```

큰따옴표 3개를 사용한 경우

`print(multiline)`을 입력해서 어떻게 출력되는지 확인해 보자.

```
>>> print(multiline)  
Life is too short  
You need python
```

두 경우 모두 결과는 동일하다. 위 예에서도 확인할 수 있듯이 문자열이 여러 줄인 경우 이스케이프 코드를 쓰는 것보다 따옴표를 연속해서 쓰는 것이 훨씬 깔끔하다.

[이스케이프 코드란?]

문자열 예제에서 여러 줄의 문장을 처리할 때 백슬래시 문자와 소문자 n을 조합한 \n 이스케이프 코드를 사용했다. 이스케이프 코드란 프로그래밍할 때 사용할 수 있도록 미리 정의해 둔 “문자 조합”이다. 주로 출력물을 보기 좋게 정렬하는 용도로 이용된다. 몇 가지 이스케이프 코드를 정리하면 다음과 같다.

코드	설명
\n	개행 (줄바꿈)
\t	수평 탭
\\"	문자 “\”
\'	단일 인용부호(‘)
\\"	이중 인용부호(“)

코드	설명
\r	캐리지 리턴
\f	폼 피드
\a	벨 소리
\b	백 스페이스
\000	널문자

이중에서 활용빈도가 높은 것은 \n, \t, \\, \', \"이다. 나머지는 프로그램에서 잘 사용되지 않는다.

문자열 연산하기

파이썬에서는 문자열을 더하거나 곱할 수 있다. 이는 다른 언어에서는 쉽게 찾아볼 수 없는 재미 있는 기능으로, 우리의 생각을 그대로 반영해 주는 파이썬만의 장점이라고 할 수 있다. 문자열을 더하거나 곱하는 방법에 대해 알아보자.

1) 문자열 더해서 연결하기(Concatenation)

```
>>> head = "Python"
>>> tail = " is fun!"
>>> head + tail
'Python is fun!'
```

위 소스 코드에서 세 번째 줄을 보자. 복잡하게 생각하지 말고 눈에 보이는 대로 생각해 보자. “Python”이라는 head 변수와 ” is fun!”라는 tail 변수를 더한 것이다. 결과는 ‘Python is fun!’이다. 즉, head와 tail 변수가 +에 의해 합쳐진 것이다.

직접 실행해 보고 결과값이 제시한 것과 똑같이 나오는지 확인해 보자.

2) 문자열 곱하기

```
>>> a = "python"
>>> a * 2
'pythonpython'
```

위 소스 코드에서 *의 의미는 우리가 일반적으로 사용하는 숫자 곱하기의 의미와는 다르다. 위 소스 코드에서 a * 2라는 문장은 a를 두 번 반복하라는 뜻이다. 즉, *는 문자열의 반복을 뜻하는 의미로 사용되었다. 굳이 코드의 의미를 설명할 필요가 없을 정도로 직관적이다.

3) 문자열 곱하기 응용

문자열 곱하기를 좀 더 응용해 보자. 다음과 같은 소스를 에디터로 작성해 실행해 보자.

```
# multistring.py

print("=" * 50)
print("My Program")
print("=" * 50)
```

결과값은 다음과 같이 나타날 것이다.

```
C:\Users>cd C:\doit
C:\doit>python multistring.py
=====
My Program
=====
```

이런 식의 표현은 앞으로 자주 사용하게 될 것이다. 프로그램을 만들어 실행시켰을 때 출력되는 화면 제일 상단에 프로그램 제목을 이와 같이 표시하면 보기 좋지 않겠는가?

문자열 인덱싱과 슬라이싱

인덱싱(Indexing)이란 무엇인가를 “가리킨다”는 의미이고, 슬라이싱(Slicing)은 무엇인가를 “잘라낸다”는 의미이다. 이런 의미를 생각하면서 다음 내용을 살펴보자.

문자열 인덱싱이란?

```
>>> a = "Life is too short, You need Python"
```

위 소스 코드에서 변수 a에 저장한 문자열의 각 문자마다 번호를 매겨 보면 다음과 같다.

```
Life is too short, You need Python
0      1      2      3
0123456789012345678901234567890123
```

“Life is too short, You need Python”이라는 문자열에서 L은 첫 번째 자리를 뜻하는 숫자인 0, 바로 다음인 i는 1 이런 식으로 계속 번호를 붙인 것이다. 중간에 있는 short의 s는 12가 된다.

이제 다음 예를 실행해 보자.

```
>>> a = "Life is too short, You need Python"
>>> a[3]
'e'
```

a[3]이 뜻하는 것은 a라는 문자열의 네 번째 문자인 e를 말한다. 프로그래밍을 처음 접하는 독자라면 a[3]에서 3이란 숫자가 왜 네 번째 문자를 뜻하는 것인지 의아할 수도 있다. 사실 이 부분이 필자도 가끔 헷갈리는 부분인데, 이렇게 생각하면 쉽게 알 수 있을 것이다.

“파이썬은 0부터 숫자를 센다.”

고로 위의 문자열을 파이썬은 다음과 같이 바라보고 있다.

```
a[0]:'L', a[1]:'i', a[2]:'f', a[3]:'e', a[4]:' ', ...
```

0부터 숫자를 센다는 것이 처음에는 익숙하지 않겠지만 계속 사용하다 보면 자연스러워질 것이다. 위의 예에서 볼 수 있듯이 a[번호]는 문자열 내 특정한 값을 뽑아내는 역할을 한다. 이러한 것을 인덱싱이라고 한다.

문자열 인덱싱 활용하기

인덱싱 예를 몇 가지 더 보도록 하자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> a = "Life is too short, You need Python"
>>> a[0]
'L'
>>> a[12]
's'
>>> a[-1]
'n'
```

앞의 a[0]과 a[12]는 쉽게 이해할 수 있는데 마지막의 a[-1]이 뜻하는 것은 뭘까? 눈치 빠른 독자는 이미 알아챘겠지만 문자열을 뒤에서부터 읽기 위해서 마이너스(-) 기호를 붙이는 것이다. 즉 a[-1]은 뒤에서부터 세어 첫 번째가 되는 문자를 말한다. a는 “Life is too short, You needPython”이라는 문장이므로 뒤에서부터 첫 번째 문자는 가장 마지막 문자인 ‘n’이다.

뒤에서부터 첫 번째 문자를 표시할 때도 0부터 세어 “a[-0]이라고 해야 하지 않을까?”라는 의문이 들 수도 있겠지만 잘 생각해 보자. 0과 -0은 똑같은 것이기 때문에 a[-0]은 a[0]과 똑같은 값을 보여 준다.

```
>>> a[-0]
'L'
```

계속해서 몇 가지 예를 더 보자.

```
>>> a[-2]
'o'
>>> a[-5]
'y'
```

위의 첫 번째 예는 뒤에서부터 두 번째 문자를 가리키는 것이고, 두 번째 예는 뒤에서부터 다섯 번째 문자를 가리키는 것이다.

문자열 슬라이싱이란?

그렇다면 “Life is too short, You need Python”이라는 문자열에서 단순히 한 문자만을 뽑아내는 것이 아니라 ‘Life’ 또는 ‘You’ 같은 단어들을 뽑아내는 방법은 없을까?

다음과 같이 하면 된다.

```
>>> a = "Life is too short, You need Python"
>>> b = a[0] + a[1] + a[2] + a[3]
>>> b
'Life'
```

위의 방법처럼 단순하게 접근할 수도 있지만 파이썬에서는 더 좋은 방법을 제공한다. 바로 슬라이싱(Slicing)이라는 기법이다.⁶

위의 예는 슬라이싱 기법으로 다음과 같이 간단하게 처리할 수 있다.

```
>>> a = "Life is too short, You need Python"
>>> a[0:4]
'Lif'
```

`a[0:4]`가 뜻하는 것은 `a`라는 문자열, 즉 “Life is too short, You need Python”이라는 문장에서 0부터 4까지의 문자를 뽑아낸다는 뜻이다. 하지만 다음과 같은 의문이 생길 것이다. `a[0]`은 L, `a[1]`은 i, `a[2]`는 f, `a[3]`은 e니까 `a[0:3]`으로도 Life라는 단어를 뽑아낼 수 있지 않을까? 다음의 예를 보도록 하자.

```
>>> a[0:3]
'Lif'
```

이렇게 되는 이유는 간단하다. `a[시작 번호:끝 번호]`를 지정하면 끝 번호에 해당하는 것은 포함되지 않는다. `a[0:3]`을 수식으로 나타내면 다음과 같다.

```
0 <= a < 3
```

이 수식을 만족하는 `a`는 `a[0]`, `a[1]`, `a[2]`일 것이다. 따라서 `a[0:3]`은 ‘Lif’이고 `a[0:4]`는 ‘Life’가 되는 것이다. 이 부분이 문자열 연산에서 가장 혼동하기 쉬운 부분이니 장 마지막의 연습문제를 많이 풀어 보면서 몸에 익히기 바란다.

문자열을 슬라이싱하는 방법

슬라이싱의 예를 조금 더 보도록 하자.

⁶인텍싱 기법과 슬라이싱 기법은 뒤에서 배울 자료형인 리스트나 튜플에서도 사용할 수 있다.

```
>>> a[0:5]
'Life '
```

위의 예는 $a[0] + a[1] + a[2] + a[3] + a[4]$ 와 동일하다. $a[4]$ 는 공백 문자이기 때문에 'Life'가 아닌 'Life'가 출력되는 것이다. 공백 문자 역시 L, i, f, e 같은 문자와 동일하게 취급되는 것을 잊지 말자. 'Life'와 'Life'는 완전히 다른 문자열이다.

슬라이싱할 때 항상 시작 번호가 '0'일 필요는 없다.

```
>>> a[0:2]
'Li'
>>> a[5:7]
'is'
>>> a[12:17]
'short'
```

$a[\text{시작 번호}:\text{끝 번호}]$ 에서 끝 번호 부분을 생략하면 시작 번호부터 그 문자열의 끝까지 뽑아낸다.

```
>>> a[19:]
'You need Python'
```

$a[\text{시작 번호}:\text{끝 번호}]$ 에서 시작 번호를 생략하면 문자열의 처음부터 끝 번호까지 뽑아낸다.

```
>>> a[:17]
'Life is too short'
```

$a[\text{시작 번호}:\text{끝 번호}]$ 에서 시작 번호와 끝 번호를 생략하면 문자열의 처음부터 끝까지를 뽑아낸다.

```
>>> a[:]
'Life is too short, You need Python'
```

슬라이싱에서도 인덱싱과 마찬가지로 마이너스(-) 기호를 사용할 수 있다.

```
>>> a[19:-7]
'You need'
```

위 소스 코드에서 a[19:-7]이 뜻하는 것은 a[19]에서부터 a[-8]까지를 말한다. 이 역시 a[-7]은 포함하지 않는다.

슬라이싱으로 문자열 나누기

다음은 자주 사용하게 되는 슬라이싱 기법 중 하나이다.

```
>>> a = "20010331Rainy"
>>> date = a[:8]
>>> weather = a[8:]
>>> date
'20010331'
>>> weather
'Rainy'
```

a라는 문자열을 두 부분으로 나누는 기법이다. 동일한 숫자 8을 기준으로 a[:8], a[8:]처럼 사용했다. a[:8]은 a[8]이 포함되지 않고, a[8:]은 a[8]을 포함하기 때문에 8을 기준으로 해서 두 부분으로 나눌 수 있는 것이다. 위의 예에서는 “20010331Rainy”라는 문자열을 날짜를 나타내는 부분인 ‘20010331’과 날씨를 나타내는 부분인 ‘Rainy’로 나누는 방법을 보여준다.

위의 문자열 “20010331Rainy”를 연도인 2001, 월과 일을 나타내는 0331, 날씨를 나타내는 Rainy의 세 부분으로 나누려면 다음과 같이 할 수 있다.

```
>>> a = "20010331Rainy"
>>> year = a[:4]
>>> day = a[4:8]
>>> weather = a[8:]
>>> year
'2001'
>>> day
'0331'
>>> weather
'Rainy'
```

위의 예는 4와 8이란 숫자로 “20010331Rainy”라는 문자열을 세 부분으로 나누는 방법을 보여준다.

지금까지 인덱싱과 슬라이싱에 대해서 살펴보았다. 인덱싱과 슬라이싱은 프로그래밍을 할 때 매우

자주 사용되는 기법이니 꼭 반복해서 연습해 두자.

[“Pithon”이라는 문자열을 “Python”으로 바꾸려면?]

“Pithon”이라는 문자열을 “Python”으로 바꾸려면 어떻게 해야 할까? 제일 먼저 떠오르는 생각은 다음과 같을 것이다.

```
>>> a = "Pithon"
>>> a[1]
'i'
>>> a[1] = 'y'
```

요컨대 a라는 변수에 “Pithon”이라는 문자열을 대입하고 a[1]의 값이 i니까 a[1]을 y로 바꾸어 준다는 생각이다. 하지만 결과는 어떻게 나올까?

당연히 에러가 발생한다. 왜냐하면 문자열의 요소값은 바꿀 수 있는 값이 아니기 때문이다(문자열, 튜플 등 의 자료형은 그 요소값을 변경할 수 없다. 그래서 immutable한 자료형이라고도 부른다). 하지만 앞서 살펴본 슬라이싱 기법을 이용하면 “Pithon”이라는 문자열을 “Python”으로 바꿀 수 있는 방법이 있다.

다음의 예를 보자.

```
>>> a = "Pithon"
>>> a[:1]
'P'
>>> a[2:]
'thon'
>>> a[:1] + 'y' + a[2:]
'Python'
```

위의 예에서 볼 수 있듯이 슬라이싱을 이용하면 ‘Pithon’이라는 문자열을 ‘P’ 부분과 ‘thon’ 부분으로 나눌 수 있기 때문에 그 사이에 ‘y’라는 문자를 추가하여 ‘Python’이라는 새로운 문자열을 만들 수 있다.

문자열 포매팅

문자열에서 또 하나 알아야 할 것으로는 문자열 포매팅(Formatting)이 있다. 이것을 공부하기에 앞서 다음과 같은 문자열을 출력하는 프로그램을 작성했다고 가정해 보자.

“현재 온도는 18도입니다.”

시간이 지나서 20도가 되면 아래와 같은 문장을 출력한다.

“현재 온도는 20도입니다”

위의 두 문자열은 모두 같은데 20이라는 숫자와 18이라는 숫자만 다르다. 이렇게 문자열 내의 특정한 값을 바꿔야 할 경우가 있을 때 이것을 가능하게 해주는 것이 바로 문자열 포매팅 기법이다.

문자열 포매팅이란 쉽게 말해 문자열 내에 어떤 값을 삽입하는 방법이다. 다음의 예들을 직접 실행해 보면서 그 사용법을 알아보자.

문자열 포매팅 따라 하기

1) 숫자 바로 대입

```
>>> "I eat %d apples." % 3
'I eat 3 apples.'
```

위 예제의 결과값을 보면 알겠지만 위의 예제는 문자열 내에 3이라는 정수를 삽입하는 방법을 보여 준다. 문자열 안에서 숫자를 넣고 싶은 자리에 %d라는 문자를 넣어 주고, 삽입할 숫자인 3은 가장 뒤에 있는 % 문자 다음에 써넣었다. 여기서 %d는 문자열 포맷 코드라고 부른다.

2) 문자열 바로 대입

문자열 내에 꼭 숫자만 넣으라는 법은 없다. 이번에는 숫자 대신 문자열을 넣어 보자.

```
>>> "I eat %s apples." % "five"
'I eat five apples.'
```

위 예제에서는 문자열 내에 또 다른 문자열을 삽입하기 위해 앞서 사용했던 문자열 포맷 코드 %d가 아닌 %s를 썼다. 어쩌면 눈치 빠른 독자는 벌써 유추하였을 것이다. 숫자를 넣기 위해서는 %d를 써야 하고, 문자열을 넣기 위해서는 %s를 써야 한다는 사실을 말이다.

※ 문자열을 대입할 때는 앞에서 배웠던 것처럼 큰따옴표나 작은따옴표를 반드시 써주어야 한다.

3) 숫자 값을 나타내는 변수로 대입

```
>>> number = 3
>>> "I eat %d apples." % number
'I eat 3 apples.'
```

1번처럼 숫자를 바로 대입하나 위 예제처럼 숫자 값을 나타내는 변수를 대입하나 결과는 같다.

4) 2개 이상의 값 넣기

그렇다면 문자열 안에 1개가 아닌 여러 개의 값을 넣고 싶을 땐 어떻게 해야 할까?

```
>>> number = 10
>>> day = "three"
>>> "I ate %d apples. so I was sick for %s days." % (number, day)
'I ate 10 apples. so I was sick for three days.'
```

위의 예문처럼 2개 이상의 값을 넣으려면 마지막 % 다음 괄호 안에 콤마(,)로 구분하여 각각의 변수를 넣어 주면 된다.

문자열 포맷 코드

문자열 포매팅 예제에서는 대입시켜 넣는 자료형으로 정수와 문자열을 사용했으나 이 외에도 다양한 것들을 대입시킬 수 있다. 문자열 포맷 코드로는 다음과 같은 것들이 있다.

코드	설명
%s	문자열 (String)
%c	문자 1개(character)
%d	정수 (Integer)
%f	부동소수 (floating-point)
%o	8진수
%x	16진수
%%	Literal % (문자 % 자체)

여기서 재미있는 것은 %s 포맷 코드인데, 이 코드는 어떤 형태의 값이든 변환해 넣을 수 있다. 무슨 말인지 예를 통해 확인해 보자.

```
>>> "I have %s apples" % 3
'I have 3 apples'
>>> "rate is %s" % 3.234
'rate is 3.234'
```

3을 문자열 안에 삽입하려면 %d를 사용하고, 3.234를 삽입하려면 %f를 사용해야 한다. 하지만 %s를 사용하면 이런 것을 생각하지 않아도 된다. 왜냐하면 %s는 자동으로 % 뒤에 있는 값을 문자열로 바꾸기 때문이다.

[포매팅 연산자 %d와 %를 같이 쓸 때는 %%를 쓴다]

```
>>> "Error is %d%." % 98
```

위 예문의 결과값으로 당연히 “Error is 98%.”가 출력될 것이라고 예상하겠지만 파이썬은 값이 올바르지 않다는 값 오류(Value Error) 메시지를 보여 준다.

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: incomplete format
```

이유는 문자열 포맷 코드인 %d와 %가 같은 문자열 내에 존재하는 경우, %를 나타내려면 반드시 %%로 써야 하는 법칙이 있기 때문이다. 이 점은 꼭 기억해 두어야 한다. 하지만 문자열 내에 %d 같은 포매팅 연산자가 없으면 %는 홀로 쓰여도 상관이 없다.

따라서 위 예를 제대로 실행하려면 다음과 같이 해야 한다.

```
>>> "Error is %d%%." % 98
'Error is 98%.'
```

포맷 코드와 숫자 함께 사용하기

위에서 보았듯이 %d, %s 등의 포맷 코드는 문자열 내에 어떤 값을 삽입하기 위해서 사용된다. 하지만 포맷 코드를 숫자와 함께 사용하면 더 유용하게 사용할 수 있다. 다음의 예를 보고 따라해보자.

1) 정렬과 공백

```
>>> "%10s" % "hi"
      hi'
```

앞의 예문에서 “%10s”의 의미는 전체 길이가 10개인 문자열 공간에서 hi를 오른쪽으로 정렬하고 그 앞의 나머지는 공백으로 넘겨 두라는 의미이다.

그렇다면 반대쪽인 왼쪽 정렬은 “%-10s”가 될 것이다. 확인해 보자.

```
>>> "%-10s{jane.}" % 'hi'
'hi      jane.'
```

hi를 왼쪽으로 정렬하고 나머지는 공백으로 채웠음을 볼 수 있다.

2) 소수점 표현하기

```
>>> "%0.4f" % 3.42134234
'3.4213'
```

3.42134234를 소수점 네 번째 자리까지만 나타내고 싶은 경우에는 위와 같이 사용한다. 즉, 여기서 ‘.’의 의미는 소수점 포인트를 말하고 그 뒤의 숫자 4는 소수점 뒤에 나올 숫자의 개수를 말한다. 다음의 예를 살펴보자.

```
>>> "%10.4f" % 3.42134234
'      3.4213'
```

위의 예는 3.42134234라는 숫자를 소수점 네 번째 자리까지만 표시하고 전체 길이가 10개인 문자열 공간에서 오른쪽으로 정렬하는 예를 보여준다.

format 함수를 이용한 포매팅

문자열의 format 함수를 이용하면 좀 더 발전된 스타일로 문자열 포맷을 지정할 수 있다. 앞에서 살펴본 문자열 포매팅 예제들을 format 함수를 이용해서 바꾸면 다음과 같다.

숫자 바로 대입하기

```
>>> "I eat {0} apples".format(3)
'I eat 3 apples'
```

“I eat {0} apples” 문자열 중 {0} 부분이 숫자 3으로 바뀌었다.

문자열 바로 대입하기

```
>>> "I eat {0} apples".format("five")
'I eat five apples'
```

문자열의 {0} 항목이 five라는 문자열로 바뀌었다.

숫자 값을 가진 변수로 대입하기

```
>>> number = 3
>>> "I eat {0} apples".format(number)
'I eat 3 apples'
```

문자열의 {0} 항목이 number 변수의 값인 3으로 바뀌었다.

2개 이상의 값 넣기

```
>>> number = 10
>>> day = "three"
>>> "I ate {0} apples. so I was sick for {1} days.".format(number, day)
'I ate 10 apples. so I was sick for three days.'
```

2개 이상의 값을 넣을 경우 문자열의 {0}, {1}과 같은 인덱스 항목들이 format 함수의 입력값들로 순서에 맞게 바뀐다. 즉, 위 예에서 {0}은 format 함수의 첫 번째 입력값인 number로 바뀌고 {1}은 format 함수의 두 번째 입력값인 day로 바뀐다.

이름으로 넣기

```
>>> "I ate {number} apples. so I was sick for {day} days.".format(number=10,
day=3)
'I ate 10 apples. so I was sick for 3 days.'
```

위 예에서 볼 수 있듯이 {0}, {1}과 같은 인덱스 항목 대신 더 편리한 {name} 형태를 이용하는 방법도 있다. {name} 형태를 이용할 경우 format 함수의 입력값에는 반드시 name=value와 같은 형태의 입력값이 있어야만 한다. 위 예는 문자열의 {number}, {day}가 format 함수의 입력값인 number=10, day=3 값으로 각각 바뀌는 것을 보여 주고 있다.

인덱스와 이름을 혼용해서 넣기

```
>>> "I ate {0} apples. so I was sick for {day} days.".format(10, day=3)
'I ate 10 apples. so I was sick for 3 days.'
```

위와 같이 인덱스 항목과 name=value 형태를 혼용하는 것도 가능하다.

왼쪽 정렬

```
>>> "{0:<10}".format("hi")
'hi      '
```

:<10 표현식을 이용하면 치환되는 문자열을 왼쪽으로 정렬하고 문자열의 총 자릿수를 10으로 맞출 수 있다.

오른쪽 정렬

```
>>> "{0:>10}".format("hi")
'      hi'
```

오른쪽 정렬은 :< 대신 :>을 이용하면 된다. 화살표 방향을 생각하면 어느 쪽으로 정렬이 되는지 금방 알 수 있을 것이다.

가운데 정렬

```
>>> "{0:^10}".format("hi")
'    hi    '
```

:^ 기호를 이용하면 가운데 정렬도 가능하다.

공백 채우기

```
>>> "{0:={}10)".format("hi")
'=====hi===='
>>> "{0:!<10)".format("hi")
'hi!!!!!!!'
```

정렬 시 공백 문자 대신에 지정한 문자 값으로 채워 넣는 것도 가능하다. 채워 넣을 문자 값은 정렬 문자인 <, >, ^ 바로 앞에 넣어야 한다. 위 예에서 첫 번째 예제는 가운데(~)로 정렬하고 빈 공간을 =문자로 채웠고, 두번째 예제는 왼쪽(<)으로 정렬하고 빈 공간을 !문자로 채웠다.

소수점 표현하기

```
>>> y = 3.42134234
>>> "{0:0.4f)".format(y)
'3.4213'
```

위 예는 format 함수를 이용해 소수점을 4자리까지만 표현하는 방법을 보여 준다. 이전에 살펴보았던 표현식 0.4f가 그대로 이용된 걸 알 수 있다.

```
>>> "{0:10.4f}".format(y)
'      3.4213'
```

위와 같이 자릿수를 10으로 맞출 수도 있다. 역시 58쪽에서 살펴보았던 “10.4f”의 표현식이 그대로 이용된 걸 알 수 있다.

{ 또는 } 문자 표현하기

```
>>> "{{ and }}".format()
'{ and }'
```

format 함수를 이용해 문자열 포매팅을 할 경우 {나 }와 같은 중괄호(brace) 문자를 포매팅 문자가 아닌 문자 그대로 사용하고 싶은 경우에는 위 예의 {{와 }}처럼 2개를 연속해서 사용하면 된다.

f 문자열 포매팅

파이썬 3.6 버전부터는 f 문자열 포매팅 기능을 사용할 수 있다. 파이썬 2.7이나 파이썬 3.6 미만 버전에서는 사용할 수 없는 기능이므로 주의해야 한다.

다음과 같이 문자열 앞에 f 접두사를 붙이면 f 문자열 포매팅 기능을 사용할 수 있다.

```
>>> name = '홍길동'
>>> age = 30
>>> f'나의 이름은 {name}입니다. 나이는 {age}입니다.'
'나의 이름은 홍길동입니다. 나이는 30입니다.'
```

f 문자열 포매팅은 표현식을 지원하기 때문에 아래와 같은 것도 가능하다.

```
>>> age = 30
>>> f'나는 내년이면 {age+1}살이 된다.'
'나는 내년이면 31살이 된다.'
```

딕셔너리는 f 문자열에서 다음과 같이 사용 가능하다.

```
>>> d = {'name': '홍길동', 'age':30}
>>> f'나의 이름은 {d["name"]}입니다. 나이는 {d["age"]}입니다.'
'나의 이름은 홍길동입니다. 나이는 30입니다.'
```

정렬은 다음과 같이 할 수 있다.

```
>>> f'{\"hi\":<10}' # 왼쪽 정렬
'hi      '
>>> f'{\"hi\":>10}' # 오른쪽 정렬
'      hi'
>>> f'{\"hi\":^10}' # 가운데 정렬
'  hi  '
```

공백 채우기는 다음과 같이 할 수 있다.

```
>>> f'{\"hi\":=^10}' # 가운데 정렬하고 '=' 문자로 공백 채우기
'=====hi===='
>>> f'{\"hi\":!<10}' # 왼쪽 정렬하고 '!' 문자로 공백 채우기
'hi!!!!!!!'
```

소수점은 다음과 같이 표현할 수 있다.

```
>>> y = 3.42134234
>>> f'{y:0.4f}' # 소수점 4자리까지만 표현
'3.4213'
>>> f'{y:10.4f}' # 소수점 4자리까지 표현하고 총 자리수를 10으로 맞춤
'    3.4213'
```

f 문자열에서 '{' 나 '}' 문자를 사용하려면 다음과 같이 해야 한다.

```
>>> f'{{ and }}'
'{ and }'
```

지금까지는 문자열을 가지고 할 수 있는 기본적인 것들에 대해 알아보았다. 이제부터는 문자열을 좀 더 자유자재로 다루기 위해 공부해야 할 것들을 설명할 것이다. 지겹다면 잠시 책을 접고 휴식을 취하도록 하자.

문자열 관련 함수들

문자열 자료형은 자체적으로 가지고 있는 함수들이 있다. 이 함수들은 다른말로 문자열 내장함수라고 말한다. 이 내장함수를 사용하려면 문자열 변수 이름 뒤에 '.'를 붙인 다음에 함수 이름을 써주면 된다. 이제 문자열이 자체적으로 가지고 있는 내장 함수들에 대해서 알아보자.

문자 개수 세기(count)

```
>>> a = "hobby"
>>> a.count('b')
2
```

문자열 중 문자 b의 개수를 반환한다.

위치 알려주기1(find)

```
>>> a = "Python is the best choice"
>>> a.find('b')
14
>>> a.find('k')
-1
```

문자열 중 문자 b가 처음으로 나온 위치를 반환한다. 만약 찾는 문자나 문자열이 존재하지 않는다면 -1을 반환한다.

※ 파이썬은 숫자를 0부터 세기 때문에 b의 위치는 15가아닌 14가 된다.

위치 알려주기2(index)

```
>>> a = "Life is too short"
>>> a.index('t')
8
>>> a.index('k')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: substring not found
```

문자열 중 문자 t가 맨 처음으로 나온 위치를 반환한다. 만약 찾는 문자나 문자열이 존재하지 않는다면 오류를 발생시킨다. 앞의 find 함수와 다른 점은 문자열 안에 존재하지 않는 문자를 찾으면 오류가 발생한다는 점이다.

문자열 삽입(join)

```
>>> a= ","
>>> a.join('abcd')
'a,b,c,d'
```

abcd라는 문자열의 각각의 문자 사이에 변수 a의 값인 ','를 삽입한다.

보통 삽입할 문자를 변수로 지정하지 않고 다음처럼 많이들 사용한다.

```
>>> ",".join('abcd')
'a,b,c,d'
```

join 함수는 문자열 뿐만 아니라 앞으로 배우게 될 리스트나 튜플을 입력으로 사용할 수 있다. (리스트와 튜플은 곧 배우게 될 내용이니 여기서는 잠시 눈으로만 살펴보기로 하자.)

join 함수의 입력으로 리스트를 사용하는 예는 다음과 같다.

```
>>> ",".join(['a', 'b', 'c', 'd'])
'a,b,c,d'
```

소문자를 대문자로 바꾸기(upper)

```
>>> a = "hi"
>>> a.upper()
'HI'
```

upper() 함수는 소문자를 대문자로 바꾸어 준다. 만약 문자열이 이미 대문자라면 아무런 변화도 일어나지 않을 것이다.

대문자를 소문자로 바꾸기(lower)

```
>>> a = "HI"
>>> a.lower()
'hi'
```

lower() 함수는 대문자를 소문자로 바꾸어 준다.

왼쪽 공백 지우기(lstrip)

```
>>> a = " hi "
>>> a.lstrip()
'hi '
```

문자열 중 가장 왼쪽에 있는 한 칸 이상의 연속된 공백들을 모두 지운다. lstrip에서 l은 left를 의미한다.

오른쪽 공백 지우기(rstrip)

```
>>> a= " hi "
>>> a.rstrip()
' hi'
```

문자열 중 가장 오른쪽에 있는 한 칸 이상의 연속된 공백들을 모두 지운다. rstrip에서 r은 right를 의미한다.

양쪽 공백 지우기(strip)

```
>>> a = " hi "
>>> a.strip()
'hi'
```

문자열 양쪽에 있는 한 칸 이상의 연속된 공백들을 모두 지운다.

문자열 바꾸기(replace)

```
>>> a = "Life is too short"
>>> a.replace("Life", "Your leg")
'Your leg is too short'
```

replace(바뀌게 될 문자열, 바꿀 문자열)처럼 사용해서 문자열 내의 특정한 값을 다른 값으로 치환해 준다.

문자열 나누기(split)

```
>>> a = "Life is too short"
>>> a.split()
['Life', 'is', 'too', 'short']
>>> a = "a:b:c:d"
>>> a.split(':')
['a', 'b', 'c', 'd']
```

a.split()처럼 괄호 안에 아무런 값도 넣어 주지 않으면 공백(스페이스, 탭, 엔터등)을 기준으로 문자열을 나누어 준다. 만약 a.split('::')처럼 괄호 안에 특정한 값이 있을 경우에는 괄호 안의 값을 구분자로 해서 문자열을 나누어 준다. 이렇게 나눈 값은 리스트에 하나씩 들어가게 된다. ['Life', 'is', 'too', 'short']나 ['a', 'b', 'c', 'd']는 리스트라는 것인데 다음 절에서 자세히 알아볼 것이니 여기서는 너무 신경 쓰지 말도록 하자.

위에서 소개한 문자열 관련 함수들은 문자열 처리에서 사용 빈도가 매우 높은 것들이고 유용한 것들이다. 이 외에도 몇 가지가 더 있지만 자주 사용되는 것들은 아니다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-2>)

[문제1] 문자열 출력 1

다음과 같은 문자열을 출력하시오.

"점프 투 파이썬" 문제를 풀어보자

[문제2] 문자열 출력 2

다음과 같은 문자열을 출력하시오.

Life is too short
You need Python

[문제3] 공백 추가

다음과 같은 6개의 문자로 이루어진 "PYTHON"이라는 문자열이 있다.

PYTHON

이 "PYTHON"이라는 문자열 앞에 공백 24개를 추가하여 다음과 같은 형태의 30자리의 문자열로 만드시오.

PYTHON

[문제4] 문자열 나누기

홍길동 씨의 주민등록번호는 881120-1068234이다. 홍길동씨의 주민등록번호를 연월일(YYYYMMDD) 부분과 그 뒤의 숫자 부분으로 나누어 출력해 보자.

[문제5] 문자열 인덱싱

주민등록번호 뒷자리의 맨 첫 번째 숫자는 성별을 나타낸다. 주민등록번호에서 성별을 나타내는 숫자를 출력해 보자.

```
>>> pin = "881120-1068234"
```

[문제6] 문자열 변경

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

다음과 같은 문자열이 있다.

```
1980M1120
```

위 문자열을 다음과 같이 변경하시오.

```
M19801120
```

[문제7] 문자열 찾기

다음의 문자열에서 “short”라는 문자열이 시작되는 인덱스를 구하시오.

```
Life is too short, you need python
```

[문제8] 문자열 바꾸기1

다음과 같은 문자열이 있다.

```
a:b:c:d
```

문자열의 replace 함수를 이용하여 위 문자열을 다음과 같이 고치시오.

```
a#b#c#d
```

[문제9] 문자열 바꾸기2

다음과 같은 문자열이 있다.

```
a:b:c:d
```

문자열의 split와 join 함수를 이용하여 위 문자열을 다음과 같이 고치시오.

```
a#b#c#d
```

02-3 리스트 자료형

지금까지 우리는 숫자와 문자열에 대해서 알아보았다. 하지만 숫자와 문자열만으로 프로그래밍을 하기엔 부족한 점이 많다. 예를 들어 1부터 10까지의 숫자 중 홀수 모음인 1, 3, 5, 7, 9라는 집합을 생각해 보자. 이런 숫자 모음을 숫자나 문자열로 표현하기는 쉽지 않다. 파이썬에는 이러한 불편함을 해소할 수 있는 자료형이 존재한다. 그것이 바로 이 절에서 공부하게 될 리스트(List)이다.

리스트는 어떻게 만들고 사용할까?

리스트를 이용하면 1, 3, 5, 7, 9라는 숫자 모음을 다음과 같이 간단하게 표현할 수 있다.

```
>>> odd = [1, 3, 5, 7, 9]
```

리스트를 만들 때는 위에서 보는 것과 같이 대괄호([])로 감싸 주고 각 요소값들은 쉼표(,)로 구분해 준다.

```
리스트명 = [요소1, 요소2, 요소3, ...]
```

여러 가지 리스트의 생김새를 살펴보면 다음과 같다.

```
>>> a = []
>>> b = [1, 2, 3]
>>> c = ['Life', 'is', 'too', 'short']
>>> d = [1, 2, 'Life', 'is']
>>> e = [1, 2, ['Life', 'is']]
```

리스트는 a처럼 아무것도 포함하지 않는, 비어 있는 리스트([])일 수도 있고 b처럼 숫자를 요소값으로 가질 수도 있고 c처럼 문자열을 요소값으로 가질 수도 있다. 또한 d처럼 숫자와 문자열을 함께 요소값으로 가질 수도 있으며 e처럼 리스트 자체를 요소값으로 가질 수도 있다. 즉, 리스트 안에는 어떠한 자료형도 포함시킬 수 있다.

※ 비어 있는 리스트는 a = list()로 생성할 수도 있다.

리스트의 인덱싱과 슬라이싱

리스트도 문자열처럼 인덱싱과 슬라이싱이 가능하다. 백문이 불여일견. 말로 설명하는 것보다 직접 예를 실행해 보면서 리스트의 기본 구조를 이해하는 것이 쉽다. 대화형 인터프리터로 따라 하며

확실하게 이해하자.

리스트의 인덱싱

리스트 역시 문자열처럼 인덱싱을 적용할 수 있다. 먼저 a 변수에 [1, 2, 3]이라는 값을 설정 한다.

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
```

a[0]은 리스트 a의 첫 번째 요소값을 말한다.

```
>>> a[0]
1
```

아래의 예는 리스트의 첫 번째 요소인 a[0]과 세 번째 요소인 a[2]의 값을 더한 것이다.

```
>>> a[0] + a[2]
4
```

이것은 $1 + 3$ 으로 해석되어 4라는 값을 출력한다.

문자열을 공부할 때 이미 살펴보았지만 파이썬은 숫자를 0부터 세기 때문에 a[1]이 리스트 a의 첫 번째 요소가 아니라 a[0]이 리스트 a의 첫 번째 요소임을 명심하자. a[-1]은 문자열에서와 마찬가지로 리스트 a의 마지막 요소값을 말한다.

```
>>> a[-1]
3
```

이번에는 아래의 예처럼 리스트 a를 숫자 1, 2, 3과 또 다른 리스트인 ['a', 'b', 'c']를 포함하도록 만들어 보자.

```
>>> a = [1, 2, 3, ['a', 'b', 'c']]
```

다음의 예를 따라 해보자.

```
>>> a[0]
1
>>> a[-1]
['a', 'b', 'c']
>>> a[3]
['a', 'b', 'c']
```

예상한 대로 `a[-1]`은 마지막 요소값인 `['a', 'b', 'c']`를 나타낸다. `a[3]`은 리스트 `a`의 네 번째 요소를 나타내기 때문에 마지막 요소를 나타내는 `a[-1]`과 동일한 결과값을 보여 준다.

그렇다면 여기서 리스트 `a`에 포함된 `['a', 'b', 'c']`라는 리스트에서 `'a'`라는 값을 인덱싱을 이용해 꼬집어낼 수 있는 방법은 없을까? 다음의 예를 보자.

```
>>> a[-1][0]
'a'
```

위와 같이 하면 `'a'`를 꼬집어낼 수 있다. `a[-1]`이 `['a', 'b', 'c']` 리스트라는 것은 이미 말했다. 바로 이 리스트에서 첫 번째 요소를 불러오기 위해 `[0]`을 붙여준 것이다.

아래의 예도 마찬가지 경우이므로 어렵지 않게 이해가 될 것이다.

```
>>> a[-1][1]
'b'
>>> a[-1][2]
'c'
```

[삼중 리스트에서 인덱싱하기]

조금 복잡하지만 다음의 예를 따라 해보자.

```
>>> a = [1, 2, ['a', 'b', ['Life', 'is']]]
```

리스트 `a`안에 `['a', 'b', ['Life', 'is']]`라는 리스트가 포함되어 있고, 그 리스트 안에 다시 `['Life', 'is']`라는 리스트가 포함되어 있다. 삼중 구조의 리스트이다.

이 경우 `'Life'`라는 문자열만 꼬집어내려면 다음과 같이 해야 한다.

```
>>> a[2][2][0]
'Life'
```

위의 예는 리스트 a의 세 번째 요소인 리스트 ['a', 'b', ['Life', 'is']]에서 세 번째 요소인 리스트 ['Life', 'is']의 첫 번째 요소를 나타낸다.

이렇듯 리스트를 삼중으로 중첩해서 쓰면 혼란스럽기 때문에 자주 사용하지는 않지만 알아두는 것이 좋다.

리스트의 슬라이싱

문자열과 마찬가지로 리스트에서도 슬라이싱 기법을 적용할 수 있다. 슬라이싱은 “나눈다”는 뜻이라고 했다.

자, 그럼 리스트의 슬라이싱에 대해서 살펴보자.

```
>>> a = [1, 2, 3, 4, 5]
>>> a[0:2]
[1, 2]
```

앞의 예를 문자열에서 슬라이싱했던 것과 비교해 보자.

```
>>> a = "12345"
>>> a[0:2]
'12'
```

2가지가 완전히 동일하게 사용됨을 눈치챘을 것이다. 문자열에서 했던 것과 사용법이 완전히 동일하다.

몇 가지 예를 더 들어 보도록 하자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> a = [1, 2, 3, 4, 5]
>>> b = a[:2]
>>> c = a[2:]
>>> b
[1, 2]
>>> c
[3, 4, 5]
```

b라는 변수는 리스트 a의 첫번째 요소부터 두 번째 요소인 a[1]까지 나타내는 리스트이다. 물론 a[2] 값인 3은 포함되지 않는다. c라는 변수는 리스트 a의 세 번째 요소부터 끝까지 나타내는 리스트이다.

[중첩된 리스트에서 슬라이싱하기]

리스트가 포함된 중첩 리스트 역시 슬라이싱 방법은 똑같이 적용된다.

```
>>> a = [1, 2, 3, ['a', 'b', 'c'], 4, 5]
>>> a[2:5]
[3, ['a', 'b', 'c'], 4]
>>> a[3][:2]
['a', 'b']
```

위의 예에서 a[3]은 ['a', 'b', 'c']를 나타낸다. 따라서 a[3][:2]는 ['a', 'b', 'c']의 첫 번째 요소부터 세 번째 요소 직전까지의 값, 즉 ['a', 'b']를 나타내는 리스트가 된다.

리스트 연산자

리스트 역시 + 기호를 이용해서 더할 수 있고, * 기호를 이용해서 반복할 수 있다. 문자열과 마찬가지로 리스트에서도 되는지 직접 확인해 보자.

1) 리스트 더하기(+)

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
```

리스트 사이에서 + 기호는 2개의 리스트를 합치는 기능을 한다. 문자열에서 "abc" + "def" = "abcdef"가 되는 것과 같은 이치이다.

2) 리스트 반복하기(*)

```
>>> a = [1, 2, 3]
>>> a * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

위에서 볼 수 있듯이 [1, 2, 3]이라는 리스트가 세 번 반복되어 새로운 리스트를 만들어낸다. 문자열에서 "abc" * 3 = "abcabcabc" 가 되는 것과 같은 이치이다.

[초보자가 범하기 쉬운 리스트 연산 오류]

다음과 같은 소스 코드를 입력했을 때 결과값은 어떻게 나올까?

```
>>> a = [1, 2, 3]
>>> a[2] + "hi"
```

a[2]의 값인 3과 문자열 hi가 더해져서 3hi가 출력될 것이라고 생각할 수 있다. 하지만 다음의 결과를 보자. 형 오류(TypeError)가 발생했다. 오류의 원인은 무엇일까?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

a[2]에 저장된 값은 3이라는 정수인데 "hi"는 문자열이다. 정수와 문자열은 당연히 서로 더할 수 없기 때문에 형 오류가 발생한 것이다.

만약 숫자와 문자열을 더해서 '3hi'처럼 만들고 싶다면 숫자 3을 문자 '3'으로 바꾸어 주어야 한다. 다음과 같이 할 수 있다.

```
>>> str(a[2]) + "hi"
```

str()은 정수나 실수를 문자열의 형태로 바꾸어 주는 파이썬의 내장 함수이다.

리스트의 수정, 변경과 삭제

다음의 예들은 서로 연관되어 있으므로 따로따로 실행하지 말고 차례대로 진행해야 한다.

1. 리스트에서 하나의 값 수정하기

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

a[2]의 요소값 3이 4로 바뀌었다.

2. 리스트에서 연속된 범위의 값 수정하기

```
>>> a[1:2]
[2]
>>> a[1:2] = ['a', 'b', 'c']
>>> a
[1, 'a', 'b', 'c', 4]
```

a[1:2]는 a[1]부터 a[2]까지를 말하는데 a[2]는 포함하지 않는다고 했으므로 a = [1, 2, 4]에서 a[1]의 값인 2만을 의미한다. 즉, a[1:2]를 ['a', 'b', 'c']로 바꾸었으므로 a 리스트에서 2라는 값 대신에 ['a', 'b', 'c']라는 값이 대입되었다.

[리스트 수정할 때 주의할 점]

2번 예제에서 리스트를 a[1:2] = ['a', 'b', 'c']로 수정하는 것과 a[1] = ['a', 'b', 'c']로 수정하는 것은 전혀 다른 결과값을 갖게 되므로 주의해야 한다. a[1] = ['a', 'b', 'c']는 리스트 a의 두 번째 요소를 ['a', 'b', 'c']로 바꾼다는 말이고 a[1:2]는 a[1]에서 a[2] 사이의 리스트를 ['a', 'b', 'c']로 바꾼다는 말이다. 따라서 a[1] = ['a', 'b', 'c']로 수정하게 되면 위와는 달리 리스트 a가 [1, ['a', 'b', 'c'], 4]라는 값으로 변하게 된다.

```
>>> a[1] = ['a', 'b', 'c']
>>> a
[1, ['a', 'b', 'c'], 4]
```

3. [] 사용해 리스트 요소 삭제하기

```
>>> a[1:3] = []
>>> a
[1, 'c', 4]
```

2번까지 진행한 리스트 a의 값은 [1, 'a', 'b', 'c', 4]였다. 여기서 a[1:3]은 a의 인덱스 1부터 3까지 ($1 \leq a < 3$), 즉 a[1], a[2]를 의미하므로 a[1:3]은 ['a', 'b']이다. 그런데 위의 예에서 볼 수 있듯이 a[1:3]을 []으로 바꿔 주었기 때문에 a에서 ['a', 'b']가 삭제된 [1, 'c', 4]가 된다.

4. del 함수 사용해 리스트 요소 삭제하기

```
>>> a
[1, 'c', 4]
>>> del a[1]
>>> a
[1, 4]
```

del a[x]는 x번째 요소값을 삭제한다. del a[x:y]는 x번째부터 y번째 요소 사이의 값을 삭제한다. 여기서는 a 리스트에서 a[1]을 삭제하는 방법을 보여 준다. del 함수는 파이썬이 자체적으로 가지고 있는 삭제 함수이며 다음과 같이 사용한다.

del 객체

※ 객체란 파이썬에서 사용되는 모든 자료형을 말한다.

리스트 관련 함수들

문자열과 마찬가지로 리스트 변수명 뒤에 '.'를 붙여서 여러 가지 리스트 관련 함수들을 이용할 수 있다. 유용하게 사용되는 리스트 관련 함수 몇 가지에 대해서만 알아보기로 하자.

리스트에 요소 추가(append)

append를 사전에서 검색해 보면 “덧붙이다, 첨부하다”라는 뜻이 있다. 이 뜻을 안다면 아래의 예가 금방 이해가 될 것이다. append(x)는 리스트의 맨 마지막에 x를 추가시키는 함수이다.

```
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
```

리스트 안에는 어떤 자료형도 추가할 수 있다.

아래의 예는 리스트에 다시 리스트를 추가한 결과이다.

```
>>> a.append([5,6])
>>> a
[1, 2, 3, 4, [5, 6]]
```

리스트 정렬(sort)

sort 함수는 리스트의 요소를 순서대로 정렬해 준다.

```
>>> a = [1, 4, 3, 2]
>>> a.sort()
>>> a
[1, 2, 3, 4]
```

문자 역시 알파벳 순서로 정렬할 수 있다.

```
>>> a = ['a', 'c', 'b']
>>> a.sort()
>>> a
['a', 'b', 'c']
```

리스트 뒤집기(reverse)

reverse 함수는 리스트를 역순으로 뒤집어 준다. 이때 리스트 요소들을 순서대로 정렬한 다음 다시 역순으로 정렬하는 것이 아니라 그저 현재의 리스트를 그대로 거꾸로 뒤집을 뿐이다.

```
>>> a = ['a', 'c', 'b']
>>> a.reverse()
>>> a
['b', 'c', 'a']
```

위치 반환(index)

`index(x)` 함수는 리스트에 `x`라는 값이 있으면 `x`의 위치값을 리턴한다.

```
>>> a = [1,2,3]
>>> a.index(3)
2
>>> a.index(1)
0
```

위의 예에서 리스트 `a`에 있는 3이라는 숫자의 위치는 `a[2]`이므로 2를 리턴하고, 1이라는 숫자의 위치는 `a[0]`이므로 0을 리턴한다.

아래의 예에서 0이라는 값은 `a` 리스트에 존재하지 않기 때문에 값 오류(ValueError)가 발생한다.

```
>>> a.index(0)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 0 is not in list
```

리스트에 요소 삽입(insert)

`insert(a, b)`는 리스트의 `a`번째 위치에 `b`를 삽입하는 함수이다. 파이썬에서는 숫자를 0부터 센다는 것을 반드시 기억하자.

```
>>> a = [1, 2, 3]
>>> a.insert(0, 4)
>>> a
[4, 1, 2, 3]
```

위의 예는 0번째 자리, 즉 첫 번째 요소($a[0]$) 위치에 4라는 값을 삽입하라는 뜻이다.

```
>>> a.insert(3, 5)
>>> a
[4, 1, 2, 5, 3]
```

위의 예는 리스트 a 의 $a[3]$, 즉 네 번째 요소 위치에 5라는 값을 삽입하라는 뜻이다.

리스트 요소 제거(remove)

$\text{remove}(x)$ 는 리스트에서 첫 번째로 나오는 x 를 삭제하는 함수이다.

```
>>> a = [1, 2, 3, 1, 2, 3]
>>> a.remove(3)
>>> a
[1, 2, 1, 2, 3]
```

a 가 3이라는 값을 2개 가지고 있을 경우 첫 번째 3만 제거되는 것을 알 수 있다.

```
>>> a.remove(3)
>>> a
[1, 2, 1, 2]
```

$\text{remove}(3)$ 을 한 번 더 실행하면 다시 3이 삭제된다.

리스트 요소 끄집어내기(pop)

$\text{pop}()$ 은 리스트의 맨 마지막 요소를 뒤로 주고 그 요소는 삭제하는 함수이다.

```
>>> a = [1,2,3]
>>> a.pop()
3
>>> a
[1, 2]
```

a 리스트 [1,2,3]에서 3을 끄집어내고 최종적으로 [1, 2]만 남는 것을 볼 수 있다.

`pop(x)`는 리스트의 x번째 요소를 돌려 주고 그 요소는 삭제한다.

```
>>> a = [1,2,3]
>>> a.pop(1)
2
>>> a
[1, 3]
```

`a.pop(1)`은 `a[1]`의 값을 끄집어낸다. 다시 a를 출력해 보면 끄집어낸 값이 삭제된 것을 확인할 수 있다.

리스트에 포함된 요소 x의 개수 세기(count)

`count(x)`는 리스트 내에 x가 몇 개 있는지 조사하여 그 개수를 돌려주는 함수이다.

```
>>> a = [1,2,3,1]
>>> a.count(1)
2
```

1이라는 값이 리스트 a에 2개 들어 있으므로 2를 돌려준다.

리스트 확장(extend)

`extend(x)`에서 x에는 리스트만 올 수 있으며 원래의 a 리스트에 x 리스트를 더하게 된다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> a = [1,2,3]
>>> a.extend([4,5])
>>> a
[1, 2, 3, 4, 5]
>>> b = [6, 7]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6, 7]
```

a.extend([4,5])는 a += [4,5]와 동일하다.

** 리스트의 요소를 제거하는 3가지 방법 **

지금까지 알아본 리스트의 요소를 제거하는 방법은 총 3가지이다.

1. 리스트의 remove 함수 이용하기
2. 리스트의 pop 함수 이용하기
3. del을 이용하기

이 3가지 방법은 결과적으로 리스트의 요소가 삭제된다는 점에서는 동일하지만 삭제에 사용되는 입력값에는 큰 차이점이 있다. 예제로 알아보자.

```
>>> a = [1, 2, 3, 'a', 'b', 'c']
>>> a.remove('a')
>>> a
[1, 2, 3, 'b', 'c']
```

a.remove('a')는 a 리스트의 'a'라는 요소값을 삭제한다. 즉, a.remove(x)의 x에는 인덱스가 아닌 요소값 만을 사용할 수 있다. 참고로 다음과 같이 리스트의 요소를 인덱스로 삭제하려고 하면 다음과 같은 오류가 발생한다.

```
>>> a.remove(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

pop은 remove와 반대로 요소값으로 삭제할 수는 없고 인덱스로만 삭제가 가능하다. 즉, a.pop(x)의 x에는 a리스트의 인덱스만 가능하다. 다음의 예제로 확인해 보자.

```
>>> a = [1, 2, 3, 'a', 'b', 'c']
>>> a.pop(4)
'b'
>>> a
[1, 2, 3, 'a', 'c']
```

a.pop(4)는 a리스트의 5번째 요소를 삭제한다. 단, pop함수는 삭제된 요소를 리턴받는 특징이 있다.

만약 pop 함수 사용시 인덱스가 아닌 요소값을 이용한다면 다음과 같은 오류를 만나게 될 것이다.

```
>>> a.pop('b')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted as an integer
```

del은 리스트의 요소를 삭제하는 함수이므로 당연히 인덱스만 가능하다.

```
>>> a = [1, 2, 3, 'a', 'b', 'c']
>>> del a[4]
>>> a
[1, 2, 3, 'a', 'c']
```

del a[4] 는 a리스트의 5번째 요소를 삭제한다는 것을 알 수 있다.

즉, 결론은 다음과 같다.

1. a.remove(x) - x는 a리스트의 요소값
2. a.pop(x) - x는 a리스트의 인덱스
3. del a[x] - x는 a리스트의 인덱스

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-3>)

[문제1] 리스트 인덱싱

다음과 같은 리스트 a가 있다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> a = ['Life', 'is', 'too', 'short', 'you', 'need', 'python']
```

a 리스트를 이용하여 다음과 같은 문자열을 출력하시오.

```
you too
```

[문제2] 리스트 조인

['Life', 'is', 'too', 'short']라는 리스트를 Life is too short라는 문자열로 만들어 출력해 보자.

[문제3] 리스트의 갯수

다음과 같은 리스트 a가 있다.

```
>>> a = [1, 2, 3]
```

이 리스트의 갯수(사이즈)를 구하시오.

(힌트. 객체의 길이를 리턴하는 파이썬 내장함수 len을 이용해 보자.)

[문제4] 리스트의 append와 extend

다음과 같은 리스트 a가 있다.

```
>>> a = [1, 2, 3]
```

리스트 a에 [4, 5]를 append 했을 때와 extend했을 때의 차이점은 무엇인가?

[문제5] 리스트 정렬

[1, 3, 5, 4, 2]라는 리스트를 [5, 4, 3, 2, 1]로 만들어보자. (힌트. 리스트의 내장함수인 sort와 reverse를 활용해 보자.)

[문제6] 리스트 삭제

[1, 2, 3, 4, 5]라는 리스트를 [1, 3, 5]로 만들어 보자.

02-4 튜플 자료형

튜플은 어떻게 만들까?

튜플(tuple)은 몇 가지 점을 제외하곤 리스트와 거의 비슷하며 리스트와 다른 점은 다음과 같다.

- 리스트는 [과]으로 둘러싸지만 튜플은 (과)으로 둘러싼다.
- 리스트는 그 값의 생성, 삭제, 수정이 가능하지만 튜플은 그 값을 바꿀 수 없다.

튜플의 모습은 다음과 같다.

```
>>> t1 = ()
>>> t2 = (1,)
>>> t3 = (1, 2, 3)
>>> t4 = 1, 2, 3
>>> t5 = ('a', 'b', ('ab', 'cd'))
```

리스트와 모습은 거의 비슷하지만 튜플에서는 리스트와 다른 2가지 차이점을 찾아볼 수 있다. t2 = (1,)처럼 단지 1개의 요소만을 가질 때는 요소 뒤에 콤마(,)를 반드시 붙여야 한다는 것과 t4 = 1, 2, 3처럼 괄호()를 생략해도 무방하다는 점이다.

얼핏 보면 튜플과 리스트는 비슷한 역할을 하지만 프로그래밍을 할 때 튜플과 리스트는 구분해서 사용하는 것이 유리하다. 튜플과 리스트의 가장 큰 차이는 값을 변화시킬 수 있는가 없는가이다. 즉, 리스트의 항목값은 변화가 가능하고 튜플의 항목값은 변화가 불가능하다. 따라서 프로그램이 실행되는 동안 그 값이 항상 변하지 않기를 바란다거나 값이 바뀔까 걱정하고 싶지 않다면 주저하지 말고 튜플을 사용해야 한다. 이와는 반대로 수시로 그 값을 변화시켜야 할 경우라면 리스트를 사용해야 한다. 실제 프로그램에서는 값이 변경되는 형태의 변수가 훨씬 많기 때문에 평균적으로 튜플보다는 리스트를 더 많이 사용하게 된다.

튜플의 요소값을 지우거나 변경하려고 하면 어떻게 될까?

앞서 설명했듯이 튜플의 요소값은 한 번 정하면 지우거나 변경할 수 없다. 다음에 소개하는 2개의 예를 살펴보면 무슨 말인지 알 수 있을 것이다.

1. 튜플 요소값 삭제 시 오류

```
>>> t1 = (1, 2, 'a', 'b')
>>> del t1[0]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

튜플의 요소를 리스트처럼 `del` 함수로 지우려고 한 예이다. 튜플은 요소를 지우는 행위가 지원되지 않는다는 메시지를 확인할 수 있다.

2. 튜플 요소값 변경 시 오류

```
>>> t1 = (1, 2, 'a', 'b')
>>> t1[0] = 'c'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

튜플의 요소값을 변경하려고 해도 마찬가지로 오류가 발생하는 것을 확인할 수 있다.

튜플의 인덱싱과 슬라이싱, 더하기(+)와 곱하기(*)

튜플은 값을 변화시킬 수 없다는 점만 제외하면 리스트와 완전히 동일하므로 간단하게만 살펴보겠다. 다음의 예제는 서로 연관되어 있으므로 차례대로 수행해 보기 바란다.

1. 인덱싱하기

```
>>> t1 = (1, 2, 'a', 'b')
>>> t1[0]
1
>>> t1[3]
'b'
```

문자열, 리스트와 마찬가지로 `t1[0]`, `t1[3]`처럼 인덱싱이 가능하다.

2. 슬라이싱하기

```
>>> t1 = (1, 2, 'a', 'b')
>>> t1[1:]
(2, 'a', 'b')
```

t1[1]부터 튜플의 마지막 요소까지 슬라이싱하는 예이다.

3. 튜플 더하기

```
>>> t2 = (3, 4)
>>> t1 + t2
(1, 2, 'a', 'b', 3, 4)
```

튜플을 더하는 방법을 보여주는 예이다.

4. 튜플 곱하기

```
>>> t2 * 3
(3, 4, 3, 4, 3, 4)
```

튜플의 곱하기(반복) 예를 보여 준다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-4>)

[문제1] 튜플 작성

숫자 3만을 요소값으로 가지는 튜플을 작성하라.

[문제2] 튜플 변경

다음은 튜플 (1, 2, 3)을 (1, 4, 3)과 같이 변경하려고 시도했을 경우이다. 오류의 원인에 대해서 설명하시오.

```
>>> a = (1, 2, 3)
>>> a[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

[문제3] 튜플 추가

(1,2,3)이라는 튜플에 4라는 값을 추가하여 (1,2,3,4)처럼 만들어 출력해 보자.

02-5 딕셔너리 자료형

딕셔너리란?

사람은 누구든지 “이름” = “홍길동”, “생일” = “몇 월 며칠” 등으로 구분할 수 있다. 파이썬은 영리하게도 이러한 대응 관계를 나타낼 수 있는 자료형을 가지고 있다. 요즘 사용하는 대부분의 언어들도 이러한 대응 관계를 나타내는 자료형을 갖고 있는데, 이를 연관 배열(Associative array) 또는 해시(Hash)라고 한다.

파이썬에서는 이러한 자료형을 딕셔너리(Dictionary)라고 하는데, 단어 그대로 해석하면 사전이라는 뜻이다. 즉, people이라는 단어에 “사람”, baseball이라는 단어에 “야구”라는 뜻이 부합되듯이 딕셔너리는 Key와 Value라는 것을 한 쌍으로 갖는 자료형이다. 예컨대 Key가 “baseball”이라면 Value는 “야구”가 될 것이다.

딕셔너리는 리스트나 튜플처럼 순차적으로(sequential) 해당 요소값을 구하지 않고 Key를 통해 Value를 얻는다. 이것이 바로 딕셔너리의 가장 큰 특징이다. baseball이라는 단어의 뜻을 찾기 위해 사전의 내용을 순차적으로 모두 검색하는 것이 아니라 baseball이라는 단어가 있는 곳만 펼쳐 보는 것이다.

딕셔너리는 어떻게 만들까?

다음은 기본적인 딕셔너리의 모습이다.

```
{Key1:Value1, Key2:Value2, Key3:Value3 ...}
```

Key와 Value의 쌍 여러 개가 {과 }로 둘러싸여 있다. 각각의 요소는 Key : Value 형태로 이루어져 있고 쉼표(,)로 구분되어 있다.

※ Key에는 변하지 않는 값을 사용하고, Value에는 변하는 값과 변하지 않는 값 모두 사용할 수 있다.

다음의 딕셔너리 예를 살펴보자.

```
>>> dic = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
```

위에서 Key는 각각 ‘name’, ‘phone’, ‘birth’이고, 각각의 Key에 해당하는 Value는 ‘pey’, ‘0119993323’, ‘1118’이 된다.

딕셔너리 dic의 정보

key	value
name	pey
phone	01199993323
birth	1118

다음의 예는 Key로 정수값 1, Value로 'hi'라는 문자열을 사용한 예이다.

```
>>> a = {1: 'hi'}
```

또한 다음의 예처럼 Value에 리스트도 넣을 수 있다.

```
>>> a = { 'a': [1,2,3]}
```

딕셔너리 쌍 추가, 삭제하기

딕셔너리 쌍을 추가하는 방법과 삭제하는 방법을 살펴보자. 1번은 딕셔너리에 쌍을 추가하는 예이다.

다음의 예를 함께 따라 해보자.

1. 딕셔너리 쌍 추가하기

```
>>> a = {1: 'a'}
>>> a[2] = 'b'
>>> a
{1: 'a', 2: 'b'}
```

{1: 'a'}라는 딕셔너리에 a[2] = 'b'와 같이 입력하면 딕셔너리 a에 Key와 Value가 각각 2와 'b'인 2 : 'b'라는 딕셔너리 쌍이 추가된다.

```
>>> a['name'] = 'pey'
>>> a
{1: 'a', 2: 'b', 'name': 'pey'}
```

딕셔너리 a에 'name': 'pey'라는 쌍이 추가되었다.

```
>>> a[3] = [1,2,3]
>>> a
{1: 'a', 2: 'b', 'name': 'pey', 3: [1, 2, 3]}
```

Key는 3, Value는 [1, 2, 3]을 가지는 한 쌍이 또 추가되었다.

2. 딕셔너리 요소 삭제하기

```
>>> del a[1]
>>> a
{2: 'b', 'name': 'pey', 3: [1, 2, 3]}
```

위의 예제는 딕셔너리 요소를 지우는 방법을 보여준다. del 함수를 사용해서 del a[key]처럼 입력하면 지정한 key에 해당하는 {key : value} 쌍이 삭제된다.

딕셔너리를 사용하는 방법

“딕셔너리는 주로 어떤 것을 표현하는 데 사용할까?”라는 의문이 들 것이다. 예를 들어 4명의 사람이 있다고 가정하고, 각자의 특기를 표현할 수 있는 좋은 방법에 대해서 생각해 보자. 리스트나 문자열로는 표현하기가 상당히 까다로울 것이다. 하지만 파이썬의 딕셔너리를 사용한다면 이 상황을 표현하기가 정말 쉽다.

다음의 예를 보자.

```
{"김연아": "피겨스케이팅", "박찬호": "야구", "박지성": "축구", "귀도": "파이썬"}
```

사람 이름과 특기를 한 쌍으로 하는 딕셔너리이다. 정말 간편하지 않은가? 지금껏 우리는 딕셔너리를 만드는 방법에 대해서만 살펴보았는데 딕셔너리를 제대로 활용하기 위해서는 알아야 할 것들이 있다. 이제부터 하나씩 알아보도록 하자.

딕셔너리에서 Key 사용해 Value 얻기

다음의 예를 살펴보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> grade = {'pey': 10, 'julliet': 99}
>>> grade['pey']
10
>>> grade['julliet']
99
```

리스트나 튜플, 문자열은 요소값을 얻어내고자 할 때 인덱싱이나 슬라이싱 기법 중 하나를 이용했다. 하지만 딕셔너리는 단 한 가지 방법뿐이다. 바로 Key를 사용해서 Value를 얻어내는 방법이다. 위의 예에서 'pey'라는 Key의 Value를 얻기 위해 grade['pey']를 사용한 것처럼 어떤 Key의 Value를 얻기 위해서는 “딕셔너리 변수[Key]”를 사용한다.

몇 가지 예를 더 보자.

```
>>> a = {1:'a', 2:'b'}
>>> a[1]
'a'
>>> a[2]
'b'
```

먼저 a라는 변수에 {1:'a', 2:'b'}라는 딕셔너리를 대입하였다. 위의 예에서 볼 수 있듯이 a[1]은 'a'라는 값을 돌려준다. 여기서 a[1]이 의미하는 것은 리스트나 튜플의 a[1]과는 전혀 다르다. 딕셔너리 변수에서 [] 안의 숫자 1은 두 번째 요소를 뜻하는 것이 아니라 Key에 해당하는 1을 나타낸다. 앞에서도 말했듯이 딕셔너리는 리스트나 튜플에 있는 인덱싱 방법을 적용할수 없다. 따라서 a[1]은 딕셔너리 {1:'a', 2:'b'}에서 Key가 1인 것의 Value인 'a'를 돌려주게 된다. a[2] 역시 마찬가지이다.

이번에는 a라는 변수에 앞의 예에서 사용했던 딕셔너리의 Key와 Value를 뒤집어 놓은 딕셔너리를 대입해 보자.

```
>>> a = {'a':1, 'b':2}
>>> a['a']
1
>>> a['b']
2
```

역시 a['a'], a['b']처럼 Key를 사용해서 Value를 얻을 수 있다. 정리하면, 딕셔너리 a는 a[Key]로 입력해서 Key에 해당하는 Value를 얻는다.

다음 예는 이전에 한 번 언급했던 딕셔너리인데 Key를 사용해서 Value를 얻는 방법을 잘 보여준다.

```
>>> dic = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
>>> dic['name']
'pey'
>>> dic['phone']
'0119993323'
>>> dic['birth']
'1118'
```

딕셔너리 만들 때 주의할 사항

먼저 딕셔너리에서 Key는 고유한 값이므로 중복되는 Key 값을 설정해 놓으면 하나를 제외한 나머지 것들이 모두 무시된다는 점을 주의해야 한다. 다음 예에서 볼 수 있듯이 동일한 Key가 2개 존재할 경우 1:'a'라는 쌍이 무시된다.

```
>>> a = {1: 'a', 1: 'b'}
>>> a
{1: 'b'}
```

이렇게 Key가 중복되었을 때 1개를 제외한 나머지 Key:Value 값이 모두 무시되는 이유는 Key를 통해서 Value를 얻는 딕셔너리의 특징에서 비롯된다. 즉, 동일한 Key가 존재하면 어떤 Key에 해당하는 Value를 불러야 할지 알 수 없기 때문이다.

또 한 가지 주의해야 할 사항은 Key에 리스트는 쓸 수 없다는 것이다. 하지만 튜플은 Key로 쓸 수 있다. 딕셔너리의 Key로 쓸 수 있느냐 없느냐는 Key가 변하는 값인지 변하지 않는 값인지에 달려 있다. 리스트는 그 값이 변할 수 있기 때문에 Key로 쓸 수 없는 것이다. 아래 예처럼 리스트를 Key로 설정하면 리스트를 키 값으로 사용할 수 없다는 형 오류(TypeError)가 발생한다.

```
>>> a = {[1,2] : 'hi'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

따라서 딕셔너리의 Key 값으로 딕셔너리를 사용할 수 없음은 당연한 얘기가 될 것이다. 단, Value에는 변하는 값이든 변하지 않는 값이든 상관없이 아무 값이나 넣을 수 있다.

딕셔너리 관련 함수들

딕셔너리를 자유자재로 사용하기 위해 딕셔너리가 자체적으로 가지고 있는 관련 함수들을 사용해 보도록 하자.

Key 리스트 만들기(keys)

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
>>> a.keys()
dict_keys(['name', 'phone', 'birth'])
```

a.keys()는 딕셔너리 a의 Key만을 모아서 dict_keys라는 객체를 리턴한다.

[파이썬 3.0 이후 버전의 keys 함수, 어떻게 달라졌나?]

파이썬 2.7 버전까지는 a.keys() 호출 시 리턴값으로 dict_keys가 아닌 리스트를 리턴한다. 리스트를 리턴하기 위해서는 메모리의 낭비가 발생하는데 파이썬 3.0 이후 버전에서는 이러한 메모리 낭비를 줄이기 위해 dict_keys라는 객체를 리턴해 준다. 다음에 소개할 dict_values, dict_items 역시 파이썬 3.0 이후 버전에서 추가된 것들이다. 만약 3.0 이후 버전에서 리턴값으로 리스트가 필요한 경우에는 “list(a.keys())”를 사용하면 된다. dict_keys, dict_values, dict_items 등은 리스트로 변환하지 않더라도 기본적인 반복성(iterate) 구문(예: for문)들을 실행할 수 있다.

dict_keys 객체는 다음과 같이 사용할 수 있다. 리스트를 사용하는 것과 차이가 없지만, 리스트 고유의 함수인 append, insert, pop, remove, sort 등의 함수를 수행할 수는 없다.

```
>>> for k in a.keys():
...     print(k)
...
name
phone
birth
```

dict_keys 객체를 리스트로 변환하려면 다음과 같이 하면 된다.

```
>>> list(a.keys())
['name', 'phone', 'birth']
```

Value 리스트 만들기(values)

```
>>> a.values()
dict_values(['pey', '0119993323', '1118'])
```

Key를 얻는 것과 마찬가지 방법으로 Value만 얻고 싶다면 a.values()처럼 values 함수를 사용하면 된다. values 함수를 호출하면 dict_values 객체가 리턴되는데, dict_values 객체 역시 dict_keys 객체와 마찬가지로 리스트를 사용하는 것과 동일하게 사용하면 된다.

Key, Value 쌍 얻기(items)

```
>>> a.items()
dict_items([('name', 'pey'), ('phone', '0119993323'), ('birth', '1118')])
```

items 함수는 key와 value의 쌍을 튜플로 묶은 값을 dict_items 객체로 돌려준다.

Key: Value 쌍 모두 지우기(clear)

```
>>> a.clear()
>>> a
{}
```

clear() 함수는 덕셔너리 안의 모든 요소를 삭제한다. 빈 리스트를 [], 빈 튜플을 ()로 표현하는 것과 마찬가지로 빈 덕셔너리도 {}로 표현한다.

Key로 Value얻기(get)

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
>>> a.get('name')
'pey'
>>> a.get('phone')
'0119993323'
```

`get(x)` 함수는 `x`라는 key에 대응되는 value를 돌려준다. 앞서 살펴보았듯이 `a.get('name')`은 `a['name']`을 사용했을 때와 동일한 결과값을 돌려받는다.

다만, 다음 예제에서 볼 수 있듯이 `a['nokey']`처럼 `a` 딕셔너리에 없는 키로 값을 가져오려고 할 경우 `a['nokey']`는 Key 오류를 발생시키고 `a.get('nokey')`는 `None`을 리턴한다는 차이가 있다. 어떤 것을 사용할지는 여러분의 선택이다.

※ 여기서 `None`은 “거짓”이라는 뜻이라고만 알아두자.

```
>>> a.get('nokey')
>>> a['nokey']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'nokey'
```

딕셔너리 안에 찾으려는 key 값이 없을 경우 미리 정해 둔 디폴트 값을 대신 가져오게 하고 싶을 때에는 `get(x, '디폴트 값')`을 사용하면 편리하다.

```
>>> a.get('foo', 'bar')
'bar'
```

`a` 딕셔너리에는 'foo'에 해당하는 값이 없다. 따라서 디폴트 값인 'bar'를 리턴한다.

해당 Key가 딕셔너리 안에 있는지 조사하기(in)

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
>>> 'name' in a
True
>>> 'email' in a
False
```

'name'이라는 문자열은 `a` 딕셔너리의 key 중 하나이다. 따라서 '`name`' in `a`를 호출하면 참(True)을 리턴한다. 반대로 'email'은 `a` 딕셔너리 안에 존재하지 않는 key이므로 거짓(False)을 리턴하게 된다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-5>)

[문제1] 딕셔너리 만들기

다음 표를 딕셔너리로 만드시오.

항목	값
name	홍길동
birth	1128
age	30

[문제2] 딕셔너리 오류

다음과 같은 딕셔너리 a가 있다.

```
>>> a = dict()
>>> a
{}
```

다음 중 오류가 발생하는 경우는 어떤 경우인가? 그리고 그 이유를 설명하시오.

1. a['name'] = 'python'
2. a[('a',)] = 'python'
3. a[[1]] = 'python'
4. a[250] = 'python'

[문제3] 딕셔너리 값 추출1

딕셔너리 a에서 'B'에 해당되는 값을 추출하고 삭제해 보자.

```
>>> a = {'A':90, 'B':80, 'C':70}
```

[문제4] 딕셔너리 값 추출2

다음은 딕셔너리의 a에서 'C'라는 key에 해당되는 value를 출력하는 프로그램이다.

```
>>> a = {'A':90, 'B':80}
>>> a['C']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'C'
```

a 딕셔너리에는 'C'라는 key가 없으므로 위와 같은 오류가 발생하게 된다. 'C'에 해당되는 키값이 없을 경우 오류 대신 70을 얻을수 있도록 수정해 보자.

[문제5] 딕셔너리 최소값

다음과 같은 딕셔너리 a가 있다.

```
>>> a = {'A':90, 'B':80, 'C':70}
```

딕셔너리 a의 value중에서 최소 값을 출력하시오.

(힌트. 여러개의 요소값중 최소값을 얻을수 있는 min함수를 이용해 보자.)

[문제6] 딕셔너리 리스트 변환

다음과 같은 딕셔너리 a가 있다.

```
>>> a = {'A':90, 'B':80, 'C':70}
```

위 딕셔너리 a를 다음과 같은 리스트로 만들어보자.

```
[(‘A’, 90), (‘B’, 80), (‘C’, 70)]
```

(힌트. 딕셔너리의 items 내장함수를 이용해 보자.)

02-6 집합 자료형

집합 자료형은 어떻게 만들까?

집합(set)은 파이썬 2.3부터 지원되기 시작한 자료형으로, 집합에 관련된 것들을 쉽게 처리하기 위해 만들어진 자료형이다.

집합 자료형은 다음과 같이 set 키워드를 이용해 만들 수 있다.

```
>>> s1 = set([1,2,3])
>>> s1
{1, 2, 3}
```

위와 같이 set()의 괄호 안에 리스트를 입력하여 만들거나 아래와 같이 문자열을 입력하여 만들 수도 있다.

```
>>> s2 = set("Hello")
>>> s2
{'e', 'H', 'l', 'o'}
```

집합 자료형의 특징

자, 그런데 위에서 살펴본 set("Hello")의 결과가 좀 이상하지 않은가? 분명 "Hello"라는 문자열로 set 자료형을 만들었는데 생성된 자료형에는 1 문자가 하나 빠져 있고 순서도 뒤죽박죽이다. 그 이유는 set에는 다음과 같은 2가지 큰 특징이 있기 때문이다.

- 중복을 허용하지 않는다.
- 순서가 없다(Unordered).

리스트나 튜플은 순서가 있기(ordered) 때문에 인덱싱을 통해 자료형의 값을 얻을 수 있지만 set 자료형은 순서가 없기(unordered) 때문에 인덱싱으로 값을 얻을 수 없다. 이는 마치 02-5절에서 살펴본 딕셔너리와 비슷하다. 딕셔너리 역시 순서가 없는 자료형이라 인덱싱을 지원하지 않는다. 만약 set 자료형에 저장된 값을 인덱싱으로 접근하려면 다음과 같이 리스트나 튜플로 변환한 후 해야 한다.

※ 중복을 허용하지 않는 set의 특징은 자료형의 중복을 제거하기 위한 필터 역할로 종종 사용되기도 한다.

```
>>> s1 = set([1,2,3])
>>> l1 = list(s1)
>>> l1
[1, 2, 3]
>>> l1[0]
1
>>> t1 = tuple(s1)
>>> t1
(1, 2, 3)
>>> t1[0]
1
```

집합 자료형 활용하는 방법

교집합, 합집합, 차집합 구하기

set 자료형이 정말 유용하게 사용되는 경우는 다음과 같이 교집합, 합집합, 차집합을 구할 때이다.

우선 다음과 같이 2개의 set 자료형을 만든 후 따라 해보자. s1은 1부터 6까지의 값을 가지게 되었고, s2는 4부터 9까지의 값을 가지게 되었다.

```
>>> s1 = set([1, 2, 3, 4, 5, 6])
>>> s2 = set([4, 5, 6, 7, 8, 9])
```

1. 교집합

s1과 s2의 교집합을 구해 보자.

```
>>> s1 & s2
{4, 5, 6}
```

“&” 기호를 이용하면 교집합을 간단히 구할 수 있다.

또는 다음과 같이 intersection 함수를 사용해도 동일한 결과를 리턴한다.

```
>>> s1.intersection(s2)
{4, 5, 6}
```

s2.intersection(s1)을 사용해도 결과는 같다.

2. 합집합

합집합은 다음과 같이 구할 수 있다. 이때 4, 5, 6처럼 중복해서 포함된 값은 한 개씩만 표현된다.

```
>>> s1 | s2
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

“|” 기호를 이용한 방법이다.

```
>>> s1.union(s2)
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

또는 union 함수를 이용하면 된다. 교집합에서 사용했던 intersection 함수와 마찬가지로 s2.union(s1)을 사용해도 동일한 결과를 리턴한다.

3. 차집합

차집합은 다음과 같이 구할 수 있다.

```
>>> s1 - s2
{1, 2, 3}
>>> s2 - s1
{8, 9, 7}
```

빼기(-) 기호를 이용한 방법이다.

```
>>> s1.difference(s2)
{1, 2, 3}
>>> s2.difference(s1)
{8, 9, 7}
```

difference 함수를 이용해도 차집합을 구할 수 있다.

집합 자료형 관련 함수들

값 1개 추가하기(add)

이미 만들어진 set 자료형에 값을 추가할 수 있다. 1개의 값만 추가(add)할 경우에는 아래와 같이 한다.

```
>>> s1 = set([1, 2, 3])
>>> s1.add(4)
>>> s1
{1, 2, 3, 4}
```

값 여러 개 추가하기(update)

여러 개의 값을 한꺼번에 추가(update)할 때는 다음과 같이 하면 된다.

```
>>> s1 = set([1, 2, 3])
>>> s1.update([4, 5, 6])
>>> s1
{1, 2, 3, 4, 5, 6}
```

특정 값 제거하기(remove)

특정 값을 제거하고 싶을 때는 아래와 같이 하면 된다.

```
>>> s1 = set([1, 2, 3])
>>> s1.remove(2)
>>> s1
{1, 3}
```

지금까지 파이썬의 가장 기본이 되는 자료형인 숫자, 문자열, 리스트, 튜플, 딕셔너리, 집합에 대해서 알아보았다. 여기까지 잘 따라온 독자라면 파이썬에 대해서 대략 50% 정도 습득했다고 보아도 된다. 그만큼 자료형은 중요하고 프로그램의 근간이 되기 때문에 확실하게 해놓지 않으면 좋은 프로그램을 만들 수 없다. 책에 있는 예제들만 따라 하지 말고 직접 여러 가지 예들을 테스트해보며 02-1부터 02-6절까지의 자료형들에 익숙해지기를 당부한다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-6>)

[문제1] 집합 만들기1

[‘a’, ‘b’, ‘c’]라는 리스트를 집합 자료형으로 만드시오.

[문제2] 집합의 중복

중복을 허용하지 않는 집합 자료형의 특징을 이용하여 다음 a 리스트에서 중복된 숫자들을 제거해 보자.

```
>>> a = [1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5]
```

[문제3] 차집합

다음과 같은 2개의 집합 자료형이 있다.

```
>>> s1 = set(['a', 'b', 'c', 'd', 'e'])
>>> s2 = set(['c', 'd', 'e', 'f', 'g'])
```

s1 집합의 항목 중 s2 집합에 포함된 항목을 제거 해 보자.

[문제4] 집합 만들기2

집합 자료형은 다음과 같이 만들 수 있다.

```
>>> a = {'a', 'b', 'c'}
>>> a
{'a', 'b', 'c'}
>>> type(a)
<class 'set'>
```

값이 하나도 없는 비어있는 set을 만들기 위해 다음과 같이 시도 해 보자.

```
>>> a = {}
>>> type(a)
<class 'dict'>
```

위와 같이 값이 있을 경우에는 집합 자료형으로 인식했지만 값이 없을경우에는 덕셔너리로 인식하게 된다. 그렇다면 값이 비어있는 집합 자료형은 어떻게 만들 수 있을까?

[문제5] 집합 추가

다음과 같은 집합 자료형 a가 있다.

```
>>> a = {'a', 'b', 'c'}
```

a에 {'d', 'e', 'f'}를 추가하시오.

02-7 불 자료형

불 자료형이란?

불(bool) 자료형이란 참(True)과 거짓(False)을 나타내는 자료형이다.

불 자료형은 다음의 2가지 값만을 가질 수 있다.

- True - 참
- False - 거짓

※ True나 False는 파이썬의 예약어로 true, false와 같이 사용하지 말고 첫 문자를 항상 대문자로 사용해야 한다.

다음은 불 자료형을 사용하는 예제이다.

```
>>> a = True
>>> b = False
```

a라는 변수에는 True를 대입하고 b라는 변수에는 False라는 불 자료형을 대입해 보았다.

불 자료형은 조건문의 리턴값으로도 사용된다. 조건문에 대해서는 if문에서 자세히 배우겠지만 잠시만 살펴보고 넘어가도록 하자.

```
>>> 1 == 1
True
```

`1 == 1` 은 “1과 1이 같은가?”를 묻는 조건문이다. 이런 조건문은 결과로 True 또는 False에 해당되는 불 자료형을 리턴하게 된다. 1과 1은 같으므로 True를 리턴한다.

```
>>> 2 > 1
True
```

2는 1보다 크기 때문에 `2 > 1` 라는 조건문은 True를 리턴한다.

```
>>> 2 < 1
False
```

2는 1보다 작지 않기 때문에 `2 < 1` 라는 조건문은 False를 리턴한다.

자료형의 참과 거짓

자료형에 참과 거짓이 있다? 조금 이상하게 들리겠지만 참과 거짓은 분명히 있다. 이는 매우 중요한 특징이며 실제로도 자주 쓰인다.

자료형의 참과 거짓을 구분하는 기준은 다음과 같다.

값	참 or 거짓
“python”	참
“”	거짓
[1, 2, 3]	참
[]	거짓
()	거짓
{}	거짓
1	참
0	거짓
None	거짓

문자열, 리스트, 튜플, 딕셔너리 등의 값이 비어 있으면(”, “[], (), { }) 거짓이 된다. 당연히 비어있지 않으면 참이 된다. 숫자에서는 그 값이 0일 때 거짓이 된다. 위의 표를 보면 None이라는 것이 있는데, 이것에 대해서는 뒷부분에서 배우니 아직은 신경 쓰지 말자. 그저 None은 거짓을 뜻한다는 것만 알아두자.

다음의 예를 보고 참과 거짓이 프로그램에서 어떻게 쓰이는지 간단히 알아보자.

```
>>> a = [1, 2, 3, 4]
>>> while a:
...     print(a.pop())
...
4
3
2
1
```

먼저 `a = [1, 2, 3, 4]`라는 리스트를 하나 만들었다.

while문은 3장에서 자세히 다루겠지만 간단히 알아보면 다음과 같다. 조건문이 참인 동안 조건문 안에 있는 문장을 반복해서 수행한다.

while 조건문:

수행할 문장

즉, 위의 예를 보면 a가 참인 경우에 a.pop()을 계속 실행하라는 의미이다. a.pop()이라는 함수는 리스트 a의 마지막 요소를 끄집어내는 함수이므로 a가 참인 동안(리스트 내에 요소가 존재하는 한) 마지막 요소를 계속해서 끄집어낼 것이다. 결국 더 이상 끄집어낼 것이 없으면 a가 빈 리스트 ([]가 되어 거짓이 된다. 따라서 while문에서 조건이 거짓이 되므로 중지된다. 위에서 본 예는 파이썬 프로그래밍에서 매우 자주 이용하는 기법 중 하나이다.

위의 예가 너무 복잡하다고 생각하는 독자는 다음의 예를 보면 쉽게 이해가 될 것이다.

```
>>> if []:
...     print("참")
... else:
...     print("거짓")
...
거짓
```

if문에 대해서 잘 모르는 독자라도 위의 문장을 해석하는 데는 무리가 없을 것이다.

※ if문에 대해서는 03장에서 자세히 다룬다.

[]는 앞의 표에서 볼 수 있듯이 비어 있는 리스트이므로 거짓이다. 따라서 “거짓”이라는 문자열이 출력된다.

```
>>> if [1, 2, 3]:
...     print("참")
... else:
...     print("거짓")
...
참
```

위 코드를 해석해 보면 다음과 같다.

만약 [1, 2, 3]이 참이면 "참"이라는 문자열을 출력하고 그렇지 않으면 "거짓"이라는 문자열을 출력하라.

위 코드의 [1, 2, 3]은 요소값이 있는 리스트이기 때문에 참이다. 따라서 “참”을 출력한다.

불 연산

자료형에 참과 거짓이 있음을 이미 알아보았다. bool이라는 내장 함수를 이용하면 자료형의 참과 거짓을 식별할 수 있다.

다음의 예제를 따라 해 보자.

```
>>> bool('python')
True
```

'python'이라는 문자열은 빈 문자열이 아니므로 bool연산의 결과로 불 자료형인 True를 리턴한다.

```
>>> bool('')
False
```

”이라는 문자열은 빈 문자열이므로 bool 연산의 결과로 불 자료형인 False를 리턴한다.

위에서 알아본 몇 가지 예제를 더 수행 해 보자.

```
>>> bool([1,2,3])
True
>>> bool([])
False
>>> bool(0)
False
>>> bool(3)
True
```

위에서 알아본 것과 동일한 참과 거짓에 대한 결과가 리턴되는 것을 확인할 수 있다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-7>)

[문제1] 불 자료형과 조건문

다음은 불 자료형을 리턴하는 조건문들이다. 각 각의 예제의 결과가 어떻게 나오는지 예상해 보자.

```
>>> 1 != 1  
>>> 3 > 1  
>>> 'a' in 'abc'  
>>> 'a' not in [1, 2, 3]
```

[문제2] bool 연산자

bool 연산자를 이용하여 다음 자료형들의 참과 거짓을 판별하시오.

```
>>> a = "python"  
>>> b = ""  
>>> c = (1,2,3)  
>>> d = 0
```

02-8 자료형의 값을 저장하는 공간, 변수

지금부터 설명할 내용은 프로그래밍 초보자가 얼른 이해하기 어려운 부분이므로 당장 이해가 되지 않는다면 그냥 건너뛰어도 무방하다. 파이썬에 대해서 공부하다 보면 자연스럽게 알게 될 것이다.

변수는 어떻게 만들까?

우리는 앞에서 이미 변수들을 사용해 왔다. 다음 예와 같은 a, b, c를 변수라고 한다.

```
>>> a = 1
>>> b = "python"
>>> c = [1,2,3]
```

변수를 만들 때는 위의 예처럼 =(assignment) 기호를 사용한다.

C 언어나 Java처럼 변수의 자료형을 함께 쓸 필요는 없다. 파이썬은 변수에 저장된 값을 스스로 판단하여 자료형을 알아낸다.

변수명 = 변수에 저장할 값

변수란?

파이썬에서 사용하는 변수는 객체를 가리키는 것이라고도 말할 수 있다. 객체란 우리가 지금껏 보아 왔던 자료형과 같은 것을 의미하는 말이다. (객체에 대해서는 클래스 챕터에서 보다 자세하게 공부한다.)

```
>>> a = [1, 2, 3]
```

만약 위의 코드처럼 a = [1, 2, 3]이라고 하면 [1, 2, 3]이라는 값을 가지는 리스트 자료형(객체)이 자동으로 메모리에 생성되고 변수 a는 [1, 2, 3]이라는 리스트가 저장된 메모리의 주소를 가리키게 된다.

a변수가 가리키는 메모리의 주소는 다음과 같이 확인할 수 있다.

```
>>> a = [1, 2, 3]
>>> id(a)
4303029896
```

`id` 함수는 변수가 가리키고 있는 객체의 주소를 리턴해 주는 파이썬 내장함수이다. 즉, `a`가 가리키고 있는 `[1, 2, 3]`이라는 리스트의 주소는 `4303029896`임을 알 수 있다.

리스트를 변수에 넣고 복사하고자 할 때

여기서는 리스트 자료형에서 가장 혼동하기 쉬운 “복사”에 대해 설명하려고 한다. 다음 예를 통해 알아보자.

```
>>> a = [1,2,3]
>>> b = a
```

`b`변수에 `a`변수를 대입하면 어떻게 될까? `b`와 `a`는 같은걸까? 아니면 다른 걸까? 결론부터 얘기하면 `b`는 `a`와 완전히 동일하다고 할 수 있다. 다만 `[1, 2, 3]`이라는 리스트를 참조하는 변수의 갯수가 `a`변수 1개에서 `b`변수가 추가되어 2개가 되었다는 차이가 있을 뿐이다.

`id` 명령을 이용하면 이러한 사실을 증명할 수 있다.

```
>>> id(a)
4303029896
>>> id(b)
4303029896
```

`id(a)`의 값이 `id(b)`의 값과 동일함을 확인할 수 있다. 즉, `a`가 가리키는 대상과 `b`가 가리키는 대상이 동일하다는 것을 알수 있다.

동일한 객체를 가리키고 있는지 아닌지에 대해서 판단하는 파이썬 명령어인 `is`를 다음과 같이 실행해도 역시 참(True)을 리턴하게 된다.

```
>>> a is b # a와 b가 가리키는 객체는 동일한가?
True
```

이제 다음 예를 계속해서 수행해 보자.

```
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> b
[1, 4, 3]
```

a 리스트의 두번째 요소를 4라는 값으로 바꾸었더니 a만 바뀌는 것이 아니라 b도 똑같이 바뀌었다. 그 이유는 앞서 살펴본 것처럼 a, b 모두 동일한 리스트를 가리키고 있기 때문이다.

그렇다면 b 변수를 생성할 때 a와 같은 값을 가지도록 복사해 넣으면서 a가 가리키는 리스트와는 다른 리스트를 가리키게 하는 방법은 없을까? 다음의 2가지 방법이 있다.

1. [:] 이용

첫 번째 방법으로는 아래와 같이 리스트 전체를 가리키는 [...]을 이용해서 복사하는 것이다.

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> b
[1, 2, 3]
```

위의 예에서 볼 수 있듯이 a 리스트 값을 바꾸더라도 b 리스트에는 영향을 끼치지 않는다.

2. copy 모듈 이용

두 번째는 copy 모듈을 이용하는 방법이다. 아래 예를 보면 from copy import copy라는 처음 보는 형태가 나오는데, 이것은 뒤에 설명할 파일 모듈 부분에서 자세히 다룬다. 여기서는 단순히 copy라는 함수를 쓰기 위해서 사용되는 것이라고만 알아두자.

```
>>> from copy import copy
>>> b = copy(a)
```

위의 예에서 b = copy(a)는 b = a[:]과 동일하다.

두 변수가 같은 값을 가지면서 다른 객체를 제대로 생성했는지 다음과 같이 확인 해 보자.

```
>>> b is a
False
```

위의 예에서 `b is a` 가 `False`를 리턴하므로 `b`와 `a`가 가리키는 객체는 서로 다르다는 것을 알 수 있다.

변수를 만드는 여러 가지 방법

```
>>> a, b = ('python', 'life')
```

위의 예문처럼 투플로 `a`, `b`에 값을 대입할 수 있다. 이 방법은 다음 예문과 완전히 동일하다.

```
>>> (a, b) = 'python', 'life'
```

튜플 부분에서도 언급했지만 투플은 괄호를 생략해도 된다.

아래처럼 리스트로 변수를 만들 수도 있다.

```
>>> [a,b] = ['python', 'life']
```

또한 여러 개의 변수에 같은 값을 대입할 수도 있다.

```
>>> a = b = 'python'
```

파이썬에서는 위의 방법을 이용하여 두 변수의 값을 아주 간단히 바꿀 수 있다.

```
>>> a = 3
>>> b = 5
>>> a, b = b, a
>>> a
5
>>> b
3
```

처음에 a에 3, b에는 5라는 값이 대입되어 있었지만 a, b = b, a라는 문장을 수행한 후에는 그 값이 서로 바뀌었음을 확인할 수 있다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#02-8>)

[문제1] 변수와 객체1

다음 예제를 실행하고 그 결과를 설명하시오.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
```

[문제2] 변수와 객체2

다음 예제를 실행하고 그 결과를 설명하시오.

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
```

[문제3] 객체의 변경

파이썬은 다음처럼 동일한 값에 여러 개의 변수를 선언할 수 있다. 아래와 같이 a, b 변수를 선언한 후 a의 첫 번째 요소값을 변경하면 b의 값은 어떻게 될까? 그리고 이런 결과가 나오는 이유에 대해서 설명해 보자.

```
>>> a = b = [1, 2, 3]
>>> a[1] = 4
>>> print(b)
```

[문제4] 리스트 복사1

다음 예제를 실행하고 그 결과를 설명하시오.

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a is b
```

[문제5] 리스트 복사2

b리스트는 a리스트를 copy하여 다음과 같이 생성하였다.

```
>>> a = [1, 2, 3]
>>> b = a[:]
```

그리고 다음과 같이 a리스트의 두번째 요소값을 2에서 4로 바꾸었다.

```
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> print(b)
```

이제 b리스트를 출력하면 어떤 값이 출력될까? 그리고 그런 결과값이 나오는 이유에 대해서 설명 하시오.

[문제6] 리스트의 더하기와 extend

다음과 같은 리스트 a가 있다.

```
>>> a = [1, 2, 3]
```

리스트 a에 [4, 5]를 + 기호를 이용하여 더한 결과는 다음과 같다.

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

리스트 a에 [4,5]를 extend를 이용하여 더한 결과는 다음과 같다.

```
>>> a = [1, 2, 3]
>>> a.extend([4, 5])
>>> a
[1, 2, 3, 4, 5]
```

+ 기호를 이용하여 더한것과 extend한 것의 차이점이 있을까? 있다면 그 차이점에 대해서 얘기해 보자.

[문제7] 리스트 복사3

a리스트는 리스트내에 [2, 3]이라는 리스트를 하나 더 포함하고 있는 리스트이다. 이 a리스트를 copy하여 b리스트를 다음과 같이 만들었다.

```
>>> a = [1, [2, 3], 4]
>>> b = a[:]
```

그리고 다음과 같이 a 리스트에 포함된 [2, 3]의 첫번째 값을 2에서 5로 바꾸어 주었다. 이렇게 a리스트에 포함된 리스트의 요소값을 변경하면 b 리스트는 어떻게 될까? 그리고 그런 결과값이 나오는 이유에 대해서 설명하시오.

```
>>> a[1][0] = 5
>>> a
[1, [5, 3], 4]
>>> print(b)
```

03장 프로그램의 구조를 쌓는다! 제어문

이번 장에서는 if, while, for 등의 제어문에 대해서 배우고자 한다. 제어문을 배우기 전에 집을 짓는 일을 생각해 보자.

집을 지을 때 나무, 돌, 시멘트 같은 재료가 되고, 철근은 집의 뼈대가 된다. 프로그램을 만드는 것도 집 짓기와 매우 비슷한 면이 있다. 나무, 돌, 시멘트와 같은 재료는 자료형이 되고, 집의 뼈대를 이루는 철근은 이번 장에서 알아볼 제어문에 해당한다. 이번 장에서는 자료형을 바탕으로 제어문을 이용하여 프로그램의 구조를 만들어 보자.

03-1 if문

if문은 왜 필요할까?

다음과 같은 상상을 해보자.

“돈이 있으면 택시를 타고, 돈이 없으면 걸어 간다.”

우리 모두에게 언제든지 일어날 수 있는 상황 중 하나이다. 프로그래밍도 사람이 하는 것이므로 위의 문장처럼 주어진 조건을 판단한 후 그 상황에 맞게 처리해야 할 경우가 생긴다. 이렇듯 프로그래밍에서 조건을 판단하여 해당 조건에 맞는 상황을 수행하는 데 쓰이는 것이 바로 if 문이다.

위와 같은 상황을 파이썬에서는 다음과 같이 표현할 수 있다.

```
>>> money = 1
>>> if money:
...     print("택시를 타고 가라")
... else:
...     print("걸어 가라")
...
택시를 타고 가라
```

money에 입력된 1은 참이다. 따라서 if문 다음의 문장이 수행되어 '택시를 타고 가라'가 출력된다.

if문의 기본 구조

다음은 if와 else를 이용한 조건문의 기본 구조이다.

```
if 조건문:
    수행할 문장1
    수행할 문장2
    ...
else:
    수행할 문장A
    수행할 문장B
    ...
```

조건문을 테스트해서 참이면 if문 바로 다음의 문장(if 블록)들을 수행하고, 조건문이 거짓이면 else

문 다음의 문장(else 블록)들을 수행하게 된다. 그러므로 else문은 if문 없이 독립적으로 사용할 수 없다.

들여쓰기

if문을 만들 때는 if 조건문: 바로 아래 문장부터 if문에 속하는 모든 문장에 들여쓰기(indentation)를 해주어야 한다. 다음과 같이 조건문이 침입 경우 “수행할 문장1”을 들여쓰기 했고 “수행할 문장2”와 “수행할 문장3”도 들여쓰기를 해주었다. 다른 프로그래밍 언어를 사용했던 사람들은 파일에서 “수행할 문장”들을 들여쓰기 하는 것을 무시하는 경우가 많으니 더 주의해야 한다.

```
if 조건문:
    수행할 문장1
    수행할 문장2
    수행할 문장3
```

다음과 같이 작성하면 오류가 발생한다. “수행할 문장2”를 들여쓰기 하지 않았기 때문이다.

```
if 조건문:
    수행할 문장1
    수행할 문장2
    수행할 문장3
```

다음 예제를 살펴보면 무슨 맥인지 이해할 수 있을 것이다.

```
>>> money = 1
>>> if money:
...     print("택시를")
...     print("타고")
File "<stdin>", line 3
    print("타고")
^
SyntaxError: invalid syntax
```

다음과 같은 경우에도 에러가 발생한다. “수행할 문장3”을 들여쓰기 했지만 “수행할 문장1”이나 “수행할 문장2”와 들여쓰기의 깊이가 다르다. 즉, 들여쓰기는 언제나 같은 깊이로 해야 한다.

```
if 조건문:
    수행할 문장1
    수행할 문장2
    수행할 문장3
```

다음의 예를 보면 들여쓰기 오류(IndentationError)가 발생한 것을 알 수 있다.

```
>>> money = 1
>>> if money:
...     print("택시를")
...     print("타고")
...     print("가자")
File "<stdin>", line 4
    print("가자")
^
IndentationError: unexpected indent
```

그렇다면 들여쓰기는 공백[Spacebar]으로 하는 것이 좋을까? 아니면 템[Tab]으로 하는 것이 좋을까? 이에 대한 논란은 파이썬을 사용하는 사람들 사이에서 아직도 계속되고 있다. 템으로 하자는 쪽과 공백으로 하자는 쪽 모두가 동의하는 내용은 단 하나, 2가지를 혼용해서 쓰지는 말자는 것이다. 공백으로 할 거면 항상 공백으로 통일하고, 템으로 할 거면 항상 템으로 통일해서 사용하자는 말이다. 템이나 공백은 프로그램 소스에서 눈으로 보이는 것이 아니기 때문에 혼용해서 쓰면 에러의 원인이 되기도 한다. 주의하도록 하자.

※ 요즘 파이썬 커뮤니티에서는 들여쓰기를 할 때 공백(Spacebar) 4개를 사용하는 것을 권장한다.

[조건문 다음에 콜론(:)을 잊지 말자!]

if 조건문 뒤에는 반드시 콜론(:)이 붙는다. 어떤 특별한 의미가 있다기보다는 파이썬의 문법 구조이다. 왜 하필 콜론(:)인지 궁금하다면 파이썬을 만든 귀도에게 직접 물어보아야 할 것이다. 앞으로 배우게 될 while이나 for, def, class문에도 역시 문장의 끝에 콜론(:)이 항상 들어간다. 초보자들은 이 콜론(:)을 빠뜨리는 경우가 많으니 특히 주의하자.

파이썬이 다른 언어보다 보기 쉽고 소스 코드가 간결한 이유는 바로 콜론(:)을 사용하여 들여쓰기(indentation)를 하도록 만들었기 때문이다. 하지만 이는 숙련된 프로그래머들이 파이썬을 처음 접할 때 제일 혼란스러워하는 부분이기도 하다. 다른 언어에서는 if문을 { } 기호로 감싸지만 파이썬에서는 들여쓰기로 해결한다는 점을 기억하자.

조건문이란 무엇인가?

if 조건문에서 “조건문”이란 참과 거짓을 판단하는 문장을 말한다. 자료형의 참과 거짓에 대해서 몇 가지만 다시 살펴보면 다음과 같은 것들이 있다.

자료형	참	거짓
숫자	0이 아닌 숫자	0
문자열	“abc”	“”
리스트	[1,2,3]	[]
튜플	(1,2,3)	()
딕셔너리	{“a”:“b”}	{}

따라서 이전에 살펴보았던 택시 예제에서 조건문은 money가 된다.

```
>>> money = 1
>>> if money:
```

money는 1이기 때문에 참이 되어 if문 다음의 문장을 수행하게 된다.

비교연산자

조건이 참인지 거짓인지 판단할 때 자료형보다는 비교연산자(<, >, ==, !=, >=, <=)를 쓰는 경우가 훨씬 많다.

다음 표는 비교연산자를 잘 설명해 준다.

비교연산자	설명
x < y	x가 y보다 작다
x > y	x가 y보다 크다
x == y	x와 y가 같다
x != y	x와 y가 같지 않다
x >= y	x가 y보다 크거나 같다
x <= y	x가 y보다 작거나 같다

이제 위의 연산자들을 어떻게 사용하는지 알아보자.

```
>>> x = 3
>>> y = 2
>>> x > y
True
>>>
```

x에 3을, y에 2를 대입한 다음에 $x > y$ 라는 조건문을 수행하면 True를 리턴한다. $x > y$ 라는 조건문이 참이기 때문이다.

```
>>> x < y
False
```

위의 조건문은 거짓이기 때문에 False를 리턴한다.

```
>>> x == y
False
```

x와 y는 같지 않다. 따라서 위의 조건문은 거짓이다.

```
>>> x != y
True
```

x와 y는 같지 않다. 따라서 위의 조건문은 참이다.

앞에서 살펴본 택시 예제를 다음처럼 바꾸려면 어떻게 해야 할까?

“만약 3000원 이상의 돈을 가지고 있으면 택시를 타고 그렇지 않으면 걸어 가라”

위의 상황은 다음처럼 프로그래밍할 수 있다.

```
>>> money = 2000
>>> if money >= 3000:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
걸어가라
>>>
```

`money >= 3000`이라는 조건문이 거짓이 되기 때문에 `else`문 다음의 문장을 수행하게 된다.

and, or, not

조건을 판단하기 위해 사용하는 다른 연산자로는 `and`, `or`, `not`이 있다. 각각의 연산자는 다음처럼 동작한다.

연산자	설명
<code>x or y</code>	<code>x</code> 와 <code>y</code> 둘중에 하나만 참이면 참이다
<code>x and y</code>	<code>x</code> 와 <code>y</code> 모두 참이어야 참이다
<code>not x</code>	<code>x</code> 가 거짓이면 참이다

다음의 예를 통해 `or` 연산자의 사용법을 알아보자.

“돈이 3000원 이상 있거나 카드가 있다면 택시를 타고 그렇지 않으면 걸어 가라”

```
>>> money = 2000
>>> card = 1
>>> if money >= 3000 or card:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
>>>
```

money는 2000이지만 card가 1이기 때문에 `money >= 3000 or card`라는 조건문이 참이 된다. 따라서 if문 다음의 “택시를 타고 가라”라는 문장이 수행된다.

x in s, x not in s

더 나아가 파이썬은 다른 프로그래밍 언어에서 쉽게 볼 수 없는 재미있는 조건문들을 제공한다. 바로 다음과 같은 것들이다.

in	not in
<code>x in 리스트</code>	<code>x not in 리스트</code>
<code>x in 튜플</code>	<code>x not in 튜플</code>
<code>x in 문자열</code>	<code>x not in 문자열</code>

`in`이라는 영어 단어의 뜻이 “~안에”라는 것을 생각해 보면 다음 예들이 쉽게 이해될 것이다.

```
>>> 1 in [1, 2, 3]
True
>>> 1 not in [1, 2, 3]
False
```

앞에서 첫 번째 예는 “[1, 2, 3]이라는 리스트 안에 1이 있는가?”라는 조건문이다. 1은 [1, 2, 3] 안에 있으므로 참이 되어 `True`를 리턴한다. 두 번째 예는 “[1, 2, 3]이라는 리스트 안에 1이 없는가?”라는 조건문이다. 1은 [1, 2, 3] 안에 있으므로 거짓이 되어 `False`를 리턴한다.

다음은 튜플과 문자열에 적용한 예이다. 각각의 결과가 나온 이유는 쉽게 유추할 수 있다.

```
>>> 'a' in ('a', 'b', 'c')
True
>>> 'j' not in 'python'
True
```

이번에는 우리가 계속 사용해 온 택시 예제에 `in`을 적용해 보자.

“만약 주머니에 돈이 있으면 택시를 타고, 없으면 걸어 가라”

```
>>> pocket = ['paper', 'cellphone', 'money']
>>> if 'money' in pocket:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
>>>
```

[‘paper’, ‘cellphone’, ‘money’]라는 리스트 안에 ‘money’가 있으므로 ‘money’ in pocket은 참이 된다. 따라서 if문 다음의 문장이 수행된다.

[조건문에서 아무 일도 하지 않게 설정하고 싶다면?]

가끔 조건문의 참, 거짓에 따라 실행할 행동을 정의할 때, 아무런 일도 하지 않도록 설정하고 싶을 때가 있다. 다음의 예를 보자.

“주머니에 돈이 있으면 가만히 있고 주머니에 돈이 없으면 카드를 꺼내라”

이럴 때 사용하는 것이 바로 pass이다. 위의 예를 pass를 적용해서 구현해 보자.

```
>>> pocket = ['paper', 'money', 'cellphone']
>>> if 'money' in pocket:
...     pass
... else:
...     print("카드를 꺼내라")
...
```

pocket이라는 리스트 안에 money라는 문자열이 있기 때문에 if문 다음 문장인 pass가 수행되고 아무런 결과값도 보여 주지 않는다.

다양한 조건을 판단하는 elif

if와 else만으로는 다양한 조건을 판단하기 어렵다. 다음과 같은 예를 보더라도 if와 else만으로는 조건을 판단하는 데 어려움을 겪게 된다.

“주머니에 돈이 있으면 택시를 타고, 주머니에 돈은 없지만 카드가 있으면 택시를 타고, 돈도 없고 카드도 없으면 걸어 가라”

위의 문장을 보면 조건을 판단하는 부분이 두 군데가 있다. 먼저 주머니에 돈이 있는지를 판단해야 하고 주머니에 돈이 없으면 다시 카드가 있는지 판단해야 한다.

if와 else만으로 위의 문장을 표현하려면 다음과 같이 할 수 있다.

```
>>> pocket = ['paper', 'handphone']
>>> card = 1
>>> if 'money' in pocket:
...     print("택시를 타고가라")
... else:
...     if card:
...         print("택시를 타고가라")
...     else:
...         print("걸어가라")
...
택시를 타고가라
>>>
```

언뜻 보기에도 이해하기 어렵고 산만한 느낌이 든다. 이런 복잡함을 해결하기 위해 파이썬에서는 다중 조건 판단을 가능하게 하는 elif라는 것을 사용한다.

위의 예를 elif를 사용하면 다음과 같이 바꿀 수 있다.

```
>>> pocket = ['paper', 'cellphone']
>>> card = 1
>>> if 'money' in pocket:
...     print("택시를 타고가라")
... elif card:
...     print("택시를 타고가라")
... else:
...     print("걸어가라")
...
택시를 타고가라
```

즉, elif는 이전 조건문이 거짓일 때 수행된다. if, elif, else를 모두 사용할 때 기본 구조는 다음과 같다.

```
If <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    ...  
elif <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    ...  
elif <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    ...  
...  
else:  
    <수행할 문장1>  
    <수행할 문장2>  
    ...
```

위에서 볼 수 있듯이 elif는 개수에 제한 없이 사용할 수 있다.

[if문을 한 줄로 작성하기]

앞의 pass를 사용한 예를 보면 if문 다음에 수행할 문장이 한 줄이고, else문 다음에 수행할 문장도 한 줄밖에 되지 않는다.

```
>>> if 'money' in pocket:  
...     pass  
... else:  
...     print("카드를 꺼내라")  
...
```

이렇게 수행할 문장이 한 줄일 때 조금 더 간략하게 코드를 작성하는 방법이 있다.

```
>>> pocket = ['paper', 'money', 'cellphone']  
>>> if 'money' in pocket: pass  
... else: print("카드를 꺼내라")  
...
```

if문 다음의 수행할 문장을 콜론(:) 뒤에 바로 적어 주었다. else문 역시 마찬가지이다. 때때로 이렇게 작성하는 것이 보기 편할 수 있다.

조건부 표현식

다음과 같은 코드를 보자.

```
if score >= 60:
    message = "success"
else:
    message = "failure"
```

위 코드는 score가 60 이상일 경우 message에 “success”를 아닐 경우에는 “failure”를 대입하는 코드이다.

파이썬의 조건부 표현식(conditional expression)을 이용하면 위 코드를 다음과 같이 간단히 표현할 수 있다.

```
message = "success" if score >= 60 else "failure"
```

조건부 표현식은 다음과 같이 정의된다.

조건문이_참인_경우 if 조건문 else 조건문이_거짓인_경우

조건부 표현식은 가독성에 유리하고 한 라인으로 작성할 수 있어 활용성이 좋다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#03-1-if>)

[문제1] 조건문1

홍길동씨는 5,000원의 돈을 가지고 있고 카드는 없다고 한다. 이러한 홍길동씨의 상태는 아래와 같이 표현할 수 있을 것이다.

```
>>> money = 5000
>>> card = False
```

홍길동씨는 택시를 타고 목적지까지 가려고 한다. 목적지까지 가기 위해서는 카드를 소유하고 있거나 4,000원 이상의 현금을 가지고 있어야 한다고 한다. 홍길동씨는 택시를 탈 수 있는지를 판별할 수 있는 조건식을 작성하고 그 결과를 출력하시오.

[문제2] 조건문2

홍길동씨의 행운권 번호는 23번이라고 한다. 다음은 행운권 당첨번호 리스트이다.

```
>>> lucky_list = [1, 9, 23, 46]
```

홍길동씨가 당첨되었다면 “야호”라는 문자열을 출력하는 프로그램을 작성하시오.

[문제3] 홀수 짝수 판별

주어진 수가 짝수인지 홀수인지 판별하는 프로그램을 작성하시오.

[문제4] 문자열 분석

다음 문자열을 분석하여 나이가 30미만이고 키가 175이상인 경우에는 YES를 출력하고 아닌 경우에는 NO를 출력하는 프로그램을 작성하시오.

```
나이:30,키:180
```

[문제5] 조건문3

다음 코드의 결과값은 무엇일까?

```
>>> a = "Life is too short, you need python"
>>> if 'wife' in a:
...     print('wife')
... elif 'python' in a and 'you' not in a:
...     print('python')
... elif 'shirt' not in a:
...     print('shirt')
... elif 'need' in a:
...     print('need')
... else:
...     print('none')
```

03-2 while문

while문의 기본 구조

반복해서 문장을 수행해야 할 경우 while문을 사용한다. 그래서 while문을 반복문이라고도 부른다.

다음은 while문의 기본 구조이다.

```
while <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    <수행할 문장3>  
    ...
```

while문은 조건문이 참인 동안에 while문 아래에 속하는 문장들이 반복해서 수행된다.

“열 번 찍어 안 넘어 가는 나무 없다” 라는 속담을 파이썬 프로그램으로 만든다면 다음과 같이 될 것이다.

```
>>> treeHit = 0  
>>> while treeHit < 10:  
...     treeHit = treeHit +1  
...     print("나무를 %d번 찍었습니다." % treeHit)  
...     if treeHit == 10:  
...         print("나무 넘어갑니다.")  
...  
나무를 1번 찍었습니다.  
나무를 2번 찍었습니다.  
나무를 3번 찍었습니다.  
나무를 4번 찍었습니다.  
나무를 5번 찍었습니다.  
나무를 6번 찍었습니다.  
나무를 7번 찍었습니다.  
나무를 8번 찍었습니다.  
나무를 9번 찍었습니다.  
나무를 10번 찍었습니다.  
나무 넘어갑니다.
```

위의 예에서 while문의 조건문은 `treeHit < 10` 이다. 즉, `treeHit`가 10보다 작은 동안에 while 문 안의 문장들을 계속 수행한다. whlie문 안의 문장을 보면 제일 먼저 `treeHit = treeHit + 1`로 `treeHit` 값이 계속 1씩 증가한다. 그리고 나무를 `treeHit`번만큼 찍었음을 알리는 문장을 출력하고 `treeHit`가 10이 되면 “나무 넘어갑니다.”라는 문장을 출력한다. 그리고 나면 `treeHit < 10`이라는 조건문이 거짓이 되므로 while문을 빠져나가게 된다.

※ `treeHit = treeHit + 1` 은 프로그래밍을 할 때 매우 자주 사용하는 기법이다. `treeHit`의 값을 1만큼씩 증가시킬 목적으로 사용되며, `treeHit += 1` 처럼 사용되기도 한다.

다음은 while문이 반복되는 과정을 순서대로 정리한 표이다. 이렇게 긴 과정을 소스 코드 단 5 줄로 만들 수 있다니 놀랍지 않은가?

<code>treeHit</code>	조건문	조건판단	수행하는 문장	while문
0	<code>0 < 10</code>	참	나무를 1번 찍었습니다.	반복
1	<code>1 < 10</code>	참	나무를 2번 찍었습니다.	반복
2	<code>2 < 10</code>	참	나무를 3번 찍었습니다.	반복
3	<code>3 < 10</code>	참	나무를 4번 찍었습니다.	반복
4	<code>4 < 10</code>	참	나무를 5번 찍었습니다.	반복
5	<code>5 < 10</code>	참	나무를 6번 찍었습니다.	반복
6	<code>6 < 10</code>	참	나무를 7번 찍었습니다.	반복
7	<code>7 < 10</code>	참	나무를 8번 찍었습니다.	반복
8	<code>8 < 10</code>	참	나무를 9번 찍었습니다.	반복
9	<code>9 < 10</code>	참	나무를 10번 찍었습니다. 나무 넘어갑니다.	반복
10	<code>10 < 10</code>	거짓		종료

while문 직접 만들기

다음을 직접 입력해 보자. 여러 가지 선택지 중 하나를 선택해서 입력받는 예제이다. 먼저 다음과 같이 여러 줄짜리 문자열을 만들어 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> prompt = """
... 1. Add
... 2. Del
... 3. List
... 4. Quit
...
... Enter number: """
>>>
```

이어서 number라는 변수에 0을 먼저 대입한다. 이렇게 변수를 먼저 설정해 놓지 않으면 다음에 나올 while문의 조건문인 number != 4에서 변수가 존재하지 않는다는 에러가 발생한다.

```
>>> number = 0
>>> while number != 4:
...     print(prompt)
...     number = int(input())
...
1. Add
2. Del
3. List
4. Quit

Enter number:
```

while문을 보면 number가 4가 아닌 동안 prompt를 출력하고 사용자로부터 번호를 입력받는다. 다음의 결과 화면처럼 사용자가 4라는 값을 입력하지 않으면 계속해서 prompt를 출력한다.

※ 여기서 number = int(input())는 사용자의 숫자 입력을 받아들이는 것이라고만 알 아두자. int나 input 함수에 대한 내용은 뒤의 내장 함수 부분에서 자세하게 다룬다.

```
Enter number:
```

```
1
```

- 1. Add
- 2. Del
- 3. List
- 4. Quit

4를 입력하면 조건문이 거짓이 되어 while문을 빠져나가게 된다.

```
Enter number:
```

```
4
```

```
>>>
```

while문 강제로 빠져나가기

while문은 조건문이 참인 동안 계속해서 while문 안의 내용을 반복적으로 수행한다. 하지만 강제로 while문을 빠져나가고 싶을 때가 있다. 예를 들어 커피 자판기를 생각해 보자. 자판기 안에 커피가 충분히 있을 때에는 동전을 넣으면 커피가 나온다. 그런데 자판기가 제대로 작동하려면 커피가 얼마나 남았는지 항상 검사해야 한다. 만약 커피가 떨어졌다면 판매를 중단하고 “판매 중지”라는 문구를 사용자에게 보여주어야 한다. 이렇게 판매를 강제로 멈추게 하는 것이 바로 break문이다.

다음의 예는 커피 자판기 이야기를 파이썬 프로그램으로 표현해 본 것이다.

```
>>> coffee = 10
>>> money = 300
>>> while money:
...     print("돈을 받았으니 커피를 줍니다.")
...     coffee = coffee -1
...     print("남은 커피의 양은 %d개입니다." % coffee)
...     if not coffee:
...         print("커피가 다 떨어졌습니다. 판매를 중지합니다.")
...         break
...
```

money가 300으로 고정되어 있으므로 while money:에서 조건문인 money는 0이 아니기 때문에

항상 참이다. 따라서 무한히 반복되는 무한 루프를 돌게 된다. 그리고 while문의 내용을 한 번 수행할 때마다 coffee = coffee - 1에 의해서 coffee의 개수가 1개씩 줄어든다. 만약 coffee가 0이 되면 if not coffee:라는 문장에서 not coffee가 참이 되므로 if문 다음의 문장인 “커피가 다 떨어졌습니다. 판매를 중지합니다.”가 수행되고 break문이 호출되어 while문을 빠져나가게 된다.

break문 이용해 자판기 작동 과정 만들기

하지만 실제 자판기는 위의 예처럼 작동하지는 않을 것이다. 다음은 자판기의 실제 작동 과정과 비슷하게 만들어 본 예이다. 이해가 안 되더라도 걱정하지 말자. 아래의 예는 조금 복잡하니까 대화형 인터프리터를 이용하지 말고 에디터를 이용해서 작성해 보자.

```
# coffee.py

coffee = 10
while True:
    money = int(input("돈을 넣어 주세요: "))
    if money == 300:
        print("커피를 줍니다.")
        coffee = coffee -1
    elif money > 300:
        print("거스름돈 %d를 주고 커피를 줍니다." % (money -300))
        coffee = coffee -1
    else:
        print("돈을 다시 돌려주고 커피를 주지 않습니다.")
        print("남은 커피의 양은 %d개 입니다." % coffee)
    if not coffee:
        print("커피가 다 떨어졌습니다. 판매를 중지 합니다.")
        break
```

위의 프로그램 소스를 따로 설명하지는 않겠다. 여러분이 소스를 입력하면서 무슨 내용인지 이해할 수 있다면 지금껏 배운 if문이나 while문을 이해했다고 보면 된다. 만약 money = int(input("돈을 넣어 주세요:"))라는 문장이 이해되지 않는다면 이 문장은 사용자로부터 입력을 받는 부분이고 입력받은 숫자를 money라는 변수에 대입하는 것이라고만 알아두자.

이제 coffee.py 파일을 저장한 후 프로그램을 직접 실행해 보자. 아래와 같은 입력란이 나타난다.

```
C:\doit>python coffee.py
```

돈을 넣어 주세요:

입력란에 여러 숫자를 입력해 보면서 결과를 확인하자.

돈을 넣어 주세요: 500

거스름돈 200를 주고 커피를 줍니다.

돈을 넣어 주세요: 300

커피를 줍니다.

돈을 넣어 주세요: 100

돈을 다시 돌려주고 커피를 주지 않습니다.

남은 커피의 양은 8개입니다.

돈을 넣어 주세요:

[에디터에서 만든 프로그램을 실행할 때 오류가 발생한다면?]

만약 파이썬 2.7 버전을 사용한다면 자판기 예제의 소스 코드에서 `input("돈을 넣어 주세요:")` 대신 `raw_input("돈을 넣어 주세요:")`로 사용해야 하며 소스 코드 가장 첫 번째 줄에 다음과 같은 문장을 반드시 넣어 주어야만 한다.

```
# -*- coding: utf-8 -*-
```

파이썬 3 버전을 사용할 때도 아래와 같은 오류가 발생할 수 있다.

```
C:\doit>python coffee.py
```

```
File "coffee.py", line 9
```

```
SyntaxError: (unicode error) 'utf-8' codec can't decode byte 0xb9 in position
0:
```

```
    invalid start byte
```

파이썬 3 버전에서 이런 오류가 발생했다면 아마 파일을 저장할 때 파일 인코딩 타입이 utf-8이 아니었을 것이다. 파일을 저장할 때 파일 인코딩을 utf-8로 맞추면 프로그램이 제대로 실행된다.

while문의 맨 처음으로 돌아가기

while문 안의 문장을 수행할 때 입력된 조건을 검사해서 조건에 맞지 않으면 while문을 빠져나간다. 그런데 프로그래밍을 하다 보면 while문을 빠져나가지 않고 while문의 맨 처음(조건문)으로

다시 돌아가게 만들고 싶은 경우가 생기게 된다. 이때 사용하는 것이 바로 continue문이다.

만약 1부터 10까지의 숫자 중에서 홀수만 출력하는 것을 while문을 이용해서 작성한다고 생각해보자. 어떤 방법이 좋을까?

```
>>> a = 0
>>> while a < 10:
...     a = a+1
...     if a % 2 == 0: continue
...     print(a)
...
1
3
5
7
9
```

위의 예는 1부터 10까지의 숫자 중 홀수만을 출력하는 예이다. a가 10보다 작은 동안 a는 1만큼씩 계속 증가한다. `if a % 2 == 0`(a를 2로 나누었을 때 나머지가 0인 경우)이 참이 되는 경우는 a가 짝수일 때이다. 즉, a가 짝수이면 continue 문장을 수행한다. 이 continue문은 while문의 맨 처음(조건문: `a<10`)으로 돌아가게 하는 명령어이다. 따라서 위의 예에서 a가 짝수이면 `print(a)`는 수행되지 않을 것이다.

무한 루프

이번에는 무한 루프(Loop)에 대해서 알아보자. 무한 루프란 무한히 반복한다는 의미이다. 우리가 사용하는 일반적인 프로그램 중에서 무한 루프의 개념을 사용하지 않는 프로그램은 거의 없다. 그 만큼 자주 사용된다는 뜻이다.

파이썬에서 무한 루프는 while문으로 구현할 수 있다. 다음은 무한 루프의 기본적인 형태이다.

```
while True:
    수행할 문장1
    수행할 문장2
    ...
    ...
```

while문의 조건문이 `True`이므로 항상 참이 된다. 따라서 while문 안에 있는 문장들은 무한하게 수행될 것이다.

다음의 무한 루프 예이다.

```
>>> while True:
...     print("Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.")
...
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
....
```

위의 문장은 영원히 출력된다. 하지만 이 예처럼 아무 의미 없이 무한 루프를 돌리는 경우는 거의 없을 것이다. [Ctrl+C]를 눌러 빠져나가도록 하자.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#03-2-while>)

[문제1] 1부터 100까지 더하기

1부터 100까지의 자연수를 모두 더하고 그 결과를 출력하시오.

[문제2] 3의 배수의 합

1부터 1000까지의 자연수 중 3의 배수의 합을 구하시오.

[문제3] 50점 이상의 총합

다음은 A학급 학생의 점수를 나타내는 리스트이다. 다음 리스트에서 50점 이상의 점수들의 총합을 구하시오.

```
A = [20, 55, 67, 82, 45, 33, 90, 87, 100, 25]
```

[문제4] 별 표시하기1

while문을 이용하여 아래와 같이 별(*)을 표시하는 프로그램을 작성해 보자.

```
*
```

```
**
```

```
***
```

```
****
```

[문제5] 별 표시하기2

while문을 이용하여 아래와 같이 별(*)을 표시하는 프로그램을 작성해 보자.

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

03-3 for문

파이썬의 직관적인 특징을 가장 잘 대변해 주는 것이 바로 이 for문이다. while문과 비슷한 반복문인 for문은 매우 유용하고 문장 구조가 한눈에 쑥 들어온다는 장점이 있다. for문을 잘 사용하면 프로그래밍이 때때로 즐거워질 것이다.

for문의 기본 구조

for문의 기본적인 구조는 다음과 같다.

```
for 변수 in 리스트(또는 튜플, 문자열):
    수행할 문장1
    수행할 문장2
    ...
    ...
```

리스트나 튜플, 문자열의 첫 번째 요소부터 마지막 요소까지 차례로 변수에 대입되어 “수행할 문장1”, “수행할 문장2” 등이 수행된다.

예제 이용해 for문 이해하기

for문은 예제를 통해서 살펴보는 것이 가장 알기 쉽다. 다음 예제를 직접 입력해 보자.

1. 전형적인 for문

```
>>> test_list = ['one', 'two', 'three']
>>> for i in test_list:
...     print(i)
...
one
two
three
```

['one', 'two', 'three']라는 리스트의 첫 번째 요소인 'one'이 먼저 i 변수에 대입된 후 print(i)라는 문장을 수행한다. 다음에 'two'라는 두 번째 요소가 i 변수에 대입된 후 print(i) 문장을 수행하고 리스트의 마지막 요소까지 이것을 반복한다.

2. 다양한 for문의 사용

```
>>> a = [(1,2), (3,4), (5,6)]
>>> for (first, last) in a:
...     print(first + last)
...
3
7
11
```

위의 예는 a 리스트의 요소값이 튜플이기 때문에 각각의 요소들이 자동으로 (first, last)라는 변수에 대입된다.

※ 이 예는 02장에서 살펴보았던 튜플을 이용한 변수값 대입 방법과 매우 비슷한 경우이다. >>> (first, last) = (1, 2)

3. for문의 응용

for문의 쓰임새를 알기 위해 다음을 가정해 보자.

“총 5명의 학생이 시험을 보았는데 시험 점수가 60점이 넘으면 합격이고 그렇지 않으면 불합격이다. 합격인지 불합격인지 결과를 보여주시오.”

우선 학생 5명의 시험 점수를 리스트로 표현해 보았다.

```
marks = [90, 25, 67, 45, 80]
```

1번 학생은 90점이고 5번 학생은 80점이다.

이런 점수를 차례로 검사해서 합격했는지 불합격했는지 통보해 주는 프로그램을 만들어 보자. 역시 에디터로 작성한다.

```
# marks1.py
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number +1
    if mark >= 60:
        print("%d번 학생은 합격입니다." % number)
    else:
        print("%d번 학생은 불합격입니다." % number)
```

각각의 학생에게 번호를 붙여 주기 위해 number라는 변수를 이용하였다. 점수 리스트인 marks에서 차례로 점수를 꺼내어 mark라는 변수에 대입하고 for문 안의 문장들을 수행하게 된다. 우선 for문이 한 번씩 수행될 때마다 number는 1씩 증가한다.

이 프로그램을 실행하면 mark가 60 이상일 때 합격 메시지를 출력하고 60을 넘지 않을 때 불합격 메시지를 출력한다.

```
C:\doit>python marks1.py
1번 학생은 합격입니다.
2번 학생은 불합격입니다.
3번 학생은 합격입니다.
4번 학생은 불합격입니다.
5번 학생은 합격입니다.
```

for문과 continue

while문에서 살펴보았던 continue를 for문에서도 사용할 수 있다. 즉, for문 안의 문장을 수행하는 도중에 continue문을 만나면 for문의 처음으로 돌아가게 된다.

앞서 for문 응용 예제를 그대로 이용해서 60점 이상인 사람에게는 축하 메시지를 보내고 나머지 사람에게는 아무런 메시지도 전하지 않는 프로그램을 에디터를 이용해 작성해 보자.

```
# marks2.py
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number +1
    if mark < 60: continue
    print("%d번 학생 축하합니다. 합격입니다. " % number)
```

점수가 60점 이하인 학생일 경우에는 `mark < 60`이 참이 되어 continue문이 수행된다. 따라서 축하 메시지를 출력하는 부분인 print문을 수행하지 않고 for문의 처음으로 돌아가게 된다.

```
C:\doit>python marks2.py
1번 학생 축하합니다. 합격입니다.
3번 학생 축하합니다. 합격입니다.
5번 학생 축하합니다. 합격입니다.
```

for와 함께 자주 사용하는 range함수

for문은 숫자 리스트를 자동으로 만들어 주는 range라는 함수와 함께 사용되는 경우가 많다. 다음은 range 함수의 간단한 사용법이다.

```
>>> a = range(10)
>>> a
range(0, 10)
```

range(10)은 0부터 10 미만의 숫자를 포함하는 range 객체를 만들어 준다.

시작 숫자와 끝 숫자를 지정하려면 range(시작 숫자, 끝 숫자) 형태를 사용하는데, 이때 끝 숫자는 포함되지 않는다.

```
>>> a = range(1, 11)
>>> a
range(1, 11)
```

range 함수의 예시 살펴보기

for와 range 함수를 이용하면 1부터 10까지 더하는 것을 다음과 같이 쉽게 구현할 수 있다.

```
>>> sum = 0
>>> for i in range(1, 11):
...     sum = sum + i
...
>>> print(sum)
55
```

range(1, 11)은 숫자 1부터 10까지(1 이상 11 미만)의 숫자를 데이터로 갖는 객체이다. 따라서

위의 예에서 i 변수에 리스트의 숫자들이 1부터 10까지 하나씩 차례로 대입되면서 sum = sum + i라는 문장을 반복적으로 수행하고 sum은 최종적으로 55가 된다.

또한 우리가 앞서 살펴보았던 60점 이상이면 합격이라는 문장을 출력하는 예제도 range 함수를 이용해서 바꿀 수 있다. 다음을 보자.

```
#marks3.py
marks = [90, 25, 67, 45, 80]
for number in range(len(marks)):
    if marks[number] < 60: continue
    print("%d번 학생 축하합니다. 합격입니다." % (number+1))
```

여기서 len이라는 함수가 처음 나왔다. len 함수는 리스트 내 요소의 개수를 돌려주는 함수이다. 따라서 len(marks)는 5가 될 것이고 range(len(marks))는 range(5)가 될 것이다. number 변수에는 차례로 0부터 4까지의 숫자가 대입될 것이고, marks[number]는 차례대로 90, 25, 67, 45, 80이라는 값을 갖게 된다. 결과는 marks2.py 예제와 동일하다.

for와 range를 이용한 구구단

for와 range 함수를 이용하면 소스 코드 단 4줄만으로 구구단을 출력할 수 있다. 들여쓰기에 주의 하며 입력해 보자.

```
>>> for i in range(2,10):
...     for j in range(1, 10):
...         print(i*j, end=" ")
...     print('')

...
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

위의 예를 보면 for문이 두 번 사용되었다. ①번 for문에서 2부터 9까지의 숫자(range(2, 10))가 차례로 i에 대입된다. i가 처음 2일 때 ②번 for문을 만나게 된다. ②번 for문에서 1부터 9까지의

숫자(range(1, 10))가 j에 대입되고 그 다음 문장인 `print(i*j)`를 수행한다.

따라서 i가 2일 때 $2*1$, $2*2$, $2*3$, ... $2*9$ 까지 차례로 수행되며 그 값을 출력하게 된다. 그 다음으로 i가 3일 때 역시 2일 때와 마찬가지로 수행될 것이고 i가 9일 때까지 계속 반복된다.

[입력 인수 end를 넣어 준 이유는 무엇일까?]

앞의 예제에서 `print(i*j, end=" ")`와 같이 입력 인수 end를 넣어 준 이유는 해당 결과값을 출력할 때 다음줄로 넘기지 않고 그 줄에 계속해서 출력하기 위해서이다. 그 다음에 이어지는 `print('')`는 2단, 3단 등을 구분하기 위해 두 번째 for문이 끝나면 결과값을 다음 줄부터 출력하게 해주는 문장이다.

만약 파이썬 2.7 버전을 사용한다면 `print(i*j, end=" ")` 대신 `print i*j,`와 같이 콤마(,)를 마지막에 넣어야 한다.

리스트 안에 for문 포함하기

리스트 안에 for문을 포함하는 리스트 내포(List comprehension)를 이용하면 좀 더 편리하고 직관적인 프로그램을 만들 수 있다. 다음의 예제를 보자.

```
>>> a = [1,2,3,4]
>>> result = []
>>> for num in a:
...     result.append(num*3)
...
>>> print(result)
[3, 6, 9, 12]
```

위 예제는 a라는 리스트의 각 항목에 3을 곱한 결과를 result라는 리스트에 담는 예제이다.

이것을 리스트 내포를 이용하면 아래와 같이 간단히 해결할 수 있다.

```
>>> result = [num * 3 for num in a]
>>> print(result)
[3, 6, 9, 12]
```

만약 짝수에만 3을 곱하여 담고 싶다면 다음과 같이 “if 조건”을 사용할 수 있다.

* 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> result = [num * 3 for num in a if num % 2 == 0]
>>> print(result)
[6, 12]
```

리스트 내포의 일반적인 문법은 다음과 같다. “if 조건” 부분은 앞의 예제에서 볼 수 있듯이 생략할 수 있다.

[표현식 for 항목 in 반복가능객체 if 조건]

조금 복잡하지만 for문을 2개 이상 사용하는 것도 가능하다. for문을 여러 개 사용할 때의 문법은 다음과 같다.

[표현식 for 항목1 in 반복가능객체1 if 조건1
 for 항목2 in 반복가능객체2 if 조건2
 ...
 for 항목n in 반복가능객체n if 조건n]

만약 구구단의 모든 결과를 리스트에 담고 싶다면 리스트 내포를 이용하여 아래와 같이 간단하게 구현할 수도 있다.

```
>>> result = [x*y for x in range(2,10)
...              for y in range(1,10)]
>>> print(result)
[2, 4, 6, 8, 10, 12, 14, 16, 18, 3, 6, 9, 12, 15, 18, 21, 24, 27, 4, 8, 12, 16,
20, 24, 28, 32, 36, 5, 10, 15, 20, 25, 30, 35, 40, 45, 6, 12, 18, 24, 30, 36, 42
, 48, 54, 7, 14, 21, 28, 35, 42, 49, 56, 63, 8, 16, 24, 32, 40, 48, 56, 64, 72,
9, 18, 27, 36, 45, 54, 63, 72, 81]
```

지금껏 우리는 프로그램의 흐름을 제어하는 if문, while문, for문에 대해서 알아보았다. 아마도 여러분은 while문과 for문을 보면서 2가지가 아주 비슷하다는 느낌을 받았을 것이다. 실제로 for문을 사용한 부분을 while문으로 바꿀 수 있는 경우도 많고, while문을 for문으로 바꾸어서 사용할 수 있는 경우도 많다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#03-3-for>)

[문제1] 1부터 100까지 출력

1부터 100까지의 숫자를 for문을 이용하여 출력하시오.

[문제2] 5의 배수의 총합

for문을 이용하여 1부터 1000까지의 자연수 중 5의 배수에 해당되는 자연수들의 총합을 구하시오.

[문제3] 학급의 평균 점수

for문을 이용하여 A 학급의 평균 점수를 구해 보자.

```
A = [70, 60, 55, 75, 95, 90, 80, 80, 85, 100]
```

[문제4] 혈액형

다음은 학생들의 혈액형(A, B, AB, O)에 대한 데이터이다.

```
['A', 'B', 'A', 'O', 'AB', 'AB', 'O', 'A', 'B', 'O', 'B', 'AB']
```

for 문을 이용하여 각 혈액형 별 학생수의 합계를 구하시오.

[문제5] 리스트 내포1

리스트 중에서 홀수에만 2를 곱하여 저장하는 다음과 같은 코드가 있다.

```
numbers = [1, 2, 3, 4, 5]

result = []
for n in numbers:
    if n % 2 == 1:
        result.append(n*2)
```

위 코드를 리스트 내포(list comprehension)를 이용하여 표현하시오.

[문제6] 리스트 내포2

리스트 내포를 이용하여 다음 문장에서 모음('aeiou')을 제거하시오.

```
Life is too short, you need python
```

04장 프로그램의 입력과 출력은 어떻게 해야 할까?

지금껏 공부한 내용을 바탕으로 함수, 입력과 출력, 파일 처리 방법 등에 대해서 알아보기로 하자.

입출력은 프로그래밍 설계와 관련이 있다. 프로그래머는 프로그램을 만들기 전에 어떤 식으로 동작하게 할 것인지 설계 부터 하게 되는데 그때 가장 중요한 부분이 바로 입출력의 설계이다. 특정 프로그램만이 사용하는 함수를 만들 것인지 아니면 모든 프로그램이 공통으로 사용하는 함수를 만들 것인지, 더 나아가 오픈 API로 공개하여 외부 프로그램들도 사용할 수 있게 만들 것인지 그 모든 것이 입출력과 관련이 있다.

04-1 함수

함수란 무엇인가?

함수를 설명하기 전에 막서기를 생각해 보자. 우리는 막서기에 파일을 넣는다. 그리고 막서기를 이용해서 파일을 갈아 파일 주스를 만든다. 우리가 막서기에 넣는 파일은 “입력”이 되고 파일 주스는 “출력(결과값)”이 된다.

그렇다면 막서기는 무엇인가?



(막서기는 파일을 입력받아 주스를 출력하는 함수와 같다.)

우리가 배우려는 함수가 바로 막서기와 비슷하다. 입력값을 가지고 어떤 일을 수행한 다음에 그 결과물을 내어놓는 것, 이것이 바로 함수가 하는 일이다. 우리는 어려서부터 함수에 대해 공부했지만 함수에 관해 깊이 생각해 본 적은 별로 없다. 예를 들어 $y = 2x + 3$ 도 함수다. 하지만 이를 수학 시간에 배운 직선 그래프로만 알고 있지 x 에 어떤 값을 넣었을 때 어떤 변화에 의해서 y 값이 나오는지 그 과정에 대해서는 별로 관심을 두지 않았을 것이다.

이제 우리는 함수에 대해서 조금 더 생각해 보는 시간을 가져야 한다. 프로그래밍에 있어서 함수는 정말 중요하기 때문이다. 자, 이제 파이썬 함수의 세계로 깊이 들어가 보자.

함수를 사용하는 이유는 무엇일까?

프로그래밍을 하다 보면 똑같은 내용을 반복해서 작성하고 있는 자신을 발견할 때가 종종 있다. 이때가 바로 함수가 필요한 때이다. 즉, 반복되는 부분이 있을 경우 “반복적으로 사용되는 가치 있는 부분”을 한 뭉치로 묶어서 “어떤 입력값을 주었을 때 어떤 결과값을 돌려준다”라는식의 함수로 작성하는 것이 현명하다.

함수를 사용하는 또 다른 이유는 자신이 만든 프로그램을 함수화하면 프로그램의 흐름을 일목요연하게 볼 수 있기 때문이다. 마치 공장에서 원재료가 여러 공정을 거쳐 하나의 상품이되는 것처럼 프로그램에서도 입력한 값이 여러 함수들을 거치면서 원하는 결과값을 내는 것을 볼 수 있다. 이렇게 되면 프로그램의 흐름도 잘 파악할 수 있고 에러가 어디에서 나는지도 금방 알아차릴 수 있다. 함수를 잘 이용하고 함수를 적절하게 만들 줄 아는 사람이 능력 있는 프로그래머이다.

파이썬 함수의 구조

파이썬 함수의 구조는 다음과 같다.

```
def 함수명(매개변수):
    <수행할 문장1>
    <수행할 문장2>
    ...
    ...
```

`def`는 함수를 만들 때 사용하는 예약어이며, 함수명은 함수를 만드는 사람이 임의로 만들 수 있다. 함수명 뒤 괄호 안의 매개변수는 이 함수에 입력으로 전달되는 값을 받는 변수이다. 이렇게 함수를 정의한 다음 `if`, `while`, `for`문 등과 마찬가지로 함수에서 수행할 문장들을 입력한다.

간단하지만 많은 것을 설명해 주는 다음의 예를 보자.

```
def sum(a, b):
    return a + b
```

위 함수는 다음과 같이 풀이된다.

“이 함수의 이름(함수명)은 `sum`이고 입력으로 2개의 값을 받으며 결과값은 2개의 입력값을 더한 값이다.”

여기서 `return`은 함수의 결과값을 돌려주는 명령어이다. 먼저 다음과 같이 `sum` 함수를 만들자.

```
>>> def sum(a, b):
...     return a+b
...
>>>
```

이제 직접 `sum` 함수를 사용해 보자.

```
>>> a = 3
>>> b = 4
>>> c = sum(a, b)
>>> print(c)
7
```

변수 a에 3, b에 4를 대입한 다음 앞서 만든 sum 함수에 a와 b를 입력값으로 넣어 준다. 그리고 변수 c에 sum 함수의 결과값을 대입하면 print(c)로 c의 값을 확인할 수 있다.

매개변수와 인수

매개변수(parameter)와 인수(arguments)는 혼용해서 사용되는 헷갈리는 용어이므로 잘 기억해 두기로 하자. 매개변수는 함수에 입력으로 전달된 값을 받는 변수를 의미하고 인수는 함수를 호출할 때 전달하는 입력값을 의미한다.

```
def sum(a, b): # a, b는 매개변수
    return a+b

print(sum(3, 4)) # 3, 4는 인수
```

[같은 의미를 가진 여러 가지 용어들에 주의하자]

프로그래밍을 공부할 때 어려운 부분 중 하나가 용어의 혼용이라고 할 수 있다. 우리는 공부하면서 원서를 보기도 하고 누군가의 번역본을 보기도 하면서 의미는 같지만 표현이 다른 용어들을 자주 만나게 된다. 한 예로 입력값을 다른 말로 함수의 인수, 입력 인수 등으로 말하기도 하고 결과값을 출력값, 리턴값, 돌려주는 값 등으로 말하기도 한다. 이렇듯 많은 용어들이 여러 가지 다른 말로 표현되지만 의미는 동일한 경우가 많다. 이런 것들을 기억해 놓아야 머리가 덜 아플 것이다.

입력값과 결과값에 따른 함수의 형태

함수는 들어온 입력값을 받아 어떤 처리를 하여 적절한 결과값을 돌려준다.

입력값 —> 함수 —> 리턴값

함수의 형태는 입력값과 결과값의 존재 유무에 따라 4가지 유형으로 나뉜다. 자세히 알아보자.

일반적인 함수

입력값이 있고 결과값이 있는 함수가 일반적인 함수이다. 앞으로 여러분이 프로그래밍을 할 때 만들 함수는 대부분 다음과 비슷한 형태일 것이다.

```
def 함수이름(매개변수):
    <수행할 문장>
    ...
    return 결과값
```

다음은 일반적인 함수의 전형적인 예이다.

```
def sum(a, b):
    result = a + b
    return result
```

sum 함수는 2개의 입력값을 받아서 서로 더한 결과값을 돌려준다.

이 함수를 사용하는 방법은 다음과 같다. 입력값으로 3과 4를 주고 결과값을 돌려받아 보자.

```
>>> a = sum(3, 4)
>>> print(a)
7
```

이처럼 입력값과 결과값이 있는 함수의 사용법을 정리하면 다음과 같다.

결과값을 받을 변수 = 함수명(입력 인수 1, 입력 인수 2, ...)

입력값이 없는 함수

입력값이 없는 함수가 존재할까? 당연히 존재한다. 다음을 보자.

```
>>> def say():
...     return 'Hi'
...
>>>
```

say라는 이름의 함수를 만들었다. 그런데 매개변수 부분을 나타내는 함수명 뒤의 괄호 안이 비어 있다. 이 함수는 어떻게 사용하는 걸까?

다음을 직접 입력해 보자.

```
>>> a = say()
>>> print(a)
Hi
```

위의 함수를 쓰기 위해서는 say()처럼 괄호 안에 아무런 값도 넣지 않아야 한다. 이 함수는 입력값은 없지만 결과값으로 Hi라는 문자열을 돌려준다. a = say()처럼 작성하면 a에 Hi라는 문자열이 대입되는 것이다.

이처럼 입력값이 없고 결과값만 있는 함수는 다음과 같이 사용된다.

결과값을 받을 변수 = 함수명()

결과값이 없는 함수

결과값이 없는 함수 역시 존재한다. 다음의 예를 보자.

```
>>> def sum(a, b):
...     print("%d, %d의 합은 %d입니다." % (a, b, a+b))
...
>>>
```

결과값이 없는 함수는 호출해도 돌려주는 값이 없기 때문에 다음과 같이 사용한다.

```
>>> sum(3, 4)
3, 4의 합은 7입니다.
```

즉, 결과값이 없는 함수는 다음과 같이 사용한다.

함수명(입력 인수1, 입력 인수2, ...)

결과값이 진짜 없는지 확인하기 위해 다음의 예를 직접 입력해 보자.

```
>>> a = sum(3, 4)
3, 4의 합은 7입니다.
```

아마도 여러분은 “3, 4의 합은 7입니다.”라는 문장을 출력해 주었는데 왜 결과값이 없다는 것인지 의아하게 생각할 것이다. 이 부분이 초보자들이 혼란스러워하는 부분이기도 한데 print문은 함수의 구성 요소 중 하나인 에 해당하는 부분일 뿐이다. 결과값은 당연히 없다. 결과값은 오직 return 명령어로만 돌려받을 수 있다.

이를 확인해 보자. 돌려받을 값을 a라는 변수에 대입하여 출력해 보면 결과값이 있는지 없는지 알 수 있다.

```
>>> a = sum(3, 4)
>>> print(a)
None
```

a의 값은 None이다. None이란 거짓을 나타내는 자료형이라고 언급한 적이 있다. sum 함수처럼 결과값이 없을 때 a = sum(3, 4)처럼 쓰게 되면 함수 sum은 리턴값으로 a 변수에 None을 돌려준다. 이것을 가지고 결과값이 있다고 생각하면 곤란하다.

입력값도 결과값도 없는 함수

입력값도 결과값도 없는 함수 역시 존재한다. 다음의 예를 보자.

```
>>> def say():
...     print('Hi')
...
>>>
```

입력 인수를 받는 매개변수도 없고 return문도 없으니 입력값도 결과값도 없는 함수이다. 이 함수를 사용하는 방법은 단 한 가지이다.

```
>>> say()
Hi
```

즉, 입력값도 결과값도 없는 함수는 다음과 같이 사용한다.

함수명()

매개변수 지정하여 호출하기

함수를 호출 할 때 매개변수를 지정하여 호출할 수도 있다. 다음의 예를 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> def sum(a, b):
...     return a+b
...
```

앞서 알아보았던 `sum`함수이다. 이 함수를 우리는 다음과 같이 매개변수를 지정하여 사용할 수 있다.

```
>>> sum(a=3, b=7) # a에 3, b에 7을 전달
10
```

매개변수를 지정하면 다음과 같이 순서에 상관없이 사용할 수 있다는 장점이 있다.

```
>>> sum(b=5, a=3) # b에 5, a에 3을 전달
8
```

입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?

입력값이 여러 개일 때 그 입력값들을 모두 더해 주는 함수를 생각해 보자. 하지만 몇 개가 입력될지 모를 때는 어떻게 해야 할까? 아마도 난감할 것이다. 파이썬은 이런 문제를 해결하기 위해 다음과 같은 방법을 제공한다.

```
def 함수이름(*매개변수):
    <수행할 문장>
    ...
```

일반적으로 볼 수 있는 함수 형태에서 괄호 안의 매개변수 부분이 *매개변수로 바뀌었다.

여러 개의 입력값을 받는 함수 만들기

다음의 예를 통해 여러 개의 입력값을 모두 더하는 함수를 직접 만들어 보자. 예를 들어 `sum_many(1, 2)`이면 3을, `sum_many(1,2,3)`이면 6을, `sum_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`이면 55를 돌려주는 함수를 만들어 보자.

```
>>> def sum_many(*args):
...     sum = 0
...     for i in args:
...         sum = sum + i
...     return sum
...
>>>
```

위에서 만든 sum_many라는 함수는 입력값이 몇 개이든 상관이 없다. *args처럼 매개변수명 앞에 *을 붙이면 입력값들을 전부 모아서 튜플로 만들어 주기 때문이다. 만약 sum_many(1, 2, 3)처럼 이 함수를 쓰면 args는 (1, 2, 3)이 되고, sum_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)처럼 쓰면 args 는 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)이 된다. 여기서 *args라는 것은 임의로 정한 변수명이다. *pey, *python처럼 아무 이름이나 써도 된다.

※ args는 입력 인수를 뜻하는 영어 단어인 arguments의 약자이며 관례적으로 자주 사용한다.

실제로 이 함수를 직접 입력해 보자.

```
>>> result = sum_many(1,2,3)
>>> print(result)
6
>>> result = sum_many(1,2,3,4,5,6,7,8,9,10)
>>> print(result)
55
```

sum_many(1,2,3)으로 함수를 호출하면 6을 리턴하고, sum_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)을 대입하면 55를 리턴한다.

`def sum_many(*args)`처럼 함수의 인수로 `*args(*매개변수)`만 사용할 수 있는 것은 아니다. 다음의 예를 보자.

```
>>> def sum_mul(choice, *args):
...     if choice == "sum":
...         result = 0
...         for i in args:
...             result = result + i
...     elif choice == "mul":
...         result = 1
...         for i in args:
...             result = result * i
...     return result
...
>>>
```

위의 예는 매개변수로 choice와 *args를 사용했다. 따라서 다음과 같이 쓸 수 있다.

즉, sum_mul('sum', 1,2,3,4) 또는 sum_mul('mul', 1,2,3,4,5)처럼 choice 부분에 'sum'이나 'mul'이라는 문자열을 입력값으로 준다. 그리고 그 뒤에는 개수에 상관없이 여러 개의 숫자를 입력값으로 준다.

실제로 한 번 테스트해 보자.

```
>>> result = sum_mul('sum', 1,2,3,4,5)
>>> print(result)
15
>>> result = sum_mul('mul', 1,2,3,4,5)
>>> print(result)
120
```

첫 번째 입력 인수에 'sum'이 입력된 경우 *args에 입력되는 모든 값을 더해서 15를 돌려주고, 'mul'이 입력된 경우 *args에 입력되는 모든 값을 곱해서 120을 돌려준다.

키워드 파라미터 kwargs

이번에는 키워드 파라미터인 ****kwargs**에 대해서 알아보자. kwargs는 keyword arguments의 약어이다. ****kwargs**는 ***args**와는 달리 별표시(*)가 두 개 사용된다. 역시 이것도 예제로 알아보도록 하자.

먼저 다음과 같은 함수를 작성 해 보자.

```
>>> def func(**kwargs):
...     print(kwargs)
...
```

그리고 이 함수를 다음과 같이 사용해 보자.

```
>>> func(a=1)
{'a': 1}
>>> func(name='foo', age=3)
{'age': 3, 'name': 'foo'}
```

func 함수의 인수로 key=value 형태를 주었을 때 입력 값 전체가 kwargs라는 딕셔너리에 저장된다는 것을 알 수 있다.

즉, `**kwargs`는 모든 key=value 형태의 입력인수가 저장되는 딕셔너리 변수이다.

이번에는 다음과 같은 형태의 호출에 대해서 생각해 보자.

```
>>> func(1, 2, 3, name='foo', age=3)
```

위 예처럼 이렇게 입력인수의 형태가 다양한 경우 입력인수의 갯수에 상관없이 입력을 받고 싶다면 어떻게 해야 할까?

이런경우 다음의 함수를 사용하면 입력인수를 모두 처리할 수 있게 된다.

```
>>> def func(*args, **kwargs):
...     print(args)
...     print(kwargs)
...
>>> func(1, 2, 3, name='foo', age=3)
(1, 2, 3)
{'age': 3, 'name': 'foo'}
```

1, 2, 3 과 같은 일반적인 입력은 args튜플에 저장되고 name='foo' 와 같은 형태의 입력은 kwargs 딕셔너리에 저장되는 것을 확인할 수 있다.

만약 다음과 같이 호출한다면 어떻게 될까?

```
>>> func(1, 2, 3, name='foo', age=3, 4, 5)
```

일반적인 입력인수가 키워드형태의 입력 뒤에 존재하는 경우이다. 호출하면 다음과 같은 오류를 만나게 된다.

```
>>> func(1, 2, 3, name='foo', age=3, 4, 5)
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

키워드 형태의 인수뒤에 키워드 형태가 아닌 인수는 올 수 없음을 알 수 있다. 따라서 다음과 같은 형태의 호출로 변경해야 할 것이다.

```
>>> func(1, 2, 3, 4, 5, name='foo', age=3)
(1, 2, 3, 4, 5)
{'age': 3, 'name': 'foo'}
```

함수의 결과값은 언제나 하나이다

먼저 다음의 함수를 만들어 보자.

```
>>> def sum_and_mul(a,b):
...     return a+b, a*b
```

※ `sum_and_mul` 함수는 2개의 입력 인수를 받아 더한 값과 곱한 값을 돌려주는 함수이다.

이 함수를 다음과 같이 호출하면 어떻게 될까?

```
>>> result = sum_and_mul(3,4)
```

결과값은 `a+b`와 `a*b` 2개인데 결과값을 받아들이는 변수는 `result` 하나만 쓰였으니 오류가 발생하지 않을까? 당연한 의문이다. 하지만 오류는 발생하지 않는다. 그 이유는 함수의 결과값은 2개가 아니라 언제나 1개라는 데 있다. `sum_and_mul` 함수의 결과값 `a+b`와 `a*b`는 튜플값 하나인 `(a+b, a*b)`로 돌려준다.

따라서 `result` 변수는 다음과 같은 값을 갖게 된다.

```
result = (7, 12)
```

즉, 결과값으로 (7, 12)라는 튜플 값을 갖게 되는 것이다.

만약 이 하나의 튜플값을 2개의 결과값처럼 받고 싶다면 다음과 같이 함수를 호출하면 된다.

```
>>> sum, mul = sum_and_mul(3, 4)
```

이렇게 호출하면 sum, mul = (7, 12)가 되어 sum은 7이 되고 mul은 12가 된다.

또 다음과 같은 의문이 생길 수도 있다.

```
>>> def sum_and_mul(a,b):
...     return a+b
...     return a*b
...
>>>
```

위와 같이 return문을 2번 사용하면 2개의 결과값을 돌려주지 않을까? 하지만 파이썬에서 위와 같은 함수는 참 어리석은 함수이다.

그 이유는 sum_and_mul 함수를 호출해 보면 알 수 있다.

```
>>> result = sum_and_mul(2, 3)
>>> print(result)
5
```

※ sum_and_mul(2, 3)의 결과값은 5 하나뿐이다. 두 번째 return문인 return a*b는 실행되지 않았다는 뜻이다.

이 예에서 볼 수 있듯이 두 번째 return문인 return a*b는 실행되지 않았다. 따라서 이 함수는 다음과 완전히 동일하다.

```
>>> def sum_and_mul(a,b):
...     return a+b
...
>>>
```

즉, 함수는 return문을 만나는 순간 결과값을 돌려준 다음 함수를 빠져나가게 된다.

[return의 또 다른 쓰임새]

어떤 특별한 상황이 되면 함수를 빠져나가고자 할 때는 return을 단독으로 써서 함수를 즉시 빠져나갈 수 있다. 다음 예를 보자.

```
>>> def say_nick(nick):
...     if nick == "바보":
...         return
...     print("나의 별명은 %s 입니다." % nick)
...
>>>
```

위의 함수는 “별명”을 입력으로 전달받아 출력하는 함수이다. 이 함수 역시 리턴값은 없다(문자열을 출력한다는 것과 리턴값이 있다는 것은 전혀 다른 말이다. 혼동하지 말자. 함수의 리턴값은 오로지 return문에 의해서만 생성된다).

만약에 입력값으로 ‘바보’라는 값이 들어오면 문자열을 출력하지 않고 함수를 즉시 빠져나간다.

```
>>> say_nick('야호')
나의 별명은 야호입니다.
>>> say_nick('바보')
>>>
```

이처럼 return으로 함수를 빠져나가는 방법은 실제 프로그래밍에서 자주 사용된다.

매개변수에 초기값 미리 설정하기

이번에는 조금 다른 형태로 함수의 인수를 전달하는 방법에 대해서 알아보자. 매개변수에 초기값을 미리 설정해 주는 경우이다.

※ 앞으로 나올 프로그램 소스에서 >>> 표시가 없으면 에디터로 작성하기를 권장한다는 뜻이다.

```
def say_myself(name, old, man=True):
    print("나의 이름은 %s 입니다." % name)
    print("나이는 %d살입니다." % old)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

※ `say_myself` 함수는 3개의 입력 인수를 받아서 마지막 인수인 `man`이 `True`이면 “남자입니다.”, `False`이면 “여자입니다.”를 출력한다.

위의 함수를 보면 매개변수가 `name`, `old`, `man=True` 이렇게 3개다. 그런데 낯선 것이 나왔다. `man=True`처럼 매개변수에 미리 값을 넣어 준 것이다. 이것이 바로 함수의 매개변수 초기값을 설정하는 방법이다. 함수의 매개변수에 전달되는 값이 항상 변하는 것이 아닐 경우에는 이렇게 함수의 초기값을 미리 설정해 두면 유용하다.

`say_myself` 함수는 다음처럼 사용할 수 있다.

```
say_myself("박응용", 27)
say_myself("박응용", 27, True)
```

입력값으로 “박응용”, 27처럼 2개를 주면 `name`에는 “박응용”이 `old`에는 27이 전달된다. 그리고 `man`이라는 변수에는 입력값을 주지 않았지만 초기값인 `True` 값을 갖게 된다.

따라서 위의 예에서 함수를 사용한 2가지 방법은 모두 동일한 결과를 출력한다.

```
나의 이름은 박응용입니다.
나이는 27살입니다.
남자입니다.
```

이제 초기값이 설정된 부분을 `False`로 바꿔 보자.

```
say_myself("박응선", 27, False)
```

`man` 변수에 `False` 값이 전달되어 다음과 같은 결과가 출력된다.

나의 이름은 박용선입니다.
나이는 27살입니다.
여자입니다.

함수 매개변수에 초기값을 설정할 때 주의할 사항

함수의 매개변수에 초기값을 설정할 때 주의할 것이 하나 있다. 만약 위에서 본 say_myself 함수를 다음과 같이 만들면 어떻게 될까?

```
def say_myself(name, man=True, old):
    print("나의 이름은 %s 입니다." % name)
    print("나이는 %d살입니다." % old)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

이전 함수와 바뀐 부분은 초기값을 설정한 매개변수의 위치이다. 결론을 미리 말하면 이것은 함수를 실행할 때 오류가 발생한다.

얼핏 생각하기에 위의 함수를 호출하려면 다음과 같이 하면 될 것 같다.

```
say_myself("박용용", 27)
```

위와 같이 함수를 호출한다면 name 변수에는 “박용용”이 들어갈 것이다. 하지만 파이썬 인터프리터는 27을 man 변수와 old 변수 중 어느 곳에 전달해야 할지 알 수 없게 된다.

오류 메시지를 보면 다음과 같다.

```
SyntaxError: non-default argument follows default argument
```

위의 오류 메시지는 초기값을 설정해 놓은 매개변수 뒤에 초기값을 설정해 놓지 않은 매개변수는 사용할 수 없다는 말이다. 즉, 매개변수로 (name, old, man=True)는 되지만 (name, man=True, old)는 안 된다는 것이다. 초기화시키고 싶은 매개변수들을 항상 뒤쪽에 위치시키는 것을 잊지 말자.

함수 안에서 선언된 변수의 효력 범위

함수 안에서 사용할 변수의 이름을 함수 밖에서도 동일하게 사용한다면 어떻게 될까? 이런 궁금증이 생겼던 독자라면 이번에 확실하게 답을 찾을 수 있을 것이다.

아래의 예를 보자.

```
# vartest.py
a = 1
def vartest(a):
    a = a +1

vartest(a)
print(a)
```

먼저 a라는 변수를 생성하고 1을 대입한다. 다음 입력으로 들어온 값에 1을 더해 주고 결과값은 돌려주지 않는 vartest 함수를 선언한다. 그리고 vartest 함수에 입력값으로 a를 주었다. 마지막으로 a의 값을 출력하는 print(a)를 입력한다. 과연 결과값은 무엇이 나올까?

당연히 vartest 함수에서 매개변수 a의 값에 1을 더했으니까 2가 출력되어야 할 것 같지만 프로그램 소스를 작성해서 실행시켜 보면 결과값은 1이 나온다. 그 이유는 함수 안에서 새로 만들어진 매개변수는 함수 안에서만 사용되는 “함수만의 변수”이기 때문이다. 즉, def vartest(a)에서 입력값을 전달받는 매개변수 a는 함수 안에서만 사용되는 변수이지 함수 밖의 변수 a가 아니라는 뜻이다.

따라서 vartest 함수는 다음처럼 변수 이름을 hello로 한 vartest 함수와 완전히 동일하다.

```
def vartest(hello):
    hello = hello + 1
```

즉, 함수 안에서 사용되는 매개변수는 함수 밖의 변수 이름들과는 전혀 상관이 없다는 말이다.

다음의 예를 보면 더욱 분명하게 이해할 수 있을 것이다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
# vartest_error.py
def vartest(a):
    a = a + 1

vartest(3)
print(a)
```

위의 프로그램 소스를 에디터로 작성해서 실행시키면 어떻게 될까? 오류가 발생할 것이라고 생각한 독자는 모든 것을 이해한 독자이다. vartest(3)을 수행하면 vartest라는 함수 내에서 a는 4가 되지만 함수를 호출하고 난 뒤에 print(a)라는 문장은 에러가 발생하게 된다. 그 이유는 print(a)에서 사용된 a라는 변수는 어디에서도 찾을 수가 없기 때문이다. 다시 말하지만 함수 안에서 선언된 매개변수는 함수 안에서만 사용될 뿐 함수 밖에서는 사용되지 않는다. 이것을 이해하는 것은 매우 중요하다.

그렇다면 vartest라는 함수를 이용해서 함수 밖의 변수 a를 1만큼 증가시킬 수 있는 방법은 없을까? 이 질문에는 2가지 해결 방법이 있다.

함수 안에서 함수 밖의 변수를 변경하는 방법

1. return 이용하기

```
# vartest_return.py
a = 1
def vartest(a):
    a = a +1
    return a

a = vartest(a)
print(a)
```

첫 번째 방법은 return을 이용하는 방법이다. vartest 함수는 입력으로 들어온 값에 1을 더한값을 돌려준다. 따라서 a = vartest(a)라고 대입하면 a가 vartest 함수의 결과값으로 바뀐다. 여기서도 물론 vartest 함수 안의 a 매개변수는 함수 밖의 a와는 다른 것이다.

2. global 명령어 이용하기

```
# vartest_global.py
a = 1
def vartest():
    global a
    a = a+1

vartest()
print(a)
```

두 번째 방법은 `global`이라는 명령어를 이용하는 방법이다. 위의 예에서 볼 수 있듯이 `vartest` 함수 안의 `global a`라는 문장은 함수 안에서 함수 밖의 `a` 변수를 직접 사용하겠다는 뜻이다. 하지만 프로그래밍을 할 때 `global` 명령어는 사용하지 않는 것이 좋다. 왜냐하면 함수는 독립적으로 존재하는 것이 좋기 때문이다. 외부 변수에 종속적인 함수는 그다지 좋은 함수가 아니다. 그러므로 가급적 `global` 명령어를 사용하는 이 방법은 피하고 첫 번째 방법을 사용하기를 권한다.

lambda

`lambda`는 함수를 생성할 때 사용하는 예약어로, `def`와 동일한 역할을 한다. 보통 함수를 한줄로 간결하게 만들 때 사용한다. 우리말로는 “람다”라고 읽고 `def`를 사용해야 할 정도로 복잡하지 않거나 `def`를 사용할 수 없는 곳에 주로 쓰인다. 사용법은 다음과 같다.

`lambda` 매개변수1, 매개변수2, ... : 매개변수를 이용한 표현식

한번 직접 만들어 보자.

```
>>> sum = lambda a, b: a+b
>>> sum(3,4)
7
```

`sum`은 두개의 인수를 받아 서로 더한 값을 리턴하는 `lambda` 함수이다. 위의 예제는 `def`를 사용한 아래 함수와 하는 일이 완전히 동일하다.

```
>>> def sum(a, b):
...     return a+b
...
>>>
```

그렇다면 def가 있는데 왜 lambda라는 것이 나오게 되었을까? 이유는 간단하다. lambda는 def보다 간결하게 사용할 수 있기 때문이다. 또한 lambda는 def를 사용할 수 없는 곳에도 사용할 수 있다. 다음 예제에서 리스트 내에 lambda가 들어간 경우를 살펴보자.

```
>>> myList = [lambda a,b:a+b, lambda a,b:a*b]
>>> myList
[ 0x811eb2c>,  0x811eb64>]
```

즉, 리스트 각각의 요소에 lambda 함수를 만들어 바로 사용할 수 있다. 첫 번째 요소 myList[0]은 2개의 입력값을 받아 두 값의 합을 돌려주는 lambda 함수이다.

```
>>> myList[0]
 0x811eb2c>
>>> myList[0](3,4)
7
```

두 번째 요소 myList[1]은 2개의 입력값을 받아 두 값의 곱을 돌려주는 lambda 함수이다.

```
>>> myList[1](3,4)
12
```

파이썬에 익숙해질수록 lambda 함수가 굉장히 편리하다는 사실을 알게 될 것이다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#04-1>)

[문제1] 홀수 짝수 판별

주어진 자연수가 홀수인지 짝수인지 판별해 주는 함수(is_odd)를 작성하시오.

[문제2] 평균값 계산

입력으로 들어오는 모든 수의 평균값을 계산해 주는 함수를 작성해 보자. (단, 입력으로 들어오는 수의 갯수는 정해져 있지 않다.)

[문제3] 구구단 출력

입력을 자연수 n(2부터 9까지의 자연수)으로 받았을 때, n에 해당되는 구구단을 출력하는 함수를 작성해 보자.

[문제4] 피보나치

입력을 정수 n으로 받았을 때, n 이하까지의 피보나치 수열을 출력하는 함수를 작성해 보자.

피보나치 수열은 다음과 같은 순서로 결과값을 리턴한다.

1. fib(0) → 0 리턴
2. fib(1) → 1 리턴
3. fib(2) → fib(0) + fib(1) → 0 + 1 → 1 리턴
4. fib(3) → fib(1) + fib(2) → 1 + 1 → 2 리턴
5. fib(4) → fib(2) + fib(3) → 1 + 2 → 3 리턴

[문제5] 5보다 큰 수만

다음은 숫자들로 이루어진 리스트를 입력으로 받아 5보다 큰 수만 필터링하여 리턴해 주는 함수이다.

```
>>> def myfunc(numbers):
...     result = []
...     for number in numbers:
...         if number > 5:
...             result.append(number)
...     return result
...
>>> myfunc([2,3,4,5,6,7,8])
[6, 7, 8]
```

위 함수를 lambda 함수로 변경해 보시오.

04-2 사용자 입력과 출력

우리들이 사용하는 대부분의 완성된 프로그램은 사용자의 입력에 따라 그에 맞는 출력을 내보낸다. 대표적인 예로 게시판에 글을 작성한 후 “확인” 버튼을 눌러야만(입력) 우리가 작성한 글이 게시판에 올라가는(출력) 것을 들 수 있다.

사용자 입력 → 처리(프로그램, 함수 등) → 출력

우리는 이미 함수 부분에서 입력과 출력이 어떤 의미를 가지는지 알아보았다. 지금부터는 좀 더 다양하게 사용자의 입력을 받는 방법과 출력하는 방법을 알아보자.

사용자 입력

사용자가 입력한 값을 어떤 변수에 대입하고 싶을 때는 어떻게 해야 할까?

input의 사용

```
>>> a = input()
Life is too short, you need python
>>> a
'Life is too short, you need python'
>>>
```

input은 입력되는 모든 것을 문자열로 취급한다.

※ 파이썬 2.7 버전의 경우 위 예제의 input 대신 raw_input을 사용해야 한다.

프롬프트를 띠워서 사용자 입력 받기

사용자에게 입력을 받을 때 “숫자를 입력하세요” 라든지 “이름을 입력하세요”라는 안내 문구 또는 질문이 나오도록 하고 싶을 때가 있다. 그럴 때는 input()의 괄호 안에 질문을 입력하여 프롬프트를 띠워주면 된다.

`input("질문 내용")`

다음의 예를 직접 입력해 보자.

```
>>> number = input("숫자를 입력하세요: ")
숫자를 입력하세요:
```

위와 같은 질문을 볼 수 있을 것이다.

숫자를 입력하라는 프롬프트에 3을 입력하면 변수 number에 3이 대입된다. print(number)로 출력해서 제대로 입력되었는지 확인해 보자.

```
>>> number = input("숫자를 입력하세요: ")
숫자를 입력하세요: 3
>>> print(number)
3
>>>
```

print 자세히 알기

지금껏 print문이 수행해온 일은 우리가 입력한 자료형을 출력하는 것이었다. 그간 알아보았던 print의 사용 예는 다음과 같다.

```
>>> a = 123
>>> print(a)
123
>>> a = "Python"
>>> print(a)
Python
>>> a = [1, 2, 3]
>>> print(a)
[1, 2, 3]
```

이제 print문으로 할 수 있는 일에 대해서 조금 더 자세하게 알아보기로 하자.

큰따옴표(“)로 둘러싸인 문자열은 + 연산과 동일하다

```
>>> print("life" "is" "too short") # ①
lifeistoo short
>>> print("life"+"is"+"too short") # ②
lifeistoo short
```

위의 예에서 ①과 ②는 완전히 동일한 결과값을 출력한다. 즉, 따옴표로 둘러싸인 문자열을연속해서 쓰면 + 연산을 한 것과 같다.

문자열 띄어쓰기는 콤마로 한다

```
>>> print("life", "is", "too short")
life is too short
```

콤마(,)를 이용하면 문자열 간에 띄어쓰기를 할 수 있다.

한 줄에 결과값 출력하기

03-3절에서 for문을 배울 때 만들었던 구구단 프로그램에서 보았듯이 한 줄에 결과값을 계속 이어서 출력하려면 입력 인수 end를 이용해 끝 문자를 지정해야 한다.

```
>>> for i in range(10):
...     print(i, end=' ')
...
0 1 2 3 4 5 6 7 8 9
```

※ 파이썬 2.7 버전의 경우 이 예제의 `print(i, end=' ')` 대신 `print i,`를 사용해야 한다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#04-2>)

[문제1] 두 수의 합은?

다음은 두 개의 숫자를 입력받아 더하여 리턴해 주는 프로그램이다.

```

input1 = input("첫번째 숫자를 입력하세요:")
input2 = input("두번째 숫자를 입력하세요:")

total = input1 + input2
print("두 수의 합은 %s 입니다" % total)

```

이 프로그램을 실행 해 보자.

```

첫번째 숫자를 입력하세요:3
두번째 숫자를 입력하세요:6
두 수의 합은 36 입니다

```

3과 6을 입력했을 때 9가 아닌 36이라는 결과가 리턴되었다. 이 프로그램의 오류를 수정하시오.

[문제2] 숫자의 총합

사용자로부터 다음과 같은 숫자들의 입력을 받아 입력받은 숫자들의 총합을 구하는 프로그램을 작성하시오. (단, 숫자들은 콤마로 구분하여 입력한다.)

```
65,45,2,3,45,8
```

[문제3] 문자열 출력

다음 중 출력결과가 다른 것 한개를 고르시오.

1. print("you" "need" "python")
2. print("you"+"need"+"python")
3. print("you", "need", "python")
4. print("".join(["you", "need", "python"]))

[문제4] 한줄 구구단

사용자로부터 2~9 까지의 숫자중 하나를 입력받아 해당 숫자의 구구단을 한줄로 출력하는 프로그램을 작성하시오.

실행 예

```

구구단을 출력할 숫자를 입력하세요(2~9): 2
2 4 6 8 10 12 14 16 18

```

04-3 파일 읽고 쓰기

우리는 이 책에서 이제까지 “입력”을 받을 때는 사용자가 직접 입력하는 방식을 사용했고, “출력” 할 때는 모니터 화면에 결과값을 출력하는 방식으로 프로그래밍해 왔다. 하지만 입출력 방법이 꼭 이것만 있는 것은 아니다. 이번에는 파일을 통한 입출력 방법에 대해서 알아보자. 먼저 파일을 새로 만든 다음 프로그램에 의해서 만들어진 결과값을 새 파일에 한번 적어 보고, 또 파일에 적은 내용을 읽어 보는 프로그램을 만드는 것으로 시작해 보자.

파일 생성하기

다음 소스 코드를 에디터로 작성해서 저장한 후 실행해 보자. 프로그램을 실행한 디렉터리에 새로운 파일이 하나 생성된 것을 확인할 수 있을 것이다

```
f = open("새파일.txt", 'w')
f.close()
```

파일을 생성하기 위해 우리는 `open`이라는 파이썬 내장 함수를 사용했다. `open` 함수는 다음과 같이 “파일 이름”과 “파일 열기 모드”를 입력값으로 받고 결과값으로 파일 객체를 돌려준다.

파일 객체 = `open(파일 이름, 파일 열기 모드)`

파일 열기 모드에는 다음과 같은 것들이 있다.

파일열기모드 설명

r	읽기모드 - 파일을 읽기만 할 때 사용
w	쓰기모드 - 파일에 내용을 쓸 때 사용
a	추가모드 - 파일의 마지막에 새로운 내용을 추가 시킬 때 사용

파일을 쓰기 모드로 열게 되면 해당 파일이 이미 존재할 경우 원래 있던 내용이 모두 사라지고, 해당 파일이 존재하지 않으면 새로운 파일이 생성된다. 위의 예에서는 디렉터리에 파일이 없는 상태에서 새파일.txt를 쓰기 모드인 ‘w’로 열었기 때문에 새파일.txt라는 이름의 새로운 파일이 현재 디렉터리에 생성되는 것이다.

만약 새파일.txt라는 파일을 C:/doit이라는 디렉터리에 생성하고 싶다면 다음과 같이 작성해야 한다.

```
f = open("C:/doit/새파일.txt", 'w')
f.close()
```

위의 예에서 `f.close()`는 열려 있는 파일 객체를 닫아 주는 역할을 한다. 사실 이 문장은 생략해도 된다. 프로그램을 종료할 때 파이썬 프로그램이 열려 있는 파일의 객체를 자동으로 닫아주기 때문이다. 하지만 `close()`를 사용해서 열려 있는 파일을 직접 닫아 주는 것이 좋다. 쓰기모드로 열었던 파일을 닫지 않고 다시 사용하려고 하면 오류가 발생하기 때문이다.

파일을 쓰기 모드로 열어 출력값 적기

위의 예에서는 파일을 쓰기 모드로 열기만 했지 정작 아무것도 쓰지는 않았다. 이번에는 에디터를 열고 프로그램의 출력값을 파일에 직접 써 보자.

```
# writedata.py
f = open("C:/doit/새파일.txt", 'w')
for i in range(1, 11):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)
f.close()
```

위의 프로그램을 다음 프로그램과 비교해 보자.

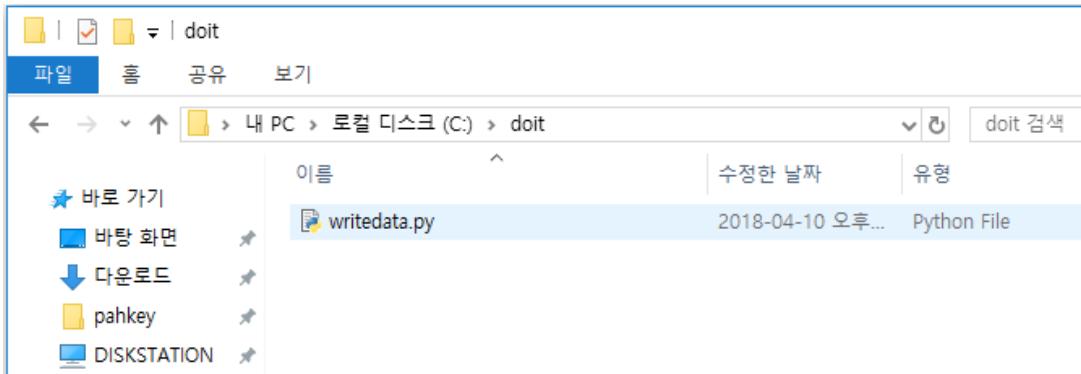
```
for i in range(1, 11):
    data = "%d번째 줄입니다.\n" % i
    print(data)
```

두 프로그램의 다른 점은 `data`를 출력하는 방법이다. 두 번째 방법은 우리가 계속 사용해 왔던 모니터 화면에 출력하는 방법이고, 첫 번째 방법은 모니터 화면 대신 파일에 결과값을 적는 방법이다. 두 방법의 차이점은 `print` 대신 파일 객체 `f`의 `write` 함수를 이용한 것 말고는 없으니 금방 눈에 들어올 것이다.

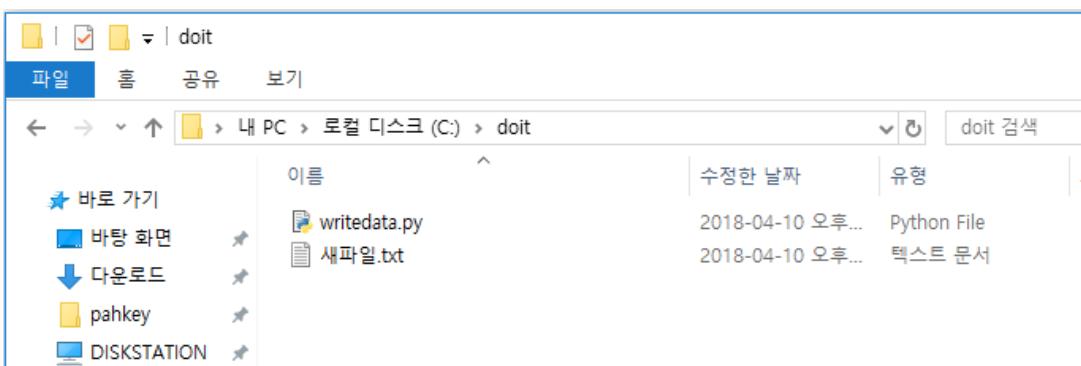
이제 첫 번째 예제를 실행해 보자.

```
C:\Users> cd C:\doit
C:\doit>python writedata.py
C:\doit>
```

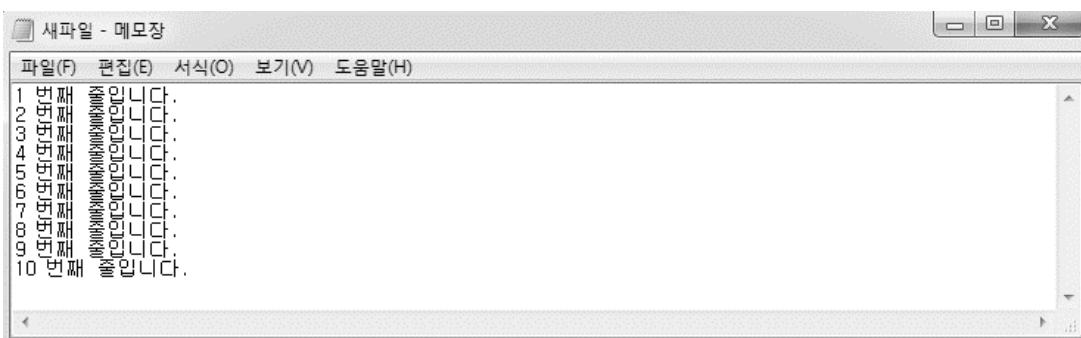
이 프로그램을 실행시킨 C:/doit 디렉터리를 살펴보면 새파일.txt라는 파일이 생성되었음을 확인할 수 있다.



위처럼 디렉터리에 파일이 없다가 다음과 같이 생성된다.



새파일.txt 파일에는 어떤 내용이 담겨 있는지 확인해 보자.



모니터 화면에 바로 출력될 내용이 고스란히 파일에 들어가 있는 것을 볼 수 있다.

프로그램의 외부에 저장된 파일을 읽는 여러 가지 방법

파이썬에는 외부 파일을 읽어 들여 프로그램에서 사용할 수 있는 여러 가지 방법이 있다. 이번에는 그 방법들을 자세히 알아보자.

readline() 함수 이용하기

첫 번째 방법은 readline() 함수를 이용하는 방법이다. 다음의 예를 보자.

```
# readline.py
f = open("C:/doit/새파일.txt", 'r')
line = f.readline()
print(line)
f.close()
```

위 예는 f.open("새파일.txt", 'r')로 파일을 읽기 모드로 연 후 readline()을 이용해서 파일의 첫 번째 줄을 읽어 출력하는 경우이다. 이전에 만들었던 새파일.txt를 수정하거나 지우지 않았다면 위 프로그램을 실행시켰을 때 새파일.txt의 가장 첫 번째 줄이 화면에 출력될 것이다.

1번째 줄입니다.

만약 모든 라인을 읽어서 화면에 출력하고 싶다면 다음과 같이 작성하면 된다.

```
# readline_all.py
f = open("C:/doit/새파일.txt", 'r')
while True:
    line = f.readline()
    if not line: break
    print(line)
f.close()
```

즉, while True:라는 무한 루프 안에서 f.readline()을 이용해 파일을 계속해서 한 줄씩 읽어들이도록 한다. 만약 더 이상 읽을 라인이 없으면 break를 수행한다(readline()은 더 이상 읽을 라인이 없을 경우 None을 출력한다).

앞의 프로그램을 다음 프로그램과 비교해 보자.

```
while 1:
    data = input()
    if not data: break
    print(data)
```

위의 예는 사용자의 입력을 받아서 그 내용을 출력하는 경우이다. 파일을 읽어서 출력하는 예제와 비교해 보자. 입력을 받는 방식만 다르다는 것을 금방 알 수 있을 것이다. 두 번째 예는 키보드를 이용한 입력 방법이고, 첫 번째 예는 파일을 이용한 입력 방법이다.

readlines() 함수 이용하기

두 번째 방법은 `readlines()` 함수를 이용하는 방법이다. 다음의 예를 보자.

```
f = open("C:/doit/새파일.txt", 'r')
lines = f.readlines()
for line in lines:
    print(line)
f.close()
```

`readlines()` 함수는 파일의 모든 라인을 읽어서 각각의 줄을 요소로 갖는 리스트로 리턴한다. 따라서 위의 예에서 `lines`는 `["1 번째 줄입니다.\n", "2 번째 줄입니다.\n", ..., "10 번째 줄입니다.\n"]`라는 리스트가 된다. `f.readlines()`에서 `f.readline()`과는 달리 `s`가 하나 더 붙어 있음에 유의하자.

read() 함수 이용하기

세 번째 방법은 `read()` 함수를 이용하는 방법이다. 다음의 예를 보자.

```
f = open("C:/doit/새파일.txt", 'r')
data = f.read()
print(data)
f.close()
```

`f.read()`는 파일의 내용 전체를 문자열로 리턴한다. 따라서 위 예의 `data`는 파일의 전체 내용이다.

파일에 새로운 내용 추가하기

쓰기 모드('w')로 파일을 열 때 이미 존재하는 파일을 열 경우 그 파일의 내용이 모두 사라지게 된다고 했다. 하지만 원래 있던 값을 유지하면서 단지 새로운 값만 추가해야 할 경우도 있다. 이런 경우에는 파일을 추가 모드('a')로 열면 된다. 에디터를 켜고 다음 소스 코드를 작성해 보자.

```
# adddata.py
f = open("C:/doit/새파일.txt", 'a')
for i in range(11, 20):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)
f.close()
```

위 예는 새파일.txt라는 파일을 추가 모드('a')로 열고 write를 이용해서 결과값을 기존 파일에 추가해 적는 예이다. 여기서 추가 모드로 파일을 열었기 때문에 새파일.txt라는 파일이 원래 가지고 있던 내용 바로 다음부터 결과값을 적기 시작한다.

새파일.txt 파일을 확인해 보면 원래 있던 파일 뒷부분에 새로운 내용이 추가되었음을 볼 수 있다.

```
C:\Users> cd C:\doit
C:\doit>python adddata.py
C:\doit>
```



with문과 함께 사용하기

지금까지 살펴본 예제를 보면 항상 다음과 같은 방식으로 파일을 열고 닫아 왔다.

```
f = open("foo.txt", 'w')
f.write("Life is too short, you need python")
f.close()
```

파일을 열면 위와 같이 항상 close해 주는 것이 좋다. 하지만 이렇게 파일을 열고 닫는 것을 자동으로 처리할 수 있다면 편리하지 않을까?

파이썬의 with문이 바로 이런 역할을 해준다. 다음의 예는 with문을 이용해서 위 예제를 다시 작성한 모습이다.

```
with open("foo.txt", "w") as f:
    f.write("Life is too short, you need python")
```

위와 같이 with문을 이용하면 with 블록을 벗어나는 순간 열린 파일 객체 f가 자동으로 close되어 편리하다.

※ with구문은 파이썬 2.5부터 지원됨

[sys 모듈로 입력 인수 주기]

도스(DOS)를 사용해 본 독자라면 다음과 같은 명령어를 사용해 봤을 것이다.

```
C:\> type a.txt
```

위의 type 명령어는 바로 뒤에 적힌 파일 이름을 인수로 받아 그 내용을 출력해 주는 도스 명령어이다. 대부분의 도스 명령어들은 다음과 같이 명령행(도스 창)에서 입력 인수를 직접 주어 프로그램을 실행시키는 방식을 따른다. 이러한 기능을 파이썬 프로그램에도 적용시킬 수가 있다.

도스 명령어 [인수1 인수2 ...]

파이썬에서는 sys라는 모듈을 이용하여 입력 인수를 직접 줄 수 있다. sys 모듈을 이용하려면 아래 예의 import sys처럼 import라는 명령어를 사용해야 한다.

※ 모듈을 이용하고 만드는 방법에 대해서는 뒤에서 자세히 다룰 것이다.

```
#sys1.py
import sys

args = sys.argv[1:]
for i in args:
    print(i)
```

위의 예는 입력받은 인수들을 for문을 이용해 차례대로 하나씩 출력하는 예이다. sys 모듈의 argv는 명령창에서 입력한 인수들을 의미한다. 즉, 아래와 같이 입력했다면 argv[0]는 파일 이름인 sys1.py 가 되고 argv[1]부터는 뒤에 따라오는 인수들이 차례로 argv의 요소가 된다.



이 프로그램을 C:/doit 디렉터리에 저장한 후 입력 인수를 함께 주어 실행시키면 다음과 같은 결과값을 얻을 수 있다.

```
C:\doit>python sys1.py aaa bbb ccc
aaa
bbb
ccc
```

위의 예를 이용해서 간단한 스크립트를 하나 만들어 보자.

```
#sys2.py
import sys
args = sys.argv[1:]
for i in args:
    print(i.upper(), end=' ')
```

문자열 관련 함수인 `upper()`를 이용하여 명령 행에 입력된 소문자를 대문자로 바꾸어 주는 간단한 프로그램이다. 명령 프롬프트 창에서 다음과 같이 입력해 보자.

※ sys2.py 파일이 C:\doit 디렉터리 안에 있어야만 한다.

```
C:\doit>python sys2.py life is too short, you need python
```

결과는 다음과 같다.

```
LIFE IS TOO SHORT, YOU NEED PYTHON
```

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#04-3>)

[문제1] 파일 읽고 출력하기

다음은 “test.txt”라는 파일에 “Life is too short”라는 문자열을 저장한 후 다시 그 파일을 읽어서 출력하는 프로그램이다.

```
f1 = open("test.txt", 'w')
f1.write("Life is too short")

f2 = open("test.txt", 'r')
print(f2.read())
```

이 프로그램은 우리가 예상한 “Life is too short”라는 문장을 출력하지 않는다. 우리가 예상한 값을 출력할 수 있도록 프로그램을 수정하시오.

[문제2] 파일저장

사용자의 입력을 파일(test.txt)에 저장하는 프로그램을 작성하시오. (단, 프로그램을 다시 실행하더라도 기존 작성한 내용을 유지하고 새로 입력한 내용이 추가되어야 한다.)

다음은 실행 예제이다.

```
스 | 스 | ㅇ ㅎ | ㄹ | ㄴ | ㅐ | ㅣ | ㅐ | ㅣ | ㅇ ㅍ | ㅇ ㅣ | ㅇ ㅍ :
```

실행 할 때마다 사용자가 입력한 내용이 test.txt파일에 추가되어야 한다.

[문제3] 역순 저장

다음과 같은 내용의 파일 abc.txt가 있다.

abc.txt

```
AAA  
BBB  
CCC  
DDD  
EEE
```

이 파일의 내용을 다음과 같이 역순으로 바꾸어 저장하시오.

```
EEE  
DDD  
CCC  
BBB  
AAA
```

[문제4] 파일 수정

다음과 같은 내용을 지닌 파일 test.txt 가 있다.

test.txt

```
Life is too short  
you need java
```

이 파일의 내용중 java라는 문자열을 python으로 바꾸어서 저장하시오.

[문제5] 평균값 구하기

다음과 같이 총 10줄로 이루어진 sample.txt 파일이 있다.

sample.txt

```
70  
60  
55  
75  
95  
90  
80  
80  
85  
100
```

sample.txt 파일의 숫자값을 모두 읽어 총합과 평균값을 구한 후 평균값을 result.txt라는 파일에 쓰는 프로그램을 작성해 보자.

05장 파이썬 날개달기

이제 프로그래밍의 꽃이라 할 수 있는 클래스와 함께 모듈, 예외 처리 및 파이썬 라이브러리에 대해서 알아보자. 이번 장을 끝으로 여러분은 파이썬 프로그램을 작성하기 위해 알아야 할 대부분의 내용들을 배우게 된다.

05-1 클래스

초보 개발자들에게 클래스(class)는 넘기 힘든 장벽과도 같은 존재이다. 독자들 중에도 클래스라는 단어를 처음 접하는 이들이 있을 것이다. 여기서는 클래스가 왜 필요한지, 도대체 클래스라는 것은 무엇인지 아주 기초적인 것부터 차근차근 함께 알아보자.

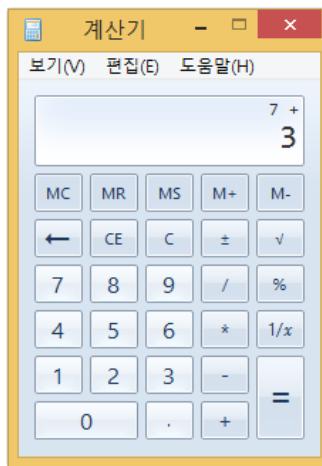
클래스는 도대체 왜 필요한가?

가장 많이 사용하는 프로그래밍 언어 중 하나인 C 언어에는 클래스가 없다. 이 말은 굳이 클래스 없이도 프로그램을 충분히 만들 수 있다는 말과도 같다. 파이썬으로 잘 만들어진 프로그램들을 살펴보아도 클래스를 이용하지 않고 작성된 것들이 상당히 많다. 클래스는 지금까지 공부한 함수나 자료형처럼 프로그램 작성을 위해 꼭 필요한 요소는 아니다.

하지만 프로그램 작성시 클래스를 적재적소에 이용하면 프로그래머가 얻을 수 있는 이익은 상당하다.

예제를 통해 한번 생각해 보자.

여러분 모두 계산기를 사용해 봤을 것이다. 계산기에 3이라는 숫자를 입력하고 + 기호를 입력한 후 4를 입력하면 결과값으로 7을 보여준다. 다시 한 번 + 기호를 입력한 후 3을 입력하면 기존 결과값 7에 3을 더해 10을 보여준다. 즉, 계산기는 이전에 계산된 결과값을 항상 메모리 어딘가에 저장하고 있어야 한다.



※ 계산기는 이전에 계산된 결과값을 기억하고 있어야 한다.

이런 내용을 우리가 앞서 익힌 함수를 이용해 구현해 보자. 계산기의 “더하기” 기능을 구현한 파이썬 코드는 다음과 같다.

```

result = 0

def add(num):
    global result
    result += num
    return result

print(add(3))
print(add(4))

```

※ add 함수는 입력 인수로 num을 받으면 이전에 계산된 결과값에 더한 후 리턴하는 함수이다.

이전에 계산된 결과값을 유지하기 위해서 result라는 전역 변수(global)를 사용했다. 실행하면 예상한 대로 다음과 같은 결과값이 출력된다.

```

3
7

```

그런데 만약 한 프로그램에서 2개의 계산기가 필요한 상황이 발생하면 어떻게 해야 할까? 각각의 계산기는 각각의 결과값을 유지해야 하기 때문에 위와 같이 add 함수 하나만으로는 결과값을 바로 유지할 수 없다.

이런 상황을 해결하려면 다음과 같이 함수를 각각 따로 만들어야 한다.

```

result1 = 0
result2 = 0

def add1(num):
    global result1
    result1 += num
    return result1

def add2(num):
    global result2
    result2 += num
    return result2

print(add1(3))
print(add1(4))
print(add2(3))
print(add2(7))

```

똑같은 일을 하는 add1과 add2라는 함수가 만들어졌고 각각의 함수에서 계산된 결과값을 유지하면서 저장하기 위한 전역 변수 result1, result2가 필요하게 되었다.

결과값은 다음과 같이 의도한 대로 출력된다.

```

3
7
3
10

```

계산기 1의 결과값이 계산기 2에 아무런 영향을 끼치지 않음을 확인할 수 있다. 하지만 계산기가 3개, 5개, 10개로 점점 더 많이 필요해진다면 어떻게 해야 할 것인가? 그때마다 전역변수와 함수를 추가할 것인가? 여기에 빼기나 곱하기등의 기능을 추가해야 한다면 상황은 점점 더 어려워 질 것이다.

아직 클래스에 대해서 배우진 않았지만 위와 같은 경우 클래스를 이용하면 다음과 같이 간단하게 해결할 수 있다.

※ 아래 예시 클래스를 아직은 이해하지 못해도 좋다. 곧 자세하게 배울 것이다. 여기서는 클래스를 개념적으로만 이해하면 된다.

```

class Calculator:
    def __init__(self):
        self.result = 0

    def add(self, num):
        self.result += num
        return self.result

cal1 = Calculator()
cal2 = Calculator()

print(cal1.add(3))
print(cal1.add(4))
print(cal2.add(3))
print(cal2.add(7))

```

실행하면 함수 2개를 사용했을 때와 동일한 결과가 출력된다.

```

3
7
3
10

```

Calculator 클래스로 만들어진 cal1, cal2라는 별개의 계산기(파이썬에서는 이것을 객체라고 한다)가 각각의 역할을 수행한다. 그리고 계산기(cal1, cal2)의 결과값 역시 다른 계산기의 결과값과 상관없이 독립적인 결과값을 유지한다. 클래스를 이용하면 계산기의 개수가 늘어나더라도 객체를 생성하기만 하면 되기 때문에 함수를 사용하는 경우와 달리 매우 간단해진다.

만약 빼기 기능이 더해진다고 해도 Calculator 클래스에 다음과 같은 빼기 기능 함수를 추가해 주면 된다.

```

def sub(self, num):
    self.result -= num
    return self.result

```

클래스의 이점은 단순히 이것만이 아니다. 하지만 이것 하나만으로도 “도대체 왜 클래스가 필요한 것일까?”라는 근본적인 물음에 대한 해답이 되었을 것이다.

클래스와 객체

클래스를 가장 잘 설명 해 주는 다음의 사진을 보자.



과자를 만드는 과자틀과 만들어진 과자들이다.

- 과자틀 → 클래스 (class)
- 과자틀에 의해서 만들어진 과자들 → 객체 (object)

이 사진을 보면 클래스가 어떤건지 감이 잡힐 것이다.

이 절에서 설명할 클래스는 과자틀과 비슷하다.

클래스(class)란 똑같은 무엇인가를 계속해서 만들어낼 수 있는 설계 도면 같은 것이고(과자 틀), 객체(object)란 클래스에 의해서 만들어진 피조물(과자틀에 의해서 만들어진 과자)을 뜻한다.

클래스에 의해서 만들어진 객체에는 중요한 특징이 있다. 그것은 객체별로 독립적인 성격을 갖는다는 것이다. 과자틀에 의해서 만들어진 과자에 구멍을 뚫거나 조금 베어먹더라도 다른 과자들에

는 아무 영향이 없는것과 마찬가지로 동일한 클래스에 의해 생성된 객체들은 서로에게 전혀 영향을 주지 않는다.

다음은 파이썬 클래스의 가장 간단한 예이다.

```
>>> class Cookie:  
>>>     pass
```

위의 클래스는 아무런 기능도 갖고 있지 않은 껍질뿐인 클래스이다. 하지만 이렇게 껍질뿐인 클래스도 객체를 생성하는 기능은 가지고 있다. “과자 틀”로 “과자”를 만드는 것처럼 말이다.

객체는 클래스에 의해서 만들어지며 1개의 클래스는 무수히 많은 객체를 만들어낼 수 있다. 위에서 만든 Cookie 클래스의 객체를 만드는 방법은 다음과 같다.

```
>>> a = Cookie()  
>>> b = Cookie()
```

Cookie()의 결과값을 돌려받은 a와 b가 바로 객체이다. 마치 함수를 사용해서 그 결과값을 돌려 받는 모습과 비슷하다.

[객체와 인스턴스의 차이]

클래스에 의해서 만들어진 객체를 인스턴스라고도 한다. 그렇다면 객체와 인스턴스의 차이는 무엇일까? 이렇게 생각해 보자. a = Cookie() 이렇게 만들어진 a는 객체이다. 그리고 a라는 객체는 Cookie의 인스턴스이다. 즉, 인스턴스라는 말은 특정 객체(a)가 어떤 클래스(Cookie)의 객체인지 를 관계 위주로 설명할 때 사용된다. 즉, “a는 인스턴스” 보다는 “a는 객체”라는 표현이 어울리며, “a는 Cookie의 객체” 보다는 “a는 Cookie의 인스턴스”라는 표현이 훨씬 잘 어울린다.

사칙연산 클래스 만들기

“백견(見)이 불여 일타(打)”라고 했다. 클래스도 직접 만들어 가며 배워 보도록 하자.

사칙연산을 해 주는 클래스를 작성해 보도록 하자. 사칙연산은 더하기, 빼기, 나누기, 곱하기를 말한다.

클래스를 어떻게 만들지 먼저 구상하기

클래스는 무작정 만들기 보다는 클래스에 의해서 만들어진 객체를 중심으로 어떤 식으로 동작하게 할 것인지 미리 구상을 한 후에 생각했던 것들을 하나씩 해결하면서 완성해 나가는 것이 좋다.

사칙연산을 가능하게 하는 FourCal이라는 클래스가 다음처럼 동작한다고 가정해 보자.

먼저 `a = FourCal()`처럼 입력해서 `a`라는 객체를 만든다.

```
>>> a = FourCal()
```

그런 다음 `a.setdata(4, 2)`처럼 입력해서 4와 2라는 숫자를 객체 `a`에 지정해 주고

```
>>> a.setdata(4, 2)
```

`a.sum()`을 수행하면 두 수를 합한 결과($4 + 2$)를 돌려주고

```
>>> print(a.sum())
6
```

`a.mul()`을 수행하면 두 수를 곱한 결과($4 * 2$)를 돌려주고

```
>>> print(a.mul())
8
```

`a.sub()`를 수행하면 두 수를 뺀 결과($4 - 2$)를 돌려주고

```
>>> print(a.sub())
2
```

`a.div()`를 수행하면 두 수를 나눈 결과($4 / 2$)를 돌려준다.

```
>>> print(a.div())
2
```

이렇게 동작하는 `FourCal` 클래스를 만드는 것이 바로 우리의 목표이다.

클래스 구조 만들기

자, 그렇다면 지금부터는 앞에서 구상했던 것처럼 동작하는 클래스를 만들어 보자. 제일 먼저 할 일은 `a = FourCal()`처럼 객체를 만들 수 있게 하는 것이다. 일단은 아무 기능이 없어도 되기 때문에 만드는 것은 매우 간단하다. 다음을 따라 해보자.

```
>>> class FourCal:
...     pass
...
>>>
```

우선 대화형 인터프리터에서 pass란 문장만을 포함한 FourCal 클래스를 만든다. 현재 상태에서 FourCal 클래스는 아무런 변수나 메서드도 포함하지 않지만 우리가 원하는 객체 a를 만들 수 있는 기능은 가지고 있다. 확인해 보자.

※ pass는 아무것도 수행하지 않는 문법이다. 임시로 코드를 작성할 때 주로 사용한다.

```
>>> a = FourCal()
>>> type(a)
<class '__main__.FourCal'>
```

위와 같이 a = FourCal()로 a라는 객체를 먼저 만들고 그 다음에 type(a)로 a라는 객체가 어떤 타입인지 알아보았다. 역시 객체 a가 FourCal 클래스의 인스턴스임을 알 수 있다.

※ type 함수는 파이썬이 자체적으로 가지고 있는 내장 함수로 객체의 타입을 출력한다.

객체에 숫자 지정할 수 있게 만들기

하지만 생성된 객체 a는 아직 아무런 기능도 하지 못한다. 이제 더하기, 나누기, 곱하기, 빼기등의 기능을 하는 객체를 만들어야 한다. 그런데 이러한 기능을 갖춘 객체를 만들려면 우선적으로 a라는 객체에 사칙연산을 할 때 사용할 2개의 숫자를 먼저 알려주어야 한다. 다음과 같이 연산을 수행할 대상(4, 2)을 객체에 지정할 수 있게 만들어 보자.

```
>>> a.setdata(4, 2)
```

위의 문장이 수행되려면 다음과 같이 소스 코드를 작성해야 한다.

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...
>>>
```

이전에 만들었던 FourCal 클래스에서 pass라는 문장을 삭제하고 class 내부에 setdata라는 함수를 만들었다. 클래스 안에 구현된 함수는 다른말로 **메서드(Method)**라고 부른다. 앞으로 클래스 내의 함수는 항상 메서드라고 표현할테니 용어를 기억해 두도록 하자.

일반적인 함수를 만들 때 우리는 다음과 같이 작성한다.

```
def 함수명(매개변수):
    수행할 문장
    ...
```

메서드도 클래스에 포함되어 있다는 점만 제외하면 일반함수와 다를 것이 없다.

setdata 메서드를 다시 보면 아래와 같다.

```
def setdata(self, first, second): # ① 메서드의 매개변수
    self.first = first           # ② 메서드의 수행문
    self.second = second          # ② 메서드의 수행문
```

① setdata 메서드의 매개변수

setdata 메서드는 매개변수로 self, first, second라는 3개의 입력값을 받는다. 그런데 일반적인 함수와는 달리 메서드의 첫 번째 매개변수 self는 특별한 의미를 가지고 있다.

self에 어떤 특별한 의미가 있는지 다음의 예를 보면서 자세히 살펴보자.

```
>>> a = FourCal()
>>> a.setdata(4, 2)
```

위에서 보는 것처럼 먼저 a라는 객체를 만들었다. 그리고 a라는 객체를 통해 setdata 메서드를 호출했다.

※ 객체를 통해 클래스의 메서드를 호출하려면 `a.setdata(4, 2)` 와 같이 도트(.) 연산자를 이용해야 한다.

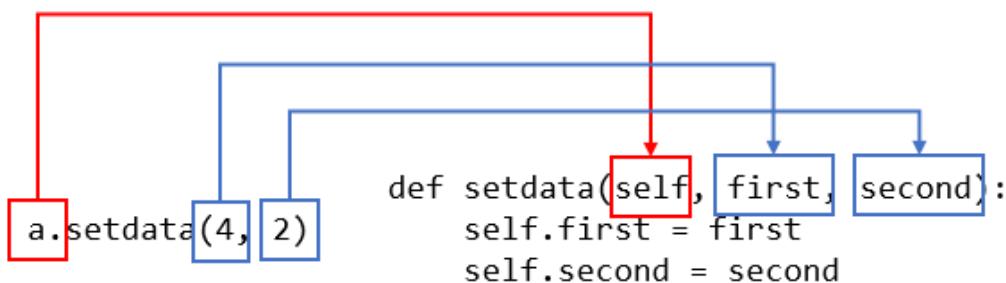
그런데 뭔가 좀 이상하지 않은가?

`setdata`라는 메서드는 `self, first, second`라는 총 3개의 매개변수를 필요로 하는데 실제로는 `a.setdata(4, 2)`처럼 4와 2라는 2개의 값만 전달한 것이다.

왜 그럴까?

그 이유는 `a.setdata(4, 2)`처럼 호출하면 `setdata` 메서드의 첫 번째 매개변수 `self`에는 `setdata` 메서드를 호출한 객체 `a`가 자동으로 전달되기 때문이다.

다음 그림을 보면 객체와 호출 입력 값들이 메서드에 어떻게 전달되는지 쉽게 이해가 갈 것이다.



파이썬 클래스에서 가장 헷갈리는 부분이 바로 이 부분이다. `setdata`라는 메서드는 매개변수로 3개를 필요로 하는데 왜 `a.setdata(4, 2)`처럼 2개만 입력해도 실행이 되는가? 이 질문에 대한 답변을 여러분도 이제는 알았을 것이다.

파이썬 메서드의 첫번째 매개변수명은 관례적으로 `self`라는 이름을 사용한다. 호출 시 호출한 객체 자신이 전달되기 때문에 `self`(“self”는 자기자신이라는 뜻을 가진 영어단어이다.)라는 이름을 사용하게 된 것이다. 물론 `self`말고 다른 이름을 사용해도 상관은 없다.

※ 메서드의 첫번째 매개변수를 `self`를 명시적으로 구현해야 하는 것은 파이썬만의 독특한 특징이다. 예를들어 자바같은 언어는 첫번째 매개변수인 `self`가 필요없다.

[메서드의 또 다른 호출 방법]

잘 사용하지는 않지만 다음과 같이 메서드를 호출하는 것도 가능하다.

```

>>> a = FourCal()
>>> FourCal.setdata(a, 4, 2)
  
```

위와 같이 “클래스명.메서드” 형태로 호출할 때는 객체 a를 입력 인수로 꼭 넣어 주어야 한다. 반면에 다음처럼 “객체.메서드” 형태로 호출할 때는 첫 번째 입력 인수(self)를 반드시 생략해야 한다.

```
>>> a = FourCal()
>>> a.setdata(4, 2)
```

그 다음으로 중요한 사항을 살펴보자.

② setdata 메서드의 수행문

setdata 메서드에는 수행할 문장이 2개 있다.

```
self.first = first
self.second = second
```

위 수행문이 뜻하는 바는 무엇일까? 입력 인수로 받은 first는 4이고 second는 2라는 것은 앞에서 이미 알았다. 그렇다면 위의 문장은 다음과 같이 바뀔 것이다.

```
self.first = 4
self.second = 2
```

여기서 중요한 것은 바로 self이다. self는 a.setdata(4, 2)처럼 호출했을 때 자동으로 들어오는 객체 a라고 했다. 그렇다면 self.first의 의미는 무엇이겠는가? 당연히 a.first가 될 것이다. 또한 self.second는 당연히 a.second가 될 것이다.

따라서 위의 두 문장을 풀어서 쓰면 다음과 같이 된다.

```
a.first = 4
a.second = 2
```

위와 같이 바뀐 문장이 실행되어 결국 a객체에는 first와 second라는 객체변수가 생성된다.

객체변수는 다음과 같이 만들어진다.

객체.객체변수 = 값

(예: a.first = 4)

객체변수(instance variable)는 객체에 정의된 변수를 의미하며 객체간 서로 공유되지 않는 특징을 갖는다.

※ 객체변수는 속성, 멤버변수 또는 인스턴스 변수라고도 표현한다.

정말로 객체변수가 생성되었는지 다음과 같이 확인해 보자.

```
>>> a = FourCal()
>>> a.setdata(4, 2)
>>> print(a.first)
4
>>> print(a.second)
2
```

a객체에 객체변수 first와 second가 생성되었음을 확인할 수 있다.

이번에는 b라는 객체를 하나 더 만들어 보자.

```
>>> b = FourCal()
>>> b.setdata(3, 7)
>>> print(b.first)
3
>>> print(a.first)
4
```

a와 b라는 객체는 모두 first라는 변수를 가지고 있지만 그 변수의 값은 각기 다르다. b 객체의 first 변수에 3이라는 값을 대입하더라도 a의 first 값이 3으로 변경되지는 않는다. a, b 객체의 first변수는 고유의 저장 영역을 가지고 있는 객체 변수이기 때문이다.

객체 변수(예: a.first)는 그 객체의 고유한 값을 저장할 수 있는 공간이다. 객체 변수는 다른 객체들에 의해 영향받지 않고 독립적으로 그 값을 유지한다는 점을 꼭 기억하도록 하자. 클래스에서는 이 부분을 이해하는 것이 가장 중요하다.

다음은 현재까지 완성된 FourCal클래스이다.

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...
>>>
```

지금까지 살펴본 내용이 바로 위의 4줄을 설명하기 위한 것이었다. 위에서 설명한 것들이 이해가 되지 않는다면 다시 한 번 읽어 보기 바란다. 이 부분을 이해하지 못하면 다음으로 넘어갈 수 없기 때문이다.

더하기 기능 만들기

자! 그럼 2개의 숫자값을 설정해 주었으니 2개의 숫자를 더하는 기능을 추가해 보자. 우리는 다음과 같이 더하기 기능을 갖춘 클래스를 만들어야 한다.

```
>>> a = FourCal()
>>> a.setdata(4, 2)
>>> print(a.sum())
6
```

이를 가능하게 하기 위해 FourCal 클래스를 다음과 같이 만들어 보자.

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...     def sum(self):
...         result = self.first + self.second
...         return result
...
>>>
```

새롭게 추가된 것은 sum이라는 메서드이다.

클래스를 위와 같이 변경하고 다음과 같이 클래스를 사용해 보자.

```
>>> a = FourCal()
>>> a.setdata(4, 2)
```

위와 같이 호출하면 앞서 살펴보았듯이 a객체의 first, second 객체변수에는 각각 4와 2라는 값이 저장될 것이다.

이제 sum메서드를 호출 해 보자.

```
>>> print(a.sum())
>>> 6
```

`a.sum()` 이라고 호출하면 `sum` 메서드가 호출되어 6이라는 값이 출력될 것이다. 어떤 과정을 거쳐 6이라는 값이 출력되는지 `sum` 메서드를 따로 빼어 내서 자세히 살펴보도록 하자.

```
def sum(self):
    result = self.first + self.second
    return result
```

`sum` 메서드의 매개변수는 `self`이고 리턴값은 `result`이다.

리턴 값인 `result`를 계산하는 부분은 다음과 같다.

```
result = self.first + self.second
```

`a.sum()` 과 같이 `a` 객체에 의해 `sum` 메서드가 수행되면 `sum` 메서드의 `self`에는 객체 `a`가 자동으로 입력되므로 위의 내용은 아래와 같이 해석된다.

```
result = a.first + a.second
```

위의 내용은 `a.sum()` 메소드 호출전에 `a.setdata(4, 2)` 가 먼저 호출되어 `a.first = 4, a.second = 2` 라고 이미 설정되었기 때문에 다시 다음과 같이 해석된다.

```
result = 4 + 2
```

따라서 다음과 같이 `a.sum()` 을 호출하면 6이 리턴된다.

```
>>> print(a.sum())
6
```

여기까지 모두 이해한 독자라면 클래스에 대해 80% 이상을 안 것이다. 파이썬의 클래스는 그다지 어렵지 않다.

곱하기, 빼기, 나누기 가능 만들기

이번에는 곱하기, 빼기, 나누기 등을 할 수 있게 만들어 보자.

```
>>> class FourCal:  
...     def setdata(self, first, second):  
...         self.first = first  
...         self.second = second  
...     def sum(self):  
...         result = self.first + self.second  
...         return result  
...     def mul(self):  
...         result = self.first * self.second  
...         return result  
...     def sub(self):  
...         result = self.first - self.second  
...         return result  
...     def div(self):  
...         result = self.first / self.second  
...         return result  
...  
>>>
```

mul, sub, div 모두 sum 메서드에서 배운 것과 동일한 방법이니 따로 설명하지는 않겠다.

정말로 모든 것이 제대로 동작하는지 확인해 보자.

```
>>> a = FourCal()
>>> b = FourCal()
>>> a.setdata(4, 2)
>>> b.setdata(3, 7)
>>> a.sum()
6
>>> a.mul()
8
>>> a.sub()
2
>>> a.div()
2
>>> b.sum()
10
>>> b.mul()
21
>>> b.sub()
-4
>>> b.div()
0
```

이상과 같이 우리가 목표로 했던 사칙연산 기능을 가진 클래스를 만들어 보았다.

생성자 (Constructor)

이번에는 우리가 만든 FourCal 클래스를 다음과 같이 사용해 보자.

```
>>> a = FourCal()
>>> a.sum()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 6, in sum
AttributeError: 'FourCal' object has no attribute 'first'
```

FourCal 클래스의 인스턴스 a에 setdata 메서드를 수행하지 않고 sum 메서드를 수행하면 “AttributeError: ‘FourCal’ object has no attribute ‘first’”라는 오류가 발생하게 된다. setdata

메서드를 수행해야 객체 a의 객체변수 first와 second이 생성되기 때문이다.

이렇게 객체에 초기값을 설정해야 할 필요가 있을때는 setdata와 같은 메서드를 호출하여 초기값을 설정하기 보다는 생성자를 구현하는 것이 안전한 방법이다.

생성자(Constructor)란 객체가 생성될 때 자동으로 호출되는 메서드를 의미한다.

파이썬 메서드명으로 `__init__` 을 사용하면 이 메서드는 생성자가 된다. 다음과 같이 FourCal 클래스에 생성자를 추가해 보자.

* `__init__` 메서드의 init 앞 뒤로 붙은 __는 언더스코어(_) 두 개를 붙여서 사용해야 한다.

```
>>> class FourCal:
...     def __init__(self, first, second):
...         self.first = first
...         self.second = second
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...     def sum(self):
...         result = self.first + self.second
...         return result
...     def mul(self):
...         result = self.first * self.second
...         return result
...     def sub(self):
...         result = self.first - self.second
...         return result
...     def div(self):
...         result = self.first / self.second
...         return result
...
>>>
```

새롭게 추가된 생성자인 `__init__` 메서드만 따로 떼어 내서 살펴보자.

```
def __init__(self, first, second):
    self.first = first
    self.second = second
```

`__init__` 메서드는 `setdata` 메서드와 이름만 다르고 모든게 동일하다. 단, 메서드 이름을 `__init__` 으로 했기 때문에 생성자로 인식되어 객체가 생성되는 시점에 자동으로 호출되는 차이가 있다.

이제 다음처럼 예제를 수행 해 보자.

```
>>> a = FourCal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'first' and
'second'
```

`a = FourCal()` 수행 시 생성자 `__init__` 이 호출되어 위와 같은 오류가 발생했다. 오류가 발생한 이유는 생성자의 매개변수인 `first`와 `second`에 해당되는 값이 전달되지 않았기 때문이다.

위 오류를 해결하려면 다음처럼 `first`와 `second`에 해당되는 값을 전달하여 객체를 생성해야 한다.

```
>>> a = FourCal(4, 2)
>>>
```

위와 같이 수행하면 `__init__` 메서드의 매개변수에는 각각 다음과 같은 값들이 대입된다.

매개변수	값
<code>self</code>	생성되는 객체
<code>first</code>	4
<code>second</code>	2

* `__init__` 메서드도 다른 메서드와 마찬가지로 첫번째 매개변수 `self`에 생성되는 객체가 자동으로 전달된다는 점을 기억하도록 하자.

따라서 `__init__` 메서드가 호출되면 `setdata` 메서드를 호출했을 때와 마찬가지로 `first`와 `second`라는 객체변수가 생성될 것이다.

다음과 같이 객체변수의 값을 확인 해 보자.

```
>>> a = FourCal(4, 2)
>>> print(a.first)
4
>>> print(a.second)
2
```

sum이나 div등의 메서드도 잘 동작하는지 다음과 같이 확인 해 보자.

```
>>> a = FourCal(4, 2)
>>> a.sum()
6
>>> a.div()
2.0
```

이상없이 잘 동작하는 것을 확인할 수 있다.

클래스의 상속

상속(Inheritance)이란 “물려받다”라는 뜻으로, “재산을 상속받다”라고 할 때의 상속과 같은 의미이다. 클래스에도 이런 개념을 적용할 수가 있다. 어떤 클래스를 만들 때 다른 클래스의 기능을 물려받을 수 있게 만드는 것이다.

이번에는 상속의 개념을 이용하여 우리가 만든 FourCal 클래스에 ab (a의 b승)을 구할 수 있는 기능을 추가 해 보자.

FourCal 클래스를 상속하는 MoreFourCal 클래스는 다음과 같이 간단하게 만들 수 있다. (앞서 구현한 FourCal 클래스는 이미 만들어 놓았다고 가정한다.)

```
>>> class MoreFourCal(FourCal):
...     pass
...
>>>
```

클래스를 상속하기 위해서는 다음처럼 클래스명 뒤 팔호 안에 상속할 클래스명을 넣어 주면 된다.

class 클래스명(상속할 클래스명)

MoreFourCal 클래스는 FourCal 클래스를 상속했으므로 FourCal 클래스의 모든 기능을 사용할 수 있어야 할 것이다.

다음과 같이 확인 해보자.

```
>>> a = MoreFourCal(4, 2)
>>> a.sum()
6
>>> a.mul()
8
>>> a.sub()
2
>>> a.div()
2
```

상속받은 FourCal 클래스의 기능을 모두 사용할 수 있음을 확인할 수 있다.

** 알아두기 **

보통 상속은 기존 클래스를 변경하지 않고 기능을 추가하거나 기존 기능을 변경하려고 할 때 사용한다.

클래스에 기능을 추가하고 싶으면 기존 클래스를 수정하면 되는데 왜 굳이 상속을 받아서 처리해야 하지? 라는 의문이 들 수도 있다. 하지만 기존 클래스가 라이브러리 형태로 제공되거나 수정이 허용되지 않는 상황이라면 상속을 이용해야만 할 것이다.

이제 원래 목적인 a의 b승 (ab) 을 계산해 주는 MoreFourCal 클래스를 만들어 보자.

```
>>> class MoreFourCal(FourCal):
...     def pow(self):
...         result = self.first ** self.second
...         return result
...
>>>
```

pass 문장을 삭제하고 위와 같이 두 수의 거듭 제곱을 구할 수 있는 pow 메서드를 추가해 주었다. 그리고 다음과 같이 pow 메서드를 수행 해 보자.

```
>>> a = MoreFourCal(4, 2)
>>> a.pow()
16
```

MoreFourCal 클래스로 만들어진 a객체에 4와 2라는 값을 세팅한 후 pow메서드를 호출하면 4의 2승 (4²)인 16을 리턴해 주는 것을 확인할 수 있다.

상속은 MoreFourCal 클래스처럼 기존 클래스(FourCal)는 그대로 놔둔채로 클래스의 기능을 확장시키고자 할 때 주로 사용된다.

메서드 오버라이딩

이번에는 FourCal 클래스를 다음과 같이 실행 해 보자.

```
>>> a = FourCal(4, 0)
>>> a.div()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    result = self.first / self.second
ZeroDivisionError: division by zero
```

FourCal클래스의 객체 a에 4와 0이라는 값을 세팅하고 div메서드를 호출하면 4를 0으로 나누려고 하기 때문에 위와같은 ZeroDivisionError 오류가 발생하게 된다.

하지만 0으로 나눌 때 오류가 아닌 0을 리턴하도록 만들고 싶다면 어떻게 해야 할까?

다음과 같이 FourCal클래스를 상속하는 SafeFourCal클래스를 만들어 보자.

```
>>> class SafeFourCal(FourCal):
...     def div(self):
...         if self.second == 0: # 나누는 값이 0인 경우 0을 리턴하도록 수정
...             return 0
...         else:
...             return self.first / self.second
...
>>>
```

SafeFourCal 클래스는 FourCal 클래스에 있는 div라는 메서드를 동일한 이름으로 다시 작성하였다. 이렇게 부모 클래스(상속한 클래스)에 있는 메서드를 동일한 이름으로 다시 만드는 것을 **메서드 오버라이딩**(Overriding, 덮어쓰기)이라고 한다. 이렇게 메서드를 오버라이딩하면 부모 클래스의 메서드 대신 오버라이딩한 메서드가 호출된다.

SafeFourCal 클래스에 오버라이딩한 div 메서드는 나누는 값이 0인 경우에는 0을 리턴하도록 수정했다.

이제 다시 위에서 수행했던 예제를 FourCal 클래스 대신 SafeFourCal 클래스를 이용하여 수행해 보자.

```
>>> a = SafeFourCal(4, 0)
>>> a.div()
0
```

FourCal 클래스와는 달리 ZeroDivisionError가 발생하지 않고 의도했던 대로 0을 리턴하는 것을 확인할 수 있을 것이다.

클래스 변수

객체변수는 다른 객체들에 의해 영향받지 않고 독립적으로 그 값을 유지한다는 점을 이미 알아보았다. 이번에는 객체변수와는 성격이 다른 클래스 변수에 대해서 알아보자.

다음의 클래스를 작성해 보자.

```
>>> class Family:
...     lastname = "김"
...
```

Family 클래스에 선언된 lastname이 바로 클래스 변수이다. 클래스 변수는 클래스 안에 함수를 선언하는 것과 마찬가지로 클래스 안에 변수를 선언하여 생성한다.

이제 Family 클래스를 다음과 같이 사용해 보자.

```
>>> print(Family.lastname)
김
```

클래스 변수는 위 예와 같이 클래스명.클래스변수로 사용할 수 있다.

또는 다음과 같이 Family 클래스에 의해 생성된 객체를 통해서도 클래스 변수를 사용할 수 있다.

```
>>> a = Family()
>>> b = Family()
>>> print(a.lastname)
김
>>> print(b.lastname)
김
```

만약 Family 클래스의 lastname을 다음과 같이 “박”이라는 문자열로 바꾸면 어떻게 될까?

```
>>> Family.lastname = "박"
```

다음과 같이 확인 해 보자.

```
>>> print(a.lastname)
박
>>> print(b.lastname)
박
```

클래스 변수의 값을 변경했더니 클래스에 의해 생성된 객체들의 lastname 값들도 모두 함께 변경 된다는 것을 확인할 수 있다. 즉, 클래스 변수는 클래스에 의해 생성된 모든 객체에 공유된다는 특징을 갖고 있다.

id 함수를 이용하면 클래스 변수가 공유된다는 사실을 증명할 수 있다. (※ id 함수는 객체의 주소를 리턴해 주는 파이썬 내장함수이다.)

```
>>> id(Family.lastname)
4480159136
>>> id(a.lastname)
4480159136
>>> id(b.lastname)
4480159136
```

id 값이 모두 같으므로 Family.lastname, a.lastname, b.lastname은 모두 같은 곳을 바라보고 있음을 알 수 있다.

클래스 변수를 가장 늦게 설명하는 이유는 클래스에서 클래스 변수보다는 객체 변수가 훨씬 중

요하기 때문이다. 실제 실무적인 프로그래밍을 할 때도 클래스 변수보다는 객체 변수를 사용하는 비율이 훨씬 높다.

클래스의 활용

지금까지 클래스가 무엇인지에 대해서 문법적인 측면에서 살펴보았다. 이번에는 활용적인 측면에서 한번 살펴보도록 하자.

우선 다음과 같은 규칙을 지닌 문자열이 있다고 가정해 보자.

홍길동|42|A

위 문자열은 이름, 나이, 성적을 |(파이프문자)로 구분하여 표기한 문자열이다. 예를들어 홍길동|42|A를 해석하면 다음과 같다.

항목	값
이름	홍길동
나이	42
성적	A

홍길동|42|A라는 문자열에서 나이를 추출해 내려면 다음과 같이 코딩해야 한다.

```
>>> data = "홍길동|42|A"
>>> tmp = data.split("|")
>>> age = tmp[1]
```

만약 이런 형식의 문자열을 전달하여 나이를 출력해야 하는 함수가 필요하다면 다음과 같이 작성해야 한다.

```
>>> def print_age(data):
...     tmp = data.split("|")
...     age = tmp[1]
...     print(age)
...
>>> data = "홍길동|42|A"
>>> print_age(data)
42
```

마찬가지로 이름과 점수를 출력해야 하는 함수가 필요하다면 다음과 같이 작성해야 한다.

```
>>> def print_grade(data):
...     tmp = data.split("|")
...     name = tmp[0]
...     grade = tmp[2]
...     print("%s님 당신의 점수는 %s입니다." % (name, grade))
...
>>> data = "홍길동|42|A"
>>> print_grade(data)
홍길동님 당신의 점수는 A입니다.
```

위 예에서 보듯이 이런 형태의 문자열을 함수 단위로 항상 주고 받아야 한다면 매번 문자열을 split 해서 사용해야 하므로 뭔가 개선이 필요함을 느낄 수 있을 것이다.

클래스를 이용하면 좀 더 개선된 코드를 작성할 수 있다.

다음과 같은 클래스를 작성해 보자.

```
>>> class Data:
...     def __init__(self, data):
...         tmp = data.split("|")
...         self.name = tmp[0]
...         self.age = tmp[1]
...         self.grade = tmp[2]
...
```

홍길동|42|A 와 같은 문자열을 생성자의 입력으로 받아서 name, age, grade라는 객체변수를 생성하는 Data클래스를 생성하였다.

위처럼 Data 클래스를 만들면 다음처럼 사용할 수 있게 된다.

```
>>> data = Data("홍길동|42|A")
>>> print(data.age)
42
>>> print(data.name)
홍길동
>>> print(data.grade)
A
```

클래스를 이용했더니 복잡한 문자열을 정형화된 객체로 사용할 수 있게 되었다. 정말 편리하지 않은가?

print_age와 print_grade 함수도 문자열 대신 객체를 전달하면 되기 때문에 다음처럼 간단해 진다.

```
>>> def print_age(data):
...     print(data.age)
...
>>> def print_grade(data):
...     print("%s님 당신의 점수는 %s입니다." % (data.name, data.grade))
...
>>> data = Data("홍길동|42|A")
>>> print_age(data)
42
>>> print_grade(data)
홍길동님 당신의 점수는 A입니다.
```

그런데 또 가만히 살펴보니 print_age, print_grade 함수를 Data클래스로 이동시켜도 좋을 것 같다. 왜냐하면 print_age나 print_grade라는 함수는 data객체에 의존적인 함수이므로 해당 클래스의 메서드로 만들어 주는것이 유리해 보이기 때문이다.

다음처럼 print_age, print_grade함수를 Data클래스의 메서드로 만들어 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> class Data:
...     def __init__(self, data):
...         tmp = data.split("|")
...         self.name = tmp[0]
...         self.age = tmp[1]
...         self.grade = tmp[2]
...     def print_age(self):
...         print(self.age)
...     def print_grade(self):
...         print("%s님 당신의 점수는 %s입니다." % (self.name, self.grade))
...
...
```

`print_age`, `print_grade` 함수는 이제 `Data`클래스의 메서드가 되었으므로 다음처럼 사용할 수 있다.

```
>>> data = Data("홍길동|42|A")
>>> data.print_age()
42
>>> data.print_grade()
홍길동님 당신의 점수는 A입니다.
```

처음에 작성했던 소스코드에 클래스를 적용했더니 사용성이 좋아지고 소스코드가 구조적으로 변경되었음을 느낄 수 있을 것이다.

클래스를 어떤 상황에서 사용하면 좋을지에 대한 규칙은 따로 없다. 클래스는 프로그램 작성시 꼭 필요한 요소가 아니기 때문이다. 여러분이 클래스를 잘 활용하려면 많은 경험을 해 보는 수 밖에 없다. 많은 코드를 작성해 보고 또 개선해보려는 노력을 많이 할 수록 클래스를 잘 활용할 수 있게 될 것이다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#05-1>)

[문제1] Calculator 1

다음은 계산기처럼 동작하는 `Calculator`클래스이다. `add`라는 메서드를 이용하면 현재 계산기의 객체 변수 `value`에 입력으로 받은 값을 더해 준다.

* 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
class Calculator:
    def __init__(self):
        self.value = 0

    def add(self, val):
        self.value += val
```

위와 같은 Calculator 클래스를 다음과 같이 사용하였다.

```
cal = Calculator()
cal.add(3)
cal.add(4)

print(cal.value) # 7을 출력해야 한다.
```

3과 4를 add했으므로 객체변수 value가 7이 되어 7을 출력해야 하지만 다음과 같은 오류가 발생했다.

```
Traceback (most recent call last):
  File "....py", line 9, in <module>
    cal.add(3)
TypeError: add() takes 1 positional argument but 2 were given
```

오류가 발생하지 않도록 클래스의 잘못된 부분을 찾아 고치시오.

[문제2] Calculator 2

다음과 같은 Calculator 클래스가 있다.

```
class Calculator:
    def __init__(self, init_value):
        self.value = init_value

    def add(self, val):
        self.value += val
```

이 클래스를 다음과 같이 사용해 보았다.

```
cal = Calculator()
cal.add(3)
cal.add(4)

print(cal.value)
```

위 코드를 실행했더니 다음과 같은 오류가 발생한다.

```
Traceback (most recent call last):
  File "....py", line 8, in <module>
    cal = Calculator()
TypeError: __init__() missing 1 required positional argument: 'init_value'
```

오류가 난 이유가 무엇인지 설명하고 오류를 해결하시오.

[문제3] UpgradeCalculator

다음은 Calculator 클래스이다.

```
class Calculator:
    def __init__(self):
        self.value = 0

    def add(self, val):
        self.value += val
```

위 클래스를 상속하는 UpgradeCalculator를 만들고 값을 뺄 수 있는 minus 메서드를 추가하시오. 즉, 다음과 같이 동작하는 클래스를 만드시오.

```
cal = UpgradeCalculator()
cal.add(10)
cal.minus(7)

print(cal.value) # 10이 3이 뺌 7이 더해져 10이 됨
```

[문제4] MaxLimitCalculator

이번에 여러분이 작성해야 하는 클래스는 MaxLimitCalculator 클래스이다. MaxLimitCalculator

클래스는 객체변수 value가 100이상의 값을 가질 수 없도록 제한하는 클래스이다. 즉, 다음과 같이 동작해야 한다.

```
cal = MaxLimitCalculator()
cal.add(50) # 50 더하기
cal.add(60) # 60 더하기

print(cal.value) # 100 출력
```

단, 한가지 전제 조건이 있다. 그 조건은 반드시 다음과 같은 Calculator 클래스를 상속해서 만들어야 한다는 것이다.

```
class Calculator:
    def __init__(self):
        self.value = 0

    def add(self, val):
        self.value += val
```

위와 같은 조건을 만족하는 MaxLimitCalculator 클래스를 작성하시오.

[문제5] Calculator 3

다음과 같이 동작하는 Calculator 클래스를 작성하시오.

```
cal1 = Calculator([1,2,3,4,5])
print(cal1.sum()) # 15 출력되며
print(cal1.avg()) # 3.0 출력된다

cal2 = Calculator([6,7,8,9,10])
print(cal2.sum()) # 40 출력되며
print(cal2.avg()) # 8.0 출력된다
```

05-2 모듈

모듈이란 함수나 변수 또는 클래스 들을 모아 놓은 파일이다. 모듈은 다른 파이썬 프로그램에서 불러와 사용할수 있게끔 만들어진 파일이라고도 할 수 있다. 우리는 파이썬으로 프로그래밍을 할 때 굉장히 많은 모듈을 사용한다. 다른 사람들이 이미 만들어 놓은 모듈을 사용할 수도 있고 우리가 직접 만들어서 사용할 수도 있다. 여기서는 모듈을 어떻게 만들고 사용할 수 있는지 알아보겠다.

모듈 만들고 불러 보기

모듈에 대해서 자세히 살펴보기 전에 간단한 모듈을 한번 만들어 보자.

```
# mod1.py
def sum(a, b):
    return a + b
```

위와 같이 sum 함수만 있는 파일 mod1.py를 만들고 C:\doit 디렉터리에 저장하자. 이 파일이 바로 모듈이다. 지금까지 에디터로 만들어 왔던 파일과 다르지 않다.

우리가 만든 mod1.py라는 파일, 즉 모듈을 파이썬에서 불러와 사용하려면 어떻게 해야 할까? 먼저 아래와 같이 명령 프롬프트 창을 열고 mod1.py를 저장한 디렉터리(이 책에서는 C:\doit)로 이동한 다음 대화형 인터프리터를 실행한다.

```
C:\Users\pahkey>cd C:\doit
C:\doit>dir
...
2014-09-23 오후 01:53 49 mod1.py
...
C:\doit>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

반드시 mod1.py를 저장한 위치로 이동한 다음 이후 예제를 진행해야 한다. 그래야만 대화형 인터프리터에서 mod1.py를 읽을 수 있다. 이제 아래와 같이 따라 해보자.

```
>>> import mod1
>>> print(mod1.sum(3,4))
7
```

mod1.py를 불러오기 위해 import mod1이라고 입력하였다. import mod1.py로 입력하는 실수를 하지 않도록 주의하자. import는 이미 만들어진 파일이나 파이썬 모듈을 사용할 수 있게 해주는 명령어이다. mod1.py 파일에 있는 sum 함수를 이용하기 위해서는 위의 예에서와 같이 mod1.sum처럼 모듈이름 뒤에 '.'(도트 연산자)를 붙이고 함수 이름을 써서 사용할 수 있다.

※ import는 현재 디렉터리에 있는 파일이나 파일이나 파이썬 라이브러리가 저장된 디렉터리에 있는 모듈만 불러올 수 있다.

import의 사용 방법은 다음과 같다.

import 모듈이름

여기서 모듈이름은 mod1.py에서 .py라는 확장자를 제거한 mod1만을 가리킨다.

이번에는 mod1.py 파일에 다음 함수를 추가해 보자.

```
def safe_sum(a, b):
    if type(a) != type(b):
        print("더할수 있는 것이 아닙니다.")
        return
    else:
        result = sum(a, b)
    return result
```

safe_sum 함수는 서로 다른 타입의 객체끼리 더하는 것을 미리 막아 준다. 만약 서로 다른 형태의 객체가 입력으로 들어오면 “더할 수 있는 값이 아닙니다”라는 메시지를 출력한다. 그리고 return 문만 단독으로 사용되어 None 값을 돌려주고 함수를 종료한다.

이 함수를 mod1.py에 추가한 다음 다시 대화형 인터프리터를 열고 다음과 같이 따라 해보자.

```
>>> import mod1
>>> print(mod1.safe_sum(3, 4))
7
```

import mod1으로 mod1.py 파일을 불러온 다음 mod1.safe_sum(3, 4)로 safe_sum 함수를 호출한다. 이렇게 하면 같은 타입의 객체가 입력으로 들어와서 3+4의 결과인 7이 출력된다.

이번에는 다음처럼 따라 해보자.

```
>>> print(mod1.safe_sum(1, 'a'))
더할 수 있는 값이 아닙니다.
None
>>>
```

위 예제에서 1은 정수형 객체, a는 문자열 객체이다. 이렇게 서로 타입이 다른 객체가 입력으로 들어오면 “더할 수 있는 값이 아닙니다.”라는 메시지를 출력하고 단독으로 사용된 return에 의해 서 None 값을 돌려주게 된다.

mod1의 sum 함수 역시 다음처럼 바로 호출할 수도 있다.

```
>>> print(mod1.sum(10, 20))
30
```

[모듈 함수를 사용하는 또 다른 방법]

때로는 mod1.sum, mod1.safe_sum처럼 쓰지 않고 그냥 sum, safe_sum처럼 함수를 쓰고 싶은 경우도 있을 것이다. 이럴 때는 “from 모듈이름 import 모듈함수”를 사용하면 된다.

from 모듈이름 import 모듈함수

from ~ import ~를 이용하면 위와 같이 모듈이름을 붙이지 않고 바로 해당 모듈의 함수를 쓸 수 있다. 다음과 같이 따라 해보자.

```
>>> from mod1 import sum
>>> sum(3, 4)
7
```

그런데 위와 같이 하면 mod1.py 파일의 sum 함수만 사용할 수 있다. sum 함수와 safe_sum 함수를 둘 다 사용하고 싶다면 어떻게 해야 할까?

2가지 방법이 있다.

```
from mod1 import sum, safe_sum
```

첫 번째 방법은 위와 같이 from 모듈이름 import 모듈함수1, 모듈함수2처럼 사용하는 방법이다. 콤마로 구분하여 필요한 함수를 불러올 수 있다.

```
from mod1 import *
```

두 번째 방법은 위와 같이 * 문자를 사용하는 방법이다. 07장에서 배울 정규 표현식에서 * 문자는 “모든것”이라는 뜻인데 파이썬에서도 마찬가지 의미로 사용된다. 따라서 `from mod1 import *`는 `mod1.py`의 모든 함수를 불러서 사용하겠다는 말이다.

`mod1.py` 파일에는 함수가 2개밖에 없기 때문에 위의 2가지 방법은 동일하게 적용된다.

if `__name__ == “__main__”`: 의 의미

이번에는 `mod1.py` 파일에 다음과 같이 추가해 보자.

```
# mod1.py
def sum(a, b):
    return a+b

def safe_sum(a, b):
    if type(a) != type(b):
        print("더할수 있는 것이 아닙니다.")
        return
    else:
        result = sum(a, b)
    return result

print(safe_sum('a', 1))
print(safe_sum(1, 4))
print(sum(10, 10.4))
```

위와 같은 `mod1.py` 파일을 에디터로 작성해서 `C:\doit`이라는 디렉터리에 저장했다면 다음처럼 실행할 수 있다.

```
C:\doit>python mod1.py
더할 수 있는 것이 아닙니다.
None
5
20.4
```

결과값은 위의 예처럼 출력될 것이다. 그런데 이 mod1.py 파일의 sum과 safe_sum 함수를 사용하기 위해 mod1.py 파일을 import하면 문제가 생긴다.

명령 프롬프트 창을 열고 다음을 따라 해보자.

```
C:\WINDOWS> cd C:\doit
C:\doit>python
>>> import mod1
더할 수 있는 것이 아닙니다.
None
5
20.4
```

엉뚱하게도 import mod1을 수행하는 순간 mod1.py가 실행이 되어 결과값을 출력한다. 우리는 단지 mod1.py 파일의 sum과 safe_sum 함수만 사용하려고 했는데 말이다. 이러한 문제를 방지하려면 다음처럼 하면 된다.

```
if __name__ == "__main__":
    print(safe_sum('a', 1))
    print(safe_sum(1, 4))
    print(sum(10, 10.4))
```

if __name__ == "__main__"을 사용하면 C:\doit>python mod1.py처럼 직접 이 파일을 실행시켰을 때는 __name__ == "__main__"이 참이 되어 if문 다음 문장들이 수행된다. 반대로 대화형 인터프리터나 다른 파일에서 이 모듈을 불러서 사용할 때는 __name__ == "__main__"이 거짓이 되어 if문 다음 문장들이 수행되지 않는다.

파이썬 모듈을 만든 다음 그 모듈을 테스트하기 위해 보통 위와 같은 방법을 사용하는데, 실제로 그런지 대화형 인터프리터를 열고 실행해 보자.

```
>>> import mod1
>>>
```

mod1.py 파일의 마지막 부분을 위와 같이 고친 다음에는 아무런 결과값도 출력되지 않는 것을 볼 수 있다.

** 알아두기 **

파이썬의 `__name__` 변수는 파이썬이 내부적으로 사용하는 특별한 변수명이다. 만약 C:\doit>python mod1.py처럼 직접 mod1.py 파일을 실행시킬 경우 mod1.py의 `__name__` 변수에는 `__main__`이라는 값이 저장된다. 하지만 파이썬 셸이나 다른 파이썬 모듈에서 mod1을 import 할 경우에는 mod1.py의 `__name__` 변수에는 “mod1”이라는 mod1.py의 모듈이름 값이 저장된다.

클래스나 변수 등을 포함한 모듈

지금까지 살펴본 모듈은 함수만 포함했지만 클래스나 변수 등을 포함할 수도 있다. 다음의 프로그램을 작성해 보자.

```
# mod2.py
PI = 3.141592

class Math:
    def solv(self, r):
        return PI * (r ** 2)

    def sum(a, b):
        return a+b

if __name__ == "__main__":
    print(PI)
    a = Math()
    print(a.solv(2))
    print(sum(PI , 4.4))
```

이 파일은 원의 넓이를 계산하는 Math 클래스와 두 값을 더하는 sum 함수 그리고 원주율 값에 해당되는 PI 변수처럼 클래스, 함수, 변수 등을 모두 포함하고 있다. 파일 이름을 mod2.py로 하고 C:\doit 디렉터리에 저장하자. mod2.py 파일은 다음과 같이 실행할 수 있다.

```
C:\doit>python mod2.py
3.141592
12.566368
7.541592
```

이번에는 대화형 인터프리터를 열고 다음과 같이 따라 해보자.

```
C:\doit>python
>>> import mod2
>>>
```

`__name__ == "__main__"`이 거짓이 되므로 아무런 값도 출력되지 않는다.

모듈에 포함된 변수, 클래스, 함수 사용하기

```
>>> print(mod2.PI)
3.141592
```

위의 예에서 볼 수 있듯이 `mod2.PI`처럼 입력해서 `mod2.py` 파일에 있는 PI라는 변수값을 사용할 수 있다.

```
>>> a = mod2.Math()
>>> print(a.solv(2))
12.566368
```

위의 예는 `mod2.py`에 있는 `Math` 클래스를 사용하는 방법을 보여 준다. 위의 예처럼 모듈 내에 있는 클래스를 이용하려면 '.'(도트 연산자)를 이용하여 클래스 이름 앞에 모듈 이름을 먼저 입력해야 한다.

```
>>> print(mod2.sum(mod2.PI, 4.4))
7.541592
```

`mod2.py`에 있는 `sum` 함수 역시 당연히 사용할 수 있다.

새 파일 안에서 이전에 만든 모듈 불러오기

지금까지는 만들어 놓은 모듈 파일을 사용하기 위해 대화형 인터프리터만을 이용했다. 이번에는 새롭게 만들 파일 안에 이전에 만들어 놓았던 모듈을 불러와서 사용하는 방법에 대해 알아보자.

방금 전에 만든 모듈인 mod2.py 파일을 새롭게 만들 파일 프로그램 파일에서 불러와 사용해 보자. 그럼 에디터로 다음과 같이 작성해 보자.

```
# modtest.py
import mod2
result = mod2.sum(3, 4)
print(result)
```

위에서 볼 수 있듯이 파일에서도 import mod2로 mod2 모듈을 불러와서 사용하면 된다. 대화형 인터프리터에서 한 것과 마찬가지 방법이다. 위의 예제가 정상적으로 실행되기 위해서는 modtest.py 파일과 mod2.py 파일이 동일한 디렉터리에 있어야 한다.

[모듈을 불러오는 또 다른 방법]

우리는 지금껏 명령 프롬프트 창을 열고 모듈이 있는 디렉터리로 이동한 다음에나 모듈을 사용할 수 있었다. 이번에는 모듈을 저장한 디렉터리로 이동하지 않고 모듈을 불러와서 사용하는 방법에 대해서 알아보자.

우선 이전에 만든 mod2.py 모듈을 C:\doit\mymod라는 디렉터리를 새로 생성해서 저장한 후 다음의 예를 따라 해보자.

1. sys.path.append(모듈을 저장한 디렉터리) 사용하기

먼저 sys 모듈을 불러온다.

```
>>> import sys
```

sys 모듈은 파이썬을 설치할 때 함께 설치되는 라이브러리 모듈이다. sys에 대해서는 뒤에서 다시 다룰 것이다. 이 sys 모듈을 이용해서 파이썬 라이브러리가 설치되어 있는 디렉터리를 확인할 수 있다.

다음과 같이 작성해 보자.

```
>>> sys.path
[‘’, ‘C:\Windows\SYSTEM32\python36.zip’, ‘c:\Python36\DLLs’,
‘c:\Python36\lib’, ‘c:\Python36’, ‘c:\Python36\lib\site-packages’]
```

sys.path는 파일 라이브러리들이 설치되어 있는 디렉터리들을 보여 준다. 만약 파일 모듈이 위의 디렉터리에 들어 있다면 모듈이 저장된 디렉터리로 이동할 필요없이 바로 불러서 사용할 수가 있다. 그렇다면 sys.path에 C:\doit\mymod라는 디렉터리를 추가하면 아무데서나 불러 사용할 수 있지 않을까?

※ 명령 프롬프트 창에서는 /, \는 상관없지만, 소스코드 안에서는 반드시 / 또는 \\ 기호를 사용해야 한다.

당연하다. sys.path의 결과값이 리스트이므로 우리는 다음과 같이 할 수 있을 것이다.

```
>>> sys.path.append("C:/doit/mymod")
>>> sys.path
[‘’, ‘C:\Windows\SYSTEM32\python36.zip’, ‘c:\Python36\DLLs’,
‘c:\Python36\lib’, ‘c:\Python36’, ‘c:\Python36\lib\site-packages’,
‘C:/doit/mymod’]
>>>
```

sys.path.append를 이용해서 C:/doit/mymod라는 디렉터리를 sys.path에 추가한 후 다시 sys.path를 보면 가장 마지막 요소에 C:/doit/mymod라고 추가된 것을 확인할 수 있다.

자, 실제로 모듈을 불러와서 사용할 수 있는지 확인해 보자.

```
>>> import mod2
>>> print(mod2.sum(3,4))
7
```

이상 없이 불러와서 사용할 수 있다. 이렇게 특정한 디렉터리에 있는 모듈을 불러와서 사용하고 싶을 때 사용할 수 있는 것이 바로 sys.path.append(모듈을 저장한 디렉터리)이다.

2. PYTHONPATH 환경 변수 사용하기

모듈을 불러와서 사용하는 또 다른 방법으로는 PYTHONPATH 환경 변수를 사용하는 방법이 있다.

다음과 같이 따라 해보자.

```
C:\Users\home>set PYTHONPATH=C:\doit\mymod
C:\Users\home>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type "help", "copyright", "credits" or "license" for more information.
>>> import mod2
>>> print(mod2.sum(3,4))
7
```

set 명령어를 이용해 PYTHONPATH 환경 변수에 mod2.py 파일이 있는 C:\doit\mymod 디렉터리를 설정한다. 그러면 디렉터리 이동이나 별도의 모듈 추가 작업 없이 mod2 모듈을 불러와서 사용할 수 있다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#05-2>)

[문제1] 모듈 사용하기 1

c:\doit이라는 디렉토리에 mymod.py라는 파일이 있다고 가정하고 파이썬 shell에서 이 모듈을 import 해서 사용할 수 있는 방법에 대해서 모두 기술하시오.

즉, 다음과 같이 import mymod 수행 시 오류가 없어야 한다.

```
>>> import mymod
>>>
```

[문제2] 모듈 작성

다음이 가능하도록 모듈 mymod.py를 작성하시오. (mymod.py는 c:\doit\mymod.py로 저장할것)

```
>>> import sys
>>> sys.path.append("c:/doit")
>>> from mymod import mysum
>>> mysum(1, 2)
3
```

[문제3] 모듈 사용하기 2

다음은 c:\doit\mymod.py 라는 모듈의 내용이다.

```
def mysum(a, b):
    return a+b

# test
print(mysum(3, 7)) # 10을 출력
```

이 모듈을 다음과 같이 파이썬 shell에서 import 하여 사용하였다.

```
>>> import sys
>>> sys.path.append("c:/doit")
>>> import mymod
10
```

import mymod 라고 입력하자마자 10이라는 숫자가 출력되었다. 그 이유는 무엇일까? 그리고 이렇게 10이라는 숫자가 출력되지 않도록 하기 위해서는 어떻게 해야 할까?

05-3 패키지

패키지란 무엇인가?

패키지(Packages)는 도트(.)를 이용하여 파이썬 모듈을 계층적(디렉터리 구조)으로 관리할 수 있게 해준다. 예를 들어 모듈명이 A.B인 경우 A는 패키지명이 되고 B는 A 패키지의 B 모듈이 된다.

파이썬 패키지는 디렉터리와 파이썬 모듈로 이루어지며 구조는 다음과 같다.

가상의 *game* 패키지 예

```
game/
    __init__.py
    sound/
        __init__.py
        echo.py
        wav.py
    graphic/
        __init__.py
        screen.py
        render.py
    play/
        __init__.py
        run.py
        test.py
```

game, *sound*, *graphic*, *play*는 디렉터리명이고 .py 확장자를 가지는 파일은 파이썬 모듈이다. *game* 디렉터리가 이 패키지의 루트 디렉터리이고 *sound*, *graphic*, *play*는 서브 디렉터리이다.

※ *__init__.py* 파일은 조금 특이한 용도로 사용되는데, 이것에 대해서는 뒤에서 자세하게 다룰 것이다.

간단한 파이썬 프로그램이 아니라면 이렇게 패키지 구조로 파이썬 프로그램을 만드는 것이 공동 작업이나 유지 보수 등 여러 면에서 유리하다. 또한 패키지 구조로 모듈을 만들면 다른 모듈과 이름이 겹치더라도 더 안전하게 사용할 수 있다.

패키지 만들기

이제 위의 예와 비슷한 *game* 패키지를 직접 만들어 보면 패키지에 대해서 알아보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

패키지 기본 구성 요소 준비하기

1. C:/doit이라는 디렉터리 밑에 game 및 기타 서브 디렉터리들을 생성하고 .py 파일들을 다음과 같이 만들어 보자(만약 C:/doit이라는 디렉터리가 없다면 먼저 생성하고 진행하자).

```
C:/doit/game/__init__.py
C:/doit/game/sound/__init__.py
C:/doit/game/sound/echo.py
C:/doit/game/graphic/__init__.py
C:/doit/game/graphic/render.py
```

2. 각 디렉터리에 __init__.py 파일을 만들어 놓기만 하고 내용은 일단 비워 둔다.

3. echo.py 파일은 다음과 같이 만든다.

```
# echo.py
def echo_test():
    print ("echo")
```

4. render.py 파일은 다음과 같이 만든다.

```
# render.py
def render_test():
    print ("render")
```

5. 다음 예제들을 수행하기 전에 우리가 만든 game 패키지를 참조할 수 있도록 다음과 같이 명령 프롬프트 창에서 set 명령을 이용하여 PYTHONPATH 환경 변수에 C:/doit 디렉터리를 추가한다. 그리고 파이썬 인터프리터(Interactive shell)를 실행하자.

```
C:\> set PYTHONPATH=C:/doit
C:\> python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

여기까지 준비가 되었다면 다음을 따라 해보자.

패키지 안의 함수 실행하기

자, 이제 패키지를 이용하여 echo.py 파일의 echo_test 함수를 실행해 보자. 패키지 안의 함수를 실행하는 방법은 다음과 같이 3가지가 있다.

※ 아래 예제들은 import 예제들이므로 하나의 예제를 실행하고 나서 다음 예제를 실행할 때에는 반드시 인터프리터를 종료하고 다시 실행해야 한다. 인터프리터를 다시 시작하지 않을 경우 이전에 import했던 것들이 메모리에 남아 있게 되어 영뚱한 결과가 나올 수 있다.

첫 번째는 echo 모듈을 import하여 실행하는 방법으로, 다음과 같이 실행한다.

```
>>> import game.sound.echo
>>> game.sound.echo.echo_test()
echo
```

두 번째는 echo 모듈이 있는 디렉터리까지를 from ... import하여 실행하는 방법이다.

```
>>> from game.sound import echo
>>> echo.echo_test()
echo
```

세 번째는 echo 모듈의 echo_test 함수를 직접 import하여 실행하는 방법이다.

```
>>> from game.sound.echo import echo_test
>>> echo_test()
echo
```

하지만 다음과 같이 echo_test 함수를 사용하는 것은 불가능하다.

```
>>> import game
>>> game.sound.echo.echo_test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'sound'
```

import game을 수행하면 game 디렉터리의 모듈 또는 game 디렉터리의 __init__.py에 정의된 것들만 참조할 수 있다.

또 다음처럼 echo_test 함수를 사용하는 것도 불가능하다.

```
>>> import game.sound.echo_test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named echo_test
```

도트 연산자(.)를 사용해서 import a.b.c처럼 import할 때 가장 마지막 항목인 c는 반드시 모듈 또는 패키지여야만 한다.

`__init__.py` 의 용도

`__init__.py` 파일은 해당 디렉터리가 패키지의 일부임을 알려주는 역할을 한다. 만약 game, sound, graphic 등 패키지에 포함된 디렉터리에 `__init__.py` 파일이 없다면 패키지로 인식되지 않는다.

※ python3.3 버전부터는 `__init__.py` 파일 없이도 패키지로 인식이 된다([PEP 420](#)).
하지만 하위 버전 호환을 위해 `__init__.py` 파일을 생성하는 것이 안전한 방법이다.

시험 삼아 sound 디렉터리의 `__init__.py`를 제거하고 다음을 수행해 보자.

```
>>> import game.sound.echo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named sound.echo
```

sound 디렉터리에 `__init__.py` 파일이 없어서 임포트 오류(ImportError)가 발생하게 된다.

`__all__`의 용도

다음을 따라 해보자.

```
>>> from game.sound import *
>>> echo.echo_test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'echo' is not defined
```

그런데 뭔가 이상하지 않은가? 분명 game.sound 패키지에서 모든 것(*)을 import하였으므로 echo 모듈을 사용할 수 있어야 할 것 같은데 echo라는 이름이 정의되지 않았다는 이름 오류(NameError)가 발생했다.

이렇게 특정 디렉터리의 모듈을 *를 이용하여 import할 때에는 다음과 같이 해당 디렉터리의 __init__.py 파일에 __all__이라는 변수를 설정하고 import할 수 있는 모듈을 정의해 주어야 한다.

```
# C:/doit/game/sound/__init__.py
__all__ = ['echo']
```

여기서 __all__이 의미하는 것은 sound 디렉터리에서 * 기호를 이용하여 import할 경우 이곳에 정의된 echo 모듈만 import된다는 의미이다.

※ 착각하기 쉬운데 from game.sound.echo import * 는 __all__과 상관없이 무조건 import된다. 이렇게 __all__과 상관없이 무조건 import되는 경우는 from a.b.c import *에서 from의 마지막 항목인 c가 모듈인 경우이다.

위와 같이 __init__.py 파일을 변경한 후 위 예제를 수행하면 원하던 결과가 출력되는 것을 확인 할 수 있다.

```
>>> from game.sound import *
>>> echo.echo_test()
echo
```

relative 패키지

만약 graphic 디렉터리의 render.py 모듈이 sound 디렉터리의 echo.py 모듈을 사용하고 싶다면 어떻게 해야 할까? 다음과 같이 render.py를 수정하면 가능하다.

```
# render.py
from game.sound.echo import echo_test
def render_test():
    print ("render")
    echo_test()
```

from game.sound.echo import echo_test라는 문장을 추가하여 echo_test() 함수를 사용할 수 있도록 수정했다.

이렇게 수정한 후 다음과 같이 수행해 보자.

```
>>> from game.graphic.render import render_test
>>> render_test()
render
echo
```

이상 없이 잘 수행된다.

위 예제처럼 from game.sound.echo import echo_test와 같이 전체 경로를 이용하여 import할 수도 있지만 다음과 같이 relative하게 import하는 것도 가능하다.

※ 이 기능은 Python 2.5부터 지원되기 시작하였다.

```
# render.py
from ..sound.echo import echo_test

def render_test():
    print ("render")
    echo_test()
```

from game.sound.echo import echo_test가 from ..sound.echo import echo_test로 변경되었다. 여기서 ..은 부모 디렉터리를 의미한다. graphic과 sound 디렉터리는 동일한 깊이(depth)이므로 부모 디렉터리(..)를 이용하여 위와 같은 import가 가능한 것이다.

relative한 접근자에는 다음과 같은 것들이 있다.

- .. – 부모 디렉터리
- . – 현재 디렉터리

..과 같은 relative한 접근자는 render.py와 같이 모듈 안에서만 사용해야 한다. 파이썬 인터프리터에서 relative한 접근자를 사용하면 “SystemError: cannot perform relative import”와 같은 오류가 발생한다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#05-3>)

[문제1] 패키지 사용 1

다음은 game 패키지의 구조이다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```

game/
__init__.py
sound/
__init__.py
echo.py
wav.py
graphic/
__init__.py
screen.py
render.py
play/
__init__.py
run.py
test.py

```

game 패키지의 echo모듈에 있는 echo_test라는 함수를 이용하려고 한다. 다음 중 잘못된 것 하나를 고르고 그 이유를 설명하시오.

(단, 위 패키지 내의 __init__.py 는 모두 빈 파일이다.)

1)

```

>>> from game.sound import echo
>>> echo.echo_test()

```

2)

```

>>> from game.sound.echo import echo_test
>>> echo_test()

```

3)

```

>>> import game
>>> game.sound.echo.echo_test()

```

4)

```
>>> import game.sound.echo  
>>> game.sound.echo.echo_test()
```

[문제2] 패키지 사용2

[문제1의 패키지 구조 참조]

다음은 game/graphic/screen.py 의 내용이다.

```
# import 가 필요하다.  
echo_test()
```

위와 같이 screen.py 파일에서 game/sound/echo.py 모듈의 echo_test라는 함수를 호출하려고 한다. echo_test함수를 사용하기 위해서는 적절한 import 가 필요하다. screen.py 에서 echo_test를 호출할 수 있도록 relative 패키지를 이용하는 import 문을 삽입하여 screen.py를 완성하시오.

05-4 예외 처리

프로그램을 만들다 보면 수없이 많은 오류를 만나게 된다. 물론 오류가 발생하는 이유는 프로그램이 잘못 동작되는 것을 막기 위한 파이썬의 배려이다. 하지만 때때로 이러한 오류를 무시하고 싶을 때도 있고 별도로 처리하고 싶을 때도 있다. 이에 파이썬은 try, except를 이용해서 오류를 처리할 수 있게 해준다.

오류는 어떤 때 발생하는가?

오류를 처리하는 방법을 알기 전에 어떤 상황에서 오류가 발생하는지 한번 알아보자. 오타를 쳤을 때 발생하는 구문 오류 같은 것이 아닌 실제 프로그램에서 자주 발생하는 오류를 중심으로 살펴본다.

먼저 디렉터리 안에 없는 파일을 열려고 시도했을 때 발생하는 오류이다.

```
>>> f = open("나없는파일", 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '나없는파일'
```

위의 예에서 볼 수 있듯이 없는 파일을 열려고 시도하면 “FileNotFoundError”라는 이름의 오류가 발생하게 된다.

※ python 2.7 버전에서는 “FileNotFoundError”가 아닌 “IOError”라는 이름의 오류가 발생한다.

이번에는 0으로 다른 숫자를 나누는 경우를 생각해 보자.

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

4를 0으로 나누려니까 “ZeroDivisionError”라는 이름의 오류가 발생한다.

마지막으로 한 가지 예만 더 들어 보자. 다음 오류는 정말 빈번하게 일어난다.

```
>>> a = [1,2,3]
>>> a[4]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

a는 [1, 2, 3]이라는 리스트인데 a[4]는 a 리스트에서 얻을 수 없는 값이다. 따라서 “IndexError”가 발생하게 된다. 파일은 이런 오류가 발생하면 프로그램을 중단하고 오류메시지를 보여 준다.

오류 예외 처리 기법

자, 이제 유연한 프로그래밍을 위한 오류 처리 기법에 대해서 살펴보자.

try, except문

다음은 오류 처리를 위한 try, except문의 기본 구조이다.

```
try:
    ...
except [발생 오류[as 오류 메시지 변수]]:
    ...
```

try 블록 수행 중 오류가 발생하면 except 블록이 수행된다. 하지만 try블록에서 오류가 발생하지 않는다면 except 블록은 수행되지 않는다.

except 구문을 자세히 살펴보자.

```
except [발생 오류 [as 오류 메시지 변수]]:
```

위 구문을 보면 [] 기호를 사용하는데, 이 기호는 괄호 안의 내용을 생략할 수 있다는 관례적인 표기법이다. 즉, except 구문은 다음처럼 3가지 방법으로 사용할 수 있다.

1. try, except만 쓰는 방법

```
try:
    ...
except:
    ...
```

이 경우는 오류 종류에 상관없이 오류가 발생하기만 하면 except 블록을 수행한다.

2. 발생 오류만 포함한 except문

```
try:  
    ...  
except 발생 오류:  
    ...
```

이 경우는 오류가 발생했을 때 except문에 미리 정해 놓은 오류 이름과 일치할 때만 except 블록을 수행한다는 뜻이다.

3. 발생 오류와 오류 메시지 변수까지 포함한 except문

```
try:  
    ...  
except 발생 오류 as 오류 메시지 변수:  
    ...
```

이 경우는 두 번째 경우에서 오류 메시지의 내용까지 알고 싶을 때 사용하는 방법이다.

이 방법의 예를 들어 보면 다음과 같다.

```
try:  
    4 / 0  
except ZeroDivisionError as e:  
    print(e)
```

※ 파이썬 2.7 버전의 경우에는 위 예제의 except ZeroDivisionError as e: 대신 except ZeroDivisionError, e:와 같이 사용해야 한다.

위처럼 4를 0으로 나누려고 하면 ZeroDivisionError가 발생하여 except 블록이 실행되고 e라는 오류 메시지를 다음과 같이 출력한다.

결과값: division by zero

try .. else

try문은 else절을 지원한다. else절은 예외가 발생하지 않은 경우에 실행되며 반드시 except절 바로 다음에 위치해야 한다.

※ else절은 else 블록과 같은 뜻이다.

다음 예를 보자.

```
try:
    f = open('foo.txt', 'r')
except FileNotFoundError as e:
    print(str(e))
else:
    data = f.read()
    f.close()
```

만약 foo.txt라는 파일이 없다면 except절이 수행되고 foo.txt 파일이 있다면 else절이 수행될 것이다.

try .. finally

try문에는 finally절을 사용할 수 있다. finally절은 try문 수행 도중 예외 발생 여부에 상관없이 항상 수행된다. 보통 finally절은 사용한 리소스를 close해야 할 경우에 많이 사용된다.

다음의 예를 보자.

```
f = open('foo.txt', 'w')
try:
    # 무언가를 수행한다.
finally:
    f.close()
```

foo.txt라는 파일을 쓰기 모드로 연 후에 try문이 수행된 후 예외 발생 여부에 상관없이 finally 절에서 f.close()로 열린 파일을 닫을 수 있다.

여러개의 오류처리하기

try문 내에서 여러개의 오류를 처리하기 위해서는 다음과 같은 구문을 이용한다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```

try:
    ...
except 발생 오류1:
    ...
except 발생 오류2:
    ...

```

즉, 다음과 같이 0으로 나누는 오류와 인덱싱 오류를 다음과 같이 처리할 수 있다.

```

try:
    a = [1,2]
    print(a[3])
    4/0
except ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
except IndexError:
    print("인덱싱 할 수 없습니다.")

```

a는 2개의 요소값을 가지고 있기 때문에 a[3]는 IndexError를 발생시키므로 “인덱싱 할 수 없습니다.”라는 문자열이 출력될 것이다. 인덱싱 오류가 먼저 발생했으므로 4/0으로 발생되는 ZeroDivisionError는 발생하지 않았다.

이전에 알아보았던 것과 마찬가지로 오류메시지도 다음과 같이 가져올 수 있다.

```

try:
    a = [1,2]
    print(a[3])
    4/0
except ZeroDivisionError as e:
    print(e)
except IndexError as e:
    print(e)

```

실행하면 “list index out of range”라는 오류 메시지가 출력될 것이다.

다음과 같이 ZerroDivisionError와 IndexError를 함께 처리할 수도 있다.

```

try:
    a = [1,2]
    print(a[3])
    4/0
except (ZeroDivisionError, IndexError) as e:
    print(e)

```

2개 이상의 오류를 동시에 처리하기 위해서는 위와같이 괄호를 이용하여 함께 묶어주어 처리하면 된다.

오류 회피하기

프로그래밍을 하다 보면 특정 오류가 발생할 경우 그냥 통과시켜야 할 때가 있을 수 있다. 다음의 예를 보자.

```

try:
    f = open("나없는파일", 'r')
except FileNotFoundError:
    pass

```

try문 내에서 FileNotFoundError가 발생할 경우 pass를 사용하여 오류를 그냥 회피하도록 한 예제이다.

오류 일부러 발생시키기

이상하게 들리겠지만 프로그래밍을 하다 보면 종종 오류를 일부러 발생시켜야 할 경우도 생긴다. 파이썬은 raise라는 명령어를 이용해 오류를 강제로 발생시킬 수 있다.

예를 들어 Bird라는 클래스를 상속받는 자식 클래스는 반드시 fly라는 함수를 구현하도록 만들고 싶은 경우(강제로 그렇게 하고 싶은 경우)가 있을 수 있다. 다음 예를 보자.

```

class Bird:
    def fly(self):
        raise NotImplementedError

```

위 예제는 Bird 클래스를 상속받는 자식 클래스는 반드시 fly라는 함수를 구현해야 한다는 의지를 보여준다. 만약 자식 클래스가 fly 함수를 구현하지 않은 상태로 fly 함수를 호출한다면 어떻게 될까?

※ NotImplementedError는 파이썬 내장 오류로, 꼭 작성해야 하는 부분이 구현되지 않았을 경우 일부러 오류를 발생시키고자 사용한다.

```
class Eagle(Bird):
    pass

eagle = Eagle()
eagle.fly()
```

Eagle 클래스는 Bird 클래스를 상속받는다. 그런데 Eagle 클래스에서 fly 함수를 구현하지 않았기 때문에 Bird 클래스의 fly 함수가 호출된다. 그리고 raise문에 의해 다음과 같은 NotImplementedError가 발생할 것이다.

※ 상속받는 클래스에서 함수를 재구현하는 것을 메서드 오버라이딩이라고 부른다.

```
Traceback (most recent call last):
  File "...", line 33, in <module>
    eagle.fly()
  File "...", line 26, in fly
    raise NotImplementedError
NotImplementedError
```

NotImplementedError가 발생되지 않게 하려면 다음과 같이 Eagle 클래스에 fly 함수를 반드시 구현해야 한다.

```
class Eagle(Bird):
    def fly(self):
        print("very fast")

eagle = Eagle()
eagle.fly()
```

위 예처럼 fly 함수를 구현한 후 프로그램을 실행하면 오류 없이 다음과 같은 문장이 출력된다.

```
very fast
```

예외 만들기

프로그램 수행 도중 특수한 경우에만 예외처리를 하기 위해서 종종 예외를 만들어서 사용하게 된다.

직접 예외를 만들어 보자. 예외는 다음과 같이 파이썬 내장 클래스인 `Exception`클래스를 상속하여 만들 수 있다.

```
class MyError(Exception):
    pass
```

그리고 별명을 출력해 주는 함수를 다음과 같이 작성해 보자.

```
def say_nick(nick):
    if nick == '바보':
        raise MyError()
    print(nick)
```

그리고 다음과 같이 `say_nick` 함수를 호출 해 보자.

```
say_nick("천사")
say_nick("바보")
```

실행 해 보면 다음과 같이 “천사”가 한번 출력된 후 `MyError`가 발생하는 것을 알 수 있다.

```
천사
Traceback (most recent call last):
  File "...", line 11, in <module>
    say_nick("바보")
  File "...", line 7, in say_nick
    raise MyError()
__main__.MyError
```

이번에는 `MyError`가 발생할 경우 예외처리기법을 이용하여 예외처리를 해 보도록 하자.

```

try:
    say_nick("천사")
    say_nick("바보")
except MyError:
    print("허용되지 않는 별명입니다.")

```

실행 하면 다음과 같이 출력된다.

```

천사
허용되지 않는 별명입니다.

```

만약 오류메시지를 이용하고 싶다면 다음처럼 예외처리를 해야 할 것이다.

```

try:
    say_nick("천사")
    say_nick("바보")
except MyError as e:
    print(e)

```

하지만 실행 해 보면 `print(e)`로 출력한 오류메시지가 아무것도 출력되지 않는 것을 확인 할 수 있다. 오류 메시지를 출력했을 때 오류 메시지가 보이게 하기 위해서는 오류 클래스에 다음과 같은 `__str__` 메써드를 구현해야 한다. `__str__` 메써드는 `print(e)` 처럼 오류메시지를 `print`문으로 출력할 경우에 호출되는 메써드이다.

```

class MyError(Exception):
    def __str__(self):
        return "허용되지 않는 별명입니다."

```

다시 실행해 보면 “허용되지 않는 별명입니다.”라는 오류메시지가 출력되는 것을 확인할 수 있다. 만약 예외 발생시점에 오류메시지를 전달하고 싶다면 다음과 같이 수정해야 한다.

```

class MyError(Exception):
    def __init__(self, msg):
        self.msg = msg

    def __str__(self):
        return self.msg


def say_nick(nick):
    if nick == '바보':
        raise MyError("허용되지 않는 별명입니다.")
    print(nick)

try:
    say_nick("천사")
    say_nick("바보")
except MyError as e:
    print(e)

```

`raise MyError("허용되지 않는 별명입니다.")`처럼 오류 발생시점에 메시지를 전달할 수 있다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#05-4>)

[문제1] 예외처리 1

다음 2가지 예제는 모두 예외가 발생한다. 예외의 원인은 무엇인지 설명하시오. 그리고 이러한 예외가 발생하지 않기 위해 코드를 수정하시오.

- 1) “a”라는 문자열과 1이라는 숫자를 더하여 “a1”이라는 문자열을 만들려고 다음과 같이 호출하였다.

```
>>> "a" + 1
```

- 2) a는 [1, 2, 3]이라는 리스트이다. a의 3번째 값을 얻으려고 다음과 같이 호출하였다.

```
>>> a = [1, 2, 3]
>>> a[3]
```

[문제2] 예외처리 2

다음 코드를 실행했을때의 결과를 예측하고 그 이유에 대해서 설명하시오.

```
a = [1, 2, 3]

try:
    result = a[-3]
except:
    print("error")
else:
    print("no error")
```

[문제3] 예외처리 3

다음 코드의 실행결과를 예측하고 그 이유에 대해서 설명하시오.

```
result = 3

try:
    result += 1
except:
    result += 2
else:
    result += 3
finally:
    result += 4

print(result)
```

[문제4] 예외처리 4

다음 코드의 실행결과를 예측하고 그 이유에 대해서 설명하시오.

```
result = 0
try:
    [1, 2, 3][3]
    "a"+1
    4 / 0
except TypeError:
    result += 1
except ZeroDivisionError:
    result += 2
except IndexError:
    result += 3
else:
    result += 4
finally:
    result += 5

print(result)
```

05-5 내장 함수

지금까지 파이썬으로 프로그래밍하기 위해 알아야 하는 대부분의 것들을 공부했다. 이제 여러분은 자신이 원하는 프로그램을 직접 만들 수 있을 것이다. 하지만 그 전에 먼저 여러분이 만들고자 하는 프로그램이 이미 만들어져 있는지 살펴보기 바란다. 물론 공부를 위해서라면 누군가 만들어 놓은 프로그램을 또 만들 수도 있다. 하지만 그런 목적이 아니라면 이미 만들어진 것을 다시 만드는 것은 불필요한 행동이다(Don't Reinvent The Wheel, 이미 있는 것을 다시 만드느라 쓸데없이 시간을 낭비하지 말라). 그리고 이미 만들어진 프로그램들은 테스트 과정을 수도 없이 거쳤기 때문에 충분히 검증되어 있다. 따라서 무엇인가 새로운 프로그램을 만들기 전에는 이미 만들어진 것들, 그중에서도 특히 파이썬 배포본에 함께 들어 있는 파이썬 라이브러리들을 살펴보는 것이 매우 중요하다.

라이브러리들을 살펴보기 전에 파이썬 내장 함수를 먼저 살펴보도록 하자. 우리는 이미 몇 가지 내장 함수들을 배웠다. print, del, type 등이 바로 그것이다. 이러한 파이썬 내장 함수들은 외부 모듈과는 달리 import를 필요로 하지 않는다. 아무런 설정 없이 바로 사용할 수가 있다. 이 책에서는 활용 빈도가 높고 중요한 함수들을 중심으로 알파벳 순서대로 간략히 정리했다. 파이썬으로 프로그래밍을 하기 위해 이 함수들을 지금 당장 모두 알아야 하는 것은 아니므로 가벼운 마음으로 천천히 살펴보자.

abs

abs(x)는 어떤 숫자를 입력으로 받았을 때, 그 숫자의 절대값을 돌려주는 함수이다.

```
>>> abs(3)
3
>>> abs(-3)
3
>>> abs(-1.2)
1.2
```

all

all(x)은 반복 가능한(iterable) 자료형 x를 입력 인수로 받으며, 이 x가 모두 참이면 True, 거짓이 하나라도 있으면 False를 리턴한다.

※ 반복 가능한 자료형이란 for문으로 그 값을 출력할 수 있는 것을 의미한다. 리스트, 튜플, 문자열, 딕셔너리, 집합 등이 있다.

다음의 예를 보자.

```
>>> all([1, 2, 3])
True
```

리스트 자료형 [1, 2, 3]은 모든 요소가 참이므로 True를 리턴한다.

```
>>> all([1, 2, 3, 0])
False
```

리스트 자료형 [1, 2, 3, 0] 중에서 요소 0은 거짓이므로 False를 리턴한다.

※ 자료형의 참과 거짓에 대해 잘 기억나지 않는다면 02-7절을 다시 한 번 읽어 보자.

any

any(x)는 x 중 하나라도 참이 있을 경우 True를 리턴하고, x가 모두 거짓일 경우에만 False를 리턴한다. all(x)의 반대 경우라고 할 수 있다.

다음의 예를 보자.

```
>>> any([1, 2, 3, 0])
True
```

리스트 자료형 [1, 2, 3, 0] 중에서 1, 2, 3이 참이므로 True를 리턴한다.

```
>>> any([0, ""])
False
```

리스트 자료형 [0, ""]의 요소 0과 “”은 모두 거짓이므로 False를 리턴한다.

chr

chr(i)는 아스키(ASCII) 코드값을 입력으로 받아 그 코드에 해당하는 문자를 출력하는 함수이다.

※ 아스키 코드란 0에서 127 사이의 숫자들을 각각 하나의 문자 또는 기호에 대응시켜 놓은 것이다.

```
>>> chr(97)
'a'
>>> chr(48)
'0'
```

dir

dir은 객체가 자체적으로 가지고 있는 변수나 함수를 보여 준다. 아래 예는 리스트와 딕셔너리 객체의 관련 함수들(메서드)을 보여 주는 예이다. 우리가 02장에서 살펴보았던 자료형 관련 함수들을 만나볼 수 있을 것이다.

```
>>> dir([1, 2, 3])
['append', 'count', 'extend', 'index', 'insert', 'pop', ...]
>>> dir({'1':'a'})
['clear', 'copy', 'get', 'has_key', 'items', 'keys', ...]
```

divmod

divmod(a, b)는 2개의 숫자를 입력으로 받는다. 그리고 a를 b로 나눈 몫과 나머지를 튜플 형태로 리턴하는 함수이다.

```
>>> divmod(7, 3)
(2, 1)
>>> divmod(1.3, 0.2)
(6.0, 0.09999999999999978)
```

enumerate

enumerate는 “열거하다”라는 뜻이다. 이 함수는 순서가 있는 자료형(리스트, 튜플, 문자열)을 입력으로 받아 인덱스 값을 포함하는 enumerate 객체를 리턴한다.

※ 보통 enumerate 함수는 아래 예제처럼 for문과 함께 자주 사용된다.

무슨 말인지 잘 이해되지 않으면 다음 예를 보자.

```
>>> for i, name in enumerate(['body', 'foo', 'bar']):
...     print(i, name)
...
0 body
1 foo
2 bar
```

순서값과 함께 body, foo, bar가 순서대로 출력되었다. 즉, 위 예제와 같이 enumerate를 for문과 함께 사용하면 자료형의 현재 순서(index)와 그 값을 쉽게 알 수 있다.

for문처럼 반복되는 구간에서 객체가 현재 어느 위치에 있는지 알려주는 인덱스 값이 필요할 때 enumerate 함수를 사용하면 매우 유용하다.

eval

eval(expression)은 실행 가능한 문자열(1+2, 'hi' + 'a' 같은 것)을 입력으로 받아 문자열을 실행한 결과값을 리턴하는 함수이다.

```
>>> eval('1+2')
3
>>> eval("'hi' + 'a'")
'hia'
>>> eval('divmod(4, 3)')
(1, 1)
```

보통 eval은 입력받은 문자열로 파이썬 함수나 클래스를 동적으로 실행하고 싶은 경우에 사용된다.

filter

filter란 무엇인가를 걸러낸다는 뜻으로, filter 함수도 동일한 의미를 가진다. filter 함수는 첫 번째 인수로 함수 이름을, 두 번째 인수로 그 함수에 차례로 들어갈 반복 가능한 자료형을 받는다. 그리고 두 번째 인수인 반복 가능한 자료형 요소들이 첫 번째 인수인 함수에 입력되었을 때 리턴값이 참인 것만 묶어서(걸러내서) 돌려준다.

다음의 예를 보자.

```
#positive.py
def positive(l):
    result = []
    for i in l:
        if i > 0:
            result.append(i)
    return result

print(positive([1,-3,2,0,-5,6]))
```

결과값: [1, 2, 6]

즉, 위에서 만든 positive 함수는 리스트를 입력값으로 받아 각각의 요소를 판별해서 양수값만 리턴하는 함수이다.

filter 함수를 이용하면 위의 내용을 아래와 같이 간단하게 작성할 수 있다.

```
#filter1.py
def positive(x):
    return x > 0

print(list(filter(positive, [1, -3, 2, 0, -5, 6])))
```

결과값: [1, 2, 6]

여기서는 두 번째 인수인 리스트의 요소들이 첫 번째 인수인 positive 함수에 입력되었을 때 리턴 값이 참인 것만 끌어서 돌려준다. 앞의 예에서는 1, 2, 6만 양수여서 $x > 0$ 이라는 문장이 참이 되므로 [1, 2, 6]이라는 결과값을 리턴하게 된 것이다.

앞의 함수는 lambda를 이용하면 더욱 간편하게 코드를 작성할 수 있다.

```
>>> print(list(filter(lambda x: x > 0, [1, -3, 2, 0, -5, 6])))
```

hex

hex(x)는 정수값을 입력받아 16진수(hexadecimal)로 변환하여 리턴하는 함수이다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> hex(234)
'0xea'
>>> hex(3)
'0x3'
```

id

`id(object)`는 객체를 입력받아 객체의 고유 주소값(레퍼런스)을 리턴하는 함수이다.

```
>>> a = 3
>>> id(3)
135072304
>>> id(a)
135072304
>>> b = a
>>> id(b)
135072304
```

위 예의 3, a, b는 고유 주소값이 모두 135072304이다. 즉, 3, a, b가 모두 같은 객체를 가리키고 있음을 알 수 있다.

만약 `id(4)`라고 입력하면 4는 3, a, b와 다른 객체이므로 당연히 다른 고유 주소값이 출력된다.

```
>>> id(4)
135072292
```

input

`input([prompt])`은 사용자 입력을 받는 함수이다. 입력 인수로 문자열을 주면 아래의 세 번째 예에서 볼 수 있듯이 그 문자열은 프롬프트가 된다.

(※ [] 기호는 괄호 안의 내용을 생략할 수 있다는 관례적인 표기법임을 기억하자.)

(※ 파이썬 2.7 버전의 경우 `input` 대신 `raw_input`을 사용해야 한다.)

```
>>> a = input()
hi
>>> a
'hi'
>>> b = input("Enter: ")
Enter: hi
```

위에서 입력받은 문자열을 확인해 보면 다음과 같다.

```
>>> b
'hi'
```

int

`int(x)`는 문자열 형태의 숫자나 소수점이 있는 숫자 등을 정수 형태로 리턴하는 함수로, 정수를 입력으로 받으면 그대로 리턴한다.

```
>>> int('3')
3
>>> int(3.4)
3
```

`int(x, radix)`는 radix 진수로 표현된 문자열 x를 10진수로 변환하여 리턴한다.

2진수로 표현된 '11'의 10진수 값은 다음과 같이 구한다.

```
>>> int('11', 2)
3
```

16진수로 표현된 '1A'의 10진수 값은 다음과 같이 구한다.

```
>>> int('1A', 16)
26
```

isinstance

`isinstance(object, class)`는 첫 번째 인수로 인스턴스, 두 번째 인수로 클래스 이름을 받는다. 입력으로 받은 인스턴스가 그 클래스의 인스턴스인지를 판단하여 참이면 `True`, 거짓이면 `False`를 리턴한다.

```
>>> class Person: pass
...
>>> a = Person()
>>> isinstance(a, Person)
True
```

위의 예는 `a`가 `Person` 클래스에 의해서 생성된 인스턴스임을 확인시켜 준다.

```
>>> b = 3
>>> isinstance(b, Person)
False
```

`b`는 `Person` 클래스에 의해 생성된 인스턴스가 아니므로 `False`를 리턴한다.

len

`len(s)`은 입력값 `s`의 길이(요소의 전체 개수)를 리턴하는 함수이다.

```
>>> len("python")
6
>>> len([1,2,3])
3
>>> len((1, 'a'))
2
```

list

`list(s)`는 반복 가능한 자료형 `s`를 입력받아 리스트로 만들어 리턴하는 함수이다.

```
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list((1,2,3))
[1, 2, 3]
```

list 함수에 리스트를 입력으로 주면 똑같은 리스트를 복사하여 돌려준다.

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> b
[1, 2, 3]
```

map

map(f, iterable)은 함수(f)와 반복 가능한(iterable) 자료형을 입력으로 받는다. map은 입력받은 자료형의 각 요소가 함수 f에 의해 수행된 결과를 끌어서 리턴하는 함수이다.

다음의 예를 보자.

```
# two_times.py
def two_times(numberList):
    result = []
    for number in numberList:
        result.append(number*2)
    return result

result = two_times([1, 2, 3, 4])
print(result)
```

two_times 함수는 리스트 요소를 입력받아 각 요소에 2를 곱한 결과값을 돌려준다. 실행 결과는 다음과 같다.

결과값: [2, 4, 6, 8]

위의 예제는 map 함수를 이용하면 다음처럼 바꿀 수 있다.

```
>>> def two_times(x): return x*2
...
>>> list(map(two_times, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

이제 앞 예제를 해석해 보자. 먼저 리스트의 첫 번째 요소인 1이 two_times 함수의 입력값으로 들어가고, $1 * 2$ 의 과정을 거쳐서 2가 된다. 다음으로 리스트의 두 번째 요소인 2가 $2 * 2$ 의 과정을 거쳐 4가 된다. 따라서 결과값 리스트는 이제 [2, 4]가 된다. 총 4개의 요소값이 모두 수행되면 최종적으로 [2, 4, 6, 8]이 리턴된다. 이것이 map 함수가 하는 일이다.

※ 위 예에서 map의 결과를 리스트로 보여 주기 위해 list 함수를 이용하여 출력하였 다. 파이썬 2.7은 map의 결과가 리스트이므로 위 예에서 list 함수를 이용하여 리스트로 변환하지 않아도 된다.

앞의 예는 lambda를 사용하면 다음처럼 간략하게 만들 수 있다.

```
>>> list(map(lambda a: a*2, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

map 함수 예를 하나 더 살펴보자.

```
# map_test.py
def plus_one(x):
    return x+1
print(list(map(plus_one, [1, 2, 3, 4, 5])))
```

결과값: [2, 3, 4, 5, 6]

위 예는 map과 plus_one 함수를 이용하여 리스트의 각 요소값을 1씩 증가시키는 예제이다.

max

max(iterable)는 인수로 반복 가능한 자료형을 입력받아 그 최대값을 리턴하는 함수이다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> max([1, 2, 3])
3
>>> max("python")
'y'
```

min

`min(iterable)`은 `max` 함수와 반대로, 인수로 반복 가능한 자료형을 입력받아 그 최소값을 리턴하는 함수이다.

```
>>> min([1, 2, 3])
1
>>> min("python")
'h'
```

oct

`oct(x)`는 정수 형태의 숫자를 8진수 문자열로 바꾸어 리턴하는 함수이다.

```
>>> oct(34)
'0o42'
>>> oct(12345)
'0o30071'
```

open

`open(filename, [mode])`은 “파일 이름”과 “읽기 방법”을 입력받아 파일 객체를 리턴하는 함수이다. 읽기 방법(mode)이 생략되면 기본값인 읽기 전용 모드(r)로 파일 객체를 만들어 리턴한다.

mode	설명
w	쓰기 모드로 파일 열기
r	읽기 모드로 파일 열기
a	추가 모드로 파일 열기

mode	설명
b	바이너리 모드로 파일 열기

b는 w, r, a와 함께 사용된다.

```
>>> f = open("binary_file", "rb")
```

위 예의 rb는 “바이너리 읽기 모드”를 의미한다.

아래 예의 fread와 fread2는 동일한 방법이다.

```
>>> fread = open("read_mode.txt", 'r')
>>> fread2 = open("read_mode.txt")
```

즉, 모드 부분이 생략되면 기본값으로 읽기 모드인 r을 갖게 된다.

다음은 추가 모드(a)로 파일을 여는 예이다.

```
>>> fappend = open("append_mode.txt", 'a')
```

ord

ord(c)는 문자의 아스키 코드값을 리턴하는 함수이다.

※ ord 함수는 chr 함수와 반대이다.

```
>>> ord('a')
97
>>> ord('0')
48
```

pow

pow(x, y)는 x의 y 제곱한 결과값을 리턴하는 함수이다.

```
>>> pow(2, 4)
16
>>> pow(3, 3)
27
```

range

`range([start,] stop [,step])`는 for문과 함께 자주 사용되는 함수이다. 이 함수는 입력받은 숫자에 해당되는 범위의 값을 반복 가능한 객체로 만들어 리턴한다.

인수가 하나일 경우

시작 숫자를 지정해 주지 않으면 range 함수는 0부터 시작한다.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

인수가 2개일 경우

입력으로 주어지는 2개의 인수는 시작 숫자와 끝 숫자를 나타낸다. 단, 끝 숫자는 해당 범위에 포함되지 않는다는 것에 주의하자.

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

인수가 3개일 경우

세 번째 인수는 숫자 사이의 거리를 말한다.

```
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

round

`round(number[, ndigits])` 함수는 숫자를 입력받아 반올림 해 주는 함수이다.

```
>>> round(4.6)
5
>>> round(4.2)
4
```

만약 5.678 라는 실수를 소수점 2자리까지만 반올림하여 표시하고 싶다면 다음과 같이 사용할 수 있다.

```
>>> round(5.678, 2)
5.68
```

round 함수의 두번째 파라미터는 반올림하여 표시하고 싶은 소수점의 자리수(ndigits)이다.

sorted

sorted(iterable) 함수는 입력값을 정렬한 후 그 결과를 리스트로 리턴하는 함수이다.

```
>>> sorted([3, 1, 2])
[1, 2, 3]
>>> sorted(['a', 'c', 'b'])
['a', 'b', 'c']
>>> sorted("zero")
['e', 'o', 'r', 'z']
>>> sorted((3, 2, 1))
[1, 2, 3]
```

리스트 자료형에도 sort라는 함수가 있다. 하지만 리스트 자료형의 sort 함수는 리스트 객체 그 자체를 정렬만 할 뿐 정렬된 결과를 리턴하지는 않는다.

다음 예제로 sorted 함수와 리스트 자료형의 sort 함수의 차이점을 확인해 보자.

```
>>> a = [3, 1, 2]
>>> result = a.sort()
>>> print(result)
None
>>> a
[1, 2, 3]
```

sort 함수는 리턴값이 없기 때문에 result 변수에 저장되는 값이 없다. 따라서 print(result)를 하면 None이 출력된다. sort 함수를 수행한 후 리턴값은 없지만 리스트 객체 a를 확인하면 [3, 1, 2]가 [1, 2, 3]으로 정렬된 것을 볼 수 있다.

str

str(object)은 문자열 형태로 객체를 변환하여 리턴하는 함수이다.

```
>>> str(3)
'3'
>>> str('hi')
'hi'
>>> str('hi'.upper())
'HI'
```

tuple

tuple(iterable)은 반복 가능한 자료형을 입력받아 튜플 형태로 바꾸어 리턴하는 함수이다. 만약 튜플이 입력으로 들어오면 그대로 리턴한다.

```
>>> tuple("abc")
('a', 'b', 'c')
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple((1, 2, 3))
(1, 2, 3)
```

type

`type(object)`은 입력값의 자료형이 무엇인지 알려주는 함수이다.

```
>>> type("abc")
<class 'str'>
>>> type([ ])
<class 'list'>
>>> type(open("test", 'w'))
<class '_io.TextIOWrapper'>
```

zip

`zip(*iterable)`은 동일한 개수로 이루어진 자료형을 묶어 주는 역할을 하는 함수이다.

※ 위에서 사용된 `*iterable`의 의미는 반복가능(iterable)한 자료형 여러개가 입력으로 가능하다는 의미이다.

잘 이해되지 않는다면 다음 예제를 살펴보자.

```
>>> list(zip([1, 2, 3], [4, 5, 6]))
[(1, 4), (2, 5), (3, 6)]
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>> list(zip("abc", "def"))
[('a', 'd'), ('b', 'e'), ('c', 'f')]
```

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#05-5>)

[문제1] 내장함수

다음 결과를 예측하시오.

1)

```
>>> all([1, 2, abs(-3)-3])
```

2)

```
>>> chr(ord('a')) == 'a'
```

[문제2] enumerate

enumerate 내장함수를 이용하여 ['a', 'b', 'c']라는 리스트를 {0:'a', 1:'b', 2:'c'}라는 딕셔너리로 바꾸시오.

[문제3] filter와 lambda

filter와 lambda를 이용하여 [1, -2, 3, -5, 8, -3]라는 리스트에서 음수를 모두 제거하시오.

[문제4] 16진수를 10진수로 변환

234라는 10진수의 16진수는 다음과 같이 구할 수 있다.

```
>>> hex(234)
'0xea'
```

이번에는 반대로 '0xea'라는 16진수 문자열을 10진수로 변경해 보시오. (힌트. 내장함수 int를 활용해 보자.)

[문제5] map과 lambda

map과 lambda를 이용하여 [1, 2, 3, 4]라는 리스트의 각 요소값에 3이 곱해진 [3, 6, 9, 12]라는 리스트를 만드시오.

[문제6] 최대값과 최소값

다음 리스트의 최대값과 최소값의 합을 구하시오.

[-8, 2, 7, 5, -3, 5, 0, 1]

[문제7] 소수점 반올림

17 / 3의 결과는 다음과 같다.

```
>>> 17 / 3
5.6666666666666667
```

위와 같은 결과값 5.666666666666667을 소수점 4자리까지만 반올림하여 표시하시오.

[문제8] zip

[1, 2, 3, 4]와 ['a', 'b', 'c', 'd']라는 리스트가 있다. 이 두개의 리스트를 합쳐 다음과 같은 리스트를 만드시오. (힌트. 내장함수 zip을 이용해 보자.)

[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

05-6 외장 함수

이제 파이썬 프로그래밍 능력을 높여 줄 더 큰 날개를 달아 보자. 전 세계의 파이썬 사용자들이 만든 유용한 프로그램들을 모아 놓은 것이 바로 파이썬 라이브러리이다. “라이브러리”는 “도서관”이라는 뜻 그대로 원하는 정보를 찾아보는 곳이다. 모든 라이브러리를 다 알 필요는 없고 어떤 일을 할 때 어떤 라이브러리를 사용해야 한다는 정도만 알면 된다. 그러기 위해 어떤 라이브러리들이 존재하고 어떻게 사용되는지 알아야 할 필요가 있다. 자주 사용되고 꼭 알아두면 좋은 라이브러리들을 중심으로 하나씩 살펴보자.

※ 파이썬 라이브러리는 파이썬 설치 시 자동으로 컴퓨터에 설치가 된다.

sys

sys 모듈은 파이썬 인터프리터가 제공하는 변수들과 함수들을 직접 제어할 수 있게 해주는 모듈이다.

명령 행에서 인수 전달하기 - sys.argv

```
C:/User/home>python test.py abc pey guido
```

명령 프롬프트 창에서 위의 예처럼 test.py 뒤에 또 다른 값들을 함께 넣어 주면 sys.argv라는 리스트에 그 값들이 추가된다.

예제를 따라 하며 확인해 보자. 우선 다음과 같은 파이썬 프로그램을 작성하자. argv_test.py 파일은 C:/doit/mymod 디렉터리에 저장했다고 가정한다(만약 C:/doit/mymod 디렉터리가 없다면 먼저 생성하고 진행하자).

```
# argv_test.py
import sys
print(sys.argv)
```

명령 프롬프트 창에서 다음과 같이 실행해 보자.

```
C:/doit/mymod>python argv_test.py you need python
['argv_test.py', 'you', 'need', 'python']
```

python이라는 명령어 뒤의 모든 것들이 공백을 기준으로 나뉘어서 sys.argv 리스트의 요소가 됨을 알 수 있다.

※ 명령 프롬프트 창에서는 /, \는 상관없지만, 소스코드 안에서는 반드시 / 또는 \\ 기호를 사용해야 한다.

강제로 스크립트 종료하기 - sys.exit

```
>>> sys.exit()
```

sys.exit는 Ctrl+Z나 Ctrl+D를 눌러서 대화형 인터프리터를 종료하는 것과 같은 기능을 한다. 프로그램 파일 내에서 사용하면 프로그램을 중단시킨다.

자신이 만든 모듈 불러와 사용하기 - sys.path

sys.path는 파이썬 모듈들이 저장되어 있는 위치를 나타낸다. 즉, 이 위치에 있는 파이썬 모듈들은 경로에 상관없이 어디에서나 불러올 수가 있다.

다음은 그 실행 결과이다.

```
>>> import sys
>>> sys.path
[‘’, ‘C:\\Windows\\SYSTEM32\\python36.zip’, ‘c:\\Python36\\DLLs’,
‘c:\\Python36\\lib’, ‘c:\\Python36’, ‘c:\\Python36\\lib\\site-packages’]
>>>
```

위의 예에서 ”는 현재 디렉터리를 말한다.

```
# path_append.py
import sys
sys.path.append("C:/doit/mymod")
```

위와 같이 파이썬 프로그램 파일에서 sys.path.append를 이용해 경로명을 추가할 수 있다. 이렇게 하고 난 후에는 C:/doit/mymod라는 디렉터리에 있는 파이썬 모듈을 불러와서 사용할 수가 있다.

pickle

pickle은 객체의 형태를 그대로 유지하면서 파일에 저장하고 불러올 수 있게 하는 모듈이다. 다음 예는 pickle 모듈의 dump 함수를 이용하여 딕셔너리 객체인 data를 그대로 파일에 저장하는 방법을 보여 준다.

```
>>> import pickle
>>> f = open("test.txt", 'wb')
>>> data = {1: 'python', 2: 'you need'}
>>> pickle.dump(data, f)
>>> f.close()
```

다음은 pickle.dump에 의해 저장된 파일을 pickle.load를 이용해서 원래 있던 딕셔너리 객체(data) 상태 그대로 불러오는 예이다.

```
>>> import pickle
>>> f = open("test.txt", 'rb')
>>> data = pickle.load(f)
>>> print(data)
{2: 'you need', 1: 'python'}
```

위의 예에서는 딕셔너리 객체를 이용하였지만 어떤 자료형이든 상관없이 저장하고 불러올 수 있다.

os

OS 모듈은 환경 변수나 디렉터리, 파일 등의 OS 자원을 제어할 수 있게 해주는 모듈이다.

내 시스템의 환경 변수값을 알고 싶을 때 - os.environ

시스템은 제각기 다른 환경 변수값을 가지고 있는데, os.environ은 현재 시스템의 환경 변수 값들을 보여 준다. 다음을 따라 해보자.

```
>>> import os
>>> os.environ
environ({'PROGRAMFILES': 'C:\\Program Files', 'APPDATA': ... 생략 ...})
>>>
```

위의 결과값은 필자의 시스템 정보이다. os.environ은 환경 변수에 대한 정보를 딕셔너리 객체로 리턴한다. 자세히 보면 여러 가지 유용한 정보를 찾을 수 있다.

리턴받은 객체가 딕셔너리이기 때문에 다음과 같이 호출할 수 있다. 다음은 필자 시스템의 PATH 환경 변수에 대한 내용이다.

```
>>> os.environ['PATH']
'C:\ProgramData\Oracle\Java\javapath;...생략...'
```

디렉터리 위치 변경하기 - os.chdir

os.chdir을 이용하면 아래와 같이 현재 디렉터리의 위치를 변경할 수 있다.

```
>>> os.chdir("C:\WINDOWS")
```

디렉터리 위치 리턴받기 - os.getcwd

os.getcwd는 현재 자신의 디렉터리 위치를 리턴한다.

```
>>> os.getcwd()
'C:\WINDOWS'
```

시스템 명령어 호출하기 - os.system

시스템 자체의 프로그램이나 기타 명령어들을 파이썬에서 호출할 수도 있다. os.system("명령어")처럼 사용한다. 다음은 현재 디렉터리에서 시스템 명령어인 dir을 실행하는 예이다.

```
>>> os.system("dir")
```

실행한 시스템 명령어의 결과값 리턴받기 - os.popen

os.popen은 시스템 명령어를 실행시킨 결과값을 읽기 모드 형태의 파일 객체로 리턴한다.

```
>>> f = os.popen("dir")
```

읽어 들인 파일 객체의 내용을 보기 위해서는 다음과 같이 하면 된다.

```
>>> print(f.read())
```

기타 유용한 os 관련 함수

함수	설명
os.mkdir(디렉터리)	디렉터리를 생성한다.
os.rmdir(디렉터리)	디렉터리를 삭제한다. 단, 디렉터리가 비어있어야 삭제가 가능하다.
os.unlink(파일)	파일을 지운다.
os.rename(src, dst)	src라는 이름의 파일을 dst라는 이름으로 바꾼다.

shutil

shutil은 파일을 복사해 주는 파이썬 모듈이다.

파일 복사하기 - shutil.copy(src, dst)

src라는 이름의 파일을 dst로 복사한다. 만약 dst가 디렉터리 이름이라면 src라는 파일 이름으로 dst라는 디렉터리에 복사하고 동일한 파일 이름이 있을 경우에는 덮어쓴다.

다음의 예를 보자.

```
>>> import shutil
>>> shutil.copy("src.txt", "dst.txt")
```

위 예를 실행해 보면 src.txt 파일과 동일한 내용의 파일이 dst.txt로 복사되는 것을 확인할 수 있다.

glob

가끔 파일을 읽고 쓰는 기능이 있는 프로그램을 만들다 보면 특정 디렉터리에 있는 파일 이름 모두를 알아야 할 때가 있다. 이럴 때 사용하는 모듈이 바로 glob이다.

디렉터리에 있는 파일들을 리스트로 만들기 - glob(pathname)

glob 모듈은 디렉터리 내의 파일들을 읽어서 리턴한다. *, ? 등의 메타 문자를 써서 원하는 파일만 읽어 들일 수도 있다.

다음은 C:/doit이라는 디렉터리에 있는 파일 중 이름이 문자 q로 시작하는 파일들을 모두 찾아서 읽어 들이는 예이다.

```
>>> import glob
>>> glob.glob("C:/doit/q*")
['C:\doit\quiz.py', 'C:\doit\quiz.py.bak']
>>>
```

tempfile

파일을 임시로 만들어서 사용할 때 유용한 모듈이 바로 tempfile이다. tempfile.mktemp()는 중복되지 않는 임시 파일의 이름을 무작위로 만들어서 리턴한다.

```
>>> import tempfile
>>> filename = tempfile.mktemp()
>>> filename
'C:\WINDOWS\TEMP\~-275151-0'
```

tempfile.TemporaryFile()은 임시 저장 공간으로 사용될 파일 객체를 리턴한다. 이 파일은 기본적으로 바이너리 쓰기 모드(wb)를 갖는다. f.close()가 호출되면 이 파일 객체는 자동으로 사라진다.

```
>>> import tempfile
>>> f = tempfile.TemporaryFile()
>>> f.close()
```

time

시간과 관련된 time 모듈에는 유용한 함수가 굉장히 많다. 그중에서 가장 유용한 몇 가지만 알아보자.

time.time

time.time()은 UTC(Universal Time Coordinated 협정 세계 표준시)를 이용하여 현재 시간을 실수 형태로 리턴하는 함수이다. 1970년 1월 1일 0시 0분 0초를 기준으로 지난 시간을 초 단위로 리턴한다.

```
>>> import time
>>> time.time()
988458015.73417199
```

time.localtime

time.localtime은 time.time()에 의해서 반환된 실수값을 이용해서 연도, 월, 일, 시, 분, 초,..의 형태로 바꾸어 주는 함수이다.

```
>>> time.localtime(time.time())
time.struct_time(tm_year=2013, tm_mon=5, tm_mday=21, tm_hour=16,
tm_min=48, tm_sec=42, tm_wday=1, tm_yday=141, tm_isdst=0)
```

time.asctime

위의 time.localtime에 의해서 반환된 튜플 형태의 값을 인수로 받아서 날짜와 시간을 알아보기 쉬운 형태로 리턴하는 함수이다.

```
>>> time.asctime(time.localtime(time.time()))
'Sat Apr 28 20:50:20 2001'
```

time.ctime

time.asctime(time.localtime(time.time()))은 time.ctime()을 이용해 간편하게 표시할 수 있다. asctime과 다른점은 ctime은 항상 현재 시간만을 리턴한다는 점이다.

```
>>> time.ctime()
'Sat Apr 28 20:56:31 2001'
```

time.strftime

```
time.strftime('출력할 형식 포맷 코드', time.localtime(time.time()))
```

strftime 함수는 시간에 관계된 것을 세밀하게 표현할 수 있는 여러 가지 포맷 코드를 제공한다.

시간에 관계된 것을 표현하는 포맷 코드

포맷코드	설명	예
%a	요일 줄임말	Mon
%A	요일	Monday
%b	달 줄임말	Jan
%B	달	January
%c	날짜와 시간을 출력함	06/01/01 17:22:21
%d	날(day)	[00,31]
%H	시간(hour)-24시간 출력 형태	[00,23]
%I	시간(hour)-12시간 출력 형태	[01,12]
%j	1년 중 누적 날짜	[001,366]
%m	달	[01,12]
%M	분	[01,59]
%p	AM or PM	AM
%S	초	[00,61]
%U	1년 중 누적 주-일요일을 시작으로	[00,53]
%w	숫자로 된 요일	[0(일요일),6]
%W	1년 중 누적 주-월요일을 시작으로	[00,53]
%x	현재 설정된 로케일에 기반한 날짜 출력	06/01/01
%X	현재 설정된 로케일에 기반한 시간 출력	17:22:21
%Y	년도 출력	2001
%Z	시간대 출력	대한민국 표준시
%%	문자	%
%y	세기부분을 제외한 년도 출력	01

포맷코드 설명

예

다음은 time.strftime을 사용하는 예이다.

```
>>> import time
>>> time.strftime('%x', time.localtime(time.time()))
'05/01/01'
>>> time.strftime('%c', time.localtime(time.time()))
'05/01/01 17:22:21'
```

time.sleep

time.sleep 함수는 주로 루프 안에서 많이 사용된다. 이 함수를 사용하면 일정한 시간 간격을 두고 루프를 실행할 수 있다. 다음의 예를 보자.

```
#sleep1.py
import time
for i in range(10):
    print(i)
    time.sleep(1)
```

위 예는 1초 간격으로 0부터 9까지의 숫자를 출력한다. 위 예에서 볼 수 있듯이 time.sleep 함수의 인수는 실수 형태를 쓸 수 있다. 즉, 1이면 1초, 0.5면 0.5초가 되는 것이다.

calendar

calendar는 파이썬에서 달력을 볼 수 있게 해주는 모듈이다.

calendar.calendar(연도)로 사용하면 그 해의 전체 달력을 볼 수 있다. 결과값은 달력이 너무 길어 생략하겠다.

```
>>> import calendar
>>> print(calendar.calendar(2015))
```

calendar.prcal(연도)를 사용해도 위와 똑같은 결과값을 얻을 수 있다.

```
>>> calendar.prcal(2015)
```

다음의 예는 2015년 12월의 달력만 보여 준다.

```
>>> calendar.prmonth(2015, 12)
December 2015
Mo Tu We Th Fr Sa Su
 2  3  4  5  6
 7  8  9  10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

calendar.weekday

calendar 모듈의 또 다른 유용한 함수를 보자. weekday(연도, 월, 일) 함수는 그 날짜에 해당하는 요일 정보를 리턴한다. 월요일은 0, 화요일은 1, 수요일은 2, 목요일은 3, 금요일은 4, 토요일은 5, 일요일은 6이라는 값을 리턴한다.

```
>>> calendar.weekday(2015, 12, 31)
3
```

위의 예에서 2015년 12월 31일은 목요일임을 보여 준다.

calendar.monthrange

monthrange(연도, 월) 함수는 입력받은 달의 1일이 무슨 요일인지와 그 달이 며칠까지 있는지를 튜플 형태로 리턴한다.

```
>>> calendar.monthrange(2015,12)
(1, 31)
```

위의 예는 2015년 12월의 1일은 화요일이고, 이 달은 31일까지 있다는 것을 보여 준다.

날짜와 관련된 프로그래밍을 할 때 위의 2가지 함수는 매우 유용하게 사용된다.

random

random은 난수(규칙이 없는 임의의 수)를 발생시키는 모듈이다. random과 randint에 대해서 알아보자.

다음은 0.0에서 1.0 사이의 실수 중에서 난수값을 리턴하는 예를 보여 준다.

```
>>> import random
>>> random.random()
0.53840103305098674
```

다음 예는 1에서 10 사이의 정수 중에서 난수값을 리턴한다.

```
>>> random.randint(1, 10)
6
```

다음 예는 1에서 55 사이의 정수 중에서 난수값을 리턴한다.

```
>>> random.randint(1, 55)
43
```

random 모듈을 이용해서 재미있는 함수를 하나 만들어 보자.

```
# random_pop.py
import random
def random_pop(data):
    number = random.randint(0, len(data)-1)
    return data.pop(number)

if __name__ == "__main__":
    data = [1, 2, 3, 4, 5]
    while data: print(random_pop(data))
```

결과값:

2
3
1
5
4

위의 random.pop 함수는 리스트의 요소 중에서 무작위로 하나를 선택하여 꺼낸 다음 그 값을 리턴한다. 물론 꺼내진 요소는 pop 메서드에 의해 사라진다.

random.pop 함수는 random 모듈의 choice 함수를 사용하여 다음과 같이 좀 더 직관적으로 만들 수도 있다.

```
def random_pop(data):
    number = random.choice(data)
    data.remove(number)
    return number
```

random.choice 함수는 입력으로 받은 리스트에서 무작위로 하나를 선택하여 리턴한다.

리스트의 항목을 무작위로 섞고 싶을 때는 random.shuffle 함수를 이용하면 된다.

```
>>> import random
>>> data = [1, 2, 3, 4, 5]
>>> random.shuffle(data)
>>> data
[5, 1, 3, 4, 2]
>>>
```

[1, 2, 3, 4, 5]라는 리스트가 shuffle 함수에 의해 섞여서 [5, 1, 3, 4, 2]로 변한 것을 확인할 수 있다.

webbrowser

webbrowser는 자신의 시스템에서 사용하는 기본 웹 브라우저가 자동으로 실행되게 하는 모듈이다. 아래의 예제는 웹 브라우저를 자동으로 실행시키고 해당 URL인 <http://google.com>으로 가게 해준다.

```
>>> import webbrowser
>>> webbrowser.open("http://google.com")
```

webbrowser의 open 함수는 웹 브라우저가 이미 실행된 상태이면 입력 주소로 이동한다. 만약 웹 브라우저가 실행되지 않은 상태이면 새로 웹 브라우저를 실행한 후 해당 주소로 이동한다.

open_new 함수는 이미 웹 브라우저가 실행된 상태이더라도 새로운 창으로 해당 주소가 열리도록 한다.

```
>>> webbrowser.open_new("http://google.com")
```

namedtuple

namedtuple이란 파이썬 자료형 중 하나로 튜플이지만 속성으로 그 값에 접근할 수 있게 해 주는 자료형이다.

먼저 다음과 같은 프로그램을 생각해 보자.

```
a = ("홍길동", 25, "Programmer")
b = ("김철수", 32, "Manager")
c = ("김영희", 41, "Designer")

for person in [a, b, c]:
    print("이름:%s" % person[0])
    print("나이:%s" % person[1])
    print("직업:%s" % person[2])
```

a, b, c라는 튜플을 순서대로 출력하는 프로그램이다. 각 튜플은 차례대로 이름, 나이, 직업을 요소로 갖는 튜플이다.

튜플의 요소에 접근하기 위해서는 인덱싱만이 가능하므로 좀 더 편리한 접근을 위해 다음과 같이 개선이 가능하다.

```

class Person:
    def __init__(self, name, age, job):
        self.name = name
        self.age = age
        self.job = job

    a = Person(name="홍길동", age=25, job="Programmer")
    b = Person(name="김철수", age=32, job="Manager")
    c = Person(name="김영희", age=41, job="Designer")

    for person in [a, b, c]:
        print("이름:%s" % person.name)
        print("나이:%s" % person.age)
        print("직업:%s" % person.job)

```

튜플 대신 Person이라는 클래스를 생성하면 위 예처럼 인덱싱이 아닌 속성으로 접근할 수 있게 된다.

이번에는 좀 더 편리한 namedtuple을 이용한 방법을 알아보도록 하자. namedtuple을 사용하면 다음과 같이 작성할 수 있다.

```

from collections import namedtuple

Person = namedtuple("Person", ["name", "age", "job"])

a = Person(name="홍길동", age=25, job="Programmer")
b = Person(name="김철수", age=32, job="Manager")
c = Person(name="김영희", age=41, job="Designer")

for person in [a, b, c]:
    print("이름:%s" % person.name)
    print("나이:%s" % person.age)
    print("직업:%s" % person.job)

```

namedtuple을 이용하면 Person 클래스를 따로 만들지 않아도 비슷하게 동작하게 할 수 있다.

namedtuple의 첫번째 입력항목은 namedtuple의 자료형의 명칭(type name)이다. 보통 namedtuple로 생성되는 객체명과 동일하게 한다. 뒤에 따라오는 리스트는 Person이라는 namedtuple의

변수로 사용될 항목들이 된다.

namedtuple의 첫번째 입력항목은 namedtuple로 생성되는 객체의 클래스 타입을 의미한다.

```
>>> from collections import namedtuple
>>> Person = namedtuple("Person", ["name", "age", "job"])
>>> Person
<class '__main__.Person'>
>>> Person = namedtuple("Human", ["name", "age", "job"])
>>> Person
<class '__main__.Human'>
```

위 예를 보면 두개의 Person은 첫번째 입력항목만 제외하고 모두 같다. 사용하는 방법도 다를게 없다. 다만 생성된 객체의 클래스 타입이 다를 뿐이다.

namedtuple은 이름 그대로 tuple이기 때문에 다음과 같이 인덱싱으로도 역시 접근이 가능하다.

```
for person in [a, b, c]:
    print("이름:%s" % person[0])
    print("나이:%s" % person[1])
    print("직업:%s" % person[2])
```

단, namedtuple은 요소값을 변경할 수 없는(immutable) 튜플의 성격을 갖기 때문에 그 값을 변경할 수는 없다. 따라서 다음과 같은 코드는 오류가 발생하게 된다.

```
>>> from collections import namedtuple
>>> Person = namedtuple("Person", ["name", "age", "job"])
>>> a = Person(name="홍길동", age=25, job="Programmer")
>>> a.name = "고길동"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

defaultdict

우리가 좋아하는 다음과 같은 문장을 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
Life is too short, you need python
```

위 문장에서 각 문자(character)가 몇 개씩 있는지 알고 싶다면 어떻게 해야 할까?

딕셔너리를 이용하면 다음과 같은 코드를 만들 수 있을 것이다.

```
>>> a = 'Life is too short, you need python'
>>> d = dict()
>>> for c in a:
...     if c in d:
...         d[c] += 1
...     else:
...         d[c] = 1
...
>>> d.items()
dict_items([('L', 1), ('i', 2), ('f', 1), ('e', 3), (' ', 6), ('s', 2), ('t', 3), ('o', 5), ('h', 2), ('r', 1), ('y', 2), ('u', 1), ('n', 2), ('d', 1), ('p', 1)])
```

위 코드 작성시 주의 할 점은 `if c in d:` 구문 처럼 딕셔너리 `d`에 해당 키 값이 있는지 조사하는 부분이다. 해당 키 값이 이미 있을때는 값을 1만큼 증가시켜 주고 없을 경우에는 1이라는 초기 값을 세팅해 주었다.

이렇듯 딕셔너리의 키에 해당되는 값을 추가하거나 변경할 경우에는 위처럼 방어적인 코드가 반드시 필요하다. `defaultdict`를 이용하면 이러한 방어적인 코드를 작성해야 하는 스트레스를 줄여줄 수 있다.

`defaultdict`를 이용하면 위 코드를 다음과 같이 작성할 수 있다.

```
>>> from collections import defaultdict
>>> a = 'Life is too short you need python'
>>> d = defaultdict(int)
>>> for c in a:
...     d[c] += 1
...
>>> d.items()
dict_items([('L', 1), ('i', 2), ('f', 1), ('e', 3), (' ', 6), ('s', 2), ('t', 3), ('o', 5), ('h', 2), ('r', 1), ('y', 2), ('u', 1), ('n', 2), ('d', 1), ('p', 1)])
```

d 객체에 키가 있는지 조사하여 방어적으로 코딩했던 부분을 생략할 수 있음을 알 수 있다. `d = defaultdict(int)`라는 문장은 d라는 defaultdict 객체의 디폴트 값은 int라는 의미이다. 따라서 d에 해당 키 값이 없을 경우 자동적으로 int의 초기값인 0이라는 값이 저장되게 된다.

이번에는 int가 아닌 list를 디폴트 값으로 활용하는 예를 보자.

```
>>> s = [('a', 100), ('b', 200), ('c', 300), ('a', 150), ('c', 120)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> d.items()
dict_items([('a', [100, 150]), ('b', [200]), ('c', [300, 120])])
```

딕셔너리의 value에 초기값으로 빈 리스트가 올 수 있도록 `d = defaultdict(list)`와 같이 선언해 주어 깔끔한 코드를 작성할 수 있는 모습이다.

[스레드를 다루는 threading 모듈]

컴퓨터에서 동작하고 있는 프로그램을 프로세스(Process)라고 한다. 보통 1개의 프로세스는 1가지 일만 하지만, 스레드를 이용하면 한 프로세스 내에서 2가지 또는 그 이상의 일을 동시에 수행하게 할 수 있다. 간단한 예제로 설명을 대신하겠다.

```

import threading
import time

def say(msg):
    while True:
        time.sleep(1)
        print(msg)

for msg in ['you', 'need', 'python']:
    t = threading.Thread(target=say, args=(msg,))
    t.daemon = True
    t.start()

for i in range(100):
    time.sleep(0.1)
    print(i)

```

첫 번째 for문에서 ['you', 'need', 'python']이라는 리스트의 요소 개수만큼 스레드가 생성되고, 생성된 스레드는 say 메서드를 수행하게 되어 1초에 한 번씩 입력으로 받은 msg 변수값을 리턴한다. 두 번째 for문은 매 0.1초마다 0부터 99까지 숫자를 출력하는데, 바로 이 부분이 메인 프로그램이 되며 이 메인 프로그램이 종료되는 순간 생성된 스레드들도 함께 종료가 된다. t.daemon = True 와 같이 daemon 플래그를 설정하면 주 프로그램이 종료되는 순간 데몬 스레드도 함께 종료된다.

위 예제의 실행 결과값은 다음과 비슷할 것이다.

```
0
you
need
python
1
2
3
4
5
6
7
8
9
10
you
need
python
11
12
...
```

위 결과값에서 볼 수 있듯이 스레드는 메인 프로그램과는 별도로 실행되는 것을 확인할 수 있다.

이러한 스레드 프로그래밍을 가능하게 해주는 것이 바로 `threading.Thread` 클래스이다. 이 클래스의 첫번째 인수는 함수 이름을, 두 번째 인수는 첫 번째 인수인 함수의 입력 변수를 받는다. 다음과 같이 스레드를 클래스로 정의해도 동일한 결과를 얻을 수 있다.

```

import threading
import time

class MyThread(threading.Thread):
    def __init__(self, msg):
        threading.Thread.__init__(self)
        self.msg = msg
        self.daemon = True

    def run(self):
        while True:
            time.sleep(1)
            print(self.msg)

for msg in ['you', 'need', 'python']:
    t = MyThread(msg)
    t.start()

for i in range(100):
    time.sleep(0.1)
    print(i)

```

스레드를 클래스로 정의할 경우에는 `__init__` 메서드에서 `threading.Thread.__init__(self)`와 같이 부모 클래스의 생성자를 반드시 호출해야 한다. `MyThread`로 생성된 객체의 `start` 메서드를 실행할 때는 `MyThread` 클래스의 `run` 메서드가 자동으로 수행된다.

연습문제

(연습문제 풀이 : <https://wikidocs.net/17090#05-6>)

[문제1] sys.argv

다음과 같이 실행할 때 입력 값을 모두 더하여 출력하는 스크립트(`c:\doit\myargv.py`)를 작성 하시오.

(힌트. 외장함수 `sys.argv`를 이용해 보자)

```
C:\> cd doit
C:\doit> python myargv.py 1 2 3 4 5 6 7 8 9 10
55
```

[문제2] os

os 모듈을 이용하여 다음과 같이 동작하도록 코드를 작성해 보자.

1. C:\doit이라는 디렉토리로 이동한다.
2. dir 명령을 실행하고 그 결과를 변수에 담는다.
3. dir 명령의 결과를 출력한다.

[문제3] glob

glob 모듈을 이용하여 C:\doit 디렉토리의 파일중 확장자가 py인 파일만 출력하는 프로그램을 작성하시오.

[문제4] time

time 모듈을 이용하여 현재 날짜와 시간을 다음과 같은 형식으로 출력하시오.

```
YYYY/MM/DD HH:mm:ss (YYYY:년도, MM:월, DD:일, HH:24시간 기준시간, mm:분, ss:초)
```

출력 예

```
2018/04/03 17:20:32
```

[문제5] random

random모듈을 이용하여 로또번호(1~45 사이의 숫자 6개)를 생성하시오. (단, 중복된 숫자가 있으면 안됨)

[문제6] namedtuple

다음 프로그램을 Student 클래스 대신 namedtuple을 이용하여 재 구성 하시오.

```
class Student:  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
  
a = Student("김준석 | 구매하신", 30)  
  
print(a.name)  
print(a.score)
```

06장 파이썬 프로그래밍, 어떻게 시작해야 할까?

이 장에서는 아주 짤막한 스크립트와 함수들을 만들어 본다. 아마 프로그래밍 감각을 키우는 데 더할 나위 없이 좋은 재료가 될 것이다. 스크립트란 에디터로 작성한 파이썬 프로그램 파일을 말한다. 앞으로는 에디터로 작성한 파이썬 프로그램파일을 파이썬 스크립트라고 부를 것이니 혼동하지 말자. 이 장에 소개된 모든 파이썬 프로그램 예제는 대화형 인터프리터가 아닌 에디터로 작성해야 한다.

06-1 내가 프로그램을 만들 수 있을까?

프로그램을 막 시작하려는 사람이 맨 먼저 부딪히게 되는 벽은 아마도 다음과 같은 것이 아닐까?

“문법도 어느 정도 알겠고, 책 내용도 대부분 이해된다. 하지만 이러한 지식을 바탕으로 내가 도대체 어떤 프로그램을 만들 수 있을까?”

이럴 때는 “어떤 프로그램을 짜야지”라는 생각보다는 다른 사람들이 만든 프로그램 파일을 자세히 들여다보고 분석하는 데서 시작하는 것이 좋다. 그러다 보면 다른 사람들의 생각도 읽을 수 있고 거기에 더해 뭔가 새로운 아이디어가 떠오를 수도 있다. 하지만 여기서 가장 중요한 것은 자신의 수준에 맞는 소스를 찾는 일이다. 그래서 이 장에서는 아주 쉬운 예제부터 시작해서 차츰 수준을 높여 실용적인 예제까지 다루려고 노력하였다. 배운 내용을 어떻게 활용하는가는 독자의 몫이다.

예전에 프로그래밍을 막 시작한 사람에게 구구단 프로그램을 짜보라고 한 적이 있다. 쉬운 과제이고 파이썬 문법도 다 공부한 사람이었는데 프로그램을 어떻게 만들어야 할지 전혀 갈피를 잡지 못했다. 그래서 필자는 다음과 같은 해결책을 알려 주었다.

프로그램을 만들려면 가장 먼저 “입력”과 “출력”을 생각하라.

가령 구구단 프로그램 중 2단을 만든다면 2를 입력값으로 주었을 때 어떻게 출력되어야 할지 생각해 보라고 했다. 그래도 그림이 그려지지 않는 것 같아 직접 연습장에 적어 가며 설명을 해주었다.

- 함수 이름은? GuGu로 짓자
- 입력받는 값은? 2
- 출력하는 값은? 2단(2, 4, 6, 8, ..., 18)
- 결과는 어떤 형태로 저장하지? 연속된 자료형이니까 리스트!

독자들도 함께 따라 해보기 바란다.

1. 먼저 에디터를 열고 다음과 같이 입력한다. GuGu라는 함수에 2를 입력값으로 주면 result라는 변수에 결과값을 넣으라는 뜻이다.

```
result = GuGu(2)
```

2. 이제 결과값을 어떤 형태로 받을 것인지 고민해 보자. 2단이니까 2, 4, 6, ... 18까지 같 것이다. 이런 종류의 데이터는 리스트 자료형이 딱이다. 따라서 `result = [2, 4, 6, 8, 10, 12, 14, 16, 18]`과 같은 결과를 얻는 것이 좋겠다는 생각을 먼저 하고 나서 프로그래밍을 시작하는 것이 필요하다. 이런 식으로 머릿속에 그림이 그려지기 시작하면 의외로 생각이 가볍게 풀려지는 것을 느낄 수 있을 것이다.

3. 자, 일단 이름을 GuGu로 지은 함수를 다음과 같이 만들어 보자.

```
def GuGu(n):
    print(n)
```

위와 같은 함수를 만들고 GuGu(2)처럼 실행하면 2를 출력하게 된다. 즉, 입력값으로 2가 잘 들어오는지 확인하는 것이다.

4. 이제 결과값을 담을 리스트를 하나 생성하자. 앞에서 작성한 print(n)은 입력이 잘 되는지를 확인하기 위한 것이었으므로 지워도 좋다.

```
def GuGu(n):
    result = []
```

5. 다음으로 result에 2, 4, 6, ..., 18을 어떻게 넣어야 할지 생각해 보자. 필자는 다음과 같이 리스트에 요소를 추가하는 append 내장 함수를 사용하기로 결정했다.

```
def GuGu(n):
    result = []
    result.append(n*1)
    result.append(n*2)
    result.append(n*3)
    result.append(n*4)
    result.append(n*5)
    result.append(n*6)
    result.append(n*7)
    result.append(n*8)
    result.append(n*9)
    return result

print(GuGu(2))
```

결과값: [2, 4, 6, 8, 10, 12, 14, 16, 18]

정말 무식한 방법이지만 입력값 2를 주었을 때 원하는 결과값을 얻을 수 있었다.

6. 그런데 위의 함수는 반복이 너무 많다. 가만히 보면 result.append(n*□)의 □위치에 1부터 9까지 숫자만 다르게 들어가 있다는 것을 알 수 있다. 똑같은 일을 반복할 때는 “반복문”을 사용한다고 했다. 그렇다면 1부터 9까지 출력해 주는 반복문을 만들면 되지 않을까?

대화형 인터프리터를 열고 다음과 같이 테스트해 보았다.

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

결과값:

```
1
2
3
4
5
6
7
8
9
```

아주 만족스러운 결과다. 이제 위와 같은 소스 코드를 GuGu 함수에 적용시켜 보자.

7. 이상의 생각들을 바탕으로 완성한 GuGu 함수는 다음과 같다.

```
def GuGu(n):
    result = []
    i = 1
    while i < 10:
        result.append(n * i)
        i = i + 1
    return result
```

다음과 같이 테스트해 보자.

```
print(GuGu(2))
```

결과값: [2, 4, 6, 8, 10, 12, 14, 16, 18]

결과는 대만족이다.

사실 GuGu 함수는 위와 같은 과정을 거치지 않고도 금방 만들 수 있는 독자들이 많을 것이다. 하지만 더 복잡한 함수를 만들 때는 위와 같이 구체적이고 단계적으로 접근하는 방식이 매우 도움

이 된다. 프로그래밍을 할 땐 매우 구체적으로 접근해야 머리가 덜 아프다는 것을 기억하자.
자, 이제 다양한 예제들을 접해 보며 여러분 나름대로 멋진 생각들을 해보기 바란다.

06-2 3과 5의 배수 합하기

자, 다음 문제를 어떻게 풀면 좋을지 생각해 보자.

10 미만의 자연수에서 3과 5의 배수를 구하면 3, 5, 6, 9이다. 이들의 총합은 23이다.

1000 미만의 자연수에서 3의 배수와 5의 배수의 총합을 구하라.

- 입력 받는 값은? 1부터 999까지(1000 미만의 자연수)
- 출력하는 값은? 3의 배수와 5의 배수의 총합
- 생각해 볼 것은?
 1. 3의 배수와 5의 배수는 어떻게 찾지?
 2. 3의 배수와 5의 배수가 겹칠 때는 어떻게 하지?

이 문제를 풀기 위한 중요 포인트는 두 가지이다. 한 가지는 1000 미만의 자연수를 구하는 방법이고 또 다른 한 가지는 3과 5의 배수를 구하는 것이다. 이 두 가지만 해결되면 문제는 쉽게 해결될 것으로 보인다.

1. 먼저 1000 미만의 자연수는 어떻게 구할 수 있을지 생각해 보자. 여러 가지 방법이 떠오를 것이다. 다음과 같이 변수에 초기값 1을 준 후 루프를 돌리며 1씩 증가시켜서 999까지 진행하는 방법이 가장 일반적인 방법일 것이다.

```
n = 1
while n < 1000:
    print(n)
    n += 1
```

또는 다음과 같이 좀 더 파이썬다운 range함수를 사용할 수도 있다.

```
for n in range(1, 1000):
    print(n)
```

두 가지 예 모두 실행하면 1부터 999까지 출력하는 것을 확인할 수 있다.

2. 1000까지의 자연수를 차례로 구하는 방법을 알았으니 3과 5의 배수를 구하는 방법을 알아보자. 1000 미만의 자연수 중 3의 배수는 다음과 같이 증가할 것이다.

3, 6, 9, 12, 15, 18, ..., 999

그렇다면 1부터 1000까지 수가 진행되는 동안 그 수가 3의 배수인지는 어떻게 알 수 있을까? 1부터 1000까지의 수 중 3으로 나누었을 때 나누어떨어지는 경우, 즉 3으로 나누었을 때 나머지가 0인 경우가 바로 3의 배수이다. 따라서 다음과 같이 % 연산자를 이용하면 3의 배수를 쉽게 찾을 수 있다.

```
for n in range(1, 1000):
    if n % 3 == 0:
        print(n)
```

그렇다면 5의 배수는 $n \% 5$ 가 0이 되는 수로 구할 수 있을 것이다.

3. 이러한 내용을 바탕으로 만들어진 최종 풀이는 다음과 같다.

```
result = 0
for n in range(1, 1000):
    if n % 3 == 0 or n % 5 == 0:
        result += n
print(result)
```

3과 5의 배수에 해당하는 수를 result 변수에 계속해서 더해 주었다.

이 문제에는 한가지 함정이 있는데 3으로도 5로도 나누어지는 15와 같은 수를 이중으로 더해서는 안 된다는 점이다. 따라서 15와 같이 3의 배수도 되고 5의 배수도 되는 값이 이중으로 더해지지 않기 위해 or 연산자를 사용하였다.

다음 예는 15와 같은 수를 이중으로 더하여 잘못된 결과를 출력하는 잘못된 풀이이다.

[잘못된 풀이]

```
result = 0
for n in range(1, 1000):
    if n % 3 == 0:
        result += n
    if n % 5 == 0:
        result += n
print(result)
```

[코딩 연습을 할 수 있는 사이트]

이 문제는 코딩 연습을 할 수 있는 “프로젝트 오일러”라는 사이트의 첫 번째 문제이다. 이 사이트는 첫 번째 문제부터 차례대로 풀 수 있으며 본인이 작성한 답이 맞는지 즉시 확인할 수도 있다.

The screenshot shows the 'Archives' section of the Project Euler website. At the top, there are navigation links for 'About', 'Archives' (which is highlighted in orange), 'Recent', 'News', 'Register', and 'Sign In'. Below the navigation bar, there are two small icons: a feed symbol and a lock symbol. The main content area is titled 'Problems Archives' and contains a message: 'The problems archives table shows problems 1 to 529. If you would like to tackle the 10 most recently published problems then go to Recent problems. Click the description/title of the problem to view details and submit your answer.' Below this message is a table with 14 rows, each representing a problem. The columns are labeled 'ID', 'Description / Title', and 'Solved By'. The data is as follows:

ID	Description / Title	Solved By
1	Multiples of 3 and 5	522552
2	Even Fibonacci numbers	426752
3	Largest prime factor	309015
4	Largest palindrome product	278242
5	Smallest multiple	290142
6	Sum square difference	291907
7	100001st prime	250148
8	Largest product in a series	215636
9	Special Pythagorean triplet	215818
10	Summation of primes	198170
11	Largest product in a grid	146272
12	Highly divisible triangular number	134543
13	Large sum	140813
14	Longest Collatz sequence	139100

프로젝트 오일러(<http://projecteuler.net/archives>)

06-3 게시판 페이징하기

A 씨는 게시판 프로그램을 작성하고 있다. 그런데 게시물의 총 건수와 한 페이지에 보여줄 게시물 수를 입력으로 주었을 때 총 페이지수를 출력하는 프로그램이 필요하다고 한다.

※ 이렇게 게시판의 페이지수를 보여주는 것을 “페이징”이라고 부른다.

- 함수 이름은? getTotalPage
- 입력 받는 값은? 게시물의 총 건수(m), 한 페이지에 보여줄 게시물 수(n)
- 출력하는 값은? 총 페이지수

A씨가 필요한 프로그램을 만들기 위해 입력값과 결과값이 어떻게 나와야 하는지 먼저 살펴보자. 게시물의 총 건수가 5이고 한 페이지에서 보여 줄 게시물 수가 10이면 총 페이지수는 당연히 1이 된다. 만약 게시물의 총 건수가 15이고 한 페이지에서 보여 줄 게시물 수가 10이라면 총 페이지수는 2가 될 것이다.

게시물의 총 건수(m)	페이지당 보여줄 게시물 수(n)	총 페이지 수
5	10	1
15	10	2
25	10	3
30	10	3

이 문제는 게시판 프로그램을 만들 때 가장 처음 마주치는 난관이라고 할 수 있는 총 페이지수를 구하는 문제이다. 사실 실제 업무에서 사용하는 페이징 기술은 훨씬 복잡한데 여기에서는 그중 가장 간단한 총 페이지수를 구하는 방법에 대해서만 알아보겠다.

1. 다음과 같이 총 건수(m)를 한 페이지에 보여줄 게시물 수(n)로 나누고 1을 더하면 총 페이지수를 얻을 수 있다.

$$\text{총 페이지수} = \frac{\text{총 건수}}{\text{한 페이지당 보여줄 건수}} + 1$$

2. 이러한 공식을 적용했을 경우 총 페이지수가 표의 값처럼 구해지는지 확인해 보자(m을 n으로 나눌 때 소수점 아래 자리를 버리기 위해서 / 대신 // 연산자를 사용하였다).

```

def getTotalPage(m, n):
    return m // n + 1

print(getTotalPage(5, 10))      # 1 출력
print(getTotalPage(15, 10))     # 2 출력
print(getTotalPage(25, 10))     # 3 출력
print(getTotalPage(30, 10))     # 4 출력

```

첫 번째, 두 번째, 세 번째 케이스는 공식에 맞게 결과가 출력된다. 하지만 네 번째 케이스는 총 건수가 30이고 한 페이지에 보여줄 건수가 10인데 4가 출력되어 실패해 버렸다. 잘 생각해보자. 총건수가 30이고 한 페이지에 보여줄 건수가 10이라면 당연히 총 페이지 수는 3이 되어야 한다.

3. 실패 케이스는 총 게시물 수와 한 페이지에 보여줄 게시물 수를 나눈 나머지 값이 0이 될 때 발생함을 유추할 수 있을 것이다. 이 실패 케이스를 해결하려면 다음과 같이 코드를 변경해야 한다.

```

def getTotalPage(m, n):
    if m % n == 0:
        return m // n
    else:
        return m // n + 1

print(getTotalPage(5, 10))
print(getTotalPage(15, 10))
print(getTotalPage(25, 10))
print(getTotalPage(30, 10))

```

나누었을 때 나머지가 0인 경우는 나누기의 몫만 리턴하고 그 이외의 경우에는 1을 더하여 리턴하도록 변경했다.

프로그램을 실행해 보면 모든 케이스가 원하던 결과를 출력함을 확인할 수 있다.

06-4 간단한 메모장 만들기

원하는 메모를 파일에 저장하고 추가 및 조회가 가능한 간단한 메모장을 만들어 보자.

- 필요한 기능은? 메모 추가하기, 메모 조회하기
- 입력 받는 값은? 메모 내용, 프로그램 실행 옵션
- 출력하는 값은? memo.txt

가장 먼저 해야 할 일은 메모를 추가하는 것이다. 다음과 같은 명령을 실행했을 때 메모를 추가할 수 있도록 만들어 보자.

```
python memo.py -a "Life is too short"
```

memo.py는 우리가 작성할 파이썬 프로그램명이다. -a는 이 프로그램의 실행 옵션이고 “Life is too short”는 추가할 메모 내용이 된다.

1. 우선 다음과 같이 입력으로 받은 옵션과 메모를 출력하는 코드를 작성해 보자.

```
# C:/doit/memo.py
import sys

option = sys.argv[1]
memo = sys.argv[2]

print(option)
print(memo)
```

sys.argv는 프로그램 실행 시 입력된 값을 읽어 들일 수 있는 파이썬 라이브러리이다. sys.argv[0]은 입력 받은 값 중에서 파이썬 프로그램명인 memo.py이므로 우리가 만들고자 하는 기능에는 필요 없는 값이다. 그리고 순서대로 sys.argv[1]은 프로그램 실행 옵션값이 되고 sys.argv[2]는 메모 내용이 된다.

2. memo.py를 작성했다면 다음과 같은 명령을 수행해 보자.

※ memo.py는 C:\doit 디렉터리에 저장한다

```
C:\doit>python memo.py -a "Life is too short"
-a
Life is too short
```

입력으로 전달한 옵션과 메모 내용이 그대로 출력되는 것을 확인할 수 있다.

3. 이제 입력으로 받은 메모를 파일에 쓰도록 코드를 변경해 보자.

```
# c:/doit/memo.py
import sys

option = sys.argv[1]

if option == '-a':
    memo = sys.argv[2]
    f = open('memo.txt', 'a')
    f.write(memo)
    f.write('\n')
    f.close()
```

옵션이 “-a”인 경우에만 memo값을 읽어 memo.txt 파일에 그 값을 쓰도록 했다. 여기서 메모는 항상 새로운 내용이 작성되는 것이 아니라 한 줄씩 추가되어야 하므로 파일 열기 모드를 “a”로 했다. 그리고 메모를 추가할 때마다 다음 줄에 저장되도록 하기 위해서 줄바꿈 문자(\n)도 추가로 파일에 쓰도록 했다.

4. 이제 다음과 같은 명령을 수행해 보자.

```
C:\doit>python memo.py -a "Life is too short"
C:\doit>python memo.py -a "You need python"
```

그리고 파일에 정상적으로 메모가 기입되었는지 다음과 같이 확인해 보자.

```
C:\doit>type memo.txt
Life is too short
You need python
```

추가한 메모가 정상적으로 저장된 것을 볼 수 있다.

5. 이번에는 작성한 메모를 출력하는 부분을 만들 차례이다. 메모 출력은 다음과 같이 동작하도록 만들어 보자.

```
python memo.py -v
```

메모 추가는 -a 옵션을 사용하고 메모 출력은 -v 옵션을 이용한다.

이제 메모 출력을 위해서 다음과 같이 코드를 변경해 보자.

```
# c:/doit/memo.py
import sys

option = sys.argv[1]

if option == '-a':
    memo = sys.argv[2]
    f = open('memo.txt', 'a')
    f.write(memo)
    f.write('\n')
    f.close()
elif option == '-v':
    f = open('memo.txt')
    memo = f.read()
    f.close()
    print(memo)
```

옵션으로 -v가 들어온 경우 memo.txt 파일을 읽어서 출력하도록 하였다.

6. 코드를 수정한 후 다음과 같은 명령을 수행해 보자.

```
C:\doit>python memo.py -v
Life is too short
You need python
```

입력했던 메모가 그대로 출력되는 것을 확인할 수 있다.

06-5 템을 4개의 공백으로 바꾸기

이번에는 문서 파일을 읽어서 그 문서 파일 내에 있는 템(tab)을 공백(space) 4개로 바꾸어주는 스크립트를 작성해 보자.

- 필요한 기능은? 문서 파일 읽어 들이기, 문자열 변경하기
- 입력 받는 값은? 템을 포함한 문서 파일
- 출력하는 값은? 템이 공백으로 수정된 문서 파일

다음과 같은 형식으로 프로그램이 수행되도록 만들 것이다.

```
python tabto4.py src dst
```

tabto4.py는 우리가 작성해야 할 파일명이고 src는 템을 포함하고 있는 원본 파일명이다. dst는 파일 내의 템을 공백 4개로 변환한 결과를 저장할 파일명이다.

예를 들어 a.txt라는 파일에 있는 템을 4개의 공백으로 바꾸어 b.txt라는 파일에 저장하고 싶다면 다음과 같이 수행해야 한다.

```
python tabto4.py a.txt b.txt
```

1. 우선 다음과 같이 tabto4.py 파일을 작성해 보자.

※ tabto4.py는 C:\doit 디렉터리에 저장한다.

```
# c:/doit/tabto4.py
import sys

src = sys.argv[1]
dst = sys.argv[2]

print(src)
print(dst)
```

sys.argv를 이용하여 입력값을 확인하도록 만든 코드이다.

2. 다음과 같이 수행했을 때 입력값들이 정상적으로 출력되는지 확인해 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
C:\doit>python tabto4.py a.txt b.txt
a.txt
b.txt
```

입력으로 전달한 a.txt와 b.txt가 정상적으로 출력되는 것을 확인할 수 있다.

3. 테스트를 위한 원본 파일(탭을 포함하는 파일)인 a.txt를 다음과 같이 작성한다. 각 단어들은 템(\t) 문자로 분리되도록 입력해야 한다.

```
Life    is    too    short
You    need    python
```

4. 이제 템 문자를 포함한 a.txt 파일을 읽어서 템을 공백 4개로 변환할 수 있도록 코드를 변경해 보자.

```
# c:/doit/tabto4.py
import sys

src = sys.argv[1]
dst = sys.argv[2]

f = open(src)
tab_content = f.read()
f.close()

space_content = tab_content.replace("\t", " "*4)
print(space_content)
```

위 코드는 src에 해당되는 입력 파일을 읽어서 그 내용을 tab_content라는 변수에 저장한 후 문자열의 replace 함수를 이용하여 템(\t)을 4개의 공백으로 변경하는 코드이다.

5. tabto4.py를 위와 같이 변경한 후 다음과 같은 명령을 수행해 보자.

```
C:\doit>python tabto4.py a.txt b.txt
Life    is    too    short
You    need    python
```

아마도 텨 문자가 공백 4개로 변경되어 출력될 것이다. 하지만 텨과 공백의 차이점을 눈으로 확인할 수는 없으므로 텨이 정상적으로 공백으로 변경되었는지 확인하기 어렵다.

6. 이제 변경된 내용을 b.txt 파일에 저장할 수 있도록 다음과 같이 프로그램을 변경해 보자.

```
# c:/doit/tabto4.py

import sys

src = sys.argv[1]
dst = sys.argv[2]

f = open(src)
tab_content = f.read()
f.close()

space_content = tab_content.replace("\t", " "*4)

f = open(dst, 'w')
f.write(space_content)
f.close()
```

탭이 공백으로 변경된 space_content를 출력 파일인 dst에 쓰도록 코드를 수정하였다.

7. 프로그램을 실행하기 위해 다음과 같은 명령을 수행한다.

```
C:\doit>python tabto4.py a.txt b.txt
```

위 명령을 수행하면 b.txt 파일이 C:\doit 디렉터리에 생성된다. 에디터로 b.txt 파일을 열어서 텨이 4개의 공백 문자로 변경되었는지 확인해 보자. 프로그램 작성 시 사용하는 에디터는 대부분 텨과 공백 문자를 다르게 표시하므로 눈으로 확인이 가능할 것이다.

06-6 하위 디렉터리 검색하기

특정 디렉터리부터 시작해서 그 하위의 모든 파일 중 파이썬 파일(*.py)만 출력해 주는 프로그램을 만들려면 어떻게 해야 할까?

1. 우선 다음과 같이 sub_dir_search.py 파일을 작성해 보자.

※ sub_dir_search.py는 C:\doit 디렉터리에 저장한다.

```
# C:/doit/sub_dir_search.py

def search(dirname):
    print (dirname)

search("c:/")
```

search라는 함수를 만들고 시작 디렉터리를 입력 받도록 작성했다.

2. 이제 이 디렉터리에 있는 파일들을 검색할 수 있도록 소스를 변경해 보자.

```
# C:/doit/sub_dir_search.py
import os

def search(dirname):
    filenames = os.listdir(dirname)
    for filename in filenames:
        full_filename = os.path.join(dirname, filename)
        print (full_filename)

search("c:/")
```

os.listdir을 이용하면 해당 디렉터리에 있는 파일들의 리스트를 구할 수 있다. 여기서 구해지는 파일 리스트는 파일명만 포함되어 있으므로 경로를 포함한 파일명을 구하기 위해서는 입력으로 받은 dirname을 앞에 덧붙여 주어야 한다. os 모듈에는 디렉터리와 파일명을 이어주는 os.path.join이라는 함수가 있으므로 이 함수를 이용하면 디렉터리를 포함한 전체 경로를 쉽게 구할 수 있다.

위 코드를 수행하면 C:/ 디렉터리에 있는 파일들이 다음과 비슷하게 출력될 것이다.

[디렉토리 출력 예]

```
c::$/Recycle.Bin
c::$/WINDOWS.^BT
c::$/Windows.^WS
c::/adb
c::/AMD
c::/android
c::/bootmgr
c::/BOOTNXT
... 생략 ...
```

3. 이제 C:/ 디렉터리에 있는 파일들 중 확장자가 .py인 파일들만을 출력하도록 코드를 변경해 보자.

```
# C:/doit/sub_dir_search.py
import os

def search(dirname):
    filenames = os.listdir(dirname)
    for filename in filenames:
        full_filename = os.path.join(dirname, filename)
        ext = os.path.splitext(full_filename)[-1]
        if ext == '.py':
            print(full_filename)

search("c:/")
```

파일명에서 확장자만 추출하기 위해 os 모듈의 os.path.splitext 함수를 사용하였다. os.path.splitext 는 파일명을 확장자를 기준으로 두 부분으로 나누어 준다. 따라서 os.path.splitext(full_filename)[-1]은 해당 파일의 확장자명이 된다. 위 코드는 확장자명이 .py인 경우만을 출력하도록 했다. C:/ 디렉터리에 파일이 없다면 아무것도 출력되지 않을 것이다.

4. 하지만 우리가 원하는 것은 C:/ 디렉터리 바로 밑에 있는 파일 뿐만 아니라 그 하위 디렉터리 (sub directory)를 포함한 모든 파일들을 검색하는 것이다. 하위 디렉터리도 검색이 가능하게 하기 위해서는 다음과 같이 코드를 변경해야 한다.

```
# C:/doit/sub_dir_search.py
import os

def search(dirname):
    try:
        filenames = os.listdir(dirname)
        for filename in filenames:
            full_filename = os.path.join(dirname, filename)
            if os.path.isdir(full_filename):
                search(full_filename)
            else:
                ext = os.path.splitext(full_filename)[-1]
                if ext == '.py':
                    print(full_filename)
    except PermissionError:
        pass

search("c:/")
```

try ... except PermissionError로 함수 전체를 감싼 이유는 os.listdir 수행 시 권한이 없는 디렉터리에 접근하더라도 프로그램이 오류로 종료되지 않고 그냥 수행되도록 하기 위해서이다.

full_filename이 디렉터리인지 파일인지 구분하기 위하여 os.path.isdir 함수를 이용하였고 디렉터리일 경우 해당 경로를 입력으로 받아 다시 search함수를 호출하도록 하였다. 이렇게 해당 디렉터리의 파일이 디렉터리일 경우 다시 search 함수를 호출해 나가게 되면 (재귀 호출) 해당 디렉터리의 하위 파일들을 다시 검색하기 시작하므로 결국 모든 파일들을 검색 할 수 있게 된다.

※ 재귀 호출이란 자기 자신을 다시 호출하는 프로그래밍 기법이다. 이 코드에서 보면 search 함수에서 다시 자기 자신인 search함수를 호출하는 것이 바로 재귀호출이다.

위 코드를 수행하면 C:/ 디렉터리에 있는 모든 파일들이 출력될 것이다.

[하위 디렉터리 검색을 쉽게 해주는 os.walk]

os.walk를 이용하면 위에서 작성한 코드를 보다 간편하게 만들 수 있다. os.walk는 시작 디렉터리부터 시작하여 그 하위의 모든 디렉터리를 차례대로 방문하게 해주는 함수이다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
import os

for (path, dir, files) in os.walk("c:/"):
    for filename in files:
        ext = os.path.splitext(filename)[-1]
        if ext == '.py':
            print("%s/%s" % (path, filename))
```

디렉터리와 파일을 검색하는 일반적인 경우라면 os.walk를 사용하는 것을 추천한다.

06-7 코딩도장

다음은 “위대한 프로그래머가 되려면 어떻게 해야할까?”라는 질문에 대한 “워드 커닝햄”의 답변이다.

저는 작지만 유용한 프로그램들을 매일 작성할 것을 추천합니다. 누군가가 똑같거나 혹은 더 나은 걸 이미 만들었다는 데에 절대 신경쓰지 마세요. 유용성과 복잡성 간의 균형 감각을 얻기 위해서는 당신 자신이 만든 프로그램의 유용성을 직접 느껴봐야만 합니다.

- 워드 커닝햄 (김창준씨와의 인터뷰중에서)

“워드 커닝햄”의 말을 실천할 수 있는 방법 중 하나로 위키독스의 자매사이트인 코딩도장을 소개한다. 코딩도장은 프로그래밍 문제풀이를 통해서 코딩 실력을 수련(Practice)하는 곳이다.

- 코딩도장 : <http://codingdojang.com>

이 곳에서 쉬운문제부터 천천히 풀어보도록 하자.

07장 유용한 파이썬 도구들

이번 장에서는 조금 어려운 내용을 살펴보려고 한다. 사실 파이썬 입문자에게는 어울리지 않는 내용일 수도 있다. 그럼에도 불구하고 이곳에 소개하는 이유는 이곳에 소개한 것들은 한 번 익혀두면 두고두고 써먹을 수 있는 것들이기 때문이다.

다시 말하지만 프로그래밍 입문자가 이해하기에는 어려운 내용이니 부담 갖지 말고 편하게 읽어주기 바란다.

07-1 정규 표현식 살펴보기

정규 표현식(Regular Expressions)은 복잡한 문자열을 처리할 때 사용하는 기법으로, 파이썬만의 고유 문법이 아니라 문자열을 처리하는 모든 곳에서 사용된다. 정규 표현식을 배우는 것은 파이썬을 배우는 것과는 또 다른 영역의 과제이다.

※ 정규 표현식은 줄여서 간단히 “정규식”이라고도 말한다.

필자는 “정규 표현식”을 이 책 《점프 투 파이썬》에 포함시켜야 할지 오랜 시간 고민했다. 왜냐하면 정규 표현식은 꽤 오랜 기간 코드를 작성해 온 프로그래머라도 잘 모를 수 있는 고급 주제여서 초보자를 대상으로 하는 이 책에는 어울리지 않기 때문이다.

하지만 정규 표현식을 배워 익히기만 하면 아주 달콤한 열매를 맛볼 수 있다. 그래서 파이썬 하우 튜(<https://docs.python.org/3.6/howto/regex.html>)를 참고하여 그곳에서 소개하는 수준의 내용만이라도 독자들이 이해하고 사용할 수 있도록 노력했다. 여러분이 정규 표현식을 잘 다루게 되면 파이썬 외에 또 하나의 강력한 무기를 얻게 되는 것이다.

정규 표현식은 왜 필요한가?

다음과 같은 문제가 주어졌다고 가정해 보자.

주민등록번호를 포함하고 있는 텍스트가 있다. 이 텍스트에 포함된 모든 주민등록번호의 뒷자리를 * 문자로 변경하시오.

우선, 정규식을 전혀 모르면 다음과 같은 순서로 프로그램을 작성해야 할 것이다.

1. 전체 텍스트를 공백 문자로 나눈다(split).
2. 나누어진 단어들이 주민등록번호 형식인지 조사한다.
3. 단어가 주민등록번호 형식이라면 뒷자리를 *로 변환한다.
4. 나누어진 단어들을 다시 조립한다.

이를 구현한 코드는 아마도 다음과 같을 것이다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```

data = """
park 800905-1049118
kim 700905-1059119
"""

result = []
for line in data.split("\n"):
    word_result = []
    for word in line.split(" "):
        if len(word) == 14 and word[:6].isdigit() and word[7:].isdigit():
            word = word[:6] + "-" + "*****"
        word_result.append(word)
    result.append(" ".join(word_result))
print("\n".join(result))

```

결과값:

```

park 800905-*****
kim 700905-*****

```

반면에, 정규식을 사용하면 다음처럼 훨씬 간편하고 직관적인 코드를 작성할 수 있다. 아직 정규식 사용 방법을 배우지 않았으니 눈으로만 살펴보자.

```

import re

data = """
park 800905-1049118
kim 700905-1059119
"""

pat = re.compile("(\\d{6})[-]\\d{7}")
print(pat.sub("\g<1>-*****", data))

```

결과값:

```

park 800905-*****
kim 700905-*****

```

정규 표현식을 사용하면 이렇게 간단한 예제에서도 코드가 상당히 간결해진다. 만약 찾고자하는 문자열 또는 바꾸어야 할 문자열의 규칙이 매우 복잡하다면 정규식의 효용은 더 커지게 된다.

이제부터 정규 표현식의 기초부터 심화 부분까지 차근차근 알아보도록 하자.

07-2 정규 표현식 시작하기

정규 표현식의 기초, 메타 문자

정규 표현식에서 사용하는 메타 문자(meta characters)에는 다음과 같은 것들이 있다.

※ 메타 문자란 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용되는 문자를 말한다.

```
. ^ $ * + ? { } [ ] \ | ( )
```

정규 표현식에 위 메타 문자들이 사용되면 특별한 의미를 갖게 된다.

자, 그럼 가장 간단한 정규 표현식부터 시작해 각 메타 문자의 의미와 사용법을 알아보자.

문자 클래스 []

우리가 가장 먼저 살펴 볼 메타 문자는 바로 문자 클래스(character class)인 []이다. 문자 클래스로 만들어진 정규식은 “[와] 사이의 문자들과 매치”라는 의미를 갖는다.

※ 문자 클래스를 만드는 메타 문자인 [와] 사이에는 어떤 문자도 들어갈 수 있다.

즉, 정규 표현식이 [abc]라면 이 표현식의 의미는 “a, b, c 중 한 개의 문자와 매치”를 뜻한다. 이해를 돋기 위해 문자열 “a”, “before”, “dude”가 정규식 [abc]와 어떻게 매치되는지 살펴보자.

- “a”는 정규식과 일치하는 문자인 “a”가 있으므로 매치
- “before”는 정규식과 일치하는 문자인 “b”가 있으므로 매치
- “dude”는 정규식과 일치하는 문자인 a, b, c 중 어느 하나도 포함하고 있지 않으므로 매치되지 않음

[] 안의 두 문자 사이에 하이픈(-)을 사용하게 되면 두 문자 사이의 범위(From - To)를 의미한다. 예를 들어 [a-c]라는 정규 표현식은 [abc]와 동일하고 [0-5]는 [012345]와 동일하다.

다음은 하이픈(-)을 이용한 문자 클래스의 사용 예이다.

- [a-zA-Z] : 알파벳 모두
- [0-9] : 숫자

문자 클래스([]) 내에는 어떤 문자나 메타 문자도 사용할수 있지만 주의해야 할 메타 문자가 1 가지 있다. 그것은 바로 ^인데, 문자 클래스 내에 ^ 메타 문자가 사용될 경우에는 반대(not)라는 의미를 갖는다. 예를 들어 [^0-9]라는 정규 표현식은 숫자가 아닌 문자만 매치된다.

[자주 사용하는 문자 클래스]

[0-9] 또는 [a-zA-Z] 같은 무척 자주 사용하는 정규 표현식이다. 이렇게 자주 사용하는 정규식들은 별도의 표기법으로 표현할 수 있다. 다음을 기억해 두자.

- \d - 숫자와 매치, [0-9]와 동일한 표현식이다.
- \D - 숫자가 아닌 것과 매치, [^0-9]와 동일한 표현식이다.
- \s - whitespace 문자와 매치, [\t\n\r\f\v]와 동일한 표현식이다. 맨 앞의 빈 칸은 공백 문자(space)를 의미한다.
- \S - whitespace 문자가 아닌 것과 매치, [^ \t\n\r\f\v]와 동일한 표현식이다.
- \w - 문자+숫자(alphanumeric)와 매치, [a-zA-Z0-9]와 동일한 표현식이다.
- \W - 문자+숫자(alphanumeric)가 아닌 문자와 매치, [^a-zA-Z0-9]와 동일한 표현식이다.

대문자로 사용된 것은 소문자의 반대임을 추측할 수 있을 것이다.

Dot(.)

정규 표현식의 Dot(.) 메타 문자는 줄바꿈 문자인 \n를 제외한 모든 문자와 매치됨을 의미한다.

※ 나중에 배우겠지만 정규식 작성 시 옵션으로 re.DOTALL이라는 옵션을 주면 \n 문자와도 매치가 된다.

다음의 정규식을 보자.

a.b

위 정규식의 의미는 다음과 같다.

“a + 모든문자 + b”

즉 a와 b라는 문자 사이에 어떤 문자가 들어가도 모두 매치된다는 의미이다.

이해를 돋기 위해 문자열 “aab”, “a0b”, “abc”가 정규식 a.b와 어떻게 매치되는지 살펴보자.

- “aab”는 가운데 문자 “a”가 모든 문자를 의미하는 .과 일치하므로 정규식과 매치된다.
- “a0b”는 가운데 문자 “0”가 모든 문자를 의미하는 .과 일치하므로 정규식과 매치된다.
- “abc”는 “a”문자와 “b”문자 사이에 어떤 문자라도 하나는 있어야 하는 이 정규식과 일치하지 않으므로 매치되지 않는다.

※ 만약 앞에서 살펴본 문자 클래스([]) 내에 Dot(.) 메타 문자가 사용된다면 이것은 “모든 문자”라는 의미가 아닌 문자 . 그대로를 의미한다. 혼동하지 않도록 주의하자.

다음의 정규식을 보자.

a[.]b

이 정규식의 의미는 다음과 같다.

“a + Dot(.)문자 + b”

따라서 정규식 a[.]b 는 “a.b”라는 문자열과는 매치되고 “a0b”라는 문자와는 매치되지 않는다.

※ .는 \n을 제외한 모든 문자와 매치되는데 심지어 \n문자와도 매치되게 할 수도 있다. 나중에 알아보겠지만 정규식 작성시 옵션으로 re.DOTALL 이라는 옵션을 주면 \n문자와도 매치되게 할 수 있다.

반복 (*)

다음의 정규식을 보자.

ca*t

이 정규식에는 반복을 의미하는 * 메타문자가 사용되었다. 여기서 사용된 *의 의미는 *바로 앞에 있는 문자 a가 0부터 무한대로 반복될 수 있다는 의미이다.

※ 여기서 * 메타문자의 반복갯수가 무한개까지라고 표현했는데 사실 메모리 제한으로 2억개 정도만 가능하다고 한다.

즉, 다음과 같은 문자열들이 모두 매치된다.

정규식	문자열	Match 여부	설명
ca*t	ct	Yes	“a”가 0번 반복되어 매치
ca*t	cat	Yes	“a”가 0번 이상 반복되어 매치 (1번 반복)
ca*t	caaat	Yes	“a”가 0번 이상 반복되어 매치 (3번 반복)

반복 (+)

반복을 나타내는 또 다른 메타 문자로 +가 있다. +는 최소 1번 이상 반복될 때 사용한다. 즉, * 가 반복 횟수 0부터라면 +는 반복 횟수 1부터인 것이다.

다음 정규식을 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

ca+t

위 정규식의 의미는 다음과 같다.

“c + a(1번 이상 반복) + t”

위 정규식에 대한 매치여부는 다음 표와 같다.

정규식	문자열	Match 여부	설명
ca+t	ct	No	“a”가 0번 반복되어 매치되지 않음
ca+t	cat	Yes	“a”가 1번 이상 반복되어 매치 (1번 반복)
ca+t	caaat	Yes	“a”가 1번 이상 반복되어 매치 (3번 반복)

반복 ($\{m,n\}$, ?)

여기서 잠깐 생각해 볼 게 있다. 반복 횟수를 3회만 또는 1회부터 3회까지만으로 제한하고 싶을 수도 있지 않을까?

{ } 메타 문자를 이용하면 반복 횟수를 고정시킬 수 있다. $\{m, n\}$ 정규식을 사용하면 반복 횟수가 m부터 n까지인 것을 매치할 수 있다. 또한 m 또는 n을 생략할 수도 있다. 만약 {3,} 처럼 사용하면 반복 횟수가 3 이상인 경우이고 {,3} 처럼 사용하면 반복 횟수가 3 이하인 것을 의미한다. 생략된 m은 0과 동일하며, 생략된 n은 무한대(2억개 미만)의 의미를 갖는다.

※ {1,}은 +와 동일하며 {0,}은 *와 동일하다.

{ }을 이용한 몇가지 정규식을 살펴보자.

1. $\{m\}$

ca{2}t

위 정규식의 의미는 다음과 같다.

“c + a(반드시 2번 반복) + t”

위 정규식에 대한 매치여부는 다음 표와 같다.

정규식	문자열	Match 여부	설명
ca{2}t	cat	No	“a”가 1번만 반복되어 매치되지 않음

정규식	문자열	Match 여부	설명
ca{2}t	caat	Yes	“a”가 2번 반복되어 매치

2. {m, n}

```
ca{2,5}t
```

위 정규식의 의미는 다음과 같다:

“c + a(2~5회 반복) + t”

위 정규식에 대한 매치여부는 다음 표와 같다.

정규식	문자열	Match 여부	설명
ca{2,5}t	cat	No	“a”가 1번만 반복되어 매치되지 않음
ca{2,5}t	caat	Yes	“a”가 2번 반복되어 매치
ca{2,5}t	caaaaat	Yes	“a”가 5번 반복되어 매치

3. ?

반복은 아니지만 이와 비슷한 개념으로 ? 이 있다. ? 메타문자가 의미하는 것은 {0, 1} 이다.

다음의 정규식을 보자.

```
ab?c
```

위 정규식의 의미는 다음과 같다:

“a + b(있어도 되고 없어도 된다) + c”

위 정규식에 대한 매치여부는 다음 표와 같다.

정규식	문자열	Match 여부	설명
ab?c	abc	Yes	“b”가 1번 사용되어 매치
ab?c	ac	Yes	“b”가 0번 사용되어 매치

즉, b문자가 있거나 없거나 둘 다 매치되는 경우이다.

* , + , ? 메타문자는 모두 {m, n} 형태로 고쳐쓰는 것이 가능하지만 가급적 이해하기 쉽고 표현도 간결한 * , + , ? 메타문자를 사용하는 것이 좋다.

지금까지 아주 기초적인 정규표현식에 대해서 알아보았다. 정규식에 대해서 알아야 할 것들이 아직 많이 남아 있지만 그에 앞서 파이썬으로 이러한 정규표현식을 어떻게 사용할 수 있는지에 대해서 먼저 알아보기로 하자.

파이썬에서 정규 표현식을 지원하는 re 모듈

파이썬은 정규 표현식을 지원하기 위해 re(regular expression의 약어) 모듈을 제공한다. re 모듈은 파이썬이 설치될 때 자동으로 설치되는 기본 라이브러리로, 사용 방법은 다음과 같다.

```
>>> import re
>>> p = re.compile('ab*')
```

re.compile 을 이용하여 정규표현식(위 예에서는 ab*)을 컴파일하고 컴파일된 패턴객체(re.compile 의 결과로 리턴되는 객체 p)를 이용하여 그 이후의 작업을 수행할 것이다.

- ※ 정규식 컴파일 시 특정 옵션을 주는 것도 가능한데, 이에 대해서는 뒤에서 자세히 살펴본다.
- ※ 패턴이란 정규식을 컴파일한 결과이다.

정규식을 이용한 문자열 검색

이제 컴파일 된 패턴 객체를 이용하여 검색을 수행 해 보자. 컴파일 된 패턴 객체는 다음과 같은 4가지 메소드를 제공한다.

Method	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사한다.
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사한다.
findall()	정규식과 매치되는 모든 문자열(substring)을 리스트로 리턴한다
finditer()	정규식과 매치되는 모든 문자열(substring)을 iterator 객체로 리턴한다

match, search는 정규식과 매치될 때에는 match 객체를 리턴하고 매치되지 않을 경우에는 None

을 리턴한다. 이 메써드들에 대한 간단한 예를 살펴보자.

※ match 객체란 정규식의 검색 결과로 리턴되는 객체이다.

우선 다음과 같은 패턴을 만들어 보자.

```
>>> import re
>>> p = re.compile('[a-z]+')
```

match

match 메서드는 문자열의 처음부터 정규식과 매치되는지 조사한다. 위 패턴에 match 메서드를 수행해 보자.

```
>>> m = p.match("python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3F9F8>
```

“python”이라는 문자열은 [a-z]+ 정규식에 부합되므로 match 객체가 리턴된다.

```
>>> m = p.match("3 python")
>>> print(m)
None
```

“3 python”이라는 문자열은 처음에 나오는 3이라는 문자가 정규식 [a-z]+에 부합되지 않으므로 None이 리턴된다.

match의 결과로 match 객체 또는 None이 리턴되기 때문에 파이썬 정규식 프로그램은 보통 다음과 같은 흐름으로 작성한다.

```
p = re.compile(정규표현식)
m = p.match('string goes here')
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

즉, match의 결과값이 있을 때만 그 다음 작업을 수행하겠다는 의도이다.

search

컴파일된 패턴 객체 p를 가지고 이번에는 search 메서드를 수행해 보자.

```
>>> m = p.search("python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3FA68>
```

“python”이라는 문자열에 search 메서드를 수행하면 match 메서드를 수행했을 때와 동일하게 매치된다.

```
>>> m = p.search("3 python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3FA30>
```

“3 python”이라는 문자열의 첫 번째 문자는 “3”이지만 search는 문자열의 처음부터 검색하는 것 이 아니라 문자열 전체를 검색하기 때문에 “3” 이후의 “python”이라는 문자열과 매치된다.

이렇듯 match 메서드와 search 메서드는 문자열의 처음부터 검색할지의 여부에 따라 다르게 사용 해야 한다.

findall

이번에는 findall 메서드를 수행해 보자.

```
>>> result = p.findall("life is too short")
>>> print(result)
['life', 'is', 'too', 'short']
```

“life is too short”라는 문자열의 “life”, “is”, “too”, “short”라는 단어들이 각각 [a-z]+ 정규식 과 매치되어 리스트로 리턴된다.

finditer

이번에는 finditer 메서드를 수행해 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> result = p.finditer("life is too short")
>>> print(result)
<callable_iterator object at 0x01F5E390>
>>> for r in result: print(r)
...
<_sre.SRE_Match object at 0x01F3F9F8>
<_sre.SRE_Match object at 0x01F3FAD8>
<_sre.SRE_Match object at 0x01F3FAA0>
<_sre.SRE_Match object at 0x01F3F9F8>
```

finditer는 findall과 동일하지만 그 결과로 반복 가능한 객체(iterator object)를 리턴한다. 반복 가능한 객체가 포함하는 각각의 요소는 match 객체이다.

match 객체의 메서드

자, 이젠 match 메서드와 search 메서드를 수행한 결과로 리턴되었던 match 객체에 대해서 알아보자. 앞에서 정규식을 이용한 문자열 검색을 수행하면서 아마도 다음과 같은 궁금증이 생겼을 것이다.

- 어떤 문자열이 매치되었는가?
- 매치된 문자열의 인덱스는 어디서부터 어디까지인가?

match 객체의 메서드들을 이용하면 이 같은 궁금증을 해결할 수 있다. 다음 표를 보자.

method	목적
group()	매치된 문자열을 리턴한다.
start()	매치된 문자열의 시작 위치를 리턴한다.
end()	매치된 문자열의 끝 위치를 리턴한다.
span()	매치된 문자열의 (시작, 끝)에 해당되는 튜플을 리턴한다.

다음의 예로 확인해 보자.

```
>>> m = p.match("python")
>>> m.group()
'python'
>>> m.start()
0
>>> m.end()
6
>>> m.span()
(0, 6)
```

예상한 대로 결과값이 출력되는 것을 확인할 수 있다. `match` 메서드를 수행한 결과로 리턴된 `match` 객체의 `start()`의 결과값은 항상 0일 수밖에 없다. 왜냐하면 `match` 메서드는 항상 문자열의 시작부터 조사하기 때문이다.

만약 `search` 메서드를 사용했다면 `start()`의 값은 다음과 같이 다르게 나올 것이다.

```
>>> m = p.search("3 python")
>>> m.group()
'python'
>>> m.start()
2
>>> m.end()
8
>>> m.span()
(2, 8)
```

[모듈 단위로 수행하기]

지금까지 우리는 `re.compile`을 이용하여 컴파일된 패턴 객체로 그 이후 작업을 수행했다. `re` 모듈은 이 것을 좀 축약한 형태의 방법을 제공한다. 다음의 예를 보자.

```
>>> p = re.compile('[a-z]+')
>>> m = p.match("python")
```

위 코드가 축약된 형태는 다음과 같다.

```
>>> m = re.match('[a-z]+', "python")
```

위 예처럼 사용하면 컴파일과 match 메서드를 한 번에 수행할 수 있다. 보통 한 번 만든 패턴 객체를 여러번 사용해야 할 때는 이 방법보다 re.compile을 사용하는 것이 유리하다.

컴파일 옵션

정규식을 컴파일할 때 다음과 같은 옵션을 사용할 수 있다.

- DOTALL(S) - . 이 줄바꿈 문자를 포함하여 모든 문자와 매치할 수 있도록 한다.
- IGNORECASE(I) - 대소문자에 관계없이 매치할 수 있도록 한다.
- MULTILINE(M) - 여러줄과 매치할 수 있도록 한다. (^, \$ 메타문자의 사용과 관계가 있는 옵션이다)
- VERBOSE(X) - verbose 모드를 사용할 수 있도록 한다. (정규식을 보기 편하게 만들수 있고 주석등을 사용할 수 있게된다.)

옵션을 사용할 때는 re.DOTALL처럼 전체 옵션명을 써도 되고 re.S처럼 약어를 써도 된다.

DOTALL, S

. 메타 문자는 줄바꿈 문자(\n)를 제외한 모든 문자와 매치되는 규칙이 있다. 만약 \n 문자도 포함하여 매치하고 싶다면 re.DOTALL 또는 re.S 옵션을 사용해 정규식을 컴파일하면 된다.

다음의 예를 보자.

```
>>> import re
>>> p = re.compile('a.b')
>>> m = p.match('a\nb')
>>> print(m)
None
```

정규식이 a.b인 경우 문자열 a\nb는 매치되지 않음을 알 수 있다. 왜냐하면 \n은 . 메타 문자와 매치되지 않기 때문이다. \n 문자와도 매치되게 하려면 다음과 같이 re.DOTALL 옵션을 사용해야 한다.

```
>>> p = re.compile('a.b', re.DOTALL)
>>> m = p.match('a\nb')
>>> print(m)
<_sre.SRE_Match object at 0x01FCF3D8>
```

보통 `re.DOTALL`은 여러줄로 이루어진 문자열에서 `\n`에 상관없이 검색하고자 할 경우에 많이 사용한다.

IGNORECASE, I

`re.IGNORECASE` 또는 `re.I` 는 대소문자 구분없이 매치를 수행하고자 할 경우에 사용하는 옵션이다.

다음의 예제를 보자.

```
>>> p = re.compile('[a-z]', re.I)
>>> p.match('python')
<_sre.SRE_Match object at 0x01FCFA30>
>>> p.match('Python')
<_sre.SRE_Match object at 0x01FCFA68>
>>> p.match('PYTHON')
<_sre.SRE_Match object at 0x01FCF9F8>
```

[a-z] 정규식은 소문자만을 의미하지만 `re.I` 옵션에 의해서 대·소문자 구분 없이 매치된다.

MULTILINE, M

`re.MULTILINE` 또는 `re.M` 옵션은 조금 후에 설명할 메타 문자인 ^, \$와 연관된 옵션이다. 이 메타 문자에 대해 간단히 설명하자면 ^는 문자열의 처음을 의미하고, \$은 문자열의 마지막을 의미한다. 예를 들어 정규식이 ^python인 경우 문자열의 처음은 항상 python으로 시작해야 매치되고, 만약 정규식이 python\$이라면 문자열의 마지막은 항상 python으로 끝나야 매치가 된다는 의미이다.

다음 예를 보도록 하자.

```

import re
p = re.compile("^python\s\w+")

data = """python one
life is too short
python two
you need python
python three"""

print(p.findall(data))

```

정규식 `^python\s\w+` 은 “python”이라는 문자열로 시작하고 그 후에 whitespace, 그 후에 단어가 와야한다는 의미이다. 검색할 문자열 `data`는 여러줄로 이루어져 있다.

이 스크립트를 실행하면 다음과 같은 결과가 리턴된다.

```
[‘python one’]
```

~ 메타 문자에 의해 `python`이라는 문자열이 사용된 첫 번째 라인만 매치가 된 것이다.

하지만 ~ 메타 문자를 문자열 전체의 처음이 아니라 각 라인의 처음으로 인식시키고 싶은 경우도 있을 것이다. 이럴 때 사용할 수 있는 옵션이 바로 `re.MULTILINE` 또는 `re.M`이다. 위 코드를 다음과 같이 수정해 보자.

```

import re
p = re.compile("^python\s\w+", re.MULTILINE)

data = """python one
life is too short
python two
you need python
python three"""

print(p.findall(data))

```

`re.MULTILINE` 옵션으로 인해 ~ 메타 문자가 문자열 전체가 아닌 각 라인의 처음이라는 의미를 갖게 되었다. 이 스크립트를 실행하면 다음과 같은 결과가 출력된다.

```
[‘python one’, ‘python two’, ‘python three’]
```

즉, re.MULTILINE 옵션은 ^, \$메타 문자를 문자열의 각 라인마다 적용해 주는 것이다.

VERBOSE, X

지금껏 알아본 정규식은 매우 간단하지만 정규식 전문가들이 만든 정규식을 보면 거의 암호수준이다. 이해하려면 하나하나 조심스럽게 뜯어보아야만 한다. 이렇게 이해하기 어려운 정규식을 주석 또는 라인 단위로 구분할 수 있다면 얼마나 보기 좋고 이해하기 쉬울까? 방법이 있다. 바로 re.VERBOSE 또는 re.X 옵션을 이용하면 된다.

다음의 예를 보자.

```
charref = re.compile(r'&[#](0[0-7]+|[0-9]+|x[0-9a-fA-F]+);')
```

위 정규식이 쉽게 이해되는가? 이제 다음의 예를 보자.

```
charref = re.compile(r"""
&#[#]           # Start of a numeric entity reference
(
    0[0-7]+      # Octal form
    | [0-9]+      # Decimal form
    | x[0-9a-fA-F]+ # Hexadecimal form
)
;               # Trailing semicolon
""", re.VERBOSE)
```

첫 번째와 두 번째 예를 비교해 보면 컴파일된 패턴 객체인 charref는 모두 동일한 역할을 한다. 하지만 정규식이 복잡할 경우 두 번째처럼 주석을 적고 여러 줄로 표현하는 것이 훨씬 가독성이 좋다는 것을 알 수 있다.

re.VERBOSE 옵션을 사용하면 문자열에 사용된 whitespace는 컴파일 시 제거된다(단 [] 내에 사용된 whitespace는 제외). 그리고 줄 단위로 #기호를 이용하여 주석문을 작성할 수 있다.

백슬래시 문제

정규표현식을 파이썬에서 사용하려 할 때 혼란을 주게 되는 요소가 한가지 있는데 그것은 바로 백슬래시(\)이다.

예를들어 LaTex파일 내에 있는 “\section”이라는 문자열을 찾기 위한 정규식을 만든다고 가정해 보자.

다음과 같은 정규식을 생각해 보자.

```
\section
```

이 정규식은 \s 문자가 whitespace로 해석되어 의도한 대로 매치가 이루어지지 않는다.

위 표현은 다음과 동일한 의미가 된다.

```
[ \t\n\r\f\v]ection
```

따라서 위 정규표현식은 다음과 같이 변경되어야 할 것이다.

```
\\"section
```

즉, 위 정규식에서 사용한 \ 문자가 문자열 그 자체임을 알려주기 위해 백슬래시 2개를 사용하여 이스케이프 처리를 해야 하는 것이다.

따라서 위 정규식을 컴파일하려면 다음과 같이 작성해야 한다.

```
>>> p = re.compile('\'\\section\'')
```

그런데 여기 또 하나의 문제가 발견된다. 위처럼 정규식을 만들어서 컴파일 하면 실제 파이썬 정규식 엔진에는 파이썬 문자열 리터럴 규칙에 의하여 \\이 \로 변경되어 \section 이 전달되는 것이다.

※ 이 문제는 위와같은 정규식을 파이썬에서 사용할 때만 적용되는 문제이다. (파이썬의 리터럴 규칙) 유닉스의 grep, vi등에서는 이러한 문제가 없다.

결국 정규식 엔진에 \\ 문자를 전달하려면 파이썬은 \\\\"처럼 백슬래시를 4개나 사용해야 한다.

※ 정규식 엔진은 정규식을 해석하고 수행하는 모듈이다.

```
>>> p = re.compile('\\\\\\section')
```

이렇게 해야만 원하는 결과를 얻을 수 있을 것이다. 하지만 너무 복잡하지 않은가?

만약 위와 같이 \를 이용한 표현이 반복되어서 사용되는 정규식이라면 너무 복잡하여 이해하기 쉽지 않을 것이다. 이러한 문제로 파이썬 정규식에는 Raw string^o이라는 것이 생겨나게 되었다. 즉 캠 파일 해야 하는 정규식이 Raw String임을 알려줄 수 있도록 파이썬 문법이 만들어진 것이다. 그 방법은 다음과 같다.

```
>>> p = re.compile(r'\\section')
```

위와 같이 정규식 문자열 앞에 r문자를 선행하면 이 정규식은 Raw String 규칙에 의하여 백슬래시 2개 대신 1개만 써도 두개를 쓴것과 동일한 의미를 갖게된다.

※ 만약 백슬래시를 사용하지 않는 정규식이라면 r의 유무에 상관없이 동일한 정규식이 될 것이다.

07-3 강력한 정규 표현식의 세계로

이제 07-2절에서 배우지 않은 몇몇 메타 문자들의 의미를 살펴보고 그룹(Group)을 만드는법, 전방 탐색 등 더욱 강력한 정규 표현식에 대해서 살펴보자.

메타문자

아직 살펴보지 않은 메타 문자들에 대해서 모두 살펴보도록 하자. 여기서 다를 메타 문자들은 앞에서 살펴보았던 메타 문자들과 성격이 조금 다르다. 이전에 살펴본 메타 문자들은 모두 매치되는 문자열들을 소모시킨다. 이 소모된다는 말의 의미가 조금 헷갈릴 수 있을 것이다. 문자열이 일단 소모되어 버리면 그 부분은 검색 대상에서 제외되지만 소모되지 않는 경우에는 다음에 또 다시 검색 대상이 된다고 생각하면 쉬울 것이다.

`+, *, [], {}` 등의 메타문자는 매치가 진행될 때 현재 매치되고 있는 문자열의 위치가 변경된다. (보통 소모된다고 표현한다.) 하지만 이와 달리 문자열을 소모시키지 않는 메타 문자들도 있다. 이번에는 이런 문자열 소모가 없는(zero-width assertions) 메타 문자들에 대해서 살펴보기로 하자.

|

| 메타문자는 “or”의 의미와 동일하다. `A|B` 라는 정규식이 있다면 이것은 A 또는 B라는 의미가 된다.

```
>>> p = re.compile('Crow|Servo')
>>> m = p.match('CrowHello')
>>> print(m)
<_sre.SRE_Match object; span=(0, 4), match='Crow'>
```

^
^ 메타문자는 문자열의 맨 처음과 일치함을 의미한다. 이전에 알아보았던 컴파일 옵션 `re.MULTILINE`을 사용할 경우에는 여러줄의 문자열에서는 각 라인의 처음과 일치하게 된다.

다음의 예를 보자.

```
>>> print(re.search('^Life', 'Life is too short'))
<sre.SRE_Match object at 0x01FCF3D8>
>>> print(re.search('^Life', 'My Life'))
None
```

`^Life` 정규식은 “Life”라는 문자열이 처음에 온 경우에는 매치하지만 처음 위치가 아닌 경우에는 매치되지 않음을 알 수 있다.

\$

\$ 메타문자는 ^ 메타문자의 반대의 경우이다. \$는 문자열의 끝과 매치함을 의미한다.

다음의 예를 보자.

```
>>> print(re.search('short$', 'Life is too short'))
<sre.SRE_Match object at 0x01F6F3D8>
>>> print(re.search('short$', 'Life is too short, you need python'))
None
```

`short$` 정규식은 검색할 문자열의 “short”로 끝난 경우에는 매치되지만 그 이외의 경우에는 매치되지 않음을 알 수 있다.

* ^ 또는 \$ 문자를 메타문자가 아닌 문자 그 자체로 매치하고 싶은 경우에는 [^], [\$] 처럼 사용하거나 \^, \\$로 사용하면 된다.

\A

\A는 문자열의 처음과 매치됨을 의미한다. ^와 동일한 의미이지만 `re.MULTILINE` 옵션을 사용할 경우에는 다르게 해석된다. `re.MULTILINE` 옵션을 사용할 경우 ^은 라인별 문자열의 처음과 매치되지만 \A는 라인과 상관없이 전체 문자열의 처음하고만 매치된다.

\Z

\Z는 문자열의 끝과 매치됨을 의미한다. 이것 역시 \A와 동일하게 `re.MULTILINE` 옵션을 사용할 경우 \$ 메타문자와는 달리 전체 문자열의 끝과 매치된다.

* 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

\b

\b는 단어 구분자(Word boundary)이다. 보통 단어는 whitespace에 의해 구분이 된다. 다음의 예를 보자.

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<_sre.SRE_Match object at 0x01F6F3D8>
```

\bclass\b 정규식은 “class”라는 단어와 매치됨을 의미한다. 따라서 no class at all의 class라는 단어와 매치됨을 확인할 수 있다.

```
>>> print(p.search('the declassified algorithm'))
None
```

위 예의 the declassified algorithm라는 문자열 안에 class라는 문자열이 포함되어 있긴 하지만 whitespace로 구분된 단어가 아니므로 매치되지 않는다.

```
>>> print(p.search('one subclass is'))
None
```

subclass라는 문자열 역시 class앞에 sub라는 문자열이 더해져 있으므로 매치되지 않음을 알 수 있다.

\b 메타문자를 이용할 경우 주의해야 할 점이 한가지 있다. \b는 파이썬 리터럴 규칙에 의하면 백스페이스(Back Space)를 의미하므로 백스페이스가 아닌 Word Boundary임을 알려주기 위해 r'\bclass\b'처럼 raw string임을 알려주는 기호 r을 반드시 붙여주어야 한다.

\B

\B 메타문자는 \b 메타문자의 반대의 경우이다. 즉, whitespace로 구분된 단어가 아닌 경우에만 매치된다.

```
>>> p = re.compile(r'\Bclass\B')
>>> print(p.search('no class at all'))
None
>>> print(p.search('the declassified algorithm'))
<_sre.SRE_Match object at 0x01F6FA30>
>>> print(p.search('one subclass is'))
None
```

class라는 문자열 좌우에 whitespace가 있는 경우에는 매치가 안되는 것을 확인할 수 있다.

그룹핑

ABC라는 문자열이 계속해서 반복되는지 조사하는 정규식을 작성하고 싶다고 하자. 어떻게 해야 할까? 지금까지 공부한 내용으로는 위 정규식을 작성할 수 없다. 이럴 때 필요한 것이 바로 그룹핑 (Grouping) 이다.

위의 경우는 다음처럼 그룹핑을 이용하여 작성할 수 있다.

```
(ABC)+
```

그룹을 만들어 주는 메타문자는 바로 (과)이다.

```
>>> p = re.compile('(ABC)+')
>>> m = p.search('ABCABCABC OK?')
>>> print(m)
<_sre.SRE_Match object at 0x01F7B320>
>>> print(m.group())
ABCABCABC
```

다음의 예를 보자.

```
>>> p = re.compile(r"\w+\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
```

\w+\s+\d+[-]\d+[-]\d+은 이름 + " " + 전화번호 형태의 문자열을 찾는 정규표현식이다. 그런데 이렇게 매치된 문자열 중에서 이름만 뽑아내고 싶다면 어떻게 해야 할까?

보통 반복되는 문자열을 찾을 때 그룹을 이용하는데, 그룹을 이용하는 보다 큰 이유는 위에서 볼 수 있듯이 매치된 문자열 중에서 특정 부분의 문자열만 뽑아내기 위해서인 경우가 더 많다.

위 예에서 만약 “이름” 부분만을 뽑아내려 한다면 다음과 같이 할 수 있다.

```
>>> p = re.compile(r"(\w+)\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(1))
park
```

이름에 해당되는 `\w+` 부분을 그룹(`(\w+)`)으로 만들면 match object의 `group(index)` 메서드를 이용하여 그룹핑된 부분의 문자열만 뽑아낼 수 있다. `group` 메서드의 `index`는 다음과 같은 의미를 갖는다.

group(인덱스) 설명

group(0)	매치된 전체 문자열
group(1)	첫 번째 그룹에 해당되는 문자열
group(2)	두 번째 그룹에 해당되는 문자열
group(n)	n 번째 그룹에 해당되는 문자열

다음의 예제를 계속해서 보자.

```
>>> p = re.compile(r"(\w+)\s+(\d+[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(2))
010-1234-1234
```

이번에는 전화번호 부분을 추가로 그룹(`(\d+[-]\d+[-]\d+)`)으로 만들었다. 이렇게 하면 `group(2)`처럼 사용하여 전화번호만을 뽑아낼 수 있다.

만약 전화번호 중에서 국번만 뽑아내고 싶으면 어떻게 해야 할까? 다음과 같이 국번 부분을 또 그룹핑하면 된다.

```
>>> p = re.compile(r"(\w+)\s+((\d+) [-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(3))
010
```

위 예에서 보듯이 `(\w+)\s+((\d+) [-]\d+[-]\d+)` 처럼 그룹을 중첩되게 사용하는 것도 가능하다. 그룹이 중첩되어 사용되는 경우는 바깥쪽부터 시작하여 안쪽으로 들어갈수로 인덱스가 증가한다.

그룹핑된 문자열 재참조하기

그룹의 또 하나 좋은 점은 한번 그룹핑된 문자열을 재참조(Backreferences)할 수 있다는 점이다. 다음의 예를 보자.

```
>>> p = re.compile(r'(\b\w+)\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

정규식 `(\b\w+)\s+\1`은 `(그룹1) + " " + "그룹1과 동일한 단어"` 와 매치됨을 의미한다. 이렇게 정규식을 만들게 되면 2개의 동일한 단어가 연속적으로 사용되어야만 매치되게 된다. 이것을 가능하게 해 주는 것이 바로 재 참조 메타문자인 `\1`이다. `\1`은 정규식의 그룹중 첫번째 그룹을 지칭한다.

※ 두번째 그룹을 참조하려면 `\2`를 사용하면 된다.

그룹핑된 문자열에 이름 붙이기

정규식 내에 그룹이 무척 많아진다고 가정해 보자. 예를 들어 정규식 내에 그룹이 10개 이상만 되어도 매우 혼란스러울 것이다. 거기에 더해 정규식이 수정되면서 그룹이 추가, 삭제되면 그 그룹을 인덱스로 참조했던 프로그램들도 모두 변경해 주어야 하는 위험도 갖게 된다. 만약 그룹을 인덱스가 아닌 이름(Named Groups)으로 참조할 수 있다면 어떨까? 그렇다면 이런 문제들에서 해방되지 않을까?

이러한 이유로 정규식은 그룹을 만들 때 그룹명을 지정할 수 있게 했다. 그 방법은 다음과 같다.

```
(?P<name>\w+)\s+((\d+) [-]\d+[-]\d+)
```

위 정규식은 이전에 보았던 이름과 전화번호를 추출하는 정규식이다. 기준과 달라진 부분은 다음과 같다.

```
(\w+) -> (?P<name>\w+)
```

대단히 복잡해 진것처럼 보이지만 (\w+)라는 그룹에 “name”이라는 이름을 붙인 것에 불과하다. 여기서 사용한 (?) 표현식은 정규표현식의 확장구문이다. (여기서 ...은 다양하게 변한다는 anything의 의미이다.) 이 확장구문을 이용하기 시작하면 가독성이 무척 떨어지긴 하지만 반면에 강력함을 갖게 된다.

그룹에 이름을 지어주기 위해서는 다음과 같은 확장구문을 사용해야 한다.

```
(?P<그룹명>...)
```

그룹에 이름을 지정하고 참조하는 다음의 예를 보자.

```
>>> p = re.compile(r"(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group("name"))
park
```

위 예에서 볼 수 있듯이 name이라는 그룹명으로 참조할 수 있다.

그룹명을 이용하면 정규식 내에서 재참조하는 것도 가능하다.

```
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
>>> p.search('Paris in the the spring').group()
'the the'
```

위 예에서 보듯이 재 참조시에는 (?P=그룹명)이라는 확장구문을 이용해야 한다.

전방 탐색

정규식에 막 입문한 사람들이 가장 어려워하는 것이 바로 전방 탐색(Lookahead Assertions) 확장 구문이다. 정규식 안에 이 확장 구문이 사용되면 순식간에 암호문처럼 알아보기 어렵게 바뀌기 때문이다. 하지만 이 전방 탐색이 꼭 필요한 경우가 있으며 매우 유용한 경우도 많으니 꼭 알아 두도록 하자.

다음의 예제를 보자.

```
>>> p = re.compile(".+:")
>>> m = p.search("http://google.com")
>>> print(m.group())
http:
```

정규식 `.+:` 과 일치하는 문자열로 “http:”가 리턴되었다. 하지만 “http:”라는 검색 결과에서 “.”을 제외하고 출력하려면 어떻게 해야 할까? 위 예는 그나마 간단하지만 훨씬 복잡한 정규식이어서 그룹핑은 추가로 할 수 없다는 조건까지 더해진다면 어떻게 해야 할까?

이럴 때 사용할 수 있는 것이 바로 전방 탐색이다. 전방 탐색에는 긍정(Positive)과 부정(Negative)의 2종류가 있고 다음과 같이 표현된다.

- 긍정형 전방 탐색(`(?=...)`) - ...에 해당되는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소모되지 않는다.
- 부정형 전방 탐색(`(?!...)`) - ...에 해당되는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소모되지 않는다.

긍정형 전방 탐색

긍정형 전방 탐색을 이용하면 `http:`의 결과를 `http`로 바꿀 수 있다. 다음의 예를 보자.

```
>>> p = re.compile(".+(?=:)")
>>> m = p.search("http://google.com")
>>> print(m.group())
http
```

정규식 중 `:`에 해당하는 부분이 긍정형 전방탐색 기법이 적용되어 `(?=:)` 으로 변경되었다. 이렇게 되면 기존 정규식과 검색에서는 동일한 효과를 발휘하지만 `:`에 해당되는 문자열이 정규식 엔진에 의해 소모되지 않아(검색에는 포함되지만 검색 결과에는 제외됨) 검색 결과에서는 `:이 제거된 후` 리턴되는 효과가 있다.

자, 이번에는 다음의 정규식을 보자.

```
.*[.].*$
```

이 정규식은 파일명 `..` + 확장자 를 나타내는 정규식이다. 이 정규식은 `foo.bar`, `autoexec.bat`, `sendmail.cf`등과 매치할 것이다.

자, 이제 이 정규식에 확장자가 “bat인 파일은 제외해야 한다”는 조건을 추가해 보자. 가장 먼저 생각할 수 있는 정규식은 다음과 같을 것이다.

```
.*[.] [^b].*$
```

이 정규식은 확장자가 b라는 문자로 시작하면 안된다는 의미이다. 하지만 이 정규식은 foo.bar라는 파일마저 걸러내 버린다. 다시 정규식을 다음과 같이 수정 해 보자.

```
.*[.]([^b]...|.^a|.^.t)$
```

이 정규식은 | 메타 문자를 사용하여 확장자의 첫 번째 문자가 b가 아니거나 두 번째 문자가 a가 아니거나 세 번째 문자가 t가 아닌 경우를 의미한다. 이 정규식에 의하여 foo.bar는 제외되지 않고 autoexec.bat은 제외되어 만족스러운 결과를 리턴한다. 하지만 이 정규식은 아쉽게도 sendmail.cf처럼 확장자의 문자 개수가 2개인 케이스를 포함하지 못 하는 오동작을 하기시작한다.

자, 이제 다음과 같이 바꾸어야 한다.

```
.*[.]([^b].?.?|.^a?.?|..?[^t]?)$
```

확장자의 문자 개수가 2개여도 통과되는 정규식이 만들어졌다. 하지만 정규식은 점점 더 복잡해지고 이해하기 어려워진다.

자, 그런데 여기서 bat 파일말고 exe 파일도 제외하라는 조건이 추가로 생긴다면 어떻게 될까? 이 모든 조건을 만족하는 정규식을 구현하려면 패턴은 더욱더 복잡해져야만 할 것이다.

부정형 전방 탐색

이러한 상황의 구원투수는 바로 “부정형 전방탐색”이다. 위 케이스는 부정형 전방탐색을 사용하면 다음과 같이 간단하게 처리된다.

```
.*[.](?!bat$).*$
```

확장자가 bat가 아닌 경우에만 통과된다는 의미이다. bat라는 문자열이 있는지 조사하는 과정에서 문자열이 소모되지 않으므로 bat가 아니라고 판단되면 그 이후 정규식 매칭이 진행된다.

exe 역시 제외하라는 조건이 추가되더라도 다음과 같이 간단히 표현할 수 있다.

```
.*[.](?!bat$|exe$).*$
```

문자열 바꾸기

sub 메서드를 이용하면 정규식과 매치되는 부분을 다른 문자로 쉽게 바꿀 수 있다.

다음의 예를 보자.

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
```

sub 메서드의 첫 번째 입력 인수는 “바꿀 문자열(replacement)”이 되고, 두 번째 입력 인수는 “대상 문자열”이 된다. 위 예에서 볼 수 있듯이 blue 또는 white 또는 red라는 문자열이 colour라는 문자열로 바뀌는 것을 확인할 수 있다.

그런데 딱 한 번만 바꾸고 싶은 경우도 있다. 이렇게 바꾸기 횟수를 제어하려면 다음과 같이 세 번째 입력 인수로 count 값을 넘기면 된다.

```
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

처음 일치하는 blue만 colour라는 문자열로 한 번만 바꾸기가 실행됨을 알 수 있다.

[sub 메서드와 유사한 subn 메서드]

subn 역시 sub와 동일한 기능을 하지만 리턴되는 결과를 튜플로 리턴한다는 차이가 있다. 리턴된 튜플의 첫 번째 요소는 변경된 문자열이고, 두 번째 요소는 바꾸기가 발생한 횟수이다.

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
```

sub 메서드 사용 시 참조 구문 사용하기

sub 메서드를 사용할 때 참조 구문을 사용할 수 있다. 다음의 예를 보자.

```
>>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
>>> print(p.sub("\g<phone> \g<name>", "park 010-1234-1234"))
010-1234-1234 park
```

위 예는 이름 + 전화번호의 문자열을 전화번호 + 이름으로 바꾸는 예이다. `sub`의 바꿀 문자열 부분에 `\g<그룹명>`을 이용하면 정규식의 그룹명을 참조할 수 있게된다.

다음과 같이 그룹명 대신 참조번호를 이용해도 마찬가지 결과가 리턴된다.

```
>>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+) [-]\d+[-]\d+)")
>>> print(p.sub("\g<2> \g<1>", "park 010-1234-1234"))
010-1234-1234 park
```

sub 메서드의 입력 인수로 함수 넣기

`sub` 메서드의 첫 번째 입력 인수로 함수를 넣을 수도 있다. 다음의 예를 보자.

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffffd2 for printing, 0xc000 for user code.'
```

`hexrepl` 함수는 `match` 객체(위에서 숫자에 매치되는)를 입력으로 받아 16진수로 변환하여 리턴하는 함수이다. `sub`의 첫 번째 입력 인수로 함수를 사용할 경우 해당 함수의 첫 번째 입력 인수에는 정규식과 매치된 `match` 객체가 입력된다. 그리고 매치되는 문자열은 함수의 리턴값으로 바뀌게 된다.

Greedy vs Non-Greedy

정규식에서 Greedy(탐욕스러운)란 어떤 의미일까? 다음의 예제를 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> s = '<html><head><title>Title</title>'  
>>> len(s)  
32  
>>> print(re.match('<.*>', s).span())  
(0, 32)  
>>> print(re.match('<.*>', s).group())  
<html><head><title>Title</title>
```

<.*> 정규식의 매치결과로 <html> 문자열이 리턴되기를 기대했을 것이다. 하지만 * 메타문자는 매우 탐욕스러워서 매치할 수 있는 최대한의 문자열인 <html><head><title>Title</title> 문자열을 모두 소모시켜 버렸다. 어떻게 하면 이 탐욕스러움을 제한하고 <html> 이라는 문자열 까지만 소모되도록 막을 수 있을까?

다음과 같이 non-greedy 문자인 ?을 사용하면 *의 탐욕을 제한할 수 있다.

```
>>> print(re.match('<.*?>', s).group())  
<html>
```

non-greedy 문자인 ?은 *?, +?, ??, {m,n}?과 같이 사용할 수 있다. 가능한 한 가장 최소한의 반복을 수행하도록 도와주는 역할을 한다.

07-4 파일로 XML 처리하기

이 절에서는 파일로 XML 문서를 다루는 방법에 대해서 살펴본다. XML 처리를 위한 파일라이브러리는 아주 많으며 아래 사이트에서 확인할 수 있다.

<http://wiki.python.org/moin/PythonXml>

※ 이 절은 XML에 대한 기초적인 지식이 있는 사람을 대상으로 한다. XML 처리에 대해 굳이 알아야 할 필요가 없다면 건너뛰어도 좋다.

이 책에서는 가장 많은 이들의 사랑을 받고 있는 라이브러리인 ElementTree를 사용하는 방법에 대해서 다룬다. ElementTree는 Tkinter로 유명한 프레드릭 런드(Fredrik Lundh)가 만든 XML 제너레이터 & 파서이다.

※ ElementTree는 외부 라이브러리로 존재하다가 파일 2.5부터 통합되었다.

XML 문서 생성하기

ElementTree를 이용하여 다음과 같은 구조의 XML문서를 생성해 보자.

```
<note date="20120104">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

먼저 다음과 같은 소스를 작성해 보자:

```
from xml.etree.ElementTree import Element, dump

note = Element("note")
to = Element("to")
to.text = "Tove"

note.append(to)
dump(note)
```

위 소스의 실행 결과는 다음과 같다:

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
<note><to>Tove</to></note>
```

엘리먼트(Element)를 이용하면 태그를 만들 수 있고, 만들어진 태그에 텍스트 값을 추가할 수 있음을 알 수 있다.

SubElement

서브엘리먼트(SubElement)를 이용하면 조금 더 편리하게 태그를 추가할 수 있다.

```
from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
to = Element("to")
to.text = "Tove"

note.append(to)
SubElement(note, "from").text = "Jani"

dump(note)
```

실행 결과는 다음과 같다.

```
<note><to>Tove</to><from>Jani</from></note>
```

서브엘리먼트는 태그명과 태그의 텍스트 값을 한 번에 설정할 수 있다.

또는 다음과 같이 태그 사이에 태그를 추가하거나 특정 태그를 삭제할 수도 있다.

```
dummy = Element("dummy")
note.insert(1, dummy)
note.remove(dummy)
```

위의 예는 dummy라는 태그를 삽입하고 삭제하는 경우이다.

애트리뷰트 추가하기

이번에는 note 태그에 애트리뷰트(attribute)를 추가해 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
to = Element("to")
to.text = "Tove"

note.append(to)
SubElement(note, "from").text = "Jani"
note.attrib["date"] = "20120104"

dump(note)
```

note.attrib[“date”]와 같은 방법으로 애트리뷰트 값(attribute value)을 추가할 수 있다.

다음과 같이 엘리먼트 생성 시 직접 애트리뷰트 값을 추가하는 방법을 사용해도 된다.

```
note = Element("note", date="20120104")
```

실행 결과는 다음과 같다.

```
<note date="20120104"><to>Tove</to><from>Jani</from></note>
```

이상으로 XML 태그와 애트리뷰트를 추가하는 방법에 대해서 알아보았다. 완성된 소스는 다음과 같다.

```

from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
note.attrib["date"] = "20120104"

to = Element("to")
to.text = "Tove"
note.append(to)

SubElement(note, "from").text = "Jani"
SubElement(note, "heading").text = "Reminder"
SubElement(note, "body").text = "Don't forget me this weekend!"
dump(note)

```

실행 결과는 다음과 같다.

```
<note date="20120104"><to>Tove</to><from>Jani</from>...생략...</note>
```

indent 함수

위 결과값은 한 줄로 이어져 있어 보기가 쉽지 않다. 정렬된 형태의 xml 값을 보려면 다음과 같이 indent 함수를 이용해 보자.

```

from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
note.attrib["date"] = "20120104"

to = Element("to")
to.text = "Tove"
note.append(to)

SubElement(note, "from").text = "Jani"
SubElement(note, "heading").text = "Reminder"
SubElement(note, "body").text = "Don't forget me this weekend!"

def indent(elem, level=0):
    i = "\n" + level*"  "
    if len(elem):
        if not elem.text or not elem.text.strip():
            elem.text = i + "  "
        if not elem.tail or not elem.tail.strip():
            elem.tail = i
        for elem in elem:
            indent(elem, level+1)
        if not elem.tail or not elem.tail.strip():
            elem.tail = i
    else:
        if level and (not elem.tail or not elem.tail.strip()):
            elem.tail = i

indent(note)
dump(note)

```

indent 함수를 이용하면 다음과 같이 정렬된 형태의 xml 값을 확인할 수 있다.

```
<note date="20120104">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

파일에 쓰기(write) 수행하기

이제 생성한 xml 문서를 가지고 다음과 같이 엘리먼트의 write 메서드를 이용하여 파일에 쓰기를 해보자.

```
from xml.etree.ElementTree import ElementTree
ElementTree(note).write("note.xml")
```

note.xml이 생성되는 것을 확인할 수 있다.

XML문서 파싱하기

이번에는 생성된 XML 문서를 파싱(parsing)하고 검색하는 방법에 대해서 알아보자.

```
from xml.etree.ElementTree import parse
tree = parse("note.xml")
note = tree.getroot()
```

ElementTree의 parse라는 함수를 이용하여 xml을 파싱할 수 있다.

에트리뷰트 값 읽기

어트리뷰트 값은 다음과 같이 읽을 수 있다:

```
print(note.get("date"))
print(note.get("foo", "default"))
print(note.keys())
print(note.items())
```

get 메서드는 애트리뷰트의 값을 읽는다. 만약 두 번째 입력 인수로 디폴트 값을 주었다면 첫 번째 인자에 해당되는 애트리뷰트 값이 없을 경우 두 번째 값을 리턴한다.

keys는 모든 애트리뷰트의 키 값을 리스트로 리턴하고, items는 key, value 쌍을 리턴한다. 애트리뷰트 값을 가져오는 방법은 앞에서 배운 딕셔너리와 동일함을 알 수 있다.

결과는 다음과 같다.

```
20120104
default
['date']
[('date', '20120104')]
```

XML 태그 접근하기

XML 태그에 접근하는 방법은 다음과 같다

```
from_tag = note.find("from")
from_tags = note.findall("from")
from_text = note.findtext("from")
```

`note.find("from")`은 `note` 태그 하위에 `from`과 일치하는 첫 번째 태그를 찾아서 리턴하고, 없으면 `None`을 리턴한다. `note.findall("from")`은 `note` 태그 하위에 `from`과 일치하는 모든 태그를 리스트로 리턴한다. `note.findtext("from")`은 `note` 태그 하위에 `from`과 일치하는 첫번째 태그의 텍스트 값을 리턴한다.

특정 태그의 모든 하위 엘리먼트를 순차적으로 처리할 때는 아래의 메서드를 사용한다.

```
childs = note.getiterator()
childs = note.getchildren()
```

`getiterator()` 함수는 첫 번째 인수로 다음과 같이 태그명을 받을 수도 있다.

```
note.getiterator("from")
```

위와 같은 경우 `from` 태그의 하위 엘리먼트들이 순차적으로 리턴되며, 보통 다음과 같이 많이 사용된다.

```
for parent in tree.getiterator():
    for child in parent:
        ... work on parent/child tuple
```

이상으로 파이썬에서 XML을 처리하기 위한 ElementTree의 사용 방법에 대해서 간략히 알아보았다. 더 자세한 내용은 프레드릭 런드가 직접 작성한 튜토리얼에서 확인하기 바란다.

<http://effbot.org/zone/element.htm>

07-5 파이썬으로 Json 처리하기

json 이란?

json(JavaScript Object Notation)은 원래 자바스크립트에서 만들어진 데이터 표현방식으로 최근 들어 사용이 부쩍 증가되고 있다. 예전에는 xml방식의 데이터 교환을 많이 했다면 요새는 거의 json으로 하는 추세이다.

json은 파이썬의 딕셔너리와 매우 비슷하게 생겼다.

json 표현의 예

```
{
    "name": "홍길동",
    "birth": "0525",
    "age": 30
}
```

json dumps, loads

파이썬 자료형을 json 문자열로 만드는 방법에 대해서 알아보자.

먼저 json을 사용하기 위해서 json 모듈을 import 해야 한다.

```
>>> import json
```

json문자열을 만들기 위해서는 json 모듈의 dumps 함수를 이용하면 된다.

딕셔너리를 json 문자열로 변경해 보자.

```
>>> j1 = {"name": "홍길동", "birth": "0525", "age": 30}
>>> j1
{'age': 30, 'birth': '0525', 'name': '홍길동'}
>>> json.dumps(j1)
'{"age": 30, "birth": "0525", "name": "\ubc15\uc751\uc6a9"}'
```

출력되는 json문자열을 보기좋게 정렬하려면 다음처럼 indent 옵션을 추가하면 된다.

```
>>> print(json.dumps(j1, indent=2))
{
    "age": 30,
    "birth": "0525",
    "name": "\ubc15\uc751\uc6a9"
}
```

리스트나 튜플도 json 문자열로 변경해 보자.

```
>>> json.dumps([1,2,3])
'[1, 2, 3]'
>>> json.dumps((4,5,6))
'[4, 5, 6]'
```

리스트나 튜플은 모두 동일한 Array형태의 json문자열로 변환됨을 알 수 있다.

이번에는 json모듈의 loads 함수를 이용하여 만들어진 json 문자열을 파이썬 객체로 다시 변경해 보자.

```
>>> j1 = {"name": "홍길동", "birth": "0525", "age": 30}
>>> d1 = json.dumps(j1)
>>> json.loads(d1)
{'name': '홍길동', 'birth': '0525', 'age': 30}
```

딕셔너리 객체를 json으로 변환(dumps)하고 변환된 json객체를 다시 딕셔너리로 변환(loads)할 수 있다.

json file

이번에는 myinfo.json 이라는 json파일을 파이썬 객체로 읽어보자.

c:/doit/myinfo.json

```
{
    "name": "홍길동",
    "birth": "0525",
    "age": 30
}
```

위와 같은 'c:/doit/myinfo.json'이라는 파일을 먼저 작성한 후 다음 예제를 실행해 보자.

```
>>> with open('c:/doit/myinfo.json') as f:
...     data = json.load(f)
...
>>> print(type(data))
<class 'dict'>
>>> print(data)
{'name': '홍길동', 'birth': '0525', 'age': 30}
```

json 파일을 읽을 때는 위 예처럼 loads 대신 load 함수를 이용하는 것이 편리하다. loads는 문자열을 읽어 들어고 load는 파일을 읽어 들어는 json 함수이다.

json 송수신

json은 URL요청시 입출력 데이터로 많이 활용된다. 다음의 간단한 예제를 살펴보자.

```
import json
import urllib.request

url = "http://ip.jsontest.com" # URL

d = {'name': '홍길동', 'birth': '0525', 'age': 30}
params = json.dumps(d).encode("utf-8") # encode: 문자열을 바이트로 변환
req = urllib.request.Request(url, data=params,
                             headers={'content-type': 'application/json'})
response = urllib.request.urlopen(req)
print(response.read().decode('utf8')) # decode: 바이트를 문자열로 변환
```

위 예제는 http://ip.jsontest.com이라는 URL에 json요청을 보내고 그 응답으로 json을 리턴받아

출력하는 예제이다. 아마도 위 예제를 수행하면 호출한 PC의 IP 주소가 출력될 것이다. (참고. <http://ip.jsontest.com> 은 호출한 클라이언트의 아이피를 출력해 주는 테스트 서비스이다.)

urllib.request.Request 사용시 json문자열이 아닌 json 바이트 배열로 주고 받아야 한다는 점에 유의하자.

07-6 소트

소트(Sort)방법에 대해서 좀 더 자세하게 알아보도록 하자.

평범한 소트

자주 사용해 왔던 소트를 먼저 보자.

```
>>> a = [5, 3, 1, 4, 2]
>>> sorted(a)
[1, 2, 3, 4, 5]
```

sorted 라는 내장함수를 사용하면 위와 같이 입력으로 받은 리스트를 소트하여 새로운 리스트를 리턴해 준다.

또는 다음과 같이 소트할 수도 있다.

```
>>> a = [5, 3, 1, 4, 2]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

리스트의 sort함수를 이용하면 리스트 그 자체를 소트할 수 있다. 위에서 살펴보았던 sorted와의 차이점은 sorted는 소트 후 새로운 리스트를 리턴한다는 점이고 리스트 자체의 sort함수는 리스트 그 자체를 소트시킨다는 점이 다르다.

하지만 sorted는 다음과 같은것도 가능하게 해 주는 기특한 점이 있다.

```
>>> b = {"c":90, "b":45, "a": 88}
>>> sorted(b)
['a', 'b', 'c']
```

딕셔너리처럼 반복이 가능한(iterable) 객체는 sorted를 사용할 수 있다. 딕셔너리에 sorted를 적용하면 딕셔너리의 키 값들을 소트하여 리턴한다.

key를 이용하여 소트하기

이번에는 조금 어려운 소트를 해 보도록 하자.

먼저 다음과 같은 리스트를 생각해 보자.

```
students = [
    ("jane", 22, 'A'),
    ("dave", 32, 'B'),
    ("sally", 17, 'B'),
]
```

students라는 리스트는 총 3개의 튜플을 가지고 있다. 각 튜플은 순서대로 “이름”, “나이”, “성적”에 해당되는 데이터를 갖는다.

이 리스트를 나이 순으로 소트하려면 어떻게 해야 할까?

이런 경우에는 sort 또는 sorted의 key파라미터를 이용하여 소트해야 한다.

```
>>> students = [
...     ("jane", 22, 'A'),
...     ("dave", 32, 'B'),
...     ("sally", 17, 'B'),
... ]
>>> sorted(students, key=lambda student: student[1])
[('sally', 17, 'B'), ('jane', 22, 'A'), ('dave', 32, 'B')]
```

key파라미터에는 함수가 와야 한다. key 파라미터에 함수가 설정되면 소트해야 할 리스트들의 항목들이 하나씩 key 함수에 전달되어 key 함수가 실행되게 된다. 이 때 수행된 key 함수의 리턴값을 기준으로 소트가 진행된다.

위 예에서는 key함수에 students의 요소인 튜플데이터가 key함수의 입력으로 순차적으로 전달될 것이다. key함수는 입력된 튜플 데이터의 “나이”를 의미하는 2번째 항목을 리턴하는 lambda함수이다. 따라서 sorted수행 후 나이순으로 소트된 리스트가 리턴된다.

이번에는 객체로 이루어진 리스트를 소트해 보도록 하자.

우선 다음과 같은 클래스를 만들어 보자.

```

class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def __repr__(self):
        return repr((self.name, self.age, self.grade))

```

Student 클래스는 이름, 나이, 성적을 생성자의 입력으로 받아서 객체를 생성하는 클래스이다. 따라서 Student 클래스는 다음과 같이 사용해야 한다.

```

>>> student = Student('jane', 22, 'A')
>>> print(student.name)
jane
>>> print(student.age)
22

```

`__repr__` 함수는 Student 클래스에 의해서 생성된 객체를 출력할 때 사용된다. 즉, 다음과 같이 `print`를 이용하여 객체를 출력할 때 호출되게 된다.

```

>>> student = Student('jane', 22, 'A')
>>> print(student)
('jane', 22, 'A')

```

이제 Student 클래스를 이용하여 다음과 같은 리스트를 생성해 보자.

```

student_objects = [
    Student('jane', 22, 'A'),
    Student('dave', 32, 'B'),
    Student('sally', 17, 'B'),
]

```

그리고 이 리스트를 `key` 함수를 이용하여 나이순서대로 소트해 보자.

```
result = sorted(student_objects, key=lambda student: student.age)
print(result)
```

위와 같이 key함수를 만들면 객체의 객체변수인 age값을 이용하여 소트하게 된다.

따라서 결과값은 나이 순서대로 소트되어 다음과 같이 출력될 것이다.

```
[('sally', 17, 'B'), ('jane', 22, 'A'), ('dave', 32, 'B')]
```

operator 모듈

이번에는 key함수에 lambda함수대신 operator 모듈을 이용하는 방법에 대해서 알아보자.

위에서 살펴본 2개의 예제는 operator모듈의 itemgetter와 attrgetter를 이용하면 좀 더 간략하게 작성할 수 있다.

첫번째 예제는 튜플의 2번째 요소로 소트를 하기 때문에 itemgetter를 대신 사용할 수 있다.

```
from operator import itemgetter

students = [
    ("jane", 22, 'A'),
    ("dave", 32, 'B'),
    ("sally", 17, 'B'),
]

result = sorted(students, key=itemgetter(1))
```

itemgetter(1)과 같이 사용하면 students의 item인 튜플의 2번째 요소로 소트를 하겠다는 의미 이다.

두번째 예제는 객체의 객체변수로 소트를 하기 때문에 다음과 같이 attrgetter를 대신 사용할 수 있다.

```

from operator import attrgetter

class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def __repr__(self):
        return repr((self.name, self.age, self.grade))

student_objects = [
    Student('jane', 22, 'A'),
    Student('dave', 32, 'B'),
    Student('sally', 17, 'B'),
]

result = sorted(student_objects, key=attrgetter('age'))

```

`attrgetter('age')`와 같이 사용하면 `student_objects`의 요소인 `Student` 객체의 객체변수 `age`로 소트를 하겠다는 의미이다.

순차 정렬과 역순 정렬

위 예제는 나이순으로 정렬을 했다. 그렇다면 나이가 큰 순서(역순)대로 정렬하려면 어떻게 하면 될까?

`sort` 또는 `sorted`함수에 `reverse`라는 파라미터를 이용하면 역순 정렬이 가능하다. 즉, 위 예제를 나이의 역순으로 정렬하려면 다음과 같이 `reverse` 파라미터를 추가하면 된다.

```

>>> sorted(student_objects, key=attrgetter('age'), reverse=True)
[('dave', 32, 'B'), ('jane', 22, 'A'), ('sally', 17, 'B')]

```

중첩 소트

이번에는 중첩해서 소트를 해야하는 경우에 대해서 알아보자.

위 두번째 예제에서 먼저 “성적”으로 소트하고 “성적”이 같을 경우 나이순으로 소트해야 한다고 가정해 보자.

만약 성적으로 소트를 하고 그 다음에 나이순으로 소트한다면 어떻게 될까?

```
>>> result = sorted(student_objects, key=attrgetter('grade'))
>>> sorted(result, key=attrgetter('age'))
[('sally', 17, 'B'), ('jane', 22, 'A'), ('dave', 32, 'B')]
```

다시 나이순으로 재 정렬되어 이미 “성적”으로 소트했던 결과가 무시되어 버린다.

이렇게 “성적”으로 소트하고 “성적”이 같을 경우 “나이”로 소트해야 할 경우에는 순서를 뒤 바꾸어 소트를 해 주면 된다. 즉 “나이”로 먼저 소트하고 그 다음에 “성적”으로 소트를 하면 된다.

```
>>> result = sorted(student_objects, key=attrgetter('age'))
>>> sorted(result, key=attrgetter('grade'))
[('jane', 22, 'A'), ('sally', 17, 'B'), ('dave', 32, 'B')]
```

이렇게 순서를 바꾸어 소트를 해 주어야 제대로 된 결과를 얻을 수 있게 된다.

이것이 가능한 이유는 소트할 때 기준의 순서를 그대로 유지하는(stable) 특성이 있기 때문이다. 위 예에서 보면 sally와 dave의 “성적”은 동일하지만 “나이”로 먼저 소트를 해 놓았기 때문에 그 순서가 유지되어 있는 것이다.

08장 종합문제

(종합문제 풀이 : <https://wikidocs.net/17116>)

[문제1] 이름 분석

주어진 문자열(공백 없이 쉼표로 구분되어 있음)을 가지고 아래 문제에 대한 프로그램을 작성하세요.

이의덕,이재명,권종수,이재수,박철호,강동희,이재수,김지석,최승만,이성만,박영희,박수호,전경식,송우환,김재식,이유정

1. 김씨와 박씨는 각각 몇 명 인가?
2. “이재수”란 이름이 몇 번 반복되나?
3. 중복을 제거한 이름을 출력하시오.
4. 중복을 제거한 이름을 오름차순으로 정렬하여 출력하시오.

[문제2] 합의 제곱과 제곱의 합의 차

1부터 10까지 자연수를 각각 제곱해 더하면 다음과 같다. (제곱의 합)

$$1^2 + 2^2 + \dots + 10^2 = 385$$

1부터 10을 먼저 더한 다음에 그 결과를 제곱하면 다음과 같다. (합의 제곱).

$$(1 + 2 + \dots + 10)^2 = 55^2 = 3025$$

따라서 1부터 10까지 자연수에 대해 “합의 제곱”과 “제곱의 합”의 차이는 $3025 - 385 = 2640$ 이 된다.

그러면 1부터 100까지 자연수에 대해 “합의 제곱”과 “제곱의 합”의 차이는 얼마일까?

[문제3] 1부터 100까지의 각 숫자의 갯수 구하기

10 ~ 15 까지의 각 숫자의 개수는 다음과 같이 구할 수 있다.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
10 = 1, 0
11 = 1, 1
12 = 1, 2
13 = 1, 3
14 = 1, 4
15 = 1, 5
```

그러므로 이 경우의 답은 0:1개, 1:7개, 2:1개, 3:1개, 4:1개, 5:1개

위와 같은 방식으로 1부터 100까지의 각 숫자의 개수를 구하시오.

[문제4] DashInsert

DashInsert 함수는 숫자로 구성된 문자열을 입력받은 뒤, 문자열 내에서 홀수가 연속되면 두 수 사이에 - 를 추가하고, 짝수가 연속되면 * 를 추가하는 기능을 갖고 있다. (예, 454 => 454, 4546793 => 454*67-9-3)

DashInsert 함수를 완성하자.

- 입력 - 화면에서 숫자로 된 문자열을 입력받는다.

```
4546793
```

- 출력 - *, -가 적절히 추가된 문자열을 화면에 출력한다.

```
454*67-9-3
```

[문제5] 문자열 압축하기

문자열을 입력 받아 같은 문자가 연속적으로 반복되는 경우에 그 반복 횟수를 표시하여 문자열을 압축하여 표시해 보자.

```
입력 예시: aaabbcccccba
```

```
출력 예시: a3b2c6a1
```

[문제6] Duplicate Numbers

0~9까지의 문자로 된 숫자를 입력받았을 때, 이 입력값이 0~9까지의 모든 숫자가 각각 한 번씩만 사용된 것인지 확인하는 함수를 작성해 보자.

입력 예시: 0123456789 01234 01234567890 6789012345 012322456789

출력 예시: true false false true false

[문제7] 모스 부호 해독

문자열 형식으로 입력받은 모스 부호(dot: . dash:-)를 해독하여 영어 문장으로 출력하는 프로그램을 작성해 보자.

글자와 글자 사이는 공백 하나, 단어와 단어 사이는 공백 두개로 구분한다.

예를 들어 다음 모스부호는 “HE SLEEPS EARLY”로 해석해야 한다.

..... -... -.. -.- -.. -.. -.. -.. -.. -..

모스부호 규칙 표

문자	부호	문자	부호
A	.-	N	-.
B	-...	O	—
C	-.-.	P	.-.
D	...	Q	-.-
E	.	R	-..
F	-..	S	...
G	-.	T	-
H	U	..-
I	..	V	...-
J	—	W	.-
K	-.-	X	-..-
L	-..	Y	-.-

문자	부호	문자	부호
M	-	Z	-..

[문제8] 정규식 1

다음 중 정규식 $a[.]^3b$ 과 매치되는 문자열은 무엇일까?

1. acccb
2. a....b
3. aaab
4. a.cccb

[문제9] 정규식 2

다음 코드의 결과값은 무엇일까?

```
>>> import re
>>> p = re.compile("[a-z]+")
>>> m = p.search("5 python")
>>> m.start() + m.end()
```

[문제10] 정규식 3

다음과 같은 문자열에서 핸드폰 번호 뒷자리인 숫자 4개를 #####로 바꾸는 프로그램을 정규식을 이용하여 작성해 보자.

```
"""
park 010-9999-9988
kim 010-9909-7789
lee 010-8789-7768
"""
```

[문제11] 정규식 4

다음은 이메일 주소를 나타내는 정규식이다. 이 정규식은 park@naver.com, kim@daum.net, lee@myhome.co.kr 등과 매치된다. 궁정형 전방 탐색 기법을 이용하여 .com, .net이 아닌 이메일 주소는 제외시키는 정규식을 작성해 보자.

```
.*[@].*[.].*$
```

[문제12] XML 문서 작성과 저장

ElementTree를 이용하여 다음 XML 문서를 작성하고 파일에 저장해 보자.

```
<blog date="20151231">
    <subject>Why python?</subject>
    <author>Eric</author>
    <content>Life is too short, You need Python!</content>
</blog>
```

[문제13] json 데이터

다음은 myinfo.json이라는 json데이터가 저장되어 있는 파일이다.

c:/doit/myinfo.json

```
{
    "name": "HONG GILDONG",
    "birth": "0525",
    "age": 30
}
```

이 파일의 데이터중 age 값을 40으로 바꾸어서 저장하시오.

[문제14] 요소값으로 소트하기

다음은 이름과 나이를 한쌍으로 갖는 튜플데이터를 요소값으로 갖는 students 리스트이다.

```
students = [
    ("홍길동", 22),
    ("김철수", 32),
    ("박영희", 17),
]
```

students 리스트를 나이순서대로 정렬하여 다음과 같이 만드시오.

```
students = [
    ("박영희", 17),
    ("홍길동", 22),
    ("김철수", 32),
]
```

[문제15] 시저 암호화

시저 암호는 고대 로마의 황제 줄리어스 시저가 만들어 낸 암호인데 예를 들어 알파벳 A를 입력했을 때 그 알파벳의 n개 뒤에 오는 알파벳이 출력되는 것이다. 예를 들어 바꾸려는 단어가 CAT이고 n을 5로 지정하였을 때 HFY가 되는 것이다.

예1) 단어는 CAT, n은 5

```
CAT -> HFY
```

예2) 단어는 ZOO, n은 3

```
ZOO -> CRR
```

어떠한 암호를 만들 단어와 n을 입력했을 때 암호를 만들어 출력하는 프로그램을 작성해라. (단, 단어는 항상 대문자로 표시한다.)

09장 마치며

점프 투 파이썬은 여기까지입니다. 많이 부족한 글을 끝까지 읽어 주셔서 감사합니다.

점프 투 파이썬은 여러분의 도움에 힘입어 계속 성장해 나가고 있습니다. 여러분의 피드백은 앞으로 이 글을 읽게 될 그 누군가에게 큰 도움이 될 것입니다.

여러분은 다음과 같은 방법으로 “점프 투 파이썬”을 도와주실 수 있습니다.

- 설명이 부족하여 잘 이해가 가지 않았던 부분을 알려주세요.
- 추가했으면 하는 내용을 알려주세요.
- 잘못된 내용이 있으면 알려주세요.
- 기타 하고 싶은 말씀이 있으면 알려주세요.

(이 글의 댓글이나 피드백 또는 이메일(pahkey@gmail.com)로 전달 부탁드립니다.)

감사합니다.

2017.12.08

A. 풀이

- 연습문제 풀이
- 종합문제 풀이

A-1 연습문제 풀이

02-1 숫자형

[풀이1] 점수의 평균

```
>>> a = 80
>>> b = 75
>>> c = 55
>>> (a+b+c)/3
70.0
```

[풀이2] 나눗셈의 몫

17을 3으로 나눗셈하면 다음과 같이 출력된다.

```
>>> 17 / 3
5.666666666666667
```

몫은 소수점 자리를 버려야 하므로 나머지를 버리고 그 몫만 리턴하는 // 연산자를 이용해야 한다.

```
>>> 17 // 3
5
```

[풀이3] 나눗셈의 나머지

나머지 값을 구할 수 있는 나머지 연산자인 %를 이용하여 나머지를 구할 수 있다.

```
>>> 17 % 3
2
```

알아두기

17 을 3으로 나누었을때의 그 몫과 나머지를 한꺼번에 구하고 싶다면 다음과 같이 divmod 내장 함수를 사용할 수 있다.

```
>>> divmod(17, 3)
(5, 2)
```

divmod를 이용하면 몫과 나머지가 튜플로 한꺼번에 리턴된다. (튜플은 02-4장에서, divmod는 05-5장에서 자세하게 다룬다.)

[풀이4] 자연수의 홀짝

나머지 연산자를 이용하면 자연수의 홀수, 짝수를 쉽게 판별할 수 있다.

```
>>> 1 % 2
1
>>> 2 % 2
0
>>> 3 % 2
1
>>> 4 % 2
0
```

1, 2, 3, 4라는 자연수를 2로 나누었을 때의 나머지 값을 출력하는 예제이다. 결과를 보면 자연수가 홀수일때는 1을 짝수일때는 0을 리턴하는 것을 확인할 수 있다.

02-2 문자열 자료형

[풀이1] 문자열 출력 1

문자열에 큰 따옴표가 있으므로 작은 따옴표를 이용하여 문자열을 구성한다.

```
>>> print('"점프 투 파이썬" 문제를 풀어보자')
"점프 투 파이썬" 문제를 풀어보자
```

알아두기

문자열을 만들기 위해서는 큰 따옴표나 작은 따옴표를 이용해야 한다. 만약 만들어야 하는 문자열에 큰 따옴표나 작은 따옴표가 있다면 사용되지 않은 따옴표를 이용하여 문자열을 구성하거나 이스케이프코드(\ " 또는 \'')를 이용해야 한다.

이스케이프 코드를 이용하는 방법은 다음과 같다.

```
>>> print("\\"점프 투 파이썬\\" 문제를 풀어보자")
"점프 투 파이썬" 문제를 풀어보자
```

또는 """", '''로 문자열을 만들어도 된다. 문자열에 큰 따옴표와 작은 따옴표가 혼재되어 있더라도 이 기호를 사용할 경우에는 문제가 없다는 장점이 있다.

```
>>> print("""점프 투 파이썬" 문제를 풀어보자""")
"점프 투 파이썬" 문제를 풀어보자
>>> print(""""점프 투 파이썬" 문제를 풀어보자""")
"점프 투 파이썬" 문제를 풀어보자
```

[풀이2] 문자열 출력 2

출력해야 할 문자열이 여러줄일 경우에는 """ 또는 ''' 기호를 이용하여 여러줄짜리 문자열을 쉽게 만들 수 있다.

```
>>> a = """
... Life is too short
... You need python
...
>>> print(a)

Life is too short
You need python

>>>
```

[풀이3] 공백 추가

공백 24개는 문자열 곱하기 * 기능을 이용하여 구한다.

```
>>> ' ' * 24
' ' '
```

24개의 공백 문자열 뒤에 "PYTHON"이라는 문자열은 문자열 더하기 + 기능을 이용하여 구한다.

```
>>> ' ' * 24 + "PYTHON"
' ' PYTHON'
```

또는 다음과 같이 문자열 포맷기능을 이용할 수 있다.

```
>>> "%30s" % "PYTHON"
'
    PYTHON'
```

[풀이4] 문자열 나누기

```
>>> pin = "881120-1068234"
>>> yyyyymmdd = pin[:6]
>>> num = pin[7:]
>>> print(yyyyymmdd)
881120
>>> print(num)
1068234
```

주민등록번호의 연월일은 앞에서부터 6자리까지의 숫자이고, 나머지 숫자는 하이픈을 제외해야 하므로 pin[7]부터 슬라이싱한다.

[풀이5] 문자열 인덱싱

```
>>> pin = "881120-1068234"
>>> print(pin[7])
1
```

성별을 나타내는 숫자는 하이픈을 포함하여 8번째 숫자이므로 8번째 자리를 인덱싱한다.

[풀이6] 문자열 변경

```
>>> a = '1980M1120'
>>> a[:4]
'1980'
>>> a[4]
'M'
>>> a[5:]
'1120'
>>> a[4]+a[:4]+a[5:]
'M19801120'
```

[풀이7] 문자열 찾기

```
>>> a = "Life is too short, you need python"
>>> a.find("short")
12
```

[풀이8] 문자열 바꾸기1

```
>>> a = "a:b:c:d"
>>> b = a.replace(":", "#")
>>> b
'a#b#c#d'
```

[풀이9] 문자열 바꾸기2

```
>>> a = "a:b:c:d"
>>> b = a.split(":")
>>> b
['a', 'b', 'c', 'd']
>>> c = "#".join(b)
>>> c
'a#b#c#d'
```

※ join을 이용한 문자열 삽입은 문자열 뿐만 아니라 리스트에도 사용 가능하다.

02-3 리스트 자료형

[풀이1] 리스트 인덱싱

리스트의 인덱싱을 이용하여 다음과 같이 구할 수 있다.

```
>>> a[4]+' '+a[2]
'you too'
```

[풀이2] 리스트 조인

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> a = ['Life', 'is', 'too', 'short']
>>> result = " ".join(a)
>>> print(result)
Life is too short
```

a 리스트의 각 단어들을 한 문장으로 조립할 때 단어들 사이마다 공백을 넣어 주어야 한다. 1개의 공백문자(“ ”)를 이용하여 join한다.

[풀이3] 리스트의 갯수

```
>>> a = [1, 2, 3]
>>> len(a)
3
```

len함수는 리스트 외에도 문자열, 딕셔너리, 튜플등에도 사용할 수 있다.

[풀이4] 리스트의 append와 extend

리스트 a에 [4, 5]를 append한 결과는 다음과 같다.

```
>>> a = [1, 2, 3]
>>> a.append([4,5])
>>> a
[1, 2, 3, [4, 5]]
```

리스트의 append 함수는 한 개의 값을 추가하는 함수이다. 여기서 사용된 한 개의 값은 [4, 5]라는 리스트이므로 리스트 a에 [4, 5]라는 리스트가 가장 마지막에 추가된 것이다.

리스트 a에 [4, 5]를 extend한 결과는 다음과 같다.

```
>>> a = [1, 2, 3]
>>> a.extend([4,5])
>>> a
[1, 2, 3, 4, 5]
```

리스트의 extend 함수는 기존 리스트에 입력 받은 리스트를 더하는 함수이므로 위와 같은 결과가 나온다.

[풀이5] 리스트 정렬

```
>>> a = [1, 3, 5, 4, 2]
>>> a.sort()
>>> a.reverse()
>>> print(a)
[5, 4, 3, 2, 1]
```

리스트의 내장 함수인 sort를 이용하여 리스트 값들을 먼저 정렬한 후 reverse 함수를 이용하여 순서를 뒤집는다.

또는 다음과 같이 sort함수의 reverse파라미터를 이용하여 한번에 역순으로 정렬할 수도 있다.

```
>>> a = [1, 3, 5, 4, 2]
>>> a.sort(reverse=True)
>>> print(a)
[5, 4, 3, 2, 1]
```

[풀이]6] 리스트 삭제

```
>>> a = [1, 2, 3, 4, 5]
>>> a.remove(2)
>>> a.remove(4)
>>> a
[1, 3, 5]
```

리스트의 내장 함수인 remove를 이용하여 2와 4라는 값을 삭제한다.

02-4 튜플 자료형

[풀이]1] 튜플 작성

1개의 요소값을 가지는 튜플은 다음과 같이 항상 콤마를 포함해 주어야 한다.

```
>>> a = (3,)
```

또는 다음과 같이 팔호를 생략하여 생성할 수도 있다.

```
>>> a = 3,
```

[풀이2] 튜플 변경

튜플은 한번 생성되면 그 값을 변경할 수 없는 immutable 객체이다. 따라서 위의 예와 같이 두번 째 요소값을 변경하려고 시도하면 오류가 발생하게 된다. 변경할 수 없는 것과 마찬가지로 다음과 같이 튜플의 요소값을 삭제하려고 할때도 비슷한 오류가 발생하게 된다.

```
>>> a = (1, 2, 3)
>>> del(a[1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

[풀이3] 튜플 추가

```
>>> a = (1, 2, 3)
>>> a = a + (4,)
>>> a
(1, 2, 3, 4)
```

a 튜플에 (4,)라는 튜플을 더하면 된다. 단, 이때 만들어지는 a + (4,)의 결과는 a 값이 변경되는 것이 아니라(튜플은 그 값을 변경할 수 없다) 새로운 튜플이 생성되고 그 값이 a 변수에 대입되는 것임에 유념하자.

다음 코드를 실행해 보면 a의 고유 주소값이 변경되는 것을 확인할 수 있다.

```
>>> a = (1, 2, 3)
>>> id(a)
4302278872
>>> a = a + (4,)
>>> a
(1, 2, 3, 4)
>>> id(a)
4302246440
```

※ id는 자료형의 주소값을 확인할 수 있는 파이썬 내장함수이다. id내장 함수에 대해서는 02-8장에서 배운다.

02-5 딕셔너리 자료형

[풀이1] 딕셔너리 만들기

```
>>> a = dict()
>>> a['name'] = '홍길동'
>>> a['birth'] = '1128'
>>> a['age'] = 30
>>> a
{'birth': '1128', 'age': 30, 'name': '홍길동'}
```

또는 다음과 같이 한번에 만들 수도 있다.

```
>>> a = {'name': '홍길동', 'birth': '1128', 'age': 30}
>>> a
{'birth': '1128', 'age': 30, 'name': '홍길동'}
```

[풀이2] 딕셔너리 오류

3번째 예를 실행하면 다음과 같은 오류가 발생한다.

```
>>> a[[1]] = 'python'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

오류가 발생하는 이유는 딕셔너리의 키로는 변하는(mutable) 값을 사용할 수 없기 때문이다. 위 예에서 키로 사용된 [1]은 리스트이므로 변하는 값이다. 다른 예에서 키로 사용된 문자열, 튜플, 숫자는 변하지 않는(immutable) 값이므로 딕셔너리의 키로 사용이 가능하다.

[풀이3] 딕셔너리 값 추출1

딕셔너리도 리스트와 마찬가지로 다음과 같이 pop 함수를 사용할 수 있다.

```
>>> a = {'A':90, 'B':80, 'C':70}
>>> result = a.pop('B')
>>> print(a)
{'A': 90, 'C': 70}
>>> print(result)
80
```

'B' 키 값에 해당되는 값이 리턴되고 딕셔너리 a에서는 그 값이 제거되는 것을 확인할 수 있다.

[풀이4] 딕셔너리 값 추출2

딕셔너리의 get 함수를 이용하면 해당 key가 없을 경우에는 두번째 파라미터로 전달된 default 값을 대신 리턴해 준다.

```
>>> a = {'A':90, 'B':80}
>>> a.get('C', 70)
70
```

위 예에서는 'C'에 해당되는 key가 없기 때문에 디폴트 값으로 전달된 70이 리턴되었다.

[풀이5] 딕셔너리 최소값

딕셔너리의 value 값을 구하기 위해서는 딕셔너리의 values 함수를 이용한다.

```
>>> a = {'A':90, 'B':80, 'C':70}
>>> a.values()
dict_values([80, 90, 70])
```

딕셔너리의 값 중에서 최소값을 구하기 위해서는 다음과 같이 min 내장함수를 사용한다.

```
>>> min(a.values())
70
```

※ min 내장함수는 05-5장에서 자세하게 다룬다.

[풀이6] 딕셔너리 리스트 변환

딕셔너리의 items 메서드를 이용하면 key, value 쌍이 합쳐진 형태의 dict_items 객체를 얻을 수 있다.

```
>>> a = {'A':90, 'B':80, 'C':70}
>>> a
{'B': 80, 'A': 90, 'C': 70}
>>> a.items()
dict_items([('B', 80), ('A', 90), ('C', 70)])
```

다시 dict_items 객체는 다음과 같이 리스트로 변경할 수 있다.

```
>>> a = list(a.items())
>>> a
[('B', 80), ('A', 90), ('C', 70)]
```

※ list 내장함수를 이용하면 dict_items 객체를 리스트로 변경할 수 있다. list 내장함수는 05-5장에서 자세하게 다룬다.

하지만 위의 결과는 우리가 원하는 결과와 순서가 다르므로 다음과 같이 소트를 해 주어야 한다.
(위 결과의 순서는 파이썬 버전에 따라 다르게 나올 수 있다.)

```
>>> a.sort()
>>> a
[('A', 90), ('B', 80), ('C', 70)]
```

02-6 집합 자료형

[풀이1] 집합 만들기1

set 함수를 이용하여 리스트를 집합 자료형으로 만들 수 있다.

```
>>> a = ['a', 'b', 'c']
>>> b = set(a)
>>> b
{'a', 'b', 'c'}
```

[풀이2] 집합의 중복

```
>>> a = [1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5]
>>> aSet = set(a) # a 리스트를 집합자료형으로 변환
>>> b = list(aSet) # 집합자료형을 리스트 자료형으로 다시 변환
>>> print(b) # [1,2,3,4,5] 출력
[1, 2, 3, 4, 5]
```

리스트 자료형이 집합 자료형으로 변환되면서 중복된 값들은 사라지게 된다. 이와 같은 성질을 이용하면 리스트 내에 중복된 값을 쉽게 제거할 수 있다.

[풀이3] 차집합

집합 자료형의 차집합 기능을 이용하면 s1집합의 항목에서 s2집합의 항목을 제거할 수 있다.

```
>>> s1 = set(['a', 'b', 'c', 'd', 'e'])
>>> s2 = set(['c', 'd', 'e', 'f', 'g'])
>>> s1 - s2
{'a', 'b'}
```

[풀이4] 집합 만들기2

set 함수를 인수없이 사용하면 비어있는 집합 자료형을 만들 수 있다.

```
>>> a = set()
>>> a
set()
>>> type(a)
<class 'set'>
```

[풀이5] 집합 추가

집합 자료형의 update 함수를 이용하면 값을 추가할 수 있다.

```
>>> a = {'a', 'b', 'c'}
>>> a.update({'d', 'e', 'f'})
>>> a
{'a', 'e', 'c', 'd', 'f', 'b'}
```

02-7 불 자료형

[풀이]1] 불 자료형과 조건문

```
>>> 1 != 1
False
>>> 3 > 1
True
>>> 'a' in 'abc'
True
>>> 'a' not in [1, 2, 3]
True
```

[풀이]2] bool 연산자

```
>>> a = "python"
>>> b = ""
>>> c = (1,2,3)
>>> d = 0
>>> bool(a)
True
>>> bool(b)
False
>>> bool(c)
True
>>> bool(d)
False
```

02-8 자료형의 값을 저장하는 공간, 변수

[풀이]1] 변수와 객체1

a, b 모두 [1, 2, 3]이라는 동일한 값을 갖는 리스트이지만 서로 같은 객체는 아니다. 따라서 동일한 객체인지를 판별하는 `is` 연산자의 결과는 `False`가 리턴된다.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

이와는 달리 값이 일치하는지 판별하는 `==` 연산자의 결과는 다음과 같이 된다.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
```

[풀이2] 변수와 객체2

`b = a` 와 같이 하면 `b`가 바라보는 객체와 `a`가 바라보는 객체가 같기 때문에 `a`와 `b`가 동일한 객체인지를 판별하는 `is` 연산의 결과는 `True`를 리턴한다.

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
```

[풀이3] 객체의 변경

`[1, 4, 3]`이 출력된다. `a`와 `b` 변수는 모두 동일한 `[1, 2, 3]`이라는 리스트 객체를 가리키고 있기 때문이다.

```
>>> a = b = [1, 2, 3]
>>> a[1] = 4
>>> print(b)
[1, 4, 3]
```

[풀이4] 리스트 복사1

`b = a[:]` 와 같이 사용하면 `b`는 `a`의 값을 copy한 새로운 값을 바라본다. 따라서 `b`와 `a`는 서로 다른 객체이므로 `False`가 출력된다.

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a is b
False
```

[풀이5] 리스트 복사2

[1, 2, 3] 이 출력된다. b리스트는 a리스트를 copy해서 만든 새로운 객체이므로 a리스트의 값을 변경하더라도 b리스트의 값이 변경되지는 않는다.

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> print(b)
[1, 2, 3]
```

[풀이6] 리스트의 더하기와 extend

리스트 a에 +기호를 이용하는 경우에 대해서 먼저 살펴보자.

```
>>> a = [1, 2, 3]
>>> id(a)
4302429640
```

id함수는 입력으로 받은 리스트 a의 주소값을 리턴해 준다. 현재 a라는 리스트는 4302429640이라는 주소에 저장되어 있다.

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

리스트 a에 + 기호를 이용하여 [4, 5]라는 리스트를 더해 보았다. 그리고 다시 다음과 같이 리스트 a의 주소값을 확인해 보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

```
>>> id(a)
4302472072
```

이전에 리스트 a가 저장되어 있던 주소와 다른 값이 리턴되는 것을 확인할 수 있다. 주소값이 다르기 때문에 +를 이용하면 리스트 a의 값이 변하는 것이 아니라 두 리스트가 더해진 새로운 리스트가 리턴된다는 것을 확인할 수 있다.

이번에는 extend를 이용해 보자.

```
>>> a = [1, 2, 3]
>>> id(a)
4302429640
```

리스트 a를 생성하고 그 주소값을 출력해 보았다.

```
>>> a.extend([4, 5])
>>> a
[1, 2, 3, 4, 5]
```

그리고 리스트 a에 extend를 이용하여 [4, 5]라는 리스트를 더 해 주었다. 그리고 다시 다음과 같이 리스트 a의 주소값을 확인해 보도록 하자.

```
>>> id(a)
4302429640
```

+기호를 이용하여 더한 경우와는 달리 주소값이 변하지 않고 그대로 유지되는 것을 확인할 수 있다.

[풀이7] 리스트 복사3

[1, [5, 3], 4]가 출력된다. a와 b는 서로 다른 객체이지만 a가 포함하고 있는 [2, 3]리스트와 b가 포함하고 있는 [2, 3]리스트는 서로 같은 곳을 바라보는 동일한 객체이다. 따라서 a가 포함하고 있는 리스트의 값을 변경하면 b가 포함하고 있는 리스트의 값도 함께 변경된다.

```
>>> a = [1, [2, 3], 4]
>>> b = a[:]
>>> a[1][0] = 5
>>> a
[1, [5, 3], 4]
>>> print(b)
[1, [5, 3], 4]
```

`b = a[:]` 와 같이 copy하는 것은 얕은 copy(shallow copy)라고도 한다. 그 이유는 위 예제에서 보는것과 같이 1 depth까지는 copy해 주지만 리스트가 포함하고 있는 2 depth 이상의 리스트는 copy 해 주지 못하기 때문이다. 만약 객체가 포함하고 있는 객체마저 모두 copy하고 싶다면 다음처럼 깊은 copy(deep copy)를 사용해야 한다.

```
>>> from copy import deepcopy
>>> a = [1, [2, 3], 4]
>>> b = deepcopy(a)
>>> a[1][0] = 5
>>> a
[1, [5, 3], 4]
>>> b
[1, [2, 3], 4]
```

03-1 if문

[풀이] 조건문1

“4,000원 이상의 돈을 가지고 있거나 카드가 있다”를 조건식으로 표현하면 다음과 같다.

```
>>> money >= 4000 or card
```

조건식을 이용하여 결과를 출력하면 다음과 같다.

```
>>> money = 5000
>>> card = False
>>> if money >= 4000 or card:
...     print("택시를 탈 수 있다.")
... else:
...     print("택시를 탈 수 없다.")
...
택시를 탈 수 있다.
```

[풀이2] 조건문2

리스트 중에 한개의 값과 일치하는지를 판별하기 위해서는 다음과 같이 in 조건식을 이용하면 된다.

```
>>> hong_number = 23
>>> lucky_list = [1, 9, 23, 46]
>>> if hong_number in lucky_list:
...     print("야호")
...
야호
```

[풀이3] 홀수 짝수 판별

주어진 수가 짝수인지 홀수인지를 판별하기 위해서는 주어진 수를 2로 나누어 보면 된다. 2로 나누었을때 나머지가 0인 경우는 짝수이고 나머지가 1인 경우에는 홀수가 될 것이다. 나머지 값을 알기 위해서는 나머지 연산자인 %를 이용해야 한다.

```
>>> num = 3 # num은 주어진 수
>>> if num % 2 == 0:
...     print("짝수")
... else:
...     print("홀수")
...
홀수
```

위와 같이 주어진 수가 3이라면 “홀수”라는 문자열이 출력된다.

[풀이4] 문자열 분석

문자열을 먼저 쉼표(,)로 나누고 나뉜 문자열을 다시 콜론(:)으로 나누면 나이와 키에 해당되는 값

을 구할 수 있다.

```
>>> a = "나이:30, 키:180"
>>> temp = a.split(",")
>>> age = temp[0].split(":")[-1]
>>> height = temp[1].split(":")[-1]
>>> if int(age) < 30 and int(height) >= 175: # 문자열을 숫자로 바꾸어 비교해야
    한다.
...     print("YES")
... else:
...     print("NO")
...
NO
```

[풀이] 5] 조건문3

결과값으로 shirt 가 출력된다.

```
>>> a = "Life is too short, you need python"
>>> if 'wife' in a: # False
...     print('wife')
... elif 'python' in a and 'you' not in a: # False
...     print('python')
... elif 'shirt' not in a: # True
...     print('shirt')
... elif 'need' in a: # True
...     print('need')
... else:
...     print('none')
...
shirt
```

가장 먼저 참이 되는 것이 세 번째 조건이므로 “shirt”가 출력된다.

03-2 while문

[풀이] 1부터 100까지 더하기

다음과 같이 while문을 이용하여 i값을 1부터 100까지 증가시켜 더한다.

```
result = 0
i = 1
while i <= 100:
    result += i
    i += 1

print(result) # 5050 출력
```

[풀이2] 3의 배수의 합

3의 배수는 3으로 나누어 떨어지는 수이다. 이러한 아이디어를 기반으로 한 파이썬 코드는 다음과 같다.

```
result = 0
i = 1
while i <= 1000:
    if i % 3 == 0: # 3으로 나누어 떨어지는 수는 3의 배수
        result += i
    i += 1

print(result) # 166833 출력
```

[풀이3] 50점 이상의 총합

```
A = [20, 55, 67, 82, 45, 33, 90, 87, 100, 25]

result = 0
while A: # A 리스트에 값이 있는 동안
    mark = A.pop() # A리스트의 가장 마지막 항목을 하나씩 뽑아냄
    if mark >= 50: # 50점 이상의 점수만 더함
        result += mark

print(result) # 481 출력
```

[풀이4] 별 표시하기1

```
i = 0
while True:
    i += 1 # while문 수행 시 1씩 증가
    if i > 5: break # i 값이 5보다 크면 while문을 벗어난다.
    print ('*' * i) # i 값 만큼 *를 출력한다.
```

while문을 수행할 때마다 i 값을 증가시킨다. 별 모양을 5번 출력해야 하므로 i 값이 5 이상일 경우 while문을 벗어나도록 한다. 별 모양을 i 값 만큼 출력하기 위해서 문자열 곱하기 기능을 이용한다.

[풀이5] 별 표시하기2

출력되는 별의 갯수는 7, 5, 3, 1 이고 앞에 포함되어야 할 공백 갯수는 0, 1, 2, 3 임을 알 수 있다. 이러한 아이디어를 기반으로 작성한 파이썬 코드는 다음과 같다.

```
star = 7 # 별의 갯수
space = 0 # 공백의 갯수
while star > 0:
    print(' ' * space + '*' * star) # 공백 + 별 출력
    star -= 2 # 별의 갯수는 2씩 감소
    space += 1 # 공백의 갯수는 1씩 증가
```

03-3 for문

[풀이1] 1부터 100까지 출력

```
>>> for i in range(1, 101):
...     print(i)
...
1
2
3
4
5
6
7
8
9
10
... 생략 ...
```

[풀이2] 5의 배수의 총합

```
>>> result = 0
>>> for i in range(1, 1001):
...     if i % 5 == 0:
...         result += i
...
>>> print(result)
100500
```

[풀이3] 학급의 평균 점수

```
A = [70, 60, 55, 75, 95, 90, 80, 80, 85, 100]
total = 0
for score in A:
    total += score # A학급의 점수를 모두 더한다.
average = total / len(A) # 평균을 구하기 위해 총 점수를 총 학생수로 나눈다.
print(average) # 평균 79.00이 출력된다.
```

for문을 이용하여 먼저 총 점수를 구한 후 총 점수를 총 학생수로 나누어 평균점수를 구한다.

[풀이4] 혈액형

헬액형 별로 합계를 구해야 하므로 딕셔너리를 이용하는 것이 유리하다. 딕셔너리를 사용할 때는 키 값의 유무에 주의하여 코딩해야 한다.

```
data = ['A', 'B', 'A', 'O', 'AB', 'AB', 'O', 'A', 'B', 'O', 'B', 'AB']
result = {}
for blood_type in data:
    if blood_type in result: # 키 값이 존재하는 경우에는 기존 값에 더함
        result[blood_type] += 1
    else: # 키 값이 없는 경우에는 새로운 키 생성
        result[blood_type] = 1

print(result) # {'A': 3, 'B': 3, 'O': 3, 'AB': 3} 출력
```

[풀이5] 리스트 내포1

```
>>> numbers = [1, 2, 3, 4, 5]
>>> result = [n*2 for n in numbers if n%2==1]
>>> result
[2, 6, 10]
```

[풀이6] 리스트 내포2

```
>>> vowels = 'aeiou'
>>> sentence = 'Life is too short, you need python'
>>> ''.join([a for a in sentence if a not in vowels])
'Lf s t shrt, y nd pythn'
```

04-1 함수

[풀이1] 홀수 짝수 판별

```
>>> def is_odd(number):
...     if number % 2 == 1: # 2로 나누었을 때 나머지가 1이면 홀수이다.
...         return True
...     else:
...         return False
...
>>> is_odd(3)
True
>>> is_odd(4)
False
```

람다와 조건부 표현식을 이용하면 다음과 같이 간단하게도 만들 수 있다.

```
>>> is_odd = lambda x: True if x % 2 == 1 else False
>>> is_odd(3)
True
```

[풀이]2 평균값 계산

```
>>> def avg_numbers(*args): # 입력 갯수에 상관없이 사용하기 위해 *args를 이용
...     result = 0
...     for i in args:
...         result += i
...     return result / len(args)
...
>>> avg_numbers(1, 2)
1.5
>>> avg_numbers(1, 2, 3, 4, 5)
3.0
```

[풀이]3 구구단 출력

```
>>> def gugu(n):
...     for i in range(1, 10):
...         print(n*i)
...
>>> gugu(2)
2
4
6
8
10
12
14
16
18
```

[풀이4] 피보나치

n 이 0일 때는 0을 리턴, 1일 때는 1을 리턴한다. n 이 2 이상일 경우에는 이전의 두 값을 더하여 리턴한다.

재귀호출을 이용하면 피보나치 함수를 다음과 같이 간단하게 작성할 수 있다. (재귀호출은 함수 내에서 다시 똑같은 함수를 호출하는 것을 의미하는 용어이다. 아래의 예를 보면 fib함수내에서 다시 fib함수를 호출한다.)

```
def fib(n):
    if n == 0 : return 0 # n이 0일 때는 0을 리턴
    if n == 1 : return 1 # n이 1일 때는 1을 리턴
    return fib(n-2) + fib(n-1) # n이 2 이상일 때는 그 이전의 두 값을 더하여 리턴
(재귀호출을 사용)

for i in range(10):
    print(fib(i))
```

0부터 9까지의 피보나치 수열의 결과값을 출력하여 그 값을 확인해 보았다.

출력결과는 다음과 같다.

```
0
1
1
2
3
5
8
13
21
34
```

[풀이5] 5보다 큰 수만

```
>>> myfunc = lambda numbers: [number for number in numbers if number > 5]
>>> myfunc([2,3,4,5,6,7,8])
[6, 7, 8]
```

04-2 사용자 입력과 출력

[풀이1] 두 수의 합은?

```
input1 = input("число1을 숫자로 입력하세요:")
input2 = input("число2를 숫자로 입력하세요:")

total = int(input1) + int(input2) # 입력은 항상 문자열이므로 숫자로 바꾸어 주어야 한다.
print("두 수의 합은 %s입니다." % total)
```

출력 결과는 다음과 같다.

```
число1을 숫자로 입력하세요:3
число2를 숫자로 입력하세요:6
두 수의 합은 9입니다.
```

[풀이2] 숫자의 총합

```

user_input = input("ㅅㅜㅅㅅ | ㄹ—ㄹ ㅇ | ㅂㄹ ㅋ ㄱ ㅎ | ㅅ ㅋ ㅇ ㅍ:")
numbers = user_input.split(",")
total = 0
for n in numbers:
    total += int(n) # 입력은 문자열이므로 숫자로 변환해야 한다.
print(total)

```

수행결과

```

ㅅㅜㅅㅅ | ㄹ—ㄹ ㅇ | ㅂㄹ ㅋ ㄱ ㅎ | ㅅ ㅋ ㅇ ㅍ:65,45,2,3,45,8
168

```

[풀이3] 문자열 출력

```

>>> print("you" "need" "python")
youneedpython
>>> print("you"+"need"+"python")
youneedpython
>>> print("you", "need", "python") # 콤마가 있는 경우 공백이 삽입되어 더해진다.
you need python
>>> print("."join(["you", "need", "python"]))
youneedpython

```

[풀이4] 한줄 구구단

```

user_input = input("ㄱㅜㄱㅜㄷ | ㄴ ㅇ—ㄹ ㅊㅜㄹㄹ ㅋ ㄱ ㅎ | ㄹ ㅅㅜㅅㅅ | ㄹ—ㄹ
ㅇ | ㅂㄹ ㅋ ㄱ ㅎ | ㅅ ㅋ ㅇ ㅍ(2~9):")
dan = int(user_input) # 입력 문자열을 숫자로 변환
for i in range(1, 10):
    print(i*dan, end= ' ') # 한줄로 출력하기 위해 줄바꿈 문자 대신 공백문자를 마지막에
    출력한다

```

04-3 파일 읽고 쓰기

[풀이1] 파일 읽고 출력하기

질문의 예와같이 파일을 닫지 않은 상태에서 다시 열면 파일에 저장한 데이터를 읽을 수 없다. 따라서 열려진 파일 객체를 close로 닫아준 후 다시 열어서 파일의 내용을 읽어야 한다.

```
f1 = open("test.txt", 'w')
f1.write("Life is too short!")
f1.close() # 열려진 파일 객체를 닫는다.

f2 = open("test.txt", 'r')
print(f2.read())
f2.close()
```

또는 다음과 같이 close를 명시적으로 할 필요가 없는 with구문을 이용한다.

```
with open("test.txt", 'w') as f1:
    f1.write("Life is too short!")

with open("test.txt", 'r') as f2:
    print(f2.read())
```

[풀이2] 파일저장

기존 내용을 유지하고 새로운 내용을 덧붙이기 위해서 다음과 같이 'a' 모드를 사용해야 한다.

```
user_input = input("스 | 스 | o 흥 | e | l | H | o 표 o o — e | o | B | e | ㅋ | ㄱ | 흥 | s | ㅋ | o 표 :")
f = open('test.txt', 'a') # 내용을 추가하기 위해서 'a'를 사용
f.write(user_input)
f.write("\n") # 입력된 내용을 줄단위로 구분하기 위해 줄바꿈 문자 삽입
f.close()
```

[풀이3] 역순 저장

파일 객체의 readlines를 이용하여 모든 라인을 읽은 후에 reversed를 이용하여 역순으로 정렬한 후에 다시 파일에 저장한다.

```

f = open('abc.txt', 'r')
lines = f.readlines() # 모든 라인을 읽음
f.close()

rlines = reversed(lines) # 읽은 라인을 역순으로 정렬

f = open('abc.txt', 'w')
for line in rlines:
    line = line.strip() # 포함되어 있는 줄바꿈 문자 제거
    f.write(line)
    f.write('\n') # 줄바꿈 문자 삽입
f.close()

```

[풀이4] 파일 수정

파일을 모두 읽은 후에 문자열의 replace를 이용하여 java라는 문자열을 python으로 변경하여 저장한다.

```

f = open('test.txt', 'r')
body = f.read()
f.close()

body = body.replace('java', 'python')

f = open('test.txt', 'w')
f.write(body)
f.close()

```

[풀이5] 평균값 구하기

```

f = open("sample.txt")
lines = f.readlines() # sample.txt를 줄단위로 모두 읽는다.
f.close()

total = 0
for line in lines:
    score = int(line) # 줄에 적힌 점수를 숫자형으로 변환한다.
    total += score
average = total / len(lines)

f = open("result.txt", "w")
f.write(str(average))
f.close()

```

sample.txt의 점수를 모두 읽기 위해 파일을 열고 readlines를 이용하여 각 줄의 점수 값을 모두 읽어 들여 총 점수를 구한다. 총 점수를 sample.txt 파일의 라인(Line) 수로 나누어 평균값을 구한 후 그 결과를 result.txt 파일에 쓴다. 숫자 값은 result.txt 파일에 바로 쓸 수 없으므로 str함수를 이용하여 문자열로 변경한 후 파일에 쓴다.

05-1 클래스

[풀이]1 Calculator 1

메서드의 첫번째 입력인수는 항상 self이어야 하므로 다음과 같이 클래스를 수정해야 한다.

```

class Calculator:
    def __init__(self):
        self.value = 0

    def add(self, val): # 첫번째 입력인수 self가 있어야 한다.
        self.value += val

```

[풀이]2 Calculator 2

생성자 메서드에 입력인수가 있으므로 객체를 생성할 때 입력인수에 해당되는 값을 전달해야 한다.

즉, 실행코드를 다음과 같이 작성해야 한다.

```
cal = Calculator(0) # 초기값으로 0을 전달
cal.add(3)
cal.add(4)

print(cal.value)
```

또는 생성자를 다음과 같이 변경하여 초기값을 디폴트로 지정해 줄 수도 있다.

```
class Calculator:
    def __init__(self, init_value=0): # 초기값 init_value에 0을 디폴트로 지정
        self.value = init_value

    def add(self, val):
        self.value += val
```

위와 같이 수정하면 다음과 같이 초기값 전달없이 수행이 가능하다.

```
cal = Calculator()
cal.add(3)
cal.add(4)

print(cal.value)
```

[풀이]3 UpgradeCalculator

다음과 같이 Calculator 클래스를 상속하는 UpgradeCalculator 클래스를 만들고 minus 메서드를 구현한다.

```
class UpgradeCalculator(Calculator):
    def minus(self, val):
        self.value -= val
```

[풀이]4 MaxLimitCalculator

Calculator 클래스를 상속하고 add메서드를 오버라이딩하여 다음과 같은 클래스를 만든다.

```
class MaxLimitCalculator(Calculator):
    def add(self, val):
        self.value += val
        if self.value > 100:
            self.value = 100
```

[풀이]5] Calculator 3

```
class Calculator:
    def __init__(self, numberList):
        self.numberList = numberList

    def sum(self):
        result = 0
        for num in self.numberList:
            result += num
        return result

    def avg(self):
        total = self.sum()
        return total / len(self.numberList)
```

05-2 모듈

[풀이]1] 모듈 사용하기 1

파이썬 shell에서 mymod.py라는 모듈을 인식하기 위해서는 다음과 같은 3가지 방법이 있을 수 있다.

1) sys 모듈 사용하기

다음과 같이 sys.path에 c:\doit이라는 디렉토리를 추가하면 c:\doit이라는 디렉토리에 있는 mymod 모듈을 사용할 수 있게 된다.

```
>>> import sys
>>> sys.path.append("c:/doit")
>>> import mymod
```

2) PYTHONPATH 환경변수 사용하기

다음처럼 PYTHONPATH 환경변수에 c:\doit 디렉토리를 지정하면 c:\doit 디렉토리에 있는 mymod 모듈을 사용할 수 있게 된다.

```
C:\Users\home>set PYTHONPATH=c:\doit
C:\Users\home>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type "help", "copyright", "credits" or "license" for more information.
>>> import mymod
```

3) 현재 디렉토리 이용하기

파이썬 shell을 mymod.py 가 있는 위치로 이동하여 실행해도 mymod 모듈을 이용할 수 있게 된다. 왜냐하면 sys.path 에는 현재디렉토리인 . 이 항상 포함되어 있기 때문이다.

```
C:\Users\home>cd c:\doit
C:\doit>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type "help", "copyright", "credits" or "license" for more information.
>>> import mymod
```

[풀이2] 모듈 작성

c:\doit 이라는 디렉토리를 생성하고 mymod.py라는 파일을 다음과 같이 작성한다.

```
def mysum(a, b):
    return a+b
```

[풀이3] 모듈 사용하기 2

10이라는 숫자가 출력되는 원인은 import mymod 수행 시 mymod.py 모듈 내에 테스트 코드로 작성된 print(mysum(3, 7)) 이라는 문장이 수행되었기 때문이다.

다음과 같이 mymod.py 모듈을 먼저 수정해 보자.

```

def mysum(a, b):
    return a+b

# test
print(mysum(3, 7)) # 10을 출력
print(__name__) # mymod.py를 직접 수행시에는 __main__ 출력, import 시에는 mymod
출력

```

마지막 문장에 `__name__`이라는 파이썬 내부 변수를 출력해 보았다. `mymod.py`가 다음과 같이 직접 호출될 때는 `__main__`이라는 값이 출력된다.

```

c:\doit> python mymod.py
10
__main__

```

이와 반대로 파이썬 shell을 이용하여 `import` 할 때는 다음과 같이 `mymod`라는 모듈명이 출력된다.

```

>>> import sys
>>> sys.path.append("c:/doit")
>>> import mymod
10
mymod

```

따라서 이렇게 `import` 시에 파이썬 모듈에 작성된 테스트성 코드가 실행되는 것을 방지하려면 `__name__`의 값을 이용하여 다음과 같이 수정해야 한다.

```

def mysum(a, b):
    return a+b

# test
if __name__ == '__main__': # 직접 호출될 경우
    print(mysum(3, 7)) # 10을 출력

```

`mymod.py` 모듈을 위와 같이 수정하면 이제 `import` 시에는 10이라는 숫자가 출력되지 않고 직접 호출될 경우에만 10을 출력하게 된다.

05-3 패키지

[풀이1] 패키지 사용 1

3)은 오류가 발생한다. 모듈을 import 하기 위해서는 모듈이 포함되도록 import를 진행해야만 한다. 따라서 echo모듈을 사용하기 위해서는 echo 모듈이 포함되도록 1번, 2번, 4번과 같은 방법으로 import를 해 주어야만 한다.

[풀이2] 패키지 사용2

다음처럼 부모 디렉터리를 의미하는 .. 을 이용하여 import 한다.

```
from ..sound.echo import echo_test
echo_test()
```

05-4 예외처리

[풀이1] 예외처리 1

- 1) 은 “a”라는 문자열과 숫자는 서로 더할 수 있는 자료형이 아니기 때문에 다음과 같은 Type-Error가 발생한다.

```
>>> "a"+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

“a1”과 같은 문자열을 생성하기 위해서는 다음과 같이 수정해야 한다.

```
>>> "a"+str(1)
'a1'
```

- 2) a[3]는 a리스트의 4번째 값을 의미하므로 다음과 같은 IndexError가 발생한다.

```
>>> a = [1, 2, 3]
>>> a[3]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

a리스트의 3번째 값을 얻기 위해서는 다음과 같이 수정해야 한다.

```
>>> a = [1, 2, 3]
>>> a[2]
3
```

[풀이2] 예외처리 2

“no error”라는 문자열이 출력된다. a[-3]은 1이라는 값이고 try문 내에서 예외가 발생하지 않았기 때문에 else구문이 실행되어 “no error”가 출력된다.

[풀이3] 예외처리 3

11이 출력된다.

1. result의 초기값은 3이다.
2. try문의 result += 1에 의해서 result는 4가 된다.
3. 오류가 발생하지 않았기 때문에 else 구문에 의해서 result는 7이 된다.
4. 오류의 유무에 상관없이 finally는 실행되므로 result는 11이 된다.
5. print(result) 가 수행되어 result의 최종값인 11이 출력된다.

[풀이4] 예외처리 4

8이 출력된다.

1. result의 초기값은 0이다.
2. try 문 안의 [1, 2, 3][3]이라는 문장 수행 시 IndexError가 발생하여 except IndexError: 구문으로 이동하게 되어 result에 3이 더해져 3이 된다.
3. 오류가 발생했으므로 else구문은 실행되지 않는다.
4. 최종적으로 finally 구문이 실행되어 result에 5가 더해져 8이 된다.
5. print(result) 가 수행되어 result의 최종값인 8이 출력된다.

05-5 내장 함수

[풀이]1] 내장함수

1)

```
>>> all([1, 2, abs(-3)-3])
False
```

`abs(-3)`은 -3 의 절대값이므로 3 이되어 `all([1, 2, 0])` 이 되고 리스트의 요소값중 0 이 있기 때문에 `all` 내장함수의 결과는 `False`가 된다.

2)

```
>>> chr(ord('a')) == 'a'
True
```

`ord('a')`의 결과는 97 이되어 `chr(97)`로 치환된다. `chr(97)`의 결과는 다시 '`a`'가 되므로 '`a`' == '`a`'가 되어 `True`가 리턴된다.

[풀이]2] enumerate

다음과 같이 `enumerate` 내장함수를 이용하면 리스트의 순서값을 구할 수 있다.

```
>>> d = dict()
>>> for i, v in enumerate(['a', 'b', 'c']):
...     d[i] = v
...
>>> d
{0: 'a', 1: 'b', 2: 'c'}
```

[풀이]3] filter와 lambda

음수를 제거하기 위한 `filter`의 함수로 `lambda`함수를 다음과 같이 만들어 실행한다.

```
>>> list(filter(lambda x:x>0, [1, -2, 3, -5, 8, -3]))
[1, 3, 8]
```

[풀이]4] 16진수를 10진수로 변환

int 내장함수를 다음과 같이 실행한다.

```
>>> int('0xea', 16)
234
```

[풀이]5] map과 lambda

입력에 항상 3을 곱하여 리턴해 주는 lambda함수를 다음과 같이 구현하고 map과 조합하여 실행 한다.

```
>>> list(map(lambda x:x*3, [1,2,3,4]))
[3, 6, 9, 12]
```

[풀이]6] 최대값과 최소값

리스트의 최대값은 max, 최소값은 min 내장함수를 이용하여 다음과 같이 구한다.

```
>>> a = [-8, 2, 7, 5, -3, 5, 0, 1]
>>> max(a) + min(a)
-1
```

[풀이]7] 소수점 반올림

round 내장함수를 이용하면 다음과 같이 반올림하여 소수점 4자리까지 표시할 수 있다.

```
>>> round(17/3, 4)
5.6667
```

[풀이]8] zip

zip 내장함수를 이용하여 다음과 같이 실행한다.

```
>>> list(zip([1,2,3,4], ['a','b','c','d']))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

05-6 외장 함수

[풀이]1] sys.argv

다음처럼 sys모듈의 argv를 이용하여 명령행 입력값 모두를 차례로 더해 준다.

```
import sys

numbers = sys.argv[1:] # 파일명을 제외한 명령행 입력 모두

result = 0
for number in numbers:
    result += int(number)
print(result)
```

[풀이]2] os

다음처럼 os 모듈의 chdir을 이용하여 C:\doit이라는 디렉터리로 이동한다.

```
>>> import os
>>> os.chdir("c:/doit")
```

그리고 다음처럼 os모듈의 popen을 이용하여 시스템 명령어인 dir을 수행한다.

```
>>> result = os.popen("dir")
```

popen의 결과를 출력하기 위해 다음과 같이 수행한다.

```
>>> print(result.read())
...
abc.txt
bidusource.html
...
```

[풀이]3] glob

다음과 같이 glob모듈을 사용한다.

```
>>> import glob
>>> glob.glob("c:/doit/*.py")
['c:/doit/doit01.py', 'c:/doit/test.py']
```

[풀이]4] time

time모듈의 strftime을 이용하여 다음과 같이 작성한다.

```
>>> import time
>>> time.strftime("%Y/%m/%d %H:%M:%S") # %Y:년, %m:월, %d:일, %H:시, %M:분,
%S:초
'2018/04/05 10:56:27'
```

[풀이]5] random

random모듈의 randint를 이용하여 다음과 같이 작성한다.

```
import random

result = []
while len(result) < 6:
    num = random.randint(1, 45) # 1부터 45까지의 난수 발생
    if num not in result:
        result.append(num)

print(result)
```

[풀이]6] namedtuple

Student클래스를 namedtuple로 다음과 같이 재 구성할 수 있다.

```
from collections import namedtuple

Student = namedtuple("Student", ["name", "score"])
a = Student("한국 | 미국", 30)

print(a.name)
print(a.score)
```

A-2 종합문제 풀이

[풀이]1] 이름 분석

```

data = '이의덕,이재명,권종수,이재수,박철호,강동희,이재수,김지석,최승만,이성만,박영희,
박수호,전경식,송우환,김재식,이유정'

names = data.split(",")
park = 0
kim = 0

for name in names:
    if name.startswith('박'):
        park += 1
    elif name.startswith('김'):
        kim += 1

print(kim, park) # 1. 김씨와 박씨 count 계산결과 출력
print(names.count('이재수')) # 2. "이재수"란 이름 count 계산결과 출력

names = list(set(names))
print(names) # 3. 중복을 제거한 이름을 출력
print(sorted(names)) # 4. 중복을 제거한 이름을 오름차순으로 정렬하여 출력

```

출력 결과

```

2 3
2
['김재식', '박철호', '강동희', '최승만', '이성만', '김지석', '이재수', '이재명', '',
'박수호', '이의덕', '전경식', '송우환', '권종수', '박영희', '이유정']
['강동희', '권종수', '김재식', '김지석', '박수호', '박영희', '박철호', '송우환', '',
'이성만', '이유정', '이의덕', '이재명', '이재수', '전경식', '최승만']

```

[풀이2] 합의 제곱과 제곱의 합의 차

합의 제곱은 1부터 100까지의 숫자를 더한 후에 제곱을 해 주고 제곱의 합은 제곱한 값을 계속 더해주어 구한다.

```
sum1 = 0
sum2 = 0

for i in range(1, 100+1):
    sum1 += i
    sum2 += i**2 # 스 || 그 모 ○ -| 흐 | 흐

sum1 = sum1 ** 2 # 흐 | 흐 ○ -| 스 || 그 모

print(sum1 - sum2)
```

출력 결과

```
25164150
```

[풀이3] 1부터 100까지의 각 숫자의 갯수 구하기

다음과 같이 숫자를 문자열로 바꾸고 문자열의 문자 하나씩 루프를 돌며 해당문자의 갯수를 증가시킨다.

```
from collections import defaultdict

d = defaultdict(int)
for number in range(1, 100+1):
    for c in str(number):
        d[c] += 1

print(dict(d))
```

출력 결과

```
{'1': 21, '2': 20, '3': 20, '4': 20, '5': 20, '6': 20, '7': 20, '8': 20, '9': 20, '0': 11}
```

[풀이4] DashInsert

다음 프로그램의 주석문을 참고하자.

```
data = "4546793"

numbers = list(map(int, data)) # 스트링을 리스트로 변환한 후 숫자로 변환
# | 한수는 그대로 두자

result = []

for i, num in enumerate(numbers):
    result.append(str(num))
    if i < len(numbers)-1: # 짝수인 경우에만 다음 수를 체크
        is_odd = num % 2 == 1 # 짝수인지 확인
        is_next_odd = numbers[i+1] % 2 == 1 # 다음 수가 짝수인지 확인
        if is_odd and is_next_odd: # 짝수인 경우에만 -
            result.append("-")
        elif not is_odd and not is_next_odd: # 짝수가 아닌 경우에만 *
            result.append("*")

print("".join(result))
```

[풀이5] 문자열 압축하기

먼저 입력 문자열의 문자를 체크하여 동일한 문자가 들어올 경우에는 해당 문자의 숫자값을 증가시킨다. 만약 다른 문자가 들어올 경우에는 해당 문자의 숫자값을 1로 초기화하는 방법을 사용하여 작성한 코드이다.

```

def compress_string(s):
    _c = ""
    cnt = 0
    result = ""
    for c in s:
        if c != _c:
            _c = c
            if cnt: result += str(cnt)
            result += c
            cnt = 1
        else:
            cnt += 1
    if cnt: result += str(cnt)
    return result

print (compress_string("aaabbccccc")) #a3b2c6a1 출력

```

[풀이]6] Duplicate Numbers

```

def chkDupNum(s):
    result = []
    for num in s:
        if num not in result:
            result.append(num)
        else:
            return False
    return len(result) == 10

print(chkDupNum("0123456789")) # True 리턴
print(chkDupNum("01234")) # False 리턴
print(chkDupNum("01234567890")) # False 리턴
print(chkDupNum("6789012345")) # True 리턴
print(chkDupNum("012322456789")) # False 리턴

```

리스트 자료형을 이용하여 중복된 값이 있는지 먼저 조사한다. 중복된 값이 있을 경우는 False를

리턴한다. 최종적으로 중복된 값이 없을 경우 0~9까지의 숫자가 모두 사용되었는지 판단하기 위해 입력 문자열의 숫자값을 저장한 리스트 자료형의 총 개수가 10인지를 조사하여 10일 경우는 True, 아닐 경우는 False를 리턴한다.

[풀이7] 모스 부호 해독

```
dic = {
    '.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E', '..-.': 'F',
    '--.': 'G', '....': 'H', '...': 'I', '---': 'J', '-.-': 'K', '-...': 'L',
    '---': 'M', '-.': 'N', '---': 'O', '-.-': 'P', '--.-': 'Q', '-.-': 'R',
    '...': 'S', '-': 'T', '---': 'U', '...-': 'V', '-.-': 'W', '-..-': 'X',
    '-.--': 'Y', '--..': 'Z'
}

def morse(src):
    result = []
    for word in src.split(" "):
        for char in word.split(" "):
            result.append(dic[char])
        result.append(" ")
    return "".join(result)

print(morse('.... . . . - . . . . --. .... . . - . - . - - - .--'))
```

모스 부호 규칙 표를 딕셔너리로 작성한 후 입력에 해당되는 모스 부호 문자열을 먼저 단어(공백 문자 2개)로 구분한다. 그 후 단어(공백문자 1개)를 문자로 구분하여 해당 모스 부호값을 딕셔너리에서 찾아서 그 결과값을 구한다.

[풀이8] 정규식 1

보기 중 이 조건에 해당되는 것은 B이다.

다음은 위 문제의 정규식 매치 결과를 확인해 보는 파이썬 코드이다.

```
import re

p = re.compile("a[.]{3,}b")

print (p.match("accb")) # None
print (p.match("a....b")) # 매치객체 출력
print (p.match("aab")) # None
print (p.match("a.ccb")) # None
```

[풀이9] 정규식 2

정규식 “[a-z]+”은 소문자로 이루어진 단어를 뜻하므로 “5 python”이라는 문자열에서 “python”과 매치될 것이다. 따라서 “python”이라는 문자열의 시작 인덱스(m.start())는 2이고 마지막 인덱스(m.end())는 8이므로 10이 출력된다.

```
import re

p = re.compile('[a-z]+')
m = p.search("5 python")
print(m.start() + m.end()) # 10 출력
```

[풀이10] 정규식 3

전화번호 패턴은 다음과 같이 작성할 수 있다.

```
pat = re.compile("\d{3}[-]\d{4}[-]\d{4}")
```

이 전화번호 패턴 중 뒤의 숫자 4개를 변경할 것이므로 필요한 앞부분을 다음과 같이 그룹핑한다.

```
pat = re.compile("( \d{3}[-]\d{4}) [-]\d{4}")
```

컴파일된 객체 pat에 sub 함수를 이용하여 다음과 같이 문자열을 변경한다.

```

import re

s = """
park 010-9999-9988
kim 010-9909-7789
lee 010-8789-7768
"""

pat = re.compile("(\\d{3}[-]\\d{4}) [-]\\d{4}")
result = pat.sub("\g<1>-####", s)

print(result)

```

[풀이]11] 정규식 4

.com과 .net에 해당되는 이메일 주소만을 매치하기 위해서 이메일 주소의 도메인 부분에 다음과 같은 긍정형 전방탐색 패턴을 적용한다.

```
pat = re.compile(".*[@].*[.](?=com$|net$).*$")
```

다음은 위 패턴을 적용한 파일 코드이다.

```

import re

pat = re.compile(".*[@].*[.](?=com$|net$).*$")

print (pat.match("pahkey@gmail.com"))
print (pat.match("kim@daum.net"))
print (pat.match("lee@myhome.co.kr"))

```

[풀이]12] XML 문서 작성과 저장

```
from xml.etree.ElementTree import Element, SubElement, ElementTree

blog = Element("blog")
blog.attrib["date"] = "20151231"

SubElement(blog, "subject").text = "Why python?"
SubElement(blog, "author").text = "Eric"
SubElement(blog, "content").text = "Life is too short, You need Python!"

ElementTree(blog).write("blog.xml")
```

[풀이]13] json 데이터

json파일을 읽어 딕셔너리로 저장한 후 값을 변경하고 변경된 값을 다시 json파일에 저장한다.

```
import json

with open('myinfo.json') as f:
    data = json.load(f) # json파일을 읽고 딕셔너리로 저장한다.

    data['age'] = 40 # age 값을 40으로 변경

with open('myinfo.json', 'w') as f:
    json.dump(data, f, indent=4) # 딕셔너리를 json 파일로 저장한다.
```

[풀이]14] 요소값으로 소트하기

다음은 operator모듈의 itemgetter함수를 이용하여 소트하는 방법이다. students의 아이템의 2번째 항목으로 소트해야 하기 때문에 sorted의 key함수로 itemgetter(1)을 사용한다.

```

from operator import itemgetter

students = [
    ("홍길동", 22),
    ("김철수", 32),
    ("박영희", 17),
]

students = sorted(students, key=itemgetter(1))

print(students)

```

[풀이]15] 시저 암호화

```

def casar(word, n):
    result = ""
    alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    for a in word:
        index = alpha.index(a)
        result += alpha[(index+n) % 26]
    return result

print(casar("CAT", 5)) # HFY 출력
print(casar("ZOO", 3)) # CRR 출력

```

단어의 각 알파벳의 인덱스를 구한 후 n 만큼 더한 index에 해당되는 알파벳을 구해서 다시 문자열을 재 구성하면 된다. 여기서 주의해야 할 점은 구한 인덱스에서 n 을 더했을 경우 총 알파벳의 갯수인 26자리를 넘어갔을 경우 다시 앞으로 돌아가야 한다는 점이다. 위 예제에서 $\%$ 연산자를 어떻게 사용했는지를 눈여겨 보도록 하자.

B. 부록

부록 챕터에서는 다음과 같은 내용을 다룬다.

- B-1 파이썬 2.7 vs 파이썬 3
- B-2 str과 repr
- B-3 유니코드(unicode)와 인코딩(encoding)
- B-4 pip

B-1 파이썬 2.7 vs 파이썬 3

이 책은 파이썬 2.7 버전과 3 버전에 상관없이 책을 읽을 수 있도록 작성되어졌다. 아래의 몇 가지 큰 차이점만 숙지하면 파이썬 버전에 상관없이 이 책의 예제를 수행하는 데 무리가 없을 것이다.

print

파이썬 3 버전은 출력할 문자열에 괄호를 필요로 한다.

파이썬 3 버전의 예

```
print ("Hello Python")
```

파이썬 2.7 버전의 예

```
print "Hello Python"
```

파이썬 2.7 버전인 경우 파이썬 3 버전처럼 괄호를 사용해도 동일하게 동작한다. (단, 2.7 버전 이하의 파이썬 구버전에서는 오류가 발생할 수 있다.)

줄바꿈 방지

print 문 실행 시 항상 문자열 마지막에 \n 문자가 출력되어 줄바꿈이 일어나게 된다. 이렇게 마지막 문자인 \n을 생략할 수 있는 방법이 있는데 이것또한 파이썬 3 버전과 파이썬 2.7 버전이 서로 다르다.

파이썬 3 버전의 예

```
print ("No new line", end=" ");print ("ok?")
```

파이썬 3 버전의 경우 줄바꿈 문자를 제거하기 위해서 위와 같이 끝 문자를 지정할 수 있는 end 파라미터를 설정하면 된다. 지정하지 않으면 디폴트로 \n 문자가 세팅된다.

파이썬 2.7 버전의 예

```
print "No new line",;print "ok?"
```

파이썬 2.7 버전의 경우 줄바꿈 문자를 제거하기 위해서 문자열의 끝에 콤마(,)를 덧붙이면 된다.

자동 형 변환

파이썬 3의 경우 숫자연산 시 자동으로 형 변환이 된다.

파이썬 3 버전의 예

```
>>> 3 / 4
0.75
```

파이썬 2.7 버전의 예

```
>>> 3 / 4
0
>>> 3 / 4.0
0.75
```

input

파이썬 3 버전의 `input` 내장함수와 파이썬 2.7버전의 `raw_input` 내장함수는 동일하다. 기존 파이썬 2.7의 `input` 내장함수는 파이썬 3부터는 더이상 지원되지 않는다.

파이썬 3 버전의 예

```
>>> name = input("이름을 입력하세요:")
```

파이썬 2.7 버전의 예

```
>>> name = raw_input("이름을 입력하세요:")
```

소스코드 인코딩

파이썬 3 버전부터는 utf-8이 기본 소스코드 인코딩이므로 다음과 같은 문자열을 소스코드 첫줄에서 생략할 수 있다.

```
# -*- coding: utf-8 -*-
```

하지만 utf-8 이 아닌 다른 형태의 소스코드 인코딩을 사용해야 할 경우에는 해당 인코딩을 명시해야 한다. 하지만 파이썬 2.7 버전은 무조건 위와 같은 문자열을 소스코드 첫 줄에 명시해야만 인코딩 오류가 발생하지 않는다.

에러처리

`try ... except...` 에러 처리 시 에러 변수명을 표기하는 방식이 파이썬 버전 3과 버전 2.7이 서로 다르다.

파이썬 3 버전의 예

```
try:  
    4 / 0  
except ZeroDivisionError as e:  
    print(e)
```

파이썬 2.7 버전의 예

```
try:  
    4 / 0  
except ZeroDivisionError, e:  
    print e
```

에러변수 설정 시 파이썬 3 버전은 `as`를 2.7 버전은 콤마(,)를 사용한다.

B-2 str 과 repr

파이썬 내장함수에는 str과 repr이라는 어찌보면 매우 비슷한 기능을 해 주는 함수가 있다. str과 repr은 모두 객체를 문자열로 리턴해 주는 함수이다. 하지만 두개의 함수에는 약간의 차이가 있다.

str과 repr의 차이점

어떤 차이가 있는지 예제를 보며 알아보자. 먼저 숫자에 적용해 보자.

```
>>> a = 123
>>> str(a)
'123'
>>> repr(a)
'123'
```

숫자는 아무런 차이가 없어 보인다. 이번에는 문자열에 적용해 보자.

```
>>> a = "Life is too short"
>>> str(a)
'Life is too short'
>>> repr(a)
'"Life is too short"'
```

문자열은 str과 repr이 다른 결과값을 리턴해 주었다. str은 문자열 그대로를 리턴해 주었는데 repr은 단일인용부호(')가 좌우로 감싸여진 형태의 문자열을 리턴해 주었다.

음 왜 그럴까?

한가지 예를 더 들어보자.

```
>>> a = datetime.datetime(2017, 9, 27)
>>> str(a)
'2017-09-27 00:00:00'
>>> repr(a)
'datetime.datetime(2017, 9, 27, 0, 0)'
```

datetime으로 만들어진 객체는 매우 다른 결과를 리턴해 주었다.

`str`과 `repr`에는 다음과 같은 차이점들이 있다.

구분	<code>str</code>	<code>repr</code>
성격	비공식적인 문자열을 출력	공식적인 문자열을 출력
사용목적	사용자가 보기 쉽게 하기 위해	문자열로 객체를 다시 생성할 수 있기 위해
누구를 위해	프로그램 사용자(end user)	프로그램 개발자(developer)

`repr`의 사용목적을 보면 “문자열로 객체를 다시 생성할 수 있기 위해”라고 되어 있다. 문자열로 객체를 생성하기 위해서는 `eval`함수를 사용한다. 즉, 다음과 같이 `datetime`객체의 `repr`로 생성된 문자열에 다시 `eval`을 수행하면 `datetime`객체가 만들어 쳐야 한다는 말이다.

```
>>> a = datetime.datetime(2017, 9, 27)
>>> b = repr(a)
>>> eval(b)
datetime.datetime(2017, 9, 27, 0, 0)
```

위 예제에서 알아본 문자열도 마찬가지이다.

```
>>> a = "Life is too short"
>>> b = repr(a)
>>> eval(b)
'Life is too short'
```

하지만 `str`으로 리턴된 문자열을 `eval`로 수행했을 때는 다음과 같은 오류가 발생한다.

```
>>> a = "Life is too short"
>>> b = str(a)
>>> eval(b)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<string>", line 1
    Life is too short
    ^
SyntaxError: unexpected EOF while parsing
```

문자열을 `repr`했을 때 왜 단일 인용부호(')가 덧붙여서 나왔는지 이제 이해가 될 것이다.

클래스의 `__str__` 과 `__repr__`

이번에는 사용자가 만든 클래스에서 `repr`과 `str`이 어떻게 적용되는지 확인해 보자.

```
class MyRepr:
    pass

obj = MyRepr()

print(repr(obj))
print(str(obj))
```

`MyRepr`이라는 아무런 내용도 없는 클래스를 작성한 후 `obj`라는 객체를 생성하였다. 그리고 `repr`과 `str`로 해당 객체를 출력해 보았다. 결과는 다음과 같이 출력된다.

```
<__main__.MyRepr object at 0x100656cc0>
<__main__.MyRepr object at 0x100656cc0>
```

`repr`이나 `str`으로 객체 출력시 디폴트 문자열이 출력되는 것을 확인 할 수 있다. 이번에는 `repr` 호출시 의미있는 문자열이 출력되도록 다음과 같이 수정해 보자.

```
class MyRepr:
    def __repr__(self):
        return "Hello MyRepr"

obj = MyRepr()

print(repr(obj))
print(str(obj))
```

클래스에 `__repr__` 메서드를 구현하면 `repr`메서드 호출시 수행되게 된다. 따라서 `repr`로 `obj` 호출시 다음과 같은 문자열이 출력될 것이다.

```
Hello MyRepr
```

마찬가지로 클래스에 `__str__` 메서드를 구현하면 `str`메서드 호출시 수행되게 된다. 하지만 위 예제에서는 `__str__` 메서드를 구현하지 않았는데도 `str(obj)` 출력시 다음과 같은 문자열이 출력되었다.

```
Hello MyRepr
```

이렇게 되는 이유는 `str`메서드 호출시 제일먼저 `__str__` 메서드가 구현되어 있는지 확인하고 없으면 `__repr__` 메서드를 호출하게 되기 때문이다.

이번에는 반대로 `__repr__` 대신 `__str__` 메서드만 구현해 보자.

```
class MyRepr:
    def __str__(self):
        return "Hello MyRepr"

obj = MyRepr()

print(repr(obj))
print(str(obj))
```

수행결과는 다음과 같이 출력된다.

```
<__main__.MyRepr object at 0x100656d68>
Hello MyRepr
```

`repr`은 호출시 `__repr__` 메서드가 없으면 `__str__`메서드가 호출되지 않고 디폴트 값이 출력된다는 점을 알 수 있다. 즉 `str`은 `__str__`이 없을 경우 `__repr__` 을 참조하지만 `repr`은 오직 `__repr__`만 참조한다는 사실을 알 수 있다.

이번에는 `repr` 메서드의 사용목적인 “문자열로 다시 객체를 생성”을 만족하기 위해서 다음과 같이 코드를 수정해 보자. (참고. `repr`의 출력 문자열을 `eval`을 이용하여 다시 객체로 만들 수 있어야 한다는 것은 필수 조건은 아니다. 다만 권고사항 정도라고 보면 된다)

```
class MyRepr:  
    def __repr__(self):  
        return "MyRepr()  
  
    def __str__(self):  
        return "Hello MyRepr"  
  
obj = MyRepr()  
  
obj_repr = repr(obj)  
new_obj = eval(obj_repr)  
print(type(new_obj))
```

repr 호출시 “MyRepr()”을 리턴하여 eval수행시 MyRepr()이 수행되어 새로운 MyRepr 클래스의 객체가 생성되는 것을 확인할 수 있다.

eval로 생성된 new_obj의 type을 출력한 결과는 다음과 같다.

```
<class '__main__.MyRepr'>
```

B-3 유니코드와 인코딩

유니코드의 유래

컴퓨터는 여러분도 알다시피 결국 0과 1이라는 값만을 인식할 수 있는 기계장치이다. 컴퓨터가 문자를 인식할 수 있게 하려면 어떻게 해야 할까? 과거부터 지금까지 사용하는 유일한 방법은 다음과 비슷한 방법의 문자셋을 만드는 것이다.

예를 들어 숫자 97는 ‘a’, 숫자 98은 ‘b’, … 이런식으로 숫자마다 문자를 매핑해 놓으면 컴퓨터는 해당 숫자를 문자열로 인식할 수 있게 될 것이다. 최초 컴퓨터가 발명되었을 때 이런 문자들을 처리하기 위해서 컴퓨터마다 각각의 문자셋을 정해 놓고 문자들을 처리하기 시작했다. 하지만 이렇게 컴퓨터들마다 각각의 문자셋을 사용했더니 데이터 호환이 안되는 문제가 발생하게 되었다. A라는 컴퓨터에서 처리하는 문자셋 규칙이 B라는 컴퓨터에서 처리하는 문자셋 규칙과 같지 않기 때문에 서로 데이터를 주고 받는 등의 일을 할 수가 없게 된 것이다.

이런 문제를 해결하기 위해 미국에서 최초로 문자셋 표준인 아스키(ASCII)라는 것이 탄생하게 된다. 아스키라는 문자셋 규칙을 정하고 이 규칙대로만 문자를 만들면 이 기종 컴퓨터간에도 문제 없이 데이터를 주고 받을 수 있게 되는 것이다. 아스키는 처리할 수 있는 문자의 갯수가 127개였는데 이것은 영어권 국가들에서 사용하는 영문자, 숫자등을 처리하기 위해서는 부족함이 없었다. 하지만 곧 비 영어권 국가들에서도 자신들의 문자를 컴퓨터로 표현해야 하는 요구들이 발생하게 되었다. 아스키는 127개의 문자만을 다룰 수 있기 때문에 아스키를 사용할 수는 없는 노릇이었다. 그래서 곧 서유럽 문자셋인 ISO8859가 등장하게 되고 한국에서는 KSC5601과 같은 문자셋들이 등장하게 된다.

이렇게 나라별로 문자셋이 만들어지고 또 한 나라에서도 여러개의 문자셋이 표준이 되기 위한 치열한 싸움을 벌이기도 하며 문자를 처리하기 위한 방법은 점점 더 계속 복잡해져만 갔다.

가장 결정적인 문제는 하나의 문서에 여러나라의 언어를 동시에 표현할 수 있는 방법이 없었다는 점이다.

이런 문제를 해결하기 위해서 등장한 구세주가 바로 유니코드(Unicode)이다. 유니코드는 모든 나라의 문자를 다 포함하게끔 넉넉하게 설계되었고 곧 세계적인 표준으로 자리잡게 되었다. 이 유니코드라는 규칙을 사용하게 되면 서로 다른 문자셋으로 고생할 일이 이제 없어지게 된 것이다.

유니코드 vs utf-8

앞서 설명한 문자셋(character set)은 규칙이다. 이 규칙대로 바이트 데이터를 만들면 컴퓨터는 해당 문자셋을 해석할 수 있게 된다. 이렇게 문자셋의 규칙대로 바이트를 만드는 방법을 인코딩(encoding)이라고 부른다.

세계적인 표준 문자셋이 유니코드라고 했다. 이 유니코드 문자열을 바이트 문자열로 만드는 방법은 1가지였으면 좋았겠지만 여러가지 이유로 다양한 방법의 인코딩들이 등장하게 된다. 이 중에서 가장 많이 사용되는 것이 바로 utf-8이다. 가장 많이 사용되는 이유로는 다른 인코딩 방법에 비해 데이터의 사이즈가 작다는 점과 기존 ascii와의 하위 호환성을 갖는다는 점을 꼽을 수 있다. (utf-8을 이용하면 이미 아스키로 작성된 데이터라도 변환없이 utf-8 그대로 사용할 수 있게 된다.)

파이썬과 유니코드

이제 파이썬과 유니코드에 대해서 알아보자.

다음과 같은 문자열을 보자.

```
>>> a = "Life is too short"
>>> type(a)
<class 'str'>
```

type명령어를 호출해 보면 문자열은 str클래스의 객체임을 알 수 있다. 파이썬에서 사용되는 문자열은 모두 유니코드 문자열이다. (파이썬 3 버전부터는 모든 문자열을 유니코드로 처리하게 변경되었다.)

인코딩 (encoding)

유니코드 문자열을 인코딩없이 그대로 파일에 적거나 다른 시스템으로 네트워크 전송을 할 수는 없다. 왜냐하면 유니코드 문자열은 단순히 문자셋의 규칙이기 때문이다. 파일에 적거나 다른 시스템으로 유니코드 문자열을 전송하기 위해서는 바이트로 변환해야만 한다. 이렇게 유니코드 문자열을 바이트로 바꾸는 것을 인코딩이라고 한다. 따라서 파일을 읽거나 바이트 문자열을 수신받을 때에는 해당 바이트가 어떤 방식의 인코딩을 사용했는지는 필수적으로 미리 알고 있어야 디코딩이 가능하다.

유니코드 문자열을 바이트 문자열로 바꾸는 방법은 다음과 같다.

```
>>> b = a.encode('utf-8')
>>> b
b'Life is too short'
>>> type(b)
<class 'bytes'>
```

유니코드 문자열을 바이트 문자열로 만들 때에는 위 예처럼 인코딩 방식을 파라미터로 넘겨 주어야 한다. 파라미터를 생략하면 디폴트 값인 utf-8로 동작한다.

`type` 명령어를 호출해 보면 `b` 객체는 `bytes` 클래스의 객체임을 알 수 있다.

이번에는 다음 예제를 보자.

```
>>> a = "한글"
>>> a.encode("ascii")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)
```

위 예는 “한글”이라는 유니코드 문자열을 ascii 방식으로 인코딩하려고 시도하는 예제이다. ascii 문자셋으로는 한글을 표현할 수 없기 때문에 위와 같은 오류가 발생한다.

“한글”이라는 유니코드 문자열을 나타낼 수 있는 문자셋에는 여려개가 있다. 보통은 utf-8로 충분하지만 이미 기존 시스템(legacy system)이 euc-kr과 같은 문자셋을 사용한다면 다음과 같이 euc-kr로 인코딩할 수도 있다.

```
>>> a = '한글'
>>> a.encode('euc-kr')
b'\xc7\xd1\xb1\xdb'
>>> a.encode('utf-8')
b'\xed\x95\x9c\xea\xb8\x80'
```

utf-8로 인코딩 했을 때와는 다른 바이트 문자열이 출력되는 것을 확인할 수 있을 것이다.

디코딩 (decoding)

이번에는 반대로 인코딩된 바이트 문자열을 유니코드 문자열로 변환할 수 있는 디코딩에 대해서 알아보자. 다음 예제처럼 euc-kr로 인코딩된 바이트 문자열은 euc-kr로만 디코딩을 해야 한다.

```
>>> a = '한글'
>>> b = a.encode('euc-kr')
>>> b.decode('euc-kr')
'한글'
```

만약 euc-kr로 인코딩된 바이트 문자열을 utf-8로 디코딩하려고 한다면 어떻게 될까?

```
>>> b.decode('utf-8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc7 in position 0:
invalid continuation byte
```

잘못된 문자셋으로 디코딩 하려고 하면 위와 같은 오류가 발생하게 된다.

입출력과 인코딩

인코딩 관련해서 개발자들이 가장 고생하는 부분은 바로 데이터의 입출력 관련해서이다. 이것 역시 문자열과 인코딩에 대한 개념만 확실히 이해하면 어렵지 않지만 이것들에 대한 이해 없이 무작정 인코딩, 디코딩을 남발하다가는 다종 인코딩되거나 다종 디코딩되면서 문자열이 꼬여 버리는 불상사가 발생하기도 한다.

파일을 읽거나 네트워크를 통해 데이터를 입력받을 때 많은 사람들이 추천하는 방법은 다음과 같다.

1. 입력으로 받은 바이트 문자열을 가능한한 가장 빨리 유니코드 문자열로 디코딩 할 것.
2. 변환된 유니코드 문자열로만 함수나 클래스등에서 사용할 것.
3. 입력에 대한 결과를 전송하는 마지막 부분에서만 유니코드 문자열을 인코딩해서 리턴할 것.

위와 같은 규칙을 준수한다면 인코딩 관련해서 큰 어려움이 없을 것이다.

다음은 euc-kr 문자셋으로 작성되어진 파일을 읽고 변경하여 저장하는 예제이다.

```
# 1. euc-kr로 작성된 파일 읽기
with open('euc_kr.txt', encoding='euc-kr') as f:
    data = f.read() # 유니코드 문자열

# 2. unicode 문자열로 프로그램 수행하기
data = data + "\n" + "추가 문자열"

# 3. euc-kr로 수정된 문자열 저장하기
with open('euc_kr.txt', encoding='euc-kr', mode='w') as f:
    f.write(data)
```

파일을 읽는 open 함수에는 encoding을 지정하여 파일을 읽을 수 있는 기능이 있다. 이 때 읽어들인 문자열은 유니코드 문자열이 된다. 마찬가지로 파일을 작성할 때도 encoding을 지정하여 파일을 작성할 수 있다. 만약 encoding항목을 생략하면 디폴트로 utf-8^{o]} 지정된다.

소스코드 인코딩

파이썬 shell 이 아닌 에디터로 파일을 코딩할 경우 소스 코드의 인코딩은 매우 중요하다. 소스 코드의 인코딩이란 소스 코드 파일이 현재 어떤 방식으로 인코딩되었지를 의미한다.

우리가 위 예제에서 알아보았듯이 파일은 utf-8 인코딩으로 저장할 수도 있고 euc-kr로 저장할 수도 있었다. 소스 코드도 파일이므로 인코딩 타입이 반드시 존재한다. 파일은 소스코드의 인코딩을 명시하기 위해 소스 코드 제일 상단에 다음과 같은 문장을 넣어 주어야 한다.

```
# -*- coding: utf-8 -*-
```

만약 소스코드가 utf-8로 인코딩된 파일이라면 위와 같이 작성하면 되고 euc-kr로 인코딩된 경우라면 다음과 같이 작성해야 한다. (파이썬 3.0 부터는 utf-8이 디폴트이므로 utf-8로 인코딩된 소스 코드인 경우 위 문장을 생략해도 된다.)

```
# -*- coding: euc-kr -*-
```

만약 소스코드는 euc-kr로 인코딩되었는데 파일상단에는 utf-8로 명시되어져 있다면 문자열 처리하는 부분에서 인코딩 관련한 오류가 발생할 것이다.

B-4 파이썬 패키지 설치 (pip)

pip 이란?

pip은 파이썬 모듈이나 패키지를 쉽게 설치할 수 있도록 도와주는 도구이다.

pip을 이용해서 파이썬 프로그램을 설치하면 의존성 있는 모듈이나 패키지를 함께 설치해 주기 때문에 매우 편리하다. 예를 들어 B라는 파이썬 패키지를 설치하려면 반드시 A라는 패키지가 먼저 설치되어 있어야 하는 규칙이 있다고 할 때 pip을 이용하면 매우 편리하다. pip을 이용하여 B 패키지를 설치할 때 A 패키지가 자동으로 설치가 되기 때문이다. pip이 없었던 과거에는 이런 의존성을 개발자가 미리 파악하고 설치해야하는 불편함이 있었다.

(참고. 우분투의 apt-get을 이용하면 우분투 시스템에 소프트웨어를 편리하게 설치할 수 있다. pip은 apt-get과 매우 비슷한 역할을 한다. apt-get이 우분투 프로그램을 설치한다면 pip은 파이썬 프로그램을 설치하는 것이다.)

파이썬 3 버전을 사용하는 사용자는 pip이 이미 설치되어 있기 때문에 pip을 따로 설치할 필요가 없다. (파이썬 2.7 버전 사용자는 pip이 자동으로 설치가 되지 않기 때문에 수동으로 pip을 설치해 주어야 한다.)

pip 수동 설치

다음은 수동으로 pip을 설치하는 방법이다. (이미 언급했지만 파이썬 3 사용자는 pip수동 설치가 필요없다.)

먼저 다음의 URL에서 get-pip.py 파일을 다운로드 받는다.

<https://bootstrap.pypa.io/get-pip.py>

다운로드 받은 후에 다음의 명령어를 실행하면 pip 설치가 자동으로 진행된다.

```
python get-pip.py
```

pip 사용

pip사용법에 대해서 간단하게 알아보자.

※ 이 책은 김준석(wnstjr8717@naver.com)님이 구매하신 문서입니다.

pip install

PyPI(Python Package Index)는 파이썬 소프트웨어가 모여 있는 저장공간이다. 현재 이곳에는 100,000건 정도의 파이썬 패키지가 등록되어 있다. 이 곳에 등록된 파이썬 패키지는 누구나 다운받아 사용할 수 있다. 이곳에서 직접 다운받아서 설치해도 되지만 pip을 이용하면 다음과 같이 간편하게 설치를 할 수 있다.

```
pip install SomePackage
```

(SomePackage는 다운받을 수 있는 임의의 패키지를 의미한다.)

pip uninstall

설치한 패키지를 삭제하고 싶다면 다음의 명령어로 삭제가 가능하다.

```
pip uninstall SomePackage
```

특정 버전으로 설치하기

다음과 같이 특정버전의 파이썬 패키지를 설치 할 수도 있다.

```
pip install SomePackage==1.0.4
```

위 명령어를 수행하면 1.0.4 버전의 SomePackage를 설치하게 된다.

다음처럼 버전을 생략한 경우 최종버전을 설치하게 된다.

```
pip install SomePackage
```

최신 버전으로 업그레이드하기

패키지를 최신버전으로 업그레이드 하려면 다음과 같이 실행한다.

```
pip install --upgrade SomePackage
```

설치된 패키지 확인하기

pip을 이용하여 설치한 패키지들의 목록을 출력하려면 다음과 같이 실행한다.

```
pip list
```

다음과 같이 설치된 패키지 목록이 출력될 것이다.

```
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

pip 따라해 보기

pip을 이용하여 flask를 설치하는 실례를 보도록 하자. flask는 유명한 파이썬 웹 프레임워크 중의 하나이다.

다음과 같이 flask를 설치해 보자. 아마도 아래와 비슷한 화면이 출력되며 설치가 진행될 것이다.

```
$ pip install flask
...
Downloading Flask-0.12-py2.py3-none-any.whl (82kB)
100% || 92kB 548kB/s
Collecting Jinja2>=2.4 (from flask)
Downloading Jinja2-2.9.5-py2.py3-none-any.whl (340kB)
100% || 348kB 2.3MB/s
Collecting Werkzeug>=0.7 (from flask)
Downloading Werkzeug-0.11.15-py2.py3-none-any.whl (307kB)
100% || 317kB 1.9MB/s
...
Successfully installed Jinja2-2.9.5 MarkupSafe-0.23 Werkzeug-0.11.15
click-6.7 flask-0.12 itsdangerous-0.24
```

flask 설치시 의존성있는 패키지인 Jinja2, Werkzeug등의 패키지도 함께 설치되는 것을 확인할 수 있을 것이다.

이번에는 uninstall을 수행해 보자. 삭제되는 파일을 보여주고 진행할것인지 묻는 프로프트(y/n)가 나타난다. 'y'를 입력하여 진행하면 flask가 uninstall되는 것을 확인할 수 있을 것이다.

```
$ pip uninstall flask
Uninstalling Flask-0.12:
...
/Library/Python/2.7/site-packages/flask/views.py
/Library/Python/2.7/site-packages/flask/views.pyc
/Library/Python/2.7/site-packages/flask/wrappers.py
/Library/Python/2.7/site-packages/flask/wrappers.pyc
/usr/local/bin/flask
Proceed (y/n)? y
Successfully uninstalled Flask-0.12
```

이때 flask 설치시 함께 설치되었던 의존성 패키지들은 함께 삭제되지 않는다.

pip을 이용한 개발환경 구축하기

여러명이서 함께 파이썬 프로그램을 만들 때는 개발시 필요한 의존성 패키지들을 반드시 동일하게 맞추고 개발해야 한다. 예를 들어 A는 1.0 버전의 SomePackage를 이용하여 개발을 하고 B는 1.1 버전으로 개발을 한다면 분명 SomePackage의 다른부분으로 인해 오류가 발생할 확율이 높을 것이다.

pip을 이용하여 협업을 통한 파이썬 개발시 필요한 의존성 패키지를 동일하게 맞추는 방법에 대해서 알아보자.

먼저 최초 개발자 A는 다음과 같이 pip을 이용하여 의존성 있는 파일리스트를 만들 수 있다.

```
pip freeze > requirements.txt
```

위 명령을 수행하면 requirements.txt 파일 내에는 다음과 같이 개발시 필요한 패키지들이 저장된다.

```
docutils==0.9.1
Jinja2==2.6
Pygments==1.5
Sphinx==1.1.2
```

패키지명과 버전이 포함된 파일임을 알 수 있다.

이제 A는 B에게 requirements.txt 파일을 제공하면 B는 다음과 같이 실행하여 파이썬 패키지를 설치할 수 있다.

```
pip install -r requirements.txt
```

위 명령을 수행하면 requirements.txt 파일내에 정의된 패키지들을 기술된 버전으로 설치한다.

이런 과정을 통해 A와 B는 동일한 파이썬 개발환경을 구축할 수 있게 되어 의존성 패키지로 인한 오류가 발생하지 않을 것이다.