
점프 투 파이썬

박응용

Contents

문서정보	1
프로그램 코드의 추가와 삭제	2
00 장 들어가기 전에	3
00-1 머리말	4
00-2 저자소개	6
저서	6
유튜브	6
01 장 파이썬이란 무엇인가?	7
01-1 파이썬이란?	8
01-2 파이썬의 특징	9
파이썬은 인간다운 언어이다	9
파이썬은 문법이 쉬워 빠르게 배울 수 있다	9
파이썬은 무료이지만 강력하다	10
파이썬은 간결하다	10
파이썬은 프로그래밍을 즐기게 해준다	11
파이썬은 개발 속도가 빠르다	11
01-3 파이썬으로 무엇을 할 수 있을까?	12
파이썬으로 할 수 있는 일	12
시스템 유틸리티 제작	12
GUI 프로그래밍	12
C/C++ 와의 결합	13
웹 프로그래밍	13
수치 연산 프로그래밍	13

데이터베이스 프로그래밍	13
데이터 분석	13
사물 인터넷	14
인공지능과 머신러닝	14
파이썬으로 할 수 없는 일	14
시스템과 밀접한 프로그래밍 영역	14
모바일 프로그래밍	14
01-4 파이썬 설치하기	15
윈도우에서 파이썬 설치하기	15
맥 (Mac) 에서 파이썬 설치하기	17
01-5 파이썬 둘러보기	18
파이썬 기초 실습 준비하기	18
파이썬 기초 문법 따라 해보기	19
사칙연산	19
변수에 숫자 대입하고 계산하기	20
변수에 문자 대입하고 출력하기	20
조건문 if	20
반복문 for	21
반복문 while	21
함수	22
01-6 파이썬과 에디터	23
IDLE 로 파이썬 프로그램 작성하기	23
명령 프롬프트 창에서 파이썬 프로그램 실행하기	30
추천 에디터	32
비주얼 스튜디오 코드	32
파이참	33
02 장 파이썬 프로그래밍의 기초, 자료형	35
02-1 숫자형	36
숫자형이란?	36
숫자형은 어떻게 만들고 사용할까?	36
정수형	36
실수형	37

8 진수와 16 진수	37
숫자형을 활용하기 위한 연산자	38
사칙연산	38
x 의 y 제곱을 나타내는 ** 연산자	38
나눗셈 후 나머지를 반환하는 % 연산자	38
나눗셈 후 몫을 반환하는 // 연산자	39
02-2 문자열 자료형	40
문자열이란?	40
문자열은 어떻게 만들고 사용할까?	40
문자열 안에 작은따옴표나 큰따옴표를 포함시키고 싶을 때	41
여러 줄인 문자열을 변수에 대입하고 싶을 때	42
이스케이프 코드란?	43
문자열 연산하기	44
문자열 더해서 연결하기 (Concatenation)	44
문자열 곱하기	44
문자열 곱하기 응용	44
문자열 길이 구하기	45
문자열 인덱싱과 슬라이싱	45
문자열 인덱싱이란?	45
문자열 인덱싱 활용하기	46
문자열 슬라이싱이란?	47
문자열을 슬라이싱하는 방법	48
슬라이싱으로 문자열 나누기	48
문자열 포매팅	50
문자열 포매팅 따라 하기	51
문자열 포맷 코드	52
포맷 코드와 숫자 함께 사용하기	53
format 함수를 사용한 포매팅	54
f 문자열 포매팅	56
문자열 관련 함수들	57
문자 개수 세기 (count)	58
위치 알려주기 1(find)	58
위치 알려주기 2(index)	58
문자열 삽입 (join)	58

소문자를 대문자로 바꾸기 (<code>upper</code>)	59
대문자를 소문자로 바꾸기 (<code>lower</code>)	59
왼쪽 공백 지우기 (<code>lstrip</code>)	59
오른쪽 공백 지우기 (<code>rstrip</code>)	59
양쪽 공백 지우기 (<code>strip</code>)	60
문자열 바꾸기 (<code>replace</code>)	60
문자열 나누기 (<code>split</code>)	60
02-3 리스트 자료형	61
리스트는 어떻게 만들고 사용할까?	61
리스트의 인덱싱과 슬라이싱	62
리스트의 인덱싱	62
리스트의 슬라이싱	63
리스트 연산하기	64
리스트 더하기 (+)	64
리스트 반복하기 (*)	65
리스트 길이구하기	65
리스트의 수정과 삭제	66
리스트에서 값 수정하기	66
<code>del</code> 함수 사용해 리스트 요소 삭제하기	66
리스트 관련 함수들	67
리스트에 요소 추가 (<code>append</code>)	67
리스트 정렬 (<code>sort</code>)	67
리스트 뒤집기 (<code>reverse</code>)	68
인덱스 반환 (<code>index</code>)	68
리스트에 요소 삽입 (<code>insert</code>)	69
리스트 요소 제거 (<code>remove</code>)	69
리스트 요소 끄집어내기 (<code>pop</code>)	69
리스트에 포함된 요소 <code>x</code> 의 개수 세기 (<code>count</code>)	70
리스트 확장 (<code>extend</code>)	70
02-4 튜플 자료형	71
튜플은 어떻게 만들까?	71
튜플의 요소값을 지우거나 변경하려고 하면 어떻게 될까?	72
튜플 다루기	72
인덱싱하기	72

슬라이싱하기	73
튜플 더하기	73
튜플 곱하기	73
튜플 길이 구하기	73
02-5 딕셔너리 자료형	74
딕셔너리란?	74
딕셔너리는 어떻게 만들까?	74
딕셔너리 쌍 추가, 삭제하기	75
딕셔너리 쌍 추가하기	75
딕셔너리 요소 삭제하기	76
딕셔너리를 사용하는 방법	76
딕셔너리에서 Key 사용해 Value 얻기	76
딕셔너리 만들 때 주의할 사항	77
딕셔너리 관련 함수들	78
Key 리스트 만들기 (keys)	78
Value 리스트 만들기 (values)	79
Key, Value 쌍 얻기 (items)	79
Key: Value 쌍 모두 지우기 (clear)	79
Key 로 Value 얻기 (get)	80
해당 Key 가 딕셔너리 안에 있는지 조사하기 (in)	80
02-6 집합 자료형	81
집합 자료형은 어떻게 만들까?	81
집합 자료형의 특징	81
교집합, 합집합, 차집합 구하기	82
집합 자료형 관련 함수들	83
값 1 개 추가하기 (add)	83
값 여러 개 추가하기 (update)	84
특정 값 제거하기 (remove)	84
02-7 불 자료형	85
불 자료형이란?	85
자료형의 참과 거짓	86
불 연산	88

02-8 자료형의 값을 저장하는 공간, 변수	89
변수는 어떻게 만들까?	89
변수란?	89
리스트를 복사하고자 할 때	90
1. [:] 이용	91
2. copy 모듈 이용	91
변수를 만드는 여러 가지 방법	92
02 장 연습문제	93
Q1. 평균 점수 구하기	93
Q2. 홀수 짝수 판별	93
Q3. 주민등록번호 나누기	93
Q4. 주민등록번호 인덱싱	93
Q5. 문자열 바꾸기	94
Q6. 리스트 역순 정렬하기	94
Q7. 리스트를 문자열로 만들기	94
Q8. 튜플 더하기	94
Q9. 딕셔너리의 키	94
Q10. 딕셔너리 POP	95
Q11. 리스트에서 중복 제거하기	95
Q12. 파이썬 변수	95
03 장 프로그램의 구조를 쌓는다! 제어문	96
03-1 if 문	97
if 문은 왜 필요할까?	97
if 문의 기본 구조	98
들여쓰기	99
조건문이란 무엇인가?	102
비교연산자	102
and, or, not	104
in, not in	104
다양한 조건을 판단하는 elif	106
조건부 표현식	108

03-2 while 문	109
while 문의 기본 구조	109
while 문 만들기	110
while 문 강제로 빠져나가기	111
while 문의 맨 처음으로 돌아가기	113
무한 루프	114
03-3 for 문	116
for 문의 기본 구조	116
예제를 통해 for 문 이해하기	116
1. 전형적인 for 문	116
2. 다양한 for 문의 사용	117
3. for 문의 응용	117
for 문과 continue	118
for 문과 함께 자주 사용하는 range 함수	118
range 함수의 예시 살펴보기	119
for 와 range 를 이용한 구구단	120
리스트 컴프리헨션 사용하기	120
03 장 연습문제	122
Q1. 조건문의 참과 거짓	122
Q2. 3 의 배수의 합	122
Q3. 별 표시하기	122
Q4. 1 부터 100 까지 출력	122
Q5. 평균점수 구하기	123
Q6. 리스트 컴프리헨션	123
04 장 파이썬의 입력과 출력	124
04-1 함수	125
함수란 무엇인가?	125
함수를 사용하는 이유는 무엇일까?	126
파이썬 함수의 구조	126
매개변수와 인수	127
입력값과 리턴값에 따른 함수의 형태	127
일반적인 함수	128

입력값이 없는 함수	128
리턴값이 없는 함수	129
입력값도 리턴값도 없는 함수	130
매개변수 지정하여 호출하기	130
입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?	131
키워드 매개변수 kwargs	132
함수의 리턴값은 언제나 하나이다	133
매개변수에 초깃값 미리 설정하기	135
함수 안에서 선언한 변수의 효력 범위	136
함수 안에서 함수 밖의 변수를 변경하는 방법	137
lambda	138
04-2 사용자 입력과 출력	140
사용자 입력	140
<input/> 의 사용	140
프롬프트를 띄워서 사용자 입력 받기	140
print 자세히 알기	141
큰따옴표 (") 로 둘러싸인 문자열은 + 연산과 동일하다	141
문자열 띄어쓰기는 콤마로 한다	142
한 줄에 줄긋값 출력하기	142
04-3 파일 읽고 쓰기	143
파일 생성하기	143
파일을 쓰기 모드로 열어 내용 쓰기	144
파일을 읽는 여러 가지 방법	145
readline 함수 이용하기	145
readlines 함수 사용하기	146
read 함수 사용하기	147
파일 객체를 for 문과 함께 사용하기	147
파일에 새로운 내용 추가하기	147
with 문과 함께 사용하기	148
04-4 프로그램의 입력과 출력	150
04 장 연습문제	152
Q1. 홀수 짝수 판별하기	152

Q2. 모든 입력의 평균 값 구하기	152
Q3. 프로그램 오류 수정하기 1	152
Q4. 출력 결과가 다른것은?	153
Q5. 프로그램 오류 수정하기 2	153
Q6. 사용자 입력 저장하기	153
Q7. 파일의 문자열 바꾸기	153
Q8. 입력값을 모두 더하여 출력하기	154
05 장 파이썬 날개달기	155
05-1 클래스	156
클래스는 왜 필요한가?	156
클래스와 객체	159
사칙연산 클래스 만들기	161
클래스를 어떻게 만들지 먼저 구상하기	161
클래스 구조 만들기	162
객체에 숫자 지정할 수 있게 만들기	162
더하기 기능 만들기	167
곱하기, 빼기, 나누기 기능 만들기	168
생성자 (Constructor)	169
클래스의 상속	171
메서드 오버라이딩	173
클래스 변수	173
05-2 모듈	176
모듈 만들기	176
모듈 불러오기	176
if __name__ == "__main__": 의 의미	178
클래스나 변수 등을 포함한 모듈	179
다른 파일에서 모듈 불러오기	180
모듈을 불러오는 또 다른 방법	181
sys.path.append 사용하기	181
PYTHONPATH 환경 변수 사용하기	182
05-3 패키지	183
패키지란 무엇인가?	183

패키지 만들기	184
패키지 안의 함수 실행하기	185
__init__.py 의 용도	186
relative 패키지	186
05-4 예외 처리	188
오류는 어떤 때 발생하는가?	188
오류 예외 처리 기법	189
try, except 문	189
try .. finally	190
여러개의 오류처리하기	190
try ... else	191
오류 회피하기	192
오류 일부러 발생시키기	192
예외 만들기	193
05-5 내장 함수	196
abs	196
all	197
any	197
chr	198
dir	198
divmod	198
enumerate	199
eval	199
filter	199
hex	200
id	201
input	201
int	202
isinstance	202
len	202
list	203
map	203
max	204
min	204

oct	205
open	205
ord	205
pow	206
range	206
round	206
sorted	207
str	207
sum	207
tuple	208
type	208
zip	208
05-6 표준 라이브러리	209
datetime.date	209
time	210
math.gcd	213
math.lcm	214
random	215
itertools.zip_longest	216
itertools.permutation	217
itertools.combination	218
functools.reduce	220
operator.itemgetter	221
shutil	223
glob	224
pickle	224
os	225
zipfile	226
threading	227
tempfile	229
traceback	229
json	231
urllib	232
webbrowser	233

05-7 외부 라이브러리	235
pip	235
pip install	235
pip uninstall	235
특정 버전으로 설치하기	236
최신 버전으로 업그레이드하기	236
설치된 패키지 확인하기	236
Faker	236
Faker 사용해 보기	237
Faker 활용	238
sympy	238
sympy 사용해 보기	238
sympy 활용	241
05 장 연습문제	242
Q1. UpgradeCalculator	242
Q2. MaxLimitCalculator	242
Q3. 참과 거짓 예측하기	243
Q4. 음수 제거하기	243
Q5. 16 진수를 10 진수로 변경하기	243
Q6. 리스트 항목마다 3 곱하여 리턴하기	243
Q7. 최댓값과 최솟값의 합	244
Q8. 소수점 반올림하기	244
Q9. 디렉터리 이동하고 파일 목록 출력하기	244
Q10. 파일 확장자가.py 인 파일만 찾기	244
Q11. 날짜 표시하기	244
Q12. 로또 번호 생성하기	245
Q13. 누나는 영철이보다 며칠 더 먼저 태어났을까?	245
Q14. 기록순으로 정렬하기	245
Q15. 청소 당번 2 명 뽑기	246
Q16. 문자열 나열하기	246
Q17. 5 명에게 할일 부여하기	246
Q18. 벽에 타일 붙이기	246
06 장 파이썬 프로그래밍, 어떻게 시작해야 할까?	247

06-1 내가 프로그램을 만들 수 있을까?	248
06-2 3 과 5 의 배수 합하기	251
06-3 게시판 페이지징하기	254
06-4 간단한 메모장 만들기	256
06-5 탭을 4 개의 공백으로 바꾸기	259
06-6 하위 디렉터리 검색하기	262
06-7 파이보	265
06-8 코딩도장	266
07 장 파이썬 날아오르기	267
07-1 파이썬과 유니코드	268
인코딩	268
디코딩	270
입출력과 인코딩	270
소스코드의 인코딩	271
07-2 클로저와 데코레이터	272
클로저	272
데코레이터	274
07-3 이터레이터와 제너레이터	278
이터레이터란?	278
이터레이터 만들기	279
제너레이터란?	281
제너레이터 표현식	282
제너레이터와 이터레이터	283
제너레이터 활용하기	283
07-4 파이썬 타입 어노테이션	285
파이썬은 동적 프로그래밍 언어	285
자바는 정적 프로그래밍 언어	285

동적 언어의 장단점	286
파이썬 타입 어노테이션	286
mypy	286
08 장 정규표현식	288
08-1 정규 표현식 살펴보기	289
정규 표현식은 왜 필요한가?	289
08-2 정규 표현식 시작하기	291
정규 표현식의 기초, 메타 문자	291
문자 클래스 []	291
Dot(.)	292
반복 (*)	293
반복 (+)	293
반복 ({m,n}, ?)	294
파이썬에서 정규 표현식을 지원하는 re 모듈	296
정규식을 이용한 문자열 검색	296
match	297
search	298
findall	298
finditer	298
match 객체의 메서드	299
컴파일 옵션	300
DOTALL, S	300
IGNORECASE, I	301
MULTILINE, M	301
VERBOSE, X	302
백슬래시 문제	303
08-3 강력한 정규 표현식의 세계로	305
메타문자	305
.	305
^	305
\$	306
\A	306

\Z	306
\b	306
\B	307
그루핑	307
그루핑된 문자열 재참조하기	309
그루핑된 문자열에 이름 붙이기	309
전방 탐색	310
긍정형 전방 탐색	311
부정형 전방 탐색	311
문자열 바꾸기	312
sub 메서드 사용 시 참조 구문 사용하기	313
sub 메서드의 매개변수로 함수 넣기	313
Greedy vs Non-Greedy	314
09 장 종합문제	315
Q1. 문자열 바꾸기	315
Q2. 딕셔너리 값 추출하기	315
Q3. 리스트의 더하기와 extend 함수	316
Q4. 리스트 총합 구하기	316
Q5. 피보나치 함수	316
Q6. 숫자의 총합 구하기	317
Q7. 한 줄 구구단	317
Q8. 파일을 읽어 역순으로 저장하기	317
Q9. 평균값 구하기	317
Q10. 사칙연산 계산기	318
Q11. 모듈을 사용하는 방법	318
Q12. 오류와 예외 처리	318
Q13. DashInsert	319
Q14. 문자열 압축하기	319
Q15. Duplicate Numbers	319
Q16. 모스 부호 해독	320
Q17. 정규식 - 기초 메타 문자	320
Q18. 정규식 - 문자열 검색	321
Q19. 정규식 - 그루핑	321
Q20. 정규식 - 전방 탐색	321

10 장 풀이	322
10-1 풀이: 02 장 연습문제	323
Q1. 평균 점수 구하기	323
Q2. 홀수 짝수 판별	323
Q3. 주민등록번호 나누기	323
Q4. 주민등록번호 인덱싱	324
Q5. 문자열 바꾸기	324
Q6. 리스트 역순 정렬하기	324
Q7. 리스트를 문자열로 만들기	325
Q8. 튜플 더하기	325
Q9. 딕셔너리의 키	325
Q10. 딕셔너리 POP	326
Q11. 리스트에서 중복 제거하기	326
Q12. 파이썬 변수	326
10-2 풀이: 03 장 연습문제	327
Q1. 조건문의 참과 거짓	327
Q2. 3 의 배수의 합	327
Q3. 별 표시하기	327
Q4. 1 부터 100 까지 출력	328
Q5. 평균점수 구하기	328
Q6. 리스트 컴프리헨션	329
10-3 풀이: 04 장 연습문제	330
Q1. 홀수 짝수 판별하기	330
Q2. 모든 입력의 평균 값 구하기	330
Q3. 프로그램 오류 수정하기 1	331
Q4. 출력 결과가 다른것은?	331
Q5. 프로그램 오류 수정하기 2	331
Q6. 사용자 입력 저장하기	332
Q7. 파일의 문자열 바꾸기	332
Q8. 입력값을 모두 더하여 출력하기	332
10-4 풀이: 05 장 연습문제	334
Q1. UpgradeCalculator	334

Q2. MaxLimitCalculator	334
Q3. 참과 거짓 예측하기	334
Q4. 음수 제거하기	335
Q5. 16 진수를 10 진수로 변경하기	335
Q6. 리스트 항목마다 3 곱하여 리턴하기	335
Q7. 최댓값과 최솟값의 합	336
Q8. 소수점 반올림하기	336
Q9. 디렉터리 이동하고 파일 목록 출력하기	336
Q10. 파일 확장자가.py 인 파일만 찾기	336
Q11. 날짜 표시하기	337
Q12. 로또 번호 생성하기	337
Q13. 누나는 영철이보다 며칠 더 먼저 태어났을까?	337
Q14. 기록순으로 정렬하기	338
Q15. 청소 당번 2 명 뽑기	338
Q16. 문자열 나열하기	339
Q17. 5 명에게 할일 부여하기	339
Q18. 벽에 타일 붙이기	340
10-5 풀이: 09 장 종합문제	341
Q1. 문자열 바꾸기	341
Q2. 딕셔너리 값 추출하기	341
Q3. 리스트의 더하기와 extend 함수	341
Q4. 리스트 총합 구하기	342
Q5. 피보나치 함수	343
Q6. 숫자의 총합 구하기	343
Q7. 한 줄 구구단	343
Q8. 파일을 읽어 역순으로 저장하기	344
Q9. 평균 값 구하기	344
Q10. 사칙연산 계산기	345
Q11. 모듈을 사용하는 방법	345
Q12. 오류와 예외 처리	346
Q13. DashInsert	346
Q14. 문자열 압축하기	346
Q15. Duplicate Numbers	347
Q16. 모스 부호 해독	348

Q17. 정규식 - 기초 메타 문자	348
Q18. 정규식 - 문자열 검색	348
Q19. 정규식 - 그룹핑	349
Q20. 정규식 - 전방 탐색	349
11 장 부록 - 챗 GPT 활용하기	351
오류의 원인 물어보기	351
작성한 코드에 문제가 있는지 물어보기	352
작성한 코드의 개선을 요청하기	353
주석을 달아달라고 요청하기	354
코드의 해석 요청하기	356
기능 구현 요청하기	356
12 장 마치며	358
함께 만드는 점프 투 파이썬	358
도전과제	358
파이썬 질문과 답변	359

문서정보

Copyright© 2022 박응용. All rights reserved.

이 책의 무단전재와 복제를 금합니다

이 책은 **송병연 (stitch1211@daum.net)** 님이 구매하신 문서입니다.

(구매: <http://wikidocs.net/book/1>)

ISBN 979-11-981113-5-7



프로그램 코드의 추가와 삭제

프로그램 코드에서 초록색 배경으로 표시한 부분은 코드의 추가 또는 강조를 의미합니다.

```
1 def hello():  
2     message = "Hello"  
3     message = message.upper() # 추가  
4     print(message)
```

분홍색 배경은 삭제를 의미합니다.

```
1 def hello():  
2     message = "Hello"  
3     message = message.upper() # 삭제  
4     print(message)
```

00 장 들어가기 전에

점프 투 파이썬이 세상에 나온지 벌써 20 년이 지났다.

책을 처음 집필하던 20 년 전의 초보 시절의 마음과 지나온 20 년 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

댓글과 피드백

“점프 투 파이썬”은 함께 만들어가는 온라인 책이다. 여러분의 피드백으로 “점프 투 파이썬”은 성장해가고 있다. 책을 쓰며 있어서 가장 두려운 것은 잘못된 정보의 전달이다. 하지만 이런 두려움은 여러분의 댓글 참여로 인해 해소될 수 있으리라 생각한다. 댓글과 피드백을 통한 여러분의 참여와 도움을 기다린다. (댓글이나 피드백 링크는 페이지 하단에 있다.)

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 20살이 넘었다. 파이썬이 버전 업을 하며 진화해 가는 동안 “점프 투 파이썬” 역시 “위키독스(wikidocs.net)”에서 파이썬의 변화와 독자들의 요구에 발 맞추어 계속된 진화를 거듭해 왔다.

사실 “점프 투 파이썬”은 필자 한 사람에게 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 홍연의, 한승우씨에게 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”의 독자 여러분들 모두에게 무한한 감사의 말씀을 전하고 싶다.

박응용 드림.

00-2 저자소개

박응용 (pahkey@gmail.com)

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

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

저서

- 점프 투 파이썬
- 점프 투 파이썬 - 라이브러리 예제 편
- 점프 투 장고
- 점프 투 플라스크
- 점프 투 자바
- 점프 투 스프링부트
- 점프 투 FastAPI

유튜브

- 응용코딩 - <https://www.youtube.com/channel/UC6YQqXMz2SyagOljg86ceOg>

01 장 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?

[추천 동영상 강의] : <https://www.youtube.com/watch?v=6MjXHIYZWRE>

파이썬 (Python) 은 1990 년 암스테르담의 귀도 반 로섬 (Guido Van Rossum) 이 개발한 인터프리터 언어이다. 귀도는 파이썬이라는 이름을 자신이 좋아하는 코미디 쇼인 “몬티 파이썬의 날아다니는 서커스 (Monty Python’s Flying Circus)” 에서 따왔다고 한다.

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

파이썬의 사전적 의미는 고대 신화에 나오는 파르나소스 산의 동굴에 살던 큰 뱀을 뜻하며, 아폴로 신이 델파이에서 파이썬을 퇴치했다는 이야기가 전해지고 있다. 대부분의 파이썬 책 표지와 아이콘이 뱀 모양으로 그려져 있는 이유가 여기에 있다.



파이썬은 컴퓨터 프로그래밍 교육을 위해 많이 사용하지만, 기업의 실무를 위해서도 많이 사용하는 언어이다. 그 대표적인 예가 바로 구글이다. 필자는 구글에서 만든 소프트웨어의 50% 이상이 파이썬으로 작성되었다는 이야기를 들었다. 이외에도 많이 알려진 예를 몇 가지 들자면 온라인 사진 공유 서비스 인스타그램 (Instagram), 파일 동기화 서비스 드롭박스 (Dropbox) 등이 있다.

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

01-2 파이썬의 특징

[추천 동영상 강의] : <https://www.youtube.com/watch?v=pFCHn0F-nuw>

필자는 파이썬을 무척 좋아한다. 모든 프로그래밍 언어에는 각기 장점이 있지만 파이썬에는 다른 언어에서는 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 술술 써 내려가는 여러분의 모습에 놀라게 될 것이다.

다음 소스 코드를 보면 이 말이 쉽게 이해될 것이다.

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

위 예제는 다음처럼 읽을 수 있다.

```
1 만약 4가 1, 2, 3, 4 중에 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 있다면 파이썬의 자료형, 함수, 클래스 만드는 법, 라이브러리 및 내장 함수 사용 방법 등을 익히는 데 1 주일이면 충분하리라 생각한다.

파이썬은 무료이지만 강력하다

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

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

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 속도가 필요한 부분은 C로 만들어서 파이썬 프로그램 안에 포함시키는 것이다 (정말 놀라우리만치 영악한 언어가 아닌가). 사실 파이썬 라이브러리 중에는 순수 파이썬만으로 제작된 것도 많지만 C로 만든 것도 많다. C로 만든 것은 대부분 속도가 빠르다.

파이썬 라이브러리는 파이썬 프로그램을 작성할 때 불러와 사용할 수 있는 미리 만들어 놓은 파이썬 파일 모음이다.

파이썬은 간결하다

귀도는 파이썬을 의도적으로 간결하게 만들었다. 만약 펄 (Perl) 과 같은 프로그래밍 언어가 100 가지 방법으로 하나의 일을 처리할 수 있다면 파이썬은 가장 좋은 방법 1 가지만 사용하는 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. 한번 구경해 보자.

```
1 # simple.py
2 languages = ['python', 'perl', 'c', 'java']
3
```

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

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

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

파이썬은 프로그래밍을 즐기게 해준다

이 부분이 가장 강조하고 싶은 부분이다. 파이썬만큼 필자에게 프로그래밍을 즐기게 해 준 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 마무리하겠다.

“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.

01-3 파이썬으로 무엇을 할 수 있을까?

[추천 동영상 강의] : <https://www.youtube.com/watch?v=YuOGpTO8x94>

프로그래밍 언어를 좋은 언어와 나쁜 언어로 구별할 수 있을까? 사실 현실에서 이런 구별은 무의미하다. 어떤 언어든지 강점과 약점이 존재하기 때문이다. 그러므로 어떤 프로그래밍 언어가 어떤 일에 효율적인지를 안다는 것은 프로그래머의 생산성을 크게 높일 수 있는 힘이 된다. 그렇다면 파이썬으로 하기에 적당한 일과 적당하지 않은 일은 무엇일까? 이를 아는 것은 매우 가치 있는 일이므로 파이썬을 배우기 전에 먼저 간단히 알아보자.

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 명령어를 사용할 수 있는 각종 도구를 갖추고 있기 때문에 이를 바탕으로 갖가지 시스템 유틸리티를 만드는 데 유리하다. 실제로 여러분은 시스템에서 사용 중인 서로 다른 유틸리티성 프로그램을 하나로 묶어서 큰 힘을 발휘하게 하는 프로그램들을 무수히 만들어낼 수 있다.

유틸리티란 컴퓨터 사용에 도움을 주는 여러 소프트웨어를 말한다.

GUI 프로그래밍

GUI(Graphic User Interface) 프로그래밍이란 쉽게 말해 화면에 윈도우 창을 만들고 그 창에 프로그램을 동작시킬 수 있는 메뉴나 버튼 등을 추가하는 것이다. 파이썬은 GUI 프로그래밍을 위한 도구들이 잘 갖추어져 있어 GUI 프로그램을 만들기 쉽다. 대표적인 예로 파이썬 프로그램과 함께 설치되는 Tkinter(티케이인터)가 있다. Tkinter를 사용하면 단 5 줄의 소스 코드만으로 윈도우 창을 띄울 수 있다.

C/C++ 와의 결합

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

웹 프로그래밍

일반적으로 익스플로러나 크롬, 파이어폭스 같은 브라우저로 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 제작한 웹 사이트는 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 파이썬은 넘파이 (NumPy) 라는 수치 연산 모듈을 제공한다. 이 모듈은 C 로 작성했기 때문에 파이썬에서도 수치 연산을 빠르게 할 수 있다.

데이터베이스 프로그래밍

파이썬은 사이베이스 (Sybase), 인포믹스 (Infomix), 오라클 (Oracle), 마이에스큐엘 (MySQL), 포스트그레스큐엘 (PostgreSQL) 등의 데이터베이스에 접근하기 위한 도구를 제공한다.

또한 이런 굵직한 데이터베이스를 직접 사용하는 것 외에도 파이썬에는 재미있는 도구가 하나 더 있다. 바로 피클 (pickle) 이라는 모듈이다. 피클은 파이썬에서 사용하는 자료를 변형 없이 그대로 파일에 저장하고 불러오는 일을 맡아 한다. 이 책에서는 외장 함수에서 피클을 어떻게 사용하고 활용하는지에 대해서 알아본다.

데이터 분석

파이썬은 데이터 분석에 이상적인 언어로, 넘파이 (NumPy), 판다스 (Pandas), 맷플로립 (Matplotlib) 등의 라이브러리들을 활용하여 데이터 처리, 통계 분석, 시각화를 손쉽게 수행할 수 있다. 데이터 분석을 할 때 아직까지는 데이터 분석에 특화된 “R” 이라는 언어를 많이 사용하고 있지만, 판다스가 등장한 이후로 파이썬을 사용하는 경우가 점점 증가하고 있다.

사물 인터넷

사물 인터넷 분야에서도 파이썬은 활용도가 높다. 한 예로 라즈베리파이 (Raspberry Pi) 는 리눅스 기반의 아주 작은 컴퓨터이다. 라즈베리파이를 사용하면 홈시어터나 아주 작은 게임기 등 여러 가지 재미있는 것들을 만들 수 있는데, 파이썬은 이 라즈베리파이를 제어하는 도구로 사용된다. 예를 들어 라즈베리파이 에 연결된 모터를 작동시키거나 LED 에 불이 들어오게 하는 일을 파이썬으로 할 수 있다.

인공지능과 머신러닝

최근 인공지능 (Artificial Intelligence) 분야의 발전 덕분에 파이썬이 인기 언어로 떠올랐다. 인공지능은 기계가 인간처럼 생각하고 행동할 수 있는 능력을 제공하는 기술이다. 파이썬을 활용하면 자연어 처리, 음성 인식, 이미지 인식과 같은 인공지능 기술을 구현할 수 있다. 머신러닝 (Machine Learning) 은 인공지능의 한 분야로, 경험을 통해 자동으로 발전하는 컴퓨터 알고리즘을 연구하는 분야이다. 파이썬은 인공지능과 머신러닝 프로그래밍을 쉽게 할 수 있도록 사이킷런 (scikit-learn), 텐서플로 (Tensorflow), 파이토치 (PyTorch), 케라스 (Keras) 등 다양한 라이브러리를 제공한다.

파이썬으로 할 수 없는 일

시스템과 밀접한 프로그래밍 영역

파이썬으로 리눅스와 같은 운영체제 또는 엄청난 횟수의 반복과 연산이 필요한 프로그램을 만드는 것은 어렵다. 즉 대단히 빠른 속도를 요구하거나 하드웨어를 직접 건드릴어야 하는 프로그램에는 적합하지 않다.

모바일 프로그래밍

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

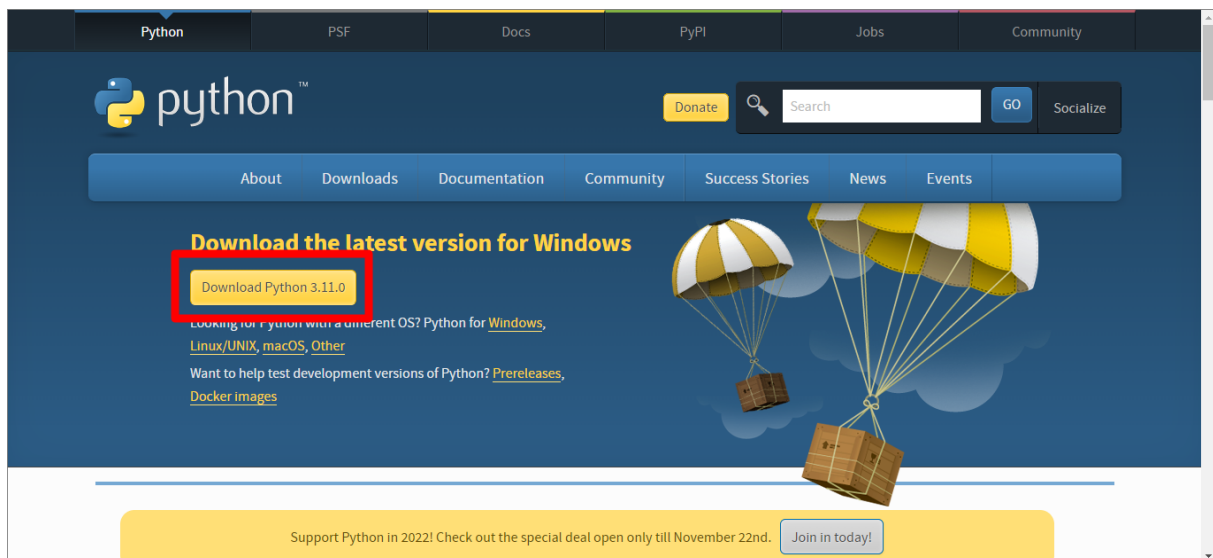
01-4 파이썬 설치하기

[추천 동영상 강의] : <https://www.youtube.com/watch?v=YmS32bo3Uas>

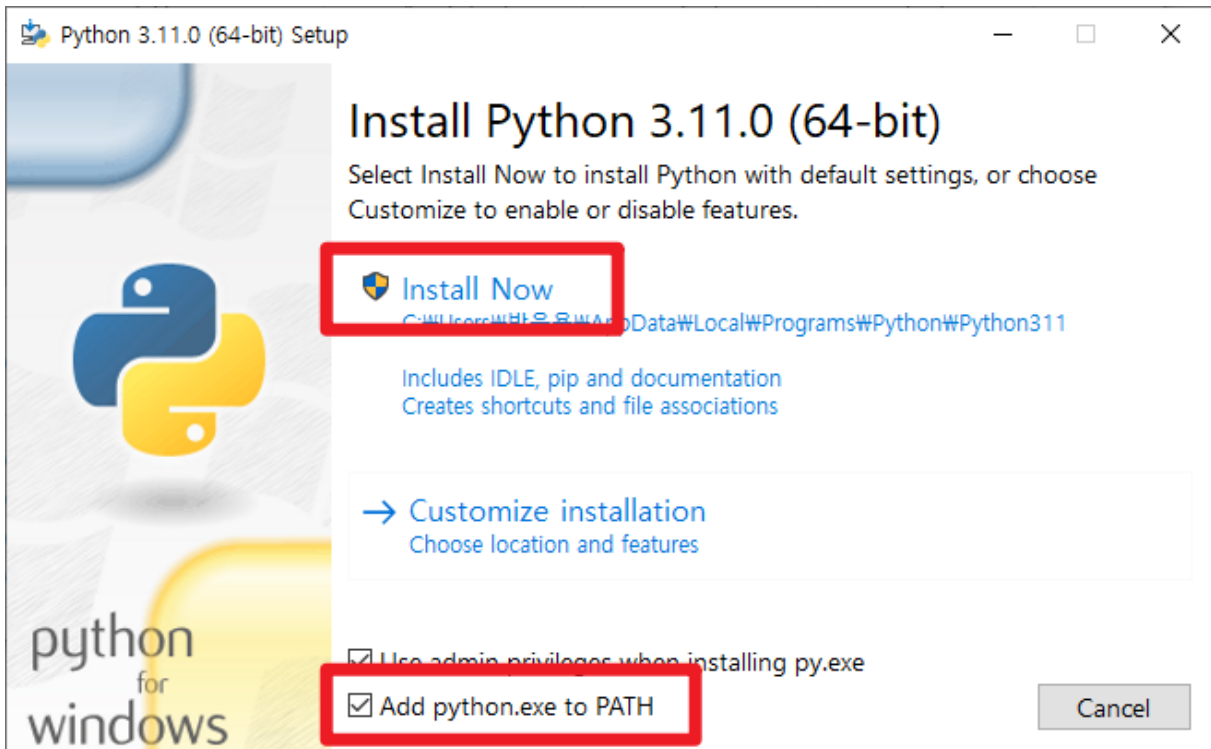
이제 실습을 위해 컴퓨터에 파이썬을 설치해 보자. 이 책에서는 윈도우와 맥에서의 설치 방법만 다룬다. 다른 시스템을 사용할 경우 파이썬 홈페이지 (<http://www.python.org>)의 설명을 참고하자.

윈도우에서 파이썬 설치하기

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 다운로드하자 (이 글을 작성하는 시점의 최신 버전은 3.11.0이다).



2. 인스톨러를 실행한 후에 “Install Now”를 선택하면 바로 설치가 진행된다. 파이썬이 어느 곳에서든지 실행될 수 있도록 “Add python.exe to PATH” 옵션을 반드시 선택해야 한다.



“Add python.exe to PATH” 옵션을 누락하면 이후 실습할 때 오류가 발생할 수 있다.

3. 설치가 완료되면 [close] 를 클릭하여 종료한다.

파이썬이 정상적으로 설치되었다면 다음 그림과 같이 프로그램 메뉴에서 확인할 수 있다.

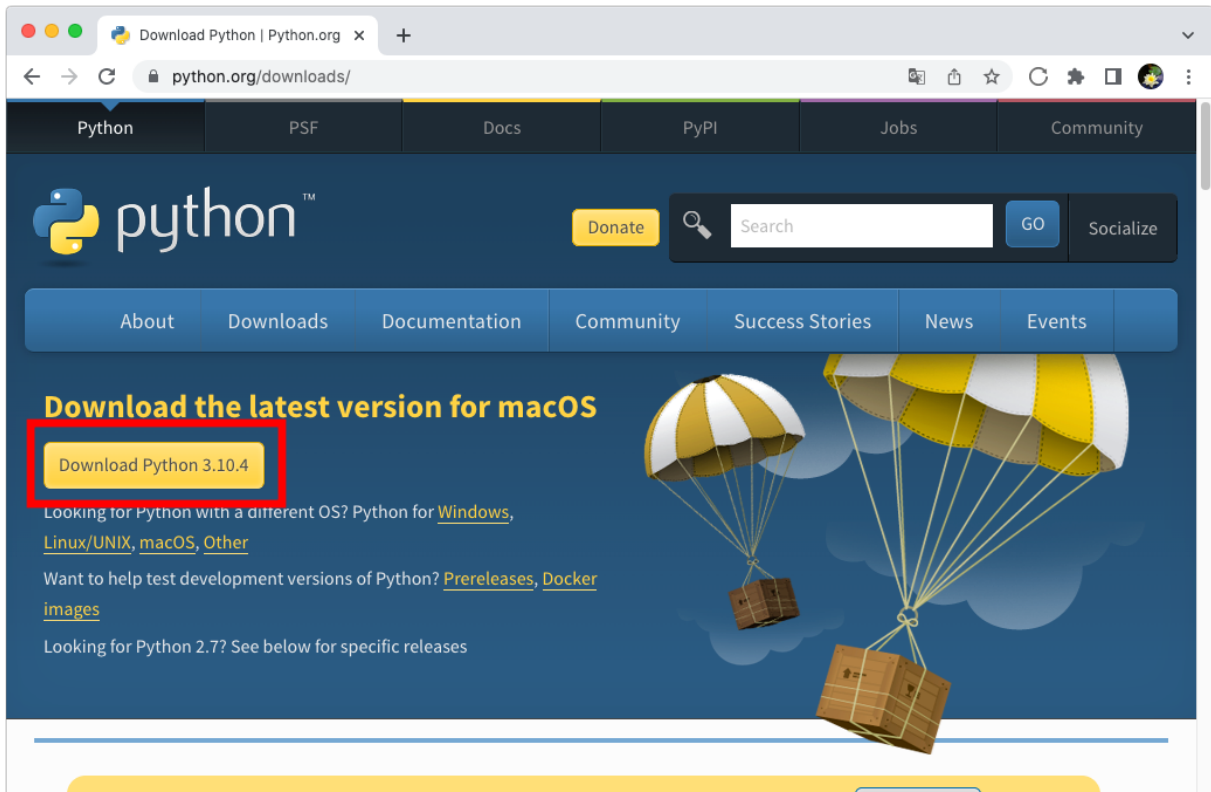
[시작 → 모든 프로그램 → Python 3.11]



맥 (Mac) 에서 파이썬 설치하기

파이썬 공식 홈페이지에서 [Downloads] 메뉴를 누르고 맥용 파이썬 설치 파일을 내려받자. 다음 화면에서 <Download Python 3.10.x>를 누르면 된다.

파이썬 공식 홈페이지 주소: www.python.org



그러면 `python-3.10.4-macos1.pkg`와 같은 파일이 다운로드된다. 이 파일을 실행하여 설치하면 된다.

설치가 완료되면 파이썬이 제대로 설치됐는지 확인하기 위해 터미널에서 다음 명령을 입력해 자신의 맥에 설치된 파이썬 버전을 확인해 보자. 오류 없이 파이썬 버전이 제대로 출력되면 설치에 성공한 것이다.

[맥 터미널]

```
1 pahkey@mymac ~ % python3 -V
2 Python 3.10.4
```

`python3` 대신 `python` 을 입력하면 파이썬 2.7 이 실행된다. 구 버전의 맥은 기본적으로 파이썬 2.7 버전이 설치되어 있다. 따라서 맥은 항상 `python` 대신 `python3` 명령을 사용하자.

01-5 파이썬 둘러보기

[추천 동영상 강의] : <https://www.youtube.com/watch?v=aodQTNWkoko>

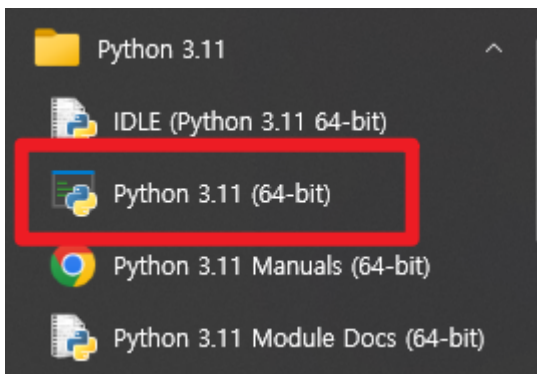
도대체 파이썬이라는 언어는 어떻게 생겼는지, 간단한 코드를 작성하며 알아보자. 파이썬을 자세히 탐구하기 전에 전체 모습을 죽 훑어보는 것은 매우 유익한 일이 될 것이다.

“백문이 불여일견, 백견이 불여일타” 라고 했다. 직접 따라 해 보자.

파이썬 기초 실습 준비하기

파이썬 프로그래밍 실습을 시작하기 전에 기초적인 것을 준비해 보자.

[시작] 메뉴에서 [프로그램 → Python 3.11 → Python 3.11] 을 선택한다.



맥 사용자의 경우 터미널에서 `python3`를 입력하자.

그러면 다음 화면이 나타난다.

```
1 Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on
  win32
2 Type "help", "copyright", "credits" or "license" for more information.
3 >>>
```

위와 같은 창을 파이썬 대화형 인터프리터라고 부른다. 여기서 인터프리터란 사용자가 입력한 소스 코드를 실행하는 환경을 뜻한다. 그렇다면 “대화형”이란 무슨 의미일까? 다음과 같이 간단한 수식을 입력해 보자.

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

입력에 따른 결과값이 바로 출력된다. 이렇게 입력과 출력이 번갈아 이어지는 것이 마치 대화하는 것처럼 느껴지기 때문에 파이썬 “대화형” 인터프리터라고 부른다. 대화형 인터프리터를 사용하면 간단한 예제를 풀 때 결과를 바로 확인할 수 있어 학습에 효과적이다. 앞으로 파이썬의 기본적인 내용은 이 창을 사용해 실습할 것이다.

파이썬 대화형 인터프리터를 간단히 파이썬 셸 (Python shell) 이라고도 부른다. 3 개의 꺾은 괄호 (>>>) 는 프롬프트 (prompt) 라고 한다.

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

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

또는 다음과 같이 내장 함수인 `quit()` 이나 `exit()` 를 사용하여 종료 할 수도 있다.

```
1 >>> quit()
```

파이썬 기초 문법 따라 해보기

여기에서 소개하는 내용은 나중에 다시 자세하게 다룰 것이니 지금 당장 이해가 되지 않는다고 절망하거나 고심하지는 말자.

파이썬 대화형 인터프리터 (파이썬 셸) 를 실행하여 다음을 직접 입력해 보자.

사칙연산

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

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

나눗셈 (/) 과 **곱셈 (*)** 역시 예상한 대로 결과값을 보여준다.

```
1 >>> 3 / 2.4
2 1.25
3 >>> 3 * 9
4 27
```

우리가 일반적으로 알고 있는 ÷ 기호나 × 기호가 아닌 것에 주의하자.

변수에 숫자 대입하고 계산하기

```
1 >>> a = 1
2 >>> b = 2
3 >>> a + b
4 3
```

a 에 1 을, b 에 2 를 대입한 다음 a 와 b 를 더하면 3 이라는 결과값을 보여 준다.

변수에 문자 대입하고 출력하기

```
1 >>> a = "Python"
2 >>> print(a)
3 Python
```

a 변수에 Python 이라는 값을 대입한 다음 print(a) 라고 작성하면 a 값을 출력한다.

파이썬은 대소문자를 구별한다. print 를 PRINT 로 쓰면 오류가 발생한다.

또는 다음과 같이 print 문을 생략하고 변수 이름 a 만 입력하여 a 의 값을 확인할 수도 있다.

```
1 >>> a = "Python"
2 >>> a
3 'Python'
```

조건문 if

다음은 간단한 조건문 if 를 사용한 예제이다.

```
1 >>> a = 3
2 >>> if a > 1:
3 ...     print("a는 1보다 큼니다.")
4 ...
5 a는 1보다 큼니다.
```

앞의 예제는 `a` 가 1 보다 크면 “`a` 는 1 보다 큼니다.” 라는 문장을 출력 (`print`) 하라는 뜻이다. 위 예제에서 `a` 는 3 이므로 1 보다 크다. 따라서 두 번째 “...” 이후에 `Enter` 키를 입력하면 `if` 문이 종료되고 “`a` 는 1 보다 큼니다.” 문장이 출력된다.

`print` 문 앞의 ‘...’ 은 아직 문장이 끝나지 않았음을 의미한다.

`if a > 1:` 다음 문장은 `Tab` 키 또는 `Spacebar` 키 4 개를 이용해 반드시 들여쓰기 한 후에 `print("a는 1보다 큼니다.")` 라고 작성해야 한다. 들여쓰기 규칙에 대해서는 03 장 제어문에서 자세하게 알아볼 것이다. 바로 뒤에 이어지는 반복문 `for`, `while` 예제도 마찬가지로 들여쓰기가 필요하다.

실습 영상: <https://youtu.be/5CU-xTiMIZ0>

반복문 for

다음은 `for` 를 사용해서 `[1, 2, 3]` 안의 값을 하나씩 출력하는 것을 보여 주는 예이다.

```
1 >>> for a in [1, 2, 3]:
2     ...     print(a)
3     ...
4     1
5     2
6     3
```

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

반복문 while

다음은 `while` 을 사용하는 예이다.

```
1 >>> i = 0
2 >>> while i < 3:
3     ...     i=i+1
4     ...     print(i)
5     ...
6     1
7     2
8     3
```

`while` 이라는 영어 단어는 “~ 인 동안” 이란 뜻이다. `for` 문과 마찬가지로 반복해서 문장을 수행할 수 있도록 해준다. 위 예제는 `i` 값이 3 보다 작은 동안 `i=i+1` 과 `print(i)` 를 수행하라는 말이다. `i=i+1` 이라는 문장은 `i` 의 값을 1 씩 더하게 한다. `i` 값이 3 보다 커지게 되면 `while` 문을 빠져나가게 된다.

실습 영상: <https://youtu.be/mtdbM3EOS68>

함수

파이썬의 **함수**는 다음과 같은 형태이다.

```
1 >>> def add(a, b):  
2     ...     return a+b  
3     ...  
4 >>> add(3,4)  
5 7
```

파이썬에서 **def** 는 함수를 만들 때 사용하는 예약어이다. 위 예제는 **add** 함수를 만들고 그 함수를 어떻게 사용하는지를 보여준다. **add(a, b)** 에서 **a, b** 는 입력값이고, **a+b** 는 결과값이다. 즉 **3, 4** 가 입력으로 들어오면 **3+4** 를 수행하고 그 결과값인 **7** 을 리턴한다.

- 예약어란 프로그래밍 언어에서 이미 문법적인 용도로 사용하고 있는 단어를 말한다.
- 리턴 (return) 은 함수 실행 후 그 값을 반환하는 행위를 말한다.

이렇게 해서 기초적인 파이썬 문법에 대해서 간략하게 알아보았다.

01-6 파이썬과 에디터

[추천 동영상 강의] : <https://www.youtube.com/watch?v=meJGRyyXY54>

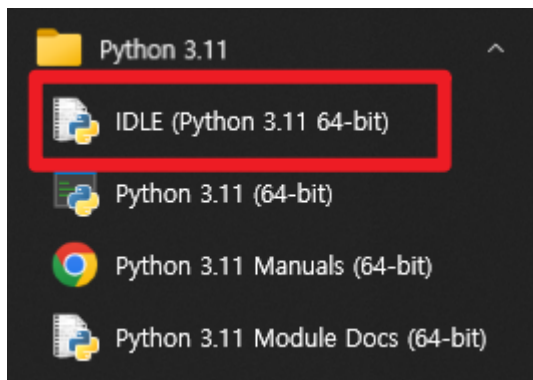
파이썬 대화형 인터프리터는 간단한 예제를 풀 때는 편리하지만 여러 줄의 복잡한 소스 코드를 가진 프로그램을 만들 때는 불편하다. 또한 인터프리터를 종료하면 작성한 프로그램이 사라지기 때문에 다시 사용하지 못한다는 단점이 있다. 그래서 여러 번 사용하기 위한 프로그램을 만들 때는 에디터를 사용해야 한다.

에디터란 소스 코드를 편집할 수 있는 프로그래밍 툴을 말한다. 에디터에는 여러 가지 종류가 있다. 이 책에서는 그중 파이썬을 설치할 때 기본으로 설치되는 파이썬 **IDLE** 에디터를 사용할 것이다. 이제 **IDLE** 을 사용해 프로그램을 작성해 보자.

IDLE 은 '아이들' 이라고 읽는다.

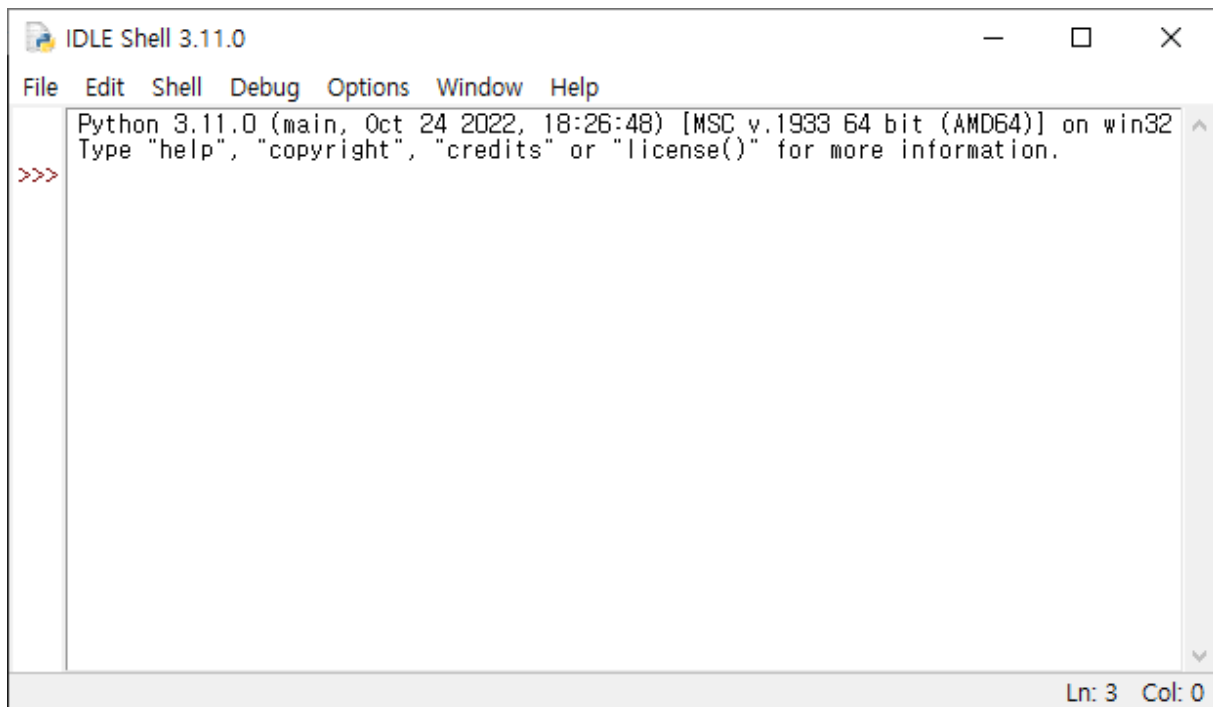
IDLE 로 파이썬 프로그램 작성하기

파이썬 IDLE(Integrated Development and Learning Environment) 은 파이썬 프로그램 작성을 도와주는 통합개발 환경이다. [시작 → 모든 프로그램 → Python 3.11 → IDLE] 을 선택해 파이썬 IDLE 을 실행해 보자.



맥 사용자는 터미널에서 `idle3` 명령을 입력하자.

그러면 다음과 같은 IDLE 셸 (Shell) 창이 먼저 나타난다.



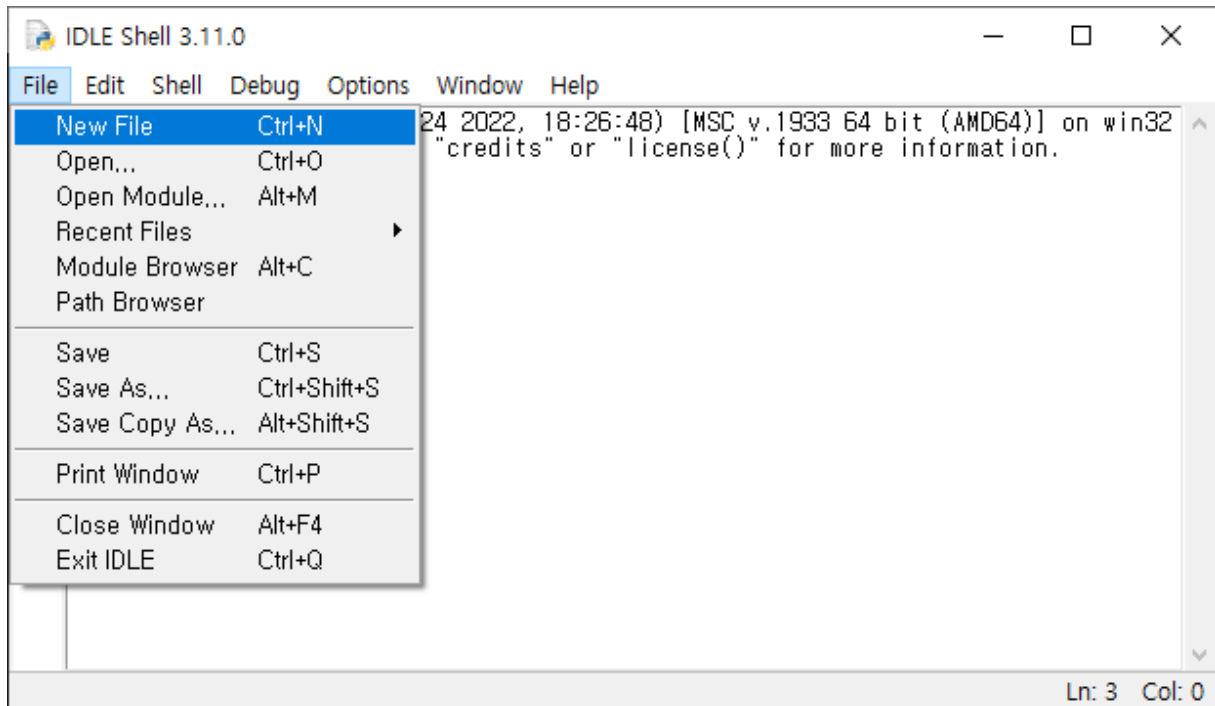
IDLE 은 크게 두 가지 창으로 구성된다.

- IDLE 셸 창 (Shell Window) - IDLE 에디터에서 실행한 프로그램의 결과가 표시되는 창으로서 파이썬 셸과 동일한 기능을 수행한다. IDLE 을 실행하면 가장 먼저 나타나는 창이다.
- IDLE 에디터 창 (Editor Window) - IDLE 에디터가 실행되는 창이다.

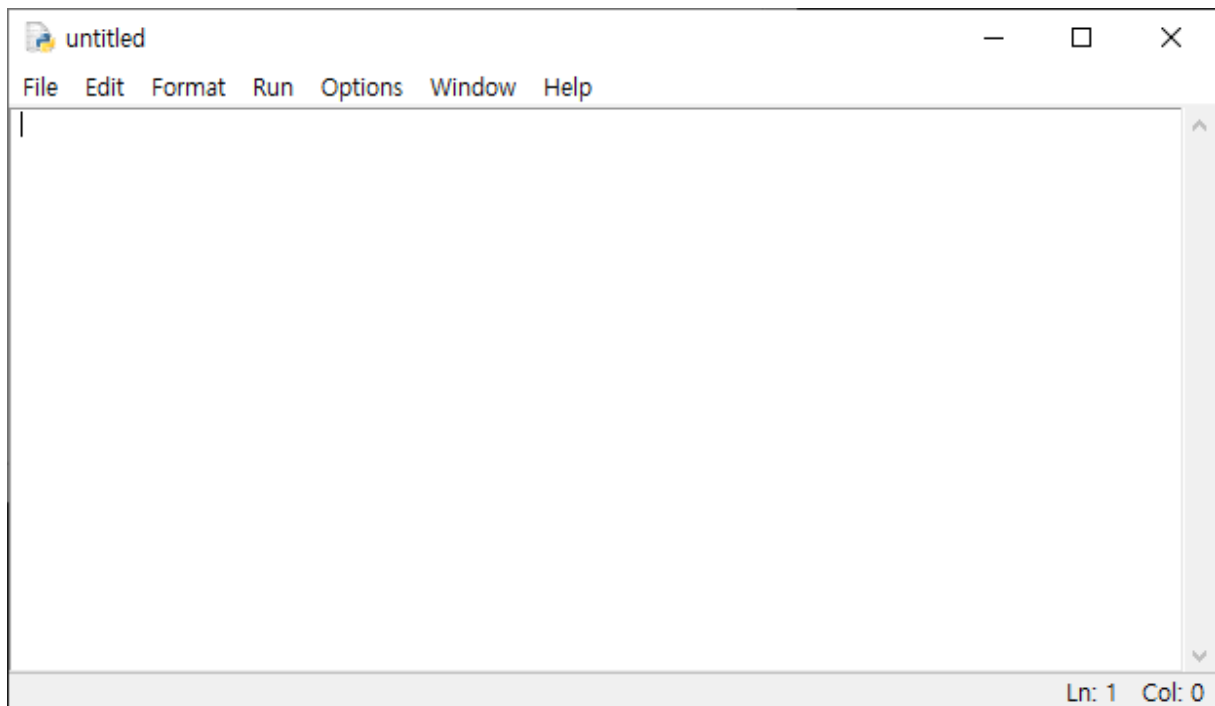
앞 장에서 본 파이썬 셸과 IDLE 셸은 기능 면에서 동일하다. 단 IDLE 셸에서 들여쓰기를 표시하는 방법이 기본 파이썬 셸과 다르므로 이 책에서는 IDLE 에디터로 작성한 프로그램의 실행 결과를 표시하는 용도로만 IDLE 셸을 사용할 것이다. 즉 앞으로 >> 프롬프트로 시작하는 예제는 IDLE 셸이 아닌 파이썬 셸로 실행해야 한다.

이제 IDLE 에디터 (Editor) 를 실행해 보자.

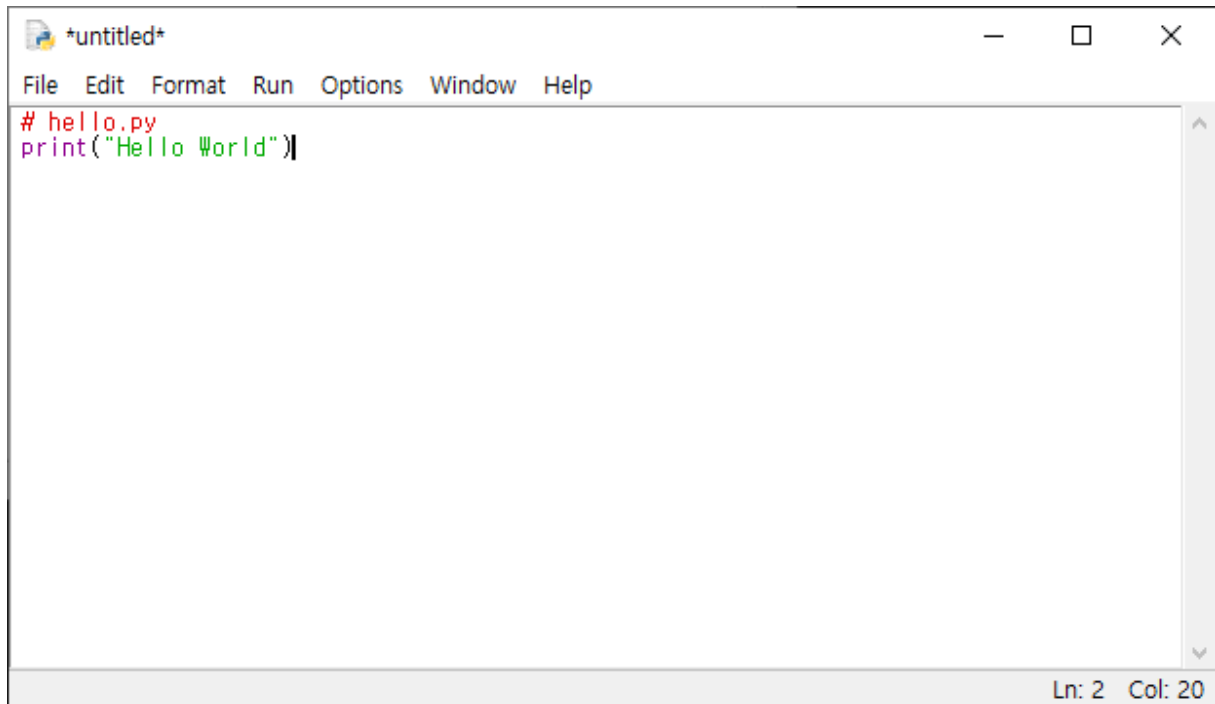
IDLE 셸 창 메뉴에서 [File → New File] 을 선택한다.



그러면 다음과 같이 빈 창이 나타나는데 이 창이 IDLE 에디터이다.



이제 IDLE 에디터에서 다음과 같은 파이썬 프로그램을 작성해 보자.



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

[여러줄짜리 주석문]

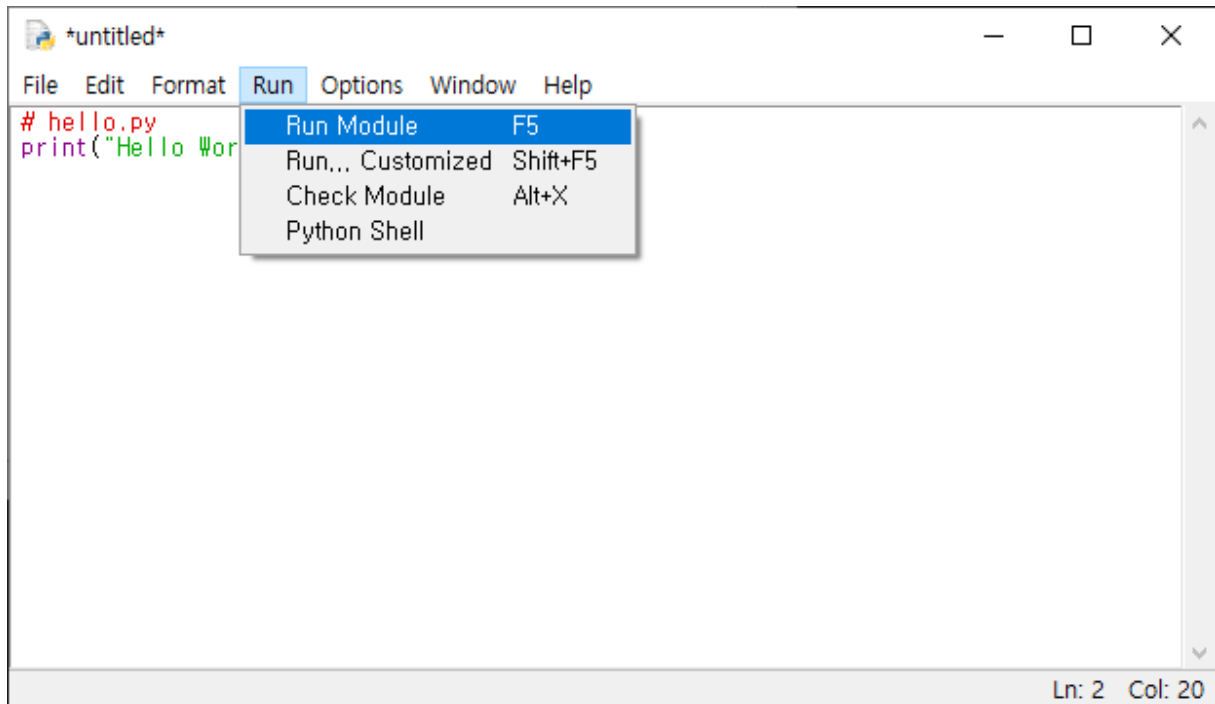
주석문이 여러 줄이라면 다음 방법을 사용하면 편리하다.

```
1 """
2 Author: EungYong Park
3 Date : 2018-05-01
4 이 프로그램은 Hello World를 출력하는 프로그램이다.
5 """
```

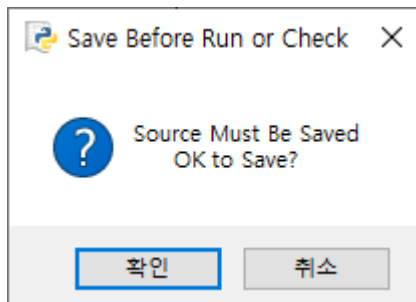
여러 줄로 이루어진 주석은 큰따옴표 세 개를 연속으로 사용한""" 기호 사이에 주석문을 작성하면 된다. 큰따옴표 대신 작은따옴표 세 개('')를 사용해도 된다.

이제 작성한 프로그램을 실행해 보자.

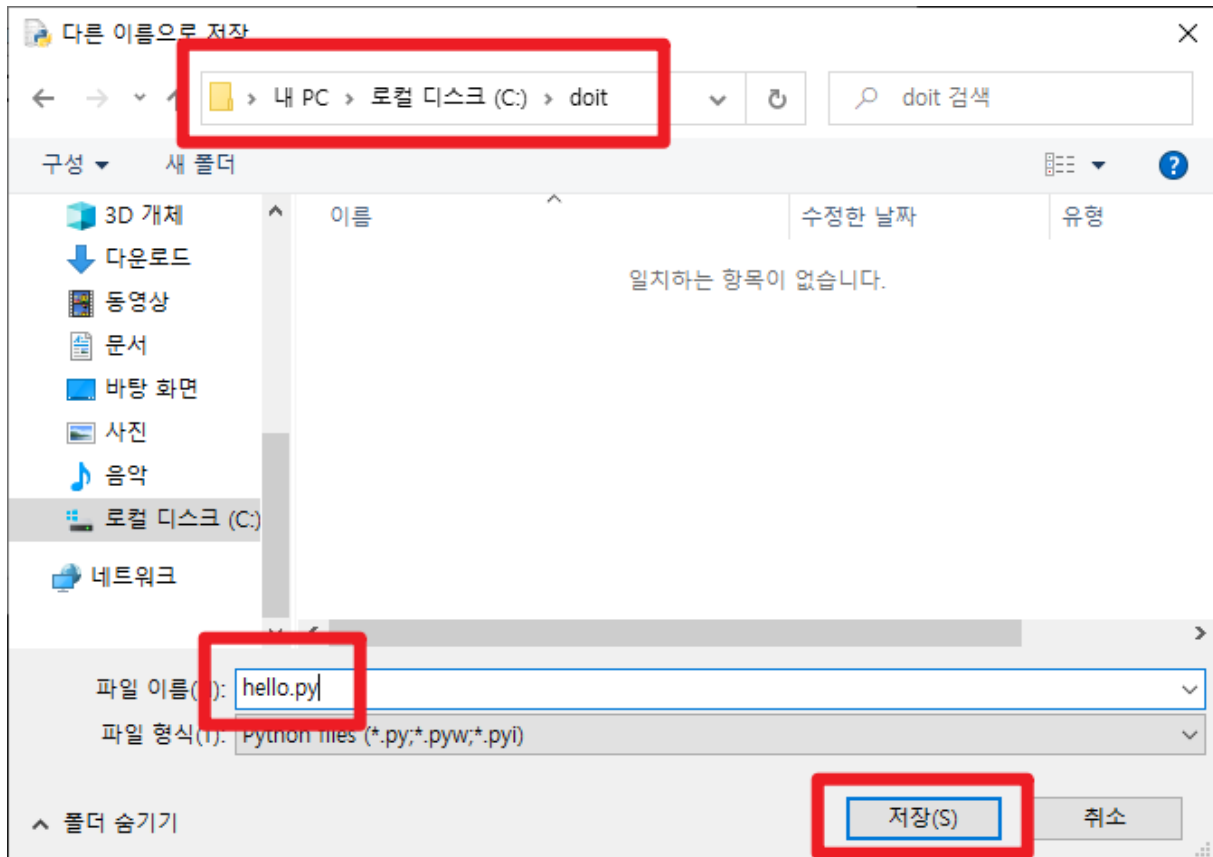
IDLE 에디터 창 메뉴에서 [Run → Run Module] 을 선택한다 (단축키: F5).



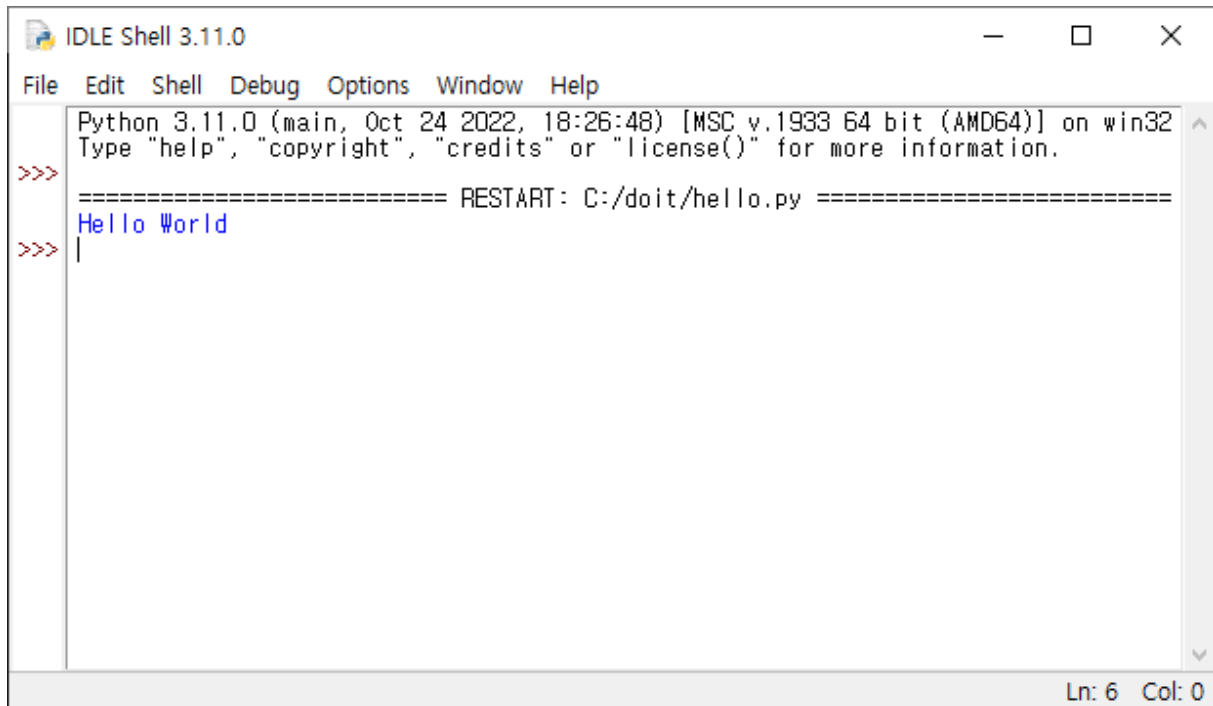
프로그램을 실행하면 파일을 먼저 저장하라는 다음과 같은 대화 상자가 나온다.



[확인] 을 선택하고 `C:\doit` 디렉터리를 생성한 후 `hello.py`라는 이름으로 저장한다. 에디터로 파이썬 프로그램을 작성한 후 저장할 때는 파일 이름의 확장자를 항상 `.py`로 해야 한다. `.py`는 파이썬 파일임을 알려주는 확장자이다.



파일을 저장하면 `hello.py` 프로그램이 실행된다. 실행 결과는 다음과 같이 IDLE 셸 창에 표시된다.



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/doit/hello.py =====
Hello World
>>> |
```

Ln: 6 Col: 0

앞으로 간단한 예제는 파이썬 셸에 입력해 결과를 바로 확인하고, 여러 줄을 한꺼번에 작성하거나 여러 번 사용해야 하는 프로그램은 IDLE 에디터에서 작성할 것이다.

파이썬 셸과 IDLE 에디터의 차이점

앞으로 이 책은 파이썬 셸과 IDLE 에디터를 번갈아 가면서 예제를 작성할 것이다. 이 때 주의할 점이 한가지 있다. 다음의 예제를 보자.

```
1 >>> a = "python"
2 >>> print(a)
3 python
4 >>> print(1+1)
5 2
```

파이썬 셸의 경우 위와 같이 변수 **a** 와 표현식 **1+1**의 값을 확인하기 위해서 **print** 함수를 사용해도 되지만 다음처럼 **print** 함수 없이도 그 값을 확인할 수 있다.

```
1 >>> a = "python"
2 >>> a
3 'python'
4 >>> 1+1
5 2
```

이것은 표현식의 실행 결과를 출력하는 파이썬 셸 본래의 기능이기 때문이다. 하지만 IDLE 에디터를 사용할 경우에는 다음처럼 **print** 문을 반드시 사용해야 한다.

[IDLE 에디터로 작성한 파이썬 파일]

```
1 a = "python"
2 print(a)
3 print(1+1)
```

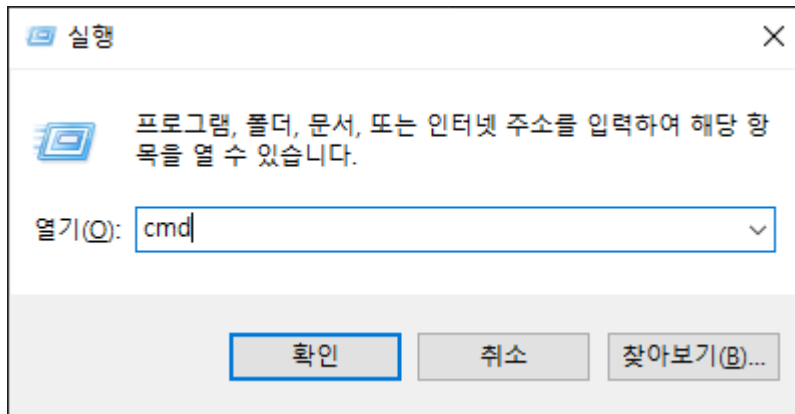
명령 프롬프트 창에서 파이썬 프로그램 실행하기

이번에는 앞에서 IDLE 로 작성한 **hello.py** 파일을 명령 프롬프트 창 (command prompt) 에서 실행해 보자. IDLE 에디터에서 **F5** 키를 눌러 실행할 수도 있지만 실제 업무에서는 일반적으로 명령 프롬프트 창에서 파이썬 프로그램을 실행한다. 이 책에서도 파이썬 프로그램 파일을 실행할 때는 주로 이 방법을 사용할 것이다.

우리는 이미 다음과 같은 프로그램을 **C:\doit** 디렉터리에 **hello.py** 라는 이름으로 저장했다.

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

hello.py 프로그램을 실행하기 위해 먼저 명령 프롬프트 창을 열어 보자. [윈도우키+ R] 을 누르면 다음과 같은 실행 창이 나타난다.



빈 칸에 'cmd'를 입력하고 [Enter]를 누르면 명령 프롬프트 창이 열린다. 첫 번째 줄에는 다음과 같은 프롬프트가 나타나 있을 것이다. `C:\Users\pahkey`에서 `pahkey`는 필자의 컴퓨터 이름이다 (물론 여러분의 명령 프롬프트는 여러분의 컴퓨터 이름으로 표시될 것이다).

```
1 C:\Users\pahkey>
```

맥 사용자의 경우 명령 프롬프트 창 대신 터미널을 사용하자.

이제 다음과 같이 프롬프트에서 `cd c:\doit`라고 입력한다. `cd`(change directory) 명령어는 바로 다음에 나오는 디렉터리로 프롬프트를 옮긴다.

```
1 C:\Users\pahkey> cd c:\doit
2
3 c:\doit>
```

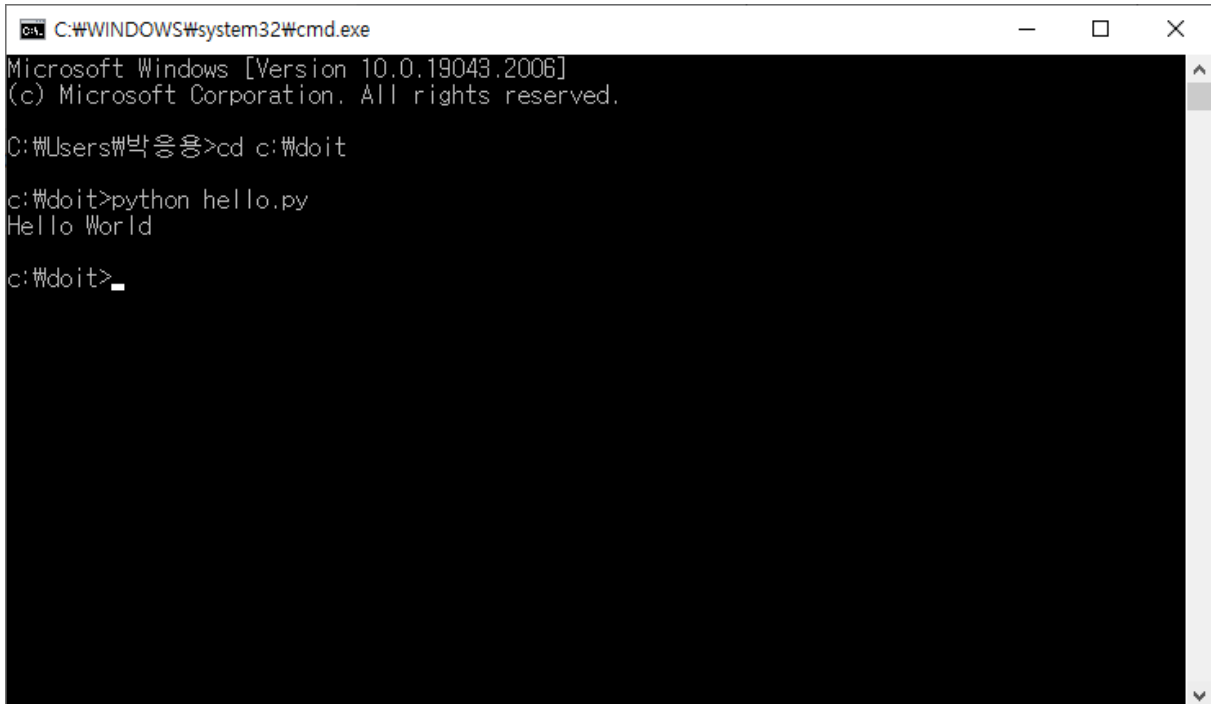
`c:\doit`에서 역슬래시 문자 (\)는 키보드의 엔터키 바로 위에 있는 키 (원화 표시 문자)를 입력하자.

프롬프트가 `c:\doit>`으로 바뀐 것을 확인할 수 있다. 이제 명령어 `python`과 실행할 프로그램 이름을 입력하면 파이썬 프로그램을 실행할 수 있다. 다음을 입력해 `hello.py` 프로그램을 실행해 보자.

```
1 c:\doit> python hello.py
2 Hello World
```

위와 같이 `hello.py` 프로그램이 실행되는 것을 확인할 수 있다. 만약 결괏값이 위와 같지 않다면 `hello.py` 파일이 `C:\doit` 디렉터리에 존재하는지 다시 한 번 살펴보자.

다음은 명령 프롬프트 창에서 위 예제를 실행한 화면이다.



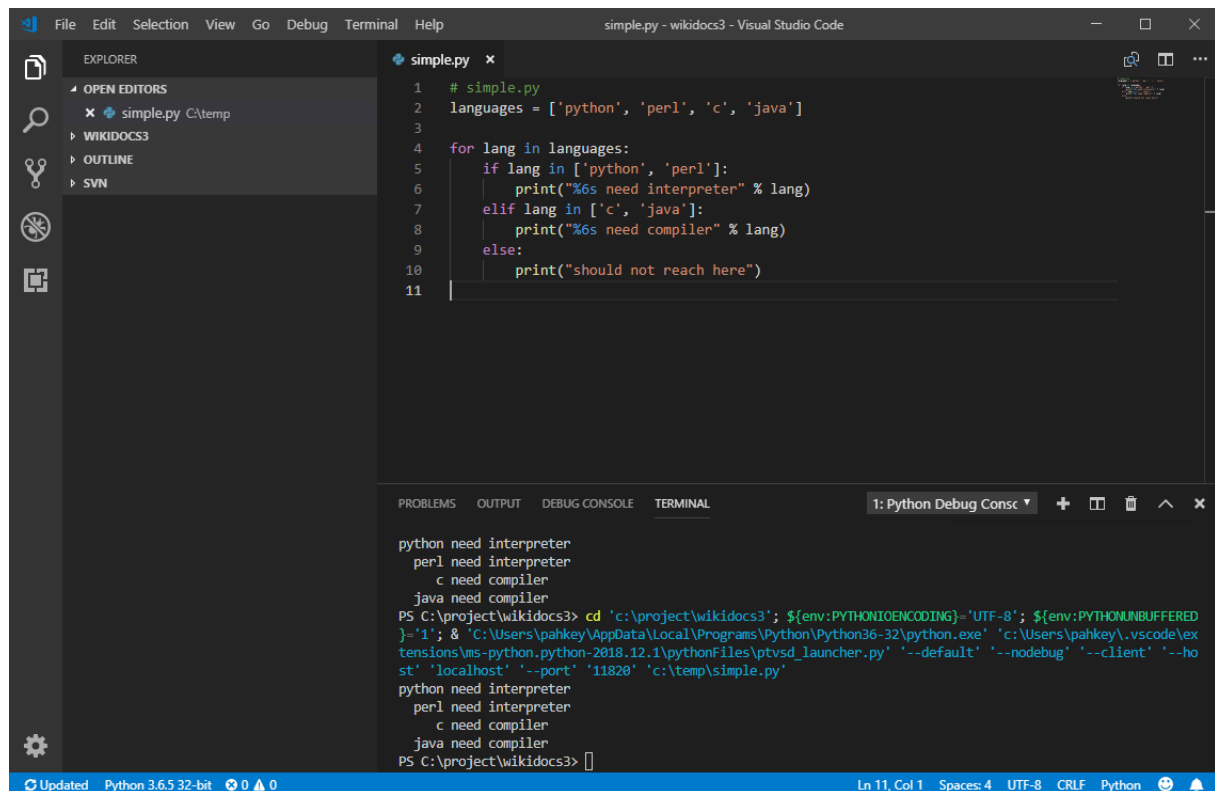
추천 에디터

파이썬으로 실무 프로그램을 작성할 때는 위에 소개한 IDLE 보다 많은 기능을 갖춘 전문 에디터가 유리하다. 다음에 추천할 만한 에디터를 소개하니 참고하자.

이 책의 예제 중 에디터를 사용하여 프로그램을 작성해야 하는 예제는 파이썬 공부에 최적화된 IDLE 을 사용하자. 다음에 소개하는 에디터들은 파이썬 공부를 끝낸 후에 사용해 보기를 권장한다.

비주얼 스튜디오 코드

비주얼 스튜디오 코드 (Visual Studio Code) 는 파이썬과 더불어 프로그래머들에게 가장 많은 사랑을 받는 파이썬의 대표적인 에디터이다. 비주얼 스튜디오 코드는 공식 다운로드 사이트 (<https://code.visualstudio.com>) 에서 내려받을 수 있다.



The screenshot shows the Visual Studio Code interface. The Explorer panel on the left shows the file structure with 'simple.py' open. The main editor area displays the following Python code:

```
1 # simple.py
2 languages = ['python', 'perl', 'c', 'java']
3
4 for lang in languages:
5     if lang in ['python', 'perl']:
6         print("%6s need interpreter" % lang)
7     elif lang in ['c', 'java']:
8         print("%6s need compiler" % lang)
9     else:
10        print("should not reach here")
11
```

The Terminal panel at the bottom shows the output of running the script:

```
python need interpreter
perl need interpreter
c need compiler
java need compiler
PS C:\project\wikidocs3> cd 'c:\project\wikidocs3'; $(env:PYTHONIOENCODING='UTF-8'; $(env:PYTHONUNBUFFERED)=1; & 'C:\Users\pahkey\AppData\Local\Programs\Python\Python36-32\python.exe' 'c:\Users\pahkey\.vscode\extensions\ms-python.python-2018.12.1\pythonFiles\ptvsd_launcher.py' '--default' '--nodebug' '--client' '--host' 'localhost' '--port' '11820' 'c:\temp\simple.py')
python need interpreter
perl need interpreter
c need compiler
java need compiler
PS C:\project\wikidocs3>
```

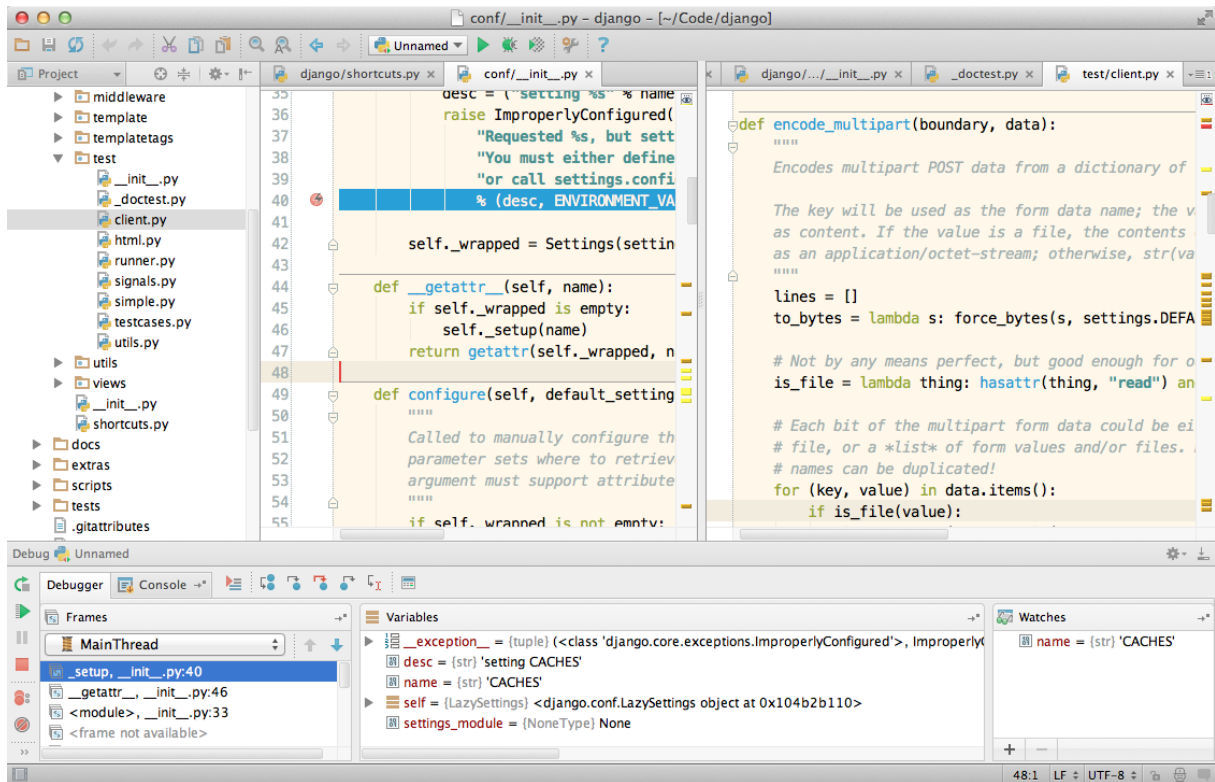
비주얼 스튜디오 코드는 파이썬 전용 에디터가 아니다 (파이썬 외에 여러 가지 언어를 지원하는 에디터이다). 따라서 비주얼 스튜디오 코드를 설치한 후 파이썬 편집을 위해 가장 먼저 해야 할 일은 파이썬 Extension을 설치하는 것이다. 파이썬 Extension은 비주얼 스튜디오 코드를 실행한 후 Extension 메뉴를 사용하여 설치할 수 있다.

파이참

파이썬에 어느 정도 익숙해졌다면 파이참 (PyCharm)을 사용해 보기를 적극 추천한다. 파이참은 가장 유명한 파이썬 에디터 중 하나로서 코드를 작성할 때 자동 완성, 문법 체크 등 편리한 기능을 많이 제공한다.

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

점프 투 파이썬



파이썬은 파이썬 전용 에디터로서 별도의 Extension 이나 플러그인을 따로 설치할 필요가 없다.

02 장 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 계산할지부터 알아야 하고, 데이터베이스 프로그램을 만들려면 어떤 자료를 저장할지부터 알아야 하는 것처럼 기본 중의 기본이다. 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

[추천 동영상 강의] : <https://www.youtube.com/watch?v=9H79uXxQCBk>

숫자형이란?

숫자형 (Number) 이란 숫자 형태로 이루어진 자료형으로, 우리가 이미 잘 알고 있는 것이다. 우리가 흔히 사용하는 숫자들을 생각해 보자. 123 같은 정수, 12.34 같은 실수, 드물게 사용하긴 하지만 8 진수나 16 진수 같은 것도 있다.

다음 표는 파이썬에서 숫자를 어떻게 사용하는지 간략하게 보여 준다.

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

이제 이런 숫자를 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 0 을 변수 a 에 대입하는 예이다.

```
1 >>> a = 123
2 >>> a = -178
3 >>> a = 0
```

실수형

파이썬에서 실수형 (Floating-point) 은 소수점이 포함된 숫자를 말한다. 다음은 실수를 변수 **a** 에 대입하는 예이다.

```
1 >>> a = 1.2
2 >>> a = -3.45
```

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

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

위 방식은 “컴퓨터식 지수 표현 방식” 으로 파이썬에서는 **4.24e10** 또는 **4.24E10** 처럼 표현한다 (**e** 와 **E** 둘 중 어느 것을 사용해도 무방하다). 여기서 **4.24E10** 은 $4.24 * 10^{10}$, **4.24e-10** 은 $4.24 * 10^{-10}$ 을 의미한다.

8 진수와 16 진수

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

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

16 진수 (Hexadecimal) 를 만들기 위해서는 **0x** 로 시작하면 된다.

```
1 >>> a = 0x8ff
2 >>> b = 0xABC
```

8 진수나 16 진수는 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 다음처럼 연산자를 사용해 사칙연산을 수행한다.

```
1 >>> a = 3
2 >>> b = 4
3 >>> a + b
4 7
5 >>> a - b
6 -1
7 >>> a * b
8 12
9 >>> a / b
10 0.75
```

x의 y 제곱을 나타내는 ** 연산자

다음으로 알아야 할 연산자로 **라는 연산자가 있다. 이 연산자는 $x ** y$ 처럼 사용했을 때 x 의 y 제곱 (xy) 값을 리턴한다. 다음의 예를 통해 알아보자.

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

나눗셈 후 나머지를 반환하는 % 연산자

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

```
1 >>> 7 % 3
2 1
3 >>> 3 % 7
4 3
```

$$\begin{array}{r} 2 \\ 3 \overline{) 7} \\ \underline{6} \\ 1 \longrightarrow \text{나머지} \end{array}$$

나눗셈 후 몫을 반환하는 // 연산자

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

```
1 >>> 7 / 4
2 1.75
```

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

```
1 >>> 7 // 4
2 1
```

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

$$\begin{array}{r} 1 \longrightarrow \text{몫} \\ 4 \overline{) 7} \\ \underline{4} \\ 3 \end{array}$$

02-2 문자열 자료형

[추천 동영상 강의] : <https://www.youtube.com/watch?v=STGCSlt3nsE>

문자열이란?

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

```
1 "Life is too short, You need Python"
2 "a"
3 "123"
```

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

문자열은 어떻게 만들고 사용할까?

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

1. 큰따옴표 (" ") 로 양쪽 둘러싸기

```
1 "Hello World"
```

2. 작은따옴표 (' ') 로 양쪽 둘러싸기

```
1 'Python is fun'
```

3. 큰따옴표 3 개를 연속 ("'''") 으로 써서 양쪽 둘러싸기

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

4. 작은따옴표 3 개를 연속 ('''') 으로 써서 양쪽 둘러싸기

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

단순함이 자랑인 파이썬이 문자열을 만드는 방법은 왜 4 가지나 가지게 되었을까? 그 이유에 대해 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 보자.

1. 문자열에 작은따옴표 (') 포함시키기

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

위와 같은 문자열을 `food` 변수에 저장하고 싶다고 가정하자. 문자열 중 `Python's` 에 작은따옴표 (') 가 포함되어 있다.

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

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

프롬프트에 `food` 를 입력해서 결과를 확인하자. 변수에 저장된 문자열이 그대로 출력되는 것을 볼 수 있다.

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

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

```
1 >>> food = 'Python's favorite food is perl'
2   File "<stdin>", line 1
3       food = 'Python's favorite food is perl'
4               ^
5   SyntaxError: invalid syntax
```

2. 문자열에 큰따옴표 (") 포함시키기

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

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

다음과 같이 문자열을 작은따옴표 (') 로 둘러싸면 된다.

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

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

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

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

작은따옴표 (') 나 큰따옴표 (") 를 문자열에 포함시키는 또 다른 방법은 백슬래시 (\) 를 사용하는 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 실행한 후 위 예문을 꼭 직접 작성해 보자.

여러 줄인 문자열을 변수에 대입하고 싶을 때

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

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

1. 줄을 바꾸기 위한 이스케이프 코드 \n 삽입하기

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

위 예처럼 줄바꿈 문자 \n을 삽입하는 방법이 있지만 읽기에 불편하고 줄이 길어지는 단점이 있다.

2. 연속된 작은따옴표 3 개 (''') 또는 큰따옴표 3 개 (""") 사용하기

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

```
1 >>> multiline='''
2 ... Life is too short
3 ... You need python
4 ... '''
```

작은따옴표 3 개를 사용한 경우

```
1 >>> multiline="""
2 ... Life is too short
3 ... You need python
4 ... """
```

큰따옴표 3 개를 사용한 경우

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

```
1 >>> print(multiline)
2 Life is too short
3 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 `n` 을 조합한 `\n` 이스케이프 코드를 사용했다. 이스케이프 코드란 프로그래밍할 때 사용할 수 있도록 미리 정의해 둔 “문자 조합” 이다. 주로 출력물을 보기 좋게 정렬하는 용도로 사용한다. 몇 가지 이스케이프 코드를 정리하면 다음과 같다.

코드	설명
<code>\n</code>	문자열 안에서 줄을 바꿀 때 사용
<code>\t</code>	문자열 사이에 탭 간격을 줄 때 사용
<code>\\</code>	문자 <code>\</code> 를 그대로 표현할 때 사용
<code>\'</code>	작은따옴표 (<code>'</code>) 를 그대로 표현할 때 사용
<code>\"</code>	큰따옴표 (<code>"</code>) 를 그대로 표현할 때 사용
<code>\r</code>	캐리지 리턴 (줄 바꿈 문자, 현재 커서를 가장 앞으로 이동)
<code>\f</code>	폼 피드 (줄 바꿈 문자, 현재 커서를 다음 줄로 이동)
<code>\a</code>	벨 소리 (출력할 때 PC 스피커에서 ‘뽁’ 소리가 난다)
<code>\b</code>	백 스페이스
<code>\000</code>	널 문자

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

문자열 연산하기

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

문자열 더해서 연결하기 (Concatenation)

```
1 >>> head = "Python"
2 >>> tail = " is fun!"
3 >>> head + tail
4 'Python is fun!'
```

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

직접 실행해 보고 결과값이 제시한 것과 똑같이 나오는지 확인해 보자.

문자열 곱하기

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

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

문자열 곱하기 응용

문자열 곱하기를 좀 더 응용해 보자. 다음 소스를 IDLE 에디터를 열고 작성해 보자.

```
1 # multistring.py
2
3 print("=" * 50)
4 print("My Program")
5 print("=" * 50)
```

입력한 소스는 C:\doit 디렉터리에 파일 이름 multistring.py 로 저장하자.

이제 프로그램을 실행해 보자. [윈도우+ R(실행) → cmd 입력 → Enter] 를 눌러 명령 프롬프트 창을 열고 다음을 따라 해 보자. 결괏값이 다음과 같이 나타날 것이다.

```
1 C:\Users>cd C:\doit
2 C:\doit>python multistring.py
3 =====
4 My Program
5 =====
```

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

문자열 길이 구하기

문자열의 길이는 다음과 같이 len 함수를 사용하면 구할 수 있다. len 함수는 print 함수처럼 파이썬의 기본 내장 함수로 별다른 설정 없이 바로 사용할 수 있다.

```
1 >>> a = "Life is too short"
2 >>> len(a)
3 17
```

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?

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

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

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

“Life is too short, You need Python” 문자열에서 L 은 첫 번째 자리를 뜻하는 숫자 0, 바로 다음인 i 는 1 이런 식으로 계속 번호를 붙인 것이다. 중간에 있는 short 의 s 는 12 가 된다.

이제 다음 예를 실행해 보자.


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

a[3] 이 뜻하는 것은 a 라는 문자열의 네 번째 문자 e 를 말한다. 프로그래밍을 처음 접하는 독자라면 a[3] 에서 숫자 3 이 왜 네 번째 문자를 뜻하는지 의아할 수도 있다. 사실 이 부분이 헷갈릴 수 있는 부분인데, 이렇게 생각하면 쉽게 알 수 있을 것이다.

“파이썬은 0 부터 숫자를 센다.”

따라서 파이썬은 위 문자열을 다음과 같이 바라보고 있다.

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

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

문자열 인덱싱 활용하기

인덱싱 예를 몇 가지 더 보자.

```
1 >>> a = "Life is too short, You need Python"
2 >>> a[0]
3 'L'
4 >>> a[12]
5 's'
6 >>> a[-1]
7 '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 위해 마이너스 (-) 기호를 붙이는 것이다. 즉 a[-1] 은 뒤에서부터 세어 첫 번째가 되는 문자를 말한다. a 의 값은 “Life is too short, You need Python” 문자열이므로 뒤에서부터 첫 번째 문자는 가장 마지막 문자’n’이다.

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

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

계속해서 몇 가지 예를 더 보자.

```
1 >>> a[-2]
2 'o'
```

```
3 >>> a[-5]
4 'y'
```

위 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” 문자열에서 단순히 한 문자만을 뽑아내는 것이 아니라 ‘Life’ 또는 ‘You’ 같은 단어를 뽑아내는 방법은 없을까?

다음과 같이 하면 된다.

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

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

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

위 예는 슬라이싱 기법으로 다음과 같이 간단하게 처리할 수 있다.

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

a[0:4] 가 뜻하는 것은 a 문자열, 즉 “Life is too short, You need Python” 문장에서 자리 번호 0 부터 4 까지의 문자를 뽑아낸다는 뜻이다. 하지만 다음과 같은 의문이 생길 것이다. a[0] 은 L, a[1] 은 i, a[2] 는 f, a[3] 은 e 니까 a[0:3] 으로도 Life 라는 단어를 뽑아낼 수 있지 않을까? 다음 예로 확인해 보자.

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

이렇게 되는 이유는 간단하다. 슬라이싱 기법으로 a[시작 번호: 끝 번호] 를 지정할 때 끝 번호에 해당하는 것은 포함하지 않기 때문이다. a[0:3] 을 수식으로 나타내면 다음과 같다.

```
1 0 <= a < 3
```

이 수식을 만족하는 것은 a[0], a[1], a[2] 이다. 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 연습 문제를 많이 풀어 보면서 몸에 익히기 바란다.

문자열을 슬라이싱하는 방법

슬라이싱의 예를 조금 더 보자.

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

위 예는 `a[0] + a[1] + a[2] + a[3] + a[4]` 와 동일하다. `a[4]` 는 공백 문자이기 때문에 'Life' 가 아닌 'Life ' 가 출력된다. 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.

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

`a[시작 번호: 끝 번호]` 에서 끝 번호 부분을 생략하면 시작 번호부터 그 문자열의 끝까지 뽑아낸다.

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

`a[시작 번호: 끝 번호]` 에서 시작 번호를 생략하면 문자열의 처음부터 끝 번호까지 뽑아낸다.

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

`a[시작 번호: 끝 번호]` 에서 시작 번호와 끝 번호를 생략하면 문자열의 처음부터 끝까지를 뽑아낸다.

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

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

```
1 >>> a[19:-7]
2 '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.

```
1 >>> a = "20010331Rainy"
2 >>> date = a[:8]
3 >>> weather = a[8:]
4 >>> date
5 '20010331'
6 >>> weather
7 'Rainy'
```

위 예는 문자열 `a` 를 두 부분으로 나누는 기법이다. 숫자 `8` 을 기준으로 문자열 `a` 를 양쪽으로 한 번씩 슬라이싱했다. `a[:8]` 은 `a[8]` 이 포함되지 않고, `a[8:]` 은 `a[8]` 을 포함하기 때문에 `8` 을 기준으로 해서 두 부분으로 나눌 수 있는 것이다. 위 예에서는 “20010331Rainy” 문자열을 날짜를 나타내는 부분인 ‘20010331’ 과 날씨를 나타내는 부분인 ‘Rainy’ 로 나누는 방법을 보여 준다.

위 문자열 “20010331Rainy” 를 연도 2001, 월과 일을 나타내는 0331, 날씨를 나타내는 Rainy 의 세 부분으로 나누려면 다음과 같이 할 수 있다.

```
1 >>> a = "20010331Rainy"
2 >>> year = a[:4]
3 >>> day = a[4:8]
4 >>> weather = a[8:]
5 >>> year
6 '2001'
7 >>> day
8 '0331'
9 >>> weather
10 'Rainy'
```

위 예는 숫자 `4` 와 `8` 로 “20010331Rainy” 문자열을 세 부분으로 나누는 방법을 보여 준다.

지금까지 인덱싱과 슬라이싱에 대해서 살펴보았다. 인덱싱과 슬라이싱은 프로그래밍을 할 때 매우 자주 사용하는 기법이니 꼭 반복해서 연습해 두자.

[“Pithon”이라는 문자열을 “Python”으로 바꾸려면?]

Pithon 문자열을 Python으로 바꾸려면 어떻게 해야 할까? 제일 먼저 떠오르는 생각은 다음과 같을 것이다.

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

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

당연히 오류가 발생한다. 왜냐하면 문자열의 요솟값은 바꿀 수 있는 값이 아니기 때문이다 (문자열 자료형은 그 요솟값을 변경할 수 없다. 그래서 **immutable**한 자료형이라고도 부른다).

하지만 앞에서 살펴본 슬라이싱 기법을 사용하면 Pithon 문자열을 사용해 Python 문자열을 만들 수 있다.

다음 예를 보자.

```
1 >>> a = "Pithon"
2 >>> a[:1]
3 'P'
4 >>> a[2:]
5 'thon'
6 >>> a[:1] + 'y' + a[2:]
7 'Python'
```

위 예에서 볼 수 있듯이 슬라이싱을 사용하면 “Pithon” 문자열을 ‘P’ 부분과 ‘thon’ 부분으로 나눌 수 있기 때문에 그 사이에 ‘y’ 문자를 추가하여 ‘Python’이라는 새로운 문자열을 만들 수 있다.

문자열 포매팅

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

“현재 온도는 18 도입니다.”

시간이 지나서 20도가 되면 다음 문장을 출력한다.

“현재 온도는 20 도입니다”

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

쉽게 말해 문자열 포매팅이란 문자열 안에 어떤 값을 삽입하는 방법이다. 다음 예를 직접 실행해 보면서

그 사용법을 알아보자.

문자열 포매팅 따라 하기

1. 숫자 바로 대입

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

위 예제의 결과값을 보면 알겠지만 위 예제는 문자열 안에 정수 3을 삽입하는 방법을 보여 준다. 문자열 안에서 숫자를 넣고 싶은 자리에 %d 문자를 넣어 주고, 삽입할 숫자 3은 가장 뒤에 있는 % 문자 다음에 써 넣었다. 여기에서 %d는 문자열 포맷 코드라고 부른다.

2. 문자열 바로 대입

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

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

위 예제에서는 문자열 안에 또 다른 문자열을 삽입하기 위해 앞에서 사용한 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 배운 것처럼 큰따옴표나 작은따옴표를 반드시 써주어야 한다.

3. 숫자 값을 나타내는 변수로 대입

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

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

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

문자열 포맷 코드

문자열 포매팅 예제에서는 대입해 넣는 자료형으로 정수와 문자열을 사용했으나 이 외에도 다양한 것을 대입할 수 있다. 문자열 포맷 코드로는 다음과 같은 것이 있다.

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

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

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

3을 문자열 안에 삽입하려면 %d 를 사용하고, 3.234 를 삽입하려면 %f 를 사용해야 한다. 하지만 %s 를 사용하면 이런 것을 생각하지 않아도 된다. 왜냐하면 %s 는 자동으로 % 뒤에 있는 값을 문자열로 바꾸기 때문이다.

[포매팅 연산자%d 와% 를 같이 쓸 때는%% 를 쓴다]

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

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

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

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

따라서 위 예를 제대로 실행하려면 다음과 같이 해야 한다.

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

포맷 코드와 숫자 함께 사용하기

위에서 보았듯이%d, %s 등의 포맷 코드는 문자열 안에 어떤 값을 삽입하기 위해 사용한다. 하지만 포맷 코드를 숫자와 함께 사용하면 더 유용하게 사용할 수 있다. 다음 예를 보고 따라해 보자.

1. 정렬과 공백

```
1 >>> "%10s" % "hi"
2 '          hi'
```

앞의 예문에서 %10s는 전체 길이가 10 개인 문자열 공간에서 대입되는 값을 오른쪽으로 정렬하고 그 앞의 나머지는 공백으로 남겨 두라는 의미이다.

그렇다면 반대쪽인 왼쪽 정렬은 %-10s가 될 것이다. 확인해 보자.

```
1 >>> "%-10sjane." % 'hi'
2 'hi      jane.'
```

hi 를 왼쪽으로 정렬하고 나머지는 공백으로 채웠음을 볼 수 있다.

2. 소수점 표현하기

```
1 >>> "%0.4f" % 3.42134234
2 '3.4213'
```

3.42134234 를 소수점 네 번째 자리까지만 나타내고 싶은 경우에는 위와 같이 사용한다. 즉 여기서? 의 의

미는 소수점 포인트를 말하고 그 뒤의 숫자 4는 소수점 뒤에 나올 숫자의 개수를 말한다.

다음 예를 살펴보자.

```
1 >>> "%10.4f" % 3.42134234
2 '      3.4213'
```

위 예는 숫자 3.42134234를 소수점 네 번째 자리까지만 표시하고 전체 길이가 10개인 문자열 공간에서 오른쪽으로 정렬하는 예를 보여 준다.

format 함수를 사용한 포매팅

문자열의 format 함수를 사용하면 좀 더 발전된 스타일로 문자열 포맷을 지정할 수 있다. 앞에서 살펴본 문자열 포매팅 예제를 format 함수를 사용해서 바꾸면 다음과 같다.

숫자 바로 대입하기

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

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

문자열 바로 대입하기

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

문자열의 {0} 항목이 five라는 문자열로 바뀌었다.

숫자 값을 가진 변수로 대입하기

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

문자열의 {0} 항목이 number 변수 값인 3으로 바뀌었다.

2개 이상의 값 넣기

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

2개 이상의 값을 넣을 경우 문자열의 {0}, {1}과 같은 인덱스 항목이 format 함수의 입력값으로 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

```
1 >>> "I ate {number} apples. so I was sick for {day} days.".format(number=10,  
    day=3)  
2 '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 사용하는 방법도 있다. {name} 형태를 사용할 경우 format 함수에는 반드시 name=value 와 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

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

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

왼쪽 정렬

```
1 >>> "{0:<10}".format("hi")  
2 'hi'
```

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

오른쪽 정렬

```
1 >>> "{0:>10}".format("hi")  
2 '          hi'
```

오른쪽 정렬은 :< 대신 :>을 사용하면 된다. 화살표 방향을 생각하면 어느 쪽으로 정렬되는지 바로 알 수 있을 것이다.

가운데 정렬

```
1 >>> "{0:^10}".format("hi")  
2 '      hi      '
```

:^ 기호를 사용하면 가운데 정렬도 가능하다.

공백 채우기

```
1 >>> "{0:==^10}".format("hi")  
2 '====hi===='  
3 >>> "{0:!!<10}".format("hi")  
4 'hi!!!!!!!!'
```

정렬할 때 공백 문자 대신에 지정한 문자 값으로 채워 넣는 것도 가능하다. 채워 넣을 문자 값은 정렬 문자

<, >, ^ 바로 앞에 넣어야 한다. 위 예에서 첫 번째 예제는 가운데 (^) 로 정렬하고 빈 공간을 = 문자로 채웠고, 두 번째 예제는 왼쪽 (<) 으로 정렬하고 빈 공간을 ! 문자로 채웠다.

소수점 표현하기

```
1 >>> y = 3.42134234
2 >>> "{0:0.4f}".format(y)
3 '3.4213'
```

위 예는 format 함수를 사용해 소수점을 4 자리까지만 표현하는 방법을 보여 준다. 앞에서 살펴보았던 표현식 0.4f 를 그대로 사용한 것을 알 수 있다.

```
1 >>> "{0:10.4f}".format(y)
2 '      3.4213'
```

위와 같이 자릿수를 10 으로 맞출 수도 있다. 역시 앞에서 살펴본 10.4f 의 표현식을 그대로 사용한 것을 알 수 있다.

{ 또는 } 문자 표현하기

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

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

f 문자열 포매팅

파이썬 3.6 버전부터는 f 문자열 포매팅 기능을 사용할 수 있다. 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.

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

f 문자열 포매팅은 위와 같이 name, age 와 같은 변수 값을 생성한 후에 그 값을 참조할 수 있다. 또한 f 문자열 포매팅은 표현식을 지원하기 때문에 다음과 같은 것도 가능하다.

표현식이란 문자열 안에서 변수와 +, -와 같은 수식을 함께 사용하는 것을 말한다.

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

딕셔너리는 f 문자열 포매팅에서 다음과 같이 사용할 수 있다.

딕셔너리는 Key 와 Value 라는 것을 한 쌍으로 갖는 자료형이다. 02-5 에서 자세히 알아본다.

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

정렬은 다음과 같이 할 수 있다.

```
1 >>> f'{"hi":<10}' # 왼쪽 정렬
2 'hi '
3 >>> f'{"hi":>10}' # 오른쪽 정렬
4 '   hi'
5 >>> f'{"hi":^10}' # 가운데 정렬
6 '  hi  '
```

공백 채우기는 다음과 같이 할 수 있다.

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

소수점은 다음과 같이 표현할 수 있다.

```
1 >>> y = 3.42134234
2 >>> f'{y:0.4f}' # 소수점 4자리까지만 표현
3 '3.4213'
4 >>> f'{y:10.4f}' # 소수점 4자리까지 표현하고 총 자리수를 10으로 맞춤
5 '      3.4213'
```

f 문자열에서 { } 문자를 표시하려면 다음과 같이 두 개를 동시에 사용해야 한다.

```
1 >>> f'{{ and }}'
2 '{ and }'
```

지금까지는 문자열을 가지고 할 수 있는 기본적인 것에 대해 알아보았다. 이제부터는 문자열을 좀 더 자유재로 다루기 위해 공부해야 할 것을 설명할 것이다. 지쳤다면 잠시 책을 접고 휴식을 취하자.

문자열 관련 함수들

문자열 자료형은 자체적으로 함수를 가지고 있다. 이들 함수를 다른 말로 문자열 내장 함수라 한다. 이 내장 함수를 사용하려면 문자열 변수 이름 뒤에 ‘.’를 붙인 다음에 함수 이름을 써주면 된다. 이제 문자열의 내장 함수에 대해서 알아보자.

문자 개수 세기 (count)

```
1 >>> a = "hobby"
2 >>> a.count('b')
3 2
```

문자열 중 문자 **b** 의 개수를 리턴한다.

위치 알려주기 1(find)

```
1 >>> a = "Python is the best choice"
2 >>> a.find('b')
3 14
4 >>> a.find('k')
5 -1
```

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

파이썬은 숫자를 0 부터 세기 때문에 **b** 의 위치는 15 가 아닌 14 가 된다.

위치 알려주기 2(index)

```
1 >>> a = "Life is too short"
2 >>> a.index('t')
3 8
4 >>> a.index('k')
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   ValueError: substring not found
```

문자열 중 문자 **t** 가 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 발생하는 점이다.

문자열 삽입 (join)

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

abcd 문자열의 각각의 문자 사이에 **,** 를 삽입한다.

join 함수는 문자열뿐만 아니라 앞으로 배울 리스트나 튜플도 입력으로 사용할 수 있다 (리스트와 튜플은 곧 배울 내용이니 여기에서는 잠시 눈으로만 살펴보자). join 함수의 입력으로 리스트를 사용하는 예는 다음과 같다.

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

소문자를 대문자로 바꾸기 (upper)

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

upper 함수는 소문자를 대문자로 바꾸어 준다. 만약 문자열이 이미 대문자라면 아무 변화도 일어나지 않을 것이다.

대문자를 소문자로 바꾸기 (lower)

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

lower 함수는 대문자를 소문자로 바꾸어 준다.

왼쪽 공백 지우기 (lstrip)

```
1 >>> a = " hi "
2 >>> a.lstrip()
3 'hi '
```

문자열 중 가장 왼쪽에 있는 한 칸 이상의 연속된 공백들을 모두 지운다. lstrip 에서 l 은 left 를 의미한다.

오른쪽 공백 지우기 (rstrip)

```
1 >>> a = " hi "
2 >>> a.rstrip()
3 ' hi'
```

문자열 중 가장 오른쪽에 있는 한 칸 이상의 연속된 공백을 모두 지운다. rstrip 에서 r 는 right 를 의미한다.

양쪽 공백 지우기 (strip)

```
1 >>> a = " hi "  
2 >>> a.strip()  
3 'hi'
```

문자열 양쪽에 있는 한 칸 이상의 연속된 공백을 모두 지운다.

문자열 바꾸기 (replace)

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

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

문자열 나누기 (split)

```
1 >>> a = "Life is too short"  
2 >>> a.split()  
3 ['Life', 'is', 'too', 'short']  
4 >>> b = "a:b:c:d"  
5 >>> b.split(':')  
6 ['a', 'b', 'c', 'd']
```

split 함수는 `a.split()`처럼 괄호 안에 아무 값도 넣어 주지 않으면 공백 (스페이스, 탭, 엔터 등) 을 기준으로 문자열을 나누어 준다. 만약 `b.split(':')`처럼 괄호 안에 특정 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']`가 리스트인데 02-3 에서 자세히 알아볼 것이니 여기에서는 너무 신경 쓰지 않아도 된다.

위에서 소개한 문자열 관련 함수는 문자열 처리에서 사용 빈도가 매우 높고 유용하다. 이 외에도 몇 가지가 더 있지만 자주 사용되지는 않는다.

02-3 리스트 자료형

[추천 동영상 강의] : <https://www.youtube.com/watch?v=x6W7t0KkTAc>

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

리스트는 어떻게 만들고 사용할까?

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

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

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

```
1 리스트명 = [요소1, 요소2, 요소3, ...]
```

여러 가지 리스트의 생김새를 살펴보면 다음과 같다.

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

리스트는 **a** 처럼 아무것도 포함하지 않아 비어 있는 리스트 ([]) 일 수도 있고 **b** 처럼 숫자를 요소값으로 가질 수도 있고 **c** 처럼 문자열을 요소값으로 가질 수도 있다. 또한 **d** 처럼 숫자와 문자열을 함께 요소값으로 가질 수도 있으며 **e** 처럼 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 이해 하자.

리스트의 인덱싱

리스트 역시 문자열처럼 인덱싱을 적용할 수 있다. 먼저 **a** 변수에 **[1, 2, 3]** 값을 설정한다.

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

a[0] 은 리스트 **a** 의 첫 번째 요솟값을 말한다.

```
1 >>> a[0]
2 1
```

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

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

이것은 **1+3** 으로 해석되어 값 **4** 를 출력한다.

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

```
1 >>> a[-1]
2 3
```

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

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

그리고 다음 예를 따라 해 보자.

```
1 >>> a[0]
2 1
3 >>> a[-1]
4 ['a', 'b', 'c']
5 >>> a[3]
6 ['a', 'b', 'c']
```

예상한 대로 `a[-1]` 은 마지막 요솟값 `['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 여기에서 리스트 `a` 에 포함된 `['a', 'b', 'c']` 리스트에서 `'a'` 값을 인덱싱을 사용해 끄집어낼 수 있는 방법은 없을까? 다음 예를 보자.

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

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

다음 예도 마찬가지로 경우이므로 어렵지 않게 이해될 것이다.

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

[삼중 리스트에서 인덱싱하기]

조금 복잡하지만 다음과 같은 것도 가능하다.

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

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

이 경우 `'Life'` 문자열만 끄집어내려면 다음과 같이 해야 한다.

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

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

리스트를 다중으로 중첩해서 사용하는 것은 혼란스럽기 때문에 자주 사용하지는 않는다.

리스트의 슬라이싱

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

자, 그럼 리스트의 슬라이싱에 대해서 살펴보자.

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

앞의 예를 문자열에서 슬라이싱했던 것과 비교해 보자.

```
1 >>> a = "12345"
2 >>> a[0:2]
3 '12'
```

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

몇 가지 예를 더 들어 보자.

```
1 >>> a = [1, 2, 3, 4, 5]
2 >>> b = a[:2]
3 >>> c = a[2:]
4 >>> b
5 [1, 2]
6 >>> c
7 [3, 4, 5]
```

b 변수는 리스트 a의 첫 번째 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.

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

위 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 연산하기

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

리스트 더하기 (+)

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

```
4 [1, 2, 3, 4, 5, 6]
```

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

리스트 반복하기 (*)

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

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

리스트 길이구하기

리스트 길이를 구하기 위해서는 다음처럼 len 함수를 사용해야 한다.

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

len 함수는 문자열, 리스트 외에 앞으로 배울 튜플과 딕셔너리에도 사용할 수 있는 함수이다. 실습에서 자주 사용하니 잘 기억해 두자.

[초보자가 범하기 쉬운 리스트 연산 오류]

다음 소스 코드를 입력했을 때 결괏값은 어떻게 나올까?

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

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

```
1 Traceback (most recent call last):
2   File "<stdin>", line 1, in <module>
3   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 다음처럼 숫자 3을 문자 '3'으로 바꾸어야 한다.

```
1 >>> str(a[2]) + "hi"
2 3hi
```

str 함수는 정수나 실수를 문자열로 바꾸어 주는 파이썬의 내장 함수이다.

리스트의 수정과 삭제

리스트는 값을 수정하거나 삭제할 수 있다.

리스트에서 값 수정하기

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

a[2]의 요소값 3이 4로 바뀌었다.

del 함수 사용해 리스트 요소 삭제하기

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

`del a[x]` 는 x 번째 요솟값을 삭제한다. 여기에서는 `a` 리스트에서 `a[1]` 을 삭제하는 방법을 보여준다. `del` 함수는 파이썬이 자체적으로 가지고 있는 삭제 함수이며 다음과 같이 사용한다.

```
1 del 객체
```

객체란 파이썬에서 사용되는 모든 자료형을 말한다.

다음처럼 슬라이싱 기법을 사용하여 리스트의 요소 여러 개를 한꺼번에 삭제할 수도 있다.

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

`a[2:]` 에 해당하는 리스트의 요소들이 삭제되었다.

리스트의 요소를 삭제하는 방법에는 2 가지가 더 있다. 그것은 리스트의 `remove` 와 `pop` 함수를 사용하는 방법인데 이것에 대해서는 바로 이어지는 리스트 관련 함수에서 설명한다.

리스트 관련 함수들

문자열과 마찬가지로 리스트 변수 이름 뒤에 “.” 를 붙여서 여러 가지 리스트 관련 함수를 사용할 수 있다. 유용하게 사용되는 리스트 관련 함수 몇 가지에 대해서만 알아보기로 하자.

리스트에 요소 추가 (`append`)

`append` 를 사전에서 검색해 보면 “덧붙이다, 첨부하다” 라는 뜻이 있다. 이 뜻을 안다면 다음 예가 바로 이해될 것이다. `append(x)` 는 리스트의 맨 마지막에 `x` 를 추가하는 함수이다.

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

리스트 안에는 어떤 자료형도 추가할 수 있다. 다음 예는 리스트에 다시 리스트를 추가한 결과이다.

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

리스트 정렬 (`sort`)

`sort` 함수는 리스트의 요소를 순서대로 정렬해 준다.

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

문자 역시 알파벳 순서로 정렬할 수 있다.

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

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

인덱스 반환 (index)

`index(x)` 함수는 리스트에 `x` 값이 있으면 `x`의 인덱스 값을 리턴한다.

```
1 >>> a = [1,2,3]
2 >>> a.index(3)
3 2
4 >>> a.index(1)
5 0
```

위 예에서 리스트 `a`에 있는 숫자 3의 위치는 `a[2]` 이므로 2를 리턴하고, 숫자 1의 위치는 `a[0]` 이므로 0을 리턴한다.

다음 예에서 값 0은 `a` 리스트에 존재하지 않기 때문에 값 오류 (`ValueError`)가 발생한다.

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

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

리스트에 요소 삽입 (insert)

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

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

위 예는 0 번째 자리, 즉 첫 번째 요소 (`a[0]`) 위치에 값 4 를 삽입하라는 뜻이다.

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

위 예는 리스트 `a` 의 `a[3]`, 즉 네 번째 요소 위치에 값 5 를 삽입하라는 뜻이다.

리스트 요소 제거 (remove)

`remove(x)` 는 리스트에서 첫 번째로 나오는 `x` 를 삭제하는 함수이다.

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

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

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

`remove(3)` 을 한 번 더 실행하면 다시 3 이 삭제된다.

리스트 요소 끄집어내기 (pop)

`pop()` 은 리스트의 맨 마지막 요소를 리턴하고 그 요소는 삭제한다.

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

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

`pop(x)` 는 리스트의 `x` 번째 요소를 리턴하고 그 요소는 삭제한다.

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

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

리스트에 포함된 요소 `x` 의 개수 세기 (`count`)

`count(x)` 는 리스트 안에 `x` 가 몇 개 있는지 조사하여 그 개수를 리턴하는 함수이다.

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

1이라는 값이 리스트 `a` 에 2 개 들어 있으므로 2 를 리턴한다.

리스트 확장 (`extend`)

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

```
1 >>> a = [1,2,3]
2 >>> a.extend([4,5])
3 >>> a
4 [1, 2, 3, 4, 5]
5 >>> b = [6, 7]
6 >>> a.extend(b)
7 >>> a
8 [1, 2, 3, 4, 5, 6, 7]
```

`a.extend([4, 5])` 는 `a += [4, 5]` 와 동일하다.

`a += [4, 5]` 는 `a = a + [4, 5]` 와 동일한 표현식이다.

02-4 튜플 자료형

[추천 동영상 강의] : <https://www.youtube.com/watch?v=YVLqi0sAJzc>

튜플은 어떻게 만들까?

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

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

튜플의 모습은 다음과 같다.

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

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

얼핏 보면 튜플과 리스트는 비슷한 역할을 하지만 프로그래밍을 할 때 튜플과 리스트는 구별해서 사용하는 것이 유리하다. 튜플과 리스트의 가장 큰 차이는 요소값을 변화시킬 수 있는가 여부이다. 즉 리스트의 요소값은 변화가 가능하고 튜플의 요소값은 변화가 불가능하다. 따라서 프로그램이 실행되는 동안 요소값이 항상 변하지 않기를 바란다면 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 사용한다.

튜플의 요소값을 지우거나 변경하려고 하면 어떻게 될까?

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

1. 튜플 요소값을 삭제하려 할 때

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

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

튜플의 요소를 리스트처럼 del 함수로 지우려고 한 예이다. 튜플은 요소값을 지울 수 없다는 오류 메시지를 확인할 수 있다.

2. 튜플 요소값을 변경하려 할 때

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

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

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

튜플 다루기

튜플은 요소값을 변화시킬 수 없다는 점만 제외하면 리스트와 완전히 동일하므로 간단하게만 살펴볼 것이다. 다음 예제는 서로 연관되어 있으므로 차례대로 수행해 보기 바란다.

인덱싱하기

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

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

슬라이싱하기

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

t1[1] 부터 튜플의 마지막 요소까지 슬라이싱하는 예이다.

튜플 더하기

```
1 >>> t1 = (1, 2, 'a', 'b')
2 >>> t2 = (3, 4)
3 >>> t3 = t1 + t2
4 >>> t3
5 (1, 2, 'a', 'b', 3, 4)
```

튜플을 더하는 방법을 보여 주는 예이다. 이 때에도 t1, t2 튜플의 요소값이 바뀌는 것은 아니다. 다만, t1, t2 튜플을 더하여 새로운 튜플 t3 를 생성한 것이다.

튜플 곱하기

```
1 >>> t2 = (3, 4)
2 >>> t3 = t2 * 3
3 >>> t3
4 (3, 4, 3, 4, 3, 4)
```

튜플의 곱하기 (반복) 예를 보여 준다.

튜플 길이 구하기

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

튜플의 길이를 구하는 예이다.

튜플은 요소값을 변경할수 없기 때문에 sort, insert, remove, pop 과 같은 내장 함수가 없다.

02-5 딕셔너리 자료형

[추천 동영상 강의] : <https://www.youtube.com/watch?v=BmXDox6ZFzo>

딕셔너리란?

사람은 누구든지 “이름” = “홍길동”, “생일” = “몇 월 며칠” 등으로 나타낼 수 있다. 파이썬은 영리하게도 이러한 대응 관계를 나타낼 수 있는 자료형을 가지고 있다. 요즘 사용하는 대부분의 언어도 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 “people” 이라는 단어에 “사람”, “baseball” 이라는 단어에 “야구” 라는 뜻이 부합되듯이 딕셔너리는 Key 와 Value 를 한 쌍으로 갖는 자료형이다. 예컨대 Key 가 “baseball” 이라면 Value 는 “야구” 가 될 것이다.

딕셔너리는 리스트나 튜플처럼 순차적으로 (sequential) 해당 요솟값을 구하지 않고 Key 를 통해 Value 를 얻는다. 이것이 바로 딕셔너리의 가장 큰 특징이다. baseball 이라는 단어의 뜻을 찾기 위해 사전의 내용을 순차적으로 모두 검색하는 것이 아니라 baseball 이라는 단어가 있는 곳만 펼쳐 보는 것이다.

딕셔너리는 어떻게 만들까?

다음은 기본 딕셔너리의 모습이다.

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

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

다음 딕셔너리 예를 살펴보자.

```
1 >>> dic = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
```

위에서 Key 는 각각 'name', 'phone', 'birth' 이고, 각각의 Key 에 해당하는 Value 는 'pey', '010-9999-1234', '1118' 이 된다.

딕셔너리 dic 의 정보

key	value
name	pey
phone	010-9999-1234
birth	1118

다음 예는 Key 로 정수 값 1, Value 로 문자열 'hi' 를 사용한 예이다.

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

또한 다음 예처럼 Value 에 리스트도 넣을 수 있다.

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

딕셔너리 쌍 추가, 삭제하기

딕셔너리 쌍을 추가하는 방법과 삭제하는 방법을 살펴보자. 먼저 딕셔너리에 쌍을 추가해 보자.

딕셔너리 쌍 추가하기

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

{1: 'a'} 딕셔너리에 a[2]='b' 와 같이 입력하면 딕셔너리 a 에 Key 와 Value 가 각각 2 와 'b' 인 {2 : 'b'} 딕셔너리 쌍이 추가된다.

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

딕셔너리 a 에 {'name': 'pey'} 쌍이 추가되었다.

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

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

Key 는 3, Value 는 [1, 2, 3] 을 가지는 한 쌍이 또 추가되었다.

딕셔너리 요소 삭제하기

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

위 예제는 딕셔너리 요소를 지우는 방법을 보여 준다. `del` 함수를 사용해서 `del a[key]` 처럼 입력하면 지정 한 Key 에 해당하는 {key : value} 쌍이 삭제된다.

딕셔너리를 사용하는 방법

“딕셔너리는 주로 어떤 것을 표현하는 데 사용할까?” 라는 의문이 들 것이다. 예를 들어 4 명의 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.

다음 예를 보자.

```
1 {"김연아":"피겨스케이팅", "류현진":"야구", "손흥민":"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 것이 더 있다. 이제부터 하나씩 알아보자.

딕셔너리에서 Key 사용해 Value 얻기

다음을 따라해 보자.

```
1 >>> grade = {'pey': 10, 'julliet': 99}
2 >>> grade['pey']
3 10
4 >>> grade['julliet']
5 99
```

리스트나 튜플, 문자열은 요솟값을 얻고자 할 때 인덱싱이나 슬라이싱 기법 중 하나를 사용했다. 하지만 딕셔너리는 단 한 가지 방법뿐이다. 바로 Key 를 사용해서 Value 를 구하는 방법이다. 위 예에

서'pey' 라는 Key 의 Value 를 얻기 위해 `grade['pey']` 를 사용한 것처럼 어떤 Key 의 Value 를 얻기 위해서는 `딕셔너리변수이름[Key]` 를 사용한다.

몇 가지 예를 더 보자.

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

먼저 `a` 변수에 `{1:'a', 2:'b'}` 딕셔너리를 대입하였다. 위 예에서 볼 수 있듯이 `a[1]` 은 'a' 값을 리턴한다. 여기에서 `a[1]` 이 의미하는 것은 리스트나 튜플의 `a[1]` 과는 전혀 다르다. 딕셔너리 변수에서 `[]` 안의 숫자 1 은 두 번째 요소를 뜻하는 것이 아니라 Key 에 해당하는 1 을 나타낸다. 앞에서도 말했듯이 딕셔너리는 리스트나 튜플에 있는 인덱싱 방법을 적용할 수 없다. 따라서 `a[1]` 은 딕셔너리 `{1:'a', 2:'b'}` 에서 Key 가 1 인 것의 Value 인 'a' 를 리턴한다. `a[2]` 역시 마찬가지이다.

이번에는 `a` 라는 변수에 앞의 예에서 사용한 딕셔너리의 Key 와 Value 를 뒤집어 놓은 딕셔너리를 대입해 보자.

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

역시 `a['a']`, `a['b']` 처럼 Key 를 사용해서 Value 를 얻을 수 있다. 정리하면, 딕셔너리 `a` 는 `a[Key]` 로 Key 에 해당하는 Value 를 얻는다.

다음 예는 이전에 한 번 언급한 딕셔너리인데 Key 를 사용해서 Value 를 얻는 방법을 잘 보여 준다.

```
1 >>> dic = {'name':'pey', 'phone':'010-9999-1234', 'birth': '1118'}
2 >>> dic['name']
3 'pey'
4 >>> dic['phone']
5 '010-9999-1234'
6 >>> dic['birth']
7 '1118'
```

딕셔너리 만들 때 주의할 사항

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

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



```
2 >>> a
3 {1: 'b'}
```

이렇게 Key 가 중복되었을 때 1 개를 제외한 나머지 Key:Value 값이 모두 무시되는 이유는 Key 를 통해서 Value 를 얻는 딕셔너리의 특징에서 비롯된다. 즉 동일한 Key 가 존재하면 어떤 Key 에 해당하는 Value 를 불러야 할지 알 수 없기 때문이다.

또 한 가지 주의해야 할 사항은 Key 에 리스트는 쓸 수 없다는 것이다. 하지만 튜플은 Key 로 쓸 수 있다. 딕셔너리의 Key 로 쓸 수 있느냐 없느냐는 Key 가 변하는 (mutable) 값인지 변하지 않는 (immutable) 값인지에 달려 있다. 리스트는 그 값이 변할 수 있기 때문에 Key 로 쓸 수 없다. 다음 예처럼 리스트를 Key 로 설정하면 리스트를 키 값으로 사용할 수 없다는 오류가 발생한다.

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

단, Value 에는 변하는 값이든 변하지 않는 값이든 상관없이 아무 값이나 넣을 수 있다.

딕셔너리 관련 함수들

딕셔너리를 자유자재로 사용하기 위해 딕셔너리가 자체적으로 가지고 있는 관련 함수를 사용해 보자.

Key 리스트 만들기 (keys)

```
1 >>> a = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
2 >>> a.keys()
3 dict_keys(['name', 'phone', 'birth'])
```

a.keys() 는 딕셔너리 a 의 Key 만을 모아서 dict_keys 객체를 리턴한다.

[파이썬 3.0 이후 버전의 keys 함수, 어떻게 달라졌나?]

파이썬 2.7 버전까지는 a.keys() 함수를 호출하면 dict_keys 가 아닌 리스트를 리턴한다. 리스트를 리턴하기 위해서는 메모리 낭비가 발생하는데 파이썬 3.0 이후 버전에서는 이러한 메모리 낭비를 줄이기 위해 dict_keys 객체를 리턴하도록 변경되었다. 다음에 소개할 dict_values, dict_items 역시 파이썬 3.0 이후 버전에서 추가된 것들이다. 만약 3.0 이후 버전에서 리턴 값으로 리스트가 필요한 경우에는 list(a.keys())를 사용하면 된다. dict_keys, dict_values, dict_items 객체는 리스트로 변환하지 않더라도 기본적인 반복 구문 (예: for 문) 에서 사용할 수 있다.

`dict_keys` 객체는 다음과 같이 사용할 수 있다. 리스트를 사용하는 것과 차이가 없지만, 리스트 고유의 `append`, `insert`, `pop`, `remove`, `sort` 함수는 수행할 수 없다.

```
1 >>> for k in a.keys():
2     ...     print(k)
3     ...
4     name
5     phone
6     birth
```

`print(k)` 를 입력할 때 들여쓰기를 하지 않으면 오류가 발생하니 주의하자. `for` 문 등 반복 구문에 대해서는 03 장에서 자세히 살펴본다.

`dict_keys` 객체를 리스트로 변환하려면 다음과 같이 하면 된다.

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

Value 리스트 만들기 (values)

```
1 >>> a.values()
2 dict_values(['pey', '010-9999-1234', '1118'])
```

Key 를 얻는 것과 마찬가지로 Value 만 얻고 싶다면 `values` 함수를 사용하면 된다. `values` 함수를 호출하면 `dict_values` 객체를 돌려준다.

Key, Value 쌍 얻기 (items)

```
1 >>> a.items()
2 dict_items([('name', 'pey'), ('phone', '010-9999-1234'), ('birth', '1118')])
```

`items` 함수는 Key 와 Value 의 쌍을 튜플로 묶은 값을 `dict_items` 객체로 돌려준다.

Key: Value 쌍 모두 지우기 (clear)

```
1 >>> a.clear()
2 >>> a
3 {}
```

`clear` 함수는 딕셔너리 안의 모든 요소를 삭제한다.

빈 리스트를 [], 빈 튜플을 ()로 표현하는 것과 마찬가지로 빈 딕셔너리도 {}로 표현한다.

Key 로 Value 얻기 (get)

```
1 >>> a = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
2 >>> a.get('name')
3 'pey'
4 >>> a.get('phone')
5 '010-9999-1234'
```

get(x) 함수는 x 라는 Key 에 대응되는 Value 를 리턴한다. 앞에서 살펴보았듯이 a.get('name')은 a['name']을 사용했을 때와 동일한 결과값을 리턴한다.

다만 다음 예제에서 볼 수 있듯이 a['nokey']처럼 딕셔너리에 존재하지 않는 키로 값을 가져오려고 할 경우 a['nokey'] 방식은 오류를 발생시키고 a.get('nokey') 방식은 None 을 리턴한다는 차이가 있다. 어떤것을 사용할지는 여러분의 선택이다.

여기에서 None 은 “거짓” 이라는 뜻이라고만 알아두자.

```
1 >>> a = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
2 >>> print(a.get('nokey'))
3 None
4 >>> print(a['nokey'])
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   KeyError: 'nokey'
```

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

```
1 >>> a.get('nokey', 'foo')
2 'foo'
```

딕셔너리 a 에는 'nokey' 에 해당하는 Key 가 없다. 따라서 디폴트 값인 'foo' 를 리턴한다.

해당 Key 가 딕셔너리 안에 있는지 조사하기 (in)

```
1 >>> a = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
2 >>> 'name' in a
3 True
4 >>> 'email' in a
5 False
```

'name' 문자열은 a 딕셔너리의 Key 중 하나이다. 따라서 'name' in a 를 호출하면 참 (True) 을 리턴한다. 반대로 'email' 은 a 딕셔너리 안에 존재하지 않는 Key 이므로 거짓 (False) 을 리턴한다.

02-6 집합 자료형

[추천 동영상 강의] : <https://www.youtube.com/watch?v=nsAtYHnovUA>

집합 자료형은 어떻게 만들까?

집합 (set) 은 파이썬 2.3 부터 지원하기 시작한 자료형으로, 집합에 관련된 것을 쉽게 처리하기 위해 만든 자료형이다.

집합 자료형은 다음과 같이 set 키워드를 사용해 만들 수 있다.

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

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

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

비어 있는 집합 자료형은 s = set() 로 만들 수 있다.

집합 자료형의 특징

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

- 중복을 허용하지 않는다.
- 순서가 없다 (Unordered).

중복을 허용하지 특징 때문에 `set` 은 자료형의 중복을 제거하기 위한 필터로 종종 사용된다.

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 02-5 에서 살펴본 딕셔너리와 비슷하다. 딕셔너리 역시 순서가 없는 자료형이라 인덱싱을 지원하지 않는다.

만약 `set` 자료형에 저장된 값을 인덱싱으로 접근하려면 다음과 같이 리스트나 튜플로 변환한후 해야 한다.

```
1 >>> s1 = set([1,2,3])
2 >>> l1 = list(s1)
3 >>> l1
4 [1, 2, 3]
5 >>> l1[0]
6 1
7 >>> t1 = tuple(s1)
8 >>> t1
9 (1, 2, 3)
10 >>> t1[0]
11 1
```

교집합, 합집합, 차집합 구하기

`set` 자료형을 정말 유용하게 사용하는 경우는 교집합, 합집합, 차집합을 구할 때이다.

우선 다음과 같이 2 개의 `set` 자료형을 만든 후 따라 해 보자. `s1` 은 1 부터 6 까지의 값을 가지게 되었고, `s2` 는 4 부터 9 까지의 값을 가지게 되었다.

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

1. 교집합

`s1` 과 `s2` 의 교집합을 구해 보자.

```
1 >>> s1 & s2
2 {4, 5, 6}
```

“&” 기호를 이용하면 교집합을 간단히 구할 수 있다.

또는 다음과 같이 `intersection` 함수를 사용해도 동일한 결과를 리턴한다.

```
1 >>> s1.intersection(s2)
2 {4, 5, 6}
```

`s2.intersection(s1)` 을 사용해도 결과는 같다.

2. 합집합

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

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

“|” 기호를 사용한 방법이다.

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

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

3. 차집합

차집합은 다음과 같이 구할 수 있다.

```
1 >>> s1 - s2
2 {1, 2, 3}
3 >>> s2 - s1
4 {8, 9, 7}
```

빼기 (-) 기호를 사용한 방법이다.

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

difference 함수를 사용해도 차집합을 구할 수 있다.

집합 자료형 관련 함수들

값 1 개 추가하기 (add)

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

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

값 여러 개 추가하기 (update)

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

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

특정 값 제거하기 (remove)

특정 값을 제거하고 싶을 때는 다음과 같이 하면 된다.

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

02-7 불 자료형

[추천 동영상 강의] : <https://www.youtube.com/watch?v=kuEJH1-8eo>

불 자료형이란?

불 (bool) 자료형이란 참 (True) 과 거짓 (False) 을 나타내는 자료형이다. 불 자료형은 다음 2 가지 값만을 가질 수 있다.

- True - 참
- False - 거짓

True 나 False 는 파이썬의 예약어로 true, false 와 같이 사용하지 말고 첫 문자를 항상 대문자로 사용해야 한다.

다음과 같이 변수 a 에는 True 를, 변수 b 에는 False 를 지정해 보자.

```
1 >>> a = True
2 >>> b = False
```

따옴표로 감싸지 않은 문자열을 변수에 지정해서 오류가 발생할 것 같지만 잘 실행된다. type 함수를 변수 a 와 b 에 사용하면 두 변수의 자료형이 bool 로 지정된 것을 확인할 수 있다.

```
1 >>> type(a)
2 <class 'bool'>
3 >>> type(b)
4 <class 'bool'>
```

type(x) 는 x 의 자료형을 확인하는 파이썬의 내장 함수이다.

불 자료형은 조건문의 리턴 값으로도 사용된다. 조건문에 대해서는 if 문에서 자세히 배우겠지만 잠시 살펴보고 넘어가자.

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


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

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

2 는 1 보다 크기 때문에 `2 > 1` 조건문은 `True` 를 리턴한다.

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

2 는 1 보다 작지 않기 때문에 `2 < 1` 조건문은 `False` 를 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
<code>"python"</code>	참
<code>""</code>	거짓
<code>[1, 2, 3]</code>	참
<code>[]</code>	거짓
<code>(1, 2, 3)</code>	참
<code>()</code>	거짓
<code>{'a': 1}</code>	참
<code>{}</code>	거짓
<code>1</code>	참
<code>0</code>	거짓
<code>None</code>	거짓

문자열, 리스트, 튜플, 딕셔너리 등의 값이 비어 있으면 (`"`, `[]`, `()`, `{}`) 거짓이 된다. 당연히 비어있지 않은

면 참이 된다. 숫자에서는 그 값이 0 일 때 거짓이 된다. 위 표를 보면 **None** 이 있는데, 이것에 대해서는 뒷 부분에서 배우니 아직은 신경 쓰지 말자. 그저 **None** 은 거짓을 뜻한다는 것만 알아두자.

다음 예를 보고 자료형의 참과 거짓이 프로그램에서 어떻게 쓰이는지 간단히 알아보자.

```
1 >>> a = [1, 2, 3, 4]
2 >>> while a:
3 ...     print(a.pop())
4 ...
5 4
6 3
7 2
8 1
```

먼저 `a = [1, 2, 3, 4]` 라는 리스트를 만들었다.

while 문은 03 장에서 자세히 다루겠지만 간단히 알아보면 다음과 같다. 조건문이 참인 동안 조건문 안에 있는 문장을 반복해서 수행한다.

```
1 while 조건문:
2     수행할 문장
```

즉 위 예를 보면 `a` 가 참인 경우에 `a.pop()` 을 계속 실행하여 출력하라는 의미이다. `a.pop()` 함수는 리스트 `a` 의 마지막 요소를 끄집어내는 함수이므로 리스트 안에 요소가 존재하는 한 (`a` 가 참인 동안) 마지막 요소를 계속해서 끄집어낼 것이다. 결국 더 이상 끄집어낼 것이 없으면 `a` 가 빈 리스트 (`[]`) 가 되어 거짓이 된다. 따라서 **while** 문에서 조건문이 거짓이 되므로 **while** 문을 빠져나가게 된다. 위에서 본 예는 파이썬 프로그래밍에서 매우 자주 사용하는 기법 중 하나이다.

위 예가 너무 복잡하다고 생각하는 독자는 다음 예를 보면 쉽게 이해될 것이다.

```
1 >>> if []:
2 ...     print("참")
3 ... else:
4 ...     print("거짓")
5 ...
6 거짓
```

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

if 문에 대해서는 03 장에서 자세히 다룬다.

`[]` 는 앞의 표에서 볼 수 있듯이 비어 있는 리스트이므로 거짓이다. 따라서 “거짓” 이란 문자열이 출력된다. **if** 문에 대해서 잘 모르는 독자라도 위 문장을 해석하는 데는 무리가 없을 것이다.

```
1 >>> if [1, 2, 3]:
2 ...     print("참")
3 ... else:
4 ...     print("거짓")
5 ...
```

6 참

위 코드를 해석해 보면 다음과 같다.

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

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

불 연산

자료형에 참과 거짓이 있음을 이제 알게 되었다. `bool` 함수를 사용하면 자료형의 참과 거짓을 보다 정확하게 식별할 수 있다.

다음 예제를 따라 해 보자.

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

‘python’ 문자열은 빈 문자열이 아니므로 `bool` 연산의 결과로 불 자료형인 `True`를 리턴한다.

```
1 >>> bool('')  
2 False
```

'' 문자열은 빈 문자열이므로 `bool` 연산의 결과로 불 자료형인 `False`를 리턴한다.

위에서 알아본 몇 가지 예제를 더 수행해 보자.

```
1 >>> bool([1,2,3])  
2 True  
3 >>> bool([])  
4 False  
5 >>> bool(0)  
6 False  
7 >>> bool(3)  
8 True
```

위에서 알아본 것과 동일한 참과 거짓에 대한 결과를 리턴하는 것을 확인할 수 있다.

지금까지 파이썬의 가장 기본이 되는 자료형인 숫자, 문자열, 리스트, 튜플, 딕셔너리, 집합, 불에 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 예제만 따라 하지 말고 직접 여러 가지 예들을 테스트해 보며 02-1~02-7의 자료형에 익숙해지기를 바란다.

02-8 자료형의 값을 저장하는 공간, 변수

[추천 동영상 강의] : <https://www.youtube.com/watch?v=k7J6dt-tsbl&feature=youtu.be>

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

변수는 어떻게 만들까?

우리는 앞에서 이미 변수를 사용해 왔다. 다음 예와 같은 **a, b, c** 를 변수라고 한다.

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

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

다른 프로그래밍 언어인 C 나 JAVA 에서는 변수를 만들 때 자료형의 타입을 직접 지정해야 한다. 하지만 파이썬은 변수에 저장된 값을 스스로 판단하여 자료형의 타입을 지정하기 때문에 더 편리하다.

```
1 변수 이름 = 변수에 저장할 값
```

변수란?

파이썬에서 사용하는 변수는 객체를 가리키는 것이라고도 말할 수 있다. 객체란 우리가 지금껏 보아 온 자료형의 데이터 (값) 와 같은 것을 의미하는 말이다 (객체에 대해서는 05-1 장에서 자세하게 공부한다).

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

만약 위 코드처럼 **a = [1, 2, 3]** 이라고 하면 **[1, 2, 3]** 값을 가지는 리스트 데이터 (객체) 가 자동으로 메모리에 생성되고 변수 **a** 는 **[1, 2, 3]** 리스트가 저장된 메모리의 주소를 가리키게 된다.

메모리란 컴퓨터가 프로그램에서 사용하는 데이터를 기억하는 공간이다.

a 변수가 가리키는 메모리의 주소는 다음과 같이 확인할 수 있다.

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

id 함수는 변수가 가리키고 있는 객체의 주소 값을 리턴하는 파이썬 내장 함수이다. 즉 여기에서 필자가 만든 변수 a 가 가리키는 [1, 2, 3] 리스트의 주소 값은 4303029896 임을 알 수 있다.

리스트를 복사하고자 할 때

여기에서는 리스트 자료형에서 가장 혼동하기 쉬운 “복사”에 대해 설명하겠다. 다음 예를 통해 알아보자.

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

b 변수에 a 변수를 대입하면 어떻게 될까? b 와 a 는 같은 걸까 다른 걸까? 결론부터 말하면 b 는 a 와 완전히 동일하다고 할 수 있다. 다만 [1, 2, 3] 이라는 리스트 객체를 참조하는 변수가 a 변수 1 개에서 b 변수가 추가되어 2 개로 늘어났다는 차이만 있을 뿐이다.

id 함수를 사용하면 이러한 사실을 확인할 수 있다.

```
1 >>> id(a)
2 4303029896
3 >>> id(b)
4 4303029896
```

id(a)의 값이 id(b)의 값과 동일함을 확인할 수 있다. 즉 a 가 가리키는 대상과 b 가 가리키는 대상이 동일하다는 것을 알 수 있다. 동일한 객체를 가리키고 있는지에 대해서 판단하는 파이썬 명령어 is 를 다음과 같이 실행해도 역시 참 (True) 을 돌려준다.

```
1 >>> a is b # a와 b가 가리키는 객체는 동일 한 가?
2 True
```

이제 다음 예를 계속해서 수행해 보자.

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

a 리스트의 두 번째 요소를 값 4로 바꾸었더니 a만 바뀌는 것이 아니라 b도 똑같이 바뀌었다. 그 이유는 앞에서 살펴본 것처럼 a, b 모두 동일한 리스트를 가리키고 있기 때문이다.

그렇다면 b 변수를 생성할 때 a 변수의 값을 가져오면서 a와는 다른 주소를 가리키도록 만들수는 없을까? 다음 2 가지 방법이 있다.

1. [:] 이용

첫 번째 방법으로는 다음과 같이 리스트 전체를 가리키는 [:] 을 사용해서 복사하는 것이다.

```
1 >>> a = [1, 2, 3]
2 >>> b = a[:]
3 >>> a[1] = 4
4 >>> a
5 [1, 4, 3]
6 >>> b
7 [1, 2, 3]
```

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

2. copy 모듈 이용

두 번째는 copy 모듈을 사용하는 방법이다. 다음 예를 보면 `from copy import copy` 라는 처음 보는 형태의 문장이 나오는데, 이것은 뒤에 설명할 파이썬 모듈 부분에서 자세히 다룬다. 여기에서는 단순히 copy 함수를 쓰기 위해서 사용하는 것이라고만 알아두자.

```
1 >>> from copy import copy
2 >>> a = [1, 2, 3]
3 >>> b = copy(a)
```

위 예에서 `b = copy(a)`는 `b = a[:]`과 동일하다.

두 변수의 값은 같지만 서로 다른 객체를 가리키고 있는지 다음과 같이 확인해 보자.

```
1 >>> b is a
2 False
```

위 예에서 `b is a`가 `False`를 돌려주므로 b와 a가 가리키는 객체는 서로 다르다는 것을 알 수 있다.

다음처럼 리스트 자료형의 자체 함수인 copy 함수를 사용해도 copy 모듈을 사용하는 것과 동일하다.

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

변수를 만드는 여러 가지 방법

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

위 예문처럼 튜플로 a, b 에 값을 대입할 수 있다. 이 방법은 다음 예문과 완전히 동일하다.

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

튜플 부분에서도 언급했지만 튜플은 괄호를 생략해도 된다.

다음처럼 리스트로 변수를 만들 수도 있다.

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

또한 여러 개의 변수에 같은 값을 대입할 수도 있다.

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

파이썬에서는 위 방법을 사용하여 두 변수의 값을 아주 간단히 바꿀 수 있다.

```
1 >>> a = 3
2 >>> b = 5
3 >>> a, b = b, a
4 >>> a
5 5
6 >>> b
7 3
```

처음에 a 에 값 3, b 에는 값 5 가 대입되어 있었지만 a, b = b, a 문장을 수행한 후에는 그 값이 서로 바뀌었음을 확인할 수 있다.

02 장 연습문제

Q1. 평균 점수 구하기

홍길동 씨의 과목별 점수는 다음과 같다. 홍길동 씨의 평균 점수를 구해 보자.

과목	점수
국어	80
영어	75
수학	55

Q2. 홀수 짝수 판별

자연수 13 이 홀수인지 짝수인지 판별할 수 있는 방법에 대해 말해 보자.

Q3. 주민등록번호 나누기

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

문자열 슬라이싱 기법을 사용해 보자.

Q4. 주민등록번호 인덱싱

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


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

문자열 인덱싱을 사용해 보자.

Q5. 문자열 바꾸기

다음과 같은 문자열 a:b:c:d 가 있다. 문자열의 `replace` 함수를 사용하여 a#b#c#d 로 바꿔서 출력해 보자.

```
1 >>> a = "a:b:c:d"
```

Q6. 리스트 역순 정렬하기

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

리스트의 내장 함수를 사용해 보자.

Q7. 리스트를 문자열로 만들기

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

문자열의 `join` 함수를 사용하면 리스트를 문자열로 쉽게 만들 수 있다.

Q8. 튜플 더하기

(1,2,3) 튜플에 값 4 를 추가하여 (1,2,3,4) 를 만들어 출력해 보자.

더하기 (+) 를 사용해 보자.

Q9. 딕셔너리의 키

다음과 같은 딕셔너리 a 가 있다.

```
1 >>> a = dict()
2 >>> a
3 {}
```

다음 중 오류가 발생하는 경우를 고르고, 그 이유를 설명해 보자.

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

Q10. 딕셔너리 POP

딕셔너리 `a` 에서 'B' 에 해당되는 값을 추출해 보자.

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

딕셔너리의 `pop` 함수를 사용해 보자.

Q11. 리스트에서 중복 제거하기

`a` 리스트에서 중복 숫자를 제거해 보자.

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

집합 자료형의 요소값이 중복될 수 없다는 특징을 사용해 보자.

Q12. 파이썬 변수

파이썬은 다음처럼 동일한 값에 여러 개의 변수를 선언할 수 있다. 다음과 같이 `a, b` 변수를 선언한 후 `a` 의 두 번째 요소값을 변경하면 `b` 값은 어떻게 될까? 그리고 이런 결과가 오는 이유에 대해 설명해 보자.

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

03 장 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 등은 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 if 문

[추천 동영상 강의] : <https://youtube.com/watch?v=B5WF6k70svo>

if 문은 왜 필요할까?

다음과 같은 상상을 해보자.

“돈이 있으면 택시를 타고, 돈이 없으면 걸어 간다.”

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

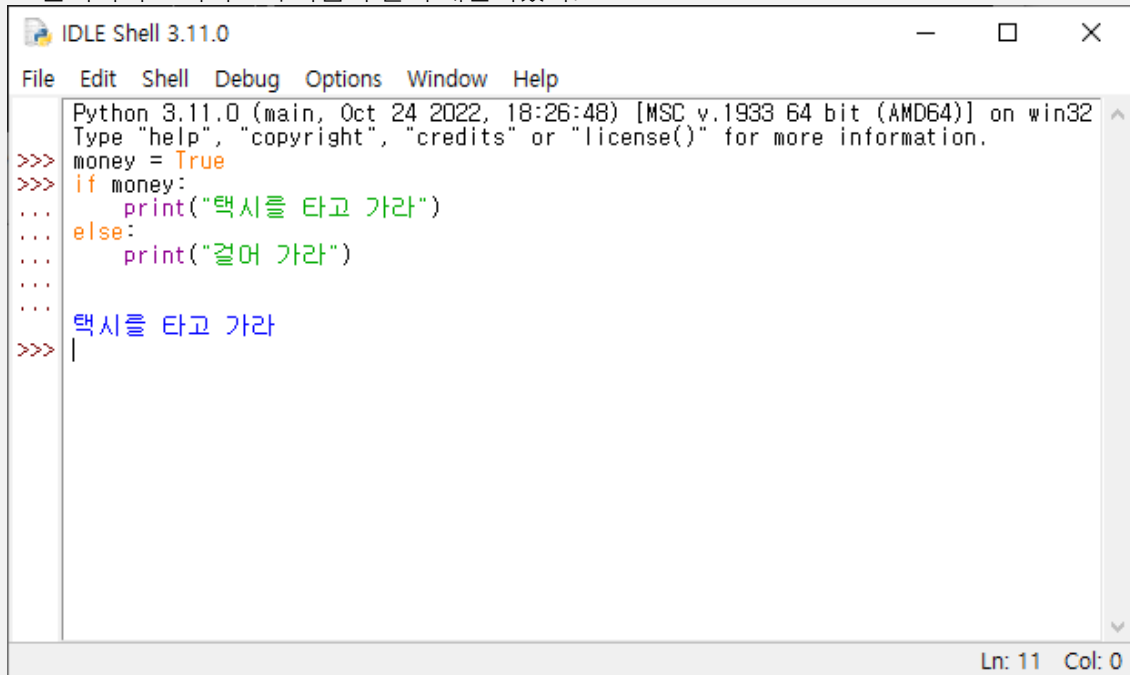
위와 같은 상황을 파이썬에서는 다음과 같이 표현할 수 있다.

```
1 >>> money = True
2 >>> if money:
3 ...     print("택시를 타고 가라")
4 ... else:
5 ...     print("걸어 가라")
6 ...
7 택시를 타고 가라
```

money에 True를 대입했으므로 money는 참이다. 따라서 if문 다음 문장이 수행되어 '택시를 타고 가라'가 출력된다.

IDLE 셸 주의사항

>>> 프롬프트가 보이는 이 책의 예제는 반드시 **IDLE 셸**이 아닌 **파이썬 셸**에서 실행하도록 하자. IDLE 셸에서는 ‘...’과 같은 줄 바꿈 표시가 보이지 않아 들여쓰기 오류를 범할 가능성이 크다. 하지만 파이썬 3.10 부터는 IDLE 셸의 >>> 프롬프트와 ...과 같은 줄 바꿈 표시가 좌측의 영역으로 분리되어 표시되도록 다음과 같이 개선되었다.



```

Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> money = True
>>> if money:
...     print("택시를 타고 가라")
... else:
...     print("걸어 가라")
...
... 택시를 타고 가라
>>> |
  
```

따라서 파이썬 3.10 이상의 버전을 사용한다면 이 책의 예제를 IDLE 셸에서 실행해도 괜찮다.

if 문의 기본 구조

다음은 if와 else를 사용한 조건문의 기본 구조이다.

```

1  if 조건문:
2      수행할 문장1
3      수행할 문장2
4      ...
5  else:
6      수행할 문장A
7      수행할 문장B
8      ...
  
```

조건문을 테스트해서 참이면 if 문 바로 다음 문장 (if 블록) 들을 수행하고, 조건문이 거짓이면 else 문 다음 문장 (else 블록) 들을 수행하게 된다. 그러므로 else 문은 if 문 없이 독립적으로 사용할 수 없다.

들여쓰기

if 문을 만들 때는 if 조건문: 바로 아래 문장부터 if 문에 속하는 모든 문장에 들여쓰기 (indentation) 를 해주어야 한다. 아래에서 보는 것과 같이 조건문이 참일 경우 “수행할 문장 1” 을 들여쓰기했고 “수행할 문장 2” 와 “수행할 문장 3” 도 들여쓰기 해 주었다. 다른 프로그래밍 언어를 사용해 온 사람들은 파이썬에서 “수행할 문장” 을 들여쓰기하는 것을 무시하는 경우가 많으니 더 주의해야 한다.

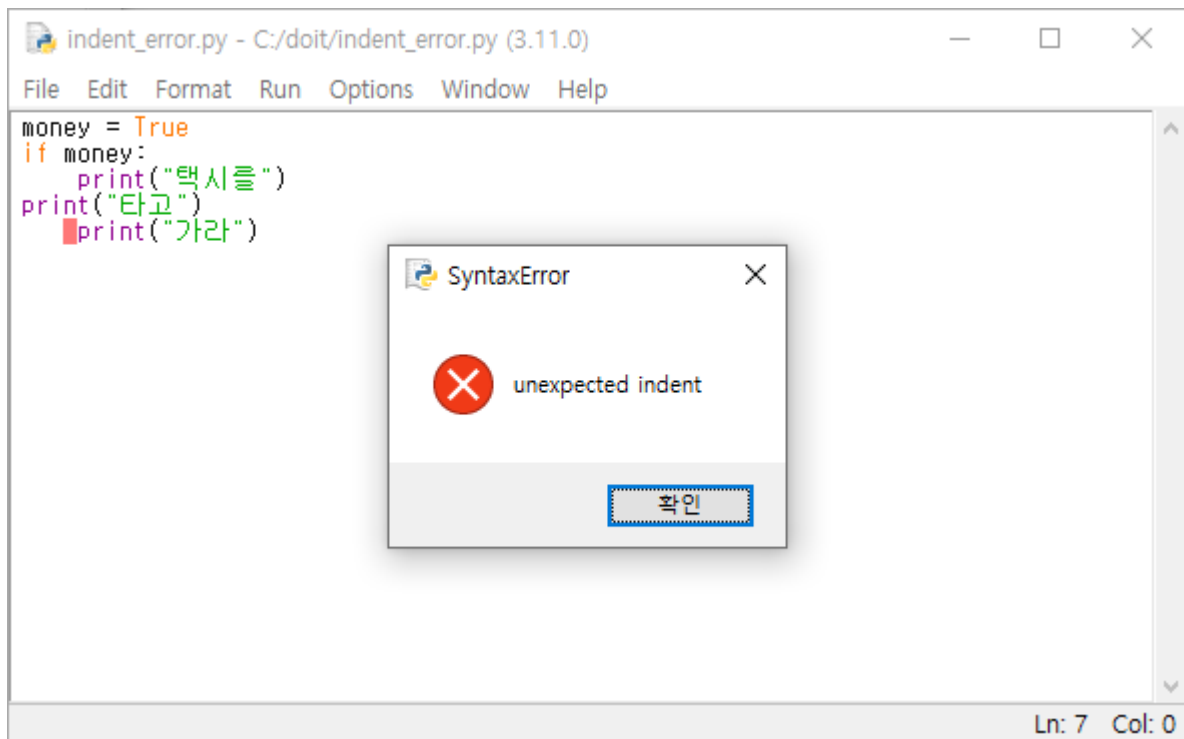
```
1 if 조건문:
2     수행할 문장1
3     수행할 문장2
4     수행할 문장3
```

다음처럼 작성하면 오류가 발생한다. “수행할 문장 2” 를 들여쓰기하지 않았기 때문이다.

```
1 if 조건문:
2     수행할 문장1
3 수행할 문장2
4     수행할 문장3
```

IDLE 에디터에서 다음과 같이 작성하여 실행해 보자.

```
1 money = True
2 if money:
3     print("택시를")
4     print("타고")
5     print("가라")
```



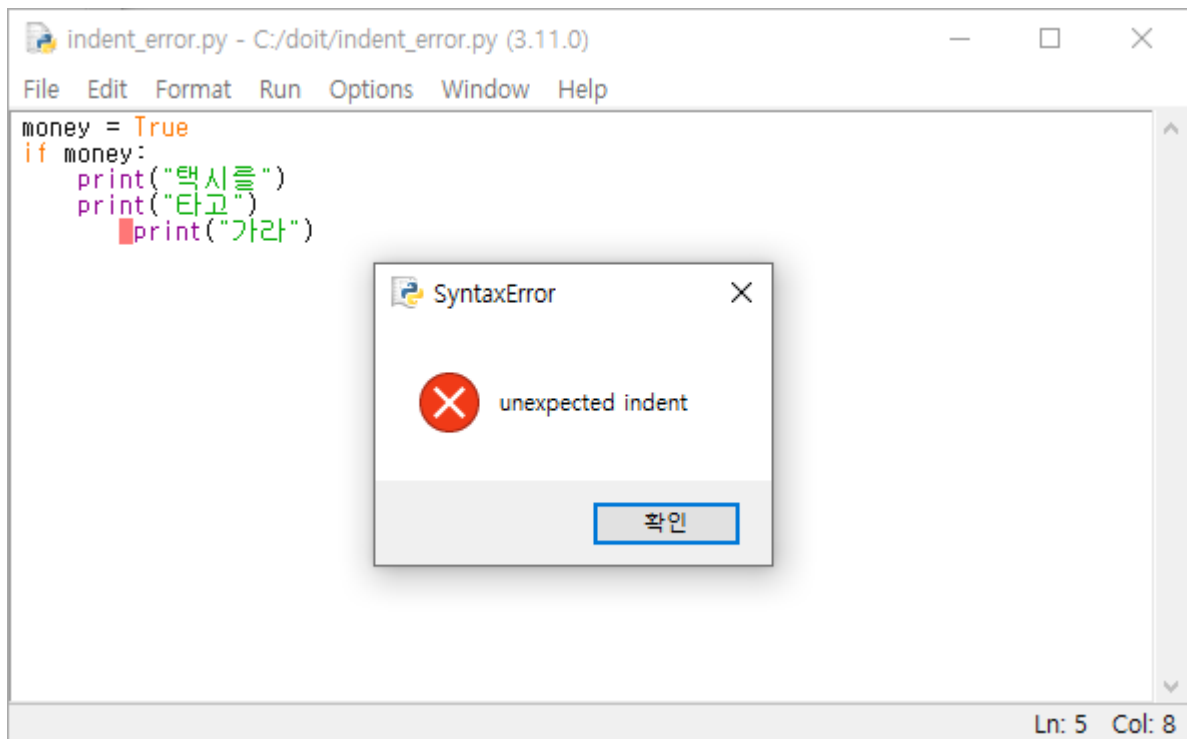
들여쓰기 오류가 발생하는 것을 확인할 수 있다.

다음과 같은 경우에도 오류가 발생한다. “수행할 문장 3”을 들여쓰기했지만 “수행할 문장 1”이나 “수행할 문장 2”와 들여쓰기의 깊이가 다르다. 즉 들여쓰기는 언제나 같은 깊이로 해야 한다.

```
1  if 조건문:
2      수행할 문장1
3      수행할 문장2
4      수행할 문장3
```

IDLE 에디터에서 다음과 같이 작성하여 실행해 보자.

```
1  money = True
2  if money:
3      print("택시를")
4      print("타고")
5      print("가라")
```



마찬가지로 들여쓰기 오류가 발생하는 것을 확인할 수 있다.

그렇다면 들여쓰기는 공백 (Spacebar) 으로 하는 것이 좋을까? 아니면 탭 (Tab) 으로 하는 것이 좋을 까? 이에 대한 논란은 파이썬을 사용하는 사람들 사이에서 아직도 계속되고 있다. 탭으로 하자는 쪽과 공백으로 하자는 쪽 모두가 동의하는 내용은 단 하나, 2 가지를 혼용해서 쓰지는 말자는 것이다. 공백으로 할 거면 항상 공백으로 통일하고, 탭으로 할 거면 항상 탭으로 통일해서 사용하자는 말이다. 탭이나 공백은 프로그램 소스에서 눈으로 보이는 것이 아니기 때문에 혼용해서 쓰면 오류의 원인이 되니 주의하자.

요즘 파이썬 커뮤니티에서는 들여쓰기를 할 때 공백 (Spacebar) 4 개를 사용하는 것을 권장한다. 또한 파이썬 에디터는 대부분 탭 (tab) 키로 들여쓰기를 하더라도 탭 문자를 공백 문자 4 개로 자동 변환하는 기능을 갖추고 있다.

[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 물어봐야 할 것이다. 앞으로 배울 while 이나 for, def, class 도 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 if 문에 속한 문장들을 { } 기호로 감싸지만 파이썬에서는 들여쓰기로 해결한다는 점을 기억하자.

조건문이란 무엇인가?

if 조건문에서 “조건문”이란 참과 거짓을 판단하는 문장을 말한다.

앞에서 살펴본 택시 예제에서 조건문은 money 가 된다.

```
1 >>> money = True
2 >>> if money:
```

money 는 True 이기 때문에 조건이 참이 되어 if 문 다음 문장을 수행한다.

비교연산자

이번에는 조건문에 비교연산자 (<, >, ==, !=, >=, <=) 를 쓰는 방법에 대해 알아보자.

다음 표는 비교 연산자를 잘 설명해 준다.

비교연산자	설명
$x < y$	x 가 y 보다 작다
$x > y$	x 가 y 보다 크다
$x == y$	x 와 y 가 같다
$x != y$	x 와 y 가 같지 않다
$x >= y$	x 가 y 보다 크거나 같다

비교연산자	설명
-------	----

<code>x <= y</code>	x 가 y 보다 작거나 같다
------------------------	-----------------

이제 위 연산자를 어떻게 사용하는지 알아보자.

```
1 >>> x = 3
2 >>> y = 2
3 >>> x > y
4 True
5 >>>
```

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

```
1 >>> x < y
2 False
```

위 조건문은 거짓이기 때문에 `False`를 리턴한다.

```
1 >>> x == y
2 False
```

x와 y는 같지 않다. 따라서 위 조건문은 거짓이다.

```
1 >>> x != y
2 True
```

x와 y는 같지 않다. 따라서 위 조건문은 참이다.

앞에서 살펴본 택시 예제를 다음처럼 바꾸려면 어떻게 해야 할까?

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

위 상황은 다음처럼 프로그래밍할 수 있다.

```
1 >>> money = 2000
2 >>> if money >= 3000:
3 ...     print("택시를 타고 가라")
4 ... else:
5 ...     print("걸어 가라")
6 ...
7 걸어 가라
8 >>>
```

`money >= 3000` 조건문이 거짓이 되기 때문에 `else` 문 다음 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>	x 와 y 둘중에 하나만 참이어도 참이다
<code>x and y</code>	x 와 y 모두 참이어야 참이다
<code>not x</code>	x 가 거짓이면 참이다

다음 예를 통해 **or** 연산자의 사용법을 알아보자.

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

```
1 >>> money = 2000
2 >>> card = True
3 >>> if money >= 3000 or card:
4 ...     print("택시를 타고 가라")
5 ... else:
6 ...     print("걸어 가라")
7 ...
8 택시를 타고 가라
9 >>>
```

`money` 는 2000 이지만 `card` 가 `True` 이기 때문에 `money >= 3000 or card` 조건문이 참이 된다. 따라서 `if` 문에 속한 “택시를 타고 가라” 문장이 출력된다.

in, not in

더 나아가 파이썬은 다른 프로그래밍 언어에서 쉽게 볼 수 없는 재미있는 조건문을 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>

영어 단어 `in` 의 뜻이 “~ 안에” 라는 것을 생각해 보면 다음 예가 쉽게 이해될 것이다.

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

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

다음은 튜플과 문자열에 `in` 과 `not in` 을 적용한 예이다. 각각의 결과가 나온 이유는 쉽게 유추할 수 있다.

```
1 >>> 'a' in ('a', 'b', 'c')
2 True
3 >>> 'j' not in 'python'
4 True
```

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

“만약 주머니에 돈이 있으면 택시를 타고, 없으면 걸어 가라.”

```
1 >>> pocket = ['paper', 'cellphone', 'money']
2 >>> if 'money' in pocket:
3 ...     print("택시를 타고 가라")
4 ... else:
5 ...     print("걸어 가라")
6 ...
7 택시를 타고 가라
8 >>>
```

[`'paper'`, `'cellphone'`, `'money'`] 리스트 안에 `'money'` 가 있으므로 `'money' in pocket` 은 참이 된다. 따라서 `if` 문에 속한 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.

다음 예를 보자.

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

이럴 때 사용하는 것이 바로 `pass` 이다. 위 예를 `pass` 를 적용해서 구현해 보자.

```
1 >>> pocket = ['paper', 'money', 'cellphone']
2 >>> if 'money' in pocket:
3 ...     pass
4 ... else:
5 ...     print("카드를 꺼내라")
6 ...
```

`pocket` 리스트 안에 `money` 문자열이 있기 때문에 `if` 문 다음 문장인 `pass` 가 수행되고 아무 결과값도 보여 주지 않는다.

다양한 조건을 판단하는 elif

`if` 와 `else` 만으로는 다양한 조건을 판단하기 어렵다. 다음 예를 보더라도 `if` 와 `else` 만으로는 조건을 판단하는 데 어려움을 겪게 된다.

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

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

`if` 와 `else` 만으로 위 문장을 표현하려면 다음과 같이 할 수 있다.

```
1 >>> pocket = ['paper', 'handphone']
2 >>> card = True
3 >>> if 'money' in pocket:
4 ...     print("택시를 타고 가라")
5 ... else:
6 ...     if card:
7 ...         print("택시를 타고 가라")
8 ...     else:
9 ...         print("걸어 가라")
10 ...
11 택시를 타고 가라
12 >>>
```

언뜻 보기에 이해하기 어렵고 산만한 느낌이 든다. 이런 복잡함을 해결하기 위해 파이썬에서는 다중 조건 판단을 가능하게 하는 **elif** 를 사용한다.

위 예를 **elif** 를 사용하면 다음과 같이 바꿀 수 있다.

```
1 >>> pocket = ['paper', 'cellphone']
2 >>> card = True
3 >>> if 'money' in pocket:
4 ...     print("택시를 타고 가라")
5 ... elif card:
6 ...     print("택시를 타고 가라")
7 ... else:
8 ...     print("걸어 가라")
9 ...
10 택시를 타고 가라
```

즉 **elif** 는 이전 조건문이 거짓일 때 수행된다. **if**, **elif**, **else** 를 모두 사용할 때 기본 구조는 다음과 같다.

```
1 if <조건문>:
2     <수행할 문장1>
3     <수행할 문장2>
4     ...
5 elif <조건문>:
6     <수행할 문장1>
7     <수행할 문장2>
8     ...
9 elif <조건문>:
10    <수행할 문장1>
11    <수행할 문장2>
12    ...
13 ...
14 else:
15    <수행할 문장1>
16    <수행할 문장2>
17    ...
```

위에서 볼 수 있듯이 **elif** 는 개수에 제한 없이 사용할 수 있다.

if 문을 한 줄로 작성하기

앞의 `pass` 를 사용한 예를 보면 `if` 문 다음에 수행할 문장이 한 줄이고, `else` 문 다음에 수행할 문장도 한 줄밖에 되지 않는다.

```
1 >>> if 'money' in pocket:
2 ...     pass
3 ... else:
4 ...     print("카드를 꺼내라")
5 ...
```

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

```
1 >>> pocket = ['paper', 'money', 'cellphone']
2 >>> if 'money' in pocket: pass
3 ... else: print("카드를 꺼내라")
4 ...
```

`if` 문 다음 수행할 문장을 콜론(:) 뒤에 바로 적어 주었다. `else` 문 역시 마찬가지이다.

조건부 표현식

다음과 같은 코드를 보자.

```
1 if score >= 60:
2     message = "success"
3 else:
4     message = "failure"
```

위 코드는 `score` 가 60 이상일 경우 `message` 에 문자열 “success” 를, 아닐 경우에는 “failure” 를 대입하는 코드이다.

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

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

조건부 표현식은 다음과 같이 정의한다.

변수 = 조건문이_참인_경우의_값 if 조건문 else 조건문이_거짓인_경우의_값

조건부 표현식은 가독성에 유리하고 한 줄로 작성할 수 있어 활용성이 좋다.

03-2 while 문

while 문의 기본 구조

반복해서 문장을 수행해야 할 경우 **while** 문을 사용한다. 그래서 **while** 문을 반복문이라고도 부른다.

다음은 **while** 문의 기본 구조이다.

```
1 while <조 건 문>:
2     <수 행 할 문 장 1>
3     <수 행 할 문 장 2>
4     <수 행 할 문 장 3>
5     ...
```

while 문은 조건문이 참인 동안에 **while** 문에 속한 문장들이 반복해서 수행된다.

“열 번 찍어 안 넘어가는 나무 없다” 는 속담을 파이썬 프로그램으로 만든다면 다음과 같이 될 것이다.

```
1 >>> treeHit = 0
2 >>> while treeHit < 10:
3 ...     treeHit = treeHit + 1
4 ...     print("나 무 를 %d 번 찍 었 습 니 다 ." % treeHit)
5 ...     if treeHit == 10:
6 ...         print("나 무 넘 어 갑 니 다 .")
7 ...
8 나 무 를 1 번 찍 었 습 니 다 .
9 나 무 를 2 번 찍 었 습 니 다 .
10 나 무 를 3 번 찍 었 습 니 다 .
11 나 무 를 4 번 찍 었 습 니 다 .
12 나 무 를 5 번 찍 었 습 니 다 .
13 나 무 를 6 번 찍 었 습 니 다 .
14 나 무 를 7 번 찍 었 습 니 다 .
15 나 무 를 8 번 찍 었 습 니 다 .
16 나 무 를 9 번 찍 었 습 니 다 .
17 나 무 를 10 번 찍 었 습 니 다 .
18 나 무 넘 어 갑 니 다 .
```

위 예에서 **while** 문의 조건문은 `treeHit < 10`이다. 즉 `treeHit` 가 10 보다 작은 동안에 **while** 문에 포함된 문장들을 계속 수행한다. **while** 문 안의 문장을 보면 제일 먼저 `treeHit = treeHit + 1`로 `treeHit` 값이 계속 1 씩 증가한다. 그리고 나무를 `treeHit` 번만큼 찍었음을 알리는 문장을 출력하고 `treeHit`

가 10 이 되면 “나무 넘어갑니다.” 라는 문장을 출력한다. 그리고 나면 `treeHit < 10` 조건문이 거짓이 되므로 `while` 문을 빠져나가게 된다.

`treeHit = treeHit + 1` 은 프로그래밍을 할 때 매우 자주 사용하는 기법이다. `treeHit` 값을 1 만큼씩 증가시킬 목적으로 사용하며, `treeHit += 1` 처럼 사용해도 된다.

다음은 `while` 문이 반복되는 과정을 순서대로 정리한 표이다. 이렇게 긴 과정을 소스 코드 단 5 줄로 만들 수 있다니 놀랍지 않은가?

treeHit	조건문	조건판단	수행하는 문장	while 문
0	0 < 10	참	나무를 1 번 찍었습니다.	반복
1	1 < 10	참	나무를 2 번 찍었습니다.	반복
2	2 < 10	참	나무를 3 번 찍었습니다.	반복
3	3 < 10	참	나무를 4 번 찍었습니다.	반복
4	4 < 10	참	나무를 5 번 찍었습니다.	반복
5	5 < 10	참	나무를 6 번 찍었습니다.	반복
6	6 < 10	참	나무를 7 번 찍었습니다.	반복
7	7 < 10	참	나무를 8 번 찍었습니다.	반복
8	8 < 10	참	나무를 9 번 찍었습니다.	반복
9	9 < 10	참	나무를 10 번 찍었습니다. 나무 넘어갑니다.	반복
10	10 < 10	거짓		종료

while 문 만들기

이번에는 여러 가지 선택지 중 하나를 선택해서 입력받는 예제를 만들어 보자. 먼저 다음과 같이 여러 줄 짜리 문자열인 `prompt` 를 만들자.

```
1 >>> prompt = """
2 ... 1. Add
3 ... 2. Del
4 ... 3. List
5 ... 4. Quit
6 ...
7 ... Enter number: """
8 >>>
```

이어서 `number` 변수에 0 을 먼저 대입한다. 이렇게 변수를 먼저 설정해 놓지 않으면 다음에 나올 `while` 문의 조건문인 `number != 4`에서 변수가 존재하지 않는다는 오류가 발생한다.

```
1 >>> number = 0
2 >>> while number != 4:
3 ...     print(prompt)
4 ...     number = int(input())
5 ...
```

```
1 1. Add
2 2. Del
3 3. List
4 4. Quit
5
6 Enter number:
```

`while` 문을 보면 `number` 가 4 가 아닌 동안 `prompt` 를 출력하고 사용자로부터 번호를 입력받는다. 다음 결과 화면처럼 사용자가 값 4 를 입력하지 않으면 계속해서 `prompt` 를 출력한다.

여기에서 `number = int(input())`는 사용자의 숫자 입력을 받아들이는 것이라고만 알아두자. `int` 나 `input` 함수에 대한 내용은 뒤의 내장 함수 부분에서 자세하게 다룬다.

```
1 Enter number:
2 1
3
4 1. Add
5 2. Del
6 3. List
7 4. Quit
```

4 를 입력하면 조건문이 거짓이 되어 `while` 문을 빠져나가게 된다.

```
1 Enter number:
2 4
3 >>>
```

while 문 강제로 빠져나가기

`while` 문은 조건문이 참인 동안 계속해서 `while` 문 안의 내용을 반복적으로 수행한다. 하지만 강제로 `while` 문을 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” 문구를 사용자에게 보여줘야 한다. 이렇게 판매를 강제로 멈추게 하는 것이 바로 `break` 문이다.



다음 예는 커피 자판기 이야기를 파이썬 프로그램으로 표현해 본 것이다.

```

1 >>> coffee = 10
2 >>> money = 300
3 >>> while money:
4 ...     print("돈을 받았으니 커피를 줍니다.")
5 ...     coffee = coffee - 1
6 ...     print("남은 커피의 양은 %d개입니다." % coffee)
7 ...     if coffee == 0:
8 ...         print("커피가 다 떨어졌습니다. 판매를 중지합니다.")
9 ...         break
10 ...

```

`money` 가 300 으로 고정되어 있으므로 `while money:`에서 조건문인 `money` 는 0 이 아니기 때문에 항상 참이다. 따라서 무한히 반복되는 무한 루프를 돌게 된다. 그리고 `while` 문의 내용을 한 번 수행할 때마다 `coffee = coffee - 1`에 의해서 `coffee` 의 개수가 1 개씩 줄어든다. 만약 `coffee` 가 0 이 되면 `if coffee == 0:` 문장에서 `coffee == 0:`이 참이 되므로 `if` 문 다음 문장 “커피가 다 떨어졌습니다. 판매를 중지합니다.” 가 수행되고 `break` 문이 호출되어 `while` 문을 빠져나가게 된다.

하지만 실제 자판기는 위 예처럼 작동하지는 않을 것이다. 다음은 자판기의 실제 작동 과정과 비슷하게 만들어본 예이다. 이해가 안 되더라도 걱정하지 말자. 다음 예는 조금 복잡하니까 대화형 인터프리터를 사용하지 말고 IDLE 에디터를 사용해서 작성해 보자.

대화형 인터프리터가 아닌 IDLE 에디터에서 소스를 입력하는 예제에서는 실제 실습 환경을 반영해 `>>>` (프롬프트) 를 표시하지 않았다. 앞으로 어떤 창에 소스를 입력해야 할지 헷갈릴 때는 이를 참고하자.

```

1 # coffee.py
2

```

```

3 coffee = 10
4 while True:
5     money = int(input("돈을 넣어 주세요: "))
6     if money == 300:
7         print("커피를 줍니다.")
8         coffee = coffee - 1
9     elif money > 300:
10        print("거스름돈 %d를 주고 커피를 줍니다." % (money - 300))
11        coffee = coffee - 1
12    else:
13        print("돈을 다시 돌려주고 커피를 주지 않습니다.")
14        print("남은 커피의 양은 %d개입니다." % coffee)
15    if coffee == 0:
16        print("커피가 다 떨어졌습니다. 판매를 중지 합니다.")
17        break

```

위 프로그램 소스를 따로 설명하지는 않겠다. 여러분이 소스를 입력하면서 무슨 내용인지 이해할 수 있다면 지금껏 배운 if 문이나 while 문을 이해했다고 보면 된다. 만약 `money = int(input("돈을 넣어 주세요: "))` 문장이 이해되지 않는다면 이 문장은 사용자로부터 값을 입력받는 부분이고 입력받은 숫자를 `money` 변수에 대입하는 것이라고만 알아두자.

이제 `coffee.py` 파일을 저장한 후 명령 프롬프트 창을 열어 프로그램을 직접 실행해 보자.

프로그램 소스를 에디터로 작성해서 실행시키는 방법이 기억나지 않는다면 01-6 장을 참고하자.

다음과 같은 입력 란이 나타난다.

```

1 C:\doit>python coffee.py
2 돈을 넣어 주세요:

```

입력 란에 여러 숫자를 입력해 보면서 결과를 확인하자.

```

1 돈을 넣어 주세요: 500
2 거스름돈 200를 주고 커피를 줍니다.
3 돈을 넣어 주세요: 300
4 커피를 줍니다.
5 돈을 넣어 주세요: 100
6 돈을 다시 돌려주고 커피를 주지 않습니다.
7 남은 커피의 양은 8개입니다.
8 돈을 넣어 주세요:

```

while 문의 맨 처음으로 돌아가기

while 문 안의 문장을 수행할 때 입력 조건을 검사해서 조건에 맞지 않으면 while 문을 빠져나간다. 그런데 프로그래밍을 하다 보면 while 문을 빠져나가지 않고 while 문의 맨 처음(조건문)으로 다시 돌아가게 만들고 싶은 경우가 생기게 된다. 이때 사용하는 것이 바로 `continue` 문이다.

1 부터 10 까지의 숫자 중에서 홀수만 출력하는 것을 while 문을 사용해서 작성한다고 생각해 보자. 어떤 방법이 좋을까?

```
1 >>> a = 0
2 >>> while a < 10:
3 ...     a = a + 1
4 ...     if a % 2 == 0: continue
5 ...     print(a)
6 ...
7 1
8 3
9 5
10 7
11 9
```

위 예는 1 부터 10 까지의 숫자 중 홀수만 출력하는 예이다. a 가 10 보다 작은 동안 a 는 1 만큼씩 계속 증가한다. **if a % 2 == 0** (a 를 2 로 나누었을 때 나머지가 0 인 경우) 이 참이 되는 경우는 a 가 짝수일 때이다. 즉 a 가 짝수이면 **continue** 문장을 수행한다. 이 **continue** 문은 **while** 문의 맨 처음 (조건문: **a<10**) 으로 돌아가게 하는 명령어이다. 따라서 위 예에서 a 가 짝수이면 **print(a)** 문장은 수행되지 않을 것이다.

무한 루프

이번에는 무한 루프 (Loop) 에 대해 알아보자. 무한 루프란 무한히 반복한다는 의미이다. 우리가 사용하는 일반 프로그램 중에서 무한 루프 개념을 사용하지 않는 프로그램은 거의 없다. 그만큼 자주 사용하는 뜻이다.

파이썬에서 무한 루프는 **while** 문으로 구현할 수 있다. 다음은 무한 루프의 기본 형태이다.

```
1 while True:
2     수행할 문장1
3     수행할 문장2
4     ...
```

while 문의 조건문이 **True** 이므로 항상 참이 된다. 따라서 **while** 문 안에 있는 문장들은 무한하게 수행될 것이다.

다음의 무한 루프 예이다.

```
1 >>> while True:
2 ...     print("Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.")
3 ...
4 Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
5 Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
6 Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
7 ....
```

위 문장은 영원히 출력된다. 하지만 이 예처럼 아무 의미 없이 무한 루프를 돌리는 경우는 거의 없을 것이다. [Ctrl+C]를 눌러 빠져나가자.

03-3 for 문

파이썬의 직관적인 특징을 가장 잘 보여주는 것이 바로 이 **for** 문이다. **while** 문과 비슷한 반복문인 **for** 문은 문장 구조가 한눈에 쏙 들어온다는 장점이 있다. **for** 문을 잘 사용하면 프로그래밍이 즐거워질 것이다.

for 문의 기본 구조

for 문의 기본 구조는 다음과 같다.

```
1 for 변수 in 리스트(또는 튜플, 문자열):
2     수행할 문장1
3     수행할 문장2
4     ...
```

리스트나 튜플, 문자열의 첫 번째 요소부터 마지막 요소까지 차례로 변수에 대입되어 “수행할 문장 1”, “수행할 문장 2” 등이 수행된다.

예제를 통해 for 문 이해하기

for 문은 예제를 통해서 살펴보는 것이 가장 알기 쉽다. 다음 예제를 직접 입력해 보자.

1. 전형적인 for 문

```
1 >>> test_list = ['one', 'two', 'three']
2 >>> for i in test_list:
3     ...     print(i)
4     ...
5 one
6 two
7 three
```

`['one', 'two', 'three']` 리스트의 첫 번째 요소인 'one' 이 먼저 `i` 변수에 대입된 후 `print(i)` 문장을 수행한다. 다음에 두 번째 요소 'two' 가 `i` 변수에 대입된 후 `print(i)` 문장을 수행하고 리스트의 마지막 요소까지 이것을 반복한다.

2. 다양한 for 문의 사용

```
1 >>> a = [(1,2), (3,4), (5,6)]
2 >>> for (first, last) in a:
3 ...     print(first + last)
4 ...
5 3
6 7
7 11
```

위 예는 `a` 리스트의 요솟값이 튜플이기 때문에 각각의 요소가 자동으로 `(first, last)` 변수에 대입된다.

3. for 문의 응용

for 문의 쓰임새를 알기 위해 다음과 같은 문제를 생각해 보자.

```
1 총 5명의 학생이 시험을 보았는데 시험 점수가 60점 이상이면 합격이고 그렇지 않으면 불합격이다. 합격인지 불합격인지 결과를 보여 주시오.
```

우선 학생 5 명의 시험 점수를 리스트로 표현해 보자.

```
1 marks = [90, 25, 67, 45, 80]
```

1 번 학생은 90 점이고 5 번 학생은 80 점이다.

이런 점수를 차례로 검사해서 합격했는지 불합격했는지 통보해 주는 프로그램을 만들어 보자.

IDLE 에디터로 작성하자.

```
1 # marks1.py
2 marks = [90, 25, 67, 45, 80]
3
4 number = 0
5 for mark in marks:
6     number = number + 1
7     if mark >= 60:
8         print("%d번 학생은 합격입니다." % number)
9     else:
10        print("%d번 학생은 불합격입니다." % number)
```

각각의 학생에게 번호를 붙여 주기 위해 `number` 변수를 사용하였다. 점수 리스트 `marks` 에서 차례로 점수를 꺼내어 `mark` 라는 변수에 대입하고 for 문 안의 문장들을 수행한다. 우선 for 문이 한 번씩 수행될 때

마다 **number** 는 1 씩 증가한다.

이 프로그램을 실행하면 **mark** 가 60 이상일 때 합격 메시지를 출력하고 60 을 넘지 않을 때 불합격 메시지를 출력한다. 명령 프롬프트 창을 열어 실행해 보자.

```
1 C:\doit>python marks1.py
2 1번 학생은 합격입니다.
3 2번 학생은 불합격입니다.
4 3번 학생은 합격입니다.
5 4번 학생은 불합격입니다.
6 5번 학생은 합격입니다.
```

for 문과 continue

while 문에서 살펴본 **continue** 문을 **for** 문에서도 사용할 수 있다. 즉 **for** 문 안의 문장을 수행하는 도중에 **continue** 문을 만나면 **for** 문의 처음으로 돌아가게 된다.

앞에서 **for** 문 응용 예제를 그대로 사용해서 60 점 이상인 사람에게는 축하 메시지를 보내고 나머지 사람에게는 아무 메시지도 전하지 않는 프로그램을 IDLE 에디터로 작성해 보자.

```
1 # marks2.py
2 marks = [90, 25, 67, 45, 80]
3
4 number = 0
5 for mark in marks:
6     number = number + 1
7     if mark < 60:
8         continue
9     print("%d번 학생 축하합니다. 합격입니다. " % number)
```

점수가 60 점 이하인 학생일 경우에는 **mark < 60**이 참이 되어 **continue** 문이 수행된다. 따라서 축하 메시지를 출력하는 부분인 **print** 문을 수행하지 않고 **for** 문의 처음으로 돌아가게 된다.

```
1 C:\doit>python marks2.py
2 1번 학생 축하합니다. 합격입니다.
3 3번 학생 축하합니다. 합격입니다.
4 5번 학생 축하합니다. 합격입니다.
```

for 문과 함께 자주 사용하는 range 함수

for 문은 숫자 리스트를 자동으로 만들어 주는 **range** 함수와 함께 사용하는 경우가 많다. 다음은 **range** 함수의 간단한 사용법이다.

```
1 >>> a = range(10)
```

```
2 >>> a
3 range(0, 10)
```

`range(10)` 은 0 부터 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.

```
1 >>> a = range(1, 11)
2 >>> a
3 range(1, 11)
```

range 함수의 예시 살펴보기

`for` 와 `range` 함수를 사용하면 1 부터 10 까지 더하는 것을 다음과 같이 쉽게 구현할 수 있다.

```
1 >>> add = 0
2 >>> for i in range(1, 11):
3 ...     add = add + i
4 ...
5 >>> print(add)
6 55
```

`range(1, 11)`은 숫자 1 부터 10 까지 (1 이상 11 미만) 의 숫자를 데이터로 갖는 객체이다. 따라서 위 예에서 `i` 변수에 숫자가 1 부터 10 까지 하나씩 차례로 대입되면서 `add = add + i` 문장을 반복적으로 수행하고 `add` 는 최종적으로 55 가 된다.

또한 우리가 앞에서 살펴본 “60 점 이상이면 합격” 이라는 문장을 출력하는 예제도 `range` 함수를 사용해서 다음과 같이 바꿀 수 있다.

```
1 # marks3.py
2 marks = [90, 25, 67, 45, 80]
3 for number in range(len(marks)):
4     if marks[number] < 60:
5         continue
6     print("%d번 학생 축하합니다. 합격입니다." % (number+1))
```

`len` 함수는 리스트 안의 요소 개수를 돌려주는 함수이다. 따라서 `len(marks)`는 5 가 될 것이고 `range(len(marks))`는 `range(5)`가 될 것이다. `number` 변수에는 차례로 0 부터 4 까지의 숫자가 대입될 것이고, `marks[number]`는 차례대로 90, 25, 67, 45, 80 값을 갖게 된다. 결과는 `marks2.py` 예제와 동일하다.

for 와 range 를 이용한 구구단

for 와 range 함수를 사용하면 소스 코드 단 4 줄만으로 구구단을 출력할 수 있다. 들여쓰기에 주의하며 입력해 보자.

```
1 >>> for i in range(2,10):          # 1번 for문
2     ...     for j in range(1, 10):  # 2번 for문
3     ...         print(i*j, end=" ")
4     ...     print('\n')
5     ...
6 2 4 6 8 10 12 14 16 18
7 3 6 9 12 15 18 21 24 27
8 4 8 12 16 20 24 28 32 36
9 5 10 15 20 25 30 35 40 45
10 6 12 18 24 30 36 42 48 54
11 7 14 21 28 35 42 49 56 63
12 8 16 24 32 40 48 56 64 72
13 9 18 27 36 45 54 63 72 81
```

위 예를 보면 for 문을 두 번 사용했다. 1 번 for 문에서 2 부터 9 까지의 숫자 (`range(2, 10)`) 가 차례로 `i` 에 대입된다. `i` 가 처음 2 일 때 2 번 for 문을 만나게 된다. 2 번 for 문에서 1 부터 9 까지의 숫자 (`range(1, 10)`) 가 `j` 에 대입되고 그다음 문장 `print(i*j)` 를 수행한다.

따라서 `i` 가 2 일 때 `2*1, 2*2, 2*3, ... 2*9` 까지 차례로 수행되며 그 값을 출력하게 된다. 그다음으로 `i` 가 3 일 때 역시 2 일 때와 마찬가지로 수행될 것이고 `i` 가 9 일 때까지 계속 반복된다. 그리고 `print(i*j, end=" ")` 와 같이 `print` 함수에 `end` 파라미터를 설정한 이유는 해당 결괏값을 출력할 때 다음줄로 넘기지 않고 그 줄에 계속해서 출력하기 위해서이다.

- `print` 문의 `end` 매개변수에는 디폴트로 줄바꿈 문자 (`\n`) 가 세팅되어 있다.
- `print` 문은 04-2 장에서 보다 자세히 다룬다.

그다음에 이어지는 `print('\n')` 는 2 단, 3 단 등을 구분하기 위해 사용했다. 두 번째 for 문이 끝나면 결괏값을 다음 줄부터 출력하게 하는 역할을 한다.

리스트 컴프리헨션 사용하기

리스트 안에 for 문을 포함하는 리스트 컴프리헨션 (List comprehension) 를 사용하면 좀 더 편리하고 직관적인 프로그램을 만들 수 있다. 다음 예제를 보자.

```
1 >>> a = [1,2,3,4]
2 >>> result = []
3 >>> for num in a:
4     ...     result.append(num*3)
5 ...
```

```
6 >>> print(result)
7 [3, 6, 9, 12]
```

위 예제는 `a` 리스트의 각 항목에 3 을 곱한 결과를 `result` 리스트에 담는 예제이다.

이것을 리스트 컴프리헨션을 사용하면 다음과 같이 간단히 해결할 수 있다.

```
1 >>> a = [1,2,3,4]
2 >>> result = [num * 3 for num in a]
3 >>> print(result)
4 [3, 6, 9, 12]
```

만약 `[1,2,3,4]` 중에서 짝수에만 3 을 곱하여 담고 싶다면 다음과 같이 리스트 컴프리헨션 안에 “if 조건” 을 사용할 수 있다.

```
1 >>> a = [1,2,3,4]
2 >>> result = [num * 3 for num in a if num % 2 == 0]
3 >>> print(result)
4 [6, 12]
```

리스트 컴프리헨션의 문법은 다음과 같다. “if 조건” 부분은 앞의 예제에서 볼 수 있듯이 생략할 수 있다.

```
1 [표현식 for 항목 in 반복가능객체 if 조건문]
```

조금 복잡하지만 `for` 문을 2 개 이상 사용하는 것도 가능하다. `for` 문을 여러 개 사용할 때의 문법은 다음과 같다.

```
1 [표현식 for 항목1 in 반복가능객체1 if 조건문1
2     for 항목2 in 반복가능객체2 if 조건문2
3     ...
4     for 항목n in 반복가능객체n if 조건문n]
```

만약 구구단의 모든 결과를 리스트에 담고 싶다면 리스트 컴프리헨션을 사용하여 다음과 같이 간단하게 구현할 수도 있다.

```
1 >>> result = [x*y for x in range(2,10)
2     ...      for y in range(1,10)]
3 >>> print(result)
4 [2, 4, 6, 8, 10, 12, 14, 16, 18, 3, 6, 9, 12, 15, 18, 21, 24, 27, 4, 8, 12, 16,
5 20, 24, 28, 32, 36, 5, 10, 15, 20, 25, 30, 35, 40, 45, 6, 12, 18, 24, 30, 36,
6 42, 48, 54, 7, 14, 21, 28, 35, 42, 49, 56, 63, 8, 16, 24, 32, 40, 48, 56, 64, 72,
7 9, 18, 27, 36, 45, 54, 63, 72, 81]
```

지금껏 우리는 프로그램 흐름을 제어하는 `if` 문, `while` 문, `for` 문에 대해 알아보았다. 아마도 여러분은 `while` 문과 `for` 문을 보면서 2 가지가 아주 비슷하다는 느낌을 받았을 것이다. 실제로 `for` 문으로 작성한 코드를 `while` 문으로 바꿀 수 있는 경우도 많고, `while` 문을 `for` 문으로 바꾸어서 사용할 수 있는 경우도 많다.

03 장 연습문제

Q1. 조건문의 참과 거짓

다음 코드의 결과값은 무엇일까?

```
1 a = "Life is too short, you need python"
2
3 if "wife" in a: print("wife")
4 elif "python" in a and "you" not in a: print("python")
5 elif "shirt" not in a: print("shirt")
6 elif "need" in a: print("need")
7 else: print("none")
```

Q2. 3 의 배수의 합

while 문을 사용해 1 부터 1000 까지의 자연수 중 3 의 배수의 합을 구해 보자.

Q3. 별 표시하기

while 문을 사용하여 다음과 같이 별 (*) 을 표시하는 프로그램을 작성해 보자.

```
1 *
2 **
3 ***
4 ****
5 *****
```

Q4. 1 부터 100 까지 출력

for 문을 사용해 1 부터 100 까지의 숫자를 출력해 보자.

Q5. 평균점수 구하기

A 학급에 총 10 명의 학생이 있다. 이 학생들의 중간고사 점수는 다음과 같다.

[70, 60, 55, 75, 95, 90, 80, 80, 85, 100]

for 문을 사용하여 A 학급의 평균 점수를 구해 보자.

Q6. 리스트 컴프리헨션

리스트 중에서 홀수에만 2 를 곱하여 저장하는 다음 코드가 있다.

```
1 numbers = [1, 2, 3, 4, 5]
2 result = []
3 for n in numbers:
4     if n % 2 == 1:
5         result.append(n*2)
```

위 코드를 리스트 컴프리헨션 (list comprehension) 을 사용하여 표현해 보자.

04 장 파이썬의 입력과 출력

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 API 로 공개하여 외부 프로그램들도 사용할 수 있게 만들 것인지 그 모든 것이 입출력과 관련이 있다.

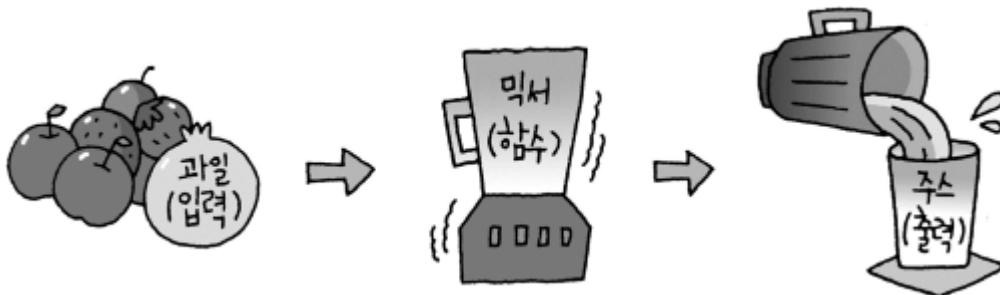
04-1 함수

[추천 동영상 강의] : <https://www.youtube.com/watch?v=0b7xo33cfa4>

함수란 무엇인가?

함수를 설명하기 전에 믹서를 생각해 보자. 우리는 믹서에 과일을 넣는다. 그리고 믹서를 사용해서 과일을 갈아 과일 주스를 만든다. 우리가 믹서에 넣는 과일은 “입력” 이 되고 과일 주스는 “출력 (결과값)” 이 된다.

그렇다면 믹서는 무엇인가?



(믹서는 과일을 입력받아 주스를 출력하는 함수와 같다.)

우리가 배우려는 함수가 바로 믹서와 비슷하다. 입력값을 가지고 어떤 일을 수행한 다음에 그 결과물을 내놓는 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 함수이다. 하지만 이를 수학 시간에 배운 직선 그래프로만 알고 있지 입력 (x) 값에 따라 출력 (y) 값이 변하는 함수라는 사실에 대해서는 별로 관심을 두지 않았을 것이다.

이제 우리는 함수에 대해 조금 더 생각해 보는 시간을 가져야 한다. 프로그래밍에서 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 문장으로 묶어서 “어떤 입력값을 주었을 때 어떤 결과값을 돌려준다”라는 식의 함수로 작성하는 것이 현명하다.

함수를 사용하는 또 다른 이유는 자신이 작성한 프로그램을 기능 단위의 함수로 분리해 놓으면 프로그램 흐름을 일목요연하게 볼 수 있기 때문이다. 마치 공장에서 원재료가 여러 공정을 거쳐 하나의 완제품이 되는 것처럼 프로그램에서도 입력한 값이 여러 함수를 거치면서 원하는 결과값을 내는 것을 볼 수 있다. 이렇게 되면 프로그램 흐름도 잘 파악할 수 있고 오류가 어디에서 나는지도 쉽게 알아차릴 수 있다.

파이썬 함수의 구조

파이썬 함수의 구조는 다음과 같다.

```
1 def 함수명(매개변수):  
2     <수행할 문장1>  
3     <수행할 문장2>  
4     ...
```

def는 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 문장을 입력한다.

간단하지만 많은 것을 설명해 주는 다음 예를 보자.

```
1 def add(a, b):  
2     return a + b
```

위 함수는 다음과 같이 풀이된다.

“이 함수의 이름은 add 이고 입력으로 2 개의 값을 받으며 리턴값 (출력값) 은 2 개의 입력값을 더한 값이다.”

여기에서 return 은 함수의 결과값 (리턴값) 을 돌려주는 명령어이다. 먼저 다음과 같이 add 함수를 만들자.

```
1 >>> def add(a, b):  
2     ...     return a+b  
3     ...  
4 >>>
```

이제 직접 `add` 함수를 사용해 보자.

```
1 >>> a = 3
2 >>> b = 4
3 >>> c = add(a, b)
4 >>> print(c)
5 7
```

변수 `a`에 3, `b`에 4를 대입한 다음 앞에서 만든 `add` 함수에 `a`와 `b`를 입력값으로 넣어 준다. 그리고 변수 `c`에 `add` 함수의 리턴값을 대입하면 `print(c)`로 `c`의 값을 확인할 수 있다.

매개변수와 인수

매개변수와 인수는 혼용해서 사용되는 헷갈리는 용어이므로 잘 기억해 두자. 매개변수는 함수에 입력으로 전달된 값을 받는 변수를 의미하고 인수는 함수를 호출할 때 전달하는 입력값을 의미한다.

```
1 def add(a, b): # a, b는 매개변수
2     return a+b
3
4 print(add(3, 4)) # 3, 4는 인수
```

- 매개변수 - 함수에 전달된 값을 저장하는 변수
- 인수 - 함수에 전달하는 값

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 용어를 자주 만나게 된다. 한 예로 입력값을 다른 말로 함수의 인수, 파라미터, 매개변수 등으로 말하기도 하고 함수의 리턴값을 결과값, 출력값, 반환 값, 돌려주는 값 등으로 말하기도 한다. 이렇듯 많은 용어가 여러 가지 다른 말로 표현되지만 의미는 동일한 경우가 많다. 따라서 이런 용어를 기억해 놓아야 머리가 덜 아플 것이다.

이 책에서는 가급적 함수를 사용할 때는 입력 값, 리턴 값이라는 용어로 통일하여 사용하겠다.

입력값과 리턴값에 따른 함수의 형태

함수는 들어온 입력값을 받아 어떤 처리를 하여 적절한 리턴값을 돌려준다.

입력값 → 함수 → 리턴값

함수의 형태는 입력값과 리턴값의 존재 유무에 따라 4 가지 유형으로 나뉜다. 자세히 알아보자.

일반적인 함수

입력값이 있고 리턴값이 있는 함수가 일반적인 함수이다. 앞으로 여러분이 프로그래밍을 할 때 만들 함수는 대부분 다음과 비슷한 형태일 것이다.

```
1 def 함수이름(매개변수):  
2     <수행할 문장>  
3     ...  
4     return 리턴값
```

다음은 일반적인 함수의 전형적인 예이다.

```
1 def add(a, b):  
2     result = a + b  
3     return result
```

add 함수는 2 개의 입력값을 받아서 서로 더한 결과값을 리턴한다.

이 함수를 사용하는 방법은 다음과 같다. 입력값으로 3 과 4 를 주고 리턴값을 받아 보자.

```
1 >>> a = add(3, 4)  
2 >>> print(a)  
3 7
```

이처럼 입력값과 리턴값이 있는 함수의 사용법을 정리하면 다음과 같다.

리턴값을 받을 변수 = 함수이름 (입력인수 1, 입력인수 2, ...)

입력값이 없는 함수

입력값이 없는 함수가 존재할까? 당연히 존재한다. 다음을 보자.

```
1 >>> def say():  
2     ...     return 'Hi'  
3     ...  
4 >>>
```

say 라는 이름의 함수를 만들었다. 그런데 매개변수 부분을 나타내는 함수 이름 뒤의 괄호 안이 비어 있다. 이 함수는 어떻게 사용하는 걸까?

다음은 직접 입력해 보자.

```
1 >>> a = say()
2 >>> print(a)
3 Hi
```

위 함수를 쓰기 위해서는 `say()`처럼 괄호 안에 아무 값도 넣지 않아야 한다. 이 함수는 입력값은 없지만 리턴값으로 “Hi” 라는 문자열을 리턴한다. 즉, `a = say()`처럼 작성하면 `a`에 “Hi” 라는 문자열이 대입 되는 것이다.

이처럼 입력값이 없고 리턴값만 있는 함수는 다음과 같이 사용한다.

리턴값을 받을 변수 = 함수이름 ()

리턴값이 없는 함수

리턴값이 없는 함수 역시 존재한다. 다음 예를 보자.

```
1 >>> def add(a, b):
2 ...     print("%d, %d의 합은 %d입니다." % (a, b, a+b))
3 ...
4 >>>
```

리턴값이 없는 함수는 호출해도 리턴되는 값이 없기 때문에 다음과 같이 사용한다.

```
1 >>> add(3, 4)
2 3, 4의 합은 7입니다.
```

즉 리턴값이 없는 함수는 다음과 같이 사용한다.

함수이름 (입력인수 1, 입력인수 2, ...)

리턴값이 진짜 없는지 확인하기 위해 다음 예를 직접 입력해 보자.

```
1 >>> a = add(3, 4)
2 3, 4의 합은 7입니다.
```

아마도 여러분은 “3, 4의 합은 7입니다.” 라는 문장을 출력해 주었는데 왜 리턴값이 없다는 것인지 의아 하게 생각할 수도 있다. 이 부분이 초보자들이 혼란스러워하는 부분이기도 한데 `print` 문은 함수의 구성 요소 중 하나인 <수행할 문장>에 해당하는 부분일 뿐이다. 리턴값은 당연히 없다. 리턴값은 오직 `return` 명령어로만 돌려받을 수 있다.

이를 확인해 보자. 리턴받을 값을 `a` 변수에 대입하여 출력해 보면 리턴값이 있는지 없는지 알 수 있다.

```
1 >>> a = add(3, 4)
2 3, 4의 합은 7입니다.
3 >>> print(a)
4 None
```

a 값으로 None 이 출력되었다. None 이란 거짓을 나타내는 자료형이라고 언급한 적이 있다. add 함수처럼 리턴값이 없을 때 `a = add(3, 4)`처럼 쓰면 함수 add 는 리턴값으로 a 변수에 None 을 리턴한다. None 을 리턴한다는 것은 리턴값이 없다는 것이다.

입력값도 리턴값도 없는 함수

입력값도 리턴값도 없는 함수 역시 존재한다. 다음 예를 보자.

```
1 >>> def say():
2 ...     print('Hi')
3 ...
4 >>>
```

입력 인수를 받는 매개변수도 없고 return 문도 없으니 입력값도 리턴값도 없는 함수이다.

이 함수를 사용하는 방법은 단 한 가지이다.

```
1 >>> say()
2 Hi
```

즉 입력값도 리턴값도 없는 함수는 다음과 같이 사용한다.

함수이름 ()

매개변수 지정하여 호출하기

함수를 호출할 때 매개변수를 지정할 수도 있다. 다음 예를 보자.

```
1 >>> def sub(a, b):
2 ...     return a - b
3 ...
```

두 개의 숫자를 입력 받아 첫번째 수에서 두번째 수를 뺄셈하여 리턴하는 sub 함수이다. 이 함수를 다음과 같이 매개변수를 지정하여 사용할 수 있다.

```
1 >>> result = sub(a=7, b=3) # a에 7, b에 3을 전달
2 >>> print(result)
3 4
```

매개변수를 지정하면 다음과 같이 순서에 상관없이 사용할 수 있다는 장점이 있다.

```
1 >>> result = sub(b=5, a=3) # b에 5, a에 3을 전달
2 >>> print(result)
3 -2
```

입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?

입력값이 여러 개일 때 그 입력값을 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.

```
1 def 함수이름(*매개변수):
2     <수행할 문장>
3     ...
```

일반적으로 볼 수 있는 함수 형태에서 괄호 안의 매개변수 부분이 ***매개변수**로 바뀌었다.

여러 개의 입력값을 받는 함수 만들기

다음 예를 통해 여러 개의 입력값을 모두 더하는 함수를 직접 만들어 보자. 예를 들어 `add_many(1, 2)` 이면 3을, `add_many(1,2,3)`이면 6을, `add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`이면 55를 리턴하는 함수를 만들어 보자.

```
1 >>> def add_many(*args):
2 ...     result = 0
3 ...     for i in args:
4 ...         result = result + i
5 ...     return result
6 ...
7 >>>
```

위에서 만든 `add_many` 함수는 입력값이 몇 개이든 상관없다. `*args`처럼 매개변수 이름 앞에 `*`을 붙이면 입력값을 전부 모아서 튜플로 만들어 주기 때문이다. 만약 `add_many(1, 2, 3)`처럼 이 함수를 쓰면 `args`는 `(1, 2, 3)`이 되고, `add_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. 여기에서 `*args`는 임의로 정한 변수 이름이다. `*pey`, `*python`처럼 아무 이름이나 써도 된다.

`args`는 인수를 뜻하는 영어 단어 `arguments`의 약자이며 관례적으로 자주 사용한다.

작성한 `add_many` 함수를 다음과 같이 사용해 보자.

```
1 >>> result = add_many(1,2,3)
2 >>> print(result)
3 6
4 >>> result = add_many(1,2,3,4,5,6,7,8,9,10)
5 >>> print(result)
6 55
```

`add_many(1,2,3)`으로 함수를 호출하면 6을 리턴하고, `add_many(1, 2, 3, 4, 5, 6, 7, 8, 9,10)`을 호출하면 55를 리턴하는 것을 확인할 수 있다.

여러 개의 입력을 처리할 때 `def add_many(*args)`처럼 함수의 매개변수로 `*args` 하나만 사용할 수 있는 것은 아니다.

다음 예를 보자.

```
1 >>> def add_mul(choice, *args):
2 ...     if choice == "add":
3 ...         result = 0
4 ...         for i in args:
5 ...             result = result + i
6 ...     elif choice == "mul":
7 ...         result = 1
8 ...         for i in args:
9 ...             result = result * i
10 ...     return result
11 ...
12 >>>
```

`add_mul` 함수는 여러 개의 입력값을 의미하는 `*args` 매개변수 앞에 `choice` 매개변수가 추가되어 있다.

이 함수는 다음과 같이 사용할 수 있다.

```
1 >>> result = add_mul('add', 1,2,3,4,5)
2 >>> print(result)
3 15
4 >>> result = add_mul('mul', 1,2,3,4,5)
5 >>> print(result)
6 120
```

매개변수 `choice` 에 'add' 가 입력된 경우 `*args`에 입력되는 모든 값을 더해서 15 를 돌려주고, 'mul' 이 입력된 경우 `*args`에 입력되는 모든 값을 곱해서 120 을 리턴한다.

키워드 매개변수 kwargs

이번에는 키워드 매개변수에 대해 알아보자. 키워드 매개변수를 사용할 때는 매개변수 앞에 별 두 개 (`**`) 를 붙인다. 역시 이것도 예제로 알아보자. 먼저 다음과 같은 함수를 작성해 보자.

```
1 >>> def print_kwargs(**kwargs):
2 ...     print(kwargs)
3 ...
```

`print_kwargs` 함수는 입력받은 매개변수 `kwargs` 를 출력하는 단순한 함수이다. 이제 이 함수를 다음과 같이 사용해 보자.

```
1 >>> print_kwargs(a=1)
2 {'a': 1}
```

```
3 >>> print_kwargs(name='foo', age=3)
4 {'age': 3, 'name': 'foo'}
```

함수의 입력값으로 `a=1`이 사용되면 `kwargs`는 `{'a': 1}`이라는 딕셔너리가 되고 입력값으로 `name='foo', age=3`이 사용되면 `kwargs`는 `{'age': 3, 'name': 'foo'}`라는 딕셔너리가 된다. 즉, `**kwargs`처럼 매개변수 이름 앞에 `**`을 붙이면 매개변수 `kwargs`는 딕셔너리가 되고 모든 `key=value` 형태의 입력값이 그 딕셔너리에 저장된다는 것을 알 수 있다.

`kwargs`는 keyword arguments의 약자이며 `args`와 마찬가지로 관례적으로 사용한다.

함수의 리턴값은 언제나 하나이다

먼저 다음 함수를 만들어 보자.

```
1 >>> def add_and_mul(a,b):
2 ...     return a+b, a*b
```

`add_and_mul` 함수는 2개의 입력 인수를 받아 더한 값과 곱한 값을 리턴하는 함수이다.

이 함수를 다음과 같이 호출하면 어떻게 될까?

```
1 >>> result = add_and_mul(3,4)
```

리턴값은 `a+b`와 `a*b` 2개인데 리턴값을 받아들이는 변수는 `result` 하나만 쓰였으니 오류가 발생하지 않을까? 당연한 의문이다. 하지만 오류는 발생하지 않는다. 그 이유는 함수의 리턴값은 2개가 아니라 언제나 1개라는 데 있다. `add_and_mul` 함수의 리턴값 `a+b`와 `a*b`는 튜플값 하나인 `(a+b, a*b)`로 리턴된다.

따라서 `result` 변수는 다음과 같은 값을 갖게 된다.

```
1 result = (7, 12)
```

즉 결과값으로 `(7, 12)`라는 튜플 값을 갖게 되는 것이다.

만약 이 하나의 튜플 값을 2개의 값으로 분리하여 받고 싶다면 다음과 같이 함수를 호출하면 된다.

```
1 >>> result1, result2 = add_and_mul(3, 4)
```

이렇게 호출하면 `result1, result2 = (7, 12)`가 되어 `result1`은 7이 되고 `result2`는 12가 된다.

또 다음과 같은 의문이 생길 수도 있다.


```

1 >>> def add_and_mul(a,b):
2 ...     return a+b
3 ...     return a*b
4 ...
5 >>>

```

위와 같이 `return` 문을 2 번 사용하면 2 개의 리턴값을 돌려주지 않을까? 하지만 파이썬에서 위와 같은 함수는 참 어리석은 함수이다.

그 이유는 `add_and_mul` 함수를 호출해 보면 알 수 있다.

```

1 >>> result = add_and_mul(2, 3)
2 >>> print(result)
3 5

```

`add_and_mul(2, 3)`의 리턴값은 5 하나뿐이다. 두 번째 `return` 문인 `return a*b`는 실행되지 않았다는 뜻이다. 따라서 이 함수는 다음과 완전히 동일하다.

```

1 >>> def add_and_mul(a,b):
2 ...     return a+b
3 ...
4 >>>

```

즉 함수는 `return` 문을 만나는 순간 리턴값을 돌려준 다음 함수를 빠져나가게 된다.

return 의 또 다른 쓰임새

특별한 상황일 때 함수를 빠져나가고 싶다면 `return` 을 단독으로 써서 함수를 즉시 빠져나갈 수 있다. 다음 예를 보자.

```

1 >>> def say_nick(nick):
2 ...     if nick == "바보":
3 ...         return
4 ...     print("나의 별명은 %s 입니다." % nick)
5 ...
6 >>>

```

위 함수는 '별명' 을 입력으로 전달받아 출력하는 함수이다. 이 함수 역시 리턴값은 없다 (문자열을 출력한다는 것과 리턴 값이 있다는 것은 전혀 다른 말이다. 혼동하지 말자. 함수의 리턴 값은 오로지 `return` 문에 의해서만 생성된다).

만약에 입력값으로 '바보' 라는 값이 들어오면 문자열을 출력하지 않고 함수를 즉시 빠져나간다.

```

1 >>> say_nick('야호')
2 나의 별명은 야호입니다.
3 >>> say_nick('바보')
4 >>>

```

이처럼 `return` 으로 함수를 빠져나가는 방법은 실제 프로그래밍에서 자주 사용된다.

매개변수에 초깃값 미리 설정하기

이번에는 조금 다른 형태로 함수의 인수를 전달하는 방법에 대해서 알아보자. 매개변수에 초깃값을 미리 설정해 주는 경우이다.

>>>(프롬프트) 표시가 없으니 IDLE 에디터에서 작성하자.

```
1 def say_myself(name, age, man=True):
2     print("나의 이름은 %s 입니다." % name)
3     print("나이는 %d살 입니다." % age)
4     if man:
5         print("남자입니다.")
6     else:
7         print("여자입니다.")
```

say_myself 함수는 3 개의 매개변수를 받아서 마지막 인수인 man 이 True 이면 “남자입니다.”, False 이면 “여자입니다.” 를 출력한다.

위 함수를 보면 매개변수가 name, age, man=True 이렇게 3 개다. 그런데 낯선 것이 나왔다. man=True처럼 매개변수에 미리 값을 넣어 준 것이다. 이것이 바로 함수의 매개변수에 초깃값을 설정하는 방법이다.

say_myself 함수는 다음처럼 2 가지 방법으로 사용할 수 있다.

```
1 say_myself("박응용", 27)
2 say_myself("박응용", 27, True)
```

입력값으로 “박응용”, 27 처럼 2 개를 주면 name 에는 “박응용” 이 age 에는 27 이 대입된다. 그리고 man 이라는 변수에는 입력값을 주지 않았지만 초깃값 True 를 갖게 된다.

따라서 위 예에서 say_myself 함수를 사용한 2 가지 방법은 모두 다음처럼 동일한 결과를 출력한다.

```
1 나의 이름은 박응용입니다.
2 나이는 27살입니다.
3 남자입니다.
```

이제 초깃값이 설정된 부분을 False 로 바꿔 호출해 보자.

```
1 say_myself("박응선", 27, False)
```

man 변수에 False 값이 대입되어 다음과 같은 결과가 출력된다.

```
1 나의 이름은 박응선입니다.
2 나이는 27살입니다.
3 여자입니다.
```

함수의 매개변수에 초깃값을 설정할 때 주의할 것이 하나 있다. 만약 위에서 본 `say_myself` 함수를 다음과 같이 만들면 어떻게 될까?

```
1 def say_myself(name, man=True, age):
2     print("나의 이름은 %s 입니다." % name)
3     print("나이는 %d살 입니다." % age)
4     if man:
5         print("남자입니다.")
6     else:
7         print("여자입니다.")
```

이전 함수와 바뀐 부분은 초깃값을 설정한 매개변수의 위치이다. 결론을 미리 말하면 이것은 함수를 실행할 때 오류가 발생한다.

얼핏 생각하기에 위 함수를 호출하려면 다음과 같이 하면 될 것 같다.

```
1 say_myself("박응용", 27)
```

위와 같이 함수를 호출한다면 `name` 변수에는 “박응용” 이 들어갈 것이다. 하지만 파이썬 인터프리터는 27을 `man` 매개변수와 `age` 매개변수 중 어느 곳에 대입해야 할지 판단하기가 어려워 이러한 상황에서는 오류를 발생시키게 했다.

오류 메시지를 보면 다음과 같다.

```
1 SyntaxError: non-default argument follows default argument
```

위 오류 메시지는 초깃값이 없는 매개변수 (`age`) 는 초깃값이 있는 매개변수 (`man`) 뒤에 사용할 수 없다는 뜻이다. 즉 매개변수로 (`name, age, man=True`)는 되지만 (`name, man=True, age`)는 안 된다는 것이다. 초기화시키고 싶은 매개변수는 항상 뒤쪽에 놓아야 함을 잊지 말자.

함수 안에서 선언한 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.

다음 예를 보자.

```
1 # vartest.py
2 a = 1
3 def vartest(a):
4     a = a + 1
5
6 vartest(a)
7 print(a)
```

먼저 `a` 라는 변수를 생성하고 `1` 을 대입했다. 그리고 입력으로 들어온 값에 `1` 을 더해 주고 결과값은 리턴하지 않는 `vartest` 함수를 선언했다. 그리고 `vartest` 함수에 입력값으로 `a` 를 주었다. 마지막으로 `a` 의 값을 `print(a)` 로 출력했다.

과연 어떤 값이 출력될까?

`vartest` 함수에서 매개변수 `a` 의 값에 `1` 을 더했으니까 `2` 가 출력될 것 같지만 프로그램 소스를 작성해서 실행해 보면 결과값은 `1` 이 나온다. 그 이유는 함수 안에서 사용하는 매개변수는 함수 안에서만 사용하는 “함수만의 변수” 이기 때문이다. 즉 `def vartest(a)` 에서 입력값을 전달받는 매개변수 `a` 는 함수 안에서만 사용하는 변수이지 함수 밖의 변수 `a` 와는 전혀 상관이 없다는 뜻이다.

따라서 `vartest` 함수는 다음처럼 매개변수 이름을 `hello` 로 바꾸어도 이전의 `vartest` 함수와 완전히 동일하게 동작한다.

```
1 def vartest(hello):
2     hello = hello + 1
```

즉 함수 안에서 사용하는 매개변수는 함수 밖의 변수 이름과는 전혀 상관이 없다는 뜻이다.

다음 예를 보면 더욱 분명하게 이해할 수 있을 것이다.

```
1 # vartest_error.py
2 def vartest(a):
3     a = a + 1
4
5 vartest(3)
6 print(a)
```

위 프로그램 소스를 에디터로 작성해서 실행하면 어떻게 될까? 오류가 발생할 것이라고 생각한 독자는 모든 것을 이해한 독자이다. `vartest(3)` 을 수행하면 `vartest` 함수 안에서 `a` 는 `4` 가 되지만 함수를 호출하고 난 뒤에 `print(a)` 문장은 오류가 발생하게 된다. 그 이유는 `print(a)` 에서 사용한 `a` 변수를 어디에서도 찾을 수가 없기 때문이다. 다시 말하지만 함수 안에서 선언한 매개변수는 함수 안에서만 사용될 뿐 함수 밖에서는 사용되지 않는다. 이것을 이해하는 것은 매우 중요하다.

함수 안에서 함수 밖의 변수를 변경하는 방법

그렇다면 `vartest` 라는 함수를 사용해서 함수 밖의 변수 `a` 를 `1` 만큼 증가시킬 수 있는 방법은 없을까? 이 질문에는 `2` 가지 해결 방법이 있다.

1. return 사용하기

```
1 # vartest_return.py
2 a = 1
3 def vartest(a):
```

```

4     a = a + 1
5     return a
6
7     a = vartest(a)
8     print(a)

```

첫 번째 방법은 `return` 을 사용하는 방법이다. `vartest` 함수는 입력으로 들어온 값에 1 을 더한 값을 리턴하도록 변경했다. 따라서 `a = vartest(a)` 라고 작성하면 `a` 에는 `vartest` 함수의 리턴값이 대입된다.

여기에서도 물론 `vartest` 함수 안의 `a` 매개변수는 함수 밖의 `a` 와는 다른 것이다.

2. global 명령어 사용하기

```

1 # vartest_global.py
2 a = 1
3 def vartest():
4     global a
5     a = a+1
6
7 vartest()
8 print(a)

```

두 번째 방법은 `global` 명령어를 사용하는 방법이다. 위 예에서 볼 수 있듯이 `vartest` 함수 안의 `global a` 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. 우리말로 “람다” 라고 읽고 `def` 를 사용해야 할 정도로 복잡하지 않거나 `def` 를 사용할 수 없는 곳에 주로 쓰인다.

사용법은 다음과 같다.

함수명 = `lambda` 매개변수 1, 매개변수 2, ... : 매개변수를 이용한 표현식

한번 직접 만들어 보자.

```

1 >>> add = lambda a, b: a+b
2 >>> result = add(3, 4)
3 >>> print(result)
4 7

```

`lambda`로 만든 함수는 `return` 명령어가 없어도 표현식의 결과값을 리턴한다.

`add`는 두 개의 인수를 받아 서로 더한 값을 리턴하는 `lambda` 함수이다. 위 예제는 `def`를 사용한 다음 함수와 하는 일이 완전히 동일하다.

```
1 >>> def add(a, b):
2 ...     return a+b
3 ...
4 >>> result = add(3, 4)
5 >>> print(result)
6 7
```

04-2 사용자 입력과 출력

[추천 동영상 강의] : https://www.youtube.com/watch?v=L7E9l_cU17U

우리들이 사용하는 대부분의 완성된 프로그램은 사용자 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.

우리는 이미 함수 부분에서 입력과 출력이 어떤 의미를 가지는지 알아보았다. 지금부터는 좀 더 다양한 입출력 방법에 대해서 알아보자.

사용자 입력

사용자가 입력한 값을 어떤 변수에 대입하고 싶을 때는 어떻게 해야 할까?

input의 사용

```
1 >>> a = input()
2 Life is too short, you need python
3 >>> a
4 'Life is too short, you need python'
5 >>>
```

input은 사용자가 키보드로 입력한 모든 것을 문자열로 저장한다.

프롬프트를 띄워서 사용자 입력 받기

사용자에게 입력받을 때 “숫자를 입력하세요”라든지 “이름을 입력하세요”라는 안내 문구 또는 질문이 나오도록 하고 싶을 때가 있다. 그럴 때는 input()의 괄호 안에 안내문구를 입력하여 프롬프트를 띄워주면 된다.

```
input("안내문구")
```

다음 예를 직접 입력해 보자.

```
1 >>> number = input("숫자를 입력하세요: ")
2 숫자를 입력하세요:
```

위와 같은 프롬프트를 볼 수 있을 것이다.

숫자를 입력하라는 프롬프트에 3을 입력하면 변수 `number`에 3이 대입된다. `print(number)`로 출력해서 제대로 입력되었는지 확인해 보자.

```
1 >>> number = input("숫자를 입력하세요: ")
2 숫자를 입력하세요: 3
3 >>> print(number)
4 3
5 >>>
```

`input`은 입력되는 모든 것을 문자열로 취급하기 때문에 `number`는 숫자가 아닌 문자열임에 주의하자.

```
1 >>> type(number)
2 <class 'str'>
3 >>>
```

print 자세히 알기

지금껏 우리가 사용한 `print` 문의 용도는 데이터를 출력하는 것이었다. 데이터를 출력하는 `print` 문의 사용예는 다음과 같다.

```
1 >>> a = 123
2 >>> print(a)
3 123
4 >>> a = "Python"
5 >>> print(a)
6 Python
7 >>> a = [1, 2, 3]
8 >>> print(a)
9 [1, 2, 3]
```

이제 `print` 문으로 할 수 있는 일에 대해서 조금 더 자세하게 알아보자.

큰따옴표 (") 로 둘러싸인 문자열은 + 연산과 동일하다

```
1 >>> print("life" "is" "too short") # 1
2 lifeistoo short
```



```
3 >>> print("life"+"is"+"too short") # 2
4 lifeis too short
```

위 예에서 1 과 2 는 완전히 동일한 결과값을 출력한다. 즉 따옴표로 둘러싸인 문자열을 연속해서 쓰면 + 연산을 한 것과 같다.

문자열 띄어쓰기는 콤마로 한다

```
1 >>> print("life", "is", "too short")
2 life is too short
```

콤마 (,) 를 사용하면 문자열 사이에 띄어쓰기를 할 수 있다.

한 줄에 결과값 출력하기

03-3 장에서 for 문을 공부할 때 만들었던 구구단 프로그램에서 보았듯이 한 줄에 결과값을 계속 이어서 출력하려면 매개변수 end 를 사용해 끝 문자를 지정해야 한다.

```
1 >>> for i in range(10):
2     ...     print(i, end=' ')
3     ...
4 0 1 2 3 4 5 6 7 8 9
```

end 매개변수의 초기값은 줄바꿈 (\n) 문자이다.

04-3 파일 읽고 쓰기

[추천 동영상 강의 04-3, 04-4 합본] : <https://www.youtube.com/watch?v=XX7y5CFqPdw>

우리는 이 책에서 이제까지 값을 “입력” 받을 때는 사용자가 직접 입력하는 방식을 사용했고 “출력” 할 때는 모니터 화면에 결괏값을 출력하는 방식을 사용했다. 하지만 입출력 방법이 꼭 이것만 있는 것은 아니다. 이번에는 파일을 통한 입출력 방법에 대해 알아보자. 여기에서는 파일을 새로 만든 다음 프로그램이 만든 결괏값을 새 파일에 적어볼 것이다. 또 파일에 적은 내용을 읽고, 새로운 내용을 추가하는 방법도 알아볼 것이다.

파일 생성하기

다음 코드를 IDLE 에디터로 작성하여 실행해 보자.

```
1 f = open("새 파일.txt", 'w')
2 f.close()
```

프로그램을 실행한 디렉터리에 새로운 파일이 하나 생성된 것을 확인할 수 있을 것이다. 파일을 생성하기 위해 파이썬 내장 함수 `open` 을 사용했다. `open` 함수는 다음과 같이 “파일 이름” 과 “파일 열기 모드” 를 입력값으로 받고 결괏값으로 파일 객체를 리턴한다.

파일 객체 = `open`(파일 이름, 파일 열기 모드)

파일 열기 모드에는 다음과 같은 것들이 있다.

파일열기모드	설명
<code>r</code>	읽기모드 - 파일을 읽기만 할 때 사용
<code>w</code>	쓰기모드 - 파일에 내용을 쓸 때 사용
<code>a</code>	추가모드 - 파일의 마지막에 새로운 내용을 추가 시킬 때 사용

파일을 쓰기 모드로 열면 해당 파일이 이미 존재할 경우 원래 있던 내용이 모두 사라지고, 해당 파일이 존재하지 않으면 새로운 파일이 생성된다. 위 예에서는 디렉터리에 파일이 없는 상태에서 새파일.txt를 쓰기 모드인 'w'로 열었기 때문에 새파일.txt라는 이름의 새로운 파일이 현재 디렉터리에 생성되었다.

만약 새파일.txt 파일을 C:/doit 디렉터리에 생성하고 싶다면 다음과 같이 작성해야 한다.

```
1 f = open("C:/doit/새 파일.txt", 'w')
2 f.close()
```

위 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.

파일 경로와 슬래시 (/)

파이썬 코드에서 파일 경로를 표시할 때 "C:/doit/새파일.txt" 처럼 슬래시 (/)를 사용할 수 있다. 만약 역슬래시 (\)를 사용한다면 "C:\\doit\\새파일.txt" 처럼 역슬래시를 2개 사용하거나 r"C:\doit\새파일.txt"와 같이 문자열 앞에 r 문자 (Raw String)를 덧붙여 사용해야 한다. 왜냐하면 "C:\note\test.txt"처럼 파일 경로에 \n과 같은 이스케이프 문자가 있을 경우 줄바꿈 문자로 해석되어 의도했던 파일 경로와 달라지기 때문이다.

파일을 쓰기 모드로 열어 내용 쓰기

위 예에서는 파일을 쓰기 모드로 열기만 했지 정작 아무것도 쓰지는 않았다. 이번에는 문자열 데이터를 파일에 직접 써 보자.

```
1 # writedata.py
2 f = open("C:/doit/새 파일.txt", 'w')
3 for i in range(1, 11):
4     data = "%d번째 줄입니다.\n" % i
5     f.write(data)
6 f.close()
```

위 프로그램을 다음 프로그램과 비교해 보자.

```
1 for i in range(1, 11):
2     data = "%d번째 줄입니다.\n" % i
3     print(data)
```

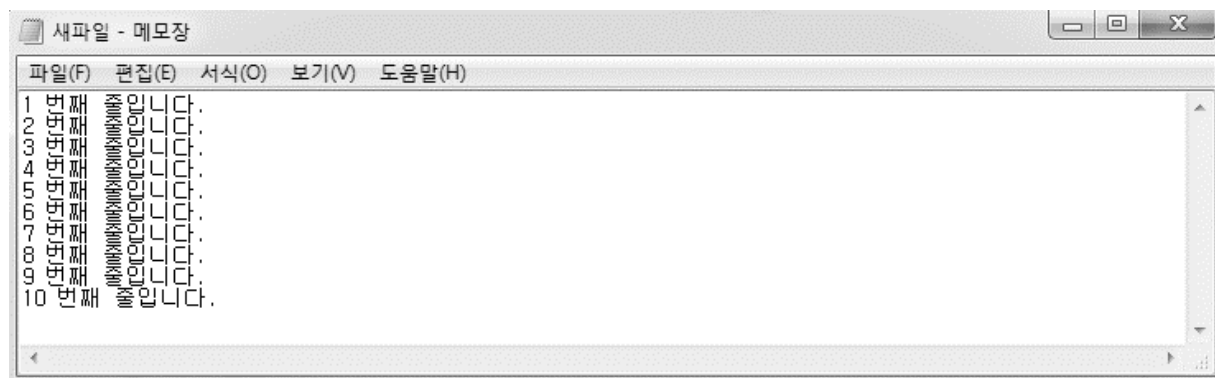
두 프로그램의 다른 점은 data를 출력하는 방법이다. 두 번째 방법은 우리가 계속 사용해 왔던 모니터 화면에 데이터를 출력하는 방법이고, 첫 번째 방법은 모니터 화면 대신 파일에 데이터를 적는 방법이다. 두

방법의 차이점은 `print` 함수 대신 파일 객체 `f`의 `write` 함수를 사용한 것 말고는 없으니 바로 눈에 들어올 것이다.

이제 명령 프롬프트 창에서 첫 번째 예제를 실행해 보자.

```
1 C:\Users> cd C:\doit
2 C:\doit> python writedata.py
3 C:\doit>
```

이 프로그램을 실행한 `C:/doit` 디렉터리를 살펴보면 새파일.txt 파일이 생성되었음을 확인할 수 있다. 새파일.txt 파일에는 어떤 내용이 담겨 있는지 확인해 보자.



모니터 화면에 출력될 내용이 고스란히 파일에 들어 있는 것을 볼 수 있다.

파일을 읽는 여러 가지 방법

파이썬에는 파일을 읽는 방법이 여러 가지 있다. 이번에는 그 방법을 자세히 알아보자.

readline 함수 이용하기

첫 번째 방법은 `readline` 함수를 사용하는 방법이다. 다음 예를 보자.

```
1 # readline_test.py
2 f = open("C:/doit/새 파일.txt", 'r')
3 line = f.readline()
4 print(line)
5 f.close()
```

위 예는 `f.open("새파일.txt", 'r')`로 파일을 읽기 모드로 연 후 `readline()`을 사용해서 파일의 첫 번째 줄을 읽어 출력하는 예제이다. 앞에서 만든 새파일.txt를 수정하거나 지우지 않았다면 위 프로그램을 실행했을 때 새파일.txt의 가장 첫 번째 줄이 화면에 출력될 것이다.

```
1 1번째 줄입니다.
```

만약 모든 줄을 읽어서 화면에 출력하고 싶다면 다음과 같이 작성하면 된다.

```
1 # readline_all.py
2 f = open("C:/doit/새 파일.txt", 'r')
3 while True:
4     line = f.readline()
5     if not line: break
6     print(line)
7 f.close()
```

즉 **while True:** 무한 루프 안에서 **f.readline()**을 사용해 파일을 계속해서 한 줄씩 읽어 들인다. 만약 더 이상 읽을 줄이 없으면 **break**를 수행한다 (**readline()**은 더 이상 읽을 줄이 없을 경우 빈 문자열('')을 리턴한다).

한 줄씩 읽어 출력할 때 줄 끝에 **\n** 문자가 있으므로 빈 줄도 같이 출력된다.

앞의 프로그램을 다음 프로그램과 비교해 보자.

```
1 while True:
2     data = input()
3     if not data: break
4     print(data)
```

위 예는 사용자의 입력을 받아서 그 내용을 출력하는 경우이다. 파일을 읽어서 출력하는 예제와 비교해 보자. 입력을 받는 방식만 다르다는 것을 바로 알 수 있을 것이다. 두 번째 예는 키보드를 사용한 입력 방법이고, 첫 번째 예는 파일을 사용한 입력 방법이다.

readlines 함수 사용하기

두 번째 방법은 **readlines** 함수를 사용하는 방법이다. 다음 예를 보자.

```
1 f = open("C:/doit/새 파일.txt", 'r')
2 lines = f.readlines()
3 for line in lines:
4     print(line)
5 f.close()
```

readlines 함수는 파일의 모든 줄을 읽어서 각각의 줄을 요소로 갖는 리스트를 리턴한다. 따라서 위 예에서 **lines**는 리스트 **["1 번째 줄입니다.\n", "2 번째 줄입니다.\n", ..., "10 번째 줄입니다.\n"]**가 된다. **f.readlines()**에서 **f.readline()**과는 달리 **s**가 하나 더 붙어 있음에 유의하자.

줄 바꿈 (\n) 문자 제거하기

파일을 읽을 때 줄 끝의 줄 바꿈 (\n) 문자를 제거하고 사용해야 할 경우가 많다. 다음처럼 `strip` 함수를 사용하면 줄 바꿈 문자를 제거할 수 있다.

```
1 f = open("C:/doit/새 파일.txt", 'r')
2 lines = f.readlines()
3 for line in lines:
4     line = line.strip() # 줄 끝의 줄 바꿈 문자를 제거한다.
5     print(line)
6 f.close()
```

read 함수 사용하기

세 번째 방법은 `read` 함수를 사용하는 방법이다. 다음 예를 보자.

```
1 f = open("C:/doit/새 파일.txt", 'r')
2 data = f.read()
3 print(data)
4 f.close()
```

`f.read()`는 파일의 내용 전체를 문자열로 리턴한다. 따라서 위 예의 `data`는 파일의 전체 내용이다.

파일 객체를 for 문과 함께 사용하기

네 번째 방법은 파일 객체를 `for` 문과 함께 사용하는 방법이다.

```
1 f = open("C:/doit/새 파일.txt", 'r')
2 for line in f:
3     print(line)
4 f.close()
```

파일 객체 (`f`)는 기본적으로 위와 같이 `for` 문과 함께 사용하여 파일을 줄 단위로 읽을 수 있다.

파일에 새로운 내용 추가하기

쓰기 모드 ('w')로 파일을 열 때 이미 존재하는 파일을 열면 그 파일의 내용이 모두 사라지게 된다. 하지만 원래 있던 값을 유지하면서 단지 새로운 값만 추가해야 할 경우도 있다. 이런 경우에는 파일을 추가 모드 ('a')로 열면 된다. IDLE 에디터로 다음 소스 코드를 작성해 보자.

```
1 # adddata.py
2 f = open("C:/doit/새 파일.txt", 'a')
```

```

3 for i in range(11, 20):
4     data = "%d번째 줄입니다.\n" % i
5     f.write(data)
6 f.close()

```

위 예는 새파일.txt 파일을 추가 모드 ('a') 로 열고 write 를 사용해서 결괏값을 기존 파일에 추가해 적는 예이다. 여기에서 추가 모드로 파일을 열었기 때문에 새파일.txt 파일이 원래 가지고 있던 내용 바로 다음부터 결괏값을 적기 시작한다.

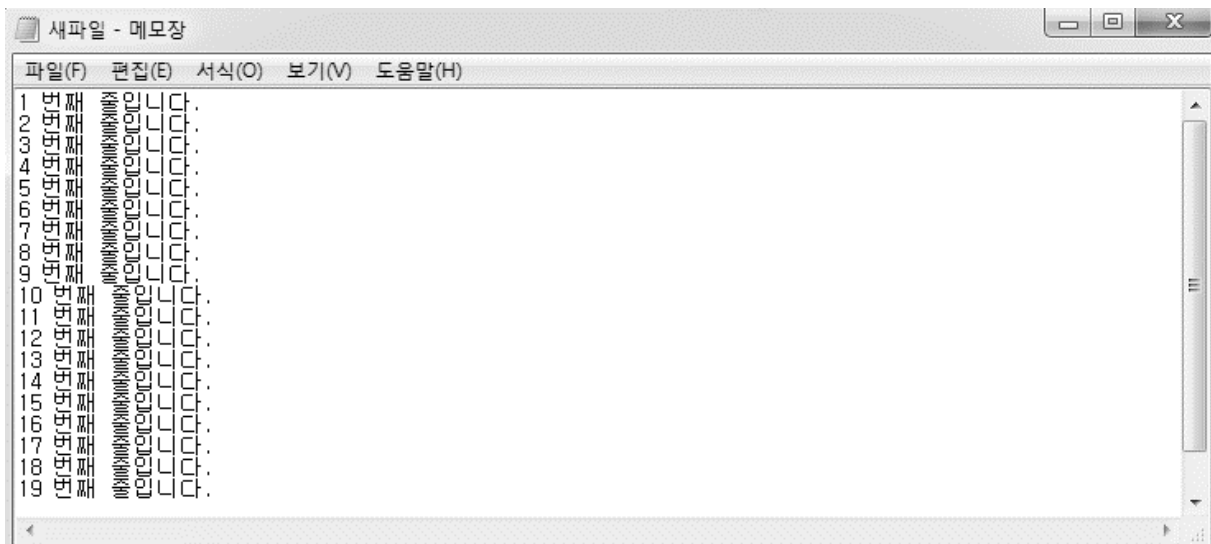
다음과 같이 작성한 코드를 실행해 보자.

```

1 C:\Users> cd C:\doit
2 C:\doit> python adddata.py
3 C:\doit>

```

새파일.txt 파일을 확인해 보면 원래 있던 내용 뒤에 새로운 내용이 추가되었음을 확인할 수 있다.



with 문과 함께 사용하기

지금까지 살펴본 예제를 보면 항상 다음과 같은 방식으로 파일을 열고 닫아 왔다.

```

1 f = open("foo.txt", 'w')
2 f.write("Life is too short, you need python")
3 f.close()

```

파일을 열면 (open) 항상 닫아 (close) 주어야 한다. 하지만 이렇게 파일을 열고 닫는 것을 자동으로 처리할 수 있다면 편리하지 않을까? 파이썬의 with 문이 바로 이런 역할을 해준다. 다음 예는 with 문을 사용해서 위 예제를 다시 작성한 모습이다.

```
1 with open("foo.txt", "w") as f:  
2     f.write("Life is too short, you need python")
```

위와 같이 **with** 문을 사용하면 **with** 블록 (**with** 문에 속해있는 문장) 을 벗어나는 순간 열린 파일 객체 **f** 가 자동으로 **close** 된다.

04-4 프로그램의 입력과 출력

명령 프롬프트 (DOS) 를 사용해 본 독자라면 다음과 같은 명령어를 사용해 봤을 것이다.

```
1 C:\> type a.txt
```

위 **type** 명령어는 바로 뒤에 적힌 파일 이름을 인수로 받아 해당 파일의 내용을 출력해 주는 명령어이다. 대부분의 명령 프롬프트에서 사용하는 명령어는 다음과 같이 인수를 전달하여 프로그램을 실행하는 방식을 따른다. 이러한 기능을 파이썬 프로그램에도 적용할 수가 있다.

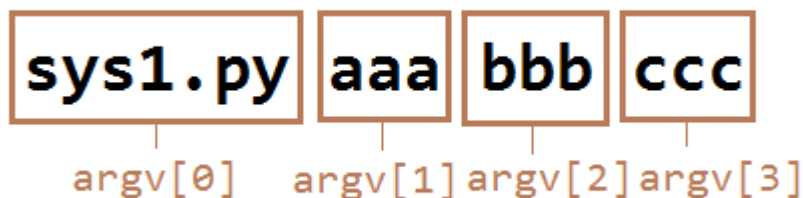
명령어 [인수 1 인수 2 ...]

파이썬에서는 **sys** 모듈을 사용하여 프로그램에 인수를 전달할 수 있다. **sys** 모듈을 사용하려면 아래 예의 **import sys**처럼 **import** 명령어를 사용해야 한다.

모듈을 사용하고 만드는 방법에 대해서는 05 장에서 자세히 다룰 것이다.

```
1 #sys1.py
2 import sys
3
4 args = sys.argv[1:]
5 for i in args:
6     print(i)
```

위 예는 프로그램 실행시 전달받은 인수를 **for** 문을 사용해 차례대로 하나씩 출력하는 예이다. **sys** 모듈의 **argv** 는 프로그램 실행시 전달된 인수를 의미한다. 즉 다음과 같이 입력했다면 **argv[0]** 은 파일 이름 **sys1.py** 가 되고 **argv[1]** 부터는 뒤에 따라오는 인수가 차례로 **argv** 의 요소가 된다.



이 프로그램을 `C:\doit` 디렉터리에 저장한 후 인수를 전달하여 실행하면 다음과 같은 결과를 볼수 있다.

명령창을 실행하고 `cd C:\doit` 명령을 실행하여 `C:\doit` 디렉터리로 이동한 후에 다음 명령을 실행하자.

```
1 C:\doit>python sys1.py aaa bbb ccc
2 aaa
3 bbb
4 ccc
```

위 예를 응용하여 다음과 같은 간단한 프로그램을 만들어 보자.

```
1 #sys2.py
2 import sys
3 args = sys.argv[1:]
4 for i in args:
5     print(i.upper(), end=' ')
```

문자열 관련 함수인 `upper()`를 사용하여 프로그램 실행시 전달된 인수를 모두 대문자로 바꾸어 주는 간단한 프로그램이다. 명령프롬프트 창에서 다음과 같이 실행해 보자.

`sys2.py` 파일이 `C:\doit` 디렉터리 안에 있어야만 한다.

```
1 C:\doit>python sys2.py life is too short, you need python
```

출력 결과는 다음과 같다.

```
1 LIFE IS TOO SHORT, YOU NEED PYTHON
```

04 장 연습문제

Q1. 홀수 짝수 판별하기

주어진 자연수가 홀수인지 짝수인지 판별해 주는 함수 (is_odd) 를 작성해 보자.

Q2. 모든 입력의 평균 값 구하기

입력으로 들어오는 모든 수의 평균 값을 계산해 주는 함수를 작성해 보자. (단 입력으로 들어오는 수의 개수는 정해져 있지 않다.)

평균 값을 구할 때 len 함수를 사용해 보자.

Q3. 프로그램 오류 수정하기 1

다음은 두 개의 숫자를 입력받아 더하여 돌려주는 프로그램이다.

```
1 input1 = input("첫 번째 숫자를 입력하세요:")
2 input2 = input("두 번째 숫자를 입력하세요:")
3
4 total = input1 + input2
5 print("두 수의 합은 %s 입니다" % total)
```

이 프로그램을 수행해 보자.

```
1 첫 번째 숫자를 입력하세요:3
2 두 번째 숫자를 입력하세요:6
3 두 수의 합은 36 입니다
```

3과 6을 입력했을 때 9가 아닌 36이라는 결괏값을 돌려주었다. 이 프로그램의 오류를 수정해 보자.

int 함수를 사용해 보자.

Q4. 출력 결과가 다른것은?

다음 중 출력 결과가 다른 것 한 개를 골라 보자.

1. `print("you""need""python")`
2. `print("you"+"need"+"python")`
3. `print("you", "need", "python")`
4. `print("".join(["you", "need", "python"]))`

Q5. 프로그램 오류 수정하기 2

다음은 “test.txt” 라는 파일에 “Life is too short” 문자열을 저장한 후 다시 그 파일을 읽어서 출력하는 프로그램이다.

```
1 f1 = open("test.txt", 'w')
2 f1.write("Life is too short")
3
4 f2 = open("test.txt", 'r')
5 print(f2.read())
```

이 프로그램은 우리가 예상한 “Life is too short” 라는 문장을 출력하지 않는다. 우리가 예상한 값을 출력할 수 있도록 프로그램을 수정해 보자.

Q6. 사용자 입력 저장하기

사용자의 입력을 파일 (test.txt) 에 저장하는 프로그램을 작성해 보자. (단 프로그램을 다시 실행하더라도 기존에 작성한 내용을 유지하고 새로 입력한 내용을 추가해야 한다.)

Q7. 파일의 문자열 바꾸기

다음과 같은 내용을 지닌 파일 test.txt 가 있다. 이 파일의 내용 중 “java” 라는 문자열을 “python” 으로 바꾸어서 저장해 보자.

```
1 Life is too short
2 you need java
```

replace 함수를 사용해 보자.

Q8. 입력값을 모두 더하여 출력하기

다음과 같이 실행할 때 입력값을 모두 더하여 출력하는 스크립트 (`C:\doit\myargv.py`) 를 작성해 보자.

```
1 C:\> cd doit
2 C:\doit> python myargv.py 1 2 3 4 5 6 7 8 9 10
3 55
```

sys.argv 를 사용해 보자.

05 장 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

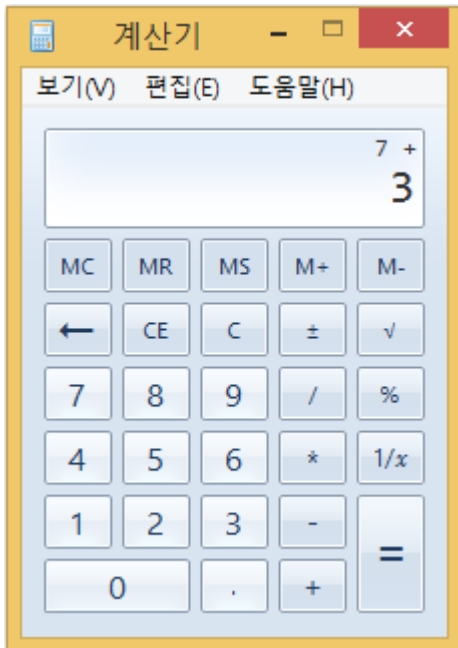
초보 개발자들에게 클래스 (class) 는 넘기 힘든 장벽과도 같은 존재이다. 독자들 중에도 클래스라는 단어를 처음 접하는 이들도 있을 것이다. 그러면 도대체 클래스가 무엇인지, 클래스가 왜 필요한지 아주 기초적인 것부터 차근차근 함께 알아보자.

클래스는 왜 필요한가?

프로그래머들이 가장 많이 사용하는 프로그래밍 언어 중 하나인 C 언어에는 클래스가 없다. 이 말은 굳이 클래스가 없어도 프로그램을 충분히 만들 수 있다는 뜻이다. 파이썬으로 잘 만든 프로그램을 살펴봐도 클래스를 사용하지 않고 작성한 것들이 상당히 많다. 클래스는 지금까지 공부한 함수나 자료형처럼 프로그램 작성을 위해 꼭 필요한 요소는 아니다.

하지만 프로그램을 작성할 때 클래스를 적재적소에 사용하면 프로그래머가 얻을 수 있는 이익은 상당하다. 예제를 통해 한번 생각해 보자.

여러분 모두 계산기를 사용해 보았을 것이다. 계산기에 숫자 3 을 입력하고 + 기호를 입력한 후 4 를 입력하면 결괏값으로 7 을 보여 준다. 다시 한 번 + 기호를 입력한 후 3 을 입력하면 기존 결괏값 7 에 3 을 더해 10 을 보여 준다. 즉 계산기는 이전에 계산한 결괏값을 항상 메모리 어딘가에 저장하고 있어야 한다.



계산기는 이전에 계산한 결괏값을 기억하고 있어야 한다.

이런 내용을 우리가 앞에서 익힌 함수를 이용해 구현해 보자. 계산기의 “더하기” 기능을 구현한 파이썬 코드는 다음과 같다.

```
1 result = 0
2
3 def add(num):
4     global result
5     result += num
6     return result
7
8 print(add(3))
9 print(add(4))
```

add 함수는 입력 값을 이전에 계산한 결괏값에 더한 후 돌려주는 함수이다.

이전에 계산한 결괏값을 유지하기 위해서 **result** 전역 변수 (**global**) 를 사용했다. 프로그램을 실행하면 예상한 대로 다음과 같은 결괏값이 출력된다.

```
1 3
2 7
```

그런데 만일 한 프로그램에서 2 대의 계산기가 필요한 상황이 발생하면 어떻게 해야 할까? 각 계산기는 각각의 결괏값을 유지해야 하기 때문에 위와 같이 **add** 함수 하나만으로는 결괏값을 따로 유지할 수 없다.

이런 상황을 해결하려면 다음과 같이 함수를 각각 따로 만들어야 한다.


```
1 result1 = 0
2 result2 = 0
3
4 def add1(num):
5     global result1
6     result1 += num
7     return result1
8
9 def add2(num):
10    global result2
11    result2 += num
12    return result2
13
14 print(add1(3))
15 print(add1(4))
16 print(add2(3))
17 print(add2(7))
```

똑같은 일을 하는 `add1` 과 `add2` 함수를 만들었고 각 함수에서 계산한 결괏값을 유지하면서 저장하는 전역 변수 `result1`, `result2` 가 필요하게 되었다.

결괏값은 다음과 같이 의도한 대로 출력된다.

```
1 3
2 7
3 3
4 10
```

계산기 1의 결괏값이 계산기 2에 아무 영향을 끼치지 않음을 확인할 수 있다. 하지만 계산기가 3개, 5개, 10개로 점점 더 많이 필요해진다면 어떻게 해야 할까? 그때마다 전역 변수와 함수를 추가할 것인가? 여기에 계산기마다 빼기나 곱하기와 같은 기능을 추가해야 한다면 상황은 점점 더 어려워질 것이다.

아직 클래스에 대해 배우지 않았지만, 위와 같은 경우에 클래스를 사용하면 다음과 같이 간단하게 해결할 수 있다.

다음 예시 클래스를 아직은 이해하지 못해도 좋다. 곧 자세하게 배울 것이다. 여기에서는 클래스 개념만 어렵듯이 이해하면 된다.

```
1 class Calculator:
2     def __init__(self):
3         self.result = 0
4
5     def add(self, num):
6         self.result += num
7         return self.result
8
9 cal1 = Calculator()
10 cal2 = Calculator()
11
12 print(cal1.add(3))
```

```
13 print(cal1.add(4))
14 print(cal2.add(3))
15 print(cal2.add(7))
```

프로그램을 실행하면 함수 2 개를 사용했을 때와 동일한 결과가 출력된다.

```
1 3
2 7
3 3
4 10
```

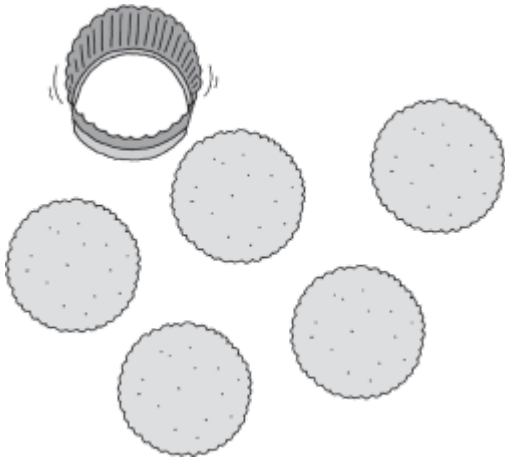
Calculator 클래스로 만든 별개의 계산기 **cal1**, **cal2**(파이썬에서는 이것을 객체라고 부른다) 가 각각의 역할을 수행한다. 그리고 계산기 (**cal1**, **cal2**) 의 결괏값 역시 다른 계산기의 결괏값과 상관없이 독립적인 값을 유지한다. 클래스를 사용하면 계산기 대수가 늘어나더라도 객체를 생성만 하면 되기 때문에 함수를 사용하는 경우와 달리 매우 간단해진다. 만약 빼기 기능을 더하려면 **Calculator** 클래스에 다음과 같은 빼기 기능 함수를 추가해 주면 된다.

```
1 class Calculator:
2     def __init__(self):
3         self.result = 0
4
5     def add(self, num):
6         self.result += num
7         return self.result
8
9     def sub(self, num):
10        self.result -= num
11        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 그림을 보자.



과자를 만드는 과자 틀과 그것을 사용해 만든 과자이다.

- 과자 틀 → 클래스 (class)
- 과자 틀에 의해서 만들어진 과자 → 객체 (object)

여기에서 설명할 클래스는 과자 틀과 비슷하다. 클래스 (class) 란 똑같은 무엇인가를 계속해서 만들어 낼 수 있는 설계 도면이고 (과자 틀), 객체 (object) 란 클래스로 만든 피조물 (과자 틀을 사용해 만든 과자) 을 뜻한다.

클래스로 만든 객체에는 중요한 특징이 있다. 바로 객체마다 고유한 성격을 가진다는 것이다. 과자 틀로 만든 과자에 구멍을 뚫거나 조금 베어 먹더라도 다른 과자에는 아무 영향이 없는 것과 마찬가지로 동일한 클래스로 만든 객체들은 서로 전혀 영향을 주지 않는다.

다음은 파이썬 클래스의 가장 간단한 예이다.

```
1 >>> class Cookie:
2 >>>     pass
```

위의 클래스는 아무 기능도 갖고 있지 않은 껍질뿐인 클래스이다. 하지만 이렇게 껍질뿐인 클래스도 객체를 생성하는 기능이 있다. “과자 틀” 로 “과자” 를 만드는 것처럼 말이다.

객체는 클래스로 만들며 1 개의 클래스는 무수히 많은 객체를 만들어 낼 수 있다. 위에서 만든 **Cookie** 클래스의 객체를 만드는 방법은 다음과 같다.

```
1 >>> a = Cookie()
2 >>> b = Cookie()
```

Cookie() 의 결과값을 돌려받은 **a** 와 **b** 가 바로 객체이다. 마치 함수를 사용해서 그 결과값을 돌려받는 모습과 비슷하다.

객체와 인스턴스의 차이

클래스로 만든 객체를 인스턴스라고도 한다. 그렇다면 객체와 인스턴스의 차이는 무엇일까? 이렇게 생각해 보자. `a = Cookie()` 이렇게 만든 `a` 는 객체이다. 그리고 `a` 객체는 `Cookie` 의 인스턴스이다. 즉 인스턴스라는 말은 특정 객체 (`a`) 가 어떤 클래스 (`Cookie`) 의 객체인지를 관계 위주로 설명할 때 사용한다. “`a` 는 인스턴스” 보다는 “`a` 는 객체” 라는 표현이 어울리며 “`a` 는 `Cookie` 의 객체” 보다는 “`a` 는 `Cookie` 의 인스턴스” 라는 표현이 훨씬 잘 어울린다.

사칙연산 클래스 만들기

“백견 (見) 이 불여 일타 (打)” 라고 했다. 클래스를 직접 만들며 배워 보자. 여기에서는 사칙연산을 하는 클래스를 만들어 볼 것이다. 사칙연산은 더하기, 빼기, 나누기, 곱하기를 말한다.

클래스를 어떻게 만들지 먼저 구상하기

클래스는 무작정 만드는 것보다 클래스로 만든 객체를 중심으로 어떤 식으로 동작하게 할것인지 미리 구상을 한 후에 생각한 것들을 하나씩 해결하면서 완성해 나가는 것이 좋다.

사칙연산을 가능하게 하는 `FourCal` 클래스가 다음처럼 동작한다고 가정해 보자.

먼저 `a = FourCal()`를 입력해서 `a` 라는 객체를 만든다.

```
1 >>> a = FourCal()
```

그런 다음 `a.setdata(4, 2)`처럼 입력해서 숫자 4와 2를 `a`에 지정해 주고

```
1 >>> a.setdata(4, 2)
```

`a.add()`를 수행하면 두 수를 합한 결과 (`4 + 2`)를 리턴하고

```
1 >>> a.add()
2 6
```

`a.mul()`을 수행하면 두 수를 곱한 결과 (`4 * 2`)를 리턴하고

```
1 >>> a.mul()
2 8
```

`a.sub()`를 수행하면 두 수를 뺀 결과 (`4 - 2`)를 리턴하고

```
1 >>> a.sub()
2 2
```

`a.div()`를 수행하면 두 수를 나눈 결과 (`4 / 2`)를 리턴한다.

```
1 >>> a.div()
2 2
```

이렇게 동작하는 `FourCal` 클래스를 만드는 것이 바로 우리의 목표이다.

클래스 구조 만들기

자, 그러면 지금부터 앞에서 구상한 것처럼 동작하는 클래스를 만들어 보자. 제일 먼저 할 일은 `a = FourCal()`처럼 객체를 만들 수 있게 하는 것이다. 일단은 아무 기능이 없어도 되기 때문에 매우 간단하게 만들 수 있다. 다음을 따라 해 보자.

```
1 >>> class FourCal:
2 ...     pass
3 ...
4 >>>
```

우선 대화형 인터프리터에서 `pass`란 문장만을 포함한 `FourCal` 클래스를 만든다. 현재 상태에서 `FourCal` 클래스는 아무 변수나 함수도 포함하지 않지만 우리가 원하는 객체 `a`를 만들 수 있는 기능은 가지고 있다. 확인해 보자.

`pass`는 아무것도 수행하지 않는 문법으로 임시로 코드를 작성할 때 주로 사용한다.

```
1 >>> a = FourCal()
2 >>> type(a)
3 <class '__main__.FourCal'>
```

위와 같이 `a = FourCal()`로 `a` 객체를 먼저 만들고 그다음에 `type(a)`로 `a` 객체가 어떤 타입인지 알아보았다. 역시 객체 `a`가 `FourCal` 클래스의 인스턴스임을 알 수 있다.

`type` 함수는 파이썬이 자체로 가지고 있는 내장 함수로 객체의 타입을 출력한다.

객체에 숫자 지정할 수 있게 만들기

하지만 생성된 객체 `a`는 아직 아무런 기능도 하지 못한다. 이제 더하기, 나누기, 곱하기, 빼기 등의 기능을 하는 객체를 만들어야 한다. 그런데 이러한 기능을 갖춘 객체를 만들려면 우선 `a` 객체에 사칙연산을 할 때 사용할 2개의 숫자를 먼저 알려주어야 한다. 다음과 같이 연산을 수행할 대상 (`4, 2`)을 객체에 지정할 수 있게 만들어 보자.

```
1 >>> a.setdata(4, 2)
```

위 문장이 동작하려면 다음과 같이 코드를 수정해야 한다.

```

1 >>> class FourCal:
2 ...     def setdata(self, first, second):
3 ...         self.first = first
4 ...         self.second = second
5 ...
6 >>>

```

앞에서 만든 `FourCal` 클래스에서 `pass` 문장을 삭제하고 그 대신 `setdata` 함수를 만들었다. 클래스 안에 구현된 함수는 다른 말로 **메서드 (Method)** 라고 부른다. 앞으로 클래스 내부의 함수는 항상 메서드라고 표현할 테니 메서드라는 용어를 기억해 두자.

일반적인 함수를 만들 때 다음과 같이 작성한다.

```

1 def 함수명(매개변수):
2     수행할 문장
3     ...

```

메서드도 클래스에 포함되어 있다는 점만 제외하면 일반 함수와 다를 것이 없다.

`setdata` 메서드를 다시 보면 다음과 같다.

```

1 def setdata(self, first, second): # 메서드의 매개변수
2     self.first = first           # 메서드의 수행문
3     self.second = second         # 메서드의 수행문

```

setdata 메서드의 매개변수

`setdata` 메서드는 매개변수로 `self`, `first`, `second` 3 개의 입력값을 받는다. 그런데 일반 함수와는 달리 메서드의 첫 번째 매개변수 `self` 는 특별한 의미를 가진다.

다음과 같이 `a` 객체를 만들고 `a` 객체를 통해 `setdata` 메서드를 호출해 보자.

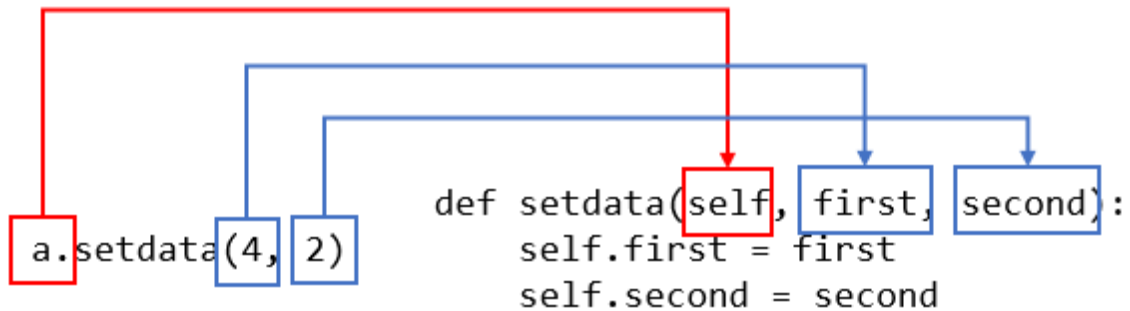
```

1 >>> a = FourCal()
2 >>> a.setdata(4, 2)

```

객체를 통해 클래스의 메서드를 호출하려면 `a.setdata(4, 2)` 와 같이 도트 (.) 연산자를 사용해야 한다.

그런데 뭔가 좀 이상하지 않은가? `setdata` 메서드에는 `self`, `first`, `second` 총 3 개의 매개변수가 필요한데 실제로는 `a.setdata(4, 2)` 처럼 2 개 값만 전달했다. 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 객체를 호출할 때 입력한 값이 메서드에 어떻게 전달되는지 쉽게 이해할 수 있을 것이다.



파이썬 메서드의 첫 번째 매개변수 이름은 관례적으로 **self** 를 사용한다. 객체를 호출할 때 호출한 객체 자신이 전달되기 때문에 **self** 라는 이름을 사용한 것이다. 물론 **self** 말고 다른 이름을 사용해도 상관없다.

메서드의 첫 번째 매개변수 **self** 를 명시적으로 구현하는 것은 파이썬만의 독특한 특징이다. 예를 들어 자바 같은 언어는 첫 번째 매개변수 **self** 가 필요없다.

메서드의 또 다른 호출 방법

잘 사용하지는 않지만 다음과 같이 클래스를 통해 메서드를 호출하는 것도 가능하다.

```
1 >>> a = FourCal()
2 >>> FourCal.setdata(a, 4, 2)
```

위와 같이 **클래스이름.메서드** 형태로 호출할 때는 객체 **a** 를 첫 번째 매개변수 **self** 에 꼭 전달해 주어야 한다. 반면에 다음처럼 **객체.메서드** 형태로 호출할 때는 **self** 를 반드시 생략해서 호출해야 한다.

```
1 >>> a = FourCal()
2 >>> a.setdata(4, 2)
```

setdata 메서드의 수행문

이제 **setdata** 메서드의 수행문에 대해 알아보자.

```
1 def setdata(self, first, second): # 메서드의 매개변수
2     self.first = first           # 메서드의 수행문
3     self.second = second         # 메서드의 수행문
```

a.setdata(4, 2)처럼 호출하면 **setdata** 메서드의 매개변수 **first, second** 에는 각각 값 4 와 2 가 전달되어 **setdata** 메서드의 수행문은 다음과 같이 해석된다.

```
1 self.first = 4
2 self.second = 2
```

self 는 전달된 객체 **a** 이므로 다시 다음과 같이 해석된다.

```
1 a.first = 4
2 a.second = 2
```

`a.first = 4` 라는 문장이 수행되면 `a` 객체에 객체변수 `first` 가 생성되고 4 라는 값이 저장된다. 마찬가지로 `a.second = 2` 라는 문장이 수행되면 `a` 객체에 객체변수 `second` 가 생성되고 2 라는 값이 저장된다.

객체에 생성되는 객체만의 변수를 객체변수라고 부른다.

다음과 같이 확인해 보자.

```
1 >>> a = FourCal()
2 >>> a.setdata(4, 2)
3 >>> a.first
4 4
5 >>> a.second
6 2
```

`a` 객체에 객체변수 `first` 와 `second` 가 생성되었음을 확인할 수 있다.

이번에는 다음과 같이 `a, b` 객체를 만들어 보자.

```
1 >>> a = FourCal()
2 >>> b = FourCal()
```

그리고 `a` 객체의 객체변수 `first` 를 다음과 같이 생성한다.

```
1 >>> a.setdata(4, 2)
2 >>> a.first
3 4
```

이번에는 `b` 객체의 객체변수 `first` 를 다음과 같이 생성한다.

```
1 >>> b.setdata(3, 7)
2 >>> b.first
3 3
```

자, 이제 여러분에게 아주 중요한 질문을 한 가지 하겠다. 위와 같이 진행하면 `b` 객체의 객체변수 `first` 에는 값 3 이 저장된다는 것을 확인할 수 있었다. 그렇다면 `a` 객체의 `first` 는 3 으로 변할까? 아니면 기존 값 4 를 유지할까?

다음과 같이 그 결과를 확인해 보자.

```
1 >>> a.first
2 4
```

`a` 객체의 `first` 값은 `b` 객체의 `first` 값에 영향받지 않고 원래 값을 유지하고 있음을 확인할 수 있다. 이 예제를 통해 여러분에게 강조하고 싶은 점이 바로 이것이다. 클래스로 만든 객체의 객체변수는 다른 객체의 객

체변수에 상관없이 독립적인 값을 유지한다. 클래스에서는 이 부분을 이해하는 것이 가장 중요하다.

다음은 현재까지 완성된 **FourCal** 클래스이다.

```
1 >>> class FourCal:
2 ...     def setdata(self, first, second):
3 ...         self.first = first
4 ...         self.second = second
5 ...
6 >>>
```

지금까지 살펴본 내용이 바로 위 4 줄을 설명하기 위한 것이었다. 위에서 설명한 것들이 이해가 되지 않는다면 다시 한 번 읽어 보기 바란다. 이 부분을 이해하지 못하면 다음으로 넘어갈 수 없기 때문이다.

IDLE 에디터에서 FourCal 클래스 작성하기

앞의 **FourCal** 클래스는 대화형 인터프리터에서 작성하여 테스트하기에 좀 불편하다. 왜냐하면 클래스를 변경할 때마다 반복적으로 **FourCal** 클래스의 코드를 대화형 인터프리터에 입력해야 하기 때문이다. 따라서 여러분은 **FourCal** 클래스를 IDLE 에디터로 작성하여 실습하기를 권한다.

다만, 이 책은 설명의 편의를 위해 인터프리터를 사용하는 예제로 설명할 것이다.

지금까지 작성한 **FourCal** 클래스를 IDLE 에디터로 작성하고 실행하려면 다음과 같이 따라하자. 먼저 IDLE 에디터를 열고 **fourcal.py** 파일을 다음과 같이 작성하여 **c:/doit** 디렉터리에 저장하자.

[파일명: **c:/doit/fourcal.py**]

```
1 class FourCal:
2     def setdata(self, first, second):
3         self.first = first
4         self.second = second
5
6
7 a = FourCal()
8 a.setdata(4, 2)
9 print(a.first) # a.first의 값을 확인하기 위해서 print 함수를 사용한다.
```

위 예제와 같이 **FourCal** 클래스를 작성하고 실행할 문장은 클래스 아래에 적으면 된다. 대화형 인터프리터를 사용할 경우 **a.first** 값을 확인하기 위해서는 **>>> a.first**와 같이 간편하게 확인할 수 있지만 에디터를 사용할 때는 **print(a.first)** 와 같이 **print** 함수를 사용해야 한다. 작성한 **fourcal.py** 파일을 실행하려면 [Run -> Run Module]을 실행하거나 F5 키를 입력하면 된다. 그러면 IDLE 셸창에 다음과 같이 프로그램의 수행결과가 표시될 것이다.

```
1 ===== RESTART: C:/doit/fourcal.py =====
2 4
```

더하기 기능 만들기

자! 그럼 2 개의 숫자 값을 설정해 주었으니 2 개의 숫자를 더하는 기능을 방금 만든 클래스에 추가해 보자. 우리는 다음과 같이 더하기 기능을 갖춘 클래스를 만들어야 한다.

```
1 >>> a = FourCal()
2 >>> a.setdata(4, 2)
3 >>> a.add()
4 6
```

이 연산이 가능하도록 다음과 같이 `FourCal` 클래스를 만들어 보자.

```
1 >>> class FourCal:
2 ...     def setdata(self, first, second):
3 ...         self.first = first
4 ...         self.second = second
5 ...     def add(self):
6 ...         result = self.first + self.second
7 ...         return result
8 ...
9 >>>
```

새롭게 추가된 것은 `add` 메서드이다. 이제 클래스를 사용해 보자.

```
1 >>> a = FourCal()
2 >>> a.setdata(4, 2)
```

위와 같이 호출하면 앞에서 살펴보았듯이 `a` 객체의 `first`, `second` 객체변수에는 각각 값 4 와 2 가 저장될 것이다.

이제 `add` 메서드를 호출해 보자.

```
1 >>> a.add()
2 >>> 6
```

`a.add()` 라고 호출하면 `add` 메서드가 호출되어 값 6 이 출력될 것이다. 어떤 과정을 거쳐 값 6 이 출력되는지 `add` 메서드를 따로 떼어 내서 자세히 살펴보자.

```
1 def add(self):
2     result = self.first + self.second
3     return result
```

`add` 메서드의 매개변수는 `self` 이고 리턴 값은 `result` 이다. 리턴 값인 `result` 를 계산하는 부분은 다음과 같다.

```
1 result = self.first + self.second
```

`a.add()` 와 같이 `a` 객체에 의해 `add` 메서드가 수행되면 `add` 메서드의 `self` 에는 객체 `a` 가 자동으로 입력되므로 위 내용은 다음과 같이 해석된다.

```
1 result = a.first + a.second
```

위 내용은 `a.add()` 메서드 호출 전에 `a.setdata(4, 2)` 가 먼저 호출되어 `a.first = 4`, `a.second = 2` 라고 이미 설정되었기 때문에 다시 다음과 같이 해석된다.

```
1 result = 4 + 2
```

따라서 다음과 같이 `a.add()`를 호출하면 6을 리턴한다.

```
1 >>> a.add()
2 6
```

여기까지 모두 이해한 독자라면 클래스에 대해 80% 이상을 안 것이다. 파이썬의 클래스는 그다지 어렵지 않다.

곱하기, 빼기, 나누기 기능 만들기

이번에는 곱하기, 빼기, 나누기 등을 할 수 있게 프로그램을 만들어 보자.

```
1 >>> class FourCal:
2 ...     def setdata(self, first, second):
3 ...         self.first = first
4 ...         self.second = second
5 ...     def add(self):
6 ...         result = self.first + self.second
7 ...         return result
8 ...     def mul(self):
9 ...         result = self.first * self.second
10 ...        return result
11 ...    def sub(self):
12 ...        result = self.first - self.second
13 ...        return result
14 ...    def div(self):
15 ...        result = self.first / self.second
16 ...        return result
17 ...
18 >>>
```

`mul`, `sub`, `div` 모두 `add` 메서드에서 배운 것과 동일한 방법이니 따로 설명하지는 않겠다.

정말로 모든 것이 제대로 동작하는지 확인해 보자.

```
1 >>> a = FourCal()
2 >>> b = FourCal()
3 >>> a.setdata(4, 2)
4 >>> b.setdata(3, 8)
5 >>> a.add()
6 6
7 >>> a.mul()
```

```

8  8
9  >>> a.sub()
10 2
11 >>> a.div()
12 2
13 >>> b.add()
14 11
15 >>> b.mul()
16 24
17 >>> b.sub()
18 -5
19 >>> b.div()
20 0.375

```

여기까지 우리가 목표로 한 사칙연산 기능을 가진 클래스를 만들어 보았다.

생성자 (Constructor)

이번에는 우리가 만든 `FourCal` 클래스를 다음과 같이 사용해 보자.

```

1 >>> a = FourCal()
2 >>> a.add()
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "<stdin>", line 6, in add
6   AttributeError: 'FourCal' object has no attribute 'first'

```

`FourCal` 클래스의 인스턴스 `a` 에 `setdata` 메서드를 수행하지 않고 `add` 메서드를 먼저 수행하면 “`AttributeError: ‘FourCal’ object has no attribute ‘first’`” 오류가 발생한다. 왜냐하면 `setdata` 메서드를 수행해야 객체 `a` 의 객체변수 `first` 와 `second` 가 생성되기 때문이다.

이렇게 객체에 `first`, `second` 와 같은 초깃값을 설정해야 할 필요가 있을 때는 `setdata` 와 같은 메서드를 호출하여 초깃값을 설정하기보다는 생성자를 구현하는 것이 안전한 방법이다. 생성자 (Constructor) 란 객체가 생성될 때 자동으로 호출되는 메서드를 의미한다.

파이썬 메서드 이름으로 `__init__` 를 사용하면 이 메서드는 생성자가 된다. 다음과 같이 `FourCal` 클래스에 생성자를 추가해 보자.

`__init__` 메서드의 `init` 앞뒤로 붙은 `__` 는 언더스코어 (`_`) 두 개를 붙여 쓴 것이다.

```

1 >>> class FourCal:
2 ...     def __init__(self, first, second):
3 ...         self.first = first
4 ...         self.second = second
5 ...     def setdata(self, first, second):
6 ...         self.first = first
7 ...         self.second = second

```

```

8 ...     def add(self):
9 ...         result = self.first + self.second
10 ...        return result
11 ...     def mul(self):
12 ...         result = self.first * self.second
13 ...        return result
14 ...     def sub(self):
15 ...         result = self.first - self.second
16 ...        return result
17 ...     def div(self):
18 ...         result = self.first / self.second
19 ...        return result
20 ...
21 >>>

```

새롭게 추가된 생성자 `__init__` 메서드만 따로 떼어 내서 살펴보자.

```

1 def __init__(self, first, second):
2     self.first = first
3     self.second = second

```

`__init__` 메서드는 `setdata` 메서드와 이름만 다르고 모든 게 동일하다. 단 메서드 이름을 `__init__`으로 했기 때문에 생성자로 인식되어 객체가 생성되는 시점에 자동으로 호출되는 차이가 있다.

이제 다음처럼 예제를 수행해 보자.

```

1 >>> a = FourCal()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: __init__() missing 2 required positional arguments: 'first' and 'second'

```

`a = FourCal()`을 수행할 때 생성자 `__init__`이 호출되어 위와 같은 오류가 발생했다. 오류가 발생한 이유는 생성자의 매개변수 `first`와 `second`에 해당하는 값이 전달되지 않았기 때문이다.

위 오류를 해결하려면 다음처럼 `first`와 `second`에 해당되는 값을 전달하여 객체를 생성해야 한다.

```

1 >>> a = FourCal(4, 2)
2 >>>

```

위와 같이 수행하면 `__init__` 메서드의 매개변수에는 각각 다음과 같은 값이 전달된다.

매개변수	값
<code>self</code>	생성되는 객체
<code>first</code>	4
<code>second</code>	2

`__init__` 메서드도 다른 메서드와 마찬가지로 첫 번째 매개변수 `self` 에 생성되는 객체가 자동으로 전달된다는 점을 기억하자.

따라서 `__init__` 메서드가 호출되면 `setdata` 메서드를 호출했을 때와 마찬가지로 `first` 와 `second` 라는 객체변수가 생성될 것이다.

다음과 같이 객체변수의 값을 확인해 보자.

```
1 >>> a = FourCal(4, 2)
2 >>> a.first
3 4
4 >>> a.second
5 2
```

`add` 나 `div` 등의 메서드도 잘 동작하는지 확인해 보자.

```
1 >>> a = FourCal(4, 2)
2 >>> a.add()
3 6
4 >>> a.div()
5 2.0
```

이상 없이 잘 동작하는 것을 확인할 수 있다.

클래스의 상속

상속 (Inheritance) 이란 “물려받다” 라는 뜻으로, “재산을 상속받다” 라고 할 때의 상속과 같은 의미이다. 클래스에도 이 개념을 적용할 수 있다. 어떤 클래스를 만들 때 다른 클래스의 기능을 물려받을 수 있게 만드는 것이다. 이번에는 상속 개념을 사용하여 우리가 만든 `FourCal` 클래스에 `ab` (`a` 의 `b` 제곱) 을 구할 수 있는 기능을 추가해 보자.

앞에서 `FourCal` 클래스는 이미 만들어 놓았으므로 `FourCal` 클래스를 상속하는 `MoreFourCal` 클래스는 다음과 같이 간단하게 만들 수 있다.

```
1 >>> class MoreFourCal(FourCal):
2 ...     pass
3 ...
4 >>>
```

클래스를 상속하기 위해서는 다음처럼 클래스 이름 뒤 괄호 안에 상속할 클래스 이름을 넣어주면 된다.

`class 클래스 이름 (상속할 클래스 이름)`

`MoreFourCal` 클래스는 `FourCal` 클래스를 상속했으므로 `FourCal` 클래스의 모든 기능을 사용할 수 있어야 한다.

다음과 같이 확인해 보자.

```
1 >>> a = MoreFourCal(4, 2)
2 >>> a.add()
3 6
4 >>> a.mul()
5 8
6 >>> a.sub()
7 2
8 >>> a.div()
9 2
```

상속받은 `FourCal` 클래스의 기능을 모두 사용할 수 있음을 확인할 수 있다.

왜 상속을 해야 할까?

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 사용해야 한다.

이제 원래 목적인 `a` 의 `b` 제곱 (`ab`) 을 계산하는 `MoreFourCal` 클래스를 만들어 보자.

```
1 >>> class MoreFourCal(FourCal):
2 ...     def pow(self):
3 ...         result = self.first ** self.second
4 ...         return result
5 ...
6 >>>
```

`pass` 문장은 삭제하고 위와 같이 두 수의 거듭제곱을 구할 수 있는 `pow` 메서드를 추가했다. 그리고 다음과 같이 `pow` 메서드를 수행해 보자.

```
1 >>> a = MoreFourCal(4, 2)
2 >>> a.pow()
3 16
4 >>> a.add()
5 6
```

`MoreFourCal` 클래스로 만든 `a` 객체에 값 4 와 2 를 설정한 후 `pow` 메서드를 호출하면 4 의 2 제곱 (4²) 인 16 을 리턴하는 것을 확인할 수 있다. 상속받은 기능인 `add` 메서드도 역시 잘 동작한다.

상속은 `MoreFourCal` 클래스처럼 기존 클래스 (`FourCal`) 는 그대로 놔둔 채 클래스의 기능을 확장시킬 때 주로 사용한다.

메서드 오버라이딩

이번에는 `FourCal` 클래스를 다음과 같이 실행해 보자.

```
1 >>> a = FourCal(4, 0)
2 >>> a.div()
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5     result = self.first / self.second
6   ZeroDivisionError: division by zero
```

`FourCal` 클래스의 객체 `a`에 4와 0 값을 설정하고 `div` 메서드를 호출하면 4를 0으로 나누려고 하기 때문에 위와 같은 `ZeroDivisionError` 오류가 발생한다. 하지만 0으로 나눌 때 오류가 아닌 0을 리턴하도록 만들고 싶다면 어떻게 해야 할까?

다음과 같이 `FourCal` 클래스를 상속하는 `SafeFourCal` 클래스를 만들어 보자.

```
1 >>> class SafeFourCal(FourCal):
2 ...     def div(self):
3 ...         if self.second == 0: # 나누는 값이 0인 경우 0을 리턴하도록 수정
4 ...             return 0
5 ...         else:
6 ...             return self.first / self.second
7 ...
8 >>>
```

`SafeFourCal` 클래스는 `FourCal` 클래스에 있는 `div` 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 부모클래스의 메서드 대신 오버라이딩한 메서드가 호출된다.

`SafeFourCal` 클래스에 오버라이딩한 `div` 메서드는 나누는 값이 0인 경우에는 0을 리턴하도록 수정했다. 이제 다시 위에서 수행한 예제를 `FourCal` 클래스 대신 `SafeFourCal` 클래스를 사용하여 수행해 보자.

```
1 >>> a = SafeFourCal(4, 0)
2 >>> a.div()
3 0
```

`FourCal` 클래스와는 달리 `ZeroDivisionError`가 발생하지 않고 의도한 대로 0이 리턴되는 것을 확인할 수 있다.

클래스 변수

객체변수는 다른 객체들의 영향을 받지 않고 독립적으로 그 값을 유지한다는 점을 이미 알아보았다. 이번에는 객체변수와는 성격이 다른 클래스 변수에 대해 알아보자.

다음 클래스를 작성해 보자.

```
1 >>> class Family:
2 ...     lastname = "김"
3 ...
```

Family 클래스에 선언한 **lastname** 이 바로 클래스 변수이다. 클래스 변수는 클래스 안에 함수를 선언하는 것과 마찬가지로 클래스 안에 변수를 선언하여 생성한다.

이제 **Family** 클래스를 다음과 같이 사용해 보자.

```
1 >>> Family.lastname
2 김
```

클래스 변수는 위 예와 같이 **클래스이름.클래스변수**로 사용할 수 있다.

또는 다음과 같이 **Family** 클래스로 만든 객체를 통해서도 클래스 변수를 사용할 수 있다.

```
1 >>> a = Family()
2 >>> b = Family()
3 >>> a.lastname
4 김
5 >>> b.lastname
6 김
```

만약 **Family** 클래스의 **lastname** 을 다음과 같이 “박” 이라는 문자열로 바꾸면 어떻게 될까?

```
1 >>> Family.lastname = "박"
```

다음과 같이 확인해 보자.

```
1 >>> a.lastname
2 박
3 >>> b.lastname
4 박
```

클래스 변수 값을 변경했더니 클래스로 만든 객체의 **lastname** 값도 모두 변경된다는 것을 확인할 수 있다. 즉 클래스 변수는 클래스로 만든 모든 객체에 공유된다는 특징이 있다.

클래스 변수와 동일한 이름의 객체 변수를 생성하면?

위의 예제에서 `a.lastname`을 다음처럼 변경하면 어떻게 될까?

```
1 >>> a.lastname = "최"  
2 >>> a.lastname  
3 최
```

이렇게 하면 `Family` 클래스의 `lastname` 이 바뀌는 것이 아니라 `a` 객체에 `lastname` 이라는 객체 변수가 새롭게 생성된다. 즉, 객체 변수는 클래스 변수와 동일한 이름으로 생성할 수 있다.

`a.lastname` 객체 변수를 생성하더라도 `Family` 클래스의 `lastname` 과는 상관없다는 것을 다음과 같이 확인할 수 있다.

```
1 >>> Family.lastname  
2 박  
3 >>> b.lastname  
4 박
```

`Family` 클래스의 `lastname` 값은 변하지 않았다.

클래스 변수를 가장 늦게 설명하는 이유는 클래스에서 클래스 변수보다는 객체 변수가 훨씬 중요하기 때문이다. 실무 프로그래밍을 할 때도 클래스 변수보다는 객체 변수를 사용하는 비율이 훨씬 높다.

05-2 모듈

모듈이란 함수나 변수 또는 클래스를 모아 놓은 파이썬 파일이다. 모듈은 다른 파이썬 프로그램에서 불러와 사용할 수 있게끔 만든 파이썬 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. 여기에서는 모듈을 어떻게 만들고 사용할 수 있는지 알아보겠다.

모듈 만들기

모듈에 대해 자세히 살펴보기 전에 간단한 모듈을 한번 만들어 보자.

```
1 # mod1.py
2 def add(a, b):
3     return a + b
4
5 def sub(a, b):
6     return a-b
```

위와 같이 `add`와 `sub` 함수만 있는 파일 `mod1.py`를 만들고 `C:\doit` 디렉터리에 저장하자. 이 `mod1.py` 파일이 바로 모듈이다. 지금까지 에디터로 만든 파이썬 파일과 다르지 않다.

파이썬 확장자 `.py`로 만든 파이썬 파일은 모두 모듈이다.

모듈 불러오기

우리가 만든 `mod1.py` 파일, 즉 모듈을 파이썬에서 불러와 사용하려면 어떻게 해야 할까?

먼저 다음과 같이 명령 프롬프트 창을 열고 `mod1.py`를 저장한 디렉터리 (이 책에서는 `C:\doit`)로 이동한 다음 대화형 인터프리터를 실행해 보자.

```
1 C:\Users\pahkey>cd C:\doit
2 C:\doit>dir
3 ...
4 2014-09-23 오후 01:53 49 mod1.py
5 ...
```

```
6 C:\doit>python
7 >>>
```

반드시 `mod1.py` 파일을 저장한 `C:\doit` 디렉터리로 이동한 다음 예제를 진행해야 한다. 그래야만 대화형 인터프리터에서 `mod1.py` 모듈을 읽을 수 있다.

그리고 다음과 같이 따라 해 보자.

```
1 >>> import mod1
2 >>> print(mod1.add(3, 4))
3 7
4 >>> print(mod1.sub(4, 2))
5 2
```

`mod1.py` 모듈을 불러오기 위해 `import mod1`이라고 입력하였다. 실수로 `import mod1.py`로 입력하지 않도록 주의하자. `import` 는 이미 만들어 놓은 파이썬 모듈을 사용할 수 있게 해주는 명령어이다. `mod1.py` 파일에 있는 `add` 함수를 사용하기 위해서는 위 예의 `mod1.add`처럼 모듈 이름 뒤에 “.”(도트 연산자)를 붙이고 함수 이름을 쓰면 된다.

`import` 는 현재 디렉터리에 있는 파일이나 파이썬 라이브러리가 저장된 디렉터리에 있는 모듈만 불러올 수 있다. 파이썬 라이브러리는 파이썬을 설치할 때 자동으로 설치되는 파이썬 모듈을 말한다.

`import` 의 사용 방법은 다음과 같다.

```
1 import 모듈이름
```

여기에서 모듈 이름은 `mod1.py` 에서 `.py` 확장자를 제거한 `mod1` 만을 가리킨다.

때로는 `mod1.add`, `mod1.sub`처럼 쓰지 않고 `add`, `sub`처럼 모듈 이름 없이 함수 이름만 쓰고 싶은 경우도 있을 것이다. 그럴 때는 다음과 같이 사용하면 된다.

```
from 모듈이름 import 모듈함수
```

위 형식을 사용하면 위와 같이 모듈 이름을 붙이지 않고 바로 해당 모듈의 함수를 쓸 수 있다.

다음과 같이 따라 해 보자.

```
1 >>> from mod1 import add
2 >>> add(3, 4)
3 7
```

그런데 위와 같이 하면 `mod1.py` 파일의 `add` 함수만 사용할 수 있다. `add` 함수와 `sub` 함수를 둘 다 사용하고 싶다면 어떻게 해야 할까?

2 가지 방법이 있다.

```
1 from mod1 import add, sub
```

첫 번째 방법은 위와 같이 `from 모듈이름 import 모듈함수1, 모듈함수2`처럼 사용하는 것이다. 콤마로 구분하여 필요한 함수를 불러올 수 있다.

```
1 from mod1 import *
```

두 번째 방법은 위와 같이 `*` 문자를 사용하는 방법이다. 07 장에서 공부할 정규 표현식에서 `*` 문자는 “모든 것”이라는 뜻인데 파이썬에서도 마찬가지로 의미로 사용한다. 따라서 `from mod1 import *`는 `mod1` 모듈의 모든 함수를 불러와 사용하겠다는 뜻이다.

`mod1.py` 파일에는 함수가 2 개밖에 없기 때문에 위 2 가지 방법은 동일하게 적용된다.

if __name__ == “__main__”: 의 의미

이번에는 `mod1.py` 파일을 다음과 같이 변경해 보자.

```
1 # mod1.py
2 def add(a, b):
3     return a+b
4
5 def sub(a, b):
6     return a-b
7
8 print(add(1, 4))
9 print(sub(4, 2))
```

`add(1, 4)`와 `sub(4, 2)`의 결과를 출력하는 문장을 추가했다. 그리고 출력한 결과값을 확인하기 위해 `mod1.py` 파일을 다음과 같이 실행해 보자.

```
1 C:\doit>python mod1.py
2 5
3 2
```

예상한 대로 결과값이 잘 출력된다. 그런데 이 `mod1.py` 파일의 `add`와 `sub` 함수를 사용하기 위해 `mod1` 모듈을 `import` 할 때는 좀 이상한 문제가 생긴다. 명령 프롬프트 창에서 다음을 따라 해 보자.

```
1 C:\Users\pahkey> cd C:\doit
2 C:\doit> python
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>> import mod1
5 5
6 2
```

영뚱하게도 `import mod1`을 수행하는 순간 `mod1.py` 파일이 실행되어 결과값을 출력한다. 우리는 단지 `mod1.py` 파일의 `add`와 `sub` 함수만 사용하려고 했는데 말이다.

이러한 문제를 방지하려면 `mod1.py` 파일을 다음처럼 변경해야 한다.

```

1 # mod1.py
2 def add(a, b):
3     return a+b
4
5 def sub(a, b):
6     return a-b
7
8 if __name__ == "__main__":
9     print(add(1, 4))
10    print(sub(4, 2))

```

`if __name__ == "__main__"`을 사용하면 `C:\doit>python mod1.py`처럼 직접 이 파일을 실행했을 때는 `__name__ == "__main__"`이 참이 되어 if 문 다음 문장이 수행된다. 반대로 대화형 인터프리터나 다른 파일에서 이 모듈을 불러서 사용할 때는 `__name__ == "__main__"`이 거짓이 되어 if 문 다음 문장이 수행되지 않는다.

위와 같이 수정한 후 다시 대화형 인터프리터를 열고 실행해 보자.

```

1 >>> import mod1
2 >>>

```

아무 결과값도 출력되지 않는 것을 확인할 수 있다.

`__name__` 변수란?

파이썬의 `__name__` 변수는 파이썬이 내부적으로 사용하는 특별한 변수 이름이다. 만약 `C:\doit>python mod1.py`처럼 직접 `mod1.py` 파일을 실행할 경우 `mod1.py`의 `__name__` 변수에는 `__main__` 값이 저장된다. 하지만 파이썬 셸이나 다른 파이썬 모듈에서 `mod1`을 import 할 경우에는 `mod1.py`의 `__name__` 변수에는 `mod1.py`의 모듈 이름 값 `mod1`이 저장된다.

```

1 >>> import mod1
2 >>> mod1.__name__
3 'mod1'

```

클래스나 변수 등을 포함한 모듈

지금까지 살펴본 모듈은 함수만 포함했지만 클래스나 변수 등을 포함할 수도 있다. 다음 프로그램을 작성해 보자.

```

1 # mod2.py
2 PI = 3.141592
3
4 class Math:
5     def solv(self, r):

```

```
6         return PI * (r ** 2)
7
8     def add(a, b):
9         return a+b
```

이 파일은 원의 넓이를 계산하는 `Math` 클래스와 두 값을 더하는 `add` 함수 그리고 원주율 값에 해당되는 `PI` 변수처럼 클래스, 함수, 변수 등을 모두 포함하고 있다.

파일 이름을 `mod2.py` 로 하고 `C:\doit` 디렉터리에 저장하자. 대화형 인터프리터를 열고 다음과 같이 따라 해 보자.

```
1 C:\Users\pahkey> cd C:\doit
2 C:\doit> python
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>> import mod2
5 >>> print(mod2.PI)
6 3.141592
```

위 예에서 볼 수 있듯이 `mod2.PI`처럼 입력해서 `mod2.py` 파일에 있는 `PI` 변수 값을 사용할 수 있다.

```
1 >>> a = mod2.Math()
2 >>> print(a.solv(2))
3 12.566368
```

위 예는 `mod2.py` 에 있는 `Math` 클래스를 사용하는 방법을 보여 준다. 위 예처럼 모듈 안에 있는 클래스를 사용하려면 “.”(도트 연산자)로 클래스 이름 앞에 모듈 이름을 먼저 입력해야 한다.

```
1 >>> print(mod2.add(mod2.PI, 4.4))
2 7.541592
```

`mod2.py` 에 있는 `add` 함수 역시 당연히 사용할 수 있다.

다른 파일에서 모듈 불러오기

지금까지는 만들어 놓은 모듈 파일을 사용하기 위해 대화형 인터프리터만 사용했다. 이번에는 다른 파이썬 파일에서 이전에 만들어 놓은 모듈을 불러와서 사용하는 방법에 대해 알아보자. 여기에서는 조금 전에 만든 모듈인 `mod2.py` 파일을 다른 파이썬 파일에서 불러와 사용할 것이다.

먼저 에디터로 `C:\doit\modtest.py` 파일을 다음과 같이 작성하자.

```
1 # modtest.py
2 import mod2
3 result = mod2.add(3, 4)
4 print(result)
```

위에서 볼 수 있듯이 다른 파이썬 파일에서도 `import mod2`로 `mod2` 모듈을 불러와서 사용할 수 있

다. 대화형 인터프리터에서 한 것과 마찬가지로 방법이다. 위 예제가 정상적으로 실행되기 위해서는 `modtest.py` 파일과 `mod2.py` 파일이 동일한 디렉터리 (`C:\doit`)에 있어야 한다.

모듈을 불러오는 또 다른 방법

우리는 지금껏 해당 모듈이 있는 디렉터리로 이동한 후에야 그 모듈을 사용할 수 있었다. 이번에는 모듈을 저장한 디렉터리로 이동하지 않고 모듈을 불러와서 사용하는 방법에 대해 알아보자.

먼저 다음과 같이 이전에 만든 `mod2.py` 파일을 `C:\doit\mymod`로 이동시킨다.

```
1 C:\Users\pahkey>cd C:\doit
2 C:\doit>mkdir mymod
3 C:\doit>move mod2.py mymod
4      1개 파일을 이동했습니다.
```

그리고 다음 예를 따라 해 보자.

`sys.path.append` 사용하기

먼저 파이썬 셸을 실행한 후 `sys` 모듈을 불러온다.

```
1 C:\doit>python
2 >>> import sys
```

`sys` 모듈은 파이썬을 설치할 때 함께 설치되는 라이브러리 모듈이다. `sys`에 대해서는 뒤에서 자세하게 다룰 것이다. 이 `sys` 모듈을 사용하면 파이썬 라이브러리가 설치되어 있는 디렉터리를 확인할 수 있다.

다음과 같이 입력해 보자.

```
1 >>> sys.path
2 ['', 'C:\\Windows\\SYSTEM32\\python311.zip', 'c:\\Python311\\DLLs',
3  'c:\\Python311\\lib', 'c:\\Python311', 'c:\\Python311\\lib\\site-packages']
```

`sys.path`는 파이썬 라이브러리가 설치되어 있는 디렉터리를 보여 준다. 만약 파이썬 모듈이 위 디렉터리에 들어 있다면 모듈이 저장된 디렉터리로 이동할 필요 없이 바로 불러서 사용할 수 있다. 그렇다면 `sys.path`에 `C:\doit\mymod` 디렉터를 추가하면 아무 곳에서나 불러 사용할 수 있지 않을까?

당연하다. `sys.path`는 리스트이므로 우리는 다음과 같이 할 수 있다.

```
1 >>> sys.path.append("C:/doit/mymod")
2 >>> sys.path
3 ['', 'C:\\Windows\\SYSTEM32\\python311.zip', 'c:\\Python311\\DLLs',
4  'c:\\Python311\\lib', 'c:\\Python311', 'c:\\Python311\\lib\\site-packages',
5  'C:/doit/mymod']
6 >>>
```


`sys.path.append` 를 사용해서 `C:/doit/mymod`라는 디렉토리를 `sys.path` 에 추가했다. 그리고 다시 `sys.path`를 출력해 보니 가장 마지막에 `C:/doit/mymod` 디렉터리가 추가되었다.

자, 실제로 모듈을 불러와서 사용할 수 있는지 확인해 보자.

```
1 >>> import mod2
2 >>> print(mod2.add(3,4))
3 7
```

이상 없이 불러와서 사용할 수 있다.

PYTHONPATH 환경 변수 사용하기

모듈을 불러와서 사용하는 또 다른 방법으로는 `PYTHONPATH` 환경 변수를 사용하는 방법이 있다.

다음과 같이 따라 해 보자.

```
1 C:\doit>set PYTHONPATH=C:\doit\mymod
2 C:\doit>python
3 >>> import mod2
4 >>> print(mod2.add(3,4))
5 7
```

`set` 명령어를 사용해 `PYTHONPATH` 환경 변수에 `mod2.py` 파일이 있는 `C:/doit/mymod` 디렉토리를 설정한다. 그러면 디렉터리 이동이나 별도의 모듈 추가 작업 없이 `mod2` 모듈을 불러와서 사용할 수 있다.

맥이나 유닉스 환경에서는 `set` 대신 `export` 명령을 사용해야 한다.

05-3 패키지

패키지란 무엇인가?

패키지 (Packages) 는 도트 (.) 를 사용하여 파이썬 모듈을 계층적 (디렉터리 구조) 으로 관리할 수 있게 해 준다. 예를 들어 모듈 이름이 A.B 인 경우에 A 는 패키지 이름이 되고 B 는 A 패키지의 B 모듈이 된다.

파이썬에서 모듈은 하나의.py 파일이다.

파이썬 패키지는 디렉터리와 파이썬 모듈로 이루어지며 구조는 다음과 같다.

가상의 *game* 패키지 예

```
1 game/  
2     __init__.py  
3     sound/  
4         __init__.py  
5         echo.py  
6         wav.py  
7     graphic/  
8         __init__.py  
9         screen.py  
10        render.py  
11    play/  
12        __init__.py  
13        run.py  
14        test.py
```

game, sound, graphic, play 는 디렉터리이고 확장자가.py 인 파일은 파이썬 모듈이다. game 디렉터리가 이 패키지의 루트 디렉터리이고 sound, graphic, play 는 서브 디렉터리이다.

`__init__.py` 파일은 조금 특이한 용도로 사용하는데 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 위 예와 비슷한 **game** 패키지를 직접 만들어 보며 패키지에 대해서 알아보자.

패키지 기본 구성 요소 준비하기

1. **C:/doit** 디렉터리 밑에 **game** 및 기타 서브 디렉터리를 생성하고 **.py** 파일들을 다음과 같이 만들어 보자 (만약 **C:/doit** 디렉터리가 없다면 먼저 생성하고 진행하자).

```
1 C:/doit/game/__init__.py
2 C:/doit/game/sound/__init__.py
3 C:/doit/game/sound/echo.py
4 C:/doit/game/graphic/__init__.py
5 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.

```
1 # echo.py
2 def echo_test():
3     print("echo")
```

4. **render.py** 파일은 다음과 같이 만든다.

```
1 # render.py
2 def render_test():
3     print("render")
```

5. 다음 예제를 수행하기 전에 우리가 만든 **game** 패키지를 참조할 수 있도록 명령 프롬프트 창에서 **set** 명령어로 **PYTHONPATH** 환경 변수에 **C:/doit** 디렉터리를 추가한다. 그리고 파이썬 인터프리터 (Interactive shell) 를 실행한다.

```
1 C:\> set PYTHONPATH=C:/doit
2 C:\> python
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>>
```

여기까지 준비가 되었다면 다음을 따라 해 보자.

중요 공지

아래의 실습은 반드시 명령 프롬프트에서 파이썬 인터프리터를 실행하여 진행해야 한다. 많은 이들이 IDLE 셸 또는 비주얼 스튜디오의 파이썬 셸에서 아래의 예제들을 실행하여 오류를 만난다.

패키지 안의 함수 실행하기

자, 이제 패키지를 사용하여 `echo.py` 파일의 `echo_test` 함수를 실행해 보자. 패키지 안의 함수를 실행하는 방법은 다음 3 가지가 있다. 다음 예제는 `import` 예제이므로 하나의 예제를 실행하고 나서 다음 예제를 실행할 때에는 반드시 인터프리터를 종료하고 다시 실행해야 한다. 인터프리터를 다시 시작하지 않을 경우 이전에 `import` 한 것들이 메모리에 남아 있어 엉뚱한 결과가 나올 수 있다.

첫 번째는 `echo` 모듈을 `import` 하여 실행하는 방법으로, 다음과 같이 실행한다.

echo 모듈은 `echo.py` 파일이다.

```
1 >>> import game.sound.echo
2 >>> game.sound.echo.echo_test()
3 echo
```

두 번째는 `echo` 모듈이 있는 디렉터리까지를 `from ... import` 하여 실행하는 방법이다.

```
1 >>> from game.sound import echo
2 >>> echo.echo_test()
3 echo
```

세 번째는 `echo` 모듈의 `echo_test` 함수를 직접 `import` 하여 실행하는 방법이다.

```
1 >>> from game.sound.echo import echo_test
2 >>> echo_test()
3 echo
```

하지만 다음과 같이 `echo_test` 함수를 사용하는 것은 불가능하다.

다음 예제는 반드시 파이썬 인터프리터를 종료하고 다시 실행해야 한다.

```
1 >>> import game
2 >>> game.sound.echo.echo_test()
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   AttributeError: 'module' object has no attribute 'sound'
```

`import game` 을 수행하면 `game` 디렉터리의 `__init__.py`에 정의한 것만 참조할 수 있다.

또 다음처럼 `echo_test` 함수를 사용하는 것도 불가능하다.

```
1 >>> import game.sound.echo.echo_test
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ImportError: No module named echo_test
```

도트 연산자 (.) 를 사용해서 `import a.b.c` 처럼 `import` 할 때 가장 마지막 항목인 `c` 는 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` 파일이 없어도 패키지로 인식한다 (PEP 420). 하지만 하위 버전 호환을 위해 `__init__.py` 파일을 생성하는 것이 안전한 방법이다.

다음에 따라 해 보자.

```
1 >>> from game.sound import *
2 >>> echo.echo_test()
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   NameError: name 'echo' is not defined
```

뭔가 이상하지 않은가? 분명 `game.sound` 패키지에서 모든 것 (*) 을 import 하였으므로 `echo` 모듈을 사용할 수 있어야 할 것 같은데 `echo` 라는 이름이 정의되지 않았다는 이름 오류 (NameError) 가 발생했다.

이렇게 특정 디렉터리의 모듈을 *를 사용하여 import 할 때에는 다음과 같이 해당 디렉터리의 `__init__.py` 파일에 `__all__` 변수를 설정하고 import 할 수 있는 모듈을 정의해 주어야 한다.

```
1 # C:/doit/game/sound/__init__.py
2 __all__ = ['echo']
```

여기에서 `__all__`이 의미하는 것은 `sound` 디렉터리에서 * 기호를 사용하여 import 할 경우 이곳에 정의된 `echo` 모듈만 import 된다는 의미이다.

착각하기 쉬운데 `from game.sound.echo import *` 는 `__all__` 과 상관없이 무조건 import 된다. 이렇게 `__all__` 과 상관없이 무조건 import 되는 경우는 `from a.b.c import *` 에서 `from` 의 마지막 항목인 `c` 가 모듈인 경우이다.

위와 같이 `__init__.py` 파일을 변경한 후 위 예제를 수행하면 원하던 결과가 출력되는 것을 확인할 수 있다.

```
1 >>> from game.sound import *
2 >>> echo.echo_test()
3 echo
```

relative 패키지

만약 `graphic` 디렉터리의 `render.py` 모듈이 `sound` 디렉터리의 `echo.py` 모듈을 사용하고 싶다면 어떻게 해야 할까? 다음과 같이 `render.py` 를 수정하면 가능하다.

```
1 # render.py
2 from game.sound.echo import echo_test
3 def render_test():
4     print("render")
5     echo_test()
```

`from game.sound.echo import echo_test` 문장을 추가하여 `echo_test` 함수를 사용할 수 있도록 수정했다.

이렇게 수정한 후 다음과 같이 수행해 보자.

```
1 >>> from game.graphic.render import render_test
2 >>> render_test()
3 render
4 echo
```

이상 없이 잘 수행된다.

위 예제처럼 `from game.sound.echo import echo_test`를 입력해 전체 경로를 사용하여 `import` 할 수도 있지만 다음과 같이 `relative` 하게 `import` 하는 것도 가능하다.

```
1 # render.py
2 from ..sound.echo import echo_test
3
4 def render_test():
5     print("render")
6     echo_test()
```

`from game.sound.echo import echo_test`가 `from ..sound.echo import echo_test`로 변경되었다. 여기에서 `..`은 `render.py` 파일의 부모 디렉터리를 의미한다. 따라서 `render.py` 파일의 부모 디렉터리는 `game`이므로 위와 같은 `import`가 가능한 것이다.

`render.py` 파일의 현재 디렉터리는 `graphic` 이고 부모 디렉터리는 `game` 이다.

`relative` 한 접근자에는 다음과 같은 것이 있다.

- `..` - 부모 디렉터리
- `.` - 현재 디렉터리

05-4 예외 처리

프로그램을 만들다 보면 수없이 많은 오류를 만나게 된다. 물론 오류가 발생하는 이유는 프로그램이 잘못 동작하는 것을 막기 위한 파이썬의 배려이다. 이번 장에서는 파이썬에서 오류를 처리하는 방법에 대해서 알아보자.

오류는 어떤 때 발생하는가?

오류를 처리하는 방법을 공부하기 전에 어떤 상황에서 오류가 발생하는지 한번 알아보자. 오타를 입력했을 때 발생하는 구문 오류 같은 것이 아닌 실제 프로그램에서 자주 발생하는 오류를 중심으로 살펴보자.

먼저 존재하지 않는 파일을 사용하려고 시도했을 때 발생하는 오류이다.

```
1 >>> f = open("나 없는 파일", 'r')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   FileNotFoundError: [Errno 2] No such file or directory: '나 없는 파일'
```

위 예에서 볼 수 있듯이 없는 파일을 열려고 시도하면 `FileNotFoundError` 오류가 발생한다.

이번에는 0으로 다른 숫자를 나누는 경우를 생각해 보자. 이 역시 자주 발생하는 오류이다.

```
1 >>> 4 / 0
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ZeroDivisionError: division by zero
```

4를 0으로 나누려니까 `ZeroDivisionError` 오류가 발생한다.

마지막으로 한 가지 예를 더 들어 보자. 다음 오류는 정말 빈번하게 일어난다.

```
1 >>> a = [1,2,3]
2 >>> a[4]
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   IndexError: list index out of range
```

`a`는 리스트 `[1, 2, 3]`이므로 `a[4]`는 `a` 리스트에서 얻을 수 없는 값이다. 따라서 `IndexError` 오류가 발생한다. 파이썬은 이런 오류가 발생하면 프로그램을 중단하고 오류 메시지를 보여 준다.

오류 예외 처리 기법

자, 이제 유연한 프로그래밍을 위한 오류 처리 방법에 대해 알아보자.

try, except 문

다음은 오류 처리를 위한 try, except 문의 기본 구조이다.

```
1 try:
2     ...
3 except [발생 오류 [as 오류 변수]]:
4     ...
```

try 블록 수행 중 오류가 발생하면 except 블록이 수행된다. 하지만 try 블록에서 오류가 발생하지 않는다면 except 블록은 수행되지 않는다.

except 구문을 자세히 살펴보자.

```
except [발생오류 [as 오류변수]]:
```

위 구문을 보면 [] 기호를 사용하는데, 이 기호는 괄호 안의 내용을 생략할 수 있다는 관례적인 표기법이다. 즉 except 구문은 다음 3 가지 방법으로 사용할 수 있다.

1. try, except 만 쓰는 방법

```
1 try:
2     ...
3 except:
4     ...
```

이 경우는 오류 종류에 상관없이 오류가 발생하면 except 블록을 수행한다.

2. 발생 오류만 포함한 except 문

```
1 try:
2     ...
3 except 발생 오류:
4     ...
```

이 경우는 오류가 발생했을 때 except 문에 미리 정해 놓은 오류와 동일한 오류일 경우에만 except 블록을 수행한다는 뜻이다.

3. 발생오류와 오류변수까지 포함한 except 문

```
1 try:
2     ...
3 except 발생 오류 as 오류 변수:
```



```
4      ...
```

이 경우는 두 번째 경우에서 오류의 내용까지 알고 싶을 때 사용하는 방법이다.

이 방법의 예를 들어 보면 다음과 같다.

```
1  try:
2      4 / 0
3  except ZeroDivisionError as e:
4      print(e)
```

위처럼 4를 0으로 나누려고 하면 `ZeroDivisionError`가 발생하여 `except` 블록이 실행되고 오류변수 `e`에 담기는 오류 메시지를 출력할 수 있다. 출력한 오류메시지는 다음과 같다.

```
division by zero
```

try .. finally

`try` 문에는 `finally` 절을 사용할 수 있다. `finally` 절은 `try` 문 수행 도중 예외 발생 여부에 상관없이 항상 수행된다. 보통 `finally` 절은 사용한 리소스를 `close` 해야 할 때에 많이 사용한다.

다음 예를 보자.

```
1  try:
2      f = open('foo.txt', 'w')
3      # 무언가를 수행한다.
4
5      (... 생략 ...)
6
7  finally:
8      f.close() # 중간에 오류가 발생하더라도 무조건 실행된다.
```

`foo.txt` 파일을 쓰기 모드로 연 후에 `try` 문을 수행한 후 예외 발생 여부와 상관없이 `finally` 절에서 `f.close()`로 열린 파일을 닫을 수 있다.

여러개의 오류처리하기

`try` 문 안에서 여러 개의 오류를 처리하려면 다음과 같이 사용한다.

```
1  try:
2      ...
3  except 발생오류1:
4      ...
5  except 발생오류2:
6      ...
```

즉 0으로 나누는 오류와 인덱싱 오류를 다음과 같이 처리할 수 있다.

```
1 try:
2     a = [1,2]
3     print(a[3])
4     4/0
5 except ZeroDivisionError:
6     print("0으로 나눌 수 없습니다.")
7 except IndexError:
8     print("인덱싱 할 수 없습니다.")
```

a는 2개의 요솟값을 가지고 있기 때문에 a[3]은 `IndexError`를 발생시키므로 “인덱싱할 수 없습니다.”라는 문자열이 출력될 것이다. 인덱싱 오류가 먼저 발생했으므로 4/0으로 발생하는 `ZeroDivisionError` 오류는 발생하지 않는다.

앞에서 알아본 것과 마찬가지로 오류 메시지도 다음과 같이 확인할 수 있다.

```
1 try:
2     a = [1,2]
3     print(a[3])
4     4/0
5 except ZeroDivisionError as e:
6     print(e)
7 except IndexError as e:
8     print(e)
```

프로그램을 실행하면 “list index out of range”라는 오류 메시지가 출력될 것이다.

다음과 같이 `ZeroDivisionError`와 `IndexError`를 함께 처리할 수도 있다.

```
1 try:
2     a = [1,2]
3     print(a[3])
4     4/0
5 except (ZeroDivisionError, IndexError) as e:
6     print(e)
```

2개 이상의 오류를 동일하게 처리하기 위해서는 위와 같이 괄호를 사용하여 함께 묶어 처리하면 된다.

try ... else

try 문에는 다음처럼 else 절을 사용할 수 있다.

```
1 try:
2     ...
3 except [발생오류 [as 오류변수]]:
4     ...
5 else: # 오류가 없을 경우에만 수행된다.
6     ...
```

try 문 수행중 오류가 발생하면 except 절이 수행되고 오류가 없으면 else 절이 수행된다.

다음은 try 문에 else 절을 사용한 간단한 예제이다.

```
1 try:
2     age=int(input('나이를 입력하세요: '))
3 except:
4     print('입력이 정확하지 않습니다.')
5 else:
6     if age <= 18:
7         print('미성년자는 출입금지입니다.')
8     else:
9         print('환영합니다.')
```

만약 '나이를 입력하세요:' 라는 질문에 숫자가 아닌 다른 값을 입력하면 오류가 발생하여 '입력이 정확하지 않습니다.' 라는 문장을 출력한다. 오류가 없을 경우에만 else 절이 수행된다.

오류 회피하기

코드를 작성하다 보면 특정 오류가 발생할 경우 그냥 통과시켜야 할 때가 있다. 다음 예를 보자.

```
1 try:
2     f = open("나 없는 파일", 'r')
3 except FileNotFoundError:
4     pass
```

try 문 안에서 FileNotFoundError 가 발생할 경우에 pass 를 사용하여 오류를 그냥 회피하도록 작성한 예제이다.

오류 일부러 발생시키기

이상하게 들리겠지만 프로그래밍을 하다 보면 종종 오류를 일부러 발생시켜야 할 경우도 생긴다. 파이썬은 raise 명령어를 사용해 오류를 강제로 발생시킬 수 있다.

예를 들어 Bird 클래스를 상속받는 자식 클래스는 반드시 fly 라는 함수를 구현하도록 만들고 싶은 경우 (강제로 그렇게 하고 싶은 경우) 가 있을 수 있다. 다음 예를 보자.

```
1 class Bird:
2     def fly(self):
3         raise NotImplementedError
```

위 예제는 Bird 클래스를 상속받는 자식 클래스는 반드시 fly 함수를 구현해야 한다는 의지를 보여 준다. 만약 자식 클래스가 fly 함수를 구현하지 않은 상태로 fly 함수를 호출한다면 어떻게 될까?

`NotImplementedError`는 파이썬에 이미 정의되어 있는 오류로, 꼭 작성해야 하는 부분이 구현되지 않았을 경우 일부러 오류를 일으키기 위해 사용한다.

```
1 class Eagle(Bird):
2     pass
3
4 eagle = Eagle()
5 eagle.fly()
```

```
1 Traceback (most recent call last):
2   File "...", line 33, in <module>
3     eagle.fly()
4   File "...", line 26, in fly
5     raise NotImplementedError
6 NotImplementedError
```

`Eagle` 클래스는 `Bird` 클래스를 상속받았다. 그런데 `Eagle` 클래스는 `fly` 메서드를 오버라이딩하여 구현하지 않았다. 따라서 `eagle` 객체의 `fly` 메서드를 수행하는 순간 `Bird` 클래스의 `fly` 메서드가 수행되어 `NotImplementedError`가 발생한다.

상속받는 클래스에서 메서드를 재구현하는 것을 메서드 오버라이딩이라고 한다.

`NotImplementedError`가 발생되지 않게 하려면 다음과 같이 `Eagle` 클래스에 `fly` 함수를 구현해야 한다.

```
1 class Eagle(Bird):
2     def fly(self):
3         print("very fast")
4
5 eagle = Eagle()
6 eagle.fly()
```

위 예처럼 `fly` 함수를 구현한 후 프로그램을 실행하면 오류 없이 다음 문장이 출력된다.

```
1 very fast
```

예외 만들기

프로그램 수행 도중 특수한 경우에만 예외 처리를 하기 위해서 종종 예외를 만들어서 사용한다. 이번에는 직접 예외를 만들어 보자. 예외는 다음과 같이 파이썬 내장 클래스인 `Exception` 클래스를 상속하여 만들 수 있다.

```
1 class MyError(Exception):
2     pass
```

그리고 별명을 출력하는 함수를 다음과 같이 작성해 보자.

```
1 def say_nick(nick):
2     if nick == '바보':
3         raise MyError()
4     print(nick)
```

그리고 다음과 같이 say_nick 함수를 호출해 보자.

```
1 say_nick("천사")
2 say_nick("바보")
```

저장한 뒤 프로그램을 실행해 보면 다음과 같이 “천사”가 한 번 출력된 후 MyError가 발생한다.

```
1 천사
2 Traceback (most recent call last):
3   File "...", line 11, in <module>
4     say_nick("바보")
5   File "...", line 7, in say_nick
6     raise MyError()
7   __main__.MyError
```

이번에는 예외 처리 기법을 사용하여 MyError 발생을 예외 처리해 보자.

```
1 try:
2     say_nick("천사")
3     say_nick("바보")
4 except MyError:
5     print("허용되지 않는 별명입니다.")
```

프로그램을 실행하면 다음과 같이 출력된다.

```
1 천사
2 허용되지 않는 별명입니다.
```

만약 오류 메시지를 사용하고 싶다면 다음처럼 예외 처리를 하면 된다.

```
1 try:
2     say_nick("천사")
3     say_nick("바보")
4 except MyError as e:
5     print(e)
```

하지만 프로그램을 실행해 보면 print(e)로 오류 메시지가 출력되지 않는 것을 확인할 수 있다. 오류 메시지를 출력했을 때 오류 메시지가 보이게 하려면 오류 클래스에 다음과 같은 __str__ 메서드를 구현해야 한다. __str__ 메서드는 print(e) 처럼 오류 메시지를 print 문으로 출력할 경우에 호출되는 메서드이다.

```
1 class MyError(Exception):
2     def __str__(self):
3         return "허용되지 않는 별명입니다."
```

프로그램을 다시 실행해 보면 “허용되지 않는 별명입니다.” 라는 오류메시지가 출력되는 것을 확인할 수 있을 것이다.

05-5 내장 함수

지금까지 파이썬으로 프로그래밍하기 위해 알아야 하는 것들을 대부분 공부했다. 이제 여러분은 원하는 프로그램을 직접 만들 수 있을 것이다. 하지만 그 전에 먼저 여러분이 만들려는 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 프로그램은 테스트 과정을 수 없이 거쳤기 때문에 충분히 검증되어 있다. 따라서 무엇인가 새로운 프로그램을 만들기 전에는 이미 만들어진 것들, 그중에서도 특히 파이썬 배포본에 함께 들어 있는 파이썬 라이브러리를 살펴보는 것이 매우 중요하다.

라이브러리를 살펴보기 전에 파이썬 내장 함수 (**Built-in**, 빌트인함수) 를 먼저 살펴보자. 우리는 이미 몇 가지 내장 함수를 배웠다. **print**, **del**, **type** 등이 바로 그것이다. 이러한 파이썬 내장 함수는 파이썬 모듈과 달리 **import** 가 필요하지 않기 때문에 아무런 설정 없이 바로 사용할 수 있다.

이 책에서는 활용 빈도가 높고 중요한 함수를 중심으로 알파벳 순서대로 간략히 정리했다. 파이썬으로 프로그래밍을 하기 위해 이들 함수를 지금 당장 모두 알아야 하는 것은 아니므로 가벼운 마음으로 천천히 살펴보자.

abs

abs(x) 는 어떤 숫자를 입력받았을 때, 그 숫자의 절댓값을 리턴하는 함수이다.

```
1 >>> abs(3)
2 3
3 >>> abs(-3)
4 3
5 >>> abs(-1.2)
6 1.2
```

all

`all(x)` 는 반복 가능한 (iterable) 데이터 `x` 를 입력 값으로 받으며 이 `x` 의 요소가 모두 참이면 `True`, 거짓이 하나라도 있으면 `False` 를 리턴한다.

반복 가능한 데이터란 `for` 문에서 사용 가능한 자료형을 의미한다. 리스트, 튜플, 문자열, 딕셔너리, 집합 등이 있다.

다음 예를 보자.

```
1 >>> all([1, 2, 3])
2 True
```

리스트 `[1, 2, 3]` 은 모든 요소가 참이므로 `True` 를 리턴한다.

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

리스트 `[1, 2, 3, 0]` 중에서 요소 `0` 은 거짓이므로 `False` 를 리턴한다.

```
1 >>> all([])
2 True
```

만약 `all` 의 입력 인수가 빈 값인 경우에는 `True` 를 리턴한다.

자료형의 참과 거짓은 02-7 장을 참고하자.

any

`any(x)` 는 반복 가능한 (iterable) 데이터 `x` 를 입력으로 받아 `x` 의 요소 중 하나라도 참이 있으면 `True` 를 리턴하고, `x` 가 모두 거짓일 때에만 `False` 를 리턴한다. `all(x)` 의 반대이다.

다음 예를 보자.

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

리스트 `[1, 2, 3, 0]` 중에서 `1, 2, 3` 이 참이므로 `True` 를 리턴한다.

```
1 >>> any([0, ""])
2 False
```

리스트 `[0, ""]` 의 요소 `0` 과 `""` 은 모두 거짓이므로 `False` 를 리턴한다.

```
1 >>> any([])
2 False
```


만약 **any** 의 입력 인수가 빈 값인 경우에는 **False** 를 리턴한다.

chr

chr(i) 는 유니코드 숫자값을 입력받아 그 코드에 해당하는 문자를 리턴하는 함수이다.

유니코드는 전 세계의 모든 문자를 컴퓨터에서 일관되게 표현하고 다룰 수 있도록 설계된 산업 표준이다.

```
1 >>> chr(97)
2 'a'
3 >>> chr(44032)
4 '가'
```

dir

dir 은 객체가 지닌 변수나 함수를 보여 주는 함수이다. 다음 예는 리스트와 딕셔너리가 지닌 함수 (메서드) 를 보여 주는 예이다. 우리가 02 장에서 살펴본 함수들을 구경할수 있다.

```
1 >>> dir([1, 2, 3])
2 ['append', 'count', 'extend', 'index', 'insert', 'pop',...]
3 >>> dir({'1':'a'})
4 ['clear', 'copy', 'get', 'has_key', 'items', 'keys',...]
```

divmod

divmod(a, b) 는 2 개의 숫자를 입력으로 받는다. 그리고 **a** 를 **b** 로 나눈 몫과 나머지를 튜플로 리턴하는 함수이다.

```
1 >>> divmod(7, 3)
2 (2, 1)
```

몫을 구하는 연산자 **//** 와 나머지를 구하는 연산자 **%** 를 각각 사용한 결과와 비교해 보자.

```
1 >>> 7 // 3
2 2
3 >>> 7 % 3
4 1
```

enumerate

`enumerate` 는 “열거하다” 라는 뜻이다. 이 함수는 순서가 있는 데이터 (리스트, 튜플, 문자열) 를 입력으로 받아 인덱스 값을 포함하는 `enumerate` 객체를 리턴한다.

보통 `enumerate` 함수는 다음 예제처럼 `for` 문과 함께 사용한다.

다음 예를 보자.

```
1 >>> for i, name in enumerate(['body', 'foo', 'bar']):
2     ...     print(i, name)
3     ...
4 0 body
5 1 foo
6 2 bar
```

순서 값과 함께 `body`, `foo`, `bar` 가 순서대로 출력되었다. 즉 위 예제와 같이 `enumerate` 를 `for` 문과 함께 사용하면 자료형의 현재 순서 (index) 와 그 값을 쉽게 알 수 있다.

`for` 문처럼 반복되는 구간에서 객체가 현재 어느 위치에 있는지 알려 주는 인덱스 값이 필요할 때 `enumerate` 함수를 사용하면 매우 유용하다.

eval

`eval(expression)` 은 문자열로 구성된 표현식을 입력으로 받아 해당 문자열을 실행한 결과값을 리턴하는 함수이다.

```
1 >>> eval('1+2')
2 3
3 >>> eval("'hi' + 'a'")
4 'hia'
5 >>> eval('divmod(4, 3)')
6 (1, 1)
```

filter

`filter` 란 무엇인가를 걸러낸다는 뜻으로 `filter` 함수도 이와 비슷한 기능을 한다.

`filter(func, iterable)`

`filter` 함수는 첫 번째 인수로 함수를, 두 번째 인수로 그 함수에 차례로 들어갈 반복 가능한 데이터를 받는다. 그리고 반복 가능한 데이터 (`iterable`) 의 요소 순서대로 함수 (`func`) 를 호출했을 때 반환 값이 참인 것만 묶어서 (걸러 내서) 리턴한다.

다음 예를 보자.

```
1 #positive.py
2 def positive(l):
3     result = []
4     for i in l:
5         if i > 0:
6             result.append(i)
7     return result
8
9 print(positive([1,-3,2,0,-5,6]))
```

결괏값: [1, 2, 6]

위에서 만든 `positive` 함수는 리스트를 입력으로 받아 각각의 요소를 판별해서 양수 값만 리턴하는 함수이다.

`filter` 함수를 사용하면 위 내용을 다음과 같이 간단하게 작성할 수 있다.

```
1 #filter1.py
2 def positive(x):
3     return x > 0
4
5 print(list(filter(positive, [1, -3, 2, 0, -5, 6])))
```

결괏값: [1, 2, 6]

`filter(positive, [1, -3, 2, 0, -5, 6])`는 `[1, -3, 2, 0, -5, 6]`의 각 요소값을 순서대로 `positive` 함수에 적용하여 반환 값이 참인 것만 묶어서 리턴한다. 즉 1, 2, 6 요소만 `x > 0` 문장이 참이 되므로 `[1, 2, 6]` 이라는 결괏값이 출력된다.

이 함수는 `lambda` 를 사용하면 더욱 심플해진다.

```
1 >>> list(filter(lambda x: x > 0, [1, -3, 2, 0, -5, 6]))
2 [1, 2, 6]
```

hex

`hex(x)` 는 정수를 입력받아 16 진수 (hexadecimal) 문자열로 변환하여 리턴하는 함수이다.

```
1 >>> hex(234)
```

```
2 '0xea'
3 >>> hex(3)
4 '0x3'
```

id

`id(object)` 는 객체를 입력받아 객체의 고유 주소 값 (레퍼런스) 을 리턴하는 함수이다.

```
1 >>> a = 3
2 >>> id(3)
3 135072304
4 >>> id(a)
5 135072304
6 >>> b = a
7 >>> id(b)
8 135072304
```

위 예의 3, a, b 는 고유 주소 값이 모두 135072304 이다. 즉 3, a, b 가 모두 같은 객체를 가리키고 있다.

만약 `id(4)` 라고 입력하면 4 는 3, a, b 와 다른 객체이므로 당연히 다른 고유 주소 값이 출력된다.

```
1 >>> id(4)
2 135072292
```

input

`input([prompt])` 은 사용자 입력을 받는 함수이다. 입력 인수로 문자열을 전달하면 그 문자열은 프롬프트가 된다.

[] 기호는 괄호 안의 내용을 생략할 수 있다는 관례 표기법임을 기억하자.

```
1 >>> a = input()
2 hi
3 >>> a
4 'hi'
5 >>> b = input("Enter: ")
6 Enter: hi
7 >>> b
8 'hi'
```

int

`int(x)` 는 문자열 형태의 숫자나 소수점이 있는 숫자를 정수로 리턴하는 함수이다. 만약 정수가 입력되면 그대로 리턴한다.

```
1 >>> int('3')
2 3
3 >>> int(3.4)
4 3
```

`int(x, radix)` 는 `radix` 진수로 표현된 문자열 `x` 를 10 진수로 변환하여 리턴한다. 예를 들어 2 진수로 표현된 “11” 의 10 진수 값은 다음과 같이 구할수 있다.

```
1 >>> int('11', 2)
2 3
```

16 진수로 표현된 “1A” 의 10 진수 값은 다음과 같이 구할수 있다.

```
1 >>> int('1A', 16)
2 26
```

isinstance

`isinstance(object, class)` 함수는 첫 번째 인수로 객체, 두 번째 인수로 클래스를 받는다. 입력으로 받은 객체가 그 클래스의 인스턴스인지를 판단하여 참이면 `True`, 거짓이면 `False` 를 리턴한다.

```
1 >>> class Person: pass
2 ...
3 >>> a = Person()
4 >>> isinstance(a, Person)
5 True
```

위 예는 `a` 객체가 `Person` 클래스에 의해 생성된 인스턴스임을 확인시켜 준다.

```
1 >>> b = 3
2 >>> isinstance(b, Person)
3 False
```

`b` 는 `Person` 클래스가 만든 인스턴스가 아니므로 `False` 를 리턴한다.

len

`len(s)` 은 입력값 `s` 의 길이 (요소의 전체 개수) 를 리턴하는 함수이다.

```
1 >>> len("python")
2 6
3 >>> len([1,2,3])
4 3
5 >>> len((1, 'a'))
6 2
```

list

`list(iterable)` 는 반복 가능한 데이터 (iterable) 를 입력받아 리스트로 만들어 리턴하는 함수이다.

```
1 >>> list("python")
2 ['p', 'y', 't', 'h', 'o', 'n']
3 >>> list((1,2,3))
4 [1, 2, 3]
```

`list` 함수에 리스트를 입력하면 똑같은 리스트를 복사하여 리턴한다.

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

map

`map(f, iterable)` 은 함수 (f) 와 반복 가능한 데이터를 입력으로 받는다. `map` 함수는 입력받은 데이터의 각 요소에 함수 f를 적용한 결과를 리턴하는 함수이다.

다음 예를 보자.

```
1 # two_times.py
2 def two_times(numberList):
3     result = []
4     for number in numberList:
5         result.append(number*2)
6     return result
7
8 result = two_times([1, 2, 3, 4])
9 print(result)
```

`two_times` 함수는 리스트를 입력받아 리스트의 각 요소에 2를 곱해 리턴하는 함수이다. 실행 결과는 다음과 같다.

결괏값: [2, 4, 6, 8]

위 예제는 `map` 함수를 사용하여 다음처럼 바꿀 수 있다.

```
1 >>> def two_times(x):
2     ...     return x*2
3     ...
4 >>> list(map(two_times, [1, 2, 3, 4]))
5 [2, 4, 6, 8]
```

이 예제를 해석해 보자. 먼저 리스트의 첫 번째 요소인 1 이 `two_times` 함수의 입력값으로 들어가고 $1 * 2$ 의 과정을 거쳐서 2 가 된다. 다음으로 리스트의 두 번째 요소인 2 가 $2 * 2$ 의 과정을 거쳐 4 가 된다. 따라서 결괏값은 이제 [2, 4] 가 된다. 총 4 개의 요소값이 모두 수행되면 [2, 4, 6, 8]이 된다. 이것이 `map` 함수가 하는 일이다.

위 예에서 `map` 함수의 결과를 리스트로 출력하기 위해 `list` 함수를 사용했다. (`map` 함수는 `map` 객체를 리턴한다.)

앞의 예는 `lambda` 를 사용하여 다음처럼 간략하게 만들 수 있다.

```
1 >>> list(map(lambda a: a*2, [1, 2, 3, 4]))
2 [2, 4, 6, 8]
```

max

`max(iterable)` 함수는 인수로 반복 가능한 데이터를 입력받아 그 최댓값을 리턴하는 함수이다.

```
1 >>> max([1, 2, 3])
2 3
3 >>> max("python")
4 'y'
```

min

`min(iterable)` 함수는 `max` 함수와 반대로, 인수로 반복 가능한 데이터를 입력받아 그 최솟값을 리턴하는 함수이다.

```
1 >>> min([1, 2, 3])
2 1
3 >>> min("python")
4 'h'
```

oct

oct(x) 는 정수를 8 진수 문자열로 바꾸어 리턴하는 함수이다.

```
1 >>> oct(34)
2 '0o42'
3 >>> oct(12345)
4 '0o30071'
```

open

open(filename, [mode]) 은 “파일 이름” 과 “읽기 방법” 을 입력받아 파일 객체를 리턴하는 함수이다. 읽기 방법 (mode) 을 생략하면 기본값인 읽기 모드 (r) 로 파일 객체를 만들어 리턴한다.

mode	설명
w	쓰기 모드로 파일 열기
r	읽기 모드로 파일 열기
a	추가 모드로 파일 열기
b	바이너리 모드로 파일 열기

b 는 w, r, a 와 함께 사용한다.

```
1 >>> f = open("binary_file", "rb")
```

위 예의 rb 는 “바이너리 읽기 모드” 를 의미한다.

ord

ord(c) 는 문자의 유니코드 숫자 값을 리턴하는 함수이다.

ord 함수는 chr 함수와 반대이다.

```
1 >>> ord('a')
2 97
3 >>> ord('가')
4 44032
```


pow

`pow(x, y)` 는 `x` 의 `y` 제곱한 결과값을 리턴하는 함수이다.

```
1 >>> pow(2, 4)
2 16
3 >>> pow(3, 3)
4 27
```

range

`range([start,] stop [,step])` 는 `for` 문과 함께 자주 사용하는 함수이다. 이 함수는 입력받은 숫자에 해당하는 범위 값을 반복 가능한 객체로 만들어 리턴한다.

인수가 하나일 경우

시작 숫자를 지정해 주지 않으면 `range` 함수는 0 부터 시작한다.

```
1 >>> list(range(5))
2 [0, 1, 2, 3, 4]
```

인수가 2 개일 경우

입력으로 주어지는 2 개의 인수는 시작 숫자와 끝 숫자를 나타낸다. 단 끝 숫자는 해당 범위에 포함되지 않는다는 것에 주의하자.

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

인수가 3 개일 경우

세 번째 인수는 숫자 사이의 거리를 말한다.

```
1 >>> list(range(1, 10, 2))
2 [1, 3, 5, 7, 9]
3 >>> list(range(0, -10, -1))
4 [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

round

`round(number[, ndigits])` 함수는 숫자를 입력받아 반올림해 리턴하는 함수이다.

[, `ndigits`]는 `ndigits` 가 있을 수도 있고 없을 수도 있다는 의미이다.

```
1 >>> round(4.6)
2 5
3 >>> round(4.2)
4 4
```

다음과 같이 실수 5.678 을 소수점 2 자리까지만 반올림하여 표시할 수 있다.

```
1 >>> round(5.678, 2)
2 5.68
```

`round` 함수의 두 번째 인수는 반올림하여 표시하고 싶은 소수점의 자릿수 (`ndigits`) 를 의미한다.

sorted

`sorted(iterable)` 함수는 입력 데이터를 정렬한 후 그 결과를 리스트로 리턴하는 함수이다.

```
1 >>> sorted([3, 1, 2])
2 [1, 2, 3]
3 >>> sorted(['a', 'c', 'b'])
4 ['a', 'b', 'c']
5 >>> sorted("zero")
6 ['e', 'o', 'r', 'z']
7 >>> sorted((3, 2, 1))
8 [1, 2, 3]
```

리스트 자료형에도 `sort` 함수가 있다. 하지만 리스트 자료형의 `sort` 함수는 리스트 객체 그 자체를 정렬만 할 뿐 정렬된 결과를 리턴하지는 않는다.

str

`str(object)` 은 문자열 형태로 객체를 변환하여 리턴하는 함수이다.

```
1 >>> str(3)
2 '3'
3 >>> str('hi')
4 'hi'
```

sum

`sum(iterable)` 함수는 입력 데이터의 합을 리턴하는 함수이다.

```
1 >>> sum([1,2,3])
2 6
3 >>> sum((4,5,6))
4 15
```

tuple

`tuple(iterable)`은 반복 가능한 데이터를 튜플로 바꾸어 리턴하는 함수이다. 만약 입력이 튜플인 경우에는 그대로 리턴한다.

```
1 >>> tuple("abc")
2 ('a', 'b', 'c')
3 >>> tuple([1, 2, 3])
4 (1, 2, 3)
5 >>> tuple((1, 2, 3))
6 (1, 2, 3)
```

type

`type(object)`은 입력값의 자료형이 무엇인지 알려 주는 함수이다.

```
1 >>> type("abc")
2 <class 'str'>
3 >>> type([ ])
4 <class 'list'>
5 >>> type(open("test", 'w'))
6 <class '_io.TextIOWrapper'>
```

zip

`zip(*iterable)`은 동일한 개수로 이루어진 데이터들을 묶어서 리턴하는 함수이다.

여기서 사용한 `*iterable`은 반복 가능 (iterable) 한 데이터 여러 개를 입력할 수 있다는 의미이다.

다음 예제로 사용법을 확인해 보자.

```
1 >>> list(zip([1, 2, 3], [4, 5, 6]))
2 [(1, 4), (2, 5), (3, 6)]
3 >>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))
4 [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
5 >>> list(zip("abc", "def"))
6 [('a', 'd'), ('b', 'e'), ('c', 'f')]
```

05-6 표준 라이브러리

이제 파이썬 프로그래밍 능력을 높여 줄 더 큰 날개를 달아 보자. 전 세계의 파이썬 고수들이 만든 유용한 프로그램을 모아 놓은 것이 바로 파이썬 표준 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 라이브러리가 존재하고 어떻게 사용하는지 알아야 한다. 자주 사용되고 꼭 알아 두면 좋은 라이브러리를 중심으로 하나씩 살펴보자.

- 파이썬 표준 라이브러리는 파이썬을 설치할 때 자동으로 컴퓨터에 설치된다.
- `sys`, `re` 모듈은 파이썬의 중요한 표준 라이브러리이다. `sys` 모듈은 04-4 장, `re` 모듈은 08 장 전체에 걸쳐 다루고 있어 이곳에서는 설명하지 않는다.

`datetime.date`

`datetime.date` 는 년, 월, 일로 **날짜**를 표현할 때 사용하는 함수이다.

만약 A 군이 여친을 2019 년 12 월 14 일부터 만나기 시작했다면 2021 년 6 월 5 일은 사귀 지 며칠째 되는 날일까? 아울러 사귀기 시작한 2019 년 12 월 14 일은 무슨 요일이었을까? `datetime.date` 함수를 사용하면 이 문제를 쉽게 해결할 수 있다.



년, 월, 일로 다음과 같이 `datetime.date` 객체를 만들 수 있다.

```
1 >>> import datetime
2 >>> day1 = datetime.date(2019, 12, 14)
3 >>> day2 = datetime.date(2021, 6, 5)
```

이처럼 년, 월, 일을 인수로 하여 2019년 12월 14일에 해당하는 날짜 객체는 **day1**, 2021년 6월 5일에 해당하는 날짜 객체는 **day2**로 생성하였다. 이렇게 날짜 객체를 만들었다면 두 날짜의 차이는 다음과 같이 뺄셈으로 쉽게 구할 수 있다.

```
1 >>> diff = day2 - day1
2 >>> diff.days
3 539
```

day2에서 **day1**을 빼면 **diff** 객체가 리턴되고 이 객체를 이용하면 두 날짜의 차이를 쉽게 확인할 수 있다.

요일은 **datetime.date** 객체의 **weekday()** 함수를 사용하면 쉽게 구할 수 있다.

```
1 >>> import datetime
2 >>> day = datetime.date(2019, 12, 14)
3 >>> day.weekday()
4 5
```

0은 월요일을 의미하며 순서대로 1은 화요일, 2는 수요일, ..., 6은 일요일이 된다. 이와는 달리 월요일은 1, 화요일은 2, ..., 일요일은 7을 리턴하려면 다음처럼 **isoweekday()** 함수를 사용하면 된다.

```
1 >>> day.isoweekday()
2 6
```

2019년 12월 14일은 토요일이므로 **isoweekday()**를 사용하면 토요일을 뜻하는 6이 리턴된다. (**weekday**를 사용하면 5가 리턴된다.)

time

시간과 관련된 **time** 모듈에는 함수가 굉장히 많다. 그중 가장 유용한 몇 가지만 알아보자.

time.time

time.time()은 UTC(Universal Time Coordinated 협정 세계 표준시)를 사용하여 현재 시간을 실수 형태로 리턴하는 함수이다. 1970년 1월 1일 0시 0분 0초를 기준으로 지난 시간을 초 단위로 돌려준다.

```
1 >>> import time
2 >>> time.time()
3 988458015.73417199
```

time.localtime

`time.localtime`은 `time.time()`이 리턴한 실수 값을 사용해서 연도, 월, 일, 시, 분, 초, ...의 형태로 바꾸어 주는 함수이다.

```
1 >>> time.localtime(time.time())
2 time.struct_time(tm_year=2013, tm_mon=5, tm_mday=21, tm_hour=16,
3   tm_min=48, tm_sec=42, tm_wday=1, tm_yday=141, tm_isdst=0)
```

time.asctime

위 `time.localtime`에 의해서 반환된 튜플 형태의 값을 인수로 받아서 날짜와 시간을 알아보기 쉬운 형태로 리턴하는 함수이다.

```
1 >>> time.asctime(time.localtime(time.time()))
2 'Sat Apr 28 20:50:20 2001'
```

time.ctime

`time.asctime(time.localtime(time.time()))`은 `time.ctime()`을 사용해 간편하게 표시할 수 있다. `asctime`과 다른 점은 `ctime`은 항상 현재 시간만을 리턴한다는 점이다.

```
1 >>> time.ctime()
2 'Sat Apr 28 20:56:31 2001'
```

time.strftime

```
1 time.strftime('출력할 형식 포맷 코드', time.localtime(time.time()))
```

`strftime` 함수는 시간에 관계된 것을 세밀하게 표현하는 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)	[01,31]
%H	시간 (hour)-24 시간 출력 형태	[00,23]
%I	시간 (hour)-12 시간 출력 형태	[01,12]

포맷코드	설명	예
%j	1 년 중 누적 날짜	[001,366]
%m	달	[01,12]
%M	분	[01,59]
%p	AM or PM	AM
%S	초	[00,59]
%U	1 년 중 누적 주-일요일을 시작으로	[00,53]
%w	숫자로 된 요일	[0(일요일),6]
%W	1 년 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

다음은 `time.strftime` 을 사용하는 예이다.

```

1 >>> import time
2 >>> time.strftime('%x', time.localtime(time.time()))
3 '05/01/01'
4 >>> time.strftime('%c', time.localtime(time.time()))
5 '05/01/01 17:22:21'
```

time.sleep

`time.sleep` 함수는 주로 루프 안에서 많이 사용한다. 이 함수를 사용하면 일정한 시간 간격을 두고 루프를 실행할 수 있다. 다음 예를 보자.

```

1 #sleep1.py
2 import time
3 for i in range(10):
4     print(i)
5     time.sleep(1)
```

위 예는 1 초 간격으로 0 부터 9 까지의 숫자를 출력한다. 위 예에서 볼 수 있듯이 `time.sleep` 함수의 인수

는 실수 형태를 쓸 수 있다. 즉 1 이면 1 초, 0.5 면 0.5 초가 되는 것이다.

math.gcd

math.gcd 함수를 이용하면 최대공약수 (gcd, greatest common divisor) 를 쉽게 구할 수 있다.

math.gcd 함수는 파이썬 3.5 버전부터 사용할 수 있다.

최대공약수란?

공약수 (common divisor) 란 두 수 이상의 여러 수의 공통된 약수를 의미한다. 예를 들어 30 과 15 는 공통으로 1, 3, 5, 15 라는 약수를 가지고 있는데, 이를 공약수라 하며 이 중 가장 큰 공약수를 최대공약수라 한다. 즉, 30 과 15 의 최대공약수는 15 이다.

어린이집에서 사탕 60 개, 초콜릿 100 개, 젤리 80 개를 준비했다. 아이들이 서로 싸우지 않도록 똑같이 나누어 봉지에 담는다고 하면 최대 몇 봉지까지 만들 수 있을까? 단, 사탕, 초콜릿, 젤리는 남기지 않고 모두 담도록 한다.



이 문제는 60, 100, 80 의 최대공약수를 구하면 바로 해결된다. 즉, 똑같이 나눌 수 있는 봉지 개수가 최대가 되는 수를 구하면 된다.

```
1 >>> import math
2 >>> math.gcd(60, 100, 80)
3 20
```

파이썬 3.9 버전부터는 math.gcd 의 인수로 여러개가 가능하지만 3.9 미만 버전에서는 2 개까지만 허용된다.

`math.gcd()` 함수로 최대공약수를 구했더니 20 이었다. 따라서 최대 20 봉지를 만들 수 있다. 각 봉지에 들어가는 사탕, 초콜릿, 젤리의 개수는 다음과 같이 전체 개수를 최대공약수 20 으로 나누면 구할 수 있다.

```
1 >>> 60/20, 100/20, 80/20
2 (3.0, 5.0, 4.0)
```

따라서 한 봉지당 사탕 3 개씩, 초콜릿 5 개씩, 젤리 4 개씩 담으면 된다.

math.lcm

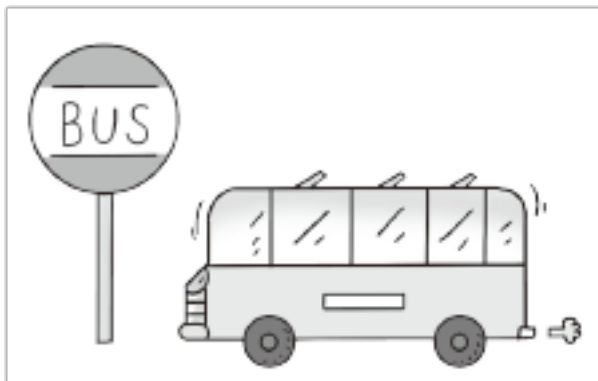
`math.lcm` 은 최소공배수 (lcm, least common multiple) 를 구할때 사용하는 함수이다.

`math.lcm()` 함수는 파이썬 3.9 버전부터 사용할 수 있다.

최소공배수란?

최소공배수란 두 수의 공통 배수 중 가장 작은 수를 말한다. 예를 들어 3 과 5 의 최소공배수는 15 가 된다.

어느 버스 정류장에 시내버스는 15 분마다 도착하고 마을버스는 25 분마다 도착한다고 한다. 오후 1 시에 두 버스가 동시에 도착했다고 할 때 두 버스가 동시에 도착할 다음 시각을 알려면 어떻게 해야 할까?



이 문제는 15 와 25 의 공통 배수 중 가장 작은 수, 즉 최소공배수를 구하면 바로 해결된다.

```
1 >>> import math
2 >>> math.lcm(15, 25)
3 75
```

`math.lcm()` 함수를 사용하여 최소공배수 75 를 구했다. 따라서 두 버스가 동시에 도착할 다음 시각은 75 분 후인 오후 2 시 15 분이다.

random

`random` 은 난수 (규칙이 없는 임의의 수) 를 발생시키는 모듈이다. `random` 과 `randint` 에 대해 알아보자.

다음은 0.0 에서 1.0 사이의 실수 중에서 난수 값을 돌려주는 예를 보여 준다.

```
1 >>> import random
2 >>> random.random()
3 0.53840103305098674
```

다음 예는 1 에서 10 사이의 정수 중에서 난수 값을 돌려준다.

```
1 >>> random.randint(1, 10)
2 6
```

다음 예는 1 에서 55 사이의 정수 중에서 난수 값을 돌려준다.

```
1 >>> random.randint(1, 55)
2 43
```

`random` 모듈을 사용해서 재미있는 함수를 하나 만들어 보자.

```
1 # random_pop.py
2 import random
3 def random_pop(data):
4     number = random.randint(0, len(data)-1)
5     return data.pop(number)
6
7 if __name__ == "__main__":
8     data = [1, 2, 3, 4, 5]
9     while data:
10         print(random_pop(data))
```

```
1 결과 값 :
2 2
3 3
4 1
5 5
6 4
```

위 `random_pop` 함수는 리스트의 요소 중에서 무작위로 하나를 선택하여 꺼낸 다음 그 값을 리턴한다. 물론 꺼낸 요소는 `pop` 메서드에 의해 사라진다.

`random_pop` 함수는 `random` 모듈의 `choice` 함수를 사용하여 다음과 같이 좀 더 직관적으로 만들 수도 있다.

```
1 def random_pop(data):
2     number = random.choice(data)
3     data.remove(number)
```

```
4 return number
```

`random.choice` 함수는 입력으로 받은 리스트에서 무작위로 하나를 선택하여 리턴한다.

리스트 요소를 무작위로 섞고 싶다면 다음처럼 `random.sample` 함수를 사용한다.

```
1 >>> a = [1,2,3,4,5]
2 >>> random.sample(a, len(a))
3 [5, 3, 2, 4, 1]
```

`random.sample` 함수에서 2 번째 인수인 `len(a)`는 무작위로 추출할 원소의 갯수를 의미한다. 만약 `random.sample(a, 3)`과 같이 사용한다면 `a` 리스트에서 무작위로 3 개를 추출하여 리턴할 것이다.

itertools.zip_longest

`itertools.zip_longest(*iterables, fillvalue=None)` 함수는 같은 개수의 자료형을 묶는 파이썬 내장 함수인 `zip()` 과 똑같이 동작한다. 하지만, `itertools.zip_longest()` 함수는 전달한 반복 가능 객체(*iterables)의 길이가 다르다면 긴 것을 기준으로 빠진 값은 `fillvalue`에 설정한 값으로 채울 수 있다.

유치원생 5 명에게 간식을 나누어 주고자 다음과 같은 파이썬 코드를 작성했다.

```
1 students = ['한민서', '황지민', '이영철', '이광수', '김승민']
2 snacks = ['사탕', '초콜릿', '젤리']
3
4 result = zip(students, snacks)
5 print(list(result))
```

그러나 간식 개수가 유치원생보다 적으므로 이 파이썬 코드를 실행하면 다음과 같은 결과가 나온다.

```
1 [('한민서', '사탕'), ('황지민', '초콜릿'), ('이영철', '젤리')]
```

`students` 와 `snacks` 의 개수가 다르므로 더 적은 `snacks` 의 개수만큼만 `zip()` 으로 묶게 된다. 하지만, `students` 가 `snacks` 보다 많더라도 다음처럼 부족한 `snacks` 는 '새우깡' 으로 채워 간식을 나누는 코드를 작성하려면 어떻게 해야 할까?

```
1 [('한민서', '사탕'), ('황지민', '초콜릿'), ('이영철', '젤리'), ('이광수', '새우깡'), ('김승민', '새우깡')]
```

`itertools.zip_longest()` 를 사용하면 개수가 많은 것을 기준으로 묶을 수 있다. 이때 부족한 항목은 `None` 으로 채우는데, 다음처럼 `fillvalue` 로 값을 지정하면 `None` 대신 다른 값으로 채울 수 있다.

```
1 import itertools
2
```

```
3 students = ['한민서', '황지민', '이영철', '이광수', '김승민']
4 snacks = ['사탕', '초콜릿', '젤리']
5
6 result = itertools.zip_longest(students, snacks, fillvalue='새우깡')
7 print(list(result))
```

실행 결과는 다음과 같다.

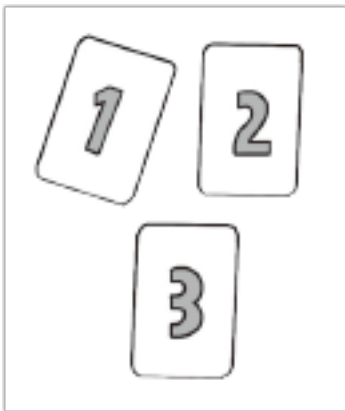
```
1 [('한민서', '사탕'), ('황지민', '초콜릿'), ('이영철', '젤리'), ('이광수', '새우깡'), ('김승민', '새우깡')]
```

itertools.permutation

`itertools.permutations(iterable, r)` 은 반복 가능 객체 (iterable) 중에서 `r` 개를 선택한 순열을 이터레이터로 리턴하는 함수이다.

이터레이터란 반복 가능한 객체를 의미한다.

1, 2, 3 숫자가 적힌 3 장의 카드에서 두 장의 카드를 꺼내 만들 수 있는 2 자리 숫자를 모두 구하려면 어떻게 해야 할까?



[1, 2, 3] 3 장의 카드 중 순서에 상관없이 2 장을 뽑는 경우의 수는 모두 3 가지이다 (조합).

- 1, 2
- 2, 3
- 1, 3

하지만, 이 문제에서는 2 자리 숫자이므로 이 3 가지에 순서를 더해 다음처럼 6 가지가 된다 (순열).

- 1, 2
- 2, 1

- 2,3
- 3,2
- 1,3
- 3,1

이 순열은 `itertools.permutations()` 를 사용하면 간단히 구할 수 있다.

```
1 >>> import itertools
2 >>> list(itertools.permutations(['1', '2', '3'], 2))
3 [('1', '2'), ('1', '3'), ('2', '1'), ('2', '3'), ('3', '1'), ('3', '2')]
```

따라서 만들 수 있는 2 자리 숫자는 다음과 같이 모두 6 가지이다.

```
1 >>> for a, b in itertools.permutations(['1', '2', '3'], 2):
2 ...     print(a+b)
3 ...
4 12
5 13
6 21
7 23
8 31
9 32
```

조합

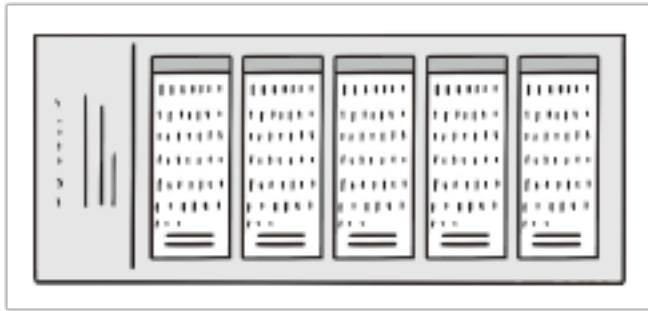
3 장의 카드에서 순서에 상관없이 2 장을 고르는 조합은 다음처럼 `itertools.combinations()` 를 사용하면 된다.

```
1 >>> import itertools
2 >>> list(itertools.combinations(['1', '2', '3'], 2))
3 [('1', '2'), ('1', '3'), ('2', '3')]
```

itertools.combination

`itertools.combinations(iterable, r)` 은 반복 가능 객체 (`iterable`) 중에서 `r` 개를 선택한 조합을 이터레이터로 리턴하는 함수이다.

1~45 중 서로 다른 숫자 6 개를 뽑는 로또 번호의 모든 경우의 수 (조합) 를 구하고 그 개수를 출력하려면 어떻게 해야 할까?



다음과 같이 `itertools.combinations()` 를 사용하면 45 개의 숫자 중 6 개를 선택하는 경우의 수를 구할 수 있다.

```
1 >>> import itertools
2 >>> it = itertools.combinations(range(1, 46), 6)
```

`itertools.combinations(range(1, 46), 6)` 은 1~45 의 숫자 중에서 6 개를 뽑는 경우의 수를 이터레이터로 리턴한다.

이터레이터 객체를 루프를 이용하여 출력하면 아마 끝도 없이 출력될 것이다. 궁금하다면 직접 실행해 봐도 좋다.

```
1 >>> for num in it:
2 ...     print(num)
3 ...
4 (1, 2, 3, 4, 5, 6)
5 (1, 2, 3, 4, 5, 7)
6 (1, 2, 3, 4, 5, 8)
7 (1, 2, 3, 4, 5, 9)
8 (1, 2, 3, 4, 5, 10)
9 (1, 2, 3, 4, 5, 11)
10 (1, 2, 3, 4, 5, 12)
11 (1, 2, 3, 4, 5, 13)
12 ...
```

하지만, 순환하여 출력하지 않고 이터레이터의 개수만 세려면 다음과 같이 하면 된다.

```
1 >>> len(list(itertools.combinations(range(1, 46), 6)))
2 8145060
```

선택할 수 있는 로또 번호의 가짓수는 8,145,060 이다.

여러분이 반드시 로또에 당첨되길 희망한다면 서로 다른 번호로 구성된 8,145,060 장의 로또를 사면 된다. 1 게임에 천 원이라 할 때 그 금액은 무려 81 억 4 천 5 백 6 만 원이다.

중복 조합

만약 로또 복권이 숫자 중복을 허용하도록 규칙이 변경된다면 경우의 수는 몇 개가 될까?

중복이 허용된다 함은 당첨 번호가 [1, 2, 3, 4, 5, 5] 처럼 5가 2 번 이상 나와도 되고 [1, 1, 1, 1, 1, 1] 처럼 1이 6 번 나와도 된다는 의미이다.

같은 숫자를 허용하는 중복 조합은 `itertools.combinations_with_replacement()` 를 사용하면 된다.

```
1 >>> len(list(itertools.combinations_with_replacement(range(1, 46), 6)))
2 15890700
```

당연히 중복을 허용하지 않을 때보다 훨씬 많은 경우의 수가 리턴되는 것을 확인할 수 있다.

functools.reduce

`functools.reduce(function, iterable)` 은 `function` 을 반복 가능한 객체 (`iterable`) 의 요소에 차례대로 (왼쪽에서 오른쪽으로) 누적 적용하여 이 객체를 하나의 값으로 줄이는 함수이다.

다음은 입력 인수 `data` 의 요소를 모두 더하여 리턴하는 `add()` 함수이다.

```
1 def add(data):
2     result = 0
3     for i in data:
4         result += i
5     return result
6
7 data = [1, 2, 3, 4, 5]
8 result = add(data)
9 print(result) # 15 출력
```

`functools.reduce()` 를 사용하여 마찬가지로 동작하는 코드를 작성하려면 어떻게 해야 할까?

`functools.reduce()` 를 사용한 코드는 다음과 같다.

```
1 import functools
2
3 data = [1, 2, 3, 4, 5]
4 result = functools.reduce(lambda x, y: x + y, data)
5 print(result) # 15 출력
```

`functools.reduce()` 를 사용하면 `reduce()` 에 선언한 람다 함수를 `data` 요소에 차례대로 누적 적용하여 다음과 같이 계산한다.

```
1 (((((1+2)+3)+4)+5)
```

따라서 앞서 본 `add()` 함수와 동일한 역할을 하게 된다.

functools.reduce()로 최댓값 구하기

```
1 num_list = [3, 2, 8, 1, 6, 7]
2 max_num = functools.reduce(lambda x, y: x if x > y else y, num_list)
3 print(max_num) # 8 출력
```

[3, 2, 8, 1, 6, 7] 요소를 차례대로 `reduce()`의 람다 함수로 전달하여 두 값 중 큰 값을 선택하고 마지막에 남은 최댓값을 리턴한다.

최 소 값 은 `functools.reduce(lambda x, y: x if x < y else y, num_list)`로 구할 수 있다.

operator.itemgetter

`operator.itemgetter`는 주로 `sorted`와 같은 함수의 `key` 매개변수에 적용하여 다양한 기준으로 정렬할 수 있도록 도와주는 모듈이다.

예를 들어 학생의 이름, 나이, 성적 등의 정보를 저장한 다음과 같은 `students` 리스트가 있다고 하자.

```
1 students = [
2     ("jane", 22, 'A'),
3     ("dave", 32, 'B'),
4     ("sally", 17, 'B'),
5 ]
```

`students` 리스트에는 3개의 튜플이 있으며 각 튜플은 순서대로 이름, 나이, 성적에 해당하는 데이터로 이루어졌다. 이 리스트를 나이순으로 정렬하려면 어떻게 해야 할까?



이 문제는 다음처럼 `sorted()` 함수의 `key` 매개변수에 `itemgetter()`를 적용하면 쉽게 해결할 수 있다.


```

1 from operator import itemgetter
2
3 students = [
4     ("jane", 22, 'A'),
5     ("dave", 32, 'B'),
6     ("sally", 17, 'B'),
7 ]
8
9 result = sorted(students, key=itemgetter(1))
10 print(result)

```

이 파일을 실행하여 출력해 보면 다음과 같이 나이 순서대로 정렬한 것을 확인할 수 있다.

```
1 [('sally', 17, 'B'), ('jane', 22, 'A'), ('dave', 32, 'B')]
```

`itemgetter(1)` 은 `students` 의 아이템인 튜플의 2 번째 요소를 기준으로 정렬하겠다는 의미이다. 만약 `itemgetter(2)` 와 같이 사용한다면 성적순으로 정렬한다. 이번에는 `students` 의 요소가 다음처럼 딕셔너리일 때를 생각해 보자.

```

1 students = [
2     {"name": "jane", "age": 22, "grade": 'A'},
3     {"name": "dave", "age": 32, "grade": 'B'},
4     {"name": "sally", "age": 17, "grade": 'B'},
5 ]

```

딕셔너리일 때도 마찬가지로 `age` 를 기준으로 정렬해 보자. 이때도 마찬가지로 `itemgetter()` 를 적용하면 된다. 단, 이번에는 `itemgetter('age')` 처럼 딕셔너리의 키를 사용해야 한다. `itemgetter('age')` 는 딕셔너리의 키인 `age` 를 기준으로 정렬하겠다는 의미이다.

```

1 from operator import itemgetter
2
3 students = [
4     {"name": "jane", "age": 22, "grade": 'A'},
5     {"name": "dave", "age": 32, "grade": 'B'},
6     {"name": "sally", "age": 17, "grade": 'B'},
7 ]
8
9 result = sorted(students, key=itemgetter('age'))
10 print(result)

```

출력 결과는 다음과 같이 `age` 순으로 정렬된 것을 확인할 수 있다.

```
1 [{ 'name': 'sally', 'age': 17, 'grade': 'B'}, { 'name': 'jane', 'age': 22, 'grade': 'A'}, { 'name': 'dave', 'age': 32, 'grade': 'B'}]
```

operator.attrgetter()

students 리스트의 요소가 튜플이 아닌 Student 클래스의 객체라면 다음처럼 attrgetter() 를 적용하여 정렬해야 한다.

```

1  from operator import attrgetter
2
3  class Student:
4      def __init__(self, name, age, grade):
5          self.name = name
6          self.age = age
7          self.grade = grade
8
9  students = [
10     Student('jane', 22, 'A'),
11     Student('dave', 32, 'B'),
12     Student('sally', 17, 'B'),
13 ]
14
15 result = sorted(students, key=attrgetter('age'))

```

attrgetter('age') 는 Student 객체의 age 속성으로 정렬하겠다는 의미이다. 마찬가지로 attrgetter('grade') 와 같이 사용하면 성적순으로 정렬한다.

shutil

shutil 은 파일을 복사 (copy) 하거나 이동 (move) 할 때 사용하는 모듈이다.

작업 중인 파일을 자동으로 백업하는 기능을 구현하고자 c:\doit\ a.txt 파일을 c:\temp\ a.txt .bak이라는 이름으로 복사하는 프로그램을 만들고자 한다. 어떻게 만들어야 할까? c:\doit 디렉터리에 a.txt 파일을 만드는 중이며 백업용 c:\temp 디렉터리는 이미 만들었다고 가정한다.

다음은 shutil 을 사용한 방법이다.

```

1  import shutil
2
3  shutil.copy("c:/doit/a.txt", "c:/temp/a.txt.bak")

```

shutil.move

휴지통으로 삭제하는 기능을 구현하고자 c:\doit\ a.txt 파일을 c:\temp\ a.txt로 이동하려면 다음과 같이 코드를 작성한다.

```

1  import shutil
2
3  shutil.move("c:/doit/a.txt", "c:/temp/a.txt")

```

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 **glob** 이다.

디렉터리에 있는 파일들을 리스트로 만들기 - **glob(pathname)**

glob 모듈은 디렉터리 안의 파일들을 읽어서 리턴한다. *, ? 등 메타 문자를 써서 원하는 파일만 읽어 들일 수도 있다.

다음은 **C:/doit** 디렉터리에 있는 파일 중 이름이 **mark** 로 시작하는 파일을 모두 찾아서 읽어들이는 예이다.

```
1 >>> import glob
2 >>> glob.glob("c:/doit/mark*")
3 ['c:/doit\\marks1.py', 'c:/doit\\marks2.py', 'c:/doit\\marks3.py']
4 >>>
```

pickle

pickle 은 객체의 형태를 그대로 유지하면서 파일에 저장하고 불러올 수 있게 하는 모듈이다. 다음 예는 **pickle** 모듈의 **dump** 함수를 사용하여 딕셔너리 객체인 **data** 를 그대로 파일에 저장하는 방법을 보여 준다.

```
1 >>> import pickle
2 >>> f = open("test.txt", 'wb')
3 >>> data = {1: 'python', 2: 'you need'}
4 >>> pickle.dump(data, f)
5 >>> f.close()
```

다음은 **pickle.dump** 로 저장한 파일을 **pickle.load** 를 사용해서 원래 있던 딕셔너리 객체 (**data**) 상태 그대로 불러오는 예이다.

```
1 >>> import pickle
2 >>> f = open("test.txt", 'rb')
3 >>> data = pickle.load(f)
4 >>> print(data)
5 {2: 'you need', 1: 'python'}
```

위 예에서는 딕셔너리 객체를 사용했지만 어떤 자료형이든 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 변수 값을 가지고 있는데, os.environ 은 현재 시스템의 환경 변수 값을 리턴한다. 다음을 따라 해 보자.

```
1 >>> import os
2 >>> os.environ
3 environ({'PROGRAMFILES': 'C:\\Program Files', 'APPDATA': ... 생략 ...})
4 >>>
```

위 결과값은 필자의 시스템 정보이다. os.environ 은 환경 변수에 대한 정보를 딕셔너리 형태로 구성된 environ 객체로 리턴한다. 자세히 보면 여러 가지 유용한 정보를 찾을 수 있다.

돌려받은 객체는 다음과 같이 호출하여 사용할 수 있다. 다음은 필자 시스템의 PATH 환경 변수 내용이다.

```
1 >>> os.environ['PATH']
2 'C:\\ProgramData\\Oracle\\Java\\javapath;... 생략 ...'
```

디렉터리 위치 변경하기 - os.chdir

os.chdir 를 사용하면 다음과 같이 현재 디렉터리 위치를 변경할 수 있다.

```
1 >>> os.chdir("C:\\WINDOWS")
```

디렉터리 위치 돌려받기 - os.getcwd

os.getcwd 는 현재 자신의 디렉터리 위치를 리턴한다.

```
1 >>> os.getcwd()
2 'C:\\WINDOWS'
```

시스템 명령어 호출하기 - os.system

시스템 자체의 프로그램이나 기타 명령어를 파이썬에서 호출할 수도 있다. os.system("명령어") 처럼 사용한다. 다음은 현재 디렉터리에서 시스템 명령어 dir 을 실행하는 예이다.

```
1 >>> os.system("dir")
```

실행한 시스템 명령어의 결과값 돌려받기 - os.popen

os.popen 은 시스템 명령어를 실행한 결과값을 읽기 모드 형태의 파일 객체로 리턴한다.

```
1 >>> f = os.popen("dir")
```

읽어 들인 파일 객체의 내용을 보기 위해서는 다음과 같이 하면 된다.

```
1 >>> 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)	src 라는 이름의 파일을 dst 라는 이름으로 바꾼다.

zipfile

zipfile 은 여러 개의 파일을 zip 형식으로 합치거나 이를 해제할 때 사용하는 모듈이다.

다음과 같은 3 개의 텍스트 파일이 있다고 하자.

```
1 a.txt
2 b.txt
3 c.txt
```

이 3 개의 텍스트 파일을 하나로 합쳐 mytext.zip 이라는 파일을 만들고, 이 파일을 원래의 텍스트 파일 3 개로 해제하는 프로그램을 만들려면 어떻게 해야 할까?

zipfile.ZipFile() 을 사용하여 해결해 보자.

```
1 import zipfile
2
3 # 파일 합치기
4 with zipfile.ZipFile('mytext.zip', 'w') as myzip:
5     myzip.write('a.txt')
6     myzip.write('b.txt')
7     myzip.write('c.txt')
8
9 # 해제하기
10 with zipfile.ZipFile('mytext.zip') as myzip:
11     myzip.extractall()
```

ZipFile 객체의 write() 함수로 개별 파일을 추가할 수 있고 extractall() 함수를 사용하면 모든 파일을 해제할 수 있다. 합친 파일에서 특정 파일만 해제하고 싶다면 다음과 같이 extract() 함수를 사용하면 된다.

```
1 with zipfile.ZipFile('mytext.zip') as myzip:
2     myzip.extract('a.txt')
```

만약 파일을 압축하여 묶고 싶은 경우에는 `compression`, `compresslevel` 옵션을 사용할 수 있다.

```
1 with zipfile.ZipFile('mytext.zip', 'w', compression=zipfile.ZIP_LZMA,
2     compresslevel=9) as myzip:
3     (... 생략 ...)
```

`compression`에는 4 가지 종류가 있다.

- `ZIP_STORED` - 압축하지 않고 파일을 Zip 으로만 묶는다. 속도가 빠르다.
- `ZIP_DEFLATED` - 일반적인 ZIP 압축으로 속도가 빠르고 압축률은 낮다. (호환성이 좋다.)
- `ZIP_BZIP2` - bzip2 압축으로 압축률이 높고 속도가 느리다.
- `ZIP_LZMA` - lzma 압축으로 압축률이 높고 속도가 느리다. (7zip 과 동일한 알고리즘으로 알려져 있다.)

`compresslevel` 은 압축 수준을 의미하는 숫자값으로 1 ~ 9 를 사용한다. 1 은 속도가 가장 빠르고 압축률이 낮고, 9 는 속도는 가장 느리지만 최대 압축을 한다.

threading

스레드 프로그래밍은 초보 프로그래머가 구현하기에는 매우 어려운 기술이다. 여기에 잠시 소개했으니 눈으로만 살펴보고 넘어가자.

컴퓨터에서 동작하고 있는 프로그램을 프로세스 (Process) 라고 한다. 보통 1 개의 프로세스는 한 가지 일만 하지만 스레드 (Thread) 를 사용하면 한 프로세스 안에서 2 가지 또는 그 이상의 일을 동시에 수행할 수 있다.

간단한 예제로 설명을 대신하겠다.

```
1 # thread_test.py
2 import time
3
4 def long_task(): # 5초의 시간이 걸리는 함수
5     for i in range(5):
6         time.sleep(1) # 1초 간 대기한다.
7         print("working:%s\n" % i)
8
9 print("Start")
10
11 for i in range(5): # long_task를 5회 수행한다.
12     long_task()
13
14 print("End")
```

`long_task` 함수는 수행하는 데 5 초의 시간이 걸리는 함수이다. 위 프로그램은 이 함수를 총 5 번 반복해서 수행하는 프로그램이다. 이 프로그램은 5 초가 5 번 반복되니 총 25 초의 시간이 걸린다.

하지만 앞에서 설명했듯이 스레드를 사용하면 5 초의 시간이 걸리는 `long_task` 함수를 동시에 실행할 수 있으니 시간을 줄일 수 있다.

다음과 같이 프로그램을 수정해 보자.

```
1 # thread_test.py
2 import time
3 import threading # 스레드를 생성하기 위해서는 threading 모듈이 필요하다.
4
5 def long_task():
6     for i in range(5):
7         time.sleep(1)
8         print("working:%s\n" % i)
9
10 print("Start")
11
12 threads = []
13 for i in range(5):
14     t = threading.Thread(target=long_task) # 스레드를 생성한다.
15     threads.append(t)
16
17 for t in threads:
18     t.start() # 스레드를 실행한다.
19
20 print("End")
```

이와 같이 프로그램을 수정하고 실행해 보면 25 초 걸리던 작업이 5 초 정도에 수행되는 것을 확인할 수 있다. `threading.Thread` 를 사용하여 만든 스레드 객체가 동시 작업을 가능하게 해 주기 때문이다.

하지만 위 프로그램을 실행해 보면 “Start” 와 “End” 가 먼저 출력되고 그 이후에 스레드의 결과가 출력되는 것을 확인할 수 있다. 그리고 프로그램이 정상 종료되지 않는다. 우리가 기대하는 것은 “Start” 가 출력되고 그다음에 스레드의 결과가 출력된 후 마지막으로 “End” 가 출력되는 것이다.

이 문제를 해결하기 위해서는 다음과 같이 프로그램을 수정해야 한다.

```
1 # thread_test.py
2 import time
3 import threading
4
5 def long_task():
6     for i in range(5):
7         time.sleep(1)
8         print("working:%s\n" % i)
9
10 print("Start")
11
12 threads = []
13 for i in range(5):
14     t = threading.Thread(target=long_task)
```

```
15     threads.append(t)
16
17     for t in threads:
18         t.start()
19
20     for t in threads:
21         t.join() # join으로 스레드가 종료될 때까지 기다린다.
22
23     print("End")
```

스레드의 `join` 함수는 해당 스레드가 종료될 때까지 기다리게 한다. 따라서 위와 같이 수정하면 우리가 원하던 출력을 보게 된다.

tempfile

파일을 임시로 만들어서 사용할 때 유용한 모듈이 바로 `tempfile` 이다. `tempfile.mkstemp()` 는 중복되지 않는 임시 파일의 이름을 무작위로 만들어서 리턴한다.

```
1 >>> import tempfile
2 >>> filename = tempfile.mkstemp()
3 >>> filename
4 'C:\\WINDOWS\\TEMP\\~-275151-0'
```

`tempfile.TemporaryFile()` 은 임시 저장 공간으로 사용할 파일 객체를 리턴한다. 이 파일은 기본적으로 바이너리 쓰기 모드 (`wb`) 를 갖는다. `f.close()` 가 호출되면 이 파일은 자동으로 삭제된다.

```
1 >>> import tempfile
2 >>> f = tempfile.TemporaryFile()
3 >>> f.close()
```

traceback

`traceback` 은 프로그램 실행 중 발생한 오류를 추적하고자 할 때 사용하는 모듈이다.

다음과 같은 코드를 작성하여 실행해 보자.

```
1 def a():
2     return 1/0
3
4 def b():
5     a()
6
7 def main():
8     try:
9         b()
```



```

10     except:
11         print("오류가 발생했습니다.")
12
13 main()

```

프로그램 실행 결과는 다음과 같다.

```

1 오류가 발생했습니다.

```

main() 함수가 시작되면 b() 함수를 호출하고 b() 함수에서 다시 a() 함수를 호출하여 1을 0으로 나누므로 오류가 발생하여 “오류가 발생했습니다.” 라는 메시지를 출력했다.

이렇게 간단한 프로그램이 아니라 복잡한 파이썬 코드라면 어디에서 어떤 오류가 발생하는지 알기 어렵다.

이때 이 코드에서 오류가 발생한 위치와 원인을 정확히 판단할 수 있도록 코드를 업그레이드하려면 어떻게 해야 할까?

오류가 발생한 위치에 다음과 같이 `traceback` 모듈을 적용해 보자.

```

1 import traceback
2
3 def a():
4     return 1/0
5
6 def b():
7     a()
8
9 def main():
10    try:
11        b()
12    except:
13        print("오류가 발생했습니다.")
14        print(traceback.format_exc())
15
16 main()

```

오류가 발생한 위치에 `print(traceback.format_exc())` 문장을 추가했다. `traceback` 모듈의 `format_exc()` 함수는 오류 추적 결과를 문자열로 리턴하는 함수이다. 이렇게 코드를 수정하고 다시 프로그램을 실행하면 다음과 같이 출력될 것이다.

```

1 오류가 발생했습니다.
2 Traceback (most recent call last):
3   File "c:\doit\traceback_sample.py", line 14, in main
4     b()
5   File "c:\doit\traceback_sample.py", line 9, in b
6     a()
7   File "c:\doit\traceback_sample.py", line 5, in a
8     return 1/0
9 ZeroDivisionError: division by zero

```

오류 추적을 통해 `main()` 함수에서 `b()` 함수를 호출하고 `b()` 함수에서 다시 `a()` 함수를 호출하여 `1/0` 을 실행하려 하므로 `0` 으로 나눌 수 없다는 `ZeroDivisionError` 가 발생했음을 로그를 통해 정확하게 확인할 수 있다.

json

`json` 은 JSON 데이터를 쉽게 처리하고자 사용하는 모듈이다.

다음은 개인정보를 JSON 형태의 데이터로 만든 `myinfo.json` 파일이다.

[파일명: `myinfo.json`]

```
1 {
2     "name": "홍길동",
3     "birth": "0525",
4     "age": 30
5 }
```

인터넷으로 얻은 이 파일을 읽어 파이썬에서 처리할 수 있도록 딕셔너리 자료형으로 만들려면 어떻게 해야 할까?

JSON 파일을 읽어 딕셔너리로 변환하려면 다음처럼 `json` 모듈을 사용하면 된다.

```
1 >>> import json
2 >>> with open('myinfo.json') as f:
3 ...     data = json.load(f)
4 ...
5 >>> type(data)
6 <class 'dict'>
7 >>> data
8 {'name': '홍길동', 'birth': '0525', 'age': 30}
```

JSON 파일을 읽을 때는 이 예처럼 `json.load(파일객체)`를 사용한다. 이렇게 `load()` 함수는 읽은 데이터를 딕셔너리 자료형으로 리턴한다. 반대로 딕셔너리 자료형을 JSON 파일로 생성할 때는 다음처럼 `json.dump(딕셔너리, 파일 객체)`를 사용한다.

```
1 >>> import json
2 >>> data = {'name': '홍길동', 'birth': '0525', 'age': 30}
3 >>> with open('myinfo.json', 'w') as f:
4 ...     json.dump(data, f)
5 ...
6 >>>
```

이번에는 파이썬 자료형을 JSON 문자열로 만드는 방법에 대해서 알아보자.

```
1 >>> import json
2 >>> d = {"name": "홍길동", "birth": "0525", "age": 30}
```

```
3 >>> json_data = json.dumps(d)
4 >>> json_data
5 '{"name": "\ud64d\uae38\u2013", "birth": "0525", "age": 30}'
```

딕셔너리 자료형을 JSON 문자열로 만들려면 `json.dumps()` 함수를 사용하면 된다. 그런데 딕셔너리를 JSON 데이터로 변경하면 '홍길동' 과 같은 한글 문자열이 코드 형태로 표시된다. 왜냐하면 `dump()`, `dumps()` 함수는 기본적으로 데이터를 아스키 형태로 저장하기 때문이다. 유니코드 문자열을 아스키 형태로 저장하다 보니 한글 문자열이 마치 깨진 것처럼 보인다.

그러나 JSON 문자열을 딕셔너리로 다시 역변환하여 사용하는 데에는 전혀 문제가 없다. JSON 문자열을 딕셔너리로 변환할 때는 다음처럼 `json.loads()` 함수를 사용한다.

```
1 >>> json.loads(json_data)
2 {'name': '홍길동', 'birth': '0525', 'age': 30}
```

한글 문자열이 아스키 형태의 문자열로 변경되는 것을 방지하는 방법도 있다.

```
1 >>> d = {"name": "홍길동", "birth": "0525", "age": 30}
2 >>> json_data = json.dumps(d, ensure_ascii=False)
3 >>> json_data
4 '{"name": "홍길동", "birth": "0525", "age": 30}'
5 >>> json.loads(json_data)
6 {'name': '홍길동', 'birth': '0525', 'age': 30}
```

이처럼 `ensure_ascii=False` 옵션을 사용하면 된다. 이 옵션은 데이터를 저장할 때 아스키 형태로 변환하지 않겠다는 뜻이다.

출력되는 JSON 문자열을 보기 좋게 정렬하려면 다음처럼 `indent` 옵션을 추가하면 된다.

```
1 >>> d = {"name": "홍길동", "birth": "0525", "age": 30}
2 >>> print(json.dumps(d, indent=2, ensure_ascii=False))
3 {
4     "name": "홍길동",
5     "birth": "0525",
6     "age": 30
7 }
```

그리고 딕셔너리 외에 리스트나 튜플처럼 다른 자료형도 JSON 문자열로 바꿀 수 있다.

```
1 >>> json.dumps([1,2,3])
2 '[1, 2, 3]'
3 >>> json.dumps((4,5,6))
4 '[4, 5, 6]'
```

urllib

`urllib` 은 URL 을 읽고 분석할 때 사용하는 모듈이다.

브라우저로 위키독스의 특정 페이지를 읽으려면 다음과 같이 요청하면 된다.

```
1 https://wikidocs.net/ 페이지 번호 (예: https://wikidocs.net/12)
```

그러면 오프라인으로도 읽을 수 있도록 페이지 번호를 입력받아 위키독스의 특정 페이지를 `wikidocs_페이지번호.html` 파일로 저장하는 함수는 어떻게 만들어야 할까?

URL 을 호출하여 원하는 리소스를 얻으려면 `urllib` 모듈을 사용해야 한다.

```
1 import urllib.request
2
3 def get_wikidocs(page):
4     resource = 'https://wikidocs.net/{}'.format(page)
5     with urllib.request.urlopen(resource) as s:
6         with open('wikidocs_%s.html' % page, 'wb') as f:
7             f.write(s.read())
```

`get_wikidocs(page)` 함수는 위키독스의 페이지 번호를 입력받아 해당 페이지의 리소스 내용을 파일로 저장하는 함수이다. 이 코드에서 보듯이 `urllib.request.urlopen(resource, context=context)` 로 `s` 객체를 생성하고 `s.read()` 로 리소스 내용 전체를 읽어 이를 저장할 수 있다. 예를 들어 `get_wikidocs(12)` 라고 호출하면 <https://wikidocs.net/12> 웹 페이지를 `wikidocs_12.html` 라는 파일로 저장한다.

webbrowser

`webbrowser` 는 파이썬 프로그램에서 시스템 브라우저를 호출할 때 사용하는 모듈이다.

개발 중 궁금한 내용이 있어 파이썬 문서를 참고하려 한다. 이를 위해 <https://python.org> 사이트를 새로운 웹 브라우저로 열려면 코드를 어떻게 작성해야 할까?



파이썬으로 웹 페이지를 새 창으로 열려면 `webbrowser` 모듈의 `open_new()` 함수를 사용한다.

```
1 import webbrowser
2
3 webbrowser.open_new('http://python.org')
```

이미 열린 브라우저로 원하는 사이트를 열고 싶다면 다음처럼 `open_new()` 대신 `open()` 을 사용하면 된다.

```
1 webbrowser.open('http://python.org')
```

파이썬 라이브러리에 대해서 보다 깊게 공부하고 싶다면 필자가 집필한 다음의 책을 추천한다.

- [점프 투 파이썬 - 라이브러리 예제 편](#)

05-7 외부 라이브러리

파이썬 설치 시 기본으로 설치되는 라이브러리를 파이썬 표준 라이브러리라고 한다. 이번에 소개하는 외부 라이브러리는 파이썬 표준 라이브러리가 아니므로 사용하려면 먼저 **pip** 도구를 이용하여 설치해야 한다.

pip 은 “핍” 이라 읽는다.

pip

pip 은 파이썬 모듈이나 패키지를 쉽게 설치할 수 있도록 도와주는 도구이다. **pip** 으로 파이썬 프로그램을 설치하면 의존성 있는 모듈이나 패키지를 함께 설치해 주기 때문에 매우 편리하다. 예를 들어 **B** 라는 파이썬 패키지를 설치하려면 **A** 라는 패키지가 먼저 설치되어야 하는 규칙이 있다고 할 때 **pip** 을 이용하면 **B** 패키지를 설치할 때 **A** 패키지도 자동으로 함께 설치된다.

pip 사용법에 대해서 간략하게 알아보자.

pip install

PyPI(Python Package Index) 는 파이썬 소프트웨어가 모인 저장 공간이다. 현재 이곳에는 100,000 건 이상의 파이썬 패키지가 등록되어 있으며 이곳에 등록된 파이썬 패키지는 누구나 내려받아 사용할 수 있다. 이곳에서 직접 내려받아 설치해도 되지만 **pip** 을 이용하면 다음과 같이 간편하게 설치를 할 수 있다.

```
1 pip install SomePackage
```

여기서 **SomePackage** 는 내려받을 수 있는 특정 패키지를 뜻한다.

pip uninstall

설치한 패키지를 삭제하고 싶다면 다음의 명령어로 삭제할 수 있다.

```
1 pip uninstall SomePackage
```

특정 버전으로 설치하기

다음과 같이 버전을 지정하여 설치할 수도 있다. 다음 명령어를 실행하면 1.0.4 버전의 `SomePackage` 를 설치한다.

```
1 pip install SomePackage==1.0.4
```

다음처럼 버전을 생략하면 최신 버전을 설치한다.

```
1 pip install SomePackage
```

최신 버전으로 업그레이드하기

패키지를 최신 버전으로 업그레이드하려면 `-upgrade` 옵션과 함께 사용한다.

```
1 pip install --upgrade SomePackage
```

설치된 패키지 확인하기

다음 명령은 `pip` 을 이용하여 설치한 패키지 목록을 출력한다.

```
1 pip list
```

다음과 같이 설치된 패키지 목록을 출력할 것이다.

1	Package	Version
2	-----	-----
3	amqp	2.1.4
4	anyjson	0.3.3
5	billiard	3.3.0.23
6	celery	3.1.0
7	defusedxml	0.4.1
8	diff-match-patch	20121119
9	(... 생략 ...)	

Faker

이번에는 `pip` 을 사용하여 유용한 외부 라이브러리중 하나인 `faker` 를 설치하고 사용해 보자. `faker` 는 테스트용 가짜 데이터를 생성할 때 사용하는 라이브러리이다.

`faker` 라이브러리는 `pip` 을 이용하여 설치해야 한다.

```
1 C:\> pip install Faker
```

Faker 사용해 보기

만약 다음과 같은 형식의 테스트 데이터 30 건이 필요하다고 가정해 보자. 직접 데이터를 작성하지 말고 좀 더 편리한 방법으로 테스트 데이터를 만들려면 어떻게 해야 할까?

```
1 [(이름1, 주소1), (이름2, 주소2), ..., (이름30, 주소30)]
```

테스트 데이터는 **faker** 를 사용하면 아주 쉽게 만들 수 있다. 이름은 다음처럼 만들 수 있다.

```
1 >>> from faker import Faker
2 >>> fake = Faker()
3 >>> fake.name()
4 'Matthew Estrada'
```

한글 이름이 필요하다면 다음과 같이 한국을 의미하는 **ko-KR** 을 전달하여 **fake** 객체를 생성하면 된다.

```
1 >>> fake = Faker('ko-KR')
2 >>> fake.name()
3 '김 하은'
```

주소는 다음과 같이 만들 수 있다.

```
1 >>> fake.address()
2 '충청북도 수원시 잠실6길 (경자주이읍)'
```

따라서 이름과 주소를 쌍으로 하는 30 건의 테스트 데이터는 다음과 같이 만들 수 있다.

```
1 >>> test_data = [(fake.name(), fake.address()) for i in range(30)]
```

실행 결과는 다음과 같다.

```
1 >>> test_data
2 [('이예진', '인천광역시 동대문구 연주거리 (경자김면)'), ('윤도윤', '광주광역시 서초구 삼성로 (주원최박리)'), ('서동현', '인천광역시 관악구 잠실가 (민석엄김마을)'), ('김광수', '울산광역시 양천구 서초대로'), ('이예은', '광주광역시 성북구 개포522길'), ('김병철', '전라북도 춘천시 양재천52로 (현숙김읍)'), ('임서윤', '제주특별자치도 중평군 봉은사3로 (지민이윤리)'), ('김지연', '충청북도 의정부시 학동64가'), ('나지혜', '대구광역시 북구 도산대717거리'), ('박하은', '서울특별시 성동구 서초중앙길'), ('김영환', '부산광역시 성북구 강남대52가'), ('이은주', '대구광역시 동대문구 도산대로'), ('지서현', '전라남도 보령시 잠실0가 (미영김마을)'), ('김예은', '인천광역시 관악구 삼성길'), ('김현정', '강원도 금산군 서초대101가'), ('김영자', '전라남도 부천시 소사구 압구정로 (미영이면)'), ('김예원', '충청남도 안양시 동안구 가락5로 (예은이읍)'), ('이지후', '경상북도 부천시 테헤란길'), ('김민준', '대구광역시 노원구 서초대가'), ('이상호', '광주광역시 강서구 삼성6로'), ('박보람', '울산광역시 강서구 석촌호수606길 (성진이조읍)'), ('이하은', '전라남도 서천군 선릉거리'), ('박미영', '경상남도 파주시 삼성로 (병철안이동)'), ('황병철', '강원도 청주시 청원구 영동대1길 (하윤손리)'), ('김영순', '충청남도 안양시 동안구 반포대가 (수진배동)'), ('박성현', '전라남도 서산시 가락27길 (준영박문읍)'), ('김성호', '경상남도 영월군 학동거리'), ('백지우', '경기도 계룡시 서초대1로'), ('권유진', '경기도 양주시 서초중앙313가 (춘자나리)'), ('윤서준', '경상남도 청주시 서원구 서초대64가')]
```


Faker 활용

faker 는 앞서 살펴본 name, address 이외에 다른 항목도 제공한다. 대표적인 몇 가지만 알아보자.

항목	설명
fake.name()	이름
fake.address()	주소
fake.postcode()	우편 번호
fake.country()	국가명
fake.company()	회사명
fake.job()	직업명
fake.phone_number()	휴대 전화 번호
fake.email()	이메일 주소
fake.user_name()	사용자명
fake.pyint(min_value=0, max_value=100)	0 부터 100 사이의 임의의 숫자
fake.ipv4_private()	IP 주소
fake.text()	임의의 문장 (한글 임의의 문장은 <code>fake.catch_phrase()</code> 사용)
fake.color_name()	색상명

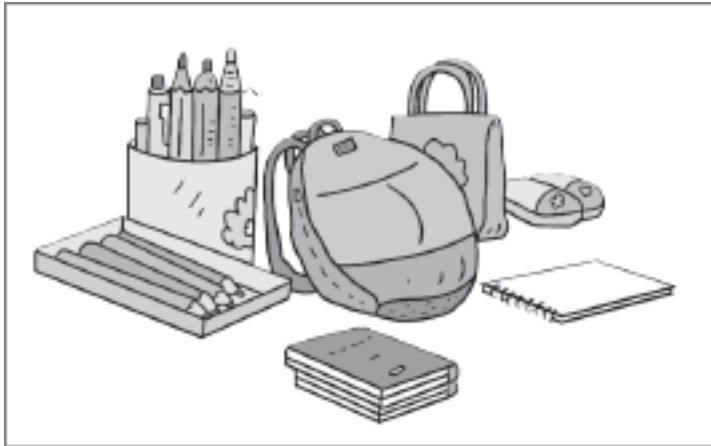
sympy

sympy 는 방정식 기호 (symbol) 를 사용하게 해주는 외부 라이브러리이다. 마찬가지로 pip 을 이용하여 sympy 를 설치하자.

```
1 C:\> pip install sympy
```

sympy 사용해 보기

시윤이는 가진 돈의 2/5 로 학용품을 샀다고 한다. 이때 학用品을 사는 데 쓴 돈이 1,760 원이라면 남은 돈은 어떻게 구하면 될까?



이 문제는 연습장과 연필만 있으면 쉽게 구할 수 있는 일차방정식 문제이다. 파이썬으로는 다음처럼 `sympy` 를 사용하면 방정식을 쉽게 풀 수 있다. 먼저 다음과 같이 `fractions` 모듈과 `sympy` 모듈이 필요하다.

```
1 >>> from fractions import Fraction
2 >>> import sympy
```

시윤이가 가진 돈을 x 라 하면 `sympy` 모듈을 사용하여 다음과 같이 표현할 수 있다.

```
1 >>> x = sympy.symbols("x")
```

`sympy.symbols()` 는 x 처럼 방정식에 사용하는 미지수를 나타내는 기호를 생성할 때 사용한다.

여러개의 기호 사용하기

x, y 두 개의 미지수가 필요하다면 다음처럼 표현할 수 있다.

```
1 x, y = sympy.symbols('x y')
```

시윤이가 가진 돈의 $2/5$ 가 1,760 원, 즉 일차방정식 $x * (2/5) = 1760$ 이므로 이를 코드로 표현하면 다음과 같다.

```
1 >>> f = sympy.Eq(x*Fraction('2/5'), 1760)
```

`sympy.Eq(a, b)` 는 a 와 b 가 같다는 방정식이다. 여기서 사용한 `Fraction` 은 유리수를 표현할 때 사용하는 파이썬 클래스로, $2/5$ 를 정확하게 계산하고자 사용했다.

fractions.Fraction

파이썬에서 유리수 연산을 정확하게 하려면 `fractions.Fraction` 을 사용해야 한다.

```
1 >>> from fractions import Fraction
```

유리수는 다음처럼 `Fraction(분자, 분모)` 형태로 만들 수 있다.

```
1 >>> a = Fraction(1, 5)
2 >>> a
3 Fraction(1, 5)
```

또는 다음과 같이 `Fraction('분자/분모')` 처럼 문자열로 만들 수도 있다.

```
1 >>> a = Fraction('1/5')
2 >>> a
3 Fraction(1, 5)
```

`f` 라는 방정식을 세웠으므로 `sympy.solve(f)` 로 `x` 에 해당하는 값을 구할 수 있다.

```
1 >>> result = sympy.solve(f)
2 >>> result
3 [4400]
```

방정식의 해는 여러 개일 수 있으므로 `solve()` 함수는 결과값으로 리스트를 리턴한다. 결과를 보면 시윤이가 원래 가진 돈이 4,400 원임을 알 수 있다. 따라서 남은 돈은 다음처럼 가진 돈에서 1,760 원을 빼면 된다.

```
1 >>> remains = result[0] - 1760
2 >>> remains
3 2640
```

지금까지 내용을 종합한 풀이는 다음과 같다.

```
1 from fractions import Fraction
2 import sympy
3
4 # 가지고 있던 돈을 x라고 하자.
5 x = sympy.symbols("x")
6
7 # 가지고 있던 돈의 2/5가 1760원이므로 방정식은 x * (2/5) = 1760 이다.
8 f = sympy.Eq(x*Fraction('2/5'), 1760)
9
10 # 방정식을 만족하는 값(result)을 구한다.
11 result = sympy.solve(f) # 결과값은 리스트
12
13 # 남은 돈은 다음과 같이 가지고 있던 돈에서 1760원을 빼면 된다.
14 remains = result[0] - 1760
15
16 print('남은 돈은 {}원 입니다.'.format(remains))
```

프로그램을 실행한 결과는 다음과 같다.

```
1 남 은 돈 은 2640 원 입 니 다 .
```

sympy 활용

$x^2 = 1$ 과 같은 이차방정식의 해를 구해보자.

```
1 >>> import sympy
2 >>> x = sympy.symbols("x")
3 >>> f = sympy.Eq(x**2, 1)
4 >>> sympy.solve(f)
5 [-1, 1]
```

또한, 다음과 같은 연립방정식의 해도 구할 수 있다.

```
1 x + y = 10
2 x - y = 4
```

```
1 >>> import sympy
2 >>> x, y = sympy.symbols('x y')
3 >>> f1 = sympy.Eq(x+y, 10)
4 >>> f2 = sympy.Eq(x-y, 4)
5 >>> sympy.solve([f1, f2])
6 {x: 7, y: 3}
```

미지수가 2 개 이상이라면 결괏값이 리스트가 아닌 딕셔너리임에 주의하자.

05 장 연습문제

Q1. UpgradeCalculator

다음은 Calculator 클래스이다.

```
1 class Calculator:
2     def __init__(self):
3         self.value = 0
4
5     def add(self, val):
6         self.value += val
```

위 클래스를 상속하는 UpgradeCalculator 를 만들고 값을 뺄 수 있는 minus 메서드를 추가해 보자. 즉 다음과 같이 동작하는 클래스를 만들어야 한다.

```
1 cal = UpgradeCalculator()
2 cal.add(10)
3 cal.minus(7)
4
5 print(cal.value) # 10에서 7을 뺀 3을 출력
```

Q2. MaxLimitCalculator

객체변수 value 가 100 이상의 값은 가질 수 없도록 제한하는 MaxLimitCalculator 클래스를 만들어 보자. 즉 다음과 같이 동작해야 한다.

```
1 cal = MaxLimitCalculator()
2 cal.add(50) # 50 더하기
3 cal.add(60) # 60 더하기
4
5 print(cal.value) # 100 출력
```

단 반드시 다음과 같은 Calculator 클래스를 상속해서 만들어야 한다.

```
1 class Calculator:
2     def __init__(self):
3         self.value = 0
```

```
4
5     def add(self, val):
6         self.value += val
```

Q3. 참과 거짓 예측하기

다음 결과를 예측해 보자.

하나.

```
1 >>> all([1, 2, abs(-3)-3])
```

둘.

```
1 >>> chr(ord('a')) == 'a'
```

Q4. 음수 제거하기

filter 와 lambda 를 사용하여 리스트 [1, -2, 3, -5, 8, -3] 에서 음수를 모두 제거해 보자.

Q5. 16 진수를 10 진수로 변경하기

234 라는 10 진수의 16 진수는 다음과 같이 구할 수 있다.

```
1 >>> hex(234)
2 '0xea'
```

이번에는 반대로 16 진수 문자열 0xea 를 10 진수로 변경해 보자.

내장 함수 int 를 활용해 보자.

Q6. 리스트 항목마다 3 곱하여 리턴하기

map 과 lambda 를 사용하여 [1, 2, 3, 4] 리스트의 각 요소값에 3 이 곱해진 리스트 [3, 6, 9, 12] 를 만들어 보자.

Q7. 최댓값과 최솟값의 합

다음 리스트의 최댓값과 최솟값의 합을 구해 보자.

```
1 [-8, 2, 7, 5, -3, 5, 0, 1]
```

Q8. 소수점 반올림하기

17 / 3 의 결과는 다음과 같다.

```
1 >>> 17 / 3
2 5.666666666666667
```

위와 같은 결괏값 5.666666666666667 을 소숫점 4 자리까지만 반올림하여 표시해 보자.

Q9. 디렉터리 이동하고 파일 목록 출력하기

os 모듈을 사용하여 다음과 같이 동작하도록 코드를 작성해 보자.

1. C:\doit 디렉터리로 이동한다.
2. dir 명령을 실행하고 그 결과를 변수에 담는다.
3. dir 명령의 결과를 출력한다.

Q10. 파일 확장자가.py 인 파일만 찾기

glob 모듈을 사용하여 C:\doit 디렉터리의 파일 중 확장자가.py 인 파일만 출력하는 프로그램을 작성해 보자.

Q11. 날짜 표시하기

time 모듈을 사용하여 현재 날짜와 시간을 다음과 같은 형식으로 출력해 보자.

```
1 2018/04/03 17:20:32
```

Q12. 로또 번호 생성하기

`random` 모듈을 사용하여 로또 번호 (1~45 사이의 숫자 6 개) 를 생성해 보자 (단 중복된 숫자가 있으면 안 됨).

Q13. 누나는 영철이보다 며칠 더 먼저 태어났을까?

영철이 누나의 생일은 1995 년 11 월 20 일이고 영철이의 생일은 1998 년 10 월 6 일이다. 영철이 누나는 영철이보다 며칠 더 먼저 태어났을까?

`datetime.date` 를 사용해 보자.

Q14. 기록순으로 정렬하기

다음은 1 학년 3 반 학생들의 100m 달리기 기록이다.

```
1 data = [('윤서현', 15.25),
2         ('김예지', 13.31),
3         ('박예원', 15.34),
4         ('송순자', 15.57),
5         ('김시우', 15.48),
6         ('배속자', 17.9),
7         ('전정웅', 13.39),
8         ('김혜진', 16.63),
9         ('최보람', 17.14),
10        ('한지영', 14.83),
11        ('이성호', 17.7),
12        ('김옥순', 16.71),
13        ('황민지', 17.65),
14        ('김영철', 16.7),
15        ('주병철', 15.67),
16        ('박상현', 14.16),
17        ('김영순', 14.81),
18        ('오지아', 15.13),
19        ('윤지은', 16.93),
20        ('문재호', 16.39)]
```

기록순으로 `data` 를 정렬해 보자.

`operator.itemgetter` 를 사용해 보자.

Q15. 청소 당번 2 명 뽑기

다음 4 명의 학생 중 청소 당번 2 명을 뽑을 수 있는 경우의 수를 모두 나열하시오.

```
1 ['나지혜', '성성민', '윤지현', '김정숙']
```

`itertools.combination` 을 사용해 보자.

Q16. 문자열 나열하기

“abcd” 문자열을 나열하는 경우의 수를 모두 출력하시오.

예) abcd, abdc, adcb, ...

`itertools.permutation` 을 사용해 보자.

Q17. 5 명에게 할일 부여하기

다음 5 명이 있다.

```
1 ['김승현', '김진호', '강춘자', '이예준', '김현주']
```

그리고 해야할 일은 다음처럼 3 가지가 있다.

```
1 ['청소', '빨래', '설거지']
```

5 명을 무작위로 섞어 앞의 3 명에게 차례로 해야할 일인 ["청소", "빨래", "설거지"]를 지정하고 나머지 2 명에게는 “휴식”을 지정할 수 있는 프로그램을 작성하시오.

`random.shuffle` 과 `itertools.zip_longest` 를 사용해 보자.

Q18. 벽에 타일 붙이기

가로 길이는 200cm 이고 세로 길이는 80cm 인 벽이 있다. 이 벽에 가능한 한 큰 정사각형 모양의 타일을 붙이려고 한다. 이때 붙이려는 타일 한 선의 길이와, 붙이는데 필요한 타일의 개수를 구하시오.

`math.gcd` 를 사용해 보자.

06 장 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. 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 같지 않을까?

“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 여기에서 가장 중요한 것은 자신의 수준에 맞는 소스를 찾는 일이다. 그래서 이 장에서는 아주 쉬운 예제부터 시작해서 차츰 수준을 높여 실용적인 예제까지 다루려고 노력하였다. 공부한 내용을 어떻게 활용하는가는 여러분의 몫이다.

필자는 예전에 프로그래밍을 막 시작한 사람에게 구구단 프로그램을 짜 보라고 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 설명해 주었다.

- 함수 이름은? `gugu` 로 짓자
- 입력받는 값은? 2
- 출력하는 값은? 2단 (2, 4, 6, 8, ..., 18)
- 결과는 어떤 형태로 저장하지? 연속된 자료형이니까 리스트!

독자들도 함께 따라 해 보기 바란다.

1. 먼저 에디터를 열고 다음과 같이 입력한다. `gugu` 라는 함수에 2를 입력값으로 주면 `result` 라는 변수에 결과값을 넣으라는 뜻이다.

```
1 result = gugu(2)
```

2. 이제 결과값을 어떤 형태로 받을 것인지 고민해 보자. 2단이니까 2, 4, 6, ... 18까지 갈 것이다. 이런 종류의 데이터는 리스트 자료형이 딱이다. 따라서 `result = [2, 4, 6, 8, 10, 12, 14, 16, 18]` 같은 결과를 얻는 것

이 좋겠다는 생각을 먼저 하고 나서 프로그래밍을 시작하는 것이 필요하다. 이런 식으로 머릿속에 그림이 그려지기 시작하면 의외로 생각이 가볍게 좁혀지는 것을 느낄 수 있을 것이다.

3. 어떻게 만들지 생각해 봤으니 1 번에서 입력한 문장은 지우고 진짜 프로그램을 짜 보자. 일단 이름을 `gugu` 로 지은 함수를 다음과 같이 만든다.

```
1 def gugu(n):
2     print(n)
```

위와 같은 함수를 만들고 `gugu(2)` 처럼 실행하면 2 를 출력하게 된다. 즉 입력값으로 2 가 잘 들어오는지 확인하는 것이다.

4. 이제 곱값을 담은 리스트를 하나 생성하자. 앞에서 작성한 `print(n)` 은 입력이 잘되는지 를 확인하기 위한 것이었으므로 지워도 좋다.

```
1 def gugu(n):
2     result = []
```

5. 다음으로 `result` 에 2, 4, 6, ... 18 을 어떻게 넣어야 할지 생각해 보자. 필자는 다음과 같이 리스트에 요소를 추가하는 `append` 내장 함수를 사용하기로 결정했다.

```
1 def gugu(n):
2     result = []
3     result.append(n*1)
4     result.append(n*2)
5     result.append(n*3)
6     result.append(n*4)
7     result.append(n*5)
8     result.append(n*6)
9     result.append(n*7)
10    result.append(n*8)
11    result.append(n*9)
12    return result
13
14 print(gugu(2))
```

결괏값: [2, 4, 6, 8, 10, 12, 14, 16, 18]

정말 무식한 방법이지만 입력값 2 를 주었을 때 원하는 곱값을 얻을 수 있었다.

6. 그런데 위 함수는 반복이 너무 많다. 가만히 보면 `result.append(n* \square)` 의 \square 위치에 1 부터 9 까지 숫자만 다르게 들어가 있다는 것을 알 수 있다. 똑같은 일을 반복할 때는 “반복문” 을 사용한다고 했다. 그렇다면 1 부터 9 까지 출력해 주는 반복문을 만들면 되지 않을까?

대화형 인터프리터를 열고 다음과 같이 테스트해 보았다.

```
1 >>> i = 1
2 >>> while i < 10:
3     ...     print(i)
```

```
4 ...     i = i + 1
```

```
1  곱  곱 값 :
2  1
3  2
4  3
5  4
6  5
7  6
8  7
9  8
10 9
```

아주 만족스러운 결과이다. 이제 위 소스 코드를 `gugu` 함수에 적용해 보자.

7. 이상의 생각을 바탕으로 완성한 `gugu` 함수는 다음과 같다.

```
1 def gugu(n):
2     result = []
3     i = 1
4     while i < 10:
5         result.append(n * i)
6         i = i + 1
7     return result
```

다음과 같이 테스트해 보자.

```
1 print(gugu(2))
```

곱값: [2, 4, 6, 8, 10, 12, 14, 16, 18]

결과는 대만족이다.

사실 `gugu` 함수는 위와 같은 과정을 거치지 않고도 바로 만들 수 있는 독자들이 많을 것이다. 하지만 더 복잡한 함수를 만들 때는 위와 같이 구체적이고 단계적으로 접근하는 방식이 매우 도움이 된다. 프로그래밍을 할 땐 매우 구체적으로 접근해야 머리가 덜 아프다는 것을 기억하자. 자, 이제 다양한 예제를 접해 보며 여러분 나름대로 멋진 생각을 해보기 바란다.

06-2 3 과 5 의 배수 합하기

자, 다음 문제를 어떻게 풀면 좋을지 생각해 보자.

```
1 10 미만의 자연수에서 3과 5의 배수를 구하면 3, 5, 6, 9이다. 이들의 총합은 23이다
2 1000 미만의 자연수에서 3의 배수와 5의 배수의 총합을 구하라.
```

- 입력 받는 값은? 1 부터 999 까지 (1000 미만의 자연수)
- 출력하는 값은? 3 의 배수와 5 의 배수의 총합
- 생각해 볼 것은?
 1. 3 의 배수와 5 의 배수는 어떻게 찾지?
 2. 3 의 배수와 5 의 배수가 겹칠 때는 어떻게 하지?

이 문제를 풀기 위한 중요 포인트는 두 가지이다. 한 가지는 1000 미만의 자연수를 구하는 방법이고 또 다른 한 가지는 3 과 5 의 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 초깃값 1 을 준 후 루프를 돌리며 1 씩 증가시켜서 999 까지 진행하는 방법이 가장 일반적인 방법일 것이다.

```
1 n = 1
2 while n < 1000:
3     print(n)
4     n += 1
```

또는 다음과 같이 좀 더 파이썬다운 range 함수를 사용할 수도 있다.

```
1 for n in range(1, 1000):
2     print(n)
```

두 가지 예 모두 실행하면 1 부터 999 까지 출력하는 것을 확인할 수 있다.

2. 1000 까지의 자연수를 차례로 구하는 방법을 알았으니 3 과 5 의 배수를 구하는 방법을 알아보자. 1000 미만의 자연수 중 3 의 배수는 다음과 같이 증가할 것이다.

```
3, 6, 9, 12, 15, 18, ..., 999
```

그렇다면 1 부터 1000 까지 수가 진행되는 동안 그 수가 3 의 배수인지는 어떻게 알 수 있을까? 1 부터 1000 까지의 수 중 3 으로 나누었을 때 나누어떨어지는 경우, 즉 3 으로 나누었을 때 나머지가 0 인 경우가 바로 3 의 배수이다. 따라서 다음과 같이 % 연산자를 사용하면 3 의 배수를 쉽게 찾을 수 있다.

```
1 for n in range(1, 1000):
2     if n % 3 == 0:
3         print(n)
```

그렇다면 5 의 배수는 $n \% 5$ 가 0 이 되는 수로 구할 수 있을 것이다.

3. 이러한 내용을 바탕으로 만든 최종 풀이는 다음과 같다.

```
1 result = 0
2 for n in range(1, 1000):
3     if n % 3 == 0 or n % 5 == 0:
4         result += n
5 print(result)
```

3 과 5 의 배수에 해당하는 수를 result 변수에 계속해서 더해 주었다.

이 문제에는 한 가지 함정이 있는데 3 으로도 5 로도 나누어지는 15 와 같은 수를 이중으로 더해서는 안 된다는 점이다. 따라서 15 와 같이 3 의 배수도 되고 5 의 배수도 되는 값이 이중으로 더해지지 않기 위해 or 연산자를 사용하였다.

다음 예는 15 와 같은 수를 이중으로 더하여 잘못된 결과를 출력하는 잘못된 풀이이다.

[잘못된 풀이]

```
1 result = 0
2 for n in range(1, 1000):
3     if n % 3 == 0:
4         result += n
5     if n % 5 == 0:
6         result += n
7 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.

Project Euler .net

About Archives Recent News Register Sign In

Problems Archives

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.

1 2 3 4 5 6 7 8 9 10 11 Go to Problem:

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	10001st 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 페이지징하기

A 씨는 게시판 프로그램을 작성하고 있다. 그런데 게시물의 총 건수와 한 페이지에 보여 줄 게시물 수를 입력으로 주었을 때 총 페이지 수를 출력하는 프로그램이 필요하다고 한다.

이렇게 게시판의 페이지 수를 보여 주는 것을 “페이징” 한다고 부른다.

- 함수 이름은? `get_total_page`
- 입력 받는 값은? 게시물의 총 건수 (**m**), 한 페이지에 보여줄 게시물 수 (**n**)
- 출력하는 값은? 총 페이지수

A 씨가 필요한 프로그램을 만들기 위해 입력값과 결과값이 어떻게 나와야 하는지 먼저 살펴보자. 게시물의 총 건수가 5 이고 한 페이지에서 보여 줄 게시물 수가 10 이면 총 페이지 수는 당연히 1 이 된다. 만약 게시물의 총 건수가 15 이고 한 페이지에서 보여 줄 게시물 수가 10 이라면 총 페이지 수는 2 가 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 페이지 수를 구하는 방법에 대해서만 알아보겠다.

1. 다음과 같이 총 건수 (**m**) 를 한 페이지에 보여 줄 게시물 수 (**n**) 로 나누고 1 을 더하면 총 페이지 수를 얻을 수 있다.

총 페이지 수 = (총 건수 / 한 페이지당 보여 줄 건수) + 1

2. 이러한 공식을 적용했을 경우 총 페이지 수가 표의 값처럼 구해지는지 확인해 보자 (**m** 을 **n** 으로 나눌 때 소수점 아래 자리를 버리기 위해 / 대신 // 연산자를 사용하였다).

```
1 def get_total_page(m, n):
2     return m // n + 1
3
4 print(get_total_page(5, 10))    # 1 출 력
5 print(get_total_page(15, 10))  # 2 출 력
6 print(get_total_page(25, 10))  # 3 출 력
7 print(get_total_page(30, 10))  # 4 출 력
```

첫 번째, 두 번째, 세 번째 케이스는 공식에 맞게 결과가 출력된다. 하지만 네 번째 케이스는 총 건수가 30 이고 한 페이지에 보여 줄 건수가 10 인데 4 가 출력되어 실패해 버렸다. 잘 생각해 보자. 총 건수가 30 이고 한 페이지에 보여 줄 건수가 10 이라면 당연히 총 페이지 수는 3 이되어야 한다.

3. 실패 케이스는 총 게시물 수와 한 페이지에 보여 줄 게시물 수를 나눈 나머지 값이 0 이 될 때 발생함을 유추할 수 있을 것이다. 이 실패 케이스를 해결하려면 다음과 같이 코드를 변경해야 한다.

```
1 def get_total_page(m, n):
2     if m % n == 0:
3         return m // n
4     else:
5         return m // n + 1
6
7 print(get_total_page(5, 10))
8 print(get_total_page(15, 10))
9 print(get_total_page(25, 10))
10 print(get_total_page(30, 10))
```

나누었을 때 나머지가 0 인 경우는 나누기의 몫만 돌려주고 그 이외의 경우에는 1 을 더하여 돌려주도록 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. 다음 명령을 실행했을 때 메모를 추가할 수 있도록 만들어 보자.

```
python memo.py -a "Life is too short"
```

memo.py 는 우리가 작성할 파이썬 프로그램 이름이다. -a 는 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.

```
1 # C:/doit/memo.py
2 import sys
3
4 option = sys.argv[1]
5 memo = sys.argv[2]
6
7 print(option)
8 print(memo)
```

sys.argv 는 프로그램을 실행할 때 입력된 값을 읽어 들일 수 있는 파이썬 라이브러리이다. sys.argv[0] 는 입력받은 값 중에서 파이썬 프로그램 이름인 memo.py 이므로 우리가 만들려는 기능에는 필요 없는 값이다. 그리고 순서대로 sys.argv[1] 은 프로그램 실행 옵션이 되고 sys.argv[2] 는 메모 내용이 된다.

2. memo.py 를 작성했다면 다음 명령을 수행해 보자.

memo.py 는 C:\doit 디렉터리에 저장한다

```
1 C:\doit>python memo.py -a "Life is too short"
2 -a
3 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.

```
1 # c:/doit/memo.py
2 import sys
3
4 option = sys.argv[1]
5
6 if option == '-a':
7     memo = sys.argv[2]
8     f = open('memo.txt', 'a')
9     f.write(memo)
10    f.write('\n')
11    f.close()
```

옵션이 -a 인 경우에만 memo 값을 읽어 memo.txt 파일에 그 값을 쓰도록 코드를 작성했다. 여기에서 메모는 항상 새로운 내용이 작성되는 것이 아니라 한 줄씩 추가되어야 하므로 파일열기 모드를 a로 했다. 그리고 메모를 추가할 때마다 다음 줄에 저장되도록 줄바꿈 문자 (\n)도 추가로 파일에 쓰게 했다.

4. 이제 다음과 같은 명령을 수행해 보자.

```
1 C:\doit>python memo.py -a "Life is too short"
2 C:\doit>python memo.py -a "You need python"
```

그리고 파일에 정상적으로 메모가 기입되었는지 다음과 같이 확인해 보자.

```
1 C:\doit>type memo.txt
2 Life is too short
3 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 사용한다.

이제 메모 출력을 위해 다음과 같이 코드를 변경해 보자.

```
1 # c:/doit/memo.py
2 import sys
3
4 option = sys.argv[1]
5
6 if option == '-a':
7     memo = sys.argv[2]
8     f = open('memo.txt', 'a')
9     f.write(memo)
```

```
10     f.write('\n')
11     f.close()
12 elif option == '-v':
13     f = open('memo.txt')
14     memo = f.read()
15     f.close()
16     print(memo)
```

옵션으로 -v 가 들어온 경우 memo.txt 파일을 읽어서 출력한다.

6. 코드를 수정한 후 다음과 같은 명령을 수행해 보자.

```
1 C:\doit>python memo.py -v
2 Life is too short
3 You need python
```

입력한 메모가 그대로 출력되는 것을 확인할 수 있다.

06-5 탭을 4 개의 공백으로 바꾸기

이번에는 문서 파일을 읽어서 그 문서 파일 안에 있는 탭 (tab) 을 공백 (space) 4 개로 바꾸어 주는 스크립트를 작성해 보자.

- 필요한 기능은? 문서 파일 읽어 들이기, 문자열 변경하기
- 입력 받는 값은? 탭을 포함한 문서 파일
- 출력하는 값은? 탭이 공백으로 수정된 문서 파일

다음과 같은 형식으로 프로그램이 수행되도록 만들 것이다.

```
1 python tabto4.py src dst
```

tabto4.py 는 우리가 작성해야 할 파이썬 프로그램 이름이고 src 는 탭을 포함하고 있는 원본 파일 이름이다. dst 는 파일 안의 탭을 공백 4 개로 변환한 결과를 저장할 파일 이름이다.

예를 들어 a.txt 파일에 있는 탭을 4 개의 공백으로 바꾸어 b.txt 파일에 저장하고 싶다면 다음과 같이 수행해야 한다.

```
1 python tabto4.py a.txt b.txt
```

1. 우선 다음과 같이 tabto4.py 파일을 작성해 보자.

tabto4.py 는 C:\doit 디렉터리에 저장한다.

```
1 # c:/doit/tabto4.py
2 import sys
3
4 src = sys.argv[1]
5 dst = sys.argv[2]
6
7 print(src)
8 print(dst)
```

sys.argv 를 사용하여 입력값을 확인하도록 만든 코드이다.

2. 다음과 같이 수행했을 때 입력값이 정상적으로 출력되는지 확인해 보자.

```
1 C:\doit>python tabto4.py a.txt b.txt
```

```
2 a.txt
3 b.txt
```

입력으로 전달한 **a.txt** 와 **b.txt** 가 정상적으로 출력되는 것을 확인할 수 있다.

3. 테스트를 위한 원본 파일 (탭을 포함하는 파일) 인 **a.txt** 를 다음과 같이 작성한다. 각 단어는 탭 (**\t**) 문자로 분리되도록 입력해야 한다.

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

4. 이제 탭 문자를 포함한 **a.txt** 파일을 읽어서 탭을 공백 4 개로 변환할 수 있도록 코드를 변경해 보자.

```
1 # c:/doit/tabto4.py
2 import sys
3
4 src = sys.argv[1]
5 dst = sys.argv[2]
6
7 f = open(src)
8 tab_content = f.read()
9 f.close()
10
11 space_content = tab_content.replace("\t", " "*4)
12 print(space_content)
```

위 코드는 **src** 에 해당되는 입력 파일을 읽어서 그 내용을 **tab_content** 라는 변수에 저장한 후 문자열의 **replace** 함수를 사용하여 탭 (**\t**) 을 4 개의 공백으로 변경하는 코드이다.

5. **tabto4.py** 를 위와 같이 변경한 후 다음과 같은 명령을 수행해 보자.

```
1 C:\doit>python tabto4.py a.txt b.txt
2 Life is too short
3 You need python
```

아마도 탭 문자가 공백 4 개로 변경되어 출력될 것이다. 하지만 탭과 공백의 차이점을 눈으로 알 수는 없으므로 탭이 정상적으로 공백으로 변경되었는지 확인하기 어렵다.

6. 이제 변경된 내용을 **b.txt** 파일에 저장할 수 있도록 다음과 같이 프로그램을 변경해 보자.

```
1 # c:/doit/tabto4.py
2 import sys
3
4 src = sys.argv[1]
5 dst = sys.argv[2]
6
7 f = open(src)
8 tab_content = f.read()
9 f.close()
10
11 space_content = tab_content.replace("\t", " "*4)
12
```

```
13 f = open(dst, 'w')
14 f.write(space_content)
15 f.close()
```

탭이 공백으로 변경된 `space_content` 를 출력 파일인 `dst` 에 쓰도록 코드를 수정하였다.

7. 프로그램을 실행하기 위해 다음 명령을 수행한다.

```
1 C:\doit>python tabto4.py a.txt b.txt
```

위 명령을 수행하면 `b.txt` 파일이 `C:\doit` 디렉터리에 생성된다. 에디터로 `b.txt` 파일을 열어서 탭이 4개의 공백 문자로 변경되었는지 확인해 보자. 프로그램을 작성할 때 사용하는 에디터는 대부분 탭과 공백 문자를 다르게 표시하므로 눈으로 확인이 가능할 것이다.

06-6 하위 디렉터리 검색하기

특정 디렉터리부터 시작해서 그 하위 모든 파일 중 파이썬 파일 (*.py) 만 출력해 주는 프로그램을 만들려면 어떻게 해야 할까?

1. 다음과 같이 sub_dir_search.py 파일을 작성해 보자.

sub_dir_search.py 파일은 C:\doit 디렉터리에 저장한다.

```
1 # C:/doit/sub_dir_search.py
2
3 def search(dirname):
4     print(dirname)
5
6 search("c:/")
```

search 함수를 만들고 시작 디렉터리를 입력받도록 코드를 작성했다.

2. 이제 이 디렉터리에 있는 파일을 검색할 수 있도록 소스를 변경해 보자.

```
1 # C:/doit/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:/")
```

os.listdir 를 사용하면 해당 디렉터리에 있는 파일들의 리스트를 구할 수 있다. 여기에서 구하는 파일 리스트는 파일 이름만 포함되어 있으므로 경로를 포함한 파일 이름을 구하기 위해서는 입력으로 받은 dirname 을 앞에 덧붙여 주어야 한다. os 모듈에는 디렉터리와 파일 이름을 이어 주는 os.path.join 함수가 있으므로 이 함수를 사용하면 디렉터리를 포함한 전체 경로를 쉽게 구할 수 있다.

위 코드를 수행하면 C:/ 디렉터리에 있는 파일이 다음과 비슷하게 출력될 것이다.

[디렉토리 출력 예]

```
1 c:/Recycle.Bin
```

```

2 c:/$WINDOWS.~BT
3 c:/$Windows.~WS
4 c:/adb
5 c:/AMD
6 c:/android
7 c:/bootmgr
8 c:/BOOTNXT
9 ... 생략 ...

```

3. 이제 C:/ 디렉터리에 있는 파일들 중 확장자가.py 인 파일만을 출력하도록 코드를 변경해 보자.

```

1 # C:/doit/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         ext = os.path.splitext(full_filename)[-1]
9         if ext == '.py':
10             print(full_filename)
11
12 search("c:/")

```

파일 이름에서 확장자만 추출하기 위해 os 모듈의 os.path.splitext 함수를 사용하였다. os.path.splitext는 파일 이름을 확장자를 기준으로 두 부분으로 나누어 준다. 따라서 os.path.splitext(full_filename)[-1]은 해당 파일의 확장자 이름이 된다. 위 코드는 확장자 이름이.py 인 경우만을 출력하도록 작성했다. C:/디렉터리에 파이썬 파일이 없다면 아무것도 출력되지 않을 것이다.

4. 하지만 우리가 원하는 것은 C:/디렉터리 바로 밑에 있는 파일뿐만 아니라 그 하위 디렉터리(sub directory)를 포함한 모든 파이썬 파일을 검색하는 것이다. 하위 디렉터리도 검색이 가능하도록 다음과 같이 코드를 변경해야 한다.

```

1 # C:/doit/sub_dir_search.py
2 import os
3
4 def search(dirname):
5     try:
6         filenames = os.listdir(dirname)
7         for filename in filenames:
8             full_filename = os.path.join(dirname, filename)
9             if os.path.isdir(full_filename):
10                 search(full_filename)
11             else:
12                 ext = os.path.splitext(full_filename)[-1]
13                 if ext == '.py':
14                     print(full_filename)
15     except PermissionError:
16         pass
17
18 search("c:/")

```

`try ... except PermissionError`로 함수 전체를 감싼 이유는 `os.listdir`를 수행할 때 권한이 없는 디렉터리에 접근하더라도 프로그램이 오류로 종료되지 않고 그냥 수행되도록 하기 위해서이다.

`full_filename`이 디렉터리인지 파일인지 구별하기 위하여 `os.path.isdir` 함수를 사용하였고 디렉터리일 경우 해당 경로를 입력받아 다시 `search` 함수를 호출하였다. 이렇게 해당 디렉터리의 파일이 디렉터리일 경우 다시 `search` 함수를 호출해 나가면 (재귀 호출) 해당 디렉터리의 하위 파일을 다시 검색하기 시작하므로 결국 모든 파일들을 검색할 수 있게 된다.

재귀 호출이란 자기 자신을 다시 호출하는 프로그래밍 기법이다. 이 코드에서 보면 `search` 함수에서 다시 자기 자신인 `search` 함수를 호출하는 것이 바로 재귀 호출이다.

위 코드를 수행하면 `C:/` 디렉터리에 있는 모든 파이썬 파일이 출력될 것이다.

하위 디렉터리 검색을 쉽게 해주는 `os.walk`

`os.walk`를 사용하면 위에서 작성한 코드를 보다 간편하게 만들 수 있다. `os.walk`는 시작 디렉터리 부터 시작하여 그 하위 모든 디렉터리를 차례대로 방문하게 해주는 함수이다.

```
1 import os
2
3 for (path, dir, files) in os.walk("c:/"):
4     for filename in files:
5         ext = os.path.splitext(filename)[-1]
6         if ext == '.py':
7             print("%s/%s" % (path, filename))
```

디렉터리와 파일을 검색하는 일반적인 경우라면 `os.walk`를 사용하는 것을 추천한다.

06-7 파이보

파이보는 파이썬 “질문과 답변” 게시판 서비스이다.

- 파이보 (<https://pybo.kr>)

파이썬을 공부하며 이해되지 않는 부분이나 궁금한 점들을 파이보에 질문해 보자. 본인이 정확히 어떤 것을 모르는지 질문을 하는 것만으로도 큰 공부가 될 것이다.

06-8 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.

07 장 파이썬 날아오르기

여러분은 아마도 이 책을 다 읽은 후에 본인에게 필요한 파이썬 프로그램을 만들어 볼 것이다. 그리고 간단한 프로그램이 아니라면 필수적으로 파이썬 표준 라이브러리를 사용해야 하는 순간이 온다. 하지만 파이썬 표준 라이브러리 중에는 이 책에서 설명하지 않은 다음과 같은 고급 주제의 내용을 미리 알고 있어야만 이해할 수 있는 것들이 있다.

- 파이썬과 유니코드
- 클로저와 데코레이터
- 이터레이터와 제너레이터
- 파이썬 타입 어노테이션

어번 장에서는 이러한 고급 주제들에 대해서 알아볼 것이다. 다만, 이러한 고급 주제의 내용은 파이썬 입문자에게 다소 어려운 내용이고 파이썬 코드를 작성하는데 필수적인 부분은 아니므로 건너뛰어도 무방하다. 하지만 여러분이 보다 심도 있는 파이썬 프로그램을 작성하고자 한다면 한번쯤 관심을 갖고 익혀두기를 권한다.

07-1 파이썬과 유니코드

컴퓨터는 0 과 1 이라는 값만 인식할 수 있는 기계 장치이다. 이런 컴퓨터가 문자를 인식하려면 어떻게 해야 할까? 과거부터 지금까지 사용하는 유일한 방법은 다음과 비슷한 방법으로 문자 셋을 만드는 것이다.

예를 들어 숫자 65 는 'A', 숫자 66 은 'B', ... 이런 식으로 숫자마다 문자를 매핑해 놓으면 컴퓨터는 해당 숫자를 문자로 대체하여 인식할 수 있을 것이다. 최초 컴퓨터가 발명되었을 때 이런 문자를 처리하고자 컴퓨터마다 각각의 문자셋을 정해 놓고 문자를 처리하기 시작했다. 하지만, 이렇게 컴퓨터마다 각각의 문자셋을 사용했더니 데이터 호환이 안 되는 문제가 발생했다. A 라는 컴퓨터에서 처리하는 문자셋 규칙이 B 라는 컴퓨터에서 처리하는 문자셋 규칙과 같지 않기 때문에 서로 데이터를 주고받는 등의 일을 할 수가 없었던 것이다.

이런 문제를 해결하고자 미국에서 최초 문자셋 표준인 아스키 (ASCII) 가 탄생하게 된다. 아스키라는 문자셋 규칙을 정하고 이 규칙대로만 문자를 만들면 다른 기종 컴퓨터 간에도 문제없이 데이터를 주고받을 수 있었다. 아스키는 처리할 수 있는 문자 개수가 127 개였는데, 영어권 국가에서 사용하는 영문자, 숫자 등을 처리하는 데는 부족함이 없었다. 하지만, 곧 비영어권 국가에서도 자신의 문자를 컴퓨터로 표현하고자 하는 요구가 생겼다. 아스키는 127 개의 문자만을 다룰 수 있으므로 아스키를 사용할 수는 없는 노릇이었다. 그래서 곧 서유럽 문자셋인 ISO8859 가 등장하게 되고 한국에서는 KSC5601 과 같은 문자셋이 등장하게 된다.

이렇게 나라마다 문자셋이 만들어지고 또 한 나라에서도 여러 개의 문자셋이 표준이 되고자 치열한 싸움을 벌이기도 하며 문자를 처리하는 방법은 점점 더 복잡해져만 갔다. 가장 결정적인 문제는 하나의 문서에 여러 나라의 언어를 동시에 표현할 방법이 없다는 점이었다.

이런 문제를 해결하고자 등장한 것이 바로 유니코드 (Unicode) 이다. 유니코드는 모든 나라의 문자를 다 포함하게끔 넉넉하게 설계되었고 곧 세계 표준으로 자리 잡게 되었다. 이 유니코드라는 규칙을 사용하면 서로 다른 문자셋으로 고생할 일이 없어진다.

인코딩

다음과 같은 문자열을 보자.

```
1 >>> a = "Life is too short"
```

파이썬에서 사용하는 문자열은 모두 유니코드 문자열이다. (파이썬 3 버전부터 모든 문자열을 유니코드로 처리하도록 변경되었다.)

유니코드 문자열은 **인코딩 (encoding)** 없이 그대로 파일에 적거나 다른 시스템으로 전송할 수 없다. 왜냐하면 유니코드 문자열은 단순히 문자셋의 **규칙**이기 때문이다. 파일에 적거나 다른 시스템으로 전송하려면 바이트로 변환해야 한다. 이렇게 유니코드 문자열을 바이트로 바꾸는 것을 인코딩이라 한다. 따라서 파일을 읽거나 네트워크를 통해 바이트 문자열을 수신할 때에는 해당 바이트가 어떤 방식의 인코딩을 사용했는지를 미리 알아야만 디코딩할 수 있다.

유니코드 문자열을 바이트 문자열로 바꾸는 방법은 다음과 같다.

```
1 >>> a = "Life is too short"
2 >>> b = a.encode('utf-8')
3 >>> b
4 b'Life is too short'
5 >>> type(b)
6 <class 'bytes'>
```

유니코드 문자열을 바이트 문자열로 만들 때에는 이 예처럼 **utf-8**과 같은 인코딩 방식을 인수로 넘겨 주어야 한다. 인수를 생략하면 기본값인 **utf-8**로 동작한다. 문자열을 변환하고 나서 **type** 명령어를 호출해 보면 **b** 객체는 **bytes** 클래스의 객체임을 알 수 있다.

이번에는 다음 예제를 보자.

```
1 >>> a = "한글"
2 >>> a.encode("ascii")
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
     ordinal not in range(128)
```

이 예에서는 한글이라는 유니코드 문자열을 아스키 (**ascii**) 방식으로 인코딩하려고 시도한다. 하지만, 아스키 방식으로는 한글을 표현할 수 없으므로 오류가 발생한다.

“한글”이라는 유니코드 문자열을 바이트로 변경하는 인코딩 방식에는 여러 개가 있다. 보통은 **utf-8**을 사용하지만, 기존 시스템이 **euc-kr**과 같은 옛날 방식의 인코딩을 사용한다면 다음과 같이 **euc-kr**로 인코딩할 수도 있다.

```
1 >>> a = '한글'
2 >>> a.encode('euc-kr')
3 b'\xc7\xd1\xb1\xdb'
4 >>> a.encode('utf-8')
5 b'\xed\x95\x9c\xea\xb8\x80'
```

utf-8로 인코딩 했을 때와는 다른 바이트 문자열을 출력하는 것을 확인할 수 있다.

`euc-kr`은 영문과 한글 처리만 가능한 인코딩 방식이다. 따라서 일본어나 중국어 등을 처리할 수 없다. 이와는 달리 `utf-8`은 유니코드 문자열 전체를 처리할 수 있는 인코딩 방식이다.

디코딩

이번에는 반대로 인코딩한 바이트 문자열을 유니코드 문자열로 변환하는 디코딩 (decoding)을 알아보자. 다음 예제처럼 `euc-kr`로 인코딩한 바이트 문자열은 `euc-kr`로만 디코딩해야 한다.

```
1 >>> a = '한글'
2 >>> b = a.encode('euc-kr')
3 >>> b.decode('euc-kr')
4 '한글'
```

이와는 달리 `euc-kr`로 인코딩한 바이트 문자열을 `utf-8`로 디코딩하려 한다면 어떻게 될까?

```
1 >>> b.decode('utf-8')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc7 in position 0: invalid continuation byte
```

잘못된 인코딩 방식으로 디코딩하려 하면 이처럼 오류가 발생한다.

입출력과 인코딩

인코딩과 관련해서 개발자가 가장 고생하는 부분은 바로 데이터 입출력 관련이다. 이것 역시 문자열과 인코딩에 대한 개념만 확실히 이해하면 어렵지 않지만, 이를 이해하지 못하고 무작정 인코딩, 디코딩을 사용하면 다중 인코딩되거나 이로 말미암아 문자열이 꼬여 버리는 불상사가 발생하기도 한다.

파일을 읽거나 네트워크를 통해 데이터를 주고받을 때 추천하는 방법은 다음과 같다.

1. 입력으로 받은 바이트 문자열은 가능한 한 가장 빨리 유니코드 문자열로 디코딩할 것
2. 유니코드 문자열만 함수나 클래스 등에서 사용할 것
3. 입력에 대한 결과를 전송하는 마지막 부분에서만 유니코드 문자열을 바이트 문자열로 인코딩해서 반환할 것

이와 같은 규칙을 지킨다면 인코딩과 관련해서 큰 어려움은 없을 것이다.

다음은 `euc-kr` 방식으로 작성한 파일을 읽고 변경하여 저장하는 예제이다.

```
1 # 1. euc-kr로 작성된 파일 읽기
2 with open('euc_kr.txt', encoding='euc-kr') as f:
```

```
3     data = f.read() # 유니코드 문자열
4
5 # 2. unicode 문자열로 프로그램 수행하기
6 data = data + "\n" + "추가 문자열"
7
8 # 3. euc-kr로 수정된 문자열 저장하기
9 with open('euc_kr.txt', encoding='euc-kr', mode='w') as f:
10     f.write(data)
```

파일을 읽는 `open()` 함수에는 `encoding` 을 지정하여 파일을 읽는 기능이 있다. 이때 읽은 문자열은 유니코드 문자열이 된다. 마찬가지로 파일을 만들 때도 `encoding` 을 지정할 수 있다. `encoding` 항목을 생략하면 기본값으로 `utf-8` 이 지정된다.

소스코드의 인코딩

파이썬 셸이 아닌 편집기로 코딩할 때는 소스 코드의 인코딩이 매우 중요하다. 소스 코드의 인코딩이란 소스 코드 파일이 현재 어떤 방식으로 인코딩되었지를 뜻한다.

앞의 예제에서 알아보았듯이 파일은 `utf-8` 인코딩으로 저장할 수도 있고 `euc-kr` 로 저장할 수도 있다. 소스 코드도 파일이므로 인코딩 타입이 반드시 필요하다. 파이썬은 소스 코드의 인코딩을 명시하고자 소스 코드 가장 위에 다음과 같은 문장을 넣어야 한다.

```
1 # -*- coding: utf-8 -*-
```

파이썬 3.0 부터는 `utf-8` 이 기본값이므로 `utf-8` 로 인코딩한 소스 코드라면 이 문장은 생략해도 된다.

소스 코드를 `utf-8` 로 인코딩한 파일이라면 이처럼 작성하면 되고 `euc-kr` 로 인코딩했다면 다음과 같이 작성해야 한다.

```
1 # -*- coding: euc-kr -*-
```

소스 코드는 `euc-kr` 로 인코딩했는데 파일 위에 `utf-8` 로 명시했다면 문자열 처리 부분에서 인코딩 관련 오류가 발생할 수 있다.

07-2 클로저와 데코레이터

데코레이터를 이해하려면 먼저 클로저를 알아야 한다. 클로저를 먼저 알아보고 나서 데코레이터를 살펴보자.

클로저

클로저는 간단히 말해 함수 안에 내부 함수 (inner function) 를 구현하고 그 내부 함수를 리턴하는 함수를 말한다. 이때 외부 함수는 자신이 가진 변수값 등을 내부 함수에 전달할 수 있다. 알쏭달쏭한 설명이지만 예제를 보면 쉽게 이해할 수 있다.

어떤 수에 항상 3 을 곱해 리턴하는 함수를 생각해 보자. 아마도 다음과 같이 함수를 만들 수 있을 것이다.

```
1 def mul3(n):  
2     return n * 3
```

mul3() 함수는 입력으로 받은 수 n 에 항상 3 을 곱하여 리턴한다. 이번에는 항상 5 를 곱하여 리턴하는 함수를 생각해 보자.

```
1 def mul5(n):  
2     return n * 5
```

이처럼 mul5() 함수를 만들 수 있을 것이다. 하지만, 이렇게 필요할 때마다 mul6(), mul7(), mul8(), ...과 같은 함수를 만드는 것은 굉장히 비효율적이다. 이 문제를 효율적으로 해결하려면 다음과 같이 클래스를 사용할 수 있다.

```
1 class Mul:  
2     def __init__(self, m):  
3         self.m = m  
4  
5     def mul(self, n):  
6         return self.m * n  
7  
8 if __name__ == "__main__":  
9     mul3 = Mul(3)  
10    mul5 = Mul(5)
```

```
11
12     print(mul3.mul(10)) # 30 출력
13     print(mul5.mul(10)) # 50 출력
```

클래스를 이용하면 이 코드처럼 특정 값을 미리 설정하고 그다음부터 `mul()` 메서드를 사용하면 원하는 형태로 호출할 수 있다. 그리고 다시 다음과 같이 `__call__` 메서드를 이용하여 이를 개선할 수 있다.

```
1 class Mul:
2     def __init__(self, m):
3         self.m = m
4
5     def __call__(self, n):
6         return self.m * n
7
8
9 if __name__ == "__main__":
10     mul3 = Mul(3)
11     mul5 = Mul(5)
12
13     print(mul3(10)) # 30 출력
14     print(mul5(10)) # 50 출력
```

`mul()` 함수 이름을 `__call__`로 바꾸었다. `__call__` 함수는 `Mul` 클래스로 만든 객체에 인수를 전달하여 바로 호출할 수 있도록 하는 메서드이다. `__call__` 메서드를 이용하면 이 예제처럼 `mul3` 객체를 `mul3(10)` 처럼 호출할 수 있다. 이렇게 클래스로 만드는 방법이 일반적이긴 하지만, 더 간편한 방법이 있다. 다음 함수를 보자.

```
1 def mul(m):
2     def wrapper(n):
3         return m * n
4     return wrapper
5
6
7 if __name__ == "__main__":
8     mul3 = mul(3)
9     mul5 = mul(5)
10
11     print(mul3(10)) # 30 출력
12     print(mul5(10)) # 50 출력
```

외부 함수 (`mul()`) 안에 내부 함수 (`wrapper()`) 를 구현했다. 그리고 외부 함수는 내부 함수 `wrapper()` 를 리턴한다. 함수가 함수를 리턴하는 것이 생소할 수 있겠지만 파이썬에서는 이것이 가능하다.

재밌는 사실은 `mul()` 함수에서 `wrapper()` 함수를 리턴할 때 `mul()` 함수 호출 시 인수로 받은 `m` 값을 `wrapper()` 함수에 저장하여 리턴한다는 점이다. 이것은 마치 클래스가 특정한 값을 설정하여 객체를 만드는 과정과 매우 비슷하다. 이러한 `mul()` 과 같은 함수를 파이썬에서는 **클로저 (Closure)** 라 한다.

데코레이터

다음은 “함수가 실행됩니다” 라는 문자열을 출력하는 `myfunc` 함수이다.

```
1 def myfunc():
2     print("함수가 실행됩니다.")
```

그런데 이 함수의 실행 시간을 측정해야 한다면 어떻게 해야 할까? 함수 실행 시간은 함수가 시작하는 순간의 시간과 함수가 종료되는 순간의 시간 차이를 구하면 알 수 있다. 따라서 다음과 같이 코드를 수정하면 함수의 실행 시간을 측정할 수 있다.

```
1 import time
2
3 def myfunc():
4     start = time.time()
5     print("함수가 실행됩니다.")
6     end = time.time()
7     print("함수 수행 시간: %f 초" % (end-start))
8
9 myfunc()
```

하지만, 실행 시간을 측정해야 하는 함수가 `myfunc()` 말고도 많다면 이런 코드를 모든 함수에 마찬가지로 적용하는 것은 너무 비효율적이다. 클로저를 이용하면 좀 더 효율적인 방법을 찾을 수 있다.

```
1 import time
2
3
4 def elapsed(original_func): # 기존 함수를 인수로 받는다.
5     def wrapper():
6         start = time.time()
7         result = original_func() # 기존 함수를 수행한다.
8         end = time.time()
9         print("함수 수행 시간: %f 초" % (end - start)) # 기존 함수의 수행 시간을
            출력한다.
10        return result # 기존 함수의 수행 결과를 리턴한다.
11    return wrapper
12
13
14 def myfunc():
15     print("함수가 실행됩니다.")
16
17
18 decorated_myfunc = elapsed(myfunc)
19 decorated_myfunc()
```

`elapsed()` 함수로 클로저를 만들었다. 이 함수는 함수를 인수로 받는다. 파이썬은 함수도 객체이므로 함수 자체를 인수로 전달할 수 있다.

이제 `decorated_myfunc = elapsed(myfunc)`로 생성한 `decorated_myfunc`를 `decorated_myfunc()`처럼 실행하면 실제로는 `elapsed()` 내부의 `wrapper()` 함수가 실행되고 이 함수는 전달받

은 `myfunc()` 함수를 실행하고 실행 시간도 함께 출력한다.

클로저를 이용하면 기존 함수에 뭔가 추가 기능을 덧붙이기가 아주 편리하다. 이렇게 기존 함수를 바꾸지 않고 추가 기능을 덧붙일 수 있도록 하는 `elapsed()` 함수와 같은 클로저를 **데코레이터 (Decorator)** 라 한다.

'decorate' 는 '꾸미다, 장식하다' 라는 뜻이므로 데코레이터를 함수를 꾸미는 함수라 생각해도 좋다.

이 코드를 실행하면 다음과 같은 결과가 출력된다.

```
1 함수가 실행됩니다.
2 함수 수행 시간: 0.000029 초
```

파이썬 데코레이터는 다음처럼 `@` 문자를 이용해 함수 위에 적용하여 사용할 수도 있다.

```
1 import time
2
3
4 def elapsed(original_func):    # 기존 함수를 인수로 받는다.
5     def wrapper():
6         start = time.time()
7         result = original_func()    # 기존 함수를 수행한다.
8         end = time.time()
9         print("함수 수행 시간: %f 초" % (end - start))    # 기존 함수의 수행 시간을
            출력한다.
10        return result    # 기존 함수의 수행 결과를 리턴한다.
11    return wrapper
12
13
14 @elapsed
15 def myfunc():
16     print("함수가 실행됩니다.")
17
18 # decorated_myfunc = elapsed(myfunc)    # @elapsed 데코레이터로 인해 더 이상 필요
    하지 않다.
19 # decorated_myfunc()
20
21 myfunc()
```

`myfunc()` 함수 바로 위에 `@elapsed(@+ 함수명)` 라는 데코레이터를 추가했다. 파이썬은 함수 위에 `@ + 함수명`이 있으면 데코레이터 함수로 인식한다. 따라서 이제 `myfunc()` 함수는 `elapsed` 데코레이터를 통해 수행될 것이다.

프로그램을 실행해 보면 마찬가지로 결과임을 알 수 있다.

```
1 함수가 실행됩니다.
2 함수 수행 시간: 0.000029 초
```

이번에는 `myfunc()` 함수를 다음과 같이 변경해 보자.

```

1 ... 생략 ...
2
3 @elapsed
4 def myfunc(msg):
5     print('%s'을 출력합니다." % msg)
6
7
8 myfunc("You need python") # 출력할 메시지를 myfunc 파라미터로 전달한다.

```

문자열을 입력받아 출력하도록 myfunc() 함수를 수정했다. 하지만, 이렇게 코드를 수정하고 실행하면 다음과 같은 오류가 발생한다.

```

1 Traceback (most recent call last):
2   File ... 생략 ...
3     myfunc("You need python")
4   TypeError: wrapper() takes 0 positional arguments but 1 was given

```

오류의 원인은 myfunc() 함수는 입력 인수가 필요하나 elapsed() 함수 내의 wrapper() 함수는 전달받은 myfunc() 함수를 입력 인수 없이 호출하기 때문이다.

데코레이터 함수는 기존 함수의 입력 인수에 상관없이 동작하도록 해야 한다. 왜냐하면 데코레이터는 기존 함수가 어떤 입력 인수를 취할지 알 수 없기 때문이다. 따라서 이렇게 전달받아야 하는 기존 함수의 입력 인수를 알 수 없는 경우에는 *args와 **kwargs 기법을 이용하여 해결해야 한다.

*args, kwargs

*args는 모든 입력 인수를 튜플로 변환하는 매개변수이고 **kwargs는 모든 키-값 형태의 입력 인수를 딕셔너리로 변환하는 매개변수이다. 다음과 같은 형태의 호출을 살펴보자.

```
1 >>> func(1, 2, 3, name='foo', age=3)
```

func() 함수가 입력 인수의 개수와 형태에 상관없이 모든 입력을 처리하려면 어떻게 해야 할까?

```

1 >>> def func(*args, **kwargs):
2 ...     print(args)
3 ...     print(kwargs)
4 ...
5 >>> func(1, 2, 3, name='foo', age=3)
6 (1, 2, 3)
7 {'age': 3, 'name': 'foo'}

```

이처럼 func() 함수에 *args, **kwargs라는 매개변수를 지정하면 다양한 입력 인수를 모두 처리할 수 있다. 이렇게 하면 1, 2, 3 같은 일반 입력은 args 튜플로 저장하고 name='foo'와 같은 키-값 형태의 입력은 kwargs 딕셔너리로 저장한다.

다음과 같이 코드를 수정하자.

```

1 import time
2

```

```
3
4 def elapsed(original_func): # 기존 함수를 인수로 받는다.
5     def wrapper(*args, **kwargs): # *args, **kwargs 매개변수 추가
6         start = time.time()
7         result = original_func(*args, **kwargs) # 전달받은 *args, **kwargs를
            입력파라미터로 기존 함수 수행
8         end = time.time()
9         print("함수 수행시간: %f 초" % (end - start)) # 수행시간을 출력한다.
10        return result # 함수의 결과를 리턴한다.
11    return wrapper
12
13
14 @elapsed
15 def myfunc(msg):
16     """ 데코레이터 확인 함수 """
17     print("'s'을 출력합니다." % msg)
18
19
20 myfunc("You need python")
```

wrapper() 함수의 매개변수로 *args와 **kwargs를 추가하고 기존 함수 실행시 *args와 **kwargs를 인수로 전달하여 호출하게 했다. 이제 프로그램을 실행하면 오류 없이 다음과 같은 결과를 출력한다.

```
1 'You need python'을 출력합니다.
2 함수 수행시간: 0.000027 초
```

지금까지 클로저와 데코레이터를 알아보았다.

07-3 이터레이터와 제너레이터

다음은 늘 사용하던 리스트의 간단한 사용법이다.

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

리스트 [1, 2, 3] 을 for 문으로 차례대로 하나씩 출력하는 예제이다. 이렇게 for 문과 같은 반복 구문을 적용할 수 있는 리스트와 같은 객체를 반복 가능 (iterable) 객체라 한다.

반복 가능한 객체에는 리스트, 튜플, 딕셔너리, 집합등이 있다.

이터레이터란?

그렇다면 **이터레이터** (iterator) 란 무엇일까? 이터레이터는 next() 함수 호출 시 계속 그다음 값을 리턴하는 객체이다. 리스트는 반복 가능 (iterable) 하다는 것을 이미 알아보았다. 그렇다면 리스트는 이터레이터일까? 다음과 같이 확인해 보자.

```
1 >>> a = [1, 2, 3]
2 >>> next(a)
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: 'list' object is not an iterator
```

a 라는 리스트로 next() 함수를 호출했더니 리스트는 이터레이터 객체가 아니라는 오류가 발생한다. 즉, 반복 가능하다고 해서 이터레이터는 아니라는 말이다. 하지만, 반복 가능하다면 다음과 같이 iter() 함수를 이용하여 이터레이터로 만들 수 있다.

```
1 >>> a = [1, 2, 3]
2 >>> ia = iter(a)
3 >>> type(ia)
4 <class 'list_iterator'>
```

이제 리스트를 이터레이터로 변경했으므로 next() 함수를 호출해 보자.

```
1 >>> next(ia)
2 1
```

```
3 >>> next(ia)
4 2
5 >>> next(ia)
6 3
7 >>> next(ia)
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 StopIteration
```

`next()` 함수를 호출할 때마다 이터레이터 객체의 요소를 차례대로 리턴하는 것을 확인할 수 있다. 하지만, 더는 리턴할 값이 없다면 `StopIteration` 예외가 발생한다. 이터레이터의 값을 가져오는 가장 일반적인 방법은 다음과 같이 `for` 문을 이용하는 것이다.

```
1 >>> a = [1, 2, 3]
2 >>> ia = iter(a)
3 >>> for i in ia:
4 ...     print(i)
5 ...
6 1
7 2
8 3
```

`for` 문을 이용하면 `next()` 함수를 따로 호출할 필요도 없고 (`for` 문이 자동으로 호출) `StopIteration` 예외에 신경 쓸 필요도 없다. 이번에는 다음과 같은 예를 보자.

```
1 >>> a = [1, 2, 3]
2 >>> ia = iter(a)
3 >>> for i in ia:
4 ...     print(i)
5 ...
6 1
7 2
8 3
9 >>> for i in ia:
10 ...     print(i)
11 ...
12 >>>
```

이처럼 이터레이터는 `for` 문을 이용하여 반복하고 난 후에는 다시 반복하더라도 더는 그 값을 가져오지 못한다. 즉, `next()` 로 그 값을 한 번 읽으면 그 값을 다시는 읽을 수 없다는 특징이 있다.

이터레이터 만들기

`iter()` 함수를 이용하면 리스트를 이터레이터로 만들 수 있었다. 이번에는 `iter()` 함수 대신 클래스로 이터레이터를 만들어보자. 이터레이터는 클래스에 `__iter__`와 `__next__`라는 두 개의 메서드를 구현하여 만들 수 있다. 다음 예를 살펴보자.

```

1 class MyItertor:
2     def __init__(self, data):
3         self.data = data
4         self.position = 0
5
6     def __iter__(self):
7         return self
8
9     def __next__(self):
10        if self.position >= len(self.data):
11            raise StopIteration
12        result = self.data[self.position]
13        self.position += 1
14        return result
15
16
17 if __name__ == "__main__":
18     i = MyItertor([1,2,3])
19     for item in i:
20         print(item)

```

MyIterator 클래스는 이터레이터 객체를 생성하기 위해 `__iter__` 메서드와 `__next__` 메서드를 구현했다. `__iter__` 메서드와 `__next__` 메서드는 생성자 `__init__` 메서드와 마찬가지로 클래스에 특별한 의미를 갖는 메서드이다. 클래스에 `__iter__` 메서드를 구현하면 해당 클래스로 생성한 객체는 반복 가능한 객체가 된다. `__iter__` 메서드는 반복 가능한 객체를 리턴해야 하며 보통 클래스의 객체를 의미하는 `self`를 리턴한다. 그리고 클래스에 `__iter__` 메서드를 구현할 경우 반드시 `__next__` 함수를 구현해야 한다. `__next__` 메서드는 반복 가능한 객체의 값을 차례대로 반환하는 역할을 한다. `__next__` 메서드는 `for` 문을 수행하거나 `next()` 함수 호출 시 수행되므로 **MyIterator** 객체 생성 시 전달한 `data`를 하나씩 리턴하도록 하고 더는 리턴할 값이 없으면 `StopIteration` 예외를 발생시키도록 구현했다.

이 코드를 실행하면 다음과 같은 결과를 확인할 수 있다.

```

1 1
2 2
3 3

```

이번에는 입력받은 데이터를 역순으로 출력하는 **ReverseIterator** 클래스를 만들어 보자 (설명은 앞의 **MyIterator**와 마찬가지로 생략한다).

```

1 class ReverseItertor:
2     def __init__(self, data):
3         self.data = data
4         self.position = len(self.data) - 1
5
6     def __iter__(self):
7         return self
8
9     def __next__(self):

```

```

10         if self.position < 0:
11             raise StopIteration
12         result = self.data[self.position]
13         self.position -= 1
14         return result
15
16
17 if __name__ == "__main__":
18     i = ReverseItertor([1,2,3])
19     for item in i:
20         print(item)

```

이를 실행하면 다음과 같이 입력받은 데이터를 역순으로 출력한다.

```

1 3
2 2
3 1

```

제너레이터란?

제너레이터 (generator) 는 이터레이터를 생성해 주는 함수이다. 제너레이터로 생성한 객체는 이터레이터와 마찬가지로 `next()` 함수 호출 시 그 값을 차례대로 얻을 수 있다. 이때 제너레이터에서는 차례대로 결과를 반환하고자 `return` 대신 `yield` 키워드를 사용한다.

가장 간단한 제너레이터의 예를 살펴보자.

```

1 >>> def mygen():
2 ...     yield 'a'
3 ...     yield 'b'
4 ...     yield 'c'
5 ...
6 >>> g = mygen()

```

`mygen()` 함수는 `yield` 구문을 포함하므로 제너레이터이다. 제너레이터 객체는 `g = mygen()` 과 같이 제너레이터를 호출하여 만들 수 있다. `type` 명령어로 확인하면 `g` 객체는 제너레이터 타입의 객체임을 알 수 있다.

```

1 >>> type(g)
2 <class 'generator'>

```

이제 다음과 같이 제너레이터로 반환된 `g` 의 값을 차례대로 얻어 보자.

```

1 >>> next(g)
2 'a'

```

이처럼 객체 `g` 로 `next()` 함수를 실행하면 `mygen()` 함수의 첫 번째 `yield` 문에 따라 'a' 값을 리턴한다. 여기

서 재밌는 점은 제너레이터는 `yield` 라는 문장을 만나면 그 값을 리턴하되 현재 상태를 그대로 기억한다는 점이다. 이건 마치 음악을 재생하다가 일시 정지 버튼으로 멈춘 것과 비슷한 모양새이다.

다시 `next()` 함수를 실행해 보자.

```
1 >>> next(g)
2 'b'
```

이번에는 두 번째 `yield` 문에 따라 'b' 값을 리턴한다. 계속해서 `next()` 함수를 호출하면 다음과 같은 결과가 출력될 것이다.

```
1 >>> next(g)
2 'c'
3 >>> next(g)
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 StopIteration
```

`mygen()` 함수에는 총 3 개의 `yield` 문이 있으므로 4 번째 `next()` 를 호출할 때는 더는 리턴할 값이 없으므로 `StopIteration` 예외가 발생한다.

제너레이터 표현식

이번에는 다음과 같은 예를 보자.

```
1 def mygen():
2     for i in range(1, 1000):
3         result = i * i
4         yield result
5
6 gen = mygen()
7
8 print(next(gen))
9 print(next(gen))
10 print(next(gen))
```

`mygen()` 함수는 1 부터 1,000 까지 각각의 숫자를 제공한 값을 순서대로 리턴하는 제너레이터이다. 이 예제를 실행하면 총 3 번의 `next()` 를 호출하므로 다음과 같은 결과가 나올 것이다.

```
1 1
2 4
3 9
```

제너레이터는 `def` 를 이용한 함수로 만들 수 있지만, 다음과 같이 튜플 표현식으로 좀 더 간단하게 만들 수도 있다.

```
1 gen = (i * i for i in range(1, 1000))
```

이 표현식은 `mygen()` 함수로 만든 제너레이터와 완전히 똑같이 기능한다. 여기서 사용한 표현식은 리스트 컴프리헨션 (list comprehension) 구문과 비슷하다. 다만 리스트 대신 튜플을 이용한 점이 다르다. 이와 같은 표현식을 **제너레이터 표현식** (generator expression) 이라 부른다.

제너레이터와 이터레이터

지금까지 살펴본 제너레이터는 이터레이터와 서로 상당히 비슷하다는 것을 알 수 있다. 클래스를 이용하여 이터레이터를 작성하면 좀 더 복잡한 행동을 하게 할 수 있다. 이와는 달리 제너레이터를 이용하면 간단하게 이터레이터를 만들 수 있다. 따라서 이터레이터의 성격에 따라 클래스로 만들 것인지 제너레이터로 만들 것인지를 선택해야 한다.

간단한 경우라면 제너레이터 함수나 제너레이터 표현식을 사용하는 것이 가독성이나 유지보수 측면에서 유리하다. 다음은 `(i * i for i in range(1, 1000))` 제너레이터를 클래스로 구현한 예이다.

```
1 class MyIterator:
2     def __init__(self):
3         self.data = 1
4
5     def __iter__(self):
6         return self
7
8     def __next__(self):
9         result = self.data * self.data
10        self.data += 1
11        if self.data >= 1000:
12            raise StopIteration
13        return result
```

이렇게 간단한 경우라면 클래스보다는 제너레이터 표현식을 사용하는 것이 훨씬 간편하고 이해하기 쉽다.

제너레이터 활용하기

제너레이터는 어떤 경우에 사용하면 좋을까? 다음의 예제를 통해 생각해 보자.

```
1 import time
2
3
4 def longtime_job():
5     print("job start")
6     time.sleep(1) # 1초 지연
7     return "done"
```

```
8
9
10 list_job = iter([longtime_job() for i in range(5)])
11 print(next(list_job))
```

`longtime_job()` 함수는 함수내에 `time.sleep(1)` 문장이 있어 실행하는데 1 초의 시간이 소요된다. 이 예제는 `longtime_job()` 함수를 5 번 실행하여 리스트에 그 결과값을 담고 이를 이터레이터로 변경한 후 그 첫 번째 결과값을 호출하는 예제이다. 실행하면 다음과 같은 결과를 출력한다.

```
1 job start
2 job start
3 job start
4 job start
5 job start
6 done
```

리스트를 만들 때 이미 5 개의 함수를 모두 실행하므로 5 초의 시간이 소요되고 이와 같은 결과를 출력한다.

이번에는 이 예제에 제너레이터를 적용해 보자. 프로그램을 다음과 같이 수정하자.

```
1 import time
2
3
4 def longtime_job():
5     print("job start")
6     time.sleep(1)
7     return "done"
8
9
10 list_job = (longtime_job() for i in range(5))
11 print(next(list_job))
```

`iter([longtime_job() for i in range(5)])` 코드를 제너레이터 표현식(`(longtime_job() for i in range(5))`)으로 바꾸었을 뿐이다. 하지만, 실행 시 1 초의 시간만 소요되고 출력되는 결과도 전혀 다르다.

```
1 job start
2 done
```

왜냐하면 제너레이터 표현식으로 인해 `longtime_job()` 함수가 5 회가 아닌 1 회만 호출되었기 때문이다. 이러한 방식을 '느긋한 계산법 (lazy evaluation)' 이라 부른다. 시간이 오래 걸리는 작업을 한꺼번에 처리하기 보다는 필요한 경우에만 호출하여 사용할 때 제너레이터는 매우 유용하다.

07-4 파이썬 타입 어노테이션

파이썬 3.5 버전부터 변수와 함수에 타입을 지정할 수 있는 타입 어노테이션 기능이 추가되었다.

파이썬은 동적 프로그래밍 언어

a 변수에 숫자 1 을 대입하고 type() 함수를 실행해 보자.

```
1 >>> a = 1
2 >>> type(a)
3 <class 'int'>
```

a 변수의 타입은 int 형임을 알 수 있다. 그리고 다시 a 변수에 문자열 "1" 을 대입하고 type() 함수를 실행해 보자.

```
1 >>> a = "1"
2 >>> type(a)
3 <class 'str'>
```

a 변수의 타입이 str 형으로 바뀌었다. 이렇게 프로그램 실행 중에 변수의 타입을 동적으로 바꿀 수 있으므로 파이썬을 **동적 프로그래밍 언어** (dynamic programming language) 라 한다.

자바는 정적 프로그래밍 언어

파이썬과 달리 자바는 정수형 (int) 변수 a 에 숫자 1 을 대입하고 다시 문자열 "1" 을 대입하려 할 때 컴파일 에러가 발생한다.

```
1 int a = 1; // a 변수를 int형으로 지정
2 a = "1"; // a 변수에 문자열을 대입할 수 없으므로 컴파일 에러
```

자바는 한 번 변수에 타입을 지정하면 지정한 타입 외에 다른 타입은 사용할 수 없으므로 **정적 프로그래밍 언어** (static programming language) 라 한다.

동적 언어의 장단점

파이썬과 같은 동적 언어는 타입에 자유로워 유연한 코딩이 가능하므로 쉽고 빠르게 프로그램을 만들 수 있다. 그리고 타입 체크를 위한 코드가 없으므로 비교적 깔끔한 소스 코드를 생성할 수 있다. 하지만, 프로젝트의 규모가 커질수록 잘못된 타입 사용으로 말미암은 버그가 생길 확률도 늘게 된다.

안전성을 선호하는 금융권 프로젝트에서는 이러한 이유로 동적 언어보다는 정적 언어를 메인 언어로 선택하는 경향이 많다.

파이썬 타입 어노테이션

파이썬은 3.5 버전부터 타입 어노테이션을 지원하기 시작한다. 다만 정적 언어에서와 같은 적극적인 타입 체크가 아니라 **타입 어노테이션** (type annotation), 즉 타입에 대한 힌트를 알려주는 정도이다. 동적 언어의 장점을 잃지 않고 기존에 작성된 코드와의 호환성을 생각하면 당연한 선택일 것이다.

타입 어노테이션은 다음과 같이 사용한다.

```
1 num: int = 1
```

변수명 바로 뒤에 : **int**와 같이 사용하여 num 변수가 int 형임을 명시한다.

```
1 def add(a: int, b: int) -> int:  
2     return a+b
```

함수의 매개변수에도 같은 규칙을 적용하여 매개변수의 타입을 명시할 수 있다. 그리고 함수의 리턴값도 -> **int**처럼 사용하여 리턴값의 타입을 명시할 수 있다.

어노테이션 타입으로 정수는 int, 문자열은 str, 리스트는 list, 튜플은 tuple, 딕셔너리는 dict, 집합은 set, 불은 bool 을 사용한다.

mypy

하지만, 파이썬 어노테이션을 사용하더라도 다음과 같이 사용할 수 있다.

[파일명: c:/doit/typing_sample.py]

```
1 def add(a: int, b: int) -> int:  
2     return a+b  
3  
4 result = add(3, 3.4)  
5 print(result) # 6.4 출력
```

`add()` 함수의 `b` 매개변수는 `int` 형이지만 3.4 와 같은 `float` 형 데이터를 사용해도 이 코드는 문제없이 돌아간다. 왜냐하면 파이썬 타입 어노테이션은 체크가 아닌 힌트이기 때문이다.

파이참과 같은 파이썬 전용 IDE 를 사용하면 타입이 맞지 않는다는 경고 메시지를 표시한다.

더 적극적으로 파이썬 어노테이션을 활용하려면 `mypy` 를 사용하는 것이 좋다. `mypy` 는 파이썬 표준 라이브러리가 아니므로 다음과 같이 설치한 후에 사용할 수 있다.

```
1 c:\doit> pip install mypy
```

`mypy` 설치 후 다음과 같이 사용해 보자.

```
1 C:\doit>mypy typing_sample.py
2 typing_sample.py:5: error: Argument 2 to "add" has incompatible type "float";
   expected "int"
3 Found 1 error in 1 file (checked 1 source file)
```

`mypy` 로 `typing_sample.py` 파일을 확인하면 타입이 맞지 않는다는 오류가 발생한다. 다음과 같이 코드를 수정해 보자.

```
1 def add(a: int, b: int) -> int:
2     return a+b
3
4 result = add(3, 4)
5 print(result)
```

오류가 발생했던 3.4 를 `int` 형에 맞게 4 로 변경했다. 그리고 `mypy` 를 다시 실행해 보면 오류가 없다는 것을 알려 준다.

```
1 C:\projects\pylib\>mypy typing_sample.py
2 Success: no issues found in 1 source file
```

파이썬 타입 어노테이션은 요새 쓰임이 점점 늘어나는 추세이다. 많은 프로젝트와 라이브러리에서 파이썬 타입 어노테이션을 적용한 코드가 심심치 않게 발견된다.

08 장 정규표현식

필자는 “정규 표현식”을 이 책 《점프 투 파이썬》에 포함시켜야 할지 오랜시간 고민했다. 왜냐하면 정규 표현식은 꽤 오랜 기간 코드를 작성해 온 프로그래머라도 잘 모를 수 있는 고급 주제여서 초보자를 대상으로 하는 이 책에는 어울리지 않을 수 있기 때문이다.

하지만 정규 표현식을 배워 익히기만 하면 아주 달콤한 열매를 맛볼 수 있다. 그래서 파이썬 하우투 (docs.python.org/3.11/howto/regex.html) 를 참고하여 그곳에서 소개하는 수준의 내용만이라도 독자들이 이해하고 사용할 수 있도록 노력했다. 여러분이 정규 표현식을 잘 다루면 파이썬 외에 또 하나의 강력한 무기를 얻게 되는 것이다.

다시 말하지만 프로그래밍 입문자가 이해하기에는 어려운 내용이니 부담 갖지 말고 편하게 산책하듯 읽어 주기 바란다.

08-1 정규 표현식 살펴보기

정규 표현식 (Regular Expressions) 은 복잡한 문자열을 처리할 때 사용하는 기법으로, 파이썬만의 고유 문법이 아니라 문자열을 처리하는 모든 곳에서 사용한다. 정규 표현식을 배우는 것은 파이썬을 배우는 것과는 또 다른 영역의 과제이다.

정규 표현식은 줄여서 간단히 “정규식” 이라고도 말한다.

정규 표현식은 왜 필요한가?

다음과 같은 문제가 주어졌다고 가정해 보자.

1 주민등록번호를 포함하고 있는 텍스트가 있다. 이 텍스트에 포함된 모든 주민등록번호의 뒷자리를 * 문자로 변경해 보자.

우선 정규식을 전혀 모르면 다음과 같은 순서로 프로그램을 작성해야 할 것이다.

1. 전체 텍스트를 공백 문자로 나눈다 (split).
2. 나뉜 단어가 주민등록번호 형식인지 조사한다.
3. 단어가 주민등록번호 형식이라면 뒷자리를 *로 변환한다.
4. 나뉜 단어를 다시 조립한다.

이를 구현한 코드는 아마도 다음과 같을 것이다.

```
1 data = ""
2 park 800905-1049118
3 kim 700905-1059119
4 ""
5
6 result = []
7 for line in data.split("\n"):
8     word_result = []
9     for word in line.split(" "):
10         if len(word) == 14 and word[:6].isdigit() and word[7:].isdigit():
11             word = word[:6] + "-" + "*****"
12             word_result.append(word)
13         result.append(" ".join(word_result))
14 print("\n".join(result))
```

```
1  결  갯  값 :
2  park 800905-*****
3  kim  700905-*****
```

반면에 정규식을 사용하면 다음처럼 훨씬 간편하고 직관적인 코드를 작성할 수 있다. 아직 정규식 사용 방법을 배우지 않았으니 눈으로만 살펴보자.

```
1  import re
2
3  data = """
4  park 800905-1049118
5  kim  700905-1059119
6  """
7
8  pat = re.compile("(\\d{6})[-]\\d{7}")
9  print(pat.sub("\\g<1>-*****", data))
```

```
1  결  갯  값 :
2  park 800905-*****
3  kim  700905-*****
```

정규 표현식을 사용하면 이렇게 간단한 예제에서도 코드가 상당히 간결해진다. 만약 찾으려는 문자열 또는 바꾸어야 할 문자열의 규칙이 매우 복잡하다면 정규식의 효용은 더 커지게 된다.

이제부터 정규 표현식의 기초부터 심화 부분까지 차근차근 알아보자.

08-2 정규 표현식 시작하기

정규 표현식의 기초, 메타 문자

정규 표현식에서 사용하는 메타 문자 (meta characters) 에는 다음과 같은 것이 있다.

메타 문자란 원래 그 문자가 가진 뜻이 아닌 특별한 용도로 사용하는 문자를 말한다.

```
1 . ^ $ * + ? { } [ ] \ | ( )
```

정규 표현식에 위 메타 문자를 사용하면 특별한 의미를 갖게 된다.

자, 그러면 가장 간단한 정규 표현식부터 시작해 각 메타 문자의 의미와 사용법을 알아보자.

문자 클래스 []

우리가 가장 먼저 살펴볼 메타 문자는 바로 문자 클래스 (character class) 인 [] 이다. 문자 클래스로 만들어진 정규식은 "[] 사이의 문자들과 매치"라는 의미를 갖는다.

문자 클래스를 만드는 메타 문자인 [] 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 돕기 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 사용하면 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 사용한 문자 클래스의 사용 예이다.

- [a-zA-Z]: 알파벳 모두

- `[0-9]`: 숫자

문자 클래스 (`[]`) 안에는 어떤 문자나 메타 문자도 사용할 수 있지만 주의해야 할 메타 문자가 1 가지 있다. 그것은 바로 `^`인데, 문자 클래스 안에 `^` 메타 문자를 사용할 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]` 등은 무척 자주 사용하는 정규 표현식이다. 이렇게 자주 사용하는 정규식은 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 공백문자 (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 있다.

Dot(.)

정규 표현식의 `Dot(.)` 메타 문자는 줄바꿈 문자인 `\n`을 제외한 모든 문자와 매치됨을 의미한다.

나중에 배우겠지만 정규식을 작성할 때 `re.DOTALL` 옵션을 주면 `\n` 문자와도 매치된다.

다음 정규식을 보자.

```
1 a.b
```

위 정규식의 의미는 다음과 같다.

“a + 모든문자 + b”

즉 `a` 와 `b` 라는 문자 사이에 어떤 문자가 들어가도 모두 매치된다는 의미이다.

이해를 돕기 위해 문자열 “aab”, “a0b”, “abc” 가 정규식 `a.b`와 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.

다음 정규식을 보자.

```
1 a[.]b
```

이 정규식의 의미는 다음과 같다.

“a + Dot(.) 문자 + b”

따라서 정규식 a[.]b는 “a.b” 문자열과 매치되고, “a0b” 문자열과는 매치되지 않는다.

만약 앞에서 살펴본 문자 클래스 ([]) 내에 Dot(.) 메타 문자가 사용된다면 이것은 “모든 문자”라는 의미가 아닌 문자 . 그대로를 의미한다. 혼동하지 않도록 주의하자.

반복 (*)

다음 정규식을 보자.

```
1 ca*t
```

이 정규식에는 반복을 의미하는 * 메타 문자가 사용되었다. 여기에서 사용한 *은 * 바로 앞에 있는 문자 a가 0 부터 무한대로 반복될 수 있다는 의미이다.

여기에서 * 메타 문자의 반복 개수가 무한대라고 표현했는데 사실 메모리 제한으로 2 억 개 정도만 가능하다고 한다.

즉 다음과 같은 문자열이 모두 매치된다.

정규식	문자열	Match 여부	설명
ca*t	ct	Yes	“a”가 0 번 반복되어 매치
ca*t	cat	Yes	“a”가 0 번 이상 반복되어 매치 (1 번 반복)
ca*t	caaat	Yes	“a”가 0 번 이상 반복되어 매치 (3 번 반복)

반복 (+)

반복을 나타내는 또 다른 메타 문자로 +가 있다. +는 최소 1 번 이상 반복될 때 사용한다. 즉 *가 반복 횟수 0 부터라면 +는 반복 횟수 1 부터인 것이다.

다음 정규식을 보자.

```
1 ca+t
```

위 정규식의 의미는 다음과 같다.

“c + a(1 번 이상 반복) + t”

위 정규식에 대한 매치여부는 다음 표와 같다.

정규식	문자열	Match 여부	설명
ca+t	ct	No	“a” 가 0 번 반복되어 매치되지 않음
ca+t	cat	Yes	“a” 가 1 번 이상 반복되어 매치 (1 번 반복)
ca+t	caaat	Yes	“a” 가 1 번 이상 반복되어 매치 (3 번 반복)

반복 ({m,n}, ?)

여기에서 잠깐 생각해 볼 게 있다. 반복 횟수를 3 회만 또는 1 회부터 3 회까지만으로 제한하고 싶을 수도 있지 않을까?

{ } 메타 문자를 사용하면 반복 횟수를 고정할 수 있다. {m, n} 정규식을 사용하면 반복 횟수가 m 부터 n 까지 매치할 수 있다. 또한 m 또는 n 을 생략할 수도 있다. 만약 {3,} 처럼 사용하면 반복 횟수가 3 이상인 경우이고 {,3} 처럼 사용하면 반복 횟수가 3 이하를 의미한다. 생략된 m 은 0 과 동일하며, 생략된 n 은 무한대 (2 억 개 미만) 의 의미를 갖는다.

{1,} 은 + 와 동일하고, {0,} 은 * 와 동일하다.

{ } 을 사용한 몇 가지 정규식을 살펴보자.

1. {m}

```
1 ca{2}t
```

위 정규식의 의미는 다음과 같다.

“c + a(반드시 2 번 반복) + t”

위 정규식에 대한 매치여부는 다음 표와 같다.

정규식	문자열	Match 여부	설명
<code>ca{2}t</code>	cat	No	“a”가 1 번만 반복되어 매치되지 않음
<code>ca{2}t</code>	caat	Yes	“a”가 2 번 반복되어 매치

2. {m, n}

```
1 ca{2,5}t
```

위 정규식의 의미는 다음과 같다:

“c + a(2~5 회 반복) + t”

위 정규식에 대한 매치여부는 다음 표와 같다.

정규식	문자열	Match 여부	설명
<code>ca{2,5}t</code>	cat	No	“a”가 1 번만 반복되어 매치되지 않음
<code>ca{2,5}t</code>	caat	Yes	“a”가 2 번 반복되어 매치
<code>ca{2,5}t</code>	caaaaat	Yes	“a”가 5 번 반복되어 매치

3. ?

반복은 아니지만 이와 비슷한 개념으로 ? 이 있다. ? 메타문자가 의미하는 것은 {0, 1} 이다.

다음 정규식을 보자.

```
1 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
<code>ab?c</code>	abc	Yes	“b”가 1 번 사용되어 매치

정규식	문자열	Match 여부	설명
<code>ab?c</code>	<code>ac</code>	Yes	“b”가 0 번 사용되어 매치

즉 **b** 문자가 있거나 없거나 둘 다 매치되는 경우이다.

`*`, `+`, `?` 메타 문자는 모두 `{m, n}` 형태로 고쳐 쓰는 것이 가능하지만 가급적 이해하기 쉽고 표현도 간결한 `*`, `+`, `?` 메타 문자를 사용하는 것이 좋다.

지금까지 아주 기초적인 정규 표현식에 대해서 알아보았다. 알아야 할 것들이 아직 많이 남아 있지만 그
에 앞에서 파이썬으로 이러한 정규 표현식을 어떻게 사용할 수 있는지 먼저 알아보기로 하자.

파이썬에서 정규 표현식을 지원하는 re 모듈

파이썬은 정규 표현식을 지원하기 위해 **re**(regular expression의 약어) 모듈을 제공한다. **re** 모듈은 파이썬을 설치할 때 자동으로 설치되는 표준 라이브러리로 사용 방법은 다음과 같다.

```
1 >>> import re
2 >>> p = re.compile('ab*')
```

re.compile 을 사용하여 정규 표현식 (위 예에서는 `ab*`) 을 컴파일한다. **re.compile** 의 결과로 돌려주는 객체 **p**(컴파일된 패턴 객체) 를 사용하여 그 이후의 작업을 수행할 것이다.

- 정규식을 컴파일할 때 특정 옵션을 주는 것도 가능한데, 이에 대해서는 뒤에서 자세히 살펴본다.
- 패턴이란 정규식을 컴파일한 결과이다.

정규식을 이용한 문자열 검색

이제 컴파일된 패턴 객체를 사용하여 문자열 검색을 수행해 보자. 컴파일된 패턴 객체는 다음과 같은 4 가지 메서드를 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.

Method	목적
--------	----

<code>finditer()</code>	정규식과 매치되는 모든 문자열 (substring) 을 반복 가능한 객체로 리턴한다.
-------------------------	---

`match`, `search` 는 정규식과 매치될 때는 `match` 객체를 리턴하고, 매치되지 않을 때는 `None` 을 리턴한다. 이들 메서드에 대한 간단한 예를 살펴보자.

`match` 객체란 정규식의 검색 결과로 리턴된 객체를 말한다.

우선 다음과 같은 패턴을 만들어 보자.

```
1 >>> import re
2 >>> p = re.compile('[a-z]+')
```

match

`match` 메서드는 문자열의 처음부터 정규식과 매치되는지 조사한다. 위 패턴에 `match` 메서드를 수행해 보자.

```
1 >>> m = p.match("python")
2 >>> print(m)
3 <re.Match object; span=(0, 6), match='python'>
```

“python” 문자열은 `[a-z]+` 정규식에 부합되므로 `match` 객체를 돌려준다.

```
1 >>> m = p.match("3 python")
2 >>> print(m)
3 None
```

“3 python” 문자열은 처음에 나오는 문자 3 이 정규식 `[a-z]+`에 부합되지 않으므로 `None` 을 돌려준다.

`match` 의 결과로 `match` 객체 또는 `None` 을 리턴하기 때문에 파이썬 정규식 프로그램은 보통 다음과 같은 흐름으로 작성한다.

```
1 p = re.compile(정규표현식)
2 m = p.match('string goes here')
3 if m:
4     print('Match found: ', m.group())
5 else:
6     print('No match')
```

즉 `match` 의 결과값이 있을 때만 그다음 작업을 수행하겠다는 것이다.

search

컴파일된 패턴 객체 `p` 를 가지고 이번에는 `search` 메서드를 수행해 보자.

```
1 >>> m = p.search("python")
2 >>> print(m)
3 <re.Match object; span=(0, 6), match='python'>
```

“python” 문자열에 `search` 메서드를 수행하면 `match` 메서드를 수행했을 때와 동일하게 매치된다.

```
1 >>> m = p.search("3 python")
2 >>> print(m)
3 <re.Match object; span=(2, 8), match='python'>
```

“3 python” 문자열의 첫 번째 문자는 “3” 이지만 `search` 는 문자열의 처음부터 검색하는 것이 아니라 문자열 전체를 검색하기 때문에 “3” 이후의 “python” 문자열과 매치된다.

이렇듯 `match` 메서드와 `search` 메서드는 문자열의 처음부터 검색할지의 여부에 따라 다르게 사용해야 한다.

findall

이번에는 `findall` 메서드를 수행해 보자.

```
1 >>> result = p.findall("life is too short")
2 >>> print(result)
3 ['life', 'is', 'too', 'short']
```

`findall` 은 패턴 (`[a-z]+`) 과 매치되는 모든 값을 찾아 리스트로 리턴한다.

finditer

이번에는 `finditer` 메서드를 수행해 보자.

```
1 >>> result = p.finditer("life is too short")
2 >>> print(result)
3 <callable_iterator object at 0x01F5E390>
4 >>> for r in result: print(r)
5 ...
6 <re.Match object; span=(0, 4), match='life'>
7 <re.Match object; span=(5, 7), match='is'>
8 <re.Match object; span=(8, 11), match='too'>
9 <re.Match object; span=(12, 17), match='short'>
```

`finditer` 는 `findall` 과 동일하지만 그 결과로 반복 가능한 객체 (iterator object) 를 리턴한다. 그리고 반복 가능한 객체가 포함하는 각각의 요소는 `match` 객체이다.

match 객체의 메서드

자, 이제 `match` 메서드와 `search` 메서드를 수행한 결과로 리턴된 `match` 객체에 대해 알아보자. 앞에서 정규식을 사용한 문자열 검색을 수행하면서 아마도 다음과 같은 궁금증이 생겼을 것이다.

- 어떤 문자열이 매치되었는가?
- 매치된 문자열의 인덱스는 어디서부터 어디까지인가?

`match` 객체의 메서드를 사용하면 이 같은 궁금증을 해결할 수 있다. 다음 표를 보자.

method	목적
<code>group()</code>	매치된 문자열을 리턴한다.
<code>start()</code>	매치된 문자열의 시작 위치를 리턴한다.
<code>end()</code>	매치된 문자열의 끝 위치를 리턴한다.
<code>span()</code>	매치된 문자열의 (시작, 끝)에 해당하는 튜플을 리턴한다.

다음 예로 확인해 보자.

```
1 >>> m = p.match("python")
2 >>> m.group()
3 'python'
4 >>> m.start()
5 0
6 >>> m.end()
7 6
8 >>> m.span()
9 (0, 6)
```

예상한 대로 결과값이 출력되는 것을 확인할 수 있다. `match` 메서드를 수행한 결과로 돌려준 `match` 객체의 `start()`의 결과값은 항상 0 일 수밖에 없다. 왜냐하면 `match` 메서드는 항상 문자열의 시작부터 조사하기 때문이다.

만약 `search` 메서드를 사용했다면 `start()` 값은 다음과 같이 다르게 나올 것이다.

```
1 >>> m = p.search("3 python")
2 >>> m.group()
3 'python'
4 >>> m.start()
5 2
6 >>> m.end()
7 8
8 >>> m.span()
9 (2, 8)
```

모듈 단위로 수행하기

지금까지 우리는 `re.compile`을 사용하여 컴파일된 패턴 객체로 그 이후의 작업을 수행했다. `re` 모듈은 이것을 보다 축약한 형태로 사용할 수 있는 방법을 제공한다. 다음 예를 보자.

```
1 >>> p = re.compile('[a-z]+')
2 >>> m = p.match("python")
```

위 코드가 축약된 형태는 다음과 같다.

```
1 >>> 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.compile`을 사용하는 것이 편하다.

컴파일 옵션

정규식을 컴파일할 때 다음 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.DOTALL`처럼 전체 옵션 이름을 써도 되고 `re.S`처럼 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.

다음 예를 보자.

```
1 >>> import re
2 >>> p = re.compile('a.b')
3 >>> m = p.match('a\nb')
4 >>> print(m)
5 None
```

정규식이 `a.b`인 경우 문자열 `a\nb`는 매치되지 않음을 알 수 있다. 왜냐하면 `\n`은 `.` 메타 문자와 매치되지 않기 때문이다. `\n` 문자와도 매치되게 하려면 다음과 같이 `re.DOTALL` 옵션을 사용해야 한다.

```
1 >>> p = re.compile('a.b', re.DOTALL)
2 >>> m = p.match('a\nb')
3 >>> print(m)
4 <re.Match object; span=(0, 3), match='a\nb'>
```

보통 `re.DOTALL` 옵션은 여러 줄로 이루어진 문자열에서 줄바꿈 문자에 상관없이 검색할 때 많이 사용한다.

IGNORECASE, I

`re.IGNORECASE` 또는 `re.I` 옵션은 대소문자 구별 없이 매치를 수행할 때 사용하는 옵션이다. 다음 예를 보자.

```
1 >>> p = re.compile('[a-z]+', re.I)
2 >>> p.match('python')
3 <re.Match object; span=(0, 6), match='python'>
4 >>> p.match('Python')
5 <re.Match object; span=(0, 6), match='Python'>
6 >>> p.match('PYTHON')
7 <re.Match object; span=(0, 6), match='PYTHON'>
```

`[a-z]+` 정규식은 소문자만을 의미하지만 `re.I` 옵션으로 대소문자 구별 없이 매치된다.

MULTILINE, M

`re.MULTILINE` 또는 `re.M` 옵션은 조금 후에 설명할 메타 문자인 `^`, `$`와 연관된 옵션이다. 이 메타 문자에 대해 간단히 설명하자면 `^`는 문자열의 처음을 의미하고, `$`는 문자열의 마지막을 의미한다. 예를 들어 정규식이 `^python`인 경우 문자열의 처음은 항상 `python`으로 시작해야 매치되고, 만약 정규식이 `python$`이라면 문자열의 마지막은 항상 `python`으로 끝나야 매치된다는 의미이다.

다음 예를 보자.

```
1 import re
2 p = re.compile("^python\s\w+")
3
4 data = """python one
5 life is too short
6 python two
7 you need python
8 python three"""
9
10 print(p.findall(data))
```

정규식 `^python\s\w+`은 `python`이라는 문자열로 시작하고 그 뒤에 `whitespace`, 그 뒤에 단어가 와야 한다는 의미이다. 검색할 문자열 `data`는 여러 줄로 이루어져 있다.

이 스크립트를 실행하면 다음과 같은 결과를 돌려준다.

```
1 ['python one']
```

^ 메타 문자에 의해 `python` 이라는 문자열을 사용한 첫 번째 줄만 매치된 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.

```
1 import re
2 p = re.compile("^python\s\w+", re.MULTILINE)
3
4 data = """python one
5 life is too short
6 python two
7 you need python
8 python three"""
9
10 print(p.findall(data))
```

`re.MULTILINE` 옵션으로 인해 ^ 메타 문자가 문자열 전체가 아닌 각 줄의 처음이라는 의미를 갖게 되었다. 이 스크립트를 실행하면 다음과 같은 결과가 출력된다.

```
1 ['python one', 'python two', 'python three']
```

즉 `re.MULTILINE` 옵션은 ^, \$ 메타 문자를 문자열의 각 줄마다 적용해 주는 것이다.

VERBOSE, X

지금껏 알아본 정규식은 매우 간단하지만 정규식 전문가들이 만든 정규식을 보면 거의 암호수준이다. 정규식을 이해하려면 하나하나 조심스럽게 뜯어보아야만 한다. 이렇게 이해하기 어려운 정규식을 주석 또는 줄 단위로 구분할 수 있다면 얼마나 보기 좋고 이해하기 쉬울까? 방법이 있다. 바로 `re.VERBOSE` 또는 `re.X` 옵션을 사용하면 된다.

다음 예를 보자.

```
1 charref = re.compile(r'&[#](0[0-7]+|0[0-9]+|x[0-9a-fA-F]+);')
```

위 정규식이 쉽게 이해되는가? 이제 다음 예를 보자.

```
1 charref = re.compile(r"""
2 &[#]           # Start of a numeric entity reference
3 (
4     0[0-7]+    # Octal form
5     | 0[0-9]+  # Decimal form
6     | x[0-9a-fA-F]+ # Hexadecimal form
```

```

7 )
8 ; # Trailing semicolon
9 """, re.VERBOSE)

```

첫 번째와 두 번째 예를 비교해 보면 컴파일된 패턴 객체인 `charref` 는 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 `whitespace` 는 컴파일할 때 제거된다 (단 `[]` 안에 사용한 `whitespace` 는 제외). 그리고 줄 단위로 `#` 기호를 사용하여 주석문을 작성할 수 있다.

백슬래시 문제

정규 표현식을 파이썬에서 사용할 때 혼란을 주는 요소가 한 가지 있는데, 바로 백슬래시 (`\`) 이다.

예를 들어 어떤 파일 안에 있는 `"\section"` 문자열을 찾기 위한 정규식을 만든다고 가정해 보자.

```
1 \section
```

이 정규식은 `\s` 문자가 `whitespace` 로 해석되어 의도한 대로 매치가 이루어지지 않는다.

위 표현은 다음과 동일한 의미이다.

```
1 [ \t\n\r\f\v]ection
```

의도한 대로 매치하고 싶다면 다음과 같이 변경해야 한다.

```
1 \\section
```

즉 위 정규식에서 사용한 `\` 문자가 문자열 자체임을 알려 주기 위해 백슬래시 2 개를 사용하여 이스케이프 처리를 해야 한다.

따라서 위 정규식을 컴파일하려면 다음과 같이 작성해야 한다.

```
1 >>> p = re.compile('\\section')
```

그런데 여기에서 또 하나의 문제가 발견된다. 위처럼 정규식을 만들어서 컴파일하면 실제 파이썬 정규식 엔진에는 파이썬 문자열 리터럴 규칙에 따라 `\\` 이 `\` 로 변경되어 `\section` 이 전달된다.

이 문제는 위와 같은 정규식을 파이썬에서 사용할 때만 발생한다 (파이썬의 리터럴 규칙). 유닉스의 `grep`, `vi` 등에서는 이러한 문제가 없다.

결국 정규식 엔진에 `\\` 문자를 전달하려면 파이썬은 `\\\\` 처럼 백슬래시를 4 개나 사용해야 한다.

정규식 엔진은 정규식을 해석하고 수행하는 모듈이다.

```
1 >>> p = re.compile('\\\\\\section')
```

이렇게 해야만 원하는 결과를 얻을 수 있다. 하지만 너무 복잡하지 않은가?

만약 위와 같이 \를 사용한 표현이 계속 반복되는 정규식이라면 너무 복잡해서 이해하기 쉽지않을 것이다. 이러한 문제를 해결하려면 **Raw String**을 사용해야 한다. 그 방법은 다음과 같다.

```
1 >>> p = re.compile(r'\\section')
```

위와 같이 정규식 문자열 앞에 **r** 문자를 삽입하면 이 정규식은 **Raw String** 규칙에 의하여 백슬래시 2 개 대신 1 개만 써도 2 개를 쓴 것과 동일한 의미를 갖게 된다.

만약 백슬래시를 사용하지 않는 정규식이라면 **r**의 유무에 상관없이 동일한 정규식이 될 것이다.

08-3 강력한 정규 표현식의 세계로

이제 07-2 에서 배우지 않은 몇몇 메타 문자의 의미를 살펴보고 그룹 (Group) 을 만드는 법, 전방 탐색 등 더욱 강력한 정규 표현식에 대해서 살펴보자.

메타문자

아직 살펴보지 않은 메타 문자에 대해서 모두 살펴보자. 여기에서 다룰 메타 문자는 앞에서 살펴본 메타 문자와 성격이 조금 다르다. 앞에서 살펴본 `+`, `*`, `[]`, `{}` 등의 메타문자는 매치가 진행될 때 현재 매치되고 있는 문자열의 위치가 변경된다 (보통 소비된다고 표현한다). 하지만 이와 달리 문자열을 소비시키지 않는 메타 문자도 있다. 이번에는 이런 문자열 소비가 없는 (zerowidth assertions) 메타 문자에 대해 살펴보자.

|

| 메타 문자는 or 과 동일한 의미로 사용된다. `A|B`라는 정규식이 있다면 A 또는 B 라는 의미가 된다.

```
1 >>> p = re.compile('Crow|Servo')
2 >>> m = p.match('CrowHello')
3 >>> print(m)
4 <re.Match object; span=(0, 4), match='Crow'>
```

^

^ 메타 문자는 문자열의 맨 처음과 일치함을 의미한다. 앞에서 살펴본 컴파일 옵션 `re.MULTILINE`을 사용할 경우에는 여러 줄의 문자열일 때 각 줄의 처음과 일치하게 된다.

다음 예를 보자.

```
1 >>> print(re.search('^Life', 'Life is too short'))
2 <re.Match object; span=(0, 4), match='Life'>
3 >>> print(re.search('^Life', 'My Life'))
4 None
```

`^Life` 정규식은 `Life` 문자열이 처음에 온 경우에는 매치하지만 처음 위치가 아닌 경우에는 매치되지 않음을 알 수 있다.

\$

`$` 메타 문자는 `^` 메타 문자와 반대의 경우이다. 즉 `$`는 문자열의 끝과 매치함을 의미한다.

다음 예를 보자.

```
1 >>> print(re.search('short$', 'Life is too short'))
2 <re.Match object; span=(12, 17), match='short'>
3 >>> print(re.search('short$', 'Life is too short, you need python'))
4 None
```

`short$` 정규식은 검색할 문자열이 `short`로 끝난 경우에는 매치되지만 그 이외의 경우에는 매치되지 않음을 알 수 있다.

^ 또는 `$` 문자를 메타 문자가 아닌 문자 그 자체로 매치하고 싶은 경우에는 `\^`, `\$`로 사용하면 된다.

\A

`\A`는 문자열의 처음과 매치됨을 의미한다. `^` 메타 문자와 동일한 의미이지만 `re.MULTILINE` 옵션을 사용할 경우에는 다르게 해석된다. `re.MULTILINE` 옵션을 사용할 경우 `^`은 각 줄의 문자열의 처음과 매치되지만 `\A`는 줄과 상관없이 전체 문자열의 처음과만 매치된다.

\Z

`\Z`는 문자열의 끝과 매치됨을 의미한다. 이것 역시 `\A`와 동일하게 `re.MULTILINE` 옵션을 사용할 경우 `$` 메타 문자와는 달리 전체 문자열의 끝과 매치된다.

\b

`\b`는 단어 구분자 (Word boundary) 이다. 보통 단어는 `whitespace`에 의해 구분된다.

다음 예를 보자.

```
1 >>> p = re.compile(r'\bclass\b')
2 >>> print(p.search('no class at all'))
3 <re.Match object; span=(3, 8), match='class'>
```

`\bclass\b` 정규식은 앞뒤가 `whitespace` 로 구분된 `class` 라는 단어와 매치됨을 의미한다. 따라서 `no class at all` 의 `class` 라는 단어와 매치됨을 확인할 수 있다.

```
1 >>> print(p.search('the declassified algorithm'))
2 None
```

위 예의 `the declassified algorithm` 문자열 안에도 `class` 문자열이 포함되어 있긴 하지만 `whitespace` 로 구분된 단어가 아니므로 매치되지 않는다.

```
1 >>> print(p.search('one subclass is'))
2 None
```

`subclass` 문자열 역시 `class` 앞에 `sub` 문자열이 더해져 있으므로 매치되지 않음을 알 수 있다.

`\b` 메타 문자를 사용할 때 주의해야 할 점이 있다. `\b`는 파이썬 리터럴 규칙에 의하면 백스페이스 (BackSpace) 를 의미하므로 백스페이스가 아닌 단어 구분자임을 알려 주기 위해 `r'\bclass\b'` 처럼 Raw string 임을 알려주는 기호 `r` 을 반드시 붙여 주어야 한다.

\B

`\B` 메타 문자는 `\b` 메타 문자와 반대의 경우이다. 즉 `whitespace` 로 구분된 단어가 아닌 경우에만 매치된다.

```
1 >>> p = re.compile(r'\Bclass\B')
2 >>> print(p.search('no class at all'))
3 None
4 >>> print(p.search('the declassified algorithm'))
5 <re.Match object; span=(6, 11), match='class'>
6 >>> print(p.search('one subclass is'))
7 None
```

`class` 단어의 앞뒤에 `whitespace` 가 하나라도 있는 경우에는 매치가 안 되는 것을 확인할 수 있다.

그루핑

ABC 문자열이 계속해서 반복되는지 조사하는 정규식을 작성하고 싶다고 하자. 어떻게 해야할까? 지금까지 공부한 내용으로는 위 정규식을 작성할 수 없다. 이럴 때 필요한 것이 바로 그루핑 (Grouping) 이다.

위 경우는 다음처럼 그루핑을 사용하여 작성할 수 있다.

```
1 (ABC)+
```

그룹을 만들어 주는 메타 문자는 바로 `()` 이다.

```

1 >>> p = re.compile('(ABC)+')
2 >>> m = p.search('ABCABCABC OK?')
3 >>> print(m)
4 <re.Match object; span=(0, 9), match='ABCABCABC'>
5 >>> print(m.group())
6 ABCABCABC

```

다음 예를 보자.

```

1 >>> p = re.compile(r"\w+\s+\d+[-]\d+[-]\d+")
2 >>> m = p.search("park 010-1234-1234")

```

`\w+\s+\d+[-]\d+[-]\d+`은 이름 + " " + 전화번호 형태의 문자열을 찾는 정규식이다. 그런데 이렇게 매치된 문자열 중에서 이름만 뽑아내고 싶다면 어떻게 해야 할까?

보통 반복되는 문자열을 찾을 때 그룹을 사용하는데, 그룹을 사용하는 보다 큰 이유는 위에서 볼 수 있듯이 매치된 문자열 중에서 특정 부분의 문자열만 뽑아내기 위해서인 경우가 더 많다.

위 예에서 만약 “이름” 부분만 뽑아내려 한다면 다음과 같이 할 수 있다.

```

1 >>> p = re.compile(r"(\w+)\s+\d+[-]\d+[-]\d+")
2 >>> m = p.search("park 010-1234-1234")
3 >>> print(m.group(1))
4 park

```

이름에 해당하는 `\w+` 부분을 그룹 (`\w+`)으로 만들면 `match` 객체의 `group`(인덱스) 메서드를 사용하여 그룹핑된 부분의 문자열만 뽑아낼 수 있다. `group` 메서드의 인덱스는 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

다음 예제를 계속해서 보자.

```

1 >>> p = re.compile(r"(\w+)\s+(\d+[-]\d+[-]\d+)")
2 >>> m = p.search("park 010-1234-1234")
3 >>> print(m.group(2))
4 010-1234-1234

```

이번에는 전화번호 부분을 추가로 그룹 (`\d+[-]\d+[-]\d+`)로 만들었다. 이렇게 하면 `group(2)`처럼 사용하여 전화번호만 뽑아낼 수 있다.

만약 전화번호 중에서 국번만 뽑아내고 싶으면 어떻게 해야 할까? 다음과 같이 국번 부분을 또 그룹핑하면 된다.

```
1 >>> p = re.compile(r'(\w+)\s+(\d+)[-]\d+[-]\d+')
2 >>> m = p.search("park 010-1234-1234")
3 >>> print(m.group(3))
4 010
```

위 예에서 볼 수 있듯이 `(\w+)\s+(\d+)[-]\d+[-]\d+`처럼 그룹을 중첩되게 사용하는 것도 가능하다. 그룹이 중첩되어 있는 경우는 바깥쪽부터 시작하여 안쪽으로 들어갈수록 인덱스가 증가한다.

그룹핑된 문자열 재참조하기

그룹의 또 하나 좋은 점은 한 번 그룹핑한 문자열을 재참조 (Backreferences) 할 수 있다는 점이다. 다음 예를 보자.

```
1 >>> p = re.compile(r'(\b\w+)\s+\1')
2 >>> p.search('Paris in the the spring').group()
3 'the the'
```

정규식 `(\b\w+)\s+\1`은 (그룹) + `" "` + 그룹과 동일한 단어와 매치됨을 의미한다. 이렇게 정규식을 만들게 되면 2개의 동일한 단어를 연속적으로 사용해야만 매치된다. 이것을 가능하게 해주는 것이 바로 재참조 메타 문자인 `\1`이다. `\1`은 정규식의 그룹 중 첫 번째 그룹을 가리킨다.

두 번째 그룹을 참조하려면 `\2`를 사용하면 된다.

그룹핑된 문자열에 이름 붙이기

정규식 안에 그룹이 무척 많아진다고 가정해 보자. 예를 들어 정규식 안에 그룹이 10개 이상만 되어도 매우 혼란스러울 것이다. 거기에 더해 정규식이 수정되면서 그룹이 추가, 삭제되면 그 그룹을 인덱스로 참조한 프로그램도 모두 변경해 주어야 하는 위험도 갖게 된다.

만약 그룹을 인덱스가 아닌 이름 (Named Groups)으로 참조할 수 있다면 어떨까? 그렇다면 이런 문제에서 해방되지 않을까?

이러한 이유로 정규식은 그룹을 만들 때 그룹 이름을 지정할 수 있게 했다. 그 방법은 다음과 같다.

```
1 (?P<name>\w+)\s+(\d+)[-]\d+[-]\d+
```

위 정규식은 앞에서 본 이름과 전화번호를 추출하는 정규식이다. 기존과 달라진 부분은 다음과 같다.

```
(\w+) -> (?P<name>\w+)
```


대단히 복잡해진 것처럼 보이지만 (`\w+`)라는 그룹에 `name`이라는 이름을 붙인 것에 불과하다. 여기에서 사용한 (`?...`) 표현식은 정규 표현식의 확장 구문이다. 이 확장 구문을 사용하기 시작하면 가독성이 상당히 떨어지긴 하지만 반면에 강력함을 갖게 된다.

그룹에 이름을 지어 주려면 다음과 같은 확장 구문을 사용해야 한다.

```
1 (?P<그룹명>...)
```

그룹에 이름을 지정하고 참조하는 다음 예를 보자.

```
1 >>> p = re.compile(r'(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)')
2 >>> m = p.search("park 010-1234-1234")
3 >>> print(m.group("name"))
4 park
```

위 예에서 볼 수 있듯이 `name`이라는 그룹 이름으로 참조할 수 있다.

그룹 이름을 사용하면 정규식 안에서 재참조하는 것도 가능하다.

```
1 >>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
2 >>> p.search('Paris in the the spring').group()
3 'the the'
```

위 예에서 볼 수 있듯이 재참조할 때에는 (`?P=그룹이름`)이라는 확장 구문을 사용해야 한다.

전방 탐색

정규식에 막 입문한 사람들이 가장 어려워하는 것이 바로 전방 탐색 (Lookahead Assertions) 확장 구문이다. 정규식 안에 이 확장 구문을 사용하면 순식간에 암호문처럼 알아보기 어렵게 바뀌기 때문이다. 하지만 이 전방 탐색이 꼭 필요한 경우가 있으며 매우 유용한 경우도 많으니 꼭 알아 두자.

다음 예를 보자.

```
1 >>> p = re.compile("http:")
2 >>> m = p.search("http://google.com")
3 >>> print(m.group())
4 http:
```

정규식 `.+:`과 일치하는 문자열로 `http:`를 돌려주었다. 만약 `http:`라는 검색 결과에서:을 제외하고 출력하려면 어떻게 해야 할까? 위 예는 그나마 간단하지만 훨씬 복잡한 정규식이어서 그루핑은 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 2 종류가 있고 다음과 같이 표현한다.

- 긍정형 전방 탐색 ((?=...))-...에 해당되는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소비되지 않는다.
- 부정형 전방 탐색 ((?!...))-...에 해당되는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소비되지 않는다.

긍정형 전방 탐색

긍정형 전방 탐색을 사용하면 `http:`의 결과를 `http`로 바꿀 수 있다. 다음 예를 보자.

```
1 >>> p = re.compile(".*(?:)")
2 >>> m = p.search("http://google.com")
3 >>> print(m.group())
4 http
```

정규식 중 `:`에 해당하는 부분에 긍정형 전방 탐색 기법을 적용하여 `(?:)`으로 변경하였다. 이렇게 되면 기존 정규식과 검색에서는 동일한 효과를 발휘하지만 `:`에 해당하는 문자열이 정규식 엔진에 의해 소비되지 않아 (검색에는 포함되지만 검색 결과에는 제외됨) 검색 결과에서는 `:`이 제거된 후 돌려주는 효과가 있다.

부정형 전방 탐색

자, 이번에는 다음 정규식을 보자.

```
1 .*[.].*
$$\$$$

```

이 정규식은 `파일이름` + `.` + `확장자`를 나타내는 정규식이다. 이 정규식은 `foo.bar`, `autoexec.bat`, `sendmail.cf` 같은 형식의 파일과 매치될 것이다.

이 정규식에 확장자가 “bat”인 파일은 제외해야 한다”는 조건을 추가해 보자. 가장 먼저 생각할 수 있는 정규식은 다음과 같다.

```
1 .*[.][^b].*
$$\$$$

```

이 정규식은 확장자가 `b`라는 문자로 시작하면 안 된다는 의미이다. 하지만 이 정규식은 `foo.bar`라는 파일마저 걸러 낸다. 정규식을 다음과 같이 수정해 보자.

```
1 .*[.](^[^b]..|.[^a].|..^[^t])
$$\$$$

```

이 정규식은 `|` 메타 문자를 사용하여 확장자의 첫 번째 문자가 `b`가 아니거나 두 번째 문자가 `a`가 아니거나 세 번째 문자가 `t`가 아닌 경우를 의미한다. 이 정규식에 의하여 `foo.bar`는 제외되지 않고 `autoexec.bat`은 제외되어 만족스러운 결과를 돌려준다. 하지만 이 정규식은 아쉽게도 `sendmail.cf`처럼 확장자의 문자 개수가 2개인 케이스를 포함하지 못하는 오동작을 하기 시작한다.

따라서 다음과 같이 바꾸어야 한다.

```
1 .*[.](^[^b].?|^[^a].?|...^[^t]?)$
```

확장자의 문자 개수가 2 개여도 통과되는 정규식이 만들어졌다. 하지만 정규식은 점점 더 복잡해지고 이해하기 어려워진다.

그런데 여기에서 **bat** 파일 말고 **exe** 파일도 제외하라는 조건이 추가로 생긴다면 어떻게 될까? 이 모든 조건을 만족하는 정규식을 구현하려면 패턴은 더욱더 복잡해질 것이다.

이러한 상황의 구원 투수는 바로 부정형 전방 탐색이다. 위 예는 부정형 전방 탐색을 사용하면 다음과 같이 간단하게 처리된다.

```
1 .*[.](?!bat$).*
```

확장자가 **bat** 가 아닌 경우에만 통과된다는 의미이다. **bat** 문자열이 있는지 조사하는 과정에서 문자열이 소비되지 않으므로 **bat** 가 아니라고 판단되면 그 이후 정규식 매치가 진행된다.

exe 역시 제외하라는 조건이 추가되더라도 다음과 같이 간단히 표현할 수 있다.

```
1 .*[.](?!bat$|exe$).*
```

문자열 바꾸기

sub 메서드를 사용하면 정규식과 매치되는 부분을 다른 문자로 쉽게 바꿀 수 있다.

다음 예를 보자.

```
1 >>> p = re.compile('(blue|white|red)')
2 >>> p.sub('colour', 'blue socks and red shoes')
3 'colour socks and colour shoes'
```

sub 메서드의 첫 번째 인수는 “바꿀 문자열 (replacement)” 이 되고, 두 번째 인수는 “대상 문자열” 이 된다. 위 예에서 볼 수 있듯이 **blue** 또는 **white** 또는 **red** 라는 문자열이 **colour** 라는 문자열로 바뀌는 것을 확인할 수 있다.

그런데 딱 한 번만 바꾸고 싶은 경우도 있다. 이렇게 바꾸기 횟수를 제어하려면 다음과 같이 세 번째 인수에 **count** 값을 설정하면 된다.

```
1 >>> p.sub('colour', 'blue socks and red shoes', count=1)
2 'colour socks and red shoes'
```

처음 일치하는 **blue** 만 **colour** 라는 문자열로 한 번만 바꾸기가 실행되는 것을 알 수 있다.

sub 메서드와 유사한 subn 메서드

subn 역시 sub 와 동일한 기능을 하지만 반환 결과를 튜플로 리턴한다는 차이가 있다. 리턴된 튜플의 첫 번째 요소는 변경된 문자열이고, 두 번째 요소는 바꾸기가 발생한 횟수이다.

```
1 >>> p = re.compile('(blue|white|red)')
2 >>> p.subn('colour', 'blue socks and red shoes')
3 ('colour socks and colour shoes', 2)
```

sub 메서드 사용 시 참조 구문 사용하기

sub 메서드를 사용할 때 참조 구문을 사용할 수 있다. 다음 예를 보자.

```
1 >>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
2 >>> print(p.sub("\g<phone> \g<name>", "park 010-1234-1234"))
```

```
1 010-1234-1234 park
```

위 예는 이름+ 전화번호의 문자열을 전화번호+ 이름으로 바꾸는 예이다. sub의 바꿀 문자열 부분에 \g<그룹이름>을 사용하면 정규식의 그룹 이름을 참조할 수 있게 된다.

다음과 같이 그룹 이름 대신 참조 번호를 사용해도 마찬가지로 결과를 돌려준다.

```
1 >>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
2 >>> print(p.sub("\g<2> \g<1>", "park 010-1234-1234"))
```

```
1 010-1234-1234 park
```

sub 메서드의 매개변수로 함수 넣기

sub 메서드의 첫 번째 인수에 함수를 전달할 수도 있다. 다음 예를 보자.

```
1 >>> def hexrepl(match):
2 ...     value = int(match.group())
3 ...     return hex(value)
4 ...
5 >>> p = re.compile(r'\d+')
6 >>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
7 'Call 0xffd2 for printing, 0xc000 for user code.'
```

hexrepl 함수는 match 객체 (위에서 숫자에 매치되는) 를 입력으로 받아 16 진수로 변환하여 돌려주는 함수이다. sub의 첫 번째 인수로 함수를 사용할 경우 해당 함수의 첫 번째 매개변수에는 정규식과 매치된 match 객체가 입력된다. 그리고 매치되는 문자열은 함수의 리턴 값으로 바뀌게 된다.

Greedy vs Non-Greedy

정규식에서 Greedy(탐욕스러운)란 어떤 의미일까? 다음 예제를 보자.

```
1 >>> s = '<html><head><title>Title</title>'  
2 >>> len(s)  
3 32  
4 >>> print(re.match('<.*>', s).span())  
5 (0, 32)  
6 >>> print(re.match('<.*>', s).group())  
7 <html><head><title>Title</title>
```

<.*> 정규식의 매치 결과로 <html> 문자열을 돌려주기를 기대했을 것이다. 하지만 * 메타 문자는 매우 탐욕스러워서 매치할 수 있는 최대한의 문자열인 <html><head><title>Title</title> 문자열을 모두 소비해 버렸다. 어떻게 하면 이 탐욕스러움을 제한하고 <html> 문자열까지만 소비하도록 막을 수 있을까?

다음과 같이 non-greedy 문자인 ?를 사용하면 *의 탐욕을 제한할 수 있다.

```
1 >>> print(re.match('<.*?>', s).group())  
2 <html>
```

non-greedy 문자인 ?는 *, +, ?, {m,n}?와 같이 사용할 수 있다. 가능한 한 가장 최소한의 반복을 수행하도록 도와주는 역할을 한다.

09 장 종합문제

파이썬은 웹, GUI, 네트워크, 머신러닝 등 상당히 많은 일을 할 수 있는 언어이다. 여러분이 지금까지 배운 내용을 충분히 숙지했다면 이제 이들을 향해 첫발을 내디딜 준비를 마친 것이다. 하지만 그전에 여기에 준비한 문제들을 풀어 보면서 여러분이 얼마나 파이썬에 익숙해졌는지 점검해 보자. 이곳에 준비한 문제들은 조금 어려울 수 있다. 하지만 파이썬과 함께라면 이 문제들을 해결하는 과정 역시 또 하나의 즐거움이라는 것을 분명 느끼게 될 것이다.

그럼 아무쪼록 즐거운 시간이 되기를 바란다.

Happy Python!!

Q1. 문자열 바꾸기

다음과 같은 문자열이 있다.

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

문자열의 `split` 와 `join` 함수를 사용하여 위 문자열을 다음과 같이 고치시오.

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

Q2. 딕셔너리 값 추출하기

다음은 딕셔너리의 `a` 에서 'C' 라는 `key` 에 해당하는 `value` 를 출력하는 프로그램이다.

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

`a` 딕셔너리에는 'C' 라는 `key` 가 없으므로 위와 같은 오류가 발생한다. 'C' 에 해당하는 `key` 값이 없을 경우 오류 대신 70 을 얻을 수 있도록 수정하시오.

Q3. 리스트의 더하기와 extend 함수

다음과 같은 리스트 a 가 있다.

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

리스트 a 에 [4,5] 를 + 기호를 사용하여 더한 결과는 다음과 같다.

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

리스트 a 에 [4,5] 를 extend 를 사용하여 더한 결과는 다음과 같다.

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

+ 기호를 사용하여 더한 것과 extend 한 것의 차이점이 있을까? 있다면 그 차이점을 설명하시오.

Q4. 리스트 총합 구하기

다음은 A 학급 학생의 점수를 나타내는 리스트이다. 다음 리스트에서 50 점 이상 점수의 총합을 구하시오.

```
1 A = [20, 55, 67, 82, 45, 33, 90, 87, 100, 25]
```

Q5. 피보나치 함수

첫 번째 항의 값이 0 이고 두 번째 항의 값이 1 일 때, 이후에 이어지는 항은 이전의 두 항을 더한 값으로 이루어지는 수열을 피보나치 수열이라고 한다.

```
1 0, 1, 1, 2, 3, 5, 8, 13, ...
```

입력을 정수 n 으로 받았을 때, n 이하까지의 피보나치 수열을 출력하는 함수를 작성해 보자.

Q6. 숫자의 총합 구하기

사용자로부터 다음과 같은 숫자를 입력받아 입력받은 숫자의 총합을 구하는 프로그램을 작성하시오. (단 숫자는 콤마로 구분하여 입력한다.)

```
1 65,45,2,3,45,8
```

Q7. 한 줄 구구단

사용자로부터 2~9 의 숫자 중 하나를 입력받아 해당 숫자의 구구단을 한 줄로 출력하는 프로그램을 작성하시오.

실행 예)

```
1 구구단을 출력할 숫자를 입력하세요 (2~9): 2
2 2 4 6 8 10 12 14 16 18
```

Q8. 파일을 읽어 역순으로 저장하기

다음과 같은 내용의 파일 abc.txt 가 있다.

```
1 AAA
2 BBB
3 CCC
4 DDD
5 EEE
```

이 파일의 내용을 다음과 같이 역순으로 바꾸어 저장하시오.

```
1 EEE
2 DDD
3 CCC
4 BBB
5 AAA
```

Q9. 평균값 구하기

다음과 같이 총 10 줄로 이루어진 sample.txt 파일이 있다. sample.txt 파일의 숫자 값을 모두 읽어 총합과 평균 값을 구한 후 평균 값을 result.txt 파일에 쓰는 프로그램을 작성하시오.


```
1 70
2 60
3 55
4 75
5 95
6 90
7 80
8 80
9 85
10 100
```

Q10. 사칙연산 계산기

다음과 같이 동작하는 클래스 `Calculator` 를 작성하시오.

```
1 >>> cal1 = Calculator([1,2,3,4,5])
2 >>> cal1.sum() # 합 계
3 15
4 >>> cal1.avg() # 평 균
5 3.0
6 >>> cal2 = Calculator([6,7,8,9,10])
7 >>> cal2.sum() # 합 계
8 40
9 >>> cal2.avg() # 평 균
10 8.0
```

Q11. 모듈을 사용하는 방법

`C:\doit` 디렉터리에 `mymod.py` 파이썬 모듈이 있다고 가정해 보자. 명령 프롬프트 창에서 파이썬 셸을 열어 이 모듈을 `import` 해서 사용할 수 있는 방법을 모두 기술하시오. (즉 다음과 같이 `import mymod`를 수행할 때 오류가 없어야 한다.)

```
1 >>> import mymod
2 >>>
```

Q12. 오류와 예외 처리

다음 코드의 실행 결과를 예측하고 그 이유에 대해 설명하시오.

```
1 result = 0
2
3 try:
```

```

4     [1, 2, 3][3]
5     "a"+1
6     4 / 0
7     except TypeError:
8         result += 1
9     except ZeroDivisionError:
10        result += 2
11    except IndexError:
12        result += 3
13    finally:
14        result += 4
15
16    print(result)

```

Q13. DashInsert

DashInsert 함수는 숫자로 구성된 문자열을 입력받은 뒤 문자열 안에서 홀수가 연속되면 두 수 사이에 - 를 추가하고, 짝수가 연속되면 * 를 추가하는 기능을 갖고 있다. DashInsert 함수를 완성하시오.

```

1 입력 예시 : 4546793
2 출력 예시 : 454*67-9-3

```

Q14. 문자열 압축하기

문자열을 입력받아 같은 문자가 연속적으로 반복되는 경우에 그 반복 횟수를 표시해 문자열을 압축하여 표시하시오.

```

1 입력 예시 : aaabbcccccca
2 출력 예시 : a3b2c6a1

```

Q15. Duplicate Numbers

0~9 의 문자로 된 숫자를 입력받았을 때, 이 입력값이 0~9 의 모든 숫자를 각각 한 번씩만 사용한 것인지 확인하는 함수를 작성하시오.

```

1 입력 예시 : 0123456789 01234 01234567890 6789012345 012322456789
2 출력 예시 : True False False True False

```

Q16. 모스 부호 해독

문자열 형식으로 입력받은 모스 부호 (dot: . dash:-) 를 해독하여 영어 문장으로 출력하는 프로그램을 작성하시오.

- 글자와 글자 사이는 공백 1 개, 단어와 단어 사이는 공백 2 개로 구분한다.
- 예를 들어 다음 모스 부호는 “HE SLEEPS EARLY” 로 해석해야 한다.

```
1 ..... . ... .-... . . .-- . ... . - .- .-... -.-
```

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	-. -
M	--	Z	-. .

Q17. 정규식 - 기초 메타 문자

다음 중 정규식 `a[.]{3,}b`과 매치되는 문자열은 무엇일까?

1. `accbb`

2. a....b
3. aaab
4. a.cccb

Q18. 정규식 - 문자열 검색

다음 코드의 결과값은 무엇일까?

```
1 >>> import re
2 >>> p = re.compile("[a-z]+")
3 >>> m = p.search("5 python")
4 >>> m.start() + m.end()
```

Q19. 정규식 - 그룹핑

다음과 같은 문자열에서 휴대폰 번호 뒷자리인 숫자 4 개를 ##### 로 바꾸는 프로그램을 정규식을 사용하여 작성하시오.

```
1 ""
2 park 010-9999-9988
3 kim 010-9909-7789
4 lee 010-8789-7768
5 ""
```

Q20. 정규식 - 전방 탐색

다음은 이메일 주소를 나타내는 정규식이다. 이 정규식은 park@naver.com, kim@daum.net, lee@myhome.co.kr 등과 매치된다. 긍정형 전방 탐색 기법을 사용하여 .com, .net 이 아닌 이메일 주소는 제외시키는 정규식을 작성하시오.

```
1 .*[@].*[*].*.$
```

10 장 풀이

연습 문제 및 종합 문제 풀이

10-1 풀이: 02 장 연습문제

Q1. 평균 점수 구하기

홍길동 씨의 과목별 점수는 각각 다음과 같다. 홍길동 씨의 평균 점수를 구해 보자.

```
1 >>> a = 80
2 >>> b = 75
3 >>> c = 55
4 >>> (a+b+c)/3
5 70.0
```

Q2. 홀수 짝수 판별

자연수 13 이 홀수인지 짝수인지 판별할 수 있는 방법에 대해서 말해 보자.

나머지 연산자를 사용하면 자연수의 홀수, 짝수를 쉽게 판별할 수 있다

```
1 >>> 1 % 2
2 1
3 >>> 2 % 2
4 0
5 >>> 3 % 2
6 1
7 >>> 4 % 2
8 0
```

1, 2, 3, 4 라는 자연수를 2 로 나누었을 때의 나머지 값을 출력하는 예제이다. 결과를 보면 자연수가 홀수 일 때는 1 을 짝수일 때는 0 을 돌려주는 것을 확인할 수 있다.

Q3. 주민등록번호 나누기

홍길동씨의 주민등록번호는 881120-1068234 이다. 홍길동씨의 주민등록번호를 연월일 (YYYY-MM-DD) 부분과 그 뒤의 숫자 부분으로 나누어 출력해 보자.

```
1 pin = "881120-1068234"
2 yyyyymmdd = pin[:6]
3 num = pin[7:]
4 print(yyyyymmdd) # 881120 출력
5 print(num)       # 1068234 출력
```

Q4. 주민등록번호 인덱싱

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

```
1 pin = "881120-1068234"
2 print(pin[7]) # 1이면 남자, 2이면 여자
```

성별을 나타내는 숫자는 하이픈을 포함하여 8 번째 숫자이므로 8 번째 자리를 인덱싱한다.

Q5. 문자열 바꾸기

다음과 같은 문자열 a:b:c:d 가 있다. 문자열의 replace 함수를 사용하여 a#b#c#d 로 바꿔서 출력해 보자.

```
1 a = "a:b:c:d"
2 b = a.replace(":", "#")
3 print(b)
4 'a#b#c#d'
```

Q6. 리스트 역순 정렬하기

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

```
1 a = [1, 3, 5, 4, 2]
2 a.sort( )
3 a.reverse( )
4 print(a) # [5, 4, 3, 2, 1] 출력
```

리스트의 내장 함수인 sort 를 사용하여 리스트 값들을 먼저 정렬한 후 reverse 함수를 사용하여 순서를 뒤집는다.

Q7. 리스트를 문자열로 만들기

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

```
1 a = ['Life', 'is', 'too', 'short']
2 result = " ".join(a)
3 print(result)
```

a 리스트의 각 단어들을 한 문장으로 조립할 때 단어들 사이마다 공백을 넣어 주어야 한다. 1 개의 공백문자 (" ") 를 사용하여 join 한다.

Q8. 튜플 더하기

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

```
1 a = (1, 2, 3)
2 a = a + (4,)
3 print(a)          # (1, 2, 3, 4) 출력
```

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 유의하자.

다음 코드를 실행해 보면 a 의 고유 주소 값이 변경됨을 확인할 수 있다.

```
1 a = (1, 2, 3)
2 print(id(a))      # a의 고유 주소 값 출력
3 a = a + (4,)
4 print(a)
5 print(id(a))      # (4,) 값이 더해진 후 a의 고유 주소 값 출력
```

Q9. 딕셔너리의 키

다음과 같은 딕셔너리 a 가 있다. 다음 중 오류가 발생하는 경우는 어떤 경우인가? 그리고 그 이유를 설명해 보자.

3 번째 예를 실행하면 다음과 같은 오류가 발생한다.

```
1 >>> a[[1]] = 'python'
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   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.

Q10. 딕셔너리 POP

딕셔너리 a 에서 'B' 에 해당되는 값을 추출해 보자.

딕셔너리도 리스트와 마찬가지로 다음과 같이 pop 함수를 사용할 수 있다.

```
1 a = {'A':90, 'B':80, 'C':70}
2 result = a.pop('B')
3 print(a)           # {'A':90, 'C':70} 출력
4 print(result)      # 80 출력
```

'B' 키 값에 해당되는 값이 리턴되고 딕셔너리 a 에서는 그 값이 제거되는 것을 확인할 수 있다.

Q11. 리스트에서 중복 제거하기

a 리스트에서 중복된 숫자들을 제거해 보자.

```
1 a = [1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5]
2 aSet = set(a)      # a 리스트를 집합 자료형으로 변환
3 b = list(aSet)     # 집합 자료형을 리스트 자료형으로 다시 변환
4 print(b)           # [1,2,3,4,5] 출력
```

리스트 자료형이 집합 자료형으로 변환되면서 중복된 값들은 사라지게 된다. 이와 같은 성질을 사용하면 리스트 내에 중복된 값을 쉽게 제거할 수 있다.

Q12. 파이썬 변수

파이썬은 다음처럼 동일한 값에 여러 개의 변수를 선언할 수 있다. 다음과 같이 a, b 변수를 선언 한 후 a 의 두 번째 요소값을 변경하면 b 의 값은 어떻게 될까? 그리고 이런 결과가 나오는 이유에 대해서 설명해 보자.

[1, 4, 3] 이 출력된다. a 와 b 변수는 모두 동일한 [1, 2, 3] 이라는 리스트 객체를 가리키고 있기 때문이다.

10-2 풀이: 03 장 연습문제

Q1. 조건문의 참과 거짓

다음 코드의 결괏값은 무엇일까?

결괏값으로 `shirt` 가 출력된다.

1. 첫 번째 조건: “wife” 라는 단어는 `a` 문자열에 없으므로 거짓이다.
2. 두 번째 조건: “python” 이라는 단어는 `a` 문자열에 있지만 “you” 역시 `a` 문자열에 있으므로 거짓이다.
3. 세 번째 조건: “shirt” 라는 단어가 `a` 문자열에 없으므로 참이다.
4. 네 번째 조건: “need” 라는 단어가 `a` 문자열에 있으므로 참이다.

가장 먼저 참이 되는 것이 세 번째 조건이므로 “shirt” 가 출력된다.

Q2. 3 의 배수의 합

`while` 문을 사용해 1 부터 1000 까지의 자연수 중 3 의 배수의 합을 구해 보자.

3 의 배수는 3 으로 나누어 떨어지는 수이다. 이러한 아이디어를 기반으로 한 파이썬 코드는 다음과 같다.

```
1 result = 0
2 i = 1
3 while i <= 1000:
4     if i % 3 == 0: # 3으로 나누어 떨어지는 수는 3의 배수
5         result += i
6         i += 1
7
8 print(result) # 166833 출력
```

Q3. 별 표시하기

while 문을 사용하여 다음과 같이 별 (*) 을 표시하는 프로그램을 작성해 보자.

```
1 i = 0
2 while True:
3     i += 1 # while문 수행 시 1씩 증가
4     if i > 5: break # i 값이 5보다 크면 while문을 벗어난다.
5     print('*' * i) # i 값 개수 만큼 *를 출력한다.
```

while 문을 수행할 때마다 i 값을 증가시킨다. 별 모양을 5 번 출력해야 하므로 i 값이 5 보다 클 경우 while 문을 벗어나도록 한다. 별 모양을 i 값 만큼 출력하기 위해서 문자열 곱하기 기능을 사용한다.

Q4. 1 부터 100 까지 출력

for 문을 사용해 1 부터 100 까지의 숫자를 출력해 보자.

```
1 >>> for i in range(1, 101):
2 ...     print(i)
3 ...
4 1
5 2
6 3
7 4
8 5
9 6
10 7
11 8
12 9
13 10
14 ... 생략 ...
```

Q5. 평균점수 구하기

for 문을 사용하여 A 학급의 평균 점수를 구해 보자.

```
1 A = [70, 60, 55, 75, 95, 90, 80, 80, 85, 100]
2 total = 0
3
4 for score in A:
5     total += score # A학급의 점수를 모두 더한다.
6
7 average = total / len(A) # 평균을 구하기 위해 총 점수를 총 학생수로 나눈다.
8 print(average) # 평균 79.0이 출력된다.
```

for 문을 사용하여 먼저 총 점수를 구한 후 총 점수를 총 학생 수로 나누어 평균 점수를 구한다.

Q6. 리스트 컴프리헨션

리스트 중에서 홀수에만 2 를 곱하여 저장하는 다음과 같은 코드가 있다.

```
1 numbers = [1, 2, 3, 4, 5]
2
3 result = []
4 for n in numbers:
5     if n % 2 == 1:
6         result.append(n*2)
```

위 코드를 리스트 컴프리헨션 (list comprehension) 을 사용하여 표현해 보자.

리스트 컴프리헨션으로 표현하면 다음과 같다.

```
1 numbers = [1, 2, 3, 4, 5]
2 result = [n*2 for n in numbers if n%2==1]
3 print(result)
4 [2, 6, 10]
```

10-3 풀이: 04 장 연습문제

Q1. 홀수 짝수 판별하기

주어진 자연수가 홀수인지 짝수인지 판별해 주는 함수 (is_odd) 를 작성해 보자.

```
1 >>> def is_odd(number):
2 ...     if number % 2 == 1: # 2로 나누었을 때 나머지가 1이면 홀수이다.
3 ...         return True
4 ...     else:
5 ...         return False
6 ...
7 >>> is_odd(3)
8 True
9 >>> is_odd(4)
10 False
```

람다와 조건부 표현식을 사용하면 다음과 같이 간단하게도 만들 수 있다.

```
1 >>> is_odd = lambda x: True if x % 2 == 1 else False
2 >>> is_odd(3)
3 True
```

Q2. 모든 입력의 평균 값 구하기

입력으로 들어오는 모든 수의 평균 값을 계산해 주는 함수를 작성해 보자.

```
1 >>> def avg_numbers(*args): # 입력 개수에 상관없이 사용하기 위해 *args를 사용
2 ...     result = 0
3 ...     for i in args:
4 ...         result += i
5 ...     return result / len(args)
6 ...
7 >>> avg_numbers(1, 2)
8 1.5
9 >>> avg_numbers(1,2,3,4,5)
10 3.0
```

Q3. 프로그램 오류 수정하기 1

3 과 6 을 입력했을 때 9 가 아닌 36 이라는 결괏값을 돌려주었다. 이 프로그램의 오류를 수정해 보자.

```
1 input1 = input("첫 번째 숫자를 입력하세요:")
2 input2 = input("두 번째 숫자를 입력하세요:")
3
4 total = int(input1) + int(input2)      # 입력은 항상 문자열이므로 숫자로 바꾸어
    주어야 한다.
5 print("두 수의 합은 %s 입니다" % total)
```

출력 결과는 다음과 같다.

```
1 첫 번째 숫자를 입력하세요:3
2 두 번째 숫자를 입력하세요:6
3 두 수의 합은 9 입니다
```

Q4. 출력 결과가 다른것은?

다음 중 출력 결과가 다른 것 한 개를 고르시오.

```
1 >>> print("you" "need" "python")
2 youneedpython
3 >>> print("you"+"need"+"python")
4 youneedpython
5 >>> print("you", "need", "python")    # 콤마가 있는 경우 공백이 삽입되어 더해진다.
6 you need python
7 >>> print("".join(["you", "need", "python"]))
8 youneedpython
```

Q5. 프로그램 오류 수정하기 2

이 프로그램은 우리가 예상한 “Life is too short” 라는 문장을 출력하지 않는다. 우리가 예상한 값을 출력할 수 있도록 프로그램을 수정해 보자.

문제의 예와 같이 파일을 닫지 않은 상태에서 다시 열면 파일에 저장한 데이터를 읽을 수 없다. 따라서 열린 파일 객체를 close 로 닫아준 후 다시 열어서 파일의 내용을 읽어야 한다.

```
1 f1 = open("test.txt", 'w')
2 f1.write("Life is too short!")
3 f1.close() # 열린 파일 객체를 닫는다.
4
5 f2 = open("test.txt", 'r')
```

```
6 print(f2.read())
7 f2.close()
```

또는 다음과 같이 `close` 를 명시적으로 할 필요가 없는 `with` 구문을 사용한다.

```
1 with open("test.txt", 'w') as f1:
2     f1.write("Life is too short! ")
3 with open("test.txt", 'r') as f2:
4     print(f2.read())
```

Q6. 사용자 입력 저장하기

사용자의 입력을 파일 (`test.txt`) 에 저장하는 프로그램을 작성해 보자.

기존 내용을 유지하고 새로운 내용을 덧붙이기 위해서 다음과 같이 ‘a’ 모드를 사용해야 한다.

```
1 user_input = input("저장할 내용을 입력하세요:")
2 f = open('test.txt', 'a') # 내용을 추가하기 위해서 'a'를 사용
3 f.write(user_input)
4 f.write("\n")             # 입력된 내용을 줄 단위로 구분하기 위해 줄 바꿈 문
                             자 삽입
5 f.close()
```

Q7. 파일의 문자열 바꾸기

다음과 같은 내용을 지닌 파일 `test.txt` 가 있다. 이 파일의 내용 중 “java” 라는 문자열을 “python” 으로 바꾸어서 저장해 보자.

파일을 모두 읽은 후에 문자열의 `replace` 함수를 사용하여 `java` 라는 문자열을 `python` 으로 변경한 다음 저장한다.

```
1 f = open('test.txt', 'r')
2 body = f.read()
3 f.close()
4
5 body = body.replace('java', 'python')
6
7 f = open('test.txt', 'w')
8 f.write(body)
9 f.close()
```

Q8. 입력값을 모두 더하여 출력하기

다음과 같이 실행할 때 입력값을 모두 더하여 출력하는 스크립트 (C:\doit\myargv.py) 를 작성해 보자.

다음처럼 sys 모듈의 argv 를 사용하여 명령 행 입력값 모두를 차례로 더해 준다.

```
1 import sys
2
3 numbers = sys.argv[1:] # 파일 이름을 제외한 명령 행의 모든 입력
4
5 result = 0
6 for number in numbers:
7     result += int(number)
8 print(result)
```


10-4 풀이: 05 장 연습문제

Q1. UpgradeCalculator

위 클래스를 상속하는 UpgradeCalculator 를 만들고 값을 뺄 수 있는 minus 메서드를 추가해 보자.

다음과 같이 Calculator 클래스를 상속하는 UpgradeCalculator 클래스를 만들고 minus 메서드를 추가한다.

```
1 class UpgradeCalculator(Calculator):
2     def minus(self, val):
3         self.value -= val
```

Q2. MaxLimitCalculator

객체변수 value 가 100 이상의 값은 가질 수 없도록 제한하는 MaxLimitCalculator 클래스를 만들어 보자.

Calculator 클래스를 상속하고 add 메서드를 오버라이딩하여 다음과 같은 클래스를 만든다.

```
1 class MaxLimitCalculator(Calculator):
2     def add(self, val):
3         self.value += val
4         if self.value > 100:
5             self.value = 100
```

Q3. 참과 거짓 예측하기

다음 결과를 예측해 보자.

하나.

```
1 >>> all([1, 2, abs(-3)-3])
2 False
```

`abs(-3)`은 -3의 절댓값이므로 3이 되어 `all([1, 2, 0])`이 되고, 리스트의 요소값중 0이 있기 때문에 `all` 내장 함수의 결과는 `False`가 된다.

들.

```
1 >>> chr(ord('a')) == 'a'
2 True
```

`ord('a')`의 결과는 97이 되어 `chr(97)`로 치환된다. `chr(97)`의 결과는 다시 'a'가 되므로 `'a' == 'a'`가 되어 `True`를 돌려준다.

Q4. 음수 제거하기

`filter`와 `lambda`를 사용하여 리스트 `[1, -2, 3, -5, 8, -3]`에서 음수를 모두 제거해 보자.

음수를 제거하기 위한 `filter`의 함수로 `lambda` 함수를 다음과 같이 만들어 실행한다.

```
1 >>> list(filter(lambda x:x>0, [1, -2, 3, -5, 8, -3]))
2 [1, 3, 8]
```

Q5. 16 진수를 10 진수로 변경하기

'0xea'라는 16진수 문자열을 10진수로 변경해 보자.

`int` 내장 함수를 다음과 같이 실행한다.

```
1 >>> int('0xea', 16)
2 234
```

Q6. 리스트 항목마다 3 곱하여 리턴하기

`map`과 `lambda`를 사용하여 `[1, 2, 3, 4]`라는 리스트의 각 요소값에 3이 곱해진 리스트 `[3, 6, 9, 12]`를 만들어 보자.

입력에 항상 3을 곱하여 돌려 주는 `lambda` 함수를 다음과 같이 만들고 `map`과 조합하여 실행한다.

```
1 >>> list(map(lambda x:x*3, [1,2,3,4]))
2 [3, 6, 9, 12]
```

Q7. 최댓값과 최솟값의 합

다음 리스트의 최댓값과 최솟값의 합을 구해 보자.

리스트의 최댓값은 `max`, 최솟값은 `min` 내장 함수를 사용하여 다음과 같이 구한다.

```
1 >>> a = [-8, 2, 7, 5, -3, 5, 0, 1]
2 >>> max(a) + min(a)
3 -1
```

Q8. 소수점 반올림하기

5.666666666666667 을 소수점 4 자리까지만 반올림하여 표시해 보자.

`round` 내장 함수를 사용하면 다음과 같이 반올림하여 소수점 4 자리까지 표시할 수 있다.

```
1 >>> round(17/3, 4)
2 5.6667
```

Q9. 디렉터리 이동하고 파일 목록 출력하기

`os` 모듈을 사용하여 다음과 같이 동작하도록 코드를 작성해 보자.

다음처럼 `os` 모듈의 `chdir` 을 사용하여 `C:\doit` 이라는 디렉터리로 이동한다.

```
1 >>> import os
2 >>> os.chdir("c:/doit")
```

그리고 다음처럼 `os` 모듈의 `popen` 을 사용하여 시스템 명령어인 `dir` 을 수행한다.

```
1 >>> result = os.popen("dir")
```

`popen` 의 결과를 출력하기 위해 다음과 같이 수행한다.

```
1 >>> print(result.read())
2 ...
3 abc.txt
4 bidusource.html
5 ...
```

Q10. 파일 확장자가.py 인 파일만 찾기

glob 모듈을 사용하여 C:\doit 디렉터리의 파일 중 확장자가.py 인 파일만 출력하는 프로그램을 작성해 보자.

다음과 같이 glob 모듈을 사용한다.

```
1 >>> import glob
2 >>> glob.glob("c:/doit/*.py")
3 ['c:/doit/doit01.py', 'c:/doit/test.py']
```

Q11. 날짜 표시하기

time 모듈을 사용하여 현재 날짜와 시간을 다음과 같은 형식으로 출력해 보자.

time 모듈의 strftime 을 사용하여 다음과 같이 작성한다.

```
1 >>> import time
2 >>> time.strftime("%Y/%m/%d %H:%M:%S") # %Y:년, %m:월, %d:일, %H:시, %M:분, %S:초
3 '2018/04/05 10:56:27'
```

Q12. 로또 번호 생성하기

random 모듈을 사용하여 로또 번호 (1~45 사이의 숫자 6 개) 를 생성해 보자.

random 모듈의 randint 를 사용하여 다음과 같이 작성한다.

```
1 import random
2
3 result = []
4 while len(result) < 6:
5     num = random.randint(1, 45) # 1부터 45까지의 난수 발생
6     if num not in result:
7         result.append(num)
8
9 print(result)
```

Q13. 누나는 영철이보다 며칠 더 먼저 태어났을까?

영철이 누나의 생일은 1995 년 11 월 20 일이고 영철이의 생일은 1998 년 10 월 6 일이다. 영철이 누나는 영철이보다 며칠 더 먼저 태어났을까?

```
1 >>> import datetime
2 >>> sister = datetime.date(1995, 11, 20)
3 >>> me = datetime.date(1998, 10, 6)
4 >>> (me-sister).days
5 1051
```

누가가 영철이보다 1051 일 먼저 태어났다.

Q14. 기록순으로 정렬하기

다음은 1 학년 3 반 학생들의 100m 달리기 기록이다. 기록순으로 data 를 정렬해 보자.

```
1 import operator
2
3 data = [('윤 서 현', 15.25),
4         ('김 예 지', 13.31),
5         ('박 예 원', 15.34),
6         ('송 순 자', 15.57),
7         ('김 시 우', 15.48),
8         ('배 속 자', 17.9),
9         ('전 정 웅', 13.39),
10        ('김 혜 진', 16.63),
11        ('최 보 람', 17.14),
12        ('한 지 영', 14.83),
13        ('이 성 호', 17.7),
14        ('김 옥 순', 16.71),
15        ('황 민 지', 17.65),
16        ('김 영 철', 16.7),
17        ('주 병 철', 15.67),
18        ('박 상 현', 14.16),
19        ('김 영 순', 14.81),
20        ('오 지 아', 15.13),
21        ('윤 지 은', 16.93),
22        ('문 재 호', 16.39)]
23
24 data = sorted(data, key=operator.itemgetter(1))
25 for d in data:
26     print(d)
```

operator.itemgetter 모듈을 사용하여 기록순으로 정렬했다. operator.itemgetter(1) 은 (“이름”, “기록”) 으로 구성된 튜플 데이터의 2 번째 항목인 “기록” 을 의미한다. 위 코드를 실행하면 기록순으로 정렬된 데이터 확인할 수 있다.

Q15. 청소 당번 2 명 뽑기

다음 4 명의 학생 중 청소 당번 2 명을 뽑을 수 있는 경우의 수를 모두 나열하시오.

```
1 ['나지혜', '성성민', '윤지현', '김정숙']
```

```
1 import itertools
2
3 students = ['나지혜', '성성민', '윤지현', '김정숙']
4 result = itertools.combinations(students, 2)
5 print(list(result))
```

itertools.combinations 모듈을 사용하여 4 명중 2 명을 뽑을 수 있는 경우의 수를 구하여 출력한다. (조합)

Q16. 문자열 나열하기

“abcd” 문자열을 나열하는 경우의 수를 모두 출력하시오.

```
1 import itertools
2
3 a = "abcd"
4 result = itertools.permutations(a, 4)
5 for r in result:
6     print(''.join(r))
```

itertools.permutations 모듈을 사용하여 문자열 “abcd”의 문자 각각을 순열로 만들면 된다. 이 때 리턴되는 순열의 항목(r)은 ('a', 'b', 'c', 'd')와 같은 튜플이므로 ''.join(r)처럼 묶어서 출력했다.

Q17. 5 명에게 할일 부여하기

5 명을 무작위로 섞어 차례대로 할일 ['청소', '빨래', '설거지']을 부여하고 나머지 2 명에게는 “휴식”을 부여하는 프로그램을 작성하시오.

```
1 import random
2 import itertools
3
4 people = ['김승현', '김진호', '강춘자', '이예준', '김현주']
5 duty = ['청소', '빨래', '설거지']
6
7 random.shuffle(people) # 무작위로 섞는다.
8 result = itertools.zip_longest(people, duty, fillvalue='휴식')
9 for r in result:
10     print(r)
```

5 명을 무작위로 섞기 위해 `random.shuffle` 모듈을 사용했다. 그리고 사람들과 할일을 차례로 묶고 나머지 2 명에게는 “휴식” 을 부여하기 위해 `itertools.zip_longest` 모듈을 사용했다.

Q18. 벽에 타일 붙이기

가로 길이는 200cm 이고 세로 길이는 80cm 인 벽이 있다. 이 벽에 가능한 한 큰 정사각형 모양의 타일을 붙이려고 한다. 이때 붙이려는 타일 한 선의 길이와, 붙이는데 필요한 타일의 개수를 구하시오.

```
1 import math
2
3 width = 200
4 height = 80
5
6 square_size = math.gcd(200, 80)
7 print("타일 한 선의 길이:{}".format(square_size))
8
9 width_count = width/square_size
10 height_count = height/square_size
11
12 print("필요한 타일의 갯수:{}".format(int(width_count * height_count)))
```

```
1 타일 한 선의 길이:40
2 필요한 타일의 갯수:10
```

200 과 80 의 최대 공약수를 구하면 벽에 붙일 수 있는 가능한 한 가장 큰 정사각형 타일의 길이를 구할 수 있다. 최대 공약수로 구한 타일 한 선의 길이는 40 이다. 타일 한 선의 길이를 알면 가로로 붙여야 할 타일 갯수와 세로로 붙여야 할 타일의 갯수를 알 수 있으므로 두 개의 값을 곱하여 최종적으로 필요한 타일의 갯수를 구할 수 있다. 필요한 타일의 갯수는 총 10 개이다.

10-5 풀이: 09 장 종합문제

Q1. 문자열 바꾸기

```
1 >>> a = "a:b:c:d"
2 >>> b = a.split(":")
3 >>> b
4 ['a', 'b', 'c', 'd']
5 >>> c = "#".join(b)
6 >>> c
7 'a#b#c#d'
```

Q2. 딕셔너리 값 추출하기

딕셔너리의 `get` 함수를 사용하면 해당 `key` 가 없을 경우에는 두 번째 매개변수로 전달된 `default` 값을 대신 돌려준다.

```
1 >>> a = {'A':90, 'B':80}
2 >>> a.get('C', 70)
3 70
```

위 예에서는 'C'에 해당되는 `key`가 없기 때문에 디폴트 값으로 전달된 70을 돌려주었다.

Q3. 리스트의 더하기와 `extend` 함수

리스트 `a`에 `+` 기호를 사용하는 경우에 대해서 먼저 살펴보자.

```
1 >>> a = [1, 2, 3]
2 >>> id(a)
3 4302429640
```

`id` 함수는 입력으로 받은 리스트 `a`의 주소 값을 돌려 준다. 현재 `a`라는 리스트는 4302429640이라는 주소에 저장되어 있다.


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

리스트 `a` 에 `+` 기호를 사용하여 `[4, 5]` 라는 리스트를 더해 보았다. 그리고 다시 다음과 같이 리스트 `a` 의 주소 값을 확인해 보자.

```
1 >>> id(a)
2 4302472072
```

이전에 리스트 `a` 가 저장되어 있던 주소와 다른 값을 돌려주는 것을 확인할 수 있다. 주소 값이 다르기 때문에 `+` 를 사용하면 리스트 `a` 의 값이 변하는 것이 아니라 두 리스트가 더해진 새로운 리스트가 반환된다는 것을 확인할 수 있다.

이번에는 `extend` 함수를 사용해 보자.

```
1 >>> a = [1, 2, 3]
2 >>> id(a)
3 4302429640
```

리스트 `a` 를 생성하고 그 주소 값을 출력해 보았다.

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

그리고 리스트 `a` 에 `extend` 를 사용하여 `[4, 5]` 라는 리스트를 더해 주었다. 그리고 다시 다음과 같이 리스트 `a` 의 주소 값을 확인해 보도록 하자.

```
1 >>> id(a)
2 4302429640
```

`+` 기호를 사용하여 더한 경우와는 달리 주소 값이 변하지 않고 그대로 유지되는 것을 확인할 수 있다.

Q4. 리스트 총합 구하기

```
1 A = [20, 55, 67, 82, 45, 33, 90, 87, 100, 25]
2
3 result = 0
4 while A:                # A 리스트에 값이 있는 동안
5     mark = A.pop()       # A 리스트의 가장 마지막 항목을 하나씩 뽑아냄
6     if mark >= 50:        # 50점 이상의 점수만 더함
7         result += mark
8
9 print(result)            # 481 출력
```

Q5. 피보나치 함수

피보나치 수열은 다음과 같은 순서로 곱값을 반환한다.

1. fib(0) → 0 반환
2. fib(1) → 1 반환
3. fib(2) → fib(0) + fib(1) → 0 + 1 → 1 반환
4. fib(3) → fib(1) + fib(2) → 1 + 1 → 2 반환
5. fib(4) → fib(2) + fib(3) → 1 + 2 → 3 반환
6. ...

n 이 0 일 때는 0 을 반환, 1 일 때는 1 을 반환한다. n 이 2 이상일 경우에는 이전의 두 값을 더하여 반환한다.

재귀 호출을 사용하면 피보나치 함수를 다음과 같이 간단하게 작성할 수 있다.

```

1 def fib(n):
2     if n == 0 : return 0          # n이 0일 때는 0을 반환
3     if n == 1 : return 1          # n이 1일 때는 1을 반환
4     return fib(n-2) + fib(n-1)    # n이 2 이상일 때는 그 이전의 두 값을 더하여
                                   반환
5
6 for i in range(10):
7     print(fib(i))

```

0 부터 9 까지의 피보나치 수열의 곱값을 출력하여 그 값을 확인해 보았다.

Q6. 숫자의 총합 구하기

```

1 user_input = input("숫자를 입력하세요: ")
2 numbers = user_input.split(",")
3 total = 0
4 for n in numbers:
5     total += int(n)    # 입력은 문자열이므로 숫자로 변환해야 한다.
6 print(total)

```

수행결과

```

1 숫자를 입력하세요: 65,45,2,3,45,8
2 168

```

Q7. 한 줄 구구단

```
1 user_input = input("구구단을 출력할 숫자를 입력하세요 (2~9):")
2 dan = int(user_input)      # 입력 문자열을 숫자로 변환
3 for i in range(1, 10):
4     print(i*dan, end= ' ') # 한 줄로 출력하기 위해 줄 바꿈 문자 대신 공백 문자
                             #를 마지막에 출력한다.
```

Q8. 파일을 읽어 역순으로 저장하기

파일 객체의 `readlines` 를 사용하여 모든 라인을 읽은 후에 `reversed` 를 사용하여 역순으로 정렬한 다음 다시 파일에 저장한다.

```
1 f = open('abc.txt', 'r')
2 lines = f.readlines()    # 모든 라인을 읽음
3 f.close()
4
5 lines.reverse()          # 읽은 라인을 역순으로 정렬
6
7 f = open('abc.txt', 'w')
8 for line in lines:
9     line = line.strip()   # 포함되어 있는 줄 바꿈 문자 제거
10    f.write(line)
11    f.write('\n')         # 줄 바꿈 문자 삽입
12 f.close()
```

Q9. 평균 값 구하기

```
1 f = open("sample.txt")
2 lines = f.readlines() # sample.txt를 줄 단위로 모두 읽는다.
3 f.close()
4
5 total = 0
6 for line in lines:
7     score = int(line) # 줄에 적힌 점수를 숫자형으로 변환한다.
8     total += score
9 average = total / len(lines)
10
11 f = open("result.txt", "w")
12 f.write(str(average))
13 f.close()
```

`sample.txt` 의 점수를 모두 읽기 위해 파일을 열고 `readlines` 를 사용하여 각 줄의 점수 값을 모두 읽어 들여 총 점수를 구한다. 총 점수를 `sample.txt` 파일의 라인 (Line) 수로 나누어 평균 값을 구한 후 그 결과를 `result.txt` 파일에 쓴다. 숫자 값은 `result.txt` 파일에 바로 쓸 수 없으므로 `str` 함수를 사용하여 문자열로 변경한 후 파일에 쓴다.

Q10. 사칙연산 계산기

```
1 class Calculator:
2     def __init__(self, numberList):
3         self.numberList = numberList
4
5     def sum(self):
6         result = 0
7         for num in self.numberList:
8             result += num
9         return result
10
11    def avg(self):
12        total = self.sum()
13        return total / len(self.numberList)
14
15 cal1 = Calculator([1,2,3,4,5])
16 print(cal1.sum())
17 print(cal1.avg())
18
19 cal2 = Calculator([6,7,8,9,10])
20 print(cal2.sum())
21 print(cal2.avg())
```

Q11. 모듈을 사용하는 방법

파이썬 셸에서 mymod.py 모듈을 인식하기 위해서는 다음과 같은 3 가지 방법이 있을 수 있다.

1) sys 모듈 사용하기

다음과 같이 `sys.path` 에 `C:\doit` 이라는 디렉터리를 추가하면 `C:\doit` 이라는 디렉터리에 있는 mymod 모듈을 사용할 수 있게 된다.

```
1 >>> import sys
2 >>> sys.path.append("c:/doit")
3 >>> import mymod
```

2) PYTHONPATH 환경 변수 사용하기

다음처럼 `PYTHONPATH` 환경 변수에 `C:\doit` 디렉터리를 지정하면 `C:\doit` 디렉터리에 있는 mymod 모듈을 사용할 수 있게 된다.

```
1 C:\Users\home>set PYTHONPATH=c:\doit
2 C:\Users\home>python
3 >>> import mymod
```

3) 현재 디렉터리 사용하기

파이썬 셸을 `mymod.py` 가 있는 위치로 이동하여 실행해도 `mymod` 모듈을 사용할 수 있게 된다. 왜냐하면 `sys.path` 에는 현재 디렉터리인 `.` 이 항상 포함되어 있기 때문이다.

```
1 C:\Users\home>cd c:\doit
2 C:\doit>python
3 >>> import mymod
```

Q12. 오류와 예외 처리

7 이 출력된다.

1. `result` 의 초기값은 0 이다.
2. `try` 문 안의 `[1, 2, 3][3]` 이라는 문장 수행 시 `IndexError`가 발생하여 `except IndexError:` 구문으로 이동하게 되어 `result` 에 3 이 더해져 3 이 된다.
3. 최종적으로 `finally` 구문이 실행되어 `result` 에 4 가 더해져 7 이 된다.
4. `print(result)` 가 수행되어 `result` 의 최종 값인 7 이 출력된다.

Q13. DashInsert

다음 프로그램의 주석문을 참고하자.

```
1 data = "4546793"
2 numbers = list(map(int, data)) # 숫자 문자열을 숫자 리스트로 변경
3 result = []
4
5 for i, num in enumerate(numbers):
6     result.append(str(num))
7     if i < len(numbers)-1:
8         is_odd = num % 2 == 1 # 다음 수가 홀수
9         is_next_odd = numbers[i+1] % 2 == 1 # 현재 수가 홀수
10        if is_odd and is_next_odd: # 다음 수가 홀수
11            result.append("-")
12        elif not is_odd and not is_next_odd: # 연속 짝수
13            result.append("*")
14
15 print("".join(result))
```

Q14. 문자열 압축하기

먼저 입력 문자열의 문자를 확인하여 동일한 문자가 들어올 경우에는 해당 문자의 숫자 값을 증가시킨다. 만약 다른 문자가 들어올 경우에는 해당 문자의 숫자 값을 1 로 초기화하는 방법을 사용하여 작성한 코드

이다.

상세한 설명은 다음 프로그램의 주석문을 참고하자.

```

1 def compress_string(s):
2     _c = "" # s 문자열중 현재 진행중인 문자를 임시저장하기 위한 변수
3     cnt = 0 # 해당 문자가 몇 번 반복했는지 알 수 있는 카운트 변수
4     result = "" # 이 함수의 최종 리턴 문자열 (예: a3b2c5a1)
5     for c in s: # 입력받은 문자열 s에서 문자 하나씩 c에 대입
6         if c != _c: # 현재 진행중인 문자와 c가 같지 않은 경우, 즉 새로운 문자의
            시작
7             _c = c # 현재 진행중인 문자와 같지 않으므로 현재진행문자는 c로 대
                입
8             if cnt: result += str(cnt) # 새로운 문자이므로 결과 문자열에 이전
                문자의 카운트(있을 경우에만)에 해당하는 값을 더해 주어야 함
9             result += c # 새로운 문자이므로 결과문자열에 새로운 문자를 더함
10            cnt = 1 # 새로운 문자이므로 카운트는 1로 초기화
11        else: # 현재 진행중인 문자와 c가 같기 때문에 카운트가 증가됨
12            cnt += 1
13        if cnt: result += str(cnt) # for loop를 벗어났을때 이전 문자의 카운트는 최
            종적으로 한번 더해주어야 함
14    return result # 최종문자열 리턴
15
16 print(compress_string("aaabbcccccca")) # a3b2c6a1 출력

```

Q15. Duplicate Numbers

```

1 def chk_dup_numbers(s):
2     result = []
3     for num in s:
4         if num not in result:
5             result.append(num)
6         else:
7             return False
8     return len(result) == 10
9
10 print(chk_dup_numbers("0123456789")) # True 리턴
11 print(chk_dup_numbers("01234")) # False 리턴
12 print(chk_dup_numbers("01234567890")) # False 리턴
13 print(chk_dup_numbers("6789012345")) # True 리턴
14 print(chk_dup_numbers("012322456789")) # False 리턴

```

리스트 자료형을 사용하여 중복된 값이 있는지 먼저 조사한다. 중복된 값이 있을 경우는 **False**를 리턴한다. 최종적으로 중복된 값이 없을 경우 0~9까지의 숫자가 모두 사용되었는지 판단하기 위해 입력 문자열의 숫자 값을 저장한 리스트 자료형의 총 개수가 10 인지를 조사하여 10 일 경우는 **True**, 아닐 경우는 **False**를 리턴한다.

Q16. 모스 부호 해독

```

1  dic = {
2      '.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E', '...-': 'F',
3      '---': 'G', '....': 'H', '...': 'I', '----': 'J', '-.-': 'K', '-.-.-': 'L',
4      '---': 'M', '-.': 'N', '---': 'O', '---': 'P', '---': 'Q', '-.-': 'R',
5      '...': 'S', '-': 'T', '...-': 'U', '...-': 'V', '-.-': 'W', '-.-.-': 'X',
6      '.-.-': 'Y', '-.-.-': 'Z'
7  }
8
9  def morse(src):
10     result = []
11     for word in src.split(" "):
12         for char in word.split(" "):
13             result.append(dic[char])
14         result.append(" ")
15     return "".join(result)
16
17
18 print(morse('... . . . . -.- . . . -.- . . . -.- . -.-'))

```

모스 부호 규칙 표를 딕셔너리로 작성한 후 입력에 해당되는 모스 부호 문자열을 먼저 단어 (공백 문자 2 개) 로 구분한다. 그 후 단어 (공백 문자 1 개) 를 문자로 구분하여 해당 모스 부호 값을 딕셔너리에서 찾아서 그 결괏값을 구한다.

Q17. 정규식 - 기초 메타 문자

보기 중 이 조건에 해당되는 것은 B 이다.

다음은 위 문제의 정규식 매치 결과를 확인해 보는 파이썬 코드이다.

```

1  import re
2
3  p = re.compile("a[.]{3,}b")
4
5  print(p.match("acccb"))      # None
6  print(p.match("a...b"))      # 매치 객체 출력
7  print(p.match("aaab"))      # None
8  print(p.match("a.cccb"))     # None

```

Q18. 정규식 - 문자열 검색

정규식 `[a-z]+`은 소문자로 이루어진 단어를 뜻하므로 5 `python` 문자열에서 `python`과 매치될 것이다. 따라서 `python` 문자열의 인덱스 범위는 `m.start()`에서 `m.end()`까지이므로 10 이 출력된다.

```
1 import re
2
3 p = re.compile('[a-z]+')
4 m = p.search("5 python")
5 print(m.start() + m.end()) # 10 출력
```

5 python 문자열에서 n 이 7 번째 문자여서 m.end()의 값이 7 이 나올것 같지만 다음과 같은 슬라이싱 규칙을 떠올려보면 왜 8 이 나오는지 쉽게 이해될 것이다.

```
1 >>> a = "5 python"
2 >>> a[2:8]
3 'python'
4 >>> a[2:7]
5 'pytho'
```

Q19. 정규식 - 그룹핑

전화번호 패턴은 다음과 같이 작성할 수 있다.

```
1 pat = re.compile("\d{3}[-]\d{4}[-]\d{4}")
```

이 전화번호 패턴 중 뒤의 숫자 4 개를 변경할 것이므로 필요한 앞부분을 다음과 같이 그룹핑한다.

```
1 pat = re.compile("(\d{3}[-]\d{4})[-]\d{4}")
```

컴파일된 객체 pat 에 sub 함수를 사용하여 다음과 같이 문자열을 변경한다.

```
1 import re
2
3 s = ""
4 park 010-9999-9988
5 kim 010-9909-7789
6 lee 010-8789-7768
7 ""
8
9 pat = re.compile("(\d{3}[-]\d{4})[-]\d{4}")
10 result = pat.sub("\g<1>-####", s)
11
12 print(result)
```

Q20. 정규식 - 전방 탐색

.com 과 .net 에 해당되는 이메일 주소만을 매치하기 위해서 이메일 주소의 도메인 부분에 다음과 같은 긍정형 전방탐색 패턴을 적용한다.


```
1 pat = re.compile(".*[@].*[(?=com$|net$).*$")
```

다음은 위 패턴을 적용한 파이썬 코드이다.

```
1 import re
2
3 pat = re.compile(".*[@].*[(?=com$|net$).*$")
4
5 print(pat.match("pahkey@gmail.com"))
6 print(pat.match("kim@daum.net"))
7 print(pat.match("lee@myhome.co.kr"))
```

11 장 부록 - 챗 GPT 활용하기

챗 GPT 를 활용하여 파이썬을 공부하는 방법에 대해서 알아보자.

챗 GPT(Generative Pre-trained Transformer) 는 OpenAI 에서 개발한 딥러닝 언어모델로 사용자가 질문을 하면 대답을 해주는 인공지능 챗봇이다. 챗 GPT 는 <https://chat.openai.com/chat> 사이트에 접속하여 가입한 후 사용할수 있다.

파이썬을 공부할때 챗 GPT 를 훌륭한 도우미로 사용할수 있다. 점프 투 파이썬을 공부하다가 궁금한 점이 생겼는데 책이 알려주지 않는다면 챗 GPT 에 물어 그 궁금증을 해결할수 있다. 또한 작성한 코드에 오류가 발생했는데 도저히 그 이유를 모르겠다면 역시 챗 GPT 에 물어서 해결할수 있다.

오류의 원인 물어보기

다음은 실행시 오류가 발생하는 예제이다.

```
1 money = True
2 if money:
3     print("택시를")
4     print("타고")
5     print("가라")
```

이렇듯 오류가 발생하는 코드를 작성했을때 그 원인을 파악하기 힘들다면 다음과 같이 챗 GPT 에 물어보자.




다음 코드에서 오류가 발생하는 이유를 알려줘:

```
money = True
if money:
    print("택시를")
    print("타고")
    print("가라")
```

질문과 코드를 같이 입력할 경우 위와 같이 질문 마지막에 `:`을 입력하고 `Shift+Enter` 키를 입력하여 다음줄로 바꾼다음 코드를 붙여넣기 하면 된다.

그러면 챗 GPT 는 다음과 같이 답변을 한다.

 위 코드에서는 ``IndentationError: unexpected indent`` 라는 오류가 발생합니다. 이는 코드 블록의 들여쓰기(indentation)가 잘못되었기 때문입니다.

``if`` 문 다음에 오는 ``print("택시를")`` 부분은 들여쓰기가 맞게 되어 있지만, 그 아래의 ``print("타고")`` 부분은 추가로 들여쓰기가 되어 있습니다. 이로 인해 파이썬은 ``print("타고")`` 부분을 ``if`` 문의 코드 블록에서 벗어나게 되어 문법적으로 올바르지 않다는 오류가 발생합니다.

따라서, 코드 블록의 들여쓰기를 올바르게 수정하면 다음과 같이 작성할 수 있습니다:

```
python
money = True
if money:
    print("택시를")
    print("타고")
    print("가라")
```

위 코드는 ``money`` 변수가 ``True`` 이므로, ``if`` 문이 실행되고 ``print("택시를")``, ``print("타고")``, ``print("가라")`` 가 순서대로 실행됩니다.

챗 GPT 는 나날이 성장하고 있다. 이 책을 작성하는 시점의 챗 GPT 의 버전은 `ChatGPT Mar 14` 이다. 따라서 더 높은 버전의 챗 GPT 를 사용한다면 더 좋은 답변을 기대할수 있을 것이다.

작성한 코드에 문제가 있는지 물어보기

다음은 예상과는 달리 엉뚱한 결과를 출력하는 예제이다.

```
1 input1 = input("첫 번째 숫자를 입력하세요:")
2 input2 = input("두 번째 숫자를 입력하세요:")
3
4 total = input1 + input2
5 print("두 수의 합은 %s 입니다" % total)
```

작성한 코드에 문제가 있는지 확인하기 위해 다음과 같이 챗 GPT 에 물어보자.



다음 파이썬 코드에 오류가 있는지 확인해줘:

```
input1 = input("첫번째 숫자를 입력하세요:")
input2 = input("두번째 숫자를 입력하세요:")

total = input1 + input2
print("두 수의 합은 %s 입니다" % total)
```

그러면 챗 GPT 는 다음과 같이 답변한다.



이 코드에는 오류가 있습니다. input() 함수를 사용하여 사용자로부터 입력 받은 값은 기본적으로 문자열 (string) 형태로 저장됩니다. 따라서 입력된 값은 먼저 숫자형으로 변환해야 합니다.



아래는 수정된 코드입니다:

python

Copy code

```
input1 = int(input("첫번째 숫자를 입력하세요:"))
input2 = int(input("두번째 숫자를 입력하세요:"))

total = input1 + input2
print("두 수의 합은 %s 입니다" % total)
```

입력된 값을 정수형으로 변환하기 위해 `int()` 함수를 사용합니다. 이제 오류 없이 실행되며 두 숫자의 합을 정확하게 출력할 것입니다.

작성한 코드의 개선을 요청하기

다음은 파일에 내용을 쓰고 다시 그 내용을 읽어 출력하는 파이썬 코드이다.

```
1 f1 = open("test.txt", 'w')
2 f1.write("Life is too short!")
3 f1.close()
4
5 f2 = open("test.txt", 'r')
6 print(f2.read())
7 f2.close()
```

이 코드보다 더 좋은 품질의 코드를 작성할수 있을까? 궁금하다면 다음과 같이 챗 GPT 에 물어보자.





다음 파이썬 코드를 개선해줘:

```
f1 = open("test.txt", 'w')
f1.write("Life is too short!")
f1.close()

f2 = open("test.txt", 'r')
print(f2.read())
f2.close()
```

그러면 챗 GPT 는 다음과 같이 개선된 코드를 보여주고 그 이유에 대해서도 친절히 설명해 준다.




이 코드는 파일에 문자열을 쓰고, 다시 읽어와서 출력하는 기본적인 파일 입출력 코드입니다. 개선할 부분은 다음과 같습니다:  

1. 파일을 닫는 부분을 with 문으로 대체합니다.
2. 파일을 읽을 때는 readline(), readlines() 등 다른 메서드도 있지만, 이 경우에는 read() 메서드를 사용하면 됩니다.
3. 파일을 닫는 부분은 with 문을 사용하면 자동으로 처리되므로, 따로 close() 함수를 호출할 필요가 없습니다.

아래는 개선된 코드입니다:

python

 Copy code

```
with open("test.txt", 'w') as f1:
    f1.write("Life is too short!")

with open("test.txt", 'r') as f2:
    print(f2.read())
```

이 코드에서는 with 문을 사용하여 파일을 열고, 파일 처리가 끝나면 자동으로 닫습니다. 또한, 파일을 읽을 때는 read() 메서드를 사용합니다. 이렇게 개선하면 코드가 더 간결해지고, 파일 입출력 시 발생할 수 있는 예외를 더욱 안전하게 처리할 수 있습니다.

주석을 달아달라고 요청하기

다음은 점프 투 파이썬의 종합문제 중 15 번 문제의 풀이 코드이다.

```

1 def chk_dup_numbers(s):
2     result = []
3     for num in s:
4         if num not in result:
5             result.append(num)
6         else:
7             return False
8     return len(result) == 10

```

하지만 풀이를 보아도 잘 이해하기가 어렵다. 이해하기 어려운 코드를 보다 잘 이해하기 위해 챗 GPT 에 주석을 만들어 달라고 요청해 보자.



다음 파이썬 코드에 주석을 만들어줘:

```

def chk_dup_numbers(s):
    result = []
    for num in s:
        if num not in result:
            result.append(num)
        else:
            return False
    return len(result) == 10

```

그러면 챗 GPT 는 다음과 같이 주석이 추가된 코드를 보여주고 코드에 대해서도 친절히 설명해 준다.



아래는 주석이 추가된 코드입니다:



python

Copy code

```

def chk_dup_numbers(s):
    result = [] # 중복되지 않은 숫자를 저장할 리스트
    for num in s: # 문자열 s에서 숫자를 하나씩 가져옴
        if num not in result: # 현재 숫자가 리스트에 없으면
            result.append(num) # 리스트에 추가함
        else: # 현재 숫자가 리스트에 이미 있으면
            return False # 중복이 있으므로 False 반환
    return len(result) == 10 # 리스트의 길이가 10인지 여부를 반환

```

이 코드는 파라미터로 전달된 문자열 s에서 중복된 숫자가 있는지 여부를 검사하는 함수입니다. 함수는 중복된 숫자가 있으면 False를 반환하고, 중복된 숫자가 없으면 s에서 나타나는 모든 숫자를 포함하는 리스트를 생성합니다. 이 리스트의 길이가 10이면 모든 숫자가 한 번씩만 나타나므로 True를 반환합니다. 함수 내부에 사용된 변수와 조건문이 어떤 역할을 하는지 주석을 추가하여 코드를 이해하기 쉽도록 만들었습니다.

코드의 해석 요청하기

다음과 같이 정규식으로 작성된 파이썬 코드를 이해하는 것은 쉽지 않다.

```
1 import re
2
3 p = re.compile("a[.]{3,}b")
```

이해하기 어려운 코드가 있다면 다음과 같이 챗 GPT 에게 해석을 부탁해 보자.



파이썬으로 작성된 다음의 정규식을 해석해줘:

```
import re
```

```
p = re.compile("a[.]{3,}b")
```

그러면 챗 GPT 는 다음과 같이 코드를 잘 설명해준다.



해당 파이썬 코드에서 사용된 정규식 `a[.]{3,}b`은 다음과 같이 해석할 수 있습니다:



- `a` : 문자열이 `a`로 시작해야 함
- `[.]{3,}` : `.` 문자가 3개 이상 연속해서 나와야 함
- `b` : 문자열이 `b`로 끝나야 함

즉, 이 정규식은 `a`로 시작하고, 중간에 `.` 문자가 3개 이상 연속해서 나오고, 마지막으로 `b`로 끝나는 문자열에 대응합니다.

예를 들어, `a...b`, `a....b`, `a.....b` 등이 해당 정규식에 매칭됩니다.

여기서 `.`은 정규식에서 메타 문자(meta character)로 사용되는데, 이는 어떤 문자와도 대응될 수 있는 와일드카드 문자입니다. 따라서 `[.]`와 같이 대괄호 안에 `.`을 넣어주면, 문자 그대로 `.`을 의미하게 됩니다.

기능 구현 요청하기

만약 여러분이 파이썬으로 두 날짜의 차이를 구하는 기능을 구현해야 한다고 가정해 보자. 무작정 구현하기 보다는 이미 그러한 일을 도와주는 라이브러리가 있는지도 알고 싶을 것이다. 챗 GPT 에게 이럴때 사용하는 라이브러리가 있는지 물어보고 또 예제도 요청해 보자.



두 날짜의 차이를 구할수 있는 파이썬 라이브러리를 알려주고 예제도 보여줘

그러면 챗 GPT 는 다음과 같이 답변한다.



Python에서 두 날짜 사이의 차이를 구하는 데는 `datetime` 모듈을 사용할 수 있습니다. 이 모듈에는 `datetime` 클래스와 `timedelta` 클래스가 있습니다. `datetime` 클래스는 날짜와 시간을 나타내는 객체를 생성하고, `timedelta` 클래스는 두 날짜 또는 시간 사이의 차이를 나타내는 객체를 생성합니다.

아래는 `datetime` 모듈을 사용하여 두 날짜 간의 차이를 계산하는 예제 코드입니다:

python

Copy code

```
from datetime import datetime

date_str1 = '2022-03-20'
date_str2 = '2022-03-25'

date1 = datetime.strptime(date_str1, '%Y-%m-%d')
date2 = datetime.strptime(date_str2, '%Y-%m-%d')

delta = date2 - date1
print(f"두 날짜의 차이는 {delta.days}일 입니다.")
```

위 코드에서 `datetime.strptime()` 함수를 사용하여 문자열을 `datetime` 객체로 변환합니다. 이 때, `strptime()` 함수의 첫 번째 인자는 변환할 문자열이고, 두 번째 인자는 문자열의 형식을 지정하는 문자열입니다. `%Y`, `%m`, `%d` 등은 각각 연도, 월, 일을 나타내는 문자열 포맷입니다. `datetime` 객체를 만든 후에는 빼기 연산을 통해 두 날짜의 차이를 계산합니다. 마지막으로 `timedelta` 객체의 `days` 속성을 사용하여 날짜 차이를 일 단위로 출력합니다.

이상과 같이 챗 GPT 를 활용할수 있는 방법에 대해서 몇 가지 알아보았다. 이 외에도 챗 GPT 를 활용할수 있는 방법은 무궁무진할 것이다. 챗 GPT 를 여러분의 충실한 파이썬 도우미로 만들어 친하게 지내보자.

12 장 마치며

“점프 투 파이썬” 졸업을 축하한다.

함께 만드는 점프 투 파이썬

“점프 투 파이썬”은 함께 만들어가는 온라인 책이다. 여러분의 피드백으로 “점프 투 파이썬”은 성장해가고 있다. 책을 쓰에 있어서 가장 두려운 것은 잘못된 정보의 전달이다. 하지만 이런 두려움은 여러분의 댓글 참여로 인해 해소될 수 있으리라 생각한다. 댓글과 피드백을 통한 여러분의 참여와 도움을 기다린다.

» 댓글이나 피드백 링크는 페이지 하단에 있다.

도전과제

이제 여러분은 파이썬으로 무엇이든 만들 수 있는 준비가 되었다.

보다 능숙하게 파이썬 코드를 작성하려면 파이썬 라이브러리에 대해서 잘 알고 있어야 한다. 좀 더 깊게 파이썬을 공부하고자 한다면 필자가 집필한 “점프 투 파이썬 - 라이브러리 예제 편” 책을 추천한다.

- [점프 투 파이썬 - 라이브러리 예제 편](#)

요새 인기있는 딥러닝 분야나 웹 프로그래밍에 도전하는 것도 좋을 것이다. 만약 웹 프로그래밍에 관심이 있다면 필자가 집필한 “점프 투 장고”와 “점프 투 플라스크”를 추천한다.

- [점프 투 장고](#)
- [점프 투 플라스크](#)

그리고 외국서적이지만 “Python Essential Reference”와 “Python Cook Book”의 저자인 David Beazley의 공개 파이썬 강좌인 “Practical Python”을 추천한다.

- [Practical Python](#)

파이썬 질문과 답변

파이썬에 대한 질문은 언제든지 환영한다. 필자가 운영하는 파이보에 파이썬 질문을 작성하면 빠른 답변을 얻을 수 있을 것이다.

- 파이썬 질문과 답변 서비스 “파이보” - <https://pybo.kr>