일단계 파이썬 프로그래밍

국립목포대학교 전기제어공학과

박장현 저

1장 파이썬 개요

1.1 파이썬 소개

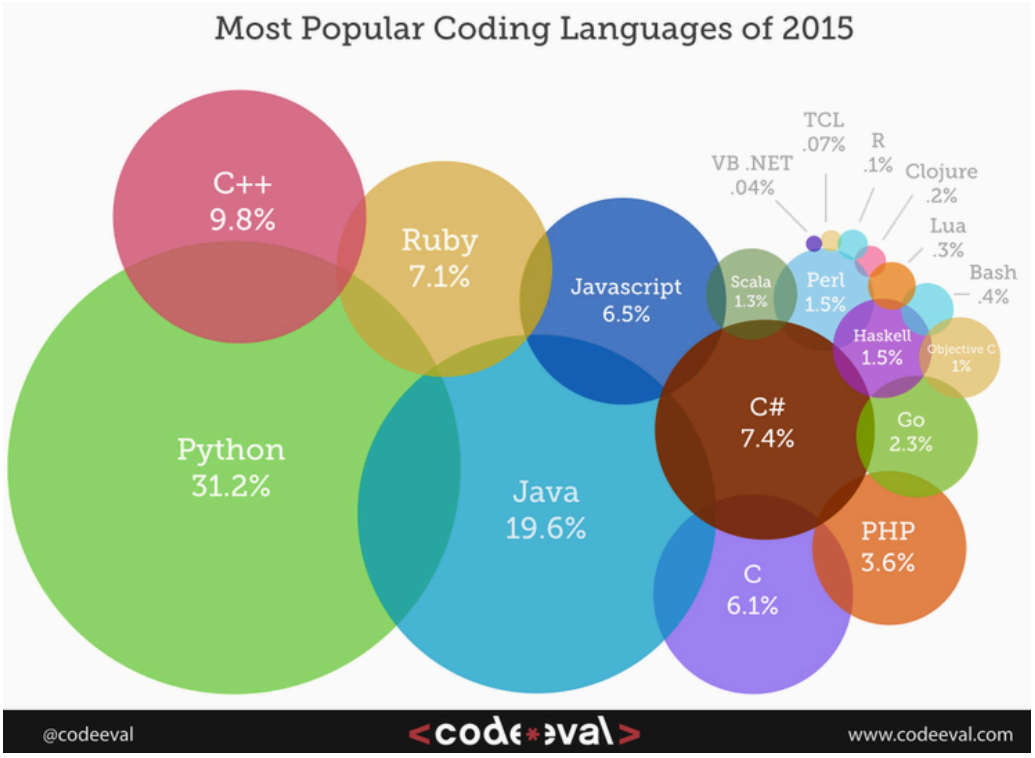
파이썬(python)은 인터프리터 언어로서 암스테르담의 귀도 반 로섬(Guido V. Rossum)에 의해서 1990년에 초기 버전이 만들어졌다. (2015년 1월 현재 3.4.x 버전) 파이썬이라는 이름은 본인이 좋아하는 "Monty Python's Flying Circus" 라는 코미디 쇼에서 따왔다고 한다. 파이썬의 사전적인 뜻은 큰 비단뱀인데 대부분의 파이썬 책 표지와 아이콘이 뱀 모양으로 그려져 있는 이유가 여기에 있다.

[그림 1] 파이썬 개발자, 비단뱀 그리고 파이썬 로고

파이썬 프로그램의 가장 큰 특징은 배우기 쉽고 직관적이라는 점이다. 인터프리터 언어이기 때문에 실행 결과를 즉시 확인해 볼 수 있으며 최근에는 실행 속도도 심지어 C/C++ 프로그램과 맞먹을 정도로 성능도 향상되고 있다. 또한 공동 작업과 유지 보수가 매우 쉽고 편하기 때문에 이미 다른 언어로 작성된 많은 프로그램과 모듈들이 파이썬으로 다시 재구성되고 있기도 하다. 국내에서도 그 가치를 인정받아 사용자 층이 더욱 넓어져 가고 있고, 파이썬을 이용한 프로그램을 개발하는 기업체들이 늘어가고 있는 추세이다.

현재 파이썬은 교육의 목적뿐만 아니라 실용적인 부분에서도 널리 사용되고 있는데 그 대표적인 예는 바로 구글(Google) 이다. 구글에서 만들어진 소프트웨어의 50%이상이 파이썬으로 만들어졌다고 한다. 이 외에도 유명한 것을 몇 가지 들어보면 Dropbox(파일 동기화 서비스), Django(파이썬 웹 프레임워크)등을 들 수 있다. 또한 빅데이터 분석이나

과학 계산 용도로도 활발히 활용되는 등 컴퓨터를 활용한 거의 모든 곳에 사용되고 있다고 해도 과언이 아닐 정도로 인기를 끌고 있다.



[그림 2] 가장 널리 사용되는 프로그래밍 언어들

파이썬의 특징을 정리하면 다음과 같다.

- 인터프리터(interpreter) 언어이다.(실행 결과를 바로 확인할 수 있다.)
- 간결하고 쉬운 문법으로 빠르게 학습할 수 있다.
- 강력한 성능을 가진다.
- 다양한 분야에 적용할 수 있는 라이브러리가 풍부하다.
- 개발 속도가 빠르다.
- 오픈 소스(open source)이며 무료이다.

최근에는 라즈베리파이(raspberry pi)나 비글본 블랙(beaglebone black) 같은 원보드 마이컴이 인기를 끌고 있는데 보통 운영체제로 리눅스를 채용한다. 이러한 시스템에서도 파이썬을 이용하여 전통적인 C/C++/JAVA로 개발하는 것보다 훨씬 더 쉽고 빠르게 응용 프로그램을 제작할 수 있다.

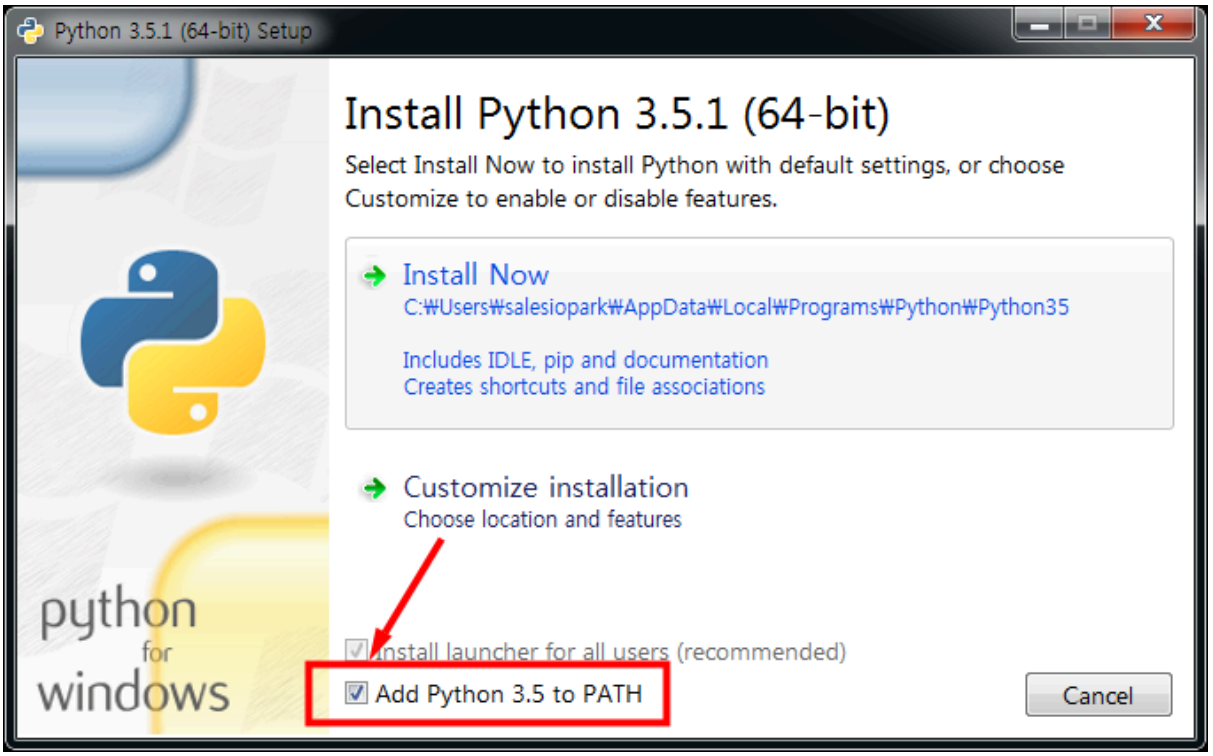
[그림 3] 라즈베리파이와 비글본블랙

현재 파이썬은 버전이 2.x 대와 3.x 대로 나뉘어 두 가지 버전이 같이 사용되고 있다는 점인데 초보 사용자가 선택하는데 문제가 생긴다. 특이하게도 3.x 버전의 문법이 2.x버전과는 달라서 100% 호환되지 않으므로 같은 언어로 작성한 프로그램인데도 불구하고 2.x 버전에서 잘 작동되는 것이 3.x 버전에서는 작동하지 않거나 반대의 경우도 발생한다.

본 교재에서는 3.x 버전의 문법을 기본으로 해서 2.x버전 과의 차이점에 대해서 필요할 때마다 설명하도록 하겠다.

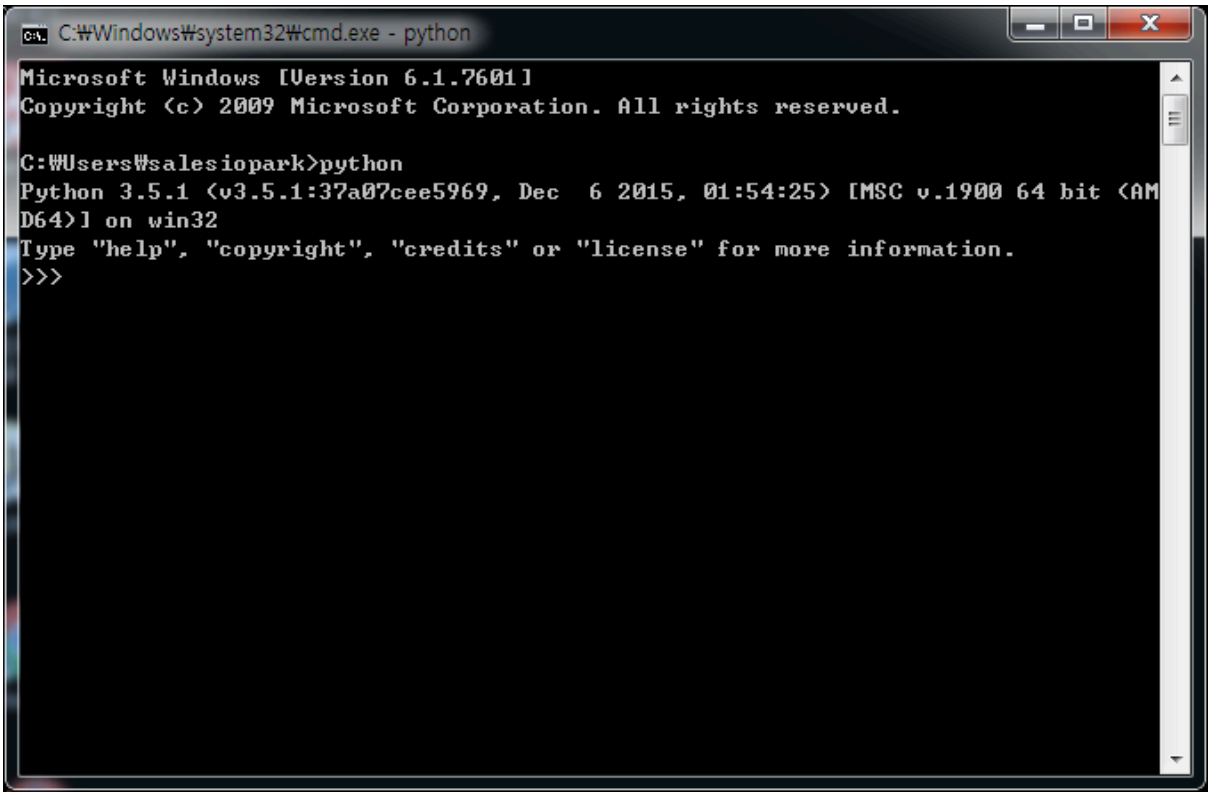
1.2 실행 환경

윈도우즈에서는 python.org 사이트에서 다운로드 받은 후 파이썬쉘을 실행시킬 수 있다. 단 path를 등록해 주어야 cmd 환경에서 실행을 할 수 있다. (다음 그림 참고)



[그림 1.2.1] 윈도우즈 설치 화면

설치가 끝나면 다음과 같이 cmd 환경에서 python 이라는 명령을 수행하면 파이썬쉘이 실행된다. 파이썬쉘에서 quit() 함수를 호출하면 종료된다.



[그림 1.2.2] cmd에서 파이썬 쉘을 실행한 화면

라즈비언의 경우에는 python2와 python3가 미리 설치되어 있으므로 별도로 설치과정이 필요치 않다.

1.3 식별자

변수명, 함수명, 클래스명 등으로 쓸 수 있는 식별자(identifier)를 만드는 방법은 다른 언어들과 거의 동일하다. 식별자를 만들 때 보통 사용되는 문자들은 다음과 같다.

- 영문자 대소문자 (a,b … z A B … Z)
- 숫자 (0 1 … 9)
- 언더바(_)

이러한 문자들을 조합하되 숫자로 시작하면 안된다. 그리고 대소문자를 다른 문자로 구별한다. 그리고 **python3**에서는 유니코드 문자를 식별자로 사용할 수 있다. 즉, 한글로도 변수명을 만들 수 있지만 권장되지는 않는다.

2장 기본 자료형과 연산자

프로그램은 결국 자료(data)를 처리하는 일을 주로 하게 된다. 여기에서는 파이썬의 내장 자료형의 종류 대해서 알아보자. 가장 중요하고 사용 빈도가 높은 파이썬 자료형은 다음과 같은 것들이 있다.

- 숫자 : 정수 (int), 실수 (float), 복소수 (complex)로 나뉜다.
- 문자열 (str)
- 리스트 (list)
- 튜플 (tuple)
- 딕셔너리(dictionary)
- 집합 (set)

이 장에서는 이들 각각에 대해서 자세히 알아보도록 하겠다.

2.1 숫자 자료형

파이썬의 숫자 데이터를 분류하면 다음과 같다.

[표 2.1.1] 숫자형의 분류

숫자형	개요
정수(int)	범위에 제한이 없다. 16진수, 8진수, 2진수로도 표현 가능하다. (파이썬2.x 버전에서는 큰 정수에 l/L 접미사를 붙여야 함.)
실수(float)	소수점 이나 e/E 가 포함된 숫자
복소수(complex)	허수부(j/J) 가 포함된 숫자

다른 언어들과는 달리 파이썬은 정수에 대해 범위 제한이 없으며 복소수까지 기본 자료형에 포함된다. 복소수는 숫자 뒤에 j 혹은 J 를 붙이면 된다. 파이썬쉘에서 다음 예를 하나씩 실행해 보고 결과를 확인해 보자.

```
>>> a=123
>>> b1=3.14
>>> b2=1e-10
>>> c=100000000000000000000000000000000 # 정수는 범위 제한이 없다.
>>> d1=1+2j
>>> d2=1j #허수를 표기할 때 1j 와 같이 1을 꼭 붙여야 한다.
>>> d3=-3-1j
>>> e=0x12a #16진수. 0x(제로 엑스) 혹은 0X 로 시작한다.
>>> f=0b1101101 #2진수. 0b(제로 비) 혹은 0B 로 시작한다.
>>> g=0o1234 #8진수. 0o (제로 오) 혹은 0O로 시작한다.
```

정수는 10진수/16진수/8진수/2진수로 표현할 수 있으며 실수형은 십진수 형식으로만 표현된다. 숫자에 소수점이나 지수를 표시하는 e/E가 있다면 실수이고 그렇지 않다면 정수로 분류된다. 또한 허수부가 포함되어 있다면 복소수형이다.

재미있는 것은 0j, -0j, +0.0j, -0.0j 도 내부적으로 complex 형으로 간주된다는 것이다. 숫자 상수나 변수의 형을 보려면 내장 함수인 type()을 이용한다.

```
>>> type(10)
>>> type(-1.2)
>>> a=3+4j; type(a)
>>> type(2+0j)
```

정수의 범위에 제한이 없다는 점도 특이한 점이다. 이러한 점에서 파이썬이 매우 큰 숫자를 다뤄야 하는 분야나(예를 들어서 천문학, 경제학) 데이터 분석에서 타 언어보다 유리하다는 점을 짐작할 수 있다. (**2.x**버전에서는 큰 정수를 표시하기 위해서는 정수 뒤에 **L/l** 접미어를 붙여야 한다. 그러나 3.x버전은 그런 제약이 없다.)

복소수는 실수부, 허수부, 켤레복소수를 구할 수 있는 속성(attribute)이 있는데 각각 real, imag, conjugate() 이다. 다음을 실습해 보자.

```
>>> c=11+22j
>>> c.real
>>> c.imag
>>> c.conjugate()
```

또한 내장 함수인 abs()를 이용하면 복소수의 크기를 구할 수 있다.

```
>>> abs(c)
```

십진수의 이진수 표현을 알고 싶다면 bin()이라는 내장함수를 사용하면 된다. 팔진수는 oct(), 십육진수는 hex()함수를 이용한다.

```
>>> a=1234
>>> bin(a)
'0b10011010010'
>>> oct(a)
'0o2322'
>>> hex(a)
'0x4d2'
```

단, 이 함수들은 정수형에 대해서만 동작하며 반환값이 문자열이라는 것에 유의해야 한다.

만약 일반적인 변수의 값을 확인하고 싶다면 파이썬쉘에서 변수명을 입력하고 엔터를 치면 변수의 값을 보여준다.

```
>>> a
>>> c, z # 여러 변수를 동시에 출력하고 싶다면 콤마(,)로 구분한다.
```

만약 ipython쉘을 사용하고 있다면 현재 정의된 모든 변수값을 보고 싶다면 'who ' 명령어를 이용하면 된다.

2.2 산술 연산자

파이썬의 산술 연산자는 다음과 같은 것들이 있다.

[표 1] 파이썬의 산술 연산자들

연산자	기능	비고/용례
+	덧셈	11+22, a+12, a+b

-	뺄셈	11-22j - 33, a-12, a-b
*	곱셈	11*22, (33+44j)*(55-66j), a*b
/	나눗셈	결과는 실수형이다. (2.x에서는 int / int의 결과는 int)
//	자리내림 나눗셈	나눗셈의 결과값에서 소수점 아래는 버리고 정수 부분만 취한다.
**	거듭제곱	2**10, (1+2j)**20
%	나머지	3%4, -10%3, 12.345%0.4

여기서 나눗셈의 경우 버전에 따라서 결과가 다르다. 2.x버전에서는 정수간 나눗셈의 결과는 정수였다. 즉 1/2는 0, 2/3은 1이다. 하지만 3.x버전에서는 정수간 나눗셈의 결과는 나누어 떨어지는 경우라도 무조건 실수형이 된다. 즉, 1/2는 0.2, 6/3은 2.0이 된다. 다음 결과를 확인해 보자.

```
>>> 4/5
>>> a,b = 11,5 #a에 11, b에 5를 대입한다.
>>> b/a
```

2.x버전에서는 정수끼리의 나눗셈의 결과는 정수라는 것에 유의해야 한다.

연산자 //는 두 피연산자가 모두 정수일 경우 결과값이 실수이면 소수점 아래는 버린다.

```
>>> 9//2 # 결과는 4 (정수)
>>> 9//2.0 # 결과는 4.5(실수)
```

연산자 %는 나눗셈 수행 후 정수몫을 구하고 난 나머지를 구하는 연산자이다. 다음을 확인해보라.

```
>>> 3%2
1
>>> 11.5%0.2
0.09999999999999937
>>> 11.5%0.3
0.10000000000000042
```

실수 연산의 경우는 결과값이 정확하지 않을 수도 있다.

연산자 **는 거듭제곱 연산자이다. 다음 연산의 예를 보자.

```
>>> 2**10
1024
>>> a,c = 3,4+5j
>>> c**a
(-236+115j)
>>> a**c
(57.00434317675227-57.54567628403595j)
```

위에서 복소수의 거듭제곱 c**a 는 (4+5j)*(4+5j)*(4+5j) 의 결과값을 보여준다. 파이썬에서는 복소수에 대한 산술 연산도 기본적으로 지원하므로 쉽게 수행할 수 있음을 알 수 있다.

---- (이하 생략 가능) ----------

그렇다면 a**c는 왜 저런 결과가 나왔을까? 다음과 같이 오일러(Euler)공식을 이용하면 된다.

$$\begin{aligned} 3^{4+j5} &= e^{(ln3)(4+j5)} \\ &= e^{4(ln3)} e^{j5(ln3)} \\ &= e^{4(ln3)}(\cos(5\ln 3) + j\sin(5\ln 3)) \end{aligned}$$

이 식의 결과와 위 예제의 결과는 동일하다.

2.3 문자열

파이썬에서 문자열은 큰따옴표("...") 혹은 작은 따옴표 ('...')로 묶인 문자들의 집합이다.

```
“Hello world.”
‘Mokpo National Univ.’
“12.3”
```

위의 예는 모두 문자열을 나타낸다. 12.3은 숫자이고 이것을 따옴표로 묶은 “12.3”은 문자열이다. 즉, 문자’1’, 문자’2’, 문자’.’, 문자’3’ 의 배열(array)이다. 파이썬에서 문자형이라는 자료형은 별도로 없다. 다른 언어(**C/C++, JAVA, C#** 등)에서는 보통 작은 따옴표는 한 문자를 표현할 때 사용되지만 파이썬에서는 문자열을 입력할 때 사용한다는 것에 유의하자.

문자열을 입력하는 방법을 왜 두 가지로 마련해 놓았을까? 작은 따옴표가 포함된 문자열 혹은 큰따옴표가 포함된 문자열을 쉽게 입력할 수 있기 때문이다.

```
‘He said “hi”. ’
“I’m your father.”
```

파이썬 문자열에도 C언어의 printf()함수에서 사용하였던 특수 문자를 사용할 수 있다. 특수문자는 역슬래시(\ 는 여기서는 escape cahracter 라고 한다.)로 시작하며 특수한 용도로 사용된다. 다음 표에 주로 쓰이는 특수 문자를 정리하였다.

[표 1] 파이썬의 문자열에 쓰이는 특수문자

문자	설명
\n	줄바꿈
\t	수평 탭(tab)
\\	‘\’문자 자체를 의미
\’	작은따옴표 문자
\”	큰따옴표 문자

예를 들면 다음과 같다.

```
“He sais \”How are you?\””
“Hi.\nHello.”
‘He\’s finished.’
```

단 문자열의 중간에 줄바꿈 기호 '\n'이 들어가면 가독성이 떨어지므로 파이썬에서는 줄바꿈 기호를 타이핑하지 않고 그대로 입력할 수 있는 방법으로 “”” … “”” 과 ‘’’ … ’’’ 를 제공한다.

```
>>> s = “””HI
      Hello”””
```

라고 입력하고 그 결과를 확인해 보자.

```
>>> s
```

문자열 s에는 사용자가 엔터키로 입력한 줄바꿈 기호가 '\n'으로 자동으로 치환되었음을 알 수 있다.

만약 문자열 안의 '\' 문자를 이스케이프 문자로 간주하지 않고 단순 문자로 사용하고 싶다면 '\\'와 같이 입력해도 되지만 한 문자열 안에 이런 경우가 많이 발생한다면 문자열 앞에 r을 붙이면 된다. 문자열 앞에 r이 붙으면 그 문자열 안의 모든 '\'는 단순 문자로 처리되며 결과 문자열에서는 '\'문자가 자동으로 '\\'로 변환된다.

```
>>> r'Hi.\nMy name:\tjhp'
'Hi.\\nMy name:\\tjhp'
```

이 방법은 특히 파일 경로나 정규식(regular express)을 다룰 때 유용하다.

2.3.1 문자열의 인덱싱과 슬라이싱

파이썬3의 문자열은 내부적으로 (유니코드) 문자의 배열로 취급된다. 예를 들어보자.

```
>>> s=“Hello World”
```

문자열	H	e	l	l	o		w	o	r	l	d
인덱스 (기본 방향)	0	1	2	3	4	5	6	7	8	9	10
인덱스 (역방향)	-11				-7	-6	-5	-4	-3	-2	-1

문자열에 포함된 각각의 문자에 매겨진 이 번호를 인덱스(index)라고 한다. 이 예에서 문자열의 길이는 11이고 인덱스는 0부터 시작한다. 인덱스가 1부터 시작하지 않고 0부터 시작함에 주의해야 한다. (다른 프로그래밍 언어에서도 배열의 인덱싱은 보통 0부터 시작한다.)

```
>>> s[0]
H
>>> s[6]
w
>>> s[-1]
d
```

마지막의 s[-1]과 같이 음수는 뒤에서부터 세는 것이다. 따라서 뒤에서 첫 번째 문자인 'd'가 된다.

```
>>> s[-2]
l
>>> s[-6]

```

만일

```
>>>  a="python is the best."
```

라는 문자열에서 첫 단어를 뽑아내고 싶다면 아래와 같이 한다.

```
>>> b=a[0:6]
```

인덱스 '0:6' 이 뜻하는 것은 '0부터 5까지' 이다. 끝 번호 6은 포함하지 않는다는 것에 주의해야 한다. 이렇게 콜론(:)을 이용하여 연속적인 인덱스를 지정하는 것을 슬라이싱(slicing)이라고 한다. 문자열의 마지막까지 지정하려면 끝 번호를 생략하면 된다.

```
>>> c = a[7:] # 'is the best.' 가 c에 저장된다.
```

반대로 시작 번호가 생략되면 문자열의 처음부터 선택된다.

```
>>> d = a[:8] # 'python is' 가 d에 저장된다.
```

그리고 시작 번호와 끝 번호가 모두 생략된다면, 즉 e=a[:] 이라고 하면 문자열 전체가 선택이 된다. 즉, e에는 a 문자열 전체가 저장된다.

슬라이싱에서도 인덱싱과 마찬가지로 음수를 사용할 수 있다.

```
>>> f = a[:-5]
```

결과를 확인해 보기 바란다. 이 경우에도 끝 번호는 포함되지 않으므로 첫 문자부터 -6번 문자까지 뽑아져서 f에 저장된다. 슬라이싱을 정리하면 다음과 같다.

- s[m:n] 은 s[m] 부터 s[n-1] 까지의 부분 문자열이다.

예를 들어서 만약 문자열 h를 5번째 문자를 기준으로 둘로 나눠서 hl, hr에 정하고 싶다면 다음과 같이 하면 될 것이다.

```
>>> hl = h[:5]
>>> hr = h[5:]
```

이러한 기능을 이용해서 문자열 자체를 바꿀 수는 없다는 것에 주의하자. 즉, 다음과 같이 문자열의 일부분을 바꾸는 것은 불가능한다.

```
>>>a[0] = 'x' #불가능하다
```

이는 문자열은 한 번 내용이 정해지면 내용을 읽는 수는 있지만 변경될 수는 없는 자료형이기 때문이다. (이러한 자료형을 immutable 하다고 한다.)

2.3.2 문자열의 덧셈과 곱셈

파이썬에서는 문자열끼리 더하거나 곱할 수 있다. 문자열끼리 연결하는데 덧셈 연산자를 사용하는 것은 다른 언어에서도 흔하지만 곱셈은 독특하다. 두 개의 문자열을 더하면 문자열이 하나로 합해진 새로운 문자열이 만들어진다.

```
>>> “My name is “+”Salesio.”
>>> q = “ipython”
>>> r = “ is useful.”
>>> s = q+r
```

이 예제에서 원래의 문자열 q와 r은 더하기 연산으로 변경되지 않는다. 앞에서도 밝혔듯이 문자열은 한 번 내용이 정해지면 그 다음에는 변경할 수 없다. (이러한 자료형을 immutable 자료형이라고 한다.) 다만 연산의 결과로 새로운 문자열이 생성되는 것이다. 만약

```
>>>s = ’Life is long.’
```

에서 long이라는 단어를 short로 바꾸고 싶다면 슬라이싱을 이용해서 다음과 같이 할 수 있다.

```
>>> s = s[:8] + ’short.’
```

이 경우 원래 문자열이 바뀌는 것이 아니라 기존의 문자열은 모두 삭제되고 새로 생성된 문자열이 다시 변수 s에 저장되는 것이다. (사실 변수 s 는 문자열에 대한 참조가 저장된다. 이 경우 참조, 즉 주소가 새로 생성된 문자열의 그것으로 바뀌는 것이다.)

유사하게 문자열에 정수를 곱하면 정수만큼 반복되는 새로운 문자열을 생성한다.

```
>>> “blah “*5
blah blah blah blah blah
```

```
>>> '='*50
==================================================
```

위에서 두 번째 예는 화면에 출력할 때 간단히 구분선을 만드는 용도로 사용된다.

2.3.3 문자열의 %포멧터

문자열 중간에 어떤 변수의 내용을 출력하고 싶다면 print()함수를 이용하면 된다.

```
>>> a=3+4j
>>> b='hi'
>>> print('a is',a,'and b is',b) #변수 앞뒤에 공백문자를 자동으로 넣어준다.
a is (3+4j) and b is hi
```

또는 변수의 내용으로 아예 치환한 문자열을 만들고 싶다면 str() 내장함수를 이용하는 방법도 있다.

```
>>> a=3+4j
>>> b='hi'
>>> 'a is '+str(a)+' b is '+ b
a is (3+4j) and b is hi
```

더 간단하고 일반적인 방법은 문자열 포맷팅 기능을 이용하는 것이다. C 언어에서 printf() 문을 이용하여 변수의 내용을 출력하는 유사한 방법을 파이썬에서도 지원한다. 이것을 문자열의 formatting이라고 하는데 예를 들어서 정수값을 출력하고 싶다면 다음과 같이 하면 된다.

```
>>> a=11
>>> "a equals %d."%a
```

이와 같이 문자열 내부의 %d는 문자열에 바로 뒤따르는 '%변수'의 변수값으로 바꾸어 준다. 두 개 이상을 출력하려면 다음과 같이 괄호로 묶어주면 된다. (괄호로 묶인 자료 집합을 튜플이라고 하며 뒤에서 자세하게 설명할 것이다.)

```
>>> a=11
>>> b=-22
>>> "vars are %d and %d."%(a,b)
'vars are 11 and -22.'
```

실수를 출력하려면 %f, 문자열을 출력하려면 %s를 사용하면 된다.

[표 1] 문자열 formatter 의 종류

문자열 포맷팅	기능
%d, %x, %o	십진수, 16진수, 8진수(복소수는 출력이 안 됨)
%f %.숫자f	실수를 출력 (복소수는 출력이 안 됨.) 표시할 소수점 아래 자리수를 명시한다.
%s	문자열 출력
%%	'%' 문자 자체를 출력

문자열을 출력하는 예를 들어보면 다음과 같다.

```
>>> ss = 'short'
>>> print('Life is %s.'%ss)
        Life is short.
```

아래 예와 같이 실수의 경우 표시할 소수점 아래 자리수를 지정해줄 수 있다.

```
>>> rn = 1.2345678
>>> 'f is about %.4f'%rn
'f is about 1.2346'
```

소수점 다섯째 자리에서 반올림하여 소수점 넷째 자리까지 표시했음을 알 수 있다.

한 가지 짚고 넘어갈 것은 %d와 %f 는 복소수는 출력하지 못한다. 복소수는 %s를 이용하여 출력하면 된다.

```
>>> c1=11+22.3j
>>> print('c1=%s'%c1)
c1=(11+22.3j)
```

재미있는 것은 %s를 이용하면 숫자나 문자열을 가리지 않고 모두 출력해준다는 것이다. 이는 숫자도 내부적으로 자동으로 문자열로 변환되기 때문이다.

```
>>> n=1024
>>> print('n equals %s.'%n)
n equals 1024.
```

따라서 숫자를 출력하든 문자를 출력하든 %s를 사용하여도 전혀 문제가 없다.

2.3.4 문자열의 format()함수

파이썬에서 문자열을 포맷팅할 때 문자열 객체의 멤버 함수인 format()을 이용할 수도 있다. (이 경우 C#의 문자열 포멧팅과 유사한 방법을 사용할 수 있다.)

```
>>> age = 44
>>> print("I'm {0} years old.'.format(age))
I'm 44 years old.
```

문자열 안의 '{번호}' 가 format() 멤버함수의 인자의 내용으로 치환이 됨을 알 수 있다. 변수의 종류는 가리지 않으며 format()함수의 첫 번째 인자가 0번, 두 번째 인자가 1번...순으로 지정된다.

여러 개의 변수를 출력할 수도 있다.

```
>>> name='Jang-Hyun Park'
>>> age=44
>>> print("My name is {0} and {1} years old.".format(name,age))
My name is Jang-Hyun Park and 44 years old.
```

여기에서 문자열 내부의 '{0}'은 name 변수로, '{1}'은 age변수의 내용으로 치환된다.

실수의 소수점 아래 자리수를 제한하려면 다음과 같이 하면 된다.

```
>>> rn = 1.2345678
>>> print("rn is about {0:.4f}.".format(rn))
rn is about 1.2346.
```

포맷팅을 하는 두 가지 방법 모두 장단점이 있으므로 상황에 따라 편한 쪽을 택해서 사용하면 된다. 예를 들어서 동일한 변수를 여러 번 출력할 때는 여기에서 소개한 방법이 더 유리하다.

```
>>> blah = 'blah'
>>> print("%s %s %s %s"%(blah, blah, blah, blah))
>>> print("{0} {0} {0} {0}".format(blah))
```

이와 같이 동일한 변수를 여러 번 출력할 경우 세 번째 줄과 같이 문자열의 format() 필드를 이용하면 두 번째 줄의 %-formatter보다 더 간략하게 사용할 수 있다.

2.3.5 문자열의 내장 함수

파이썬 문자열의 멤버 함수를 정리하면 다음 표와 같다.

문자열 함수	기능
format()	변수의 내용을 표시하기 위한 포맷팅을 수행한다.
lower(), casefold() upper()	대문자를 소문자로 바꾼다. 소문자를 대문자로 바꾼다.

swapcase() title() capitalize()	대문자는 소문자로, 소문자는 대문자로 바꾼다. 모든 단어의 첫 문자만 대문자로 나머지는 소문자로 바꾼다. 문자열의 첫 글자만 대문자로 나머지는 소문자로 바꾼다.
islower() isupper()	모든 문자가 소문자이면 True 반환 모든 문자가 대문자이면 True 반환
count(str)	str이 포함된 개수를 센다.
find(str) index(str)	str의 첫 위치를 알아낸다. (없다면 -1 반환) str의 첫 위치를 알아낸다. (없다면 예외 발생)
join(str)	str을 구성하는 각 문자 사이에 원 문자열을 끼워 넣는다.
lsrtip() rstrip() strip() center(n[,str]) ljust(n[,str])) rjust(n[,str]))	좌측 공백을 지운다. 우측 공백을 지운다. 양쪽의 공백을 지운다. 크기 n의 문자열의 중앙에 원 문자열을 정렬한다. 크기 n의 문자열의 왼쪽에 원 문자열을 정렬한다.. 크기 n의 문자열의 오른쪽에 원 문자열을 정렬한다.. (두 번째 인수가 없다면 공백 문자를, 있다면 그 문자로 채운다.)
replace(str1, str2)	원 문자열 안의 str1을 str2로 바꾼다.
split() split(sep)	공백문자를 기준으로 나누어서 리스트에 저장한다. sep(문자열)을 기준으로 나누어서 리스트에 저장한다.
isalnum() isalpha() isidentifier()	모든 문자가 알파벳 혹은 숫자(alphanumeric)이면 True 반환. 모든 문자가 알파벳(alphabet)이면 True 반환. 문자열이 식별자의 조건에 맞다면 True 반환.
isdecimal() isdigit() isnumeric()	십진 정수이면 True 반환. 모든 문자가 0에서 9가지의 숫자이면 True 반환. 모든 문자가 0에서 9가지의 숫자이면 True 반환. (세 함수의 차이점이 명확하지 않음.)

위의 함수들은 모두 문자열 혹은 문자열 변수에 바로 이어서 점(.)을 찍은 다음 호출할 수 있다. 예를 들면 다음과 같다.

```
>>> s = "HELLO"
>>> s2 = s.lower() #문자열 s를 전부 소문자로 만들어서 s2에 저장
>>> "Hi. My name is jhp.".count('jhp') #문자열에서 문자열 'jhp'의 개수를 센다.
```

join()함수의 경우 str1.join(str2) 라고 입력하면 str2를 구성하는 각 문자의 사이에 str1을 끼워 넣어서 새로운 문자열을 생성한다. (반대로 짐작하기 쉬우므로 순서에 유의해야 한다.)

```
>>>','.join('abc')
'a,b,c'
>>>'_and_'.join('jhpark')
'j_and_h_and_p_and_a_and_r_and_k'
```

2.4 진리값

파이썬에서 참, 거짓은 True 와 False 라는 키워드를 사용한다. (첫 글자가 대문자라는 것에 유의해야 한다.) 진리값은 자체로 변수의 값으로 사용될 수 있으며 논리 연산의 결과를 표현하는데 사용된다.

한 가지 알아둘 것은 파이썬은 False 뿐만 아니라 None, 0 (0.0, 0j, 0.0j, 0+0j 도 마찬가지), 빈 문자열(“” 혹은 ‘’), 빈 집합 ((), {}, [])도 False 로 간주한다는 것이다.

재밌는 것은 진리값과 숫자형의 연산에서 True는 1, False는 0으로 간주되어 계산이 수행된다는 것이다.

```
>>> a=True
>>> a+1
2
>>> a*1
1
```

2.5 None

파이썬에는 None이라는 값이 있는데 이것은 ‘비어 있음’ 혹은 ‘아무것도 없음’을 표시하는 자료이다.

```
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> a = None
>>> print(a)
None

>>> b='Hi'
>>> b=None
>>> print(b)
None
```

어떤 변수가 undefined되어 있는 경우와 값이 None인 경우는 구별해야 한다. 전자는 변수가 아예 존재하지 않는 것이고 후자는 변수가 존재하되 값이 None인 경우이다.

2.6 비교 연산자

파이썬에는 다음과 같은 비교연산자들이 있다.

[표 1] 파이썬의 비교 연산자들

연산자	의미
x < y	x보다 y가 크면 True
x <= y	x보다 y가 크거나 같다면 True
x > y	x보다 y가 작으면 True
x >= y	x보다 y가 작거나 같다면 True
x == y	x와 y가 같다면 True
x != y	x와 y가 다르다면 True
x is y	동일한 객체인지를 판별 (x와 y가 같은 참조인지 판별)
x is not y	is 의 부정형 (x와 y가 다른 참조인지 판별)

연산자 '=='와 '!='은 객체의 내용(content)를 직접 비교하여 참/거짓을 판별한다. 객체의 형(type)이 다르면 두 객체는 항상 다르다. 단, 숫자형인 경우 int, float, complex 간에도

실제로 같은 숫자라면 같다. 즉, 1 (int형), 1.0 (float형), 1+0j (complex형), 1.0+0j (complex형), 1.0+0.0j (complex형) 는 모두 같은 숫자로 판별된다.

```
>>> sa = 'hello everybody'
>>> sb = 'hello everybody'
>>> sa == sb
True
```

반면 연산자 **is** 는 두 객체가 동일한 참조(주소)를 가지고 있는 지를 판별하는 것이다. 만약 id(x)와 id(y)가 같다면 x is y 는 True값을 가진다.

```
>>> sa='hello everybody'
>>> sb='hello everybody'
>>> sa is sb
False

>>> x=[1,2]
>>> y=[1,2]
>>> x==y
True
>>> x is y
False
>>> z=x
>>> z is x
True
```

2.7 논리 연산자

논리 연산자는 조건식들끼리 묶는 역할을 한다.

[표 2] 파이썬의 논리 연산자들

연산자	기능
x and y	x와 y 둘 다 True 일 때 True 그 외는 False
x or y	x와 y 둘 다 False 일 때 False 그 외는 True
not x	x가 True 일 때 False

이와 같이 파이썬의 논리 연산자는 다른 언어들과는 달리 자연어(and, or, not)을 사용한다.

2.8 리스트

리스트는 여러 개의 객체를 하나로 묶는 데이터 형이다. (엄밀히 말하면 mutable sequence 형임) 기본적으로 대괄호 **[..]**를 이용하여 묶고자 하는 데이터들을 콤마**(,)**로 구별하여 나열한다. 보통은 같은 형의 데이터를 묶어서 많이 사용하지만 리스트의 요소는 서로 다른 데이터 형일 수 있다.

```
>>> a = [11, 22, 33] # 리스트를 생성하여 a 에 저장한다.
>>> b = ['life', 'is', 'short'] # 문자열의 리스트
>>> c = [True, 'hi', 33.4, 2-3j]
>>> d = [] # empty list를 생성하여 d에 저장
```

이 예제와 같이 리스트의 요소는 어떠한 자료형이라도 가능하다. 리스트 안에 리스트가 오는 것도 얼마든지 가능하다.

```
>>> e = [ [11,22], [33, 44, 55] ]
```

이것을 중첩된 리스트라고 한다.

2.8.1 리스트의 인덱싱과 슬라이싱

리스트도 문자열과 동일한 인덱싱과 슬라이싱이 가능하다.

2.8.2 리스트의 덧셈과 곱셈

리스트는 변형 가능(mutable)하기 때문에 mutable sequence에서 공통으로 사용 가능한 연산(덧셈과 곱셈)을 적용할 수 있다. 이 부분은 문자열의 인덱싱/슬라이싱 포스트와 문자열의 덧셈과 곱셈를 참조하기 바란다.

예를 들어 두 리스트를 더하면 두 리스트의 요소들을 병합하여 새로운 리스트가 만들어진다.

```
>>> [11,22]+[33,44,55]
[11,22,33,44,55]
```

2.8.3 list comprehension

리스트를 만드는 또 다른 방법은 list comprehension 을 이용하는 것이다. 예를 들어서 제곱수의 리스트틀 만든다고 가정하자. 다음과 같이 for 문을 이용할 수 도 있을 것이다.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

하지만 이 방법은 변수 x가 생성되고 for 문이 종료된 후에도 변수 x 가 메모리에 남아 있게 된다. 좀 더 깔끔한 방법은 다음과 같이 map()과 익명 함수를 이용하는 것이다.

```
>>> squares = list(map(lambda x: x**2, range(10)))
```

이것과 완전히 동일한 방법이 바로 다음과같은 list comprehension 이고 훨씬 더 간결하고 가독성이 높다.

```
>>> squares = [x**2 for x in range(10)]
```

일반적으로 다음과 같이 구성된다.

```
[ 표현식 for문 (for 문 | if 문)* ]
```

예를 들어서 두 리스트 요소 중 다른 것들로만 조합하는 리스트는 다음과 같다.

```
>>> [(x,y) for x in [1,2,3] for y in [3,1,4] if x!=y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

이것은 다음 프로그램과 동일한 동작을 수행한다.

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

두 가지 경우에서 for 문과 if 문의 순서가 같다는 것을 알 수 있다.
표현식이 튜플일 경우 반드시 괄호로 묶어야 한다.

```
>>> [(x,x**2) for x in range(6)]
      [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

만약 다음과 같이 괄호를 생략하면 에러가 발생한다.

```
>>> [ x,x**2  for x in range(6)] # 에러 발생
```

여기서 소개한 내용이 리스트를 생성하는 기본적인 방법들이다.

2.8.4 리스트의 내장 메소드를

리스트의 내장 메소드들(method)들은 다음 표와 같다.

[표 1] 리스트의 메소드들

리스트 함수	동작
append(x)	x를 리스트의 마지막 요소로 추가한다.
extend(list)	list의 요소로 원 리스트를 확장한다.

insert(i, x)	x를 i번째 위치로 끼워 넣는다.
remove(x)	x와 같은 첫 번째 요소를 삭제한다.
pop() pop(i)	마지막 요소를 삭제하고 반환한다. i번째 요소를 삭제하고 반환한다.
clear()	모든 요소를 삭제한다.
index(x)	x와 같은 첫 번째 요소의 인덱스를 반환한다.
count(x)	x와 같은 요소들의 개수를 구한다.
sort()	정렬
reverse()	역순으로 배열
copy()	얕은 복사본을 반환한다.

이 함수들의 사용 예를 들면 다음과 같다.

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

append()함수와 extend()함수는 그 동작에 차이가 있다.

```
>>> x=list(range(5))
>>> x.append([5.6]) # x의 마지막 요소로 [5,6] 이 들어간다.
[0,1,2,3,4,[5,6]]

>>> x=list(range(5))
>>> x.extend([5.6]) # x의 오른쪽에 [5,6]의 요소를 병합시킨다.
[0,1,2,3,4, 5,6]
```

그리고 append() 함수와 pop()함수를 이용하면 리스트를 스택(stack)으로 운영할 수 있다.

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

이와 같이 리스트의 메쏘드를 이용하면 stack/queue 와 같은 고급 자료형을 쉽게 구현할 수 있다.

2.9 튜플

튜플(tuple, 터플이라고도 읽는다)은 객체들의 묶음이라는 점에서 리스트와 유사하다. 정의할 때 대괄호 대신 괄호 (..)를 사용한다.

```
>>> t1 = () # empty tuple 생성
>>> t2 = (11,) # tuple의 요소가 하나일 경우 반드시 끝에 콤마(,)를 붙여야 한다.
>>> t3 = (11, 22)
>>> t4 = ('abc', 11, [22, 33])
>>> t5 = ( (11,22), ('hi', 'world'))
>>> t6 = ((True, False))
```

위에서 빈 튜플을 생성하는 첫 번째 경우만 제외하고 모두 괄호를 생략할 수 있다.

```
>>> t2 = 11,
>>> t3 = 11, 22
>>> t4 = 'abc', 11, [22, 33]
>>> t5 = (11,22), ('hi', 'world')
>>> t6 = (True, False), # 이 경우 마지막에 콤마가 반드시 붙어야 한다.
```

이것을 보면 이전에도 몇 번 나왔던 여러 개의 변수를 동시에 생성시키는 경우도 튜플이 사용됨을 알 수 있다.

```
>>> a, b, c = 11, 22+33j, True # 좌변과 우변 모두 (괄호가 생략된) 튜플이다.

>>> t = 11,22,33 # t는 튜플임
>>> e,f,g = t # e=t[0];f=t[1];g=t[2] 와 같다.
>>> e
11
>>> f
22
>>> g
33
```

리스트와의 가장 차이점은 튜플은 그 크기나 개별 요소를 전혀 변경시킬 수 없다는 점이다. 즉, 한 번 생성되고 나면 변경시킬 수 없다. (뒤에서 설명할 immutable sequence 이다.) 튜플도 인덱싱과 슬라이싱이 가능하고 immutable sequence 들의 공통적인 연산(덧셈과 곱셈)이 가능하다.

2.10 딕셔너리

딕셔너리(dictionary)는 키(key)와 그것에 딸린 값(value)의 쌍들의 집합으로 이루어진 자료형이다. 어떤 사람과 관련된 데이터의 예를 들면 'name'은 '박장현' , 'age'가 '44', 등이 있을 것이다. 여기서 'name', 'age' 등이 키가 되고 '박장현'은 'name'이라는 키에 해당되는 값, '44'는 'age'라는 키에 해당되는 값이다. 이를 파이썬의 딕셔너리로 표현하면 다음과 같다.

```
>>> man1 = {'name':'박장현', 'age':44 }
```

이렇게 입력하면 man1 이라는 딕셔너리가 생성된다. 딕셔너리는 {...}기호로 생성된다. 이와 같이 딕셔너리는 '키:값' 쌍들의 집합이다. 키와 값은 콜론(:)으로 각 쌍은 콤마(,)로 구분한다.

```
{ key1:val1, key2:val2, ..... }
```

키로 쓸 수 있는 객체는 숫자와 문자열 등 immutable한 객체라면 무엇이든지 사용할 수 있다. 값에 해당하는 것은 어떤 파이썬 객체라도 올 수 있다.

만약 앞의 예에서 man1의 이름에 해당하는 값객체를 얻고 싶다면 다음과 같이 접근할 수 있다.

```
>>> man1['name']
```

이것은 man1 딕셔너리의 'name' 키에 해당하는 값 객체를 반환한다.

```
>>> capital = {'kr':'seoul', 'jp':'tokyo', 'cn':'beijing'}
```

이 예에서도 키가 문자열이고 값도 문자열이다. 딕셔너리는 데이터의 저장 '순서'라는 개념이 없다. 따라서 내부적인 인덱스는 없으며 오직 '키'로만 연결된 '값'에 접근할 수 있을 뿐이다.

```
>>> color = {0:’red’, 1:’yellow’, 2:’white’, 3:’black’}
```

위의 예도 딕셔너리이지만 이것은 키로 정수를 갖는다. 따라서

```
>>> color[1]            # ‘yellow’ 값을 읽는다.
>>> x = color[0]  # x에 문자열 ‘red’가 저장된다.
>>> color[3] = ‘grey’   # 키 3 에 해당하는 값을 변경한다.
>>> color[4] = ‘blue’   #새로운 키-값 쌍을 추가한다.
```

전술한 바와 같이 딕셔너리는 인덱스가 없다. 이 예제에서 color[1] 과 같은 용례는 인덱싱이 아니라 1이라는 키를 지정해 준 것이므로 혼동하면 안된다. 딕셔너리는 인덱싱도 할 수 없고 슬라이싱도 당연히 허용되지 않는다. 따라서 color[0:2] 같은 사용은 에러를 발생시킨다.

또 한 가지 주의할 것은 중복된 키는 허용되지 않는다는 것이다.

```
>>> a={0:'a', 1:'b', 0:'c'}
>>> a
{0: 'c', 1: 'b'}
```

위 예에서 딕셔너리 a에 0이라는 키가 중복으로 지정되었는데 결과를 보면 하나는 무시되었다. 딕셔너리는 내부 데이터를 키값으로 구별하기 때문에 중복된 키는 허용하지 않는 것이다.

딕셔너리의 키로 리스트는 사용 불가지만 튜플은 가능하다. 리스트는 mutable 이고 튜플은 immutable이기 때문이다.

```
>>> a={(1,):'hi', (1,2):'world'}
>>> a
{(1, 2): 'world', (1,): 'hi'}
```

값에는 어떠한 파이썬 객체도 올 수 있으며 딕셔너리 안에 값으로 딕셔너리가 다시 올 수도 있는 등 중첩도 얼마든지 가능하다.

2.10.1 딕셔너리의 내장 메서드들

딕셔너리에 대해서 사용할 수 있는 메써드는 다음과 같은 것들이 있다. 여기서 d는 딕셔너리 객체를 나타낸다.

[표 1] 딕셔너리의 내장 메소드

소속 함수	기능
d.keys() d.values() d.items()	키들을 모아서 dict_keys 객체로 반환한다. 값들을 모아서 dict_values 객체로 반환한다. (키,값) 튜플을 모아서 dict_items 객체로 반환한다.
d.clear()	모든 키:값 쌍을 제거하고 빈 딕셔너리로 만든다.
d.get(key)	key에 해당하는 값을 가져온다. d[key]와의 차이점은 해당 키가 없을 경우 에러를 발생시킨다는 것이다. (d[key]는 None을 반환함)
d.update(dict) d.update(**kwargs)	주어진 dict 혹은 kwargs 로 딕셔너리를 확장한다.

여기에서 dict_keys, dict_values, dict_items 객체는 모두 iterable 이다. 따라서 for 문에서 사용할 수 있다. 예를 들면 다음과 같다.

```
>>> capital = {'kr':'seoul', 'jp':'tokyo', 'cn':'beijing'}
>>> for val in capital.values():
...     print(val)
...
seoul
beijing
tokyo
```

만약 이 객체로부터 리스트를 생성할 필요가 있다면 파이썬 내장 함수 list() 를 이용하면 된다.

```
>>> list( capital.keys() ) # 키들로부터 리스트를 생성한다.
```

만약 딕셔너리 안에 해당 키가 있는지 조사하려면 in 연산자를 이용한다.

```
>>> 'kr' in capital # 'kr'이라는 키가 captal 딕셔닐에 있으면 True
True
>>> 'de' in capital
False
```

만약 값을 검색하려면 values() 메써드를 이용해야 한다.

```
>>> 'seoul' in capital.values()
True
```

특정 키:값 쌍을 삭제하려면 파이썬 내부 명령어인 del 을 이용하면 된다.

```
>>> del color[0] # 키가 0인 키-값 쌍을 삭제한다.
```

리스트, 튜플, 딕셔너리와 같은 자료형은 파이썬 프로그램에서 기본적으로 사용되는 자료형이기 때문에 확실하게 이해하지 않으면 프로그램을 효율적으로 작성할 수 없으며 다른 사람의 프로그램도 제대로 이해하기 힘들다.
update() 내장함수는 주어진 딕셔너리나 키워드인자로 원래의 딕셔너리를 확장한다.

```
>>> d={'a':11}
>>> d.update(b=22, c=33)
{'a': 11, 'b': 22, 'c': 33}
>>> a={'d':44, 'e':55}
>>> d.update(a)
{'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}
```

만약 두 개의 딕셔너리의 요소들을 통합하여 하나의 새로운 딕셔너리를 생성하려면 다음과 같이 하면 된다.

```
>>> a={'x':12, 'y':34}
>>> b={'a':56, 'y':78}
>>> d = {**a, **b}
 {'a': 56, 'e': 78, 'x': 12, 'y': 34}
```

2.11 나열형 (옵션)

나열형(sequence type)은 데이터들의 집합이며 기본적인 나열형 객체는 list, tuple, range 등이다. str형과 bytes형, bytearray 등도 나열형에 속한다. 나열형은 기본적으로 iterable 이다.

[표 1] 대표적인 가변(mutable) 불변(immutable) 나열형

가변 나열형 (mutable sequence)	불변 나열형 (immutable sequence)
list, bytearray	str, tuple, range, bytes

다음 표는 가변/불변 나열형 객체에서 지원하는 연산을 우선 순위 차례로 나열한 것이다. x는 임의의 객체이고 s, s1, s2는 나열형 객체들을 표기한다. n, i, j, k는 정수이다.

[표 2] 나열형의 기본 연산의 종류

연산	결과
x in s x not in s	s의 한 요소가 x와 같다면 True s의 한 요소가 x와 같다면 False
s1 + s2 s*n 혹은 n*s	두 시퀀스를 결합한다. 시퀀스를 n번 반복한다. (immutable 시퀀스의 경우 항상 새로운 시퀀스를 생성한다.)
s[i]	i번째 요소 (0부터 시작)
s[i:j]	i번째 부터 j-1번째 요소까지의 슬라이스
s[i:j:k]	i번째 부터 j-1번째 요소까지의(k는 스텝) 슬라이스
len(s)	요소의 개수
min(s)	가장 작은 요소
max(s)	가장 큰 요소
s.index(x[, i[, j]])	x와 같은 첫 번째 요소의 인덱스
s.count(x)	x와 같은 요소들의 개수

같은 형의 나열형 객체끼리는 비교도 가능하다. 특히 리스트와 튜플은 길이와 같은 인덱스를 가지는 모든 요소들끼리 같다면 두 리스트/튜플은 같은 것으로 판별된다.

```
>>> ‘gg’ in ‘eggs’
True
```

나열형 객체의 복사는 ‘얕은 복사’라는 것도 유의해야 한다. 즉, 중첩된 구조는 복사되지 않는다.

```
>>> lst=[ [] ]*3
>>> lst
[[], [], []]
>>> lst[0].append(1)
>>> lst
[[1], [1], [1]]
```

이 예제는 곱(*) 복사가 ‘얕은 복사’이기 때문에 원래의 빈 리스트의 참조를 복사해서 붙임을 알 수 있다. 따라서 하나가 변하면 다른 것들도 변한다. 같은 참조를 가지기 때문이다.

서로 다른 리스트를 만들려면 다음과 같이 하면 된다.

```
lst = [ [] for _ in range(3)]
>>> lst
[[], [], []]
>>> lst[0].append(1)
>>> lst[1].append(2)
>>> lst[2].append(3)
>>> lst
[[1], [2], [3]]
```

만약 ‘깊은 복사’를 수행하려면 copy 모듈의 deepcopy 함수를 이용하면 된다.

```
>>> x=[11,22]
>>> y=[x, 33]
>>> y
[[11, 22], 33]
>>> from copy import deepcopy
>>> z = deepcopy(y)
>>> z
[[11, 22], 33]
>>> x[0]=-44
>>> y
[[-44, 22], 33] #x가 변하면 y도 변한다.
>>> z
[[11, 22], 33] # x가 변해도 z는 변함이 없다.
```

가변 나열형 객체의 경우 다음과 같은 조작이 추가로 가능하다.

[표 3] 가변 나열형 객체의 조작

연산	결과
s[i] = x	s의 i번째 요소를 x로 교체
s[i:j] = t	i번째 요소부터 j-1번째 요소를 t(iterable)로 교체
del s[i:j]	i번째 요소부터 j-1번째 요소를 삭제 (s[i:j] = [] 와 동일)
s[i:j:k] = t	i번째 요소부터 j-1번째 요소(k는 스텝)를 t(iterable)로 교체 ❶
del s[i:j:k]	i번째 요소부터 j-1번째 요소(k는 스텝)를 삭제
s.append(x)	s의 마지막 요소로 x를 삽입
s.extend(t)	t의 내용물로 s를 확장 (s[len(s):len(s)]=t 와 동일)
s.insert(i, x)	i 번째에 x를 삽입
s.pop() s.pop(i)	마지막 삭제하고 그것을 반환한다. i 번째 요소를 삭제하고 그것을 반환한다.
s.remove(x)	s의 요소 중 x와 같은 첫 번째 것을 제거 ❷
s.reverse()	요소들을 역순으로 배열한다. ❸
s.clear()	모든 요소 삭제 (del s[:] 과 동일) ver3.3부터 도입
s.copy()	얕은 복사본 생성 (s[:] 와 동일) ver3.3부터 도입

❶ t 와 슬라이싱 된 요소들의 크기가 같아야 한다.
❷ s 안에 x가 없다면 ValueError 예외가 발생한다.
❸ 요소의 순서를 역순으로 바꾼다. (역순으로 바뀐 객체가 반환되는 것이 아니다.)

2.12 Enum클래스 (옵션)

파이썬 3.4 이상에서 표준화된 Enum객체를 지원한다. 자세한 설명은 여기에 있다. 간략한 사용법을 알아보도록 하자.

일단 Enum 클래스를 임포트해야 한다.

```
>>> from enum import Enum
```

첫 번째로 다음과 같이 클래스를 Enum을 상속해서 생성할 수 있다.

```
>>> class Color(Enum):
...     red = 1
...     green = 2
...     blue = 3
```

이제 Color.red 또는 Color.green 과 같이 사용할 수 있다. 첫 번째 값은 보통 1부터 시작한다. (0으로 시작하면 그 필드는 False가 된다.)

다른 방법으로 다음과 같이 더 간단히 생성할 수 있다.

```
>>> Color = Enum('Color', 'red green blue')
```

유의할 점은 인스턴스 이름과 Enum생성자의 첫 인자가 (위에서 밑줄 쳐진 두 부분) 같아야 한다. 이제 이전과 마찬가지로 Color.red 또는 Color.green 과 같이 사용할 수 있다.

enum 필드는 name과 value를 가진다. 예를 들어 Color.blue 의 name 은 'blue' value 는 3 이다.

```
>>> Color.blue.name
'blue'

>>> Color.blue.value
3

>>> type(Color.blue)
<enum 'Color'>
```

3장 제어문

3.1 if ~ elif ~ else

if 명령은 그 뒤에 오는 조건식의 참/거짓 여부에 따라 속한 블럭을 수행할지 말지를 결정하는 명령이다. 가장 기본 문법은 다음과 같다.

```
if 조건식:
    실행문1
    실행문2
    ...
```

여기에서 조건식이 참(True)이면 실행문1, 실행문2 ... 가 수행되고 그렇지 않으면 실행되지 않는다. 주의할 점은 if 문에 속한 모든 실행문은 들여쓰기가 같아야 한다는 것이다. 한 칸이라도 틀리면 문법 오류를 발생하게 된다. 파이썬에는 관행적으로 한 수준의 들여쓰기는 공백문자 4칸으로 한다. 또한 조건식 뒤의 콜론(:)도 처음에는 빠뜨리기 쉬우니 조심하자.

파이썬은 들여쓰기가 문법에 포함된 거의 유일한 언어이다. 처음에 이러한 문법에 익숙치 않다면 tab과 공백문자들을 섞어쓰는 오류를 범하기 쉬운데 주의해야 한다. 탭과 공백문자들이 보기에 같은 크기만큼 들어갔다고 하더라도 서로 다른 들여쓰기로 해석되므로 예외를 발생시킨다.

조건식에는 보통 관계연산자와 논리연산자가 사용된다.

[표 1] 조건식에서 사용되는 관계/논리 연산자들

분류	연산자들
관계연산자	==, !=, <, >, <=, >=, is, is not, in, not in
논리연산자	and, or, not

예를 들면 다음과 같다.

```
n = int(input("integer:"))
if n%2==0:
    print("even number!")

#실행:
integer:12
even number!
```

이 예는 입력 받은 정수가 짝수일 경우 화면에 ‘even number’라고 출력하고 홀수일 경우는 아무런 일도 하지 않는 것이다.

좀 더 일반적인 if 명령의 문법은 else 와 짝을 이루는 것이다.

```
if 조건식 :
    실행문1
    실행문2
    …
else :
    실행문3
    실행문4
    …
```

이 경우 조건식이 거짓이면 else 블럭(실행문 3, 실행문4…)을 수행하게 된다.

```
age = int(input("나이:"))
if age<30:
    print("청년")
else:
    print("중년")

#실행:
나이:44
중년
```

사용자의 나이 입력을 받아서 나이가 30 미만으면 ‘청년’, 30 이상이면 ‘중년’이라고 출력하는 간단한 예제이다.

일반적인 if 명령의 구조는 다음과 같다.

```
if 조건식1 :
    실행문1
    ...
elif 조건식2
    실행문2
    …
elif 조건식3
    실행문3
    …
else :
    실행문n
    …
```

키워드 elif 는 else if 를 줄인 단어이다. 이 구조에서 조건식1이 참이라면 실행문1을 수행하고 if 블럭을 완전히 빠져나간다. 만약 조건문1이 거짓이라면 조건식2를 판별한다. 그래서 조건식2가 참이면 실행문2를 실행하고 if 블럭을 빠져나간다. 모든 조건이 거짓일 경우 else 문에 속한 실행문n이 실행된다.

```
s = 'hello'
if 'a' in s:
    print("'a' is contained")
elif 'b' in s:
    print("'b' is contained")
else:
    print("both 'a' and 'b' are not contained")

#실행결과:
both 'a' and 'b' are not contained
```

파이썬에는 C/C++/JAVA/C# 등에는 있는 switch - case 명령문이 없다. 따라서 비교적 많은 수의 다중 조건을 판단해야 할 경우에도 if - elif- else 문을 조합하여 구성해야 한다.

가끔 조건문을 판단하고 참 거짓에 따라 행동을 정의 할 때 아무런 일도 하지 않도록 설정을 하고 싶을 때가 생기게 된다. 다음의 예를 보자.

"집에 돈이 있으면 가만히 있고 집에 돈이 없으면 노트북을 팔아라"

위의 예를 pass를 적용해서 구현해 보자.

```
>>> home = [‘money’, 'TV', 'radio', 'notebook']
>>> if 'money' in home:
...     pass
... else:
...     print("sell notebook.")
```

home이라는 리스트 안에 'money'라는 문자열이 있기 때문에 if 문 다음 문장인 pass가 수행되었고 아무 일도 수행하지 않는 것을 확인 할 수 있다.

위의 예를 보면 if문 다음의 수행할 문장이 한 줄이고 else문 다음에 수행할 문장도 한 줄이다. 이렇게 수행할 문장이 한 줄일 때 조금 더 간편한 방법이 있다. 위에서 알아본 pass를 사용한 예는 다음처럼 간략화할 수 있다.

```
>>> home = [‘money’, 'TV', 'radio', 'notebook']
>>> if 'money' in home: pass
... else: print("sell notebook.")
```

if 문 다음의 수행할 문장을 ':'뒤에 바로 적어 주었다. else 문 역시 마찬가지이다. 이렇게 하는 이유는 때로는 이렇게 하는 것이 보기에 편하게 때문이다.

3.2 for 반복문과 range() 함수

파이썬의 for 반복문은 C/C++의 그것과는 약간 다르다. (JAVA나 C#의 foreach 명령과 유사한 점이 많다.) 기본적인 문법은 다음과 같다.

```
for 변수 in 반복형:
      수행문
```

여기서 반복형은 열거형(sequence, range형, 문자열, 리스트, 튜플 등), 딕셔너리 등이 있다.

예로 기존의 리스트를 이용하여 리스트 요소들 중 짝수의 개수를 세는 프로그램을 작성해 보자.

```
>>> lst = [11, 44, 21, 55, 101]
>>> cnt = 0
>>> for n in lst :
...     if n%2==0:
...           cnt += 1
>>> cnt
1
```

이 예제는 리스트 lst 의 각 요소가 순서대로 n 변수에 대입되고 반복문이 수행된다.

for 반복문의 in 지정어 다음에 딕셔너리가 오는 경우는 키값만 추출된다.

```
d = {‘name1’:’salesio’, ‘name2’:’park’}
for k in d:
    print(k)

name1
name2
```

이 예제의 출력은 d의 키값들만 표시된다. 키와 값 두 개를 차례로 취하고 싶으면 딕셔너리의 items() 메서드를 이용하면 된다.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

만약 리스트의 위치까지 반복 문자로 사용하려면 enuerate()함수를 이용하면 된다.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

이 예제에서 변수 i가 리스트 요소의 인덱스값을 가지고 반복문이 수행된다.

3.2.1 range()함수와 range 형

for 구문과 자주 같이 사용되는 파이썬 내장 함수 중에서 range() 함수가 있는데 이 함수는 range 형을 만들어 준다. (range 형은 시퀀스 형에 속한다.) for 반복문과 조합되어서 자주 사용되므로 여기에서 자세히 설명하도록 하겠다. range() 함수는 세 가지 용법이 있다. 다음에서 m, n, s는 정수이다.

```
range(n)  # 0 부터 n-1 까지의 요소를 가지는 range 형 반환
range(m,n)  # m 부터 n-1 까지의 요소를 가지는 range 형 반환
range(m,n,s)  # m 부터 n-1 까지의 요소(s는 step)를 가지는 range 형 반환
```

한 가지 주의할 점은 파이썬 3.x에서는 range()함수는 range 형을 반환한다. (반면 파이썬 2.x에서는 리스트를 반환한다.) 따라서 파이썬 3.x에서는 이 rangeg형을 리스트로 만들기 위해서는 list()함수를 반드시 명시적으로 사용해야 한다.

```
>>> range(5)
range(0,5)
>>> list(range(5))
[0,1,2,3,4]
>>> list(range(1,11))
[1,2,3,4,5,6,7,8,9,10]
>>> list(range(0,30,5)) # 5가 증분
[0,5,10,15,20,25]
>>> list(range(0,-5,-1)) # -1이 증분
[0,-1,-2,-3,-4]
>>> list(range(0))
[]
>>> list(range(1,0))
[]
```

하지만 for문 안에서 range 함수를 사용할 경우 굳이 리스트로 바꿀 필요는 없다. 왜냐면 for 문에는 반복형(iterable)이 사용되며 range 객체는 열거형(sequence)이므로 반복형이기 때문이다.

예를 들어서 'hi.'라는 문자열을 다섯 번 출력하는 프로그램을 작성해 보자.

```
for _ in range(5):
    print('hi.')
```

이 반복문은 변수 _ 가 0,1,2,3,4 값을 가지고 각각 반복을 수행하게 된다. 실제 이 변수가 사용되지는 않으므로 그냥 _ 로 지정하였다. (파이썬에서 dummy variable 의 이름은 보통 _ 로 지정한다.)

구구단의 2단을 출력하는 프로그램을 작성해 보자.

```
>>> for n in range(2,10):
...     print('2 x %d = %d'%(n, 2*n))

2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
```

여기서 변수 n은 2부터 9까지의 값을 가지고 반복문이 수행된다. range(2,10)은 2부터 9까지의 숫자를 차례로 생성한다.

for 반복문은 중첩해서도 얼마든지 사용할 수 있다. 구구단의 2단부터 9단까지 한꺼번에 출력하려면 다음과 같이 작성하면 될 것이다.

```
>>> for m in range(2,10):
...     for n in range(2,10):
...             print('%d x %d = %d'%(m,n,m*n))
...     print('-'*10)
```

여기에서 print('-'*10)은 '----------' 을 출력한다.

3.2.2 break 명령과 for~else 구문

파이썬의 for 반복문 안에서 break 명령을 사용할 수 있는데 이 명령은 즉시 그것이 포함된 가장 안 쪽의 반복문을 빠져 나온다. 다음 반복문을 보자.

```
for n in lst:
    if n==0:
        break
    print(n)
```

이것은 lst 안의 요소들을 차례대로 프린트하다가 0이 발견되면 바로 반복을 멈추는 프로그램이다. 만약 lst=[1,2,3,0,4,5] 라면 1,2,3만 출력되고 반복문은 종료될 것이다. 만약 반복문의 중첩되어 있다면 가장 안쪽의 반복문만 빠져 나온다는 점에 유의해야 한다.

파이썬의 for 반복문은 else 명령과 짝을 이룰 수도 있다.

```
for 변수 in 반복가능자:
    수행문들1
else:
    수행문들2
```

else 블럭에 포함된 '수행문들2'는 for 반복문이 반복형의 마지막 요소까지 모두 반복했을 경우 그 다음에 수행된다. 즉, 모든 반복이 성공적으로 수행된 경우에 한 번 수행된다. 하지만 **break**문을 만나면 **else** 구문은 수행되지 않고 **for** 블럭을 완전히 빠져나간다.

```
for n in lst:
    if n==0: break
    print(n)
else:
    print('There is no 0.') # break문을 만나지 않았다면 수행된다.
```

이 예제의 경우 lst 안에 0이 있다면 break를 만나게 되고 따라서 else 블럭은 수행되지 않고 for 반목문을 빠져 나오게 된다.

따라서 **for** 반복문이 종료된 시점에서 이 종료가 모든 반복을 다 수행한 후의 정상적인 종료인지, 아니면 **break** 명령에 의한 강제 종료인지에 따라서 수행해야 될 일을 구분할 필요가 있는 경우에 for~else 구문을 사용하면 된다.

3.3 while 반복문

파이썬의 while 명령은 단순 반복을 수행한다.

```
while 조건식:
      실행문
      ....
```

여기서 조건식이 참이면 실행문을 수행하고 다시 조건을 검사한다. 즉, 조건문이 거짓이 될 때까지 실행문을 반복 수행하는 것이다.

```
>>> m,p = 1,1
>>> while m<=10:
...     p *= m
...     m += 1
...
>>> p
3628800
```

이 예는 10! 을 구하는 프로그램이다. 10!=3628800 이라는 것을 알 수 있다.

for문과 마찬가지로 while 문도 else 절이 붙을 수 있다.

```
while 조건식:
      실행문들1
else:
      실행문들2
```

조건식이 거짓으로 판정되어서 '실행문들1'이 수행되지 않을 때 else 절의 '실행문들2'가 수행된다. 만약 break 문에 의해서 반복이 끝난다면 for 반복문과 마찬가지로 else절은 수행되지 않고 그 바깥으로 빠져 나가게 된다.

```
n=3
while n>=0:
    m = input("Enter an integer :")
    if int(m)==0: break
    n -= 1
else:
    print('4 inputs.')
```

이 예제는 4개의 0이 아닌 정수를 입력 받으면 else 절이 수행된다.

```
.#실행 결과 1
Enter an integer :1
Enter an integer :2
Enter an integer :3
Enter an integer :4
4 inputs.
```

만약 그 전에 0이 입력된다면 else 절이 수행되지 않고 while 반복문을 완전히 종료하게 된다.

```
# 실행 결과 2
Enter an integer :0
```

for 반복문과 마찬가지로 break 문에 의한 반복 종료인지 아니면 조건문이 False 가 되어서 반복문을 종료하는 지를 구별하여 다른 실행문을 수행할 경우에 while ~ else 절을 사용하면 된다.

```
m=997
n=2
while n<m//2:
    if m%n==0: break
    n += 1
else:
    print(m,'is a prime number!')
```

이 예는 997이 소수(prime number)인지 아닌지 판별하는 함수이다. 2부터 498까지 차례로 나누어서 나머지가 한번이라도 0이 된다면 break 문에 걸리게 된다. 만약 한 번도 0이 아니라서 반복문이 끝까지 돌았다면 else 절이 수행되어 소수임을 표시한다.(음영진 부분은 seq 형이다.)

3.4 continue 명령

반복문 안에서 continue 명령은 그 이후의 반복문은 수행하지 않고 즉시로 다음 반복 실행으로 넘어가는 동작을 수행한다.

```
for n in lst:
    if n==0: continue
    print(n)
```

이 예제에서 lst의 요소가 만약 0 이라면 continue 명령을 만나게 되고 그 이후의 print(n)은 수행되지 않고 다음으로 바로 넘어가게 된다.

이 예를 while 문으로 작성하면 다음과 같다.

```
k = 0
while k<len(lst) :
    if lst[k]==0:
        k +=1
        continue
    print(lst[k])
    k +=1
```

이와 같이 continue 명령은 반복문 안에서 사용되며 그것을 둘러싼 가장 안쪽의 반복문의 다음 단계로 즉시 넘어가는 동작을 수행한다.

3.5 제어 블럭에서 변수 범위

앞에서 살펴본 바와 같이 if, for, while 문은 동일한 들여쓰기가 적용된 실행 블럭(block)을 동반한다. C와 문법이 유사한 언어들(c++, java, c# 등등)과는 다르게 파이썬에서는 이 블럭 내부에서 사용된 변수들을 실행이 끝난 다음에도 그 내용을 가지고 남아있게 된다.

다음과 같은 간단한 예를 보자.

```
for k in range(10):
    a = k+1
print(k, a) # 9, 10 이 출력된다.
```

이 for 명령이 수행되는 동안 k변수가 생성되고 for 블럭 안에서 a 변수가 새로 만들어져 사용된다. 문제는 for 반복이 종료된 이후에도 이 변수들은 남아있다는 점이다. C계열의 언어들 (c++, java, c# 등등)에서는 반복문이나 조건문 내부에서 선언된 변수는 내부 변수이므로 실행 종료 후 소멸되므로 이러한 점이 의아할 수 있다. 하지만 파이썬에서는 이렇게 동작하므로 염두에 두어야 한다.

다른 예를 보자

```
b = 0
for k in range(101):
    b += k
print(b) # 5050
```

이 코드는 잘 수행된다. 내부블럭에서 b를 참조할 때 외부 변수를 참조하기 때문이다. 하지만 외부에서 b를 생성하지 않고 참조하려면 예외가 발생된다.

```
for k in range(101):
    b += k # 예외 발생
```

만약 내부 블럭에서 생성되지 않은 변수라면 외부에서 찾아보기 때문이다. 외부에도 없다면 예외가 발생된다.

즉, 제어 블럭의 경우는 변수 공간이 외부와 분리되지 않는다고 생각하면 된다. 이 예에서는 for 블럭만 예를 들었지만 if, while 블럭도 마찬가지로 동작한다. 변수의 범위에 대해서는 함수를 설명하는 부분에서도 다시 한 번 다루게 될 것이다.

4장 함수

4.1 함수의 정의와 호출

함수(function)란 실행문들을 묶어서 하나의 블럭으로 만든 후 이름을 붙인 것을 말한다. 이렇게 수행문들의 집합을 함수로 정의하면 그 수행문들을 동작시킬 때 함수 이름을 이용한 간단한 호출(call)로 반복해서 실행시킬 수 있다.

파이썬 함수는 다음과 같이 정의된다.

```
def 함수명(인자1, 인자2, ...):
      함수 본체
```

함수의 정의는 항상 def 로 시작한다. 함수명은 일반적인 식별자를 사용하며 관례적으로 (변수명과 마찬가지로) 소문자로 시작한다. 그다음에 호출하는 쪽에서 넘겨 받을 인자들의 리스트를 괄호(...)안에 콤마로 구별하여 지정해 주고 콜론(:) 뒤에 함수의 본체를 작성해 주면 된다. 함수의 본체는 반드시 def 의 첫 글자 시작 위치보다 들여 써야 한다.

간단한 예를 들어보자

```
>>> def sayHi():
...     print('hi')
```

이 함수는 'hi'라는 문자열을 출력하는 함수이며 함수의 이름은 sayHi()이다. 입력 인자는 없으며 반환값도 없다. 이 함수를 호출하려면 다음과 같이 하면 된다.

```
>>> sayHi()
Hi
```

구구단을 출력하는 함수를 예로 들어보자.

```
>>> def gugu(n):
...     for x in range(2,10):
...         print('%d x %d = %d'%(n,x,n*x))
```

이 함수는 하나의 입력 인자를 받도록 되어 있다. 다음과 같이 호출한다.

```
>>> gugu(4)
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36

>>> gugu(8)
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
```

함수명은 함수객체이며 다른 객체와 유사하게 대입이 가능하다. 예를 들어 앞에서 정의한 함수 sayHi()와 gugu()를 하나의 리스트로 묶을 수 있다.

```
>>> fl = [sayHi, gugu]
```

여기서 fl 리스트의 첫 번째 요소는 함수 sayHi 이고 두 번째 요소는 gugu 이다. 따라서 다음과 같은 함수 호출이 가능하다.

```
>>> fl[0]()
Hi.
>>> fl[1](9)
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
```

또는 함수를 다른 변수에 대입할 수도 있다.

```
>>> kuku = gugu
```

이제 kuku 는 함수이며 gugu()함수와 같은 함수이다. 따라서 다음과 같이 동일하게 호출할 수도 있다.

```
>>> kuku(7)
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
```

가끔 함수의 본체를 구현하지 않고 껍데기만 작성해야 될 경우도 있다. 이럴 경우 다음과 같이 하면 된다.

```
>>> def nop(): pass
```

이 함수는 호출은 할 수 있으나 아무런 일도 수행하지 않는다.

4.2 함수의 일반인자와 반환값

함수에는 입력 인자(argument)가 있을 수도 있고 없을 수도 있으며 반환값도 마찬가지이다. 호출하는 쪽에 반환값을 되돌려주기 위해선 return 이라는 키워드를 이용하여 그 뒤에 반환할 값을 써 주면 된다.

```
>>> def sayHi():
...         print('Hi.')
>>>a = sayHi()
Hi.
```

여기에서는 반환값이 없는 함수의 결과값을 a변수에 저장하였는데 (일반적이지 않은 방법이긴 하지만) 오류를 발생하지 않는다. 이 경우 a라는 변수에 None 값이 저장된다.

```
>>> type(a)
NoneType
```

함수를 호출하는 쪽에서는 순서대로 값을 입력해야 한다.

```
>>> def mod(x,y):
.....   return x%y
>> mod(3,2) # x에 3, y에 2가 전달된다.
```

이 예는 x를 y로 나눈 나머지를 반환하는 함수인데 입력 인수는 순서대로 정의된 모든 변수에 주어야 한다. 다음과 같은 호출은 오류를 발생시킨다.

```
>>> mod()
>>> mod(3)
>>> mod(3,2,1)
```

이와 같이 함수의 정의부에 변수명만 있는 인자를 일반 인자(standard argument)라고 하며, 호출하는 쪽에서는 반드시 순서와 개수를 맞추어서 넘겨주어야 한다.

반환 값이 두 개 이상일 경우에는 return 명령 뒤에 콤마(,)로 구분해야 한다.

```
>>> def cal(x,y):
...     return x+y, x-y
...

>>> cal(11,22)  # 반환값은 튜플이다.
(33, -11)

>>> a,b=cal(33,44)
>>> a
77
>>> b
-11
```

이 간단한 예제에서 cal() 함수는 두 수의 합과 차 두 개를 반환한다. 호출하는 쪽에서는 결과 값들의 튜플을 받게 된다. 다른 변수로 각각 받으려면 위와 같이 하면 된다.

```
>>> a, b = cal(33,44)
```

a변수에는 합이, b변수에는 차가 저장됨을 알 수 있다.

4.3 함수의 가변 인수 인자

어떤 함수는 호출하는 쪽에선 몇 개의 인수를 넘겨주는 지 모르는 경우가 있다. 이럴 경우에 사용되는 것이 가변 개수 인자 기능으로써 함수는 넘어온 인자들을 묶어서 단일 튜플로 받는다.

예를 들어 넘어 온 인자들 중 짝수의 개수를 구하는 함수를 작성해야 하는데 몇 개의 입력이 들어올 지는 모른다고 가정하자. 인자들을 묶어서 한 개의 리스트를 받는 방식을 먼저 생각해 볼 수 있겠다.

```
def count_even(lst):
    cnt = 0
    for v in lst:
        if v%2==0:
            cnt +=1
    return cnt

>>> count_even([1,2,3,4,5,6,7])
      3
>>> count_even([11,22,33])
1
```

이런 식으로 하나의 입력 인수를 받되 리스트 요소의 개수는 그때 그때 변할 수 있겠다.

하지만 이런 경우에 있어서 좀 더 일반적인 해법은 가변 개수 인수를 받는 것이다. 가변 개수 인자를 지정할 때 함수의 정의부에서 변수 앞에 별표(*)를 붙인다.

```
>>> def count_even(*args):
    cnt = 0
    for v in args:
        if v%2==0:
            cnt +=1
    return cnt

>>> count_even(11,22,33)
1
>>> count_even(1,2,3,4,5,6,7)
3
```

앞의 경우와 다르게 함수를 호출할 때 콤마로 구분된 입력 인자를 몇 개라도 입력할 수 있다는 것이다. 함수 내부에서는 이들 인자들을 요소로 갖는 튜플이 변수 **args**에 넘어오게 된다. 즉 type(**args**)은 tuple 이다.

한 가지 주의할 점은 일반 인자와 가변 인자가 동시에 오는 경우 반드시 가변 인자는 일반 인자 뒤에 와야 한다는 점이다. 또한 가변 인자는 단 하나만 사용할 수 있다는 것도 유의해야 한다.

```
>>> def  f1(a, *args): # 정상
>>> def  f2(a, *args, c): # 오류 - 가변 인자 뒤에 일반 인자가 올 수 없다.
>>> def  f3(*args, *args): #오류 - 가변 인자는 하나만 사용 가능
>>> def  f4(a, b, *args) # 정상
```

이러한 사항에 주의해서 함수를 작성해야 한다.

4.4 함수의 기본값 인자

파이썬 함수의 기본값 인자(default parameter)란 함수를 호출할 때 인자의 값을 설정하지 않아도 기본값이 할당되도록 하는 기능이다. 기본값은 함수를 정의할 때 지정해준다. 예를 들어 보자.

```
def funcA(a=10):
    print('a='+str(a))
```

이 함수는 다음과 같이 두 가지 방법으로 호출할 수 있다.

```
>>> funcA() # a에 기본값인 10이 자동으로 저장된다.
a=10
>>> funcA(20) #a에 20이 전달된다.
a=20
```

이 예에서 인자 a는 기본값 인자이고 호출하는 쪽에서 값을 주지 않으면 기본적으로 10 값을 갖게 된다. 그래서 funcA()와 같이 값을 주지 않으면 10이 출력되고 funcA(20)과 같이 값을 주면 그 값이 인자 a 로 전달되는 것이다.

다른 예를 들어 보자.

```
def printName(firstName, secondName='Kim'):
    print('My name is '+ firstName +' ' + secondName +'.')
```

이 함수도 다음과 같이 두 가지 방법으로 호출 가능하다.

```
>>> printName('Jang-Hyun')
My name is Jang-Hyun Kim.
>>> printName('Jang-Hyun', 'Park')
My name is Jang-Hyun Park.
```

기본값 인자는 값을 생략해서 호출할 수 있지만 일반 인자는 반드시 값을 지정해 주어야 한다. 따라서 printName()함수를 호출할 때 첫 문자열은 반드시 정해서 넘겨주어야 한다. 여기에서 printName()함수의 두 번째 인자를 입력하지 않으면 기본적으로 'Kim'으로 지정된다. 두 번째 인자를 명시적으로 지정하면 그것이 secondName 으로 지정된다.

한 가지 주의할 점은 일반 인자와 기본값 인자가 같이 올 때에는 반드시 기본값 인자는 뒤에 와야 한다는 점이다. 즉, 아래와 같이 정의하면 오류가 발생한다.

```
>>> def printName(firstName='Kim', secondName):
```

기본값 인자가 두 개 이상일 때에서도 항상 일반 인자 뒤에 와야 한다.

```
>>> def add(a, b=0, c=0):
...     return a+b+c
...
>>> add(1)
1
>>> add(1,2) # b에 2가 들어간다.
3
>>> add(1,2,3) # b에 2, c에 3이 들어간다.)
6
```

이 예에서 보듯이 함수를 호출할 때 주는 순서 대로 기본값 인자에 값이 할당 됨을 알 수 있다.

또 한 가지 주의할 점은 리스트나 딕셔너리와 같은 가변(mutable) 객체를 기본 인자로 사용할 때 최초의 호출 시에만 지정된 값으로 초기화 되고 이후의 호출에서는 그렇지 않다는 점이다.

```
>>> def f(a, L=[]):
...     L.append(a)
...     return L
...
>>> f(1)
[1]
>>> f(2)
[1, 2]
>>> f(3)
[1, 2, 3]
```

이 예제를 보면 기본 인자 L은 최초의 호출에서만 빈 리스트로 초기화 되고 그 이후의 호출에서는 그 내용물은 유지된다. 따라서 리스트의 요소가 축적되는 것이다. (마치 C/C++에서의 static 변수와 비슷한 동작을 수행한다.)

후속 호출에도 mutable 객체를 초기화하려면 다음과 같은 방법으로 코딩하면 된다.

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

이 경우 실행 결과는 다음과 같다.

```
>>> f(1)
[1]
>>> f(2)
[2]
>>> f(3)
[3]
```

4.5 함수의 키워드 인자

키워드 인자(keyword argument)는 함수를 호출할 때 인자의 값 뿐만 아니라 그 이름까지 명시적으로 지정해서 전달하는 방법이다. 만약 기본 인자가 세 개인 다음과 같은 간단한 함수가 있다고 가정하자.

```
def add(a=0, b=0, c=0):
    return a+b+c
```

이 함수를 호출하는 방법은 다음과 같은 것들이 있다.

```
>>> add()
0
>>> add(11) # a에 11이 대입됨
11
>>> add(11,22) # a에 11, b에 22가 대입됨
33
>>> add(11,22,33) # a에 11, b에 22, c에 33이 대입됨
66
```

여기서 보듯이 인자의 순서대로 기본값 인자가 결정된다. a만, 혹은 a와 b에만 원하는 값을 대입하는 것은 문제가 없다. 그런데 만약 a와 b는 기본 인자 그대로 사용하고 c에만 다른 값을 넣어주고 싶을 때는 어떻게 해야 할까? 이런 경우 호출하는 쪽에서 변수명(키워드)를 지정하면 된다.

```
>> add(c=33) # c에만 33이 대입된다.
>> add(c=33, b=22) #c에 33, b에 22가 대입된다.
```

이와 같이 키워드 인자값을 넘겨줄 때에는 함수 정의부의 인자 순서와 상관 없이 나열할 수 있다.

기본값 인자와 마찬가지로 키워드 인자는 함수 정의시에 일반 인자(standard argument) 뒤에 와야 한다.

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
        print("-- This parrot wouldn't", action)
        print("if you put", voltage, "volts through it.")
        print("-- Lovely plumage, the", type)
        print("-- It's", state, "!")
```

이 함수는 일반 인자(voltage) 하나와 세 개의 기본 인자(state, action, type)을 가지고 있다. 이 함수는 다음과 같이 호출될 수 있다.

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='VOOOOOM')
parrot(action='VOOOOOM', voltage=1000000)
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

하지만 다음과 같이 호출할 수는 없다.

```
parrot()                # 일반 인자값 없음
parrot(voltage=5.0, 'dead')  #키워드 인자 뒤에 일반 인자값을 주지 못함
parrot(110, voltage=220)    # 한 인자에 두 값을 주지 못함
parrot(actor='John Cleese')  # actor라는 키워드가 없음
```

함수 호출할 때에도 키워드 인자는 일반 인자 뒤에 와야 한다. 또한 키워드 인자로 넘겨지는 것은 반드시 함수 정의에서 매칭되는 변수가 있어야 하며 호출하는 순서는 중요하지 않다.

만약 함수 정의부에 ****kwargs** 와 같은 형태의 인자가 있다면 **kwargs**는 딕셔너리로 키워드 인자들 중에서 일반 인자가 아닌 것들을 넘겨 받는다.

```
def foo(**kwargs):
    print(kwargs)
```

이 함수를 호출할 경우에는 반드시 인자의 이름까지 명시해 주어야 한다.

```
>>> foo(x=11, y=22)
{'x':11, 'y':22}
```

따라서 foo 함수의 내부에서는 어떤 인자에 어떤 값이 넘어왔는지 kwargs 딕셔너리를 검사하여 파악할 수 있다. 가변 키워드 인자 **kwargs 는 반드시 일반인자나 키워드 인자 뒤에 와야 한다.

만약 가변 개수 인자 *args 와 혼용하는 경우에는 반드시 이것 뒤에 와야 한다.

```
def f(a, *args, **kwargs):
    print('a=',a)
    print('args=',args)
    print('kwargs=',kwargs)
```

이 함수는 다음과 같이 호출될 수 있다.

```
>>> f(11)
a= 11
args= ()
kwargs= {}
>>> f(11,22,33)
a= 11
args= (22, 33)
kwargs= {}
>>> f(11,22,33,b=44,c=55)
a= 11
args= (22, 33)
kwargs= {'c': 55, 'b': 44}
```

이 예에서 보듯이 반드시 일반 인자(a), 가변 개수 인자(*args), 키워드 인자(**kwarg) 의 순으로 나열되어야 함을 주의하자.

어떤 딕셔너리의 요소들을 키워드 인자로 어떤 함수로 넘겨줄 경우에도 ** 지정자를 이용하면 된다.

```
>>> def foo(**kwargs):print(kwargs)
>>> d={'x':22, 'y':'hi'} #변수명을 문자열키로 지정
>>> foo(**d)
{'x': 22, 'y': 'hi'}
```

위 예와 같이 키워드 인자의 변수명이 문자열 키로 저장된 딕셔너리를 foo()함수의 인자로 foo(**d)와 같이 넘겨주면 함수를 다음과 같이 호출하는 것과 동일하다.

```
>>> foo(x=22, y='hi')
```

이와 같이 딕셔너리를 어떤 함수의 키워드 인자들로 풀어서 넘겨줄 수도 있다.

또한 **을 이용하면 두 개의 딕셔너리 요소들을 모아서 하나의 딕셔너리로 만들 수 있다.(python 3.x에만 해당됨.)

```
>>> a = {'x':11, 'y':22}
>>> b= {'z':33}
>>> c = {**a, **b}
{'x':11, 'y':22, 'z':33}
```

4.6 익명 함수

익명 함수(lambda function)란 말 그대로 이름이 없는 함수이며 파이썬에서는 lambda 라는 키워드로 익명 함수를 정의할 수 있다. 주로 비교적 간단한 기능의 함수가 컨테이너의 요소로 들어가는 경우 혹은 다른 함수의 인자로 함수를 넘겨줄 필요가 있을 때 사용된다.

익명 함수는 다음과 같이 생성된다.

```
lambda 인자1,인자2, ... : 표현식
```

익명 함수는 보통 한 줄로 정의되고 **return**문도 없으며 단지 인자들과 반환값들의 관계식으로만 표현된다. 예를 들어 두 수의 합을 반환하는 익명 함수는 다음과 같다.

```
>>> add = lambda a, b : a+b
>>> add(1,2)
3
```

이 익명 함수는 다음과 같이 일반 함수를 정의하는 것과 동일하다.

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

그렇다면 일반 함수와 익명 함수와의 차이점은 무엇인가? 익명 함수의 기능은 일반 함수보다도 훨씬 제한적이고 익명 함수로 할 수 있는 것은 일반 함수로도 모두 할 수 있다. 그렇다면 왜 익명 함수를 사용할까? 어떤 경우에는 굳이 번거롭게 일반 함수를 정의할 필요가 없이 간단한 기능만을 구현해도 되는 때가 있다. 예를 들어서 입력 인수가 0보다 크면 True를 반환하는 함수를 생각해 보자.

```
>>> def pos(x):
.....    return x>0
>>> list(filter(pos, range(-5,6)))
[1,2,3,4,5]
```

이것을 익명 함수를 이용하면 다음과 같이 간단하게 처리할 수 있다.

```
>>> list( filter(lambda x:x>0, range(-5,6)) )
[1,2,3,4,5]
```

또는 다음과 같이 간단한 함수들을 리스트의 요소로 지정할 경우도 있을 것이다.

```
>>> fl = [lambda x,y:x+y, lambda x,y:x*y]
>>> fl[0](1,2)
3
>>> fl[1](3,4)
12
```

이와 같이 람다 함수의 용도는 보다 간결하게 원하는 기능을 수행할 수 있도록 하는 것이다.

5장 파이썬 내장 함수

파이썬의 내장 함수는 import 하지 않고 즉시 사용 가능한 함수들이다. 내장 함수명은 일종의 키워드로 간주하여야 하며 사용자의 식별자로 사용하는 것은 피하여야 한다.

5.1 기본 입출력과 관계된 함수들

이하 표에서 대괄호 [..]로 표시된 것은 '생략 가능함'을 나타내는 것이다.

[표 1] 기본 입출력과 관련된 파이썬 내장 함수들

함수명	기능
print(x)	객체를 문자열로 표시한다.
input([prompt])	사용자 입력을 '문자열'로 반환한다.
help([x])	x에 대한 도움말을 출력한다.
globals()	전역 변수의 리스트를 반환한다.
locals() 혹은 vars() vars(obj)	지역 변수의 리스트를 반환한다. __dict__ 어트리뷰트를 반환한다. (객체의 내부 변수가 저장된 딕셔너리)
del(x) 혹은 del x	객체를 변수 공간에서 삭제한다.
eval(expr) exec(obj)	값을 구한다. 파이썬 명령을 실행시킨다.
open(filename[,mode]))	파일을 연다

eval()함수는 파이썬 표현식을 실행해서 결과값을 얻는 함수이다.

```
>>> x = 1
>>> eval('x+1')
2
```

반면 exex()함수는 파이썬 프로그램 조각을 입력 받아서 파싱(parsing)한 후 실행시키는 함수이다. 파이썬 코드를 문자열로 넘겨줄 수도 있고 파일 객체를 넘겨줄 수도 있다.

```
>> a=10
>>> exec('b=a+10')
>>> b
20
```

open()함수는 존재하는 파일을 열거나 새로 파일을 생성하여 file객체를 반환해 주는 함수이다.

```
>>> f = open('test.txt') # 존재하는 test.txt 파일을 연다.
```

위와 같이 mode 인자가 생략되면 읽기 모드인 'r' 로 기본 설정된다. 모드는 다음과 같은 것이 있다.

[표 2] 내장함수 open()의 mode 옵션

mode	기능
'r' 'w' 'x' 'a'	읽기 모드로 연다 (기본값) 쓰기 모드로 연다. 기존 내용이 있다면 삭제됨 독점 모드로 파일을 생성한다. 기존 파일이 있다면 오류 발생 쓰기 모드로 연다. 기존 내용에 이어서 첨가하는 모드이다.
'b' 't'	바이너리 모드(파일 내용이 bytes 객체로 반환됨) 텍스트모드(기본값, 파일 내용이 str 객체로 반환됨)
'+'	내용을 갱신하기 위해서 파일을 연다.(읽기/쓰기)

5.2 기본 자료형의 생성과 변환

다음은 기본 자료형의 생성과 변환에 관련된 내장 함수들이다.

[표 2] 기본 자료형의 생성과 변환에 관련된 내장 함수들

함수명	기능
object()	새로운 object (모든 객체의 base)를 생성한다.
bool(obj)	객체의 진리값을 반환한다.
int(obj)	문자열 형태의 숫자나 실수를 정수로 변환한다.

float(obj)	문자열 형태의 숫자나 정수를 실수로 변환한다.
complex(re [, img])	문자열이나 주어진 숫자로 복소수를 생성한다.
str(obj)	객체를 출력할 수 있는 문자열로 반환한다.
list(seq)	시퀀스형을 받아서 같은 순서의 리스트로 만들어 반환한다,
tuple(seq)	시퀀스형을 받아서 같은 순서의 튜플로 만들어 반환한다,
range(stop) range(start,stop[,step])	0부터 stop-1 까지의 sequence 반환 start부터 stop-1 까지 (step은 간격) sequence 반환
set(seq) frozenset()	시퀀스형을 받아서 같은 순서의 집합(set)으로 만들어 반환한다,
bytes() bytearray()	(immutable) (mutable)
memoryview()	
dict(**kwarg)	시퀀스형을 받아서 딕셔너리로 만들어 반환한다,

(음영진 부분은 sequence 형이다.)

다음은 기본 자료형의 정보를 얻는 내장 함수들이다.

[표 2] 기본 자료형의 정보를 얻는 내장 함수들

함수명	기능
type(obj)	객체의 형을 반환한다.
dir(obj)	객체가 가진 함수와 변수들을 리스트 형태로 반환한다.
repr(obj) ascii(obj)	evla()함수로 다시 객체를 복원할 수 있는 문자열 생성 repr()과 유사하나 non-ascii 문자는 escape한다.(?)
id(obj)	객체의 고유번호(int형)을 반환한다.
hash(obj)	객체의 해시값(int형)을 반환. (같은 값이면 해시도 같다.)
chr(num) ord(str)	ASCII 값을 문자로 반환 한 문자의 ASCII 값을 반환
isinstance(obj, className)	객체가 클래스의 인스턴스인지를 판단한다.
issubclass(class, classinfo)	class가 classinfo 의 서브클래스일때 True 반환

classmethod() staticmethod()	클래스 메서드로 지정 (@classmethod) 정적 메서드로 지정 (@staticmethod)
callable(obj)	obj가 호출 가능한 객체면 True반환 (ver 3.2에서 다시 도입)
getattr(obj, name) setattr(obj, name, val) delattr(obj, name) hasattr(obj, name)	obj의 attribute (name) 를 얻는다. obj의 attribute (name) 를 설정한다. obj의 attribute (name) 를 삭제한다.

5.3 열거형의 정보를 얻는 내장 함수들

다음은 열거형(sequence)의 정보를 얻는 내장 함수들이다.

[표 1] 열거형의 정보를 얻는 내장 함수들

함수명	기능
len(seq)	시퀀스형을 받아서 그 길이를 반환한다.
iter(obj [,sentinel]) next(iterator)	객체의 이터레이터(iterator)를 반환한다. 이터레이터의 현재 요소를 반환하고 포인터를 하나 넘긴다.
enumerate(iterable, start=0)	이터러블에서 enumerate 형을 반환한다.
sorted(iterable[,key][,reverse])	정렬된 *리스트*를 반환
reversed(seq)	역순으로 된 *iterator*를 반환한다.
filter(func, iterable)	iterable의 각 요소 중 func()의 반환값이 참인 것만을 묶어서 이터레이터로 반환.
map(func, iterable)	iterable의 각 요소를 func()의 반환값으로 매핑해서 이터레이터로 반환.

iter()은 seq형으로부터 이터레이터를 반환하는 함수이다. next()는 이터레이터의 현재 요소를 반환하고 이터레이터의 포인터를 그 다음 요소로 넘기는 함수이다.

enumerate()은 시퀀스로부터 enumerate형을 반환한다. 이것은 인덱스와 해당 값의 튜플을 반환하는 이터레이터를 가진다.

```
>>> seasons = ['spring','summer','fall','winter']
>>> list(enumerate(seasons))
[(0, 'spring'), (1, 'summer'), (2, 'fall'), (3, 'winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'spring'), (2, 'summer'), (3, 'fall'), (4, 'winter')]
```

보통은 for 문과 같이 사용되어서 반복문 안에서 이터러블의 요소값 뿐만 아니라 그 인덱스 정보도 필요할 경우 편리하게 사용할 수 있다.

```
>>> seasons=['spring','summer','fall','winter']
>>> for i,v in enumerate(seasons):
  ...:    print("%s:%s"%(i,v))

0:spring
1:summer
2:fall
3:winter
```

filter(func, iterable)는 함수와 이터러블을 입력으로 받아서 이터러블의 요소가 하나씩 함수에 인수로 전달될 때, 참을 반환시키는 것만을 따로 모아서 이터레이터로 반환하는 함수이다.

다음 예를 살펴보자.

```
def positive(l):
    result = []
    for i in l:
        if i > 0:
            result.append(i)
    return result

>>> print(positive([1,-3,2,0,-5,6]))
[1, 2, 6]
```

위의 positive함수는 리스트를 입력값으로 받아서 각각의 요소를 판별해 양수값만 따로 리스트에 모아 그 결과값을 돌려주는 함수이다.

filter()함수를 이용하면 아래와 같이 똑같은 일을 동일하게 구현할 수 있다.

```
>>> def pos(x):
...     return x > 0
>>> print(list(filter(pos, [1,-3,2,0,-5,6])))
[1, 2, 6]
```

filter() 함수는 첫 번째 인수로 함수명을, 두 번째 인수로는 그 함수에 차례로 들어갈 이터러블(리스트, 터플, 문자열 등)을 받는다. filter 함수는 두 번째 인수인 이터러블의 각 요소들이 함수에 들어갔을 때 리턴값이 참인 것만을 이터레이터로 묶어서 돌려준다. 위의 예에서는 1, 2, 6 만이 양수로 x > 0 이라는 문장이 참이 되므로 [1, 2, 6]이라는 결과 값을 돌려주게 된다.

익명함수(lambda)를 쓰면 더욱 간편하게 쓸 수 있다.

```
>>> list(filter(lambda x: x > 0, [1,-3,2,0,-5,6]))
[1, 2, 6]
```

filter()와 유사한 map() 함수가 있다. 이것은 함수와 이터러블을 입력으로 받아서 각각의 요소가 함수의 입력으로 들어간 다음 나오는 출력값을 묶어서 이터레이터로 돌려주는 함수이다. 만약 어떠 리스트의 모든 요소의 제곱값을 갖는 새로운 리스트를 만들고 싶다면 다음과 같이 하면 된다.

```
>>> a = [2, 3, 4, 7, 8, 10]
>>> a2 = list(map(lambda x:x**2, a))
[4,9,16,49,64,100]
```

파이썬 2.x은 filter()와 map()의 결과가 리스트이므로 위 예에서 list함수를 이용하여 리스트로 변환하지 않아도 된다. 파이썬 3.x에서는 이들 함수의 반환값이 리스트가 아닌 반복형(iterable)이므로 이것을 리스트로 만들기 위해서는 list()함수를 명시적으로 사용해야 한다.

5.4 연산을 수행하는 내장 함수들

다음 표는 산술/논리 연산에 관련된 내장 함수들이다.

[표 1] 산술/논리 연산에 관련된 내장 함수들

hex(n) oct(n) bin(n)	정수 n의 16진수 값을 구해서 '문자열'로 반환한다. 정수 n의 8진수 값을 구해서 '문자열'로 반환한다. 정수 n의 2진수 값을 구해서 '문자열'로 반환한다.
abs(n)	절대값을 구한다. n이 복소수라면 크기를 구한다.
pow(x,y[,z])	거듭제곱을 구한다. pow(x,y)은 x**y 와 같다.
divmod(a,b)	a를 b로 나눈 (몫, 나머지)를 구한다. 튜플 반환.
all(iterable) any(iterable)	iterable 의 모든 요소가 True 일 경우 True를 반환. iterable 의 하나 이상의 요소가 True 일 경우 True를 반환.
max(iterable) max(arg1, arg2, …)	최대값을 구한다.
min(iterable) min(arg1, arg2, …)	최소값을 구한다.
round()	반올림을 한다.

hex(), oct(), bin() 함수는 각각 '0x', '0o', '0b' 로 시작하는 '문자열'로 결과를 반환한다. 이 문자열을 파이썬 값으로 변환하려면 eval()함수를 이용하면 된다.