A Byte of Python
(파이썬 기초 문법)

Swaroop C H <swaroop@swaroopch.com>

Translated by Jeongbin Park <pjb7687@gmail.com>

Modified by Byeong-Chun Shin

- 이 책의 원 저자로 부터 판매되는 것이 **아님**니다.
- http://swaroopch.com/notes/python 에서 원 저자의 글을 내려받을 수 있습니다.

1장. Be a Good Programmer !

2장. 소개

파이썬은 배우기 쉽고, 강력한 프로그래밍 언어입니다. 파이썬은 효율적인 고수준데이터 구조를 갖추고 있으며, 간단하지만 효과적인 객체 지향 프로그래밍 접근법 또한 갖추고 있습니다. 우아한 문법과 동적 타이핑, 그리고 인터프리팅 환경을 갖춘 파이썬은 다양한 분야, 다양한 플랫폼에서 사용될 수 있는 최적의 스크립팅,RAD(rapid application development – 빠른 프로그램 개발) 언어입니다.

파이썬이라는 이름의 유래

파이썬의 창시자 귀도 반 로섬(Guido van Rossum)이 BBC에서 방영되던 "Monty Python's Flying Circus"라는 TV 프로그램의 이름을 따서 지었습니다. 사실 귀도는 뱀이라는 긴 몸으로 다른 동물의 몸을 휘감아 으깨어 부수고 먹어치우는 동물을 딱히 좋아하 지는 않는다고 합니다.

2.1. 파이썬의 특징

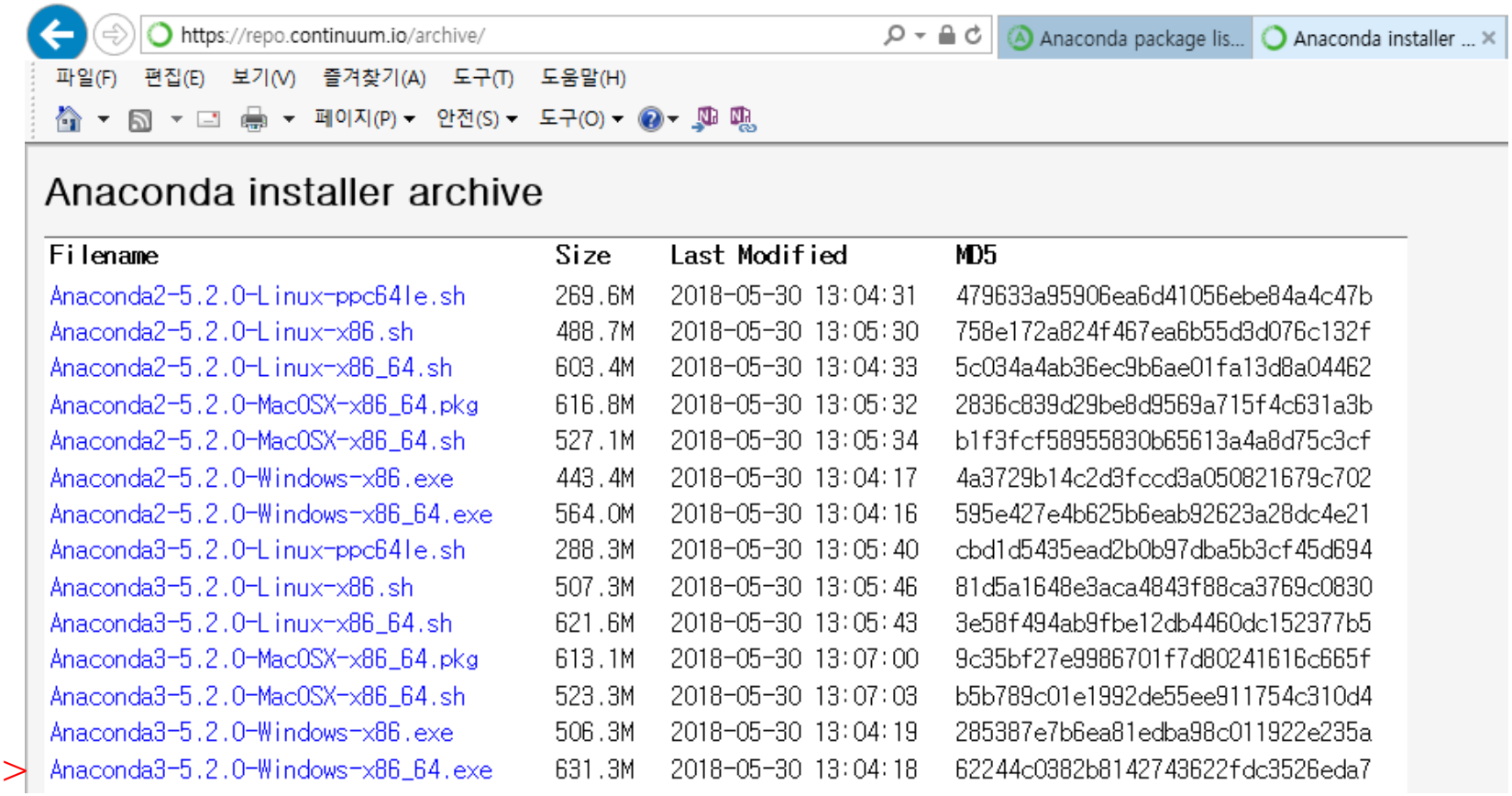
단순함 **배우기 쉬운 언어** **자유, 오픈 소스 소프트웨어** **고수준 언어** **이식성**

객체 지향 언어 **확장성** **포함성** **확장 가능한 라이브러리**

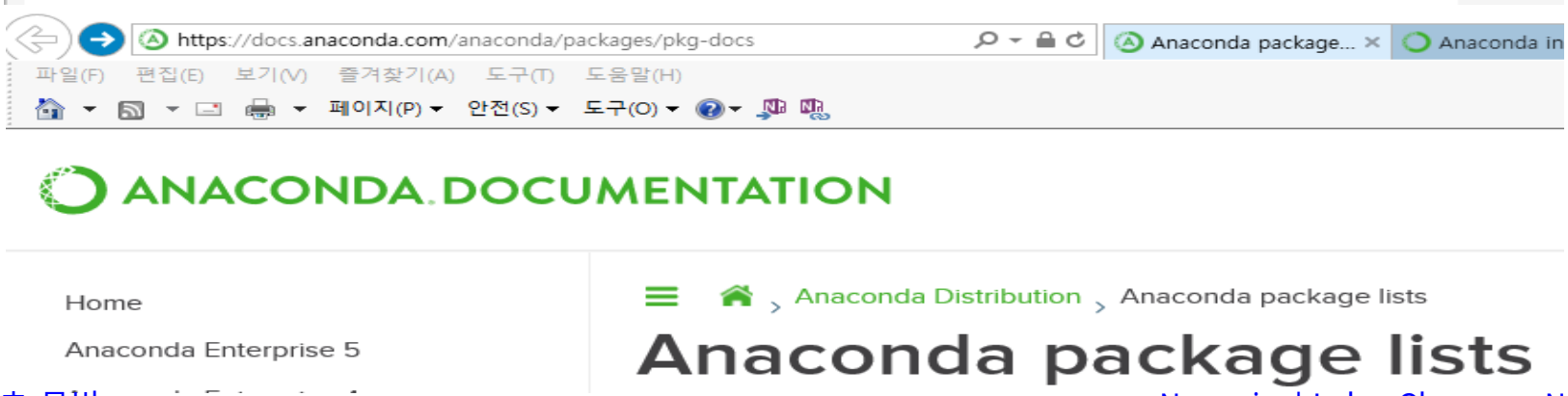
요약 파이썬은 흥미진진하고 강력한 언어입니다. 파이썬의 빠른 성능과 여러 기능들의 조화는 여러분이 재미있고 쉽게 파이썬으로 프로그램을 작성할 수 있도록 해 줍니다.

3장. 설치 : Anaconda Installer Archive

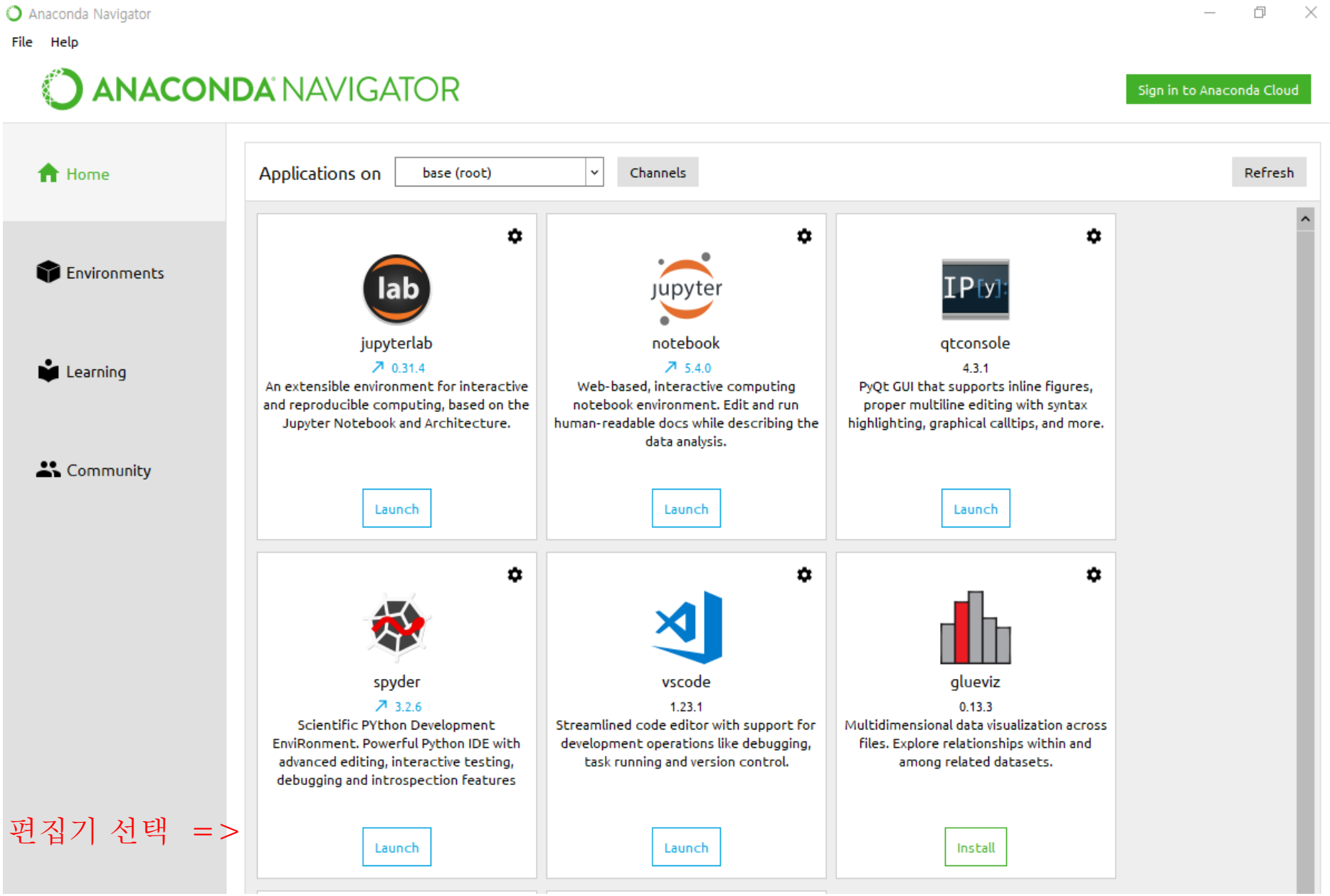
Visit https://repo.continuum.io/archive/



Click =>



4장 편집기 선택 : Spyder 또는 원하는 것



편집기 선택 =>

5장. 기초

5.1. 주석 :

5.2. 리터럴 상수

리터럴 상수는 `5`, `1.23` 과 같은 숫자나, `'This is a string'` 혹은 `"It's a string!"` 과 같은 문자열 등을 말합니다.

5.3. 숫자형

정수형 숫자의 예는 `2` 입니다. 이것은 단순히 2 라는 숫자를 의미하는 것입니다.

부동 소수점 숫자의 예는 `3.23`, `52.3E-4`와 같은 값입니다. `E` 표기법은 `E`뒤의 값이 `10`의 지수임을 나타냅니다. 예를 들어 `52.3E-4` 는 +52.3 * 10^{-4}+라는 값을 의미합니다.

5.4. 문자열

문자열이란 _문자_의 _나열_을 뜻합니다. 문자열은 간단하게 말하자면 문자들의 집합입니다. 여 러분은 아마 앞으로 작성하게 될 거의 모든 파이썬 프로그램에서 문자열을 사용하게 될 것입니다. 따라서, 아래 항목들을 주의깊게 살펴보세요.

5.4.1. 작은 따옴표

여러분은 작은 따옴표를 이용하여 문자열을 지정할 수 있습니다. 예를 들어 `'Quote me on this'` 와 같이 하면 됩니다.

모든 공백 문자, 즉 띄어쓰기나 탭 등은 입력한 그대로 유지됩니다.

5.4.2. 큰 따옴표

큰 따옴표로 둘러싸인 문자열은 작은 따옴표로 둘러싸인 문자열과 완전히 동일하게 취급됩니다.

예를 들면, `"What's your name?"` 과 같습니다 (큰 따옴표로 둘러싸인 문자열 안에 작은 따 옴표가 포함되어도 됩니다).

5.4.3. 따옴표 세 개

여러 줄에 걸친 문자열은 세 개의 따옴표로 표현할 수 있습니다 – (""" 또는 ''').
세 개의 따 옴표로 묶여진 문자열 안에서는 작은 따옴표든 큰 따옴표든 마음대로 사용할 수 있습니다.

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked. He said "Bond, James Bond."
'''
```

5.4.4. 문자열 포맷팅 : format method

문자열을 생성하려고 할 때, 종종 다른 정보들을 포함하여 생성하고 싶을 때가 있습니다.

이것을 문자열 포맷팅이라고 하며, 이를 위해 format() 을 이용합니다.

```
age = 20
name = 'Swaroop'

print('{0} was {1} years old when he wrote this book'.format(name, age) )
print( 'Why is {0} playing with that python?'.format(name) )
```

실행 결과:

```
Swaroop was 20 years old when he wrote this book
Why is Swaroop playing with that python?
```

```
name + ' is ' + str(age) + ' years old'
```

이 때 중괄호 내에 주어진 숫자는 생략할 수 있습니다. 다음 예제를 확인하세요.

5.4.5. 문자열 포맷팅 : format method

```
# 소수점 이하 셋째 자리까지 부동 소숫점 숫자 표기 (0.333)
print( '{0:.3f}'.format(1.0/3) )
# 밑줄(_)로 11칸을 채우고 가운데 정렬(^)하기 (___hello___)
print( '{0:_^11}'.format('hello') )
# 사용자 지정 키워드를 이용해 (Swaroop wrote A Byte of Python) 표기
print( '{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python') )
```

실행 결과:

```
0.333
___hello___
Swaroop wrote A Byte of Python
```

지금까지 문자열 포맷팅에 대해 알아보았습니다. 여기서 print 명령은 언제나 주어진 문자열의 끝에 "줄바꿈" 문자 (\n) 을 덧붙인다는 것 또한 기억하세요. 따라서 print 명령을 호출할 때마다 인자로 주어진 내용들은 항상 그 다음 줄에 출력됩니다. 이것을 막기 위해서는, print 명령 뒤에 쉼 표(,)를 붙여주면 됩니다.

```
print( "a", )
print( "b", )
```

실행 결과:

```
a b
```

python 버전 2.7 : print("a") 대신 print "a" 라고 쓸수 있음
이 책에서 괄호를 사용하지 않는 print 명령이라고 하더라도 버전 3.0 이상에서는 괄호 사용해야함.

5.4.6. 이스케이프(Escape) 문자 : \

```
What ' s your name?
```

```
print( "What\'s your name?" )
```

```
print( 'This is the first line. \nThis is the second line.' )
```

```
print( "This is the first sentence. \    # 아래 줄과 연결
 This is the second sentence." )
```

```
print( 'This is the first line. \tThis is the second line.')
```

5.4.7. 순 문자열 : r or R

문자열 내에 포함된 이스케이프 문자 등을 처리하지 않고 그대로 출력하고 싶을 때, 문자열 앞에 r 또는 R 문자를 붙여 *순(Raw)* 문자열임을 표기합니다.

```
print( r"Newlines are indicated by \n" )
```

5.5. 변수

변수는 이름 그대로 _변_할 수 있는 공간을 말하며, 여기에는 무엇이든 저장할 수 있습니다.
변수들은 단순히 정보를 저장할 때 사용되는 컴퓨터의 기억 장치의 한 부분을 가져다가 적당한 이름을 붙여 사용하는 것

5.6. 식별자 이름 짓기

변수 이름은 식별자의 한 예입니다. _식별자_란 _무언가_를 식별하기 위해 주어진 그것의 이름을 말합니다.
식별자를 짓는데는 다음과 같은 규칙이 있습니다.

- 식별자의 첫 문자는 알파벳 문자 (ASCII 대/소문자 혹은 유니코드 문자)이거나 밑줄 (_) 이 어야 합니다.
- 나머지는 문자 (ASCII 대/소문자 혹은 유니코드 문자), 밑줄 (_), 또는 숫자 (0-9)가 될 수 있습니다.
- 식별자는 대/소문자를 구분합니다. 예를 들어, `myname` 과 `myName` 은 *다릅니다*. 전자의 `n` 은 소문자이고, 후자의 `N`은 대문자입니다.
- *올바른* 식별자 이름은 `i` , `name_2_3` 등과 같습니다. *올바르지 않은* 식별자 이름은 `2things` , `this is spaced out` , `my-name` , `>a1b2_c3` 등입니다.

5.7. 자료형

변수는 여러 가지 _자료형_의 값을 담을 수 있습니다. 가장 간단한 자료형의 예는 앞에서 이야기 한 숫자형과 문자열
뒷장에서, 클래스를 이용한 사용자 정의 자료형을 만드는 법 또한 배 우게 될 것입니다.

5.8. 객체

파이썬에서 사용되는 모든 것은 *객체_입니다. "_그것"*"라고 표현하는 대신, "그 *객체*" 라고 말합 니다.

5.14. 들여쓰기

파이썬에서 공백은 중요한 역할을 합니다. 사실, **한 행의 앞에 붙어있는 공백이 정말 중요합니다**.
이것을 _들여쓰기_라 부릅니다. 한 논리적 명령행의 앞에 붙어있는 공백 (빈 칸 혹은 탭)은 논리적 명령행의 들여쓰기 단계를 의미하며, 이것은 한 명령의 범위를 구분하는 데 사용됩니다.

이것은 같은 들여쓰기 단계에 있는 명령들은 *반드시* 같은 들여쓰기를 사용해야 함을 의미합니다.
이러한 같은 들여쓰기를 사용하고 있는 명령들의 집합을 **블록(block)** 이라고 부릅니다.

지금 여러분이 기억하셔야 할 것은 잘못된 들여쓰기는 오류를 일으킨다는 것입니다. 다음 예제를 봅시다.

```
i = 5 # 다음 행에서 오류가 발생합니다! 행 앞에 잘못된 공백이 한 칸 있습니다.
 print 'Value is ', i
print 'I repeat, the value is ', i
```

위 예제를 실행하면 다음과 같이 오류가 발생합니다.

```
  File "whitespace.py", line 5     print 'Value is ', i
    ^ IndentationError: unexpected indent
```

들여쓰기 하는 법

들여쓰기를 할 때에는 공백 4개를 이용하세요.
이것은 파이썬 언어에서 공식적으로 추천하는 방법입니다.
좋은 편집기들은 이 사항을 자동으로 준수합니다.
또, 들여쓰기를 할 때 에는 항상 같은 개수의 공백을 사용해야 한다는 점에 유의하시기 바랍니다.

6장. 연산자와 수식

6.1. 연산자

`+` **(덧셈 연산자)**

두 객체를 더합니다.
`3 + 5` 는 `8` 을 반환합니다. `'a' + 'b'` 는 `'ab'` 를 반환합니다.

`-` **(뺄셈 연산자)**

한 숫자에서 다른 숫자를 뺍니다. 첫 번째 피연산자가 주어지지 않으면, 0으로 간주됩니다.
`-5.2` 는 음수를 표현합니다. `50 - 24` 는 `26` 을 반환합니다.

`*` **(곱셈 연산자)**

두 숫자의 곱 혹은 지정된 숫자만큼 반복된 문자열을 반환합니다. `2 * 3` 은 `6` 을 반환합니다.
`'la' * 3` 은 `'lalala'` 를 반환합니다.

`**` **(거듭제곱 연산자)**

x 의 y제곱을 반환합니다.
`3 ** 4` 는 `81` 을 반환합니다 (이 값은 `3 * 3 * 3 * 3` 과 같습니다).

`/` **(나눗셈 연산자)**

x를 y로 나눈 몫을 반환합니다.

`13 / 3` 은 `4` 를 반환합니다. `13.0 / 3` 은 `4.333333333333333` 을 반환합니다.

`%` **(나머지 연산자)** x를 y로 나눈 나머지를 반환합니다.

`13 % 3` 은 `1` 을 반환합니다. `-25.5 % 2.25` 는 `1.5` 를 반환합니다.

<< (왼쪽 시프트 연산자)

지정된 숫자에 대해 지정된 비트 수 만큼 왼쪽 시프트 연산합니다
(모든 숫자는 메모리상에서 0 또는 1 의 비트로 구성된 이진수로 표현됩니다).
2 << 2 는 8 을 반환합니다. 2 는 이진수로 10 으로 표현됩니다.

>> (오른쪽 시프트 연산자)

지정된 숫자에 대해 지정된 비트 수 만큼 오른쪽 시프트 연산합니다. 11 >> 1 은 5 를 반환합니다.

11 은 이진수로 1011 로 표현됩니다. 이것으로 오른쪽으로 1비트 시프트 연산하면 이진수 101 이 되고, 이것을 정수로 표현하면 5 가 됩니다.

이것을 왼쪽으로 2비트 시프트 연산하면 이진수 1000 이 되고, 이것을 정수로 표현하면 8 이 됩니다.

& (비트 AND 연산자)

비트 AND 연산값을 반환합니다.

```
5 & 3 은 1 을 반환
   101
&  001
=  001 = 1
```

| (비트 OR 연산자)

비트 OR 연산값을 반환합니다.
5 | 3 은 7 을 반환합니다.

```
   101
|  011
=  111 = 7
```

^ (비트 XOR 연산자)

비트 XOR 연산값을 반환합니다.
5 ^ 3 은 6 을 반환합니다.

```
  101
^ 011
= 110 = 6
```

~ (비트 반전 연산자)

숫자 x의 비트 반전 연산값 -(x+1) 을 반환합니다.
~5 는 -6 을 반환합니다.

< (작음)

x가 y보다 작은지 여부를 반환합니다. 모든 비교 연산자는 True (참) 또는 'False (거짓)'을 반환합니다.
각 반환값의 첫글자는 대문자라는 점에 유의하세요.
5 < 3 는 False 를 반환합니다. 3 < 5 는 True 를 반환합니다.

다음과 같이 여러 숫자에 대해 한꺼번에 비교 연산을 수행할 수 있습니다.
3 < 5 < 7 은 True 를 반환합니다.

> (큼)

x 가 y 보다 큰지 여부를 반환합니다.
5 > 3 은 True 를 반환합니다.
만약 두 피연산자가 모두 숫자라면, 같은 숫자형으로 변환된 후 크기를 비교합니다.
피연산자가 숫자형이 아닐 경우, 항상 False 를 반환합니다.

⇐ **(작거나 같음)**

x = 3; y = 6; x ⇐ y 는 True 를 반환합니다.

〉= **(크거나 같음)**

x = 4; y = 3; x 〉= 3 는 True 를 반환합니다.

== **(같음)**

x = 2; y = 2; x == y 는 True 를 반환합니다.

x = 'str'; y = 'stR'; x == y 는 False 를 반환합니다.

x = 'str'; y = 'str'; x == y 는 True 를 반환합니다.

!= **(같지 않음)**

x = 2; y = 3; x != y 는 True 를 반환합니다.

not **(불리언 NOT 연산자)**

x 가 True 라면, False 를 반환합니다. x 가 False 라면, True 를 반환합니다.

and **(불리언 AND 연산자)**

x and y 를 계산할 경우,
x 가 False 이면 무조건 False 를 반환하며
x 가 True 이면 y 가 True 일때만 True

이것을 단축 계산(short-circuit evalulation)이라고 부릅니다.

or **(불리언 OR 연산자)**

x 가 True 이면 True 가 반환되며, False 이면 y 와의 or 연산값을 반환합니다.
여기서도 단축계산 이 사용됨

6.2. 연산 및 할당 연산자

```
a = 2
a = a * 3
```

```
a = 2
a *= 3
```

6.3. 연산 순서

`2 + 3 * 4`와 같은 수식을 계산한다고 합시다. 덧셈이 먼저일까요, 곱셈이 먼저일까요?

6.4. 연산 순서 변경

괄호를 사용하여 수식을 좀 더 읽기 쉽게 할 수 있습니다. 예를 들어, `2 + (3 * 4)` 라고 쓰면 `2 + 3 * 4` 로 쓰는 것에 비해 연산자 순서를 잘 모르는 사람도 쉽게 읽을 수 있을 것입니다.

6.5. 같은 연산 순서를 가질 경우

기본적으로 연산자는 왼쪽에서 오른쪽으로 차례대로 계산됩니다.

즉, 같은 연산 순서를 가진 연산 자들의 경우 왼쪽에서 오른쪽으로 순서대로 계산됨을 의미합니다.

예를 들어, `2 + 3 + 4` 는 `(2 + 3) + 4` 와 같이 계산됩니다.

다만, 할당 연산자와 같은 몇몇 특별한 연산자들은 오른쪽에서 왼 쪽으로 계산됩니다.

예를 들어 `a = b = c` 는 `a = (b = c)` 와 같이 계산됩니다.

7장. 흐름 제어

7.1. if 문

if 문은 조건을 판별할 때 사용됩니다. **if (만약)** 조건이 참이라면, _if 블록_의 명령문을 실행하며 **else (아니면)** _else 블록_의 명령문을 실행합니다. 이 때 else 조건절은 생략이 가능합니다.

예제 (if.py): 콜론 : 은 다음 줄부터 새로운 블록이 실행된다는 의미

```
number = 23
guess = int(raw_input('Enter an integer : '))
if guess == number:
    # New block starts here     print 'Congratulations, you guessed it.'
    print '(but you do not win any prizes!)'     # New block ends here
elif guess < number:
    # Another block     print 'No, it is a little higher than that'
    # You can do whatever you want in a block ... else:
    print 'No, it is a little lower than that'
    # you must have guessed > number to reach here

print 'Done' # This last statement is always executed,
# after the if statement is executed.
```

실행 결과:

```
$ python if.py Enter an integer : 50
No, it is a little lower than that
```

elif 와 else 문을 사용할 경우, 논리적 명령행의 마지막에는 항상 콜론이 붙어 있어야 하며 그 다음 줄에는 다른 들여쓰기 단계로 시작되는 새로운 명령문 블록이 시작되어야 합니다.

7.2. while 문

while 문은 특정 조건이 참일 경우 계속해서 블록의 명령문들을 반복하여 실행할 수 있도록 합니 다.
while 문에는 else 절이 따라올 수 있습니다.

예제 (`while.py`):

```
number = 23
running = True

while running:
    guess = int(raw_input('Enter an integer : '))

    if guess == number:
        print 'Congratulations, you guessed it.'
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print 'No, it is a little higher than that.'
    else:
        print 'No, it is a little lower than that.'
else:
    print 'The while loop is over.'
    # Do anything else you want to do here

print 'Done'
```

7.3. for 루프

for..in 문은 객체의 열거형(Sequence)을 따라서 반복하여 실행할 때 사용되는 파이썬에 내장된 또 하나의 반복문으로, 열거형에 포함된 각 항목을 하나씩 거쳐가며 실행합니다.
열거형이란 여러 항목이 나열된 어떤 목록을 의미한다고 생각하시기 바랍니다.

예제 (`for.py`):

```
for i in range(1, 5):
    print i
else:
    print 'The for loop is over'
```

실행 결과:

```
$ python for.py
1
2
3
4

The for loop is over
```

range(,) 함수 : 이 함수는 첫 번째 숫자 이상, 그리고 두 번째 숫자 미만까지의 숫자 목록(리스트)을 반환합니다
range(1,5) 는 리스트 [1, 2, 3, 4] 를 반환
range 는 1씩 증가하는 숫자의 리스트를 반환합니다.
range 에 세 번 째 숫자를 입력하면, 이 세 번째 숫자만큼씩 증가하는 숫자들의 리스트를 얻을 수 있습니다

range(1,5,2) 는 [1,3] 을 반환합니다.

range 함수는 호출될 때 해당 범위 내의 모든 숫자를 한번에 생성하여 반환해 주기 때문에, 매 우 큰 숫자 범위를 지정해 주면 시간이 오래 걸릴 수 있습니다.

따라서 매우 큰 범위의 숫자를 다 룰 때는 한번에 하나씩 숫자를 생성하도록 하는 쪽이 낫습니다.
이 경우 for 문에 xrange() 를 대신 사용할 수 있습니다.

또한, 추가로 else 절을 포함시켜 줄 수 있습니다. 이것이 포함되면, break 문으로 루프를 강제 로 빠져나오지 않는 한 루프를 다 돌고 난 뒤에는 이 절이 항상 실행되게 됩니다.

7.4. break 문

break 문은 루프 문을 강제로 빠져나올 때, 즉 아직 루프 조건이 `False`가 되지 않았거나 열거형 의 끝까지 루프가 도달하지 않았을 경우에 루프 문의 실행을 강제로 정지시키고 싶을 때 사용됩니다.

중요한 점은 만약 여러분이 break 문을 써서 for 루프나 while 루프를 빠져나왔을 경우, 루프에 딸린 else 블록은 **실행되지 않습니다**.

예제 (`break.py`):

```
while True:
    s = raw_input('Enter something : ')
    if s == 'quit':
        break
    print 'Length of the string is', len(s)
print 'Done'
```

7.5. continue 문

continue 문은 현재 실행중인 루프 블록의 나머지 명령문들을 실행하지 않고 곧바로 다음 루프로 넘어가도록 합니다.

예제 (continue.py):

```
while True:
    s = raw_input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print 'Too small'
        continue
    print 'Input is of sufficient length'
    # Do other kinds of processing here...
```

8장. 함수

함수는 재사용 가능한 프로그램의 조각을 말합니다. 이것은 특정 블록의 명령어 덩어리를 묶어 이 름을 짓고,
그 이름을 프로그램 어디에서건 사용함으로써 그 블록에 포함된 명령어들을 몇번이고 다시 실행할 수 있게 하는 것입니다.
이를 보고 함수를 **호출한다** 라고 합니다. 사실 우리는 이미 앞 에서 `len` 이나 `range` 와 같은 많은 내장 함수들을
사용해 왔습니다.

함수는 `def` 키워드를 통해 정의됩니다. `def` 뒤에는 함수의 **식별자** 이름을 입력하고, 괄호로 감 싸여진 함수에서
사용될 인자(arguments)의 목록을 입력하며 마지막으로 콜론을 입력하면 함수 의 정의가 끝납니다.
새로운 블록이 시작되는 다음 줄부터는 이 함수에서 사용될 명령어들을 입력 해 줍니다.

예제 (`function1.py` 로 저장합니다):

```
def say_hello():     # block belonging to the function
    print 'hello world' # End of function

say_hello() # call the function
say_hello() # call the function again
```

실행 결과:

```
$ python function1.py
hello world
hello world
```

8.1. 함수와 매개 변수

함수를 정의할 때 매개 변수를 지정할 수 있습니다. 매개 변수란 함수로 넘겨지는 값들의 이름을 말하며, 함수는 이 값들을 이용해 무언가를 **할** 수 있습니다. 매개 변수는 변수와 거의 같이 취급되지만,

이 때 함수를 정의할 때 주어진 이름을 **매개 변수** 라 부르고, 함수에 넘겨준 값들을 **인자** 라 부릅니다.

예제 (function_param.py 로 저장하세요):

```
def print_max(a, b):
    if a > b:
        print a, 'is maximum'     elif a == b:
        print a, 'is equal to', b     else:
        print b, 'is maximum'

# directly pass literal values
print_max(3, 4)

x = 5
y = 7

# pass variables as arguments
print_max(x, y)
```

실행 결과:

```
$ python function_param.py
4 is maximum
7 is maximum
```

8.2. 지역 변수

여러분이 정의한 함수 안에서 변수를 선언하고 사용할 경우, 함수 밖에 있는 같은 이름의 변수들 과 함수 안에 있는 변수들과는 서로 연관이 없습니다.
이러한 변수들을 함수의 **지역(local)** 변수라고 하며, 그 범위를 변수의 **스코프(scope)** 라고 부릅니다.
모든 변수들은 변수가 정의되는 시점에서의 블록을 스코프로 가지게 됩니다.

예제 (`function_local.py` 로 저장하세요):

```
x = 50

def func(x):
    print 'x is', x
    x = 2
    print 'Changed local x to', x

func(x)
print 'x is still', x
```

실행 결과:

```
$ python function_local.py
x is 50 Changed local x to 2
x is still 50
```

8.3. global 문

함수나 클래스 내부에서 상위 블록에서 선언된 변수의 값을 변경하고 싶을 경우, 파이썬에게 이 변수를 앞으로 지역 변수가 아닌 **전역(global)** 변수로 사용할 것임을 알려 주어야 합니다.

global 문을 사용하지 않으면, 함수 외부에서 선언된 변수의 값을 함 수 내부에서 변경할 수 없습니다.

함수 안에서 동일한 이름으로 선언된 변수가 없을 경우, 함수 밖의 변수값을 함수 안에서 읽고 변 경할 수도 있습니다.

예제 (function_global.py 로 저장하세요):

```
x = 50
def func():
    global x

    print 'x is', x
    x = 2
    print 'Changed global x to', x

func()
print 'Value of x is', x
```

실행 결과:

```
$ python function_global.py
x is 50 Changed global x to 2
Value of x is 2
```

8.4. 기본 인수값

어떤 특별한 경우, 함수를 호출할 때 인수를 **선택적으로** 넘겨주게 하여 사용자가 값을 넘겨주지 않으면 자동으로 기본값을 사용하도록 하는 것이 편할 때가 있습니다.
함수를 선언할 때 원하는 매개 변수 뒤에 대입 연산자 (=)와 기본값을 입력하 여 기본 인수값을 지정합니다.

이 때, 기본 인수값은 반드시 상수이어야 합니다. 좀 더 정확히 말하자면, 불변값이어야 합니다.

예제 (function_default.py 로 저장하세요):

```
def say(message, times=1):
    print message * times

say('Hello')
say('World', 5)
```

실행 결과:

```
$ python function_default.py
Hello
WorldWorldWorldWorldWorld
```

함수 say 는 지정된 숫자 만큼 문자열을 반복하여 출력하는 합수입니다.
숫자를 지정 하지 않으면, 기본값이 적용되어, 문자열이 한 번 출력됩니다.

프로그램에서 처음 say 를 호출할 때에는 함수에 문자열만 넘겨 주어 한번 출력하게 합니다.
두 번 째 호출에서는 문자열과 인수 5 를 넘겨 주어 함수가 문자열을 5번 반복하여 말(say)하게 합니다.

매개 변수 목록에서 마지막에 있는 매개 변수들에만 기본 인수값을 지정해 줄 수 있습니다.
즉, 기본 인수값을 지정하지 않은 매개 변수의 앞에 위치한 매개 변수에 기본 인수값을 지정할 수 없습니다.

이것은 함수 를 호출할 때 매개 변수의 위치에 맞춰서 값이 지정되기 때문 입니다.
예를 들어, def func(a, b=5) 는 옳은 함수 정의이지만 def func(a=5, b) 는 **옳지 않습니다**.

8.5. 키워드 인수

여러 개의 매개 변수를 가지고 있는 함수를 호출할 때, 그 중 몇 개만 인수를 넘겨주고 싶을 때가 있습니다.
이때 매개 변수의 이름을 지정하여 직접 값을 넘겨줄 수 있는데 이것을 **키워드 인수** 라 부릅니다.
함수 선언시 지정된 매개 변수의 순서대로 값을 넘겨주는 것 대신, 매개 변수의 이름 (키 워드) 를 사용하여
각각의 매개 변수에 인수를 넘겨 주도록 지정해 줍니다.

키워드 인수를 사용하는 데 두가지 장점이 있습니다. 첫째로, 인수의 순서를 신경쓰지 않고도 함 수를 쉽게 호출할 수 있는 점입니다.
둘째로는, 특정한 매개 변수에만 값을 넘기도록 하여 나머지 는 자동으로 기본 인수값으로 채워지게 할 수 있습니다.

예제 (`function_keyword.py` 로 저장하세요):

```
def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

실행 결과:

```
$ python function_keyword.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

8.6. VarArgs 매개 변수

가끔 함수에 *임의의* 개수의 매개 변수를 지정해주고 싶을 때가 있습니다.
아래 예제와 같이 별 기호를 사용하여 임의의(**Var**iable) 개수의 인수(**Arg**uments) 를 표현합니다.

```
def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number
    for key in keywords:
        count += keywords[key]
    return count

print total(10, 1, 2, 3, vegetables=50, fruits=100)
```

실행 결과:

```
$ python function_varargs.py
166
```

앞에 별 기호가 달린 매개 변수, 예를 들어 `*param` 과 같이 매개 변수를 지정해 주면 함 수에 넘겨진 모든 위치 기반 인수들이 'param' 이라는 이름의 튜플로 묶여서 넘어옵니다.

앞에 별 두 개가 달린 매개 변수, 예를 들어 `**param` 과 같이 매개 변수를 지정 해 주면 함수에 넘겨진 모든 키워드 인수들이 'param' 이라는 이름의 사전으로 묶여서 넘어옵니다.

8.7. return 문

return 문은 함수로부터 되돌아(return) 나올 때, 즉 함수를 빠져 나올 때 사용됩니다.
이 때 return 값처럼 값을 지정해 주면, 함수가 종료될 때 그 값을 반환하도록 할 수 있습니다.

```
def maximum(x, y):
    if x > y:
        return x

    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print maximum(2, 3)
```

실행 결과:

```
$ python function_return.py
3
```

8.8. DocString

파이썬은 설명(Documentation) 문자열(String) 이라고 불리우는, 짧게 줄여서 DocStrings라 불리우는 편리한 기능을 가지고 있습니다.
DocString은 프로그램을 알아보기 쉽게 해 주고, 설명서를 작성할 때 유용하게 사용될 수 있는 중요한 도구

예제 (function_docstring.py 로 저장하세요):

```
def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    # convert to integers, if possible
    x = int(x)
    y = int(y)

    if x > y:
        print(x, 'is maximum')
    else:

        print(y, 'is maximum')

print_max(3, 5)
print(print_max.__doc__ )
```

실행 결과:

```
5 is maximum
Prints the maximum of two numbers.


    The two values must be integers.
```

함수에 포함된 첫 문자열은 함수의 **DocString** .
DocString은 모듈 과 클래스 에도 똑같이 적용

첫째줄의 첫문자는 대문자로, 마지막 문자는 마침표로 끝
두번째 줄은 비워 두고,
세번째 줄부터는 어떤 기능을 하는지에 대해 상세하게 작성

print_max 함수의 DocString은 함수의 doc 속성을 통해 접근 (밑줄이 **두 개** 임을 다시한번 확인).

doc 은 함수 객체가 갖고 있는 기본 속성입니다.

파이썬에서 help() 가 하는 일은 주어진 대상의 doc 속성을 가져와 화면에 보여주는 것

위에서 만든 함수에 대해서도 마찬가지로 동작

help(print_max) 라고 한 줄 추가해 보시기 바랍니다.

help 창을 닫으려면 q 키를 누르세요.

9장. 모듈(module)

코드를 재사용하는 방법에 여러 함수들을 한꺼번에 불러들여 재사용하는 방법

.py 확장자를 가진 파 일을 하나 만들고 그 안에 함수들과 변수들을 정의해 두는 것

모듈을 작성하는 또 한 가지 방법은 여러분이 현재 사용중인 파이썬 인터프리터를 만드는데 사용되는 프로그래밍 언어로 모듈을 작성 하는 것입니다.
표준 파이썬 인터프리터를 사용 중인 경우 C 언어[1]를 이용하여 모듈을 작성하고 컴파일하면 파이썬에서 사용

다른 프로그램에서 **import** 명령을 통해 모듈을 불러와 사용할 수 있습니다.
파이썬 표준 라이브러리 또한 동일한 방법을 통해 이용이 가능합니다.

예제 (module_using_sys.py 로 저장하세요):

```
import sys

print('The command line arguments are:')
for i in sys.argv:
    print i

print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

동작 원리: 표준 라이브러리 중 하나인 sys 모듈을 import 문을 통해 불러왔습니다. 이것 은 단순히 파이썬에게 이 모듈을 앞으로 사용할 것이라고 알려주는 과정이라고 생각하시면 편합니다.
sys 모듈은 파이썬 인터프리터와 인터프리터가 실행중인 환경, 즉 시스템 (**sys**tem)에 관련된 기능들이 담겨 있습니다.

실행 결과:

```
$ python module_using_sys.py we are arguments
The command line arguments are:
module_using_sys.py   # sys.argv[0]
we      # sys.argv[1]
are     # sys.argv[2]
arguments   # sys.argv[3]

The PYTHONPATH is ['/tmp/py',

'/Library/Python/2.7/site-packages',
'/usr/local/lib/python2.7/site-packages']
# many entries here, not shown here
```

9.1. 바이트 컴파일된 .pyc 파일

모듈을 불러오는 것은 상대적으로 무거운 작업이기 때문에, 파이썬은 약간의 트릭을 사용해서 좀 더 이 과정을 빠르게 수행할 수 있게 합니다.
그것은 바로 **.pyc`의 확장자를 가지는 *바이트 컴파일*된, 일종의 중간 단계의 파일을 만들어 두는 것**

이러한 `.pyc 파일은 다른 프로그 램에서 그 모듈을 다시 필요로 할 때 사용되며, 이 경우 모듈을 읽어들이는 데 필요한 몇가지 선행 작업을 수행하지 않아도 되게 되어 더 빨리 모듈을 불러올 수 있습니다.

.pyc 파일은 .py 파일이 저장되어 있는 디렉토리에 새롭게 생성됩니다.

9.2. from ··· import 문

매번 sys. 를 입력하지 않고서도 argv 변수를 프로그램에서 곧바로 불러와서 사용할 수도 있습 니다. 이런 경우, from sys import argv 와 같은 구문을 이용합니다.

하지만 식별자 이름간의 충돌을 피하고 프로그램을 좀 더 읽기 쉽게 작성하기 위해서, 가능하면 이 렇게 사용하는 경우를 **피하고** import 문을 사용하기를 권합니다.

예제:

```
from math import sqrt
print("Square root of 16 is", sqrt(16) )
```

9.3. 모듈의 name 속성(attribution)

모든 모듈은 이름을 갖고 있으며, 모듈 내에 포함된 명령을 통해 모듈의 이름을 알아올 수 있습니 다.
이 속성은 현재 모듈이 불러들여져서 사용되고 있는지 아니면 인터프리터에서 곧바로 실행된 것인지를 구문하는데 편리
이러한 속성을 통해 모듈이 외부로부터 불러들 여졌을 때와 곧바로 실행되었을 때에 각각 다르게 처리

모든 파이썬 모듈은 name 속성을 가지고 있습니다. 만약 그 이름이 *main* 일 경우,
이 것은 모듈이 사용자로부터 직접 실행된 것임을 의미하며 따라서 이에 맞는 적절한 처리를 해 줄 수 있습니다.

예제 (module_using_name.py 로 저장하세요):

```
if __name__ == '__main__':
    print 'This program is being run by itself'
else:
    print 'I am being imported from another module'
```

실행 결과:

```
$ python module_using_name.py
This program is being run by itself

$ python
>>> import module_using_name
I am being imported from another module
>>>
```

9.4. 새로운 모듈 작성하기

모듈을 작성하는 것은 쉽습니다. 모든 파이썬 프로그램은 곧 모듈이기 때문입니다.

예제 (mymodule.py 로 저장하세요):

```
def say_hi():
    print 'Hi, this is mymodule speaking.'

__version__ = '0.1'
```

위 예제 프로그램 파일이 아래 예제 프로그램과 같은 디렉토리에 있 거나 혹은 sys.path 중 하나에 있어야 함

모듈 불러오기 예제 (mymodule_demo.py 로 저장하세요):

```
import mymodule

mymodule.say_hi()
print('Version', mymodule.__version__ )
```

실행 결과:

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

모듈의 구성 요소에 접근하는 데에도 마침표를 이용하여 접근

from..import 구문 사용 (`mymodule_demo2.py`):

```
from mymodule import say_hi, __version__

say_hi()
print 'Version', __version__
```

실행한 결과는 mymodule_demo.py 의 결과와 같습니다.

```
from mymodule import *
```

say_hi와 같이 모듈에 포함된 모든 공개된 이름들을 불러옵니다.
그러나 밑줄 두 개로 시작하는 version 과 같은 이름들은 불러오지 않습니다.

주의: 가능하면 from mymodule import * 처럼 사용하는 것을 피함

Zen of Python (파이썬 정신)

파이썬의 여러 지향점 중 하나는 "명시적인 것이 암시적인 것 보다 낫다 (Explicit is better than Implicit)" 입니다.

9.5. dir 내장 함수(built in function)

dir 내장 함수를 이용하여 객체에 정의되어 있는 식별자들의 목록을 불러올 수 있습니다.
예를 들어, 모듈의 경우 함수와 클래스 및 변수들의 식별자 이름이 정의되어 있을 것입니다.

```
$ python
>>> import sys

# sys 모듈 내에 선언된 속성들의 식별자 이름 목록

>>> dir(sys)
['__displayhook__', '__doc__', 'argv', 'builtin_module_names',
'version', 'version_info']

# 실제로는 너무 길어 여기에는 몇 개만 적었음

# 현재 모듈에 선언된 속성들의 식별자 이름 목록
>>> dir()
['__builtins__', '__doc__',
'__name__', '__package__']

# 새로운 변수 'a' 생성
>>> a = 5

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a']
```

```
# 식별자 이름 제거
>>> del a

>>> dir()
['__builtins__', '__doc__', '__name__',
 '__package__']

>>> dir(str)  # str클래스의 속성
```

9.6. 패키지(package) : 모듈을 구성하는 계층구조

그렇다면 모듈은 어디에 포함되는 것일까요? 파이썬에서는 패키지라는 단위가 이에 해당됩니다.

패키지란 그냥 단순한 폴더입니다만, 파이썬에게 이 폴더는 파이썬 모듈을 담고 있다는 것을 알려 주는 역할을 하는 `init.py` 라는 특별한 파일을 한 개 포함하고 있습니다.

여러분이 'asia', 'africa'라는 하위 패키지를 포함하고 있는 'world' 라는 패키지를 만들고 싶다고 가정해 봅시다.
또한 각각의 하위 패키지는 'india', 'madagascar' 등등의 하위 패키지를 하나씩 더 갖고 있습니다.

이 경우, 아래와 같이 폴더 구조를 만들어 주면 됩니다:

```
- <some folder present in the sys.path>/
    - world/
        - __init__.py
        - asia/
            - __init__.py
            - india/
                - __init__.py
                - foo.py
        - africa/
            - __init__.py
            - madagascar/
                - __init__.py
                - bar.py
```

10장. 자료 구조(data structure)

파이썬에는 네 종류의 자료 구조가 있는데, 각각 _리스트, 튜플, 사전, 집합_입니다.

10.1. 리스트(list)

리스트란 순서대로 정리된 항목들을 담고 있는 자료 구조입니다.

리스트를 정의할 때는 대괄호 `[]` 를 이용해서 파이썬에게 이것이 리스트를 의미한다는 것을 알 려 줍니다.
한번 리스트를 만들어 두면 여기에 새로운 항목을 추가하거나 삭제할 수 있으며, 특정 항목이 존재하는지
검색할 수도 있습니다.
이 때 항목을 추가 및 삭제가 가능하다는 것을 *비정적 (mutable)*이라고 하며,
리스트는 비정적 자료구조로 내부 항목을 변경할 수 있습니다.

10.2. 객체(object)와 클래스(class)에 대한 간단한 소개

리스트는 객체와 클래스가 사용된 한 예입니다.
변수 `i`를 선언하고 `5` 라는 값을 할당 해 주는 것은, `int`라는 **클래스**(또는 타입)의 **객체**(또는 인스턴스) `i` 를 만드는 것
클래스는 *메소드*를 가질 수 있는데, 여기서 메소드란 그 클래스 내에 정의된 고유의 내장 함수 들을 말합니다.
예를 들어, `list` 클래스에 `append` 라는 메소드를 제공하며 이는 리스트에 항목을 한 개 추가할 때 사용
즉 `mylist.append('an item')` 라 하면 리스트 `mylist` 의 마지막에 해당 문자열을 추가
또 클래스는 *필드*를 가질 수 있는데 이것은 단순히 그 클래스 내에 정의된 변수들을 의미합니다.

예를 들면 `mylist.field` 와 같습니다.

예제 (ds_using_list.py 로 저장하세요):

```
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print 'I have', len(shoplist), 'items to purchase.'

print 'These items are:',

for item in shoplist:
    print item,

print '\nI also have to buy rice.'
shoplist.append('rice')
print 'My shopping list is now', shoplist

print 'I will sort my list now'
shoplist.sort()

print 'Sorted shopping list is', shoplist

print 'The first item I will buy is', shoplist[0]
olditem = shoplist[0]
del shoplist[0]
print 'I bought the', olditem
print 'My shopping list is now', shoplist
```

실행 결과:

```
$ python ds_using_list.py
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'car
rot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'ca
rrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'm
ango', 'rice']
```

- print 문 맨 뒤에 추가한 쉼표는 출력될 내용 뒤에 줄바꿈 기호 대신 공백을 출력하도록 지정
- 쉼표는 앞으로 출력될 내용을 같은 줄에 이어서 출력하라고 알려 주는 것

10.3. 튜플(tuples)

튜플은 여러 개의 객체를 모아 담는 데 사용됩니다.
튜플은 리스트와 비슷하지만, 리스트 클래스에 있는 여러가지 기능이 없습니다.
또 튜플은 수정이 불가능하며, 그래서 주로 문자열과 같이 * 비정적*인 객체들을 담을 때 사용됩니다.
튜플은 생략할 수 있는 괄호로 묶인 쉼표로 구분된 여러 개의 항목으로 정의됩니다.
튜플에 저장된 값들은 수정이 불가능하기 때문에, 단순 값들의 목록을 다루는 구문이나 사용자 정 의 함수에서 주로 사용

예제 (ds_using_tuple.py 로 저장하세요):

```
# I would recommend always using parentheses to indicate start and end of tuple
# even though parentheses are optional. # Explicit is better than implicit.
zoo = ('python', 'elephant', 'penguin')
print 'Number of animals in the zoo is', len(zoo)
new_zoo = 'monkey', 'camel', zoo
print 'Number of cages in the new zoo is', len(new_zoo)
print 'All animals in new zoo are', new_zoo
print 'Animals brought from old zoo are', new_zoo[2]
print 'Last animal brought from old zoo is', new_zoo[2][2]
print 'Number of animals in the new zoo is', \
len(new_zoo)-1+len(new_zoo[2])
```

```
Number of animals in the zoo is 3
Number of cages in the new zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python', 'elephant', 'penguin'))
Animals brought from old zoo are ('python', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

10.4. 사전(dictionary)

사전은 이를테면 전화번호부 같은 것인데, 누군가의 이름을 찾으면 그 사람의 주소와 연락처를 알 수 있는 것과 같습니다. 이 때 그 사람의 이름에 해당하는 것을 **키(key)** 라 부르고, 주소와 연락처 등에 해당하는 것을 **값(value)** 이라 부릅니다. 전화번호부에 동명이인이 있을 경우 어떤 정보가 맞는 정보인지 제대로 알아낼 수 없듯이, 사전의 키는 사전에서 유일한 값
사전의 키는 정적 객체(문자열 등등)이어야 하지만, 값으로는 정적 객체나 비정적 객체 모두 사용 할 수 있습니다. 이것을 간단하게 다시 말하면 사전의 키로는 단순 객체만 사용할 수 있다고 표 현합니다.

사전을 정의할 때 키와 값의 쌍은 `d = {key1 : value1, key2 : value2 }` 와 같이 지정해 줍니다. 키와 값은 콜론으로 구분하며 각 키-값 쌍은 쉼표로 구분하고 이 모든 것을 중괄호 `{}`로 묶어 준다는 것

여기서 사전의 키-값 쌍은 자동으로 정렬되지 않습니다. 이를 위해서는 사용하기 전에 먼저 직접 정렬을 해 주어야 합니다.

예제 (ds_using_dict.py 로 저장하세요):

```
# 'ab' is short for 'a'ddress'b'ook
ab = {  'Swaroop'   : 'swaroop@swaroopch.com',
        'Larry'     : 'larry@wall.org',
        'Matsumoto' : 'matz@ruby-lang.org',
        'Spammer'   : 'spammer@hotmail.com'
    }
print "Swaroop's address is", ab['Swaroop']
# Deleting a key-value pair
del ab['Spammer']
print '\nThere are {} contacts in the address-book\n'.format(len(ab) )

for name, address in ab.items():
    print 'Contact {} at {}'.format(name, address)
# Adding a key-value pair
ab['Guido'] = 'guido@python.org'
if 'Guido' in ab:
     print "\nGuido's address is", ab['Guido']
```

실행 결과:

```
Swaroop's address is swaroop@swaroopch.com
There are 3 contacts in the address-book
Contact Swaroop at swaroop@swaroopch.com
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org
Guido's address is guido@python.org
```

10.5. 열거형

열거형들은 리스트, 튜플, 문자열 같은 것입니다. 그러면 열거형이란 무엇이고 열거형에서는 무 엇이 중요할까요?

열거형의 기능은 **멤버십 테스트** (`in` 과 `not in` 연산)와 열거형의 특정 항목을 얻어올 수 있는 *인덱싱 연산*입니다.

또한 리스트, 튜플, 문자열의 세 가지 열거형은 **슬라이스** 연산 기능을 가지고 있는데, 이것은 열거 형의 일부분을 잘라낸(slice) 것을 반환하는 연산, 즉 부분 집합을 반환해 주는 연산입니다.

예제 (`ds_seq.py` 로 저장하세요):

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'
# Indexing or 'Subscription' operation #

print 'Item 0 is', shoplist[0]
print 'Item 1 is', shoplist[1]
print 'Item 2 is', shoplist[2]
print 'Item -1 is', shoplist[-1]
print 'Item -2 is', shoplist[-2]
print 'Character 0 is', name[0]

# Slicing on a list #
print 'Item 1 to 3 is', shoplist[1:3]
print 'Item 2 to end is', shoplist[2:]
print 'Item 1 to -1 is', shoplist[1:-1]
print 'Item start to end is', shoplist[:]
# Slicing on a string #
print 'characters 1 to 3 is', name[1:3]
print 'characters 2 to end is', name[2:]
print 'characters 1 to -1 is', name[1:-1]
print 'characters start to end is', name[:]
```

슬라이스 숫자에 세 번째 인수를 지정해 줄 수 있는데,
이것은 슬라이스 _스텝_에 해당합니다
(기본값은 1 입니다):

```
>>> shoplist = ['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::1]
['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::2]
['apple', 'carrot']
>>> shoplist[::3]
['apple', 'banana']
>>> shoplist[::-1]
['banana', 'carrot', 'mango', 'apple']
```

10.6. 집합

집합은 *정렬되지 않은* 단순 객체의 묶음입니다. 집합은 포함된 객체들의 순서나 중복에 상관없이 객체를 묶음 자체를 필요로 할 때 주로 사용합니다.

집합끼리는 멤버십 테스트를 통해 한 집합이 다른 집합의 부분집합인지 확인할 수 있으며, 두 집 합의 교집합 등을 알아낼 수도 있습니다.

```
>>> bri = set(['brazil', 'russia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('russia')
>>> bri & bric # OR bri.intersection(bric)
{'brazil', 'india'}
```

10.7. 참조

객체를 생성하고 변수에 할당해 줄 때, 사실 실제 객체가 변수에 할당되는 것은 아닙니다! 변수에는 객체의 *참조* 가 할당
참조란, 그 변수의 이름이 여러분의 컴퓨터 메모리 어딘가에 저장되어 있는 실제 객체의 위치를 가리키는 것
이를 객체에 이름을 **바인딩** 한다고 말합니다.
일반적으로는 이에 대해 크게 신경 쓸 필요가 없습니다만, 참조로 인해 발생하는 몇 가지 현상

예제 (ds_reference.py 로 저장하세요):

```
print 'Simple Assignment'
shoplist = ['apple', 'mango', 'carrot', 'banana']
# mylist is just another name pointing to the same object!
mylist = shoplist

# I purchased the first item, so I remove it from the list
del shoplist[0]

print 'shoplist is', shoplist
print 'mylist is', mylist
# Notice that both shoplist and mylist both print
# the same list without the 'apple' confirming that
# they point to the same object

print 'Copy by making a full slice'
# Make a copy by doing a full slice
mylist = shoplist[:]
# Remove first item
del mylist[0]

print 'shoplist is', shoplist
print 'mylist is', mylist
# Notice that now the two lists are different
```

실행 결과:

```
$ python ds_reference.py
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

복잡한 객체 (정수형 제외)의 복사본을 생성하고 싶을 때에는,
슬라이스 연산자를 이용하여 복사본을 생성

단순히 한 변수를 다른 변수에 할당하게 되면
두 변수는 같은 객체를 "참조"

10.8. 문자열에 대한 좀 더 자세한 설명

앞서 문자열에 대해 이미 상세히 다루었지만, 몇 가지 더 알아두면 좋을 것들이 있습니다. 문자열 도 객체이므로 여러 메소드를 가지고 있는데 이를 통해 문자열의 앞 뒤 공백을 제거한다거나 하 는 일들을 할 수 있습니다!

파이썬에서 사용되는 모든 문자열은 `str` 클래스의 객체입니다.

예제 (`ds_str_methods.py` 로 저장하세요):

```
# This is a string object
name = 'Swaroop'

if name.startswith('Swa'):
    print 'Yes, the string starts with "Swa"'

if 'a' in name:     print 'Yes, it contains the string "a"'

if name.find('war') != -1:
    print 'Yes, it contains the string "war"'

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print delimiter.join(mylist)
```

```
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

`startswith` 메소드는 주어진 문자열로 시작하는지 여부
`in` 연산자는 주어진 문 자열이 포함되어 있는지 확인

`find` 메소드는 특정 문자열의 위치를 반환
주어진 문자열을 찾 지 못한 경우 `find` 는 -1 을 반환

`join` 메소드, 이것은 주어진 문자열들을 해당 문자열을 구분자로 하여 결합한 하나의 큰 문자열을 만들어 반환

11장. 실생활 문제 해결: 원문을 참조 하세요

backup_ver1.py 로 저장하세요:

```
import os
import time

# 1. The files and directories to be backed up are # specified in a list.
# Example on Windows: # source = ['"C:\\My Documents"', 'C:\\Code'] # Example on Mac OS X and Linux:
source = ['/Users/swa/notes']
# Notice we had to use double quotes inside the string # for names with spaces in it.

# 2. The backup must be stored in a
# main backup directory
# Example on Windows:
# target_dir = 'E:\\Backup'
# Example on Mac OS X and Linux:
target_dir = '/Users/swa/backup' # Remember to change this to which folder you will be using

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_dir + os.sep + time.strftime('%Y%m%d%H%M%S') + '.zip'

# Create target directory if it is not present
if not os.path.exists(target_dir):
    os.mkdir(target_dir) # make directory

# 5. We use the zip command to put the files in a zip archive
zip_command = "zip -r {0} {1}".format(target,' '.join(source))

# Run the backup
print "Zip command is:"
print zip_command
print "Running:"
if os.system(zip_command) == 0:
```

12장. 객체 지향 프로그래밍

지금까지 프로그램을 작성할 때, 우리는 데이터를 다루는 명령들의 블록인 함수들의 조합으로 프로그램을 구성하였습니다.
이러한 설계 방식을 *절차 지향* 프로그래밍 기법이라고 부릅니다.

이와 달리 데이터와 기능을 객체라고 불리우는 것으로 묶어서 프로그램을 구성하는 또 다른 기법이 있습니다.
이것을 *객체 지향* 프로그래밍 기법이라고 부릅니다.

12.1. self 에 대하여

클래스 메소드는 일반적인 함수와 딱 한 가지 다른 점이 있는데,
그것은 메소드의 경우 매개 변수의 목록에 항상 추가로 한 개의 변수가 맨 앞에 추가되어야 한다는 점입니다.
또한 메소드를 호출할 때 이 변수에는 우리가 직접 값을 넘겨주지 **않으며**, 대신 파이썬이 자동으로 값을 할당합니다.
이 변수에는 현재 객체 *자신의* 참조가 할당되며, 일반적으로 `self` 라 이름을 짓습니다.

이 변수의 이름은 마음대로 지을 수 있지만, `self` 라는 이름을 사용할 것을 *강력히 권합니다*.

파이썬이 `self` 에 어떻게 값을 할당하는 것인지 그리고 정말 값을 직접 할당할 필요가 없는지 궁금할 것입니다.
이해를 돕기 위해 예를 하나 들어 보겠습니다.

여러분이 `MyClass` 라는 클래스를 생성했고,

> 이 클래스의 객체를 `myobject` 라는 이름으로 생성했다고 해 봅시다.
> 이제 이 객체의 메소드를 호출할 때는 `myobject.method(arg1, arg2)` 와 같이 하며,
> 이것은 파이썬에 의해 자동적으로 `MyClass.method(myobject, arg1, arg2)` 의 형태로 바뀌게 됩니 다.
> 이것이 `self` 에 대한 모든 것입니다.

12.2. 클래스

가장 단순한 클래스의 예시가 아래 예제에 나타나 있습니다(oop_simplestclass.py 로 저장 하세요).

```
class Person:
    pass # An empty block

p = Person()
print(p)
```

```
$ python oop_simplestclass.py
<__main__.Person instance at 0x10171f518>
```

pass 문으로 해당 블록이 빈 블록
객체가 실제로 저장된 컴퓨터 메모리의 위치가 함께 반환

12.3. 메소드

```
class Person:
    def say_hi(self):
        print('Hello, how are you?')

p = Person()
p.say_hi()
# The previous 2 lines can also be written as
# Person().say_hi()
```

```
$ python oop_method.py
Hello, how are you?
```

여기서 say_hi 메소드는 아무 매개 변수도 넘겨받지 않지만 함수 정의에 self 를 가지고 있음을 확인

12.4. init 메소드

init 메소드는 클래스가 인스턴스화 될 때 호출됩니다.
따라서 이 메소드는 객체가 생성될 때 여 러가지 **초기화** 명령들이 필요할 때 유용하게 사용됩니다.
여기서 init의 앞과 뒤에 있는 밑줄은 두 번씩 입력해야 한다는 점을 기억하시기 바랍니다.

예제 (oop_init.py 로 저장하세요):

```
class Person:
    def __init__(self, name):
        self.name = name
    def say_hi(self):
        print 'Hello, my name is', self.name
p = Person('Swaroop')
p.say_hi()
# The previous 2 lines can also be written as
# Person('Swaroop').say_hi()
```

```
$ python oop_init.py
Hello, my name is Swaroop
```

먼저 매개 변수 name 을 넘겨 받는 init 메소드를 정의합니다(물론 self`를 포함 하여 정의합니다).
이 때 두 다른 변수의 이름으로 'name' 이라는 동일한 이름을 지정해 주었다는 점에 주목하시기 바랍니다.
이것이 문제가 되지 않는 이유는 하나는 "self" 라 칭해지는 객체에 내장된 것으로써 self.name 의 형태로 사용
또 하나인 name 은 지역 변수를 의미하는 것으로 사용되기 때문입니다.
프로그램 상에서 각각을 완전하게 구분할 수 있으므로, 혼란이 일어나지 않습니다.

위 예제에서 가장 중요한 것은, 우리가 init 메소드를 직접 호출해 주지 않고 클래스로부터 인 스턴스를 생성할 때 괄호 안에 인수를 함께 넘겨 주었다는 점입니다. 이 점이 이 메소드가 좀 특 별하게 다뤄지는 이유입니다.

12.5. 클래스 변수와 객체 변수

클래스 변수 는 공유됩니다. 즉, 그 클래스로부터 생성된 모든 인스턴스들이 접근할 수 있습니다.

클래스 변수는 한 개만 존재하며 어떤 객체가 클래스 변수를 변경하면 모든 다른 인스턴스들에 변 경 사항이 반영됩니다.

객체 변수 는 클래스로부터 생성된 각각의 객체/인스턴스에 속해 있는 변수입니다.

(oop_objvar.py):

```
class Robot:
    """Represents a robot, with a name."""  # DocString called by Robot.doc

    # A class variable, counting the number of robots
    population = 0    # 클래스 변수

    def __init__(self, name):
        """Initializes the data."""
        self.name = name  # 객체 변수
        print "(Initializing {})".format(self.name)

        # When this person is created, the robot
        # adds to the population
        Robot.population += 1

    def die(self):
        """I am dying."""    # DocString called by Robot.die.doc
        print "{} is being destroyed!".format(self.name)

        Robot.population -= 1

        if Robot.population == 0:
            print "{} was the last one.".format(self.name)

        else:

            print "There are still {:d} robots working.".format(Robot.population)
```

```
    def say_hi(self):
        """Greeting by the robot.

        Yeah, they can do that."""
        print "Greetings, my masters call me {}.".format(self.name)
    @classmethod    # 클래스 메소드
    def how_many(cls):
        """Prints the current population."""
        print "We have {:d} robots.".format( cls.population)
```

```
droid1 = Robot("알파")
droid1.say_hi()
Robot.how_many()

droid2 = Robot("베타")
droid2.say_hi()
Robot.how_many()

print "\nRobots can do some work here.\n"
print "Robots have finished their work. \n
          So let's destroy them."

droid1.die()
droid2.die()
Robot.how_many()
```

실행 결과:

```
$ python oop_objvar.py
(Initializing 알파)
Greetings, my masters call me 알파.
We have 1 robots.
(Initializing 베타)
Greetings, my masters call me 베타.
We have 2 robots.
Robots can do some work here.
Robots have finished their work.
So let's destroy them.
알파 is being destroyed!
There are still 1 robots working.
베타 is being destroyed!
베타 was the last one.
We have 0 robots.
```

population 클래스 변수는 Robot.population 과 같이 사용하며 self.population 과 같이 사용하지 않습니다.
반면 객체 변수 name 은 그 객체 안에서 self.name 과 같이 사용 됩니다.

메소드 how_many 는 객체에 소속되어 있지 않고 클래스에 소속되어 있는 메소드입니다.
여기 서 우리가 해당 클래스의 어떤 부분까지 알아야 할 지에 따라 메소드를 클래스 메소드(class mathod)
스태틱 메소드(static method) 로 정의할지 결정할 수 있습니다.

여기서는 how_many 메소드를 클래스 메소드로 만들어 주기 위해 데코레이터 를 이용하였습니다.

데코레이터는 어떤 일을 추가로 해 주는 더 큰 함수로 해당 부분을 감싸주는 것이라고 생각하면 됩 니다.
즉, @classmethod 데코레이터는 아래처럼 호출하는 것과 같습니다:

```
how_many = classmethod(how_many)
```

init 메소드는 Robot 의 인스턴스를 초기화시킬 때 사용됩니다.
이 메소드를 통해 로봇이 하나 추가될 때마다 로봇의 개수를 의미하는 변수 population 을 1 씩 증가시켜 줍니다.
또한 각 생 성된 객체별로 객체 변수 self.name 의 값을 따로따로 지정해 주었습니다.

객체에 속해 있는 변수와 메소드에 접근하기 위해서는 **반드시** self 를 사용해야 한다는 점을 기억하시기 바랍니다.
이것을 다른 말로 **속성 참조(attribute reference)** 라 부릅니다.

프로그램을 살펴보면 메소드에 정의된 것 처럼 클래스에도 **DocString** 이 정의되어 있는 것을 보 실 수 있습니다.
마찬가지로 이 DocString에도 Robot.doc 을 통해 접근할 수 있고,
또 메소드 의 DocString 은 Robot.say_hi.doc 과 같이 접근할 수 있습니다.

12.6. 상속

객체 지향 프로그래밍의 또 다른 큰 장점은 코드를 **재사용** 할 수 있다는 것인데 이를 위한 한 가지 방법으로 **상속** 이 사용됩니다.
어떤 대학의 교수들과 학생들의 명부를 작성하는 프로그램을 작성한다고 해 봅시다.
이 때 교수와 학생 모두 공통적으로 이름, 나이, 주소 등의 성질을 가지고 있을 것이며,
교수에만 적용되는 성질로는 연봉, 과목, 휴가 등이 있을 것이고, 학생에만 적용되는 성질로는 성적, 등록금 등

방법은 `SchoolMember` 라는 이름으로 공통 클래스를 생성한 뒤
교수와 학생 클래스를 이 클래스로부터 *상속* 받아 생성하는 것입니다.
이 경우 상속받은 클래스들은 이를테면 상위 형 식(클래스) 의 세부 형식이 되는 것이고,
따라서 이 세부 형식에 각 상황에 맞는 세부적인 성질들 을 추가해 줄 수 있는 것입니다.

장점은 우리가 `SchoolMember` 에 새로 운 기능을 추가하거나 혹은 있던 기능을 수정하게 되면,
그 하위 클래스인 교수와 학생 클래스에도 이러한 변경 사항이 자동으로 추가된다는 점입니다.
또 다른 장점은 여러분이 예를 들어 대학에 소속된 사람들의 모든 숫자를 파악해야 한다고 할 경우 교수와 학생 객체를 `SchoolMember` 객체로써 참조하여 사용할 수 있다는 점입니다.
이것을 **다형성** 이라고 부르는데,

하위 형식이 부모 형식을 필요로 하는 어떤 상황에서건 이를 대신하여 사용될 수 있다는 것을 의미 합니다.
즉, 자식 클래스의 객체를 부모 클래스의 인스턴스인 것처럼 다루어질 수 있습니다.

따라서 상속을 이용하면 부모 클래스의 코드를 재사용할 수 있고 서로 완전히 독립적인 클래스들
을 정의했을 때처럼 각각 다른 클래스에 이를 또 반복해서 써 줄 필요가 없다는 것입니다.

이 상황에서의 `SchoolMember` 클래스를 **기본 클래스** 혹은 **슈퍼 클래스** 라고 부릅니다.
또 `Teacher` 와 `Student` 클래스는 **파생 클래스** 혹은 **서브 클래스** 라고 부릅니다.

(oop_subclass.py):

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print '(Initialized SchoolMember: {})'.format(self.name)

    def tell(self):
        '''Tell my details.'''
        print 'Name:"{}" Age:"{}"'.format(self.name, self.age),

class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print '(Initialized Teacher: {})'.format(self.name)

    def tell(self):
        SchoolMember.tell(self)
        print 'Salary: "{:d}"'.format(self.salary)

class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print '(Initialized Student: {})'.format(self.name)

    def tell(self):
        SchoolMember.tell(self)
        print 'Marks: "{:d}"'.format(self.marks)
```

```
t = Teacher('Mrs. Hong', 40, 30000)

s = Student('Swaroop', 25, 75)

# prints a blank line
print

members = [t, s]
for member in members:
    # Works for both Teachers and Students
    member.tell()
```

실행 결과:

```
$ python oop_subclass.py

(Initialized SchoolMember: Mrs. Hong)
(Initialized Teacher: Mrs. Hong)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Hong" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"25" Marks: "75"
```

13장. 입력과 출력

13.1. 사용자로부터 입력받기

io_input.py 로 저장하세요:

```
def reverse(text):
    return text[::-1]
def is_palindrome(text):
    return text == reverse(text)

something = raw_input("Enter text: ")
if is_palindrome(something):
    print "Yes, it is a palindrome"
else:
    print "No, it is not a palindrome"
```

문자열을 뒤집기 위해서는 슬라이스를 사용합니다.
앞서 보았듯이 열거형의 슬라이스 기 능을 이용하여
seq[a:b] 와 같은 코드를 통해 위치 a부터 위치 b까지 문자열
슬라이스 숫자에 세 번째 인수를 넘겨 주어 슬라이스 *스텝* 을 지정
음의 스텝을 지정하면 열거형의 마지막부터 반대 방향으로 슬라이스
예를 들어 -1 을 지정하면 뒤집혀진 문자열이 반환됩니다.

raw_input() 함수는 인수로 넘겨받은 문자열을 화면에 표시
사용자가 입력한 내용을 문자열로 반환해 줍니다.

이제 이 문자열을 받아서 뒤집어 줍니다.
여기서 뒤집혀진 문자열이 뒤집혀지지 않았을 때의 문자열과
동일할 때, 이것을 영어로 palindrome[1] 이라고 부릅니다.

실행 결과:

```
Enter text: sir
No, it is not a palindrome

$ python io_input.py
Enter text: madam
Yes, it is a palindrome

$ python io_input.py
Enter text: racecar
Yes, it is a palindrome
```

13.2. 파일 입/출력

입/출력을 위해 파일을 열고 사용하려면 file 클래스의 객체를 생성한 후
read , readline , write 사용
파일을 열때 파일을 읽는 모드와 쓰는 모드 지정

마지막으로 파일을 읽거나 쓰는 일을 모두 마친 후,
close 메소드를 호출

예제 (io_using_file.py 로 저장하세요):

```
poem = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
'''

# Open for 'w'riting
f = open('poem.txt', 'w')
# Write text to file
f.write(poem)
# Close the file
f.close()

# If no mode is specified,
# 'r'ead mode is assumed by default
f = open('poem.txt')
while True:
    line = f.readline()
    # Zero length indicates EOF
    if len(line) == 0:
        break
    # The `line` already has a newline
    # at the end of each line
    # since it is reading from a file.
    print line,
# close the file
f.close()
```

실행 결과:

```
$ python io_using_file.py
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```

13.3. Pickle

파이썬은 pickle 이라고 불리우는 기본 모듈을 제공하는데,
이것은 *어떤* 파이썬 객체이든지 파 일로 저장해 두었다가 나중에 불러와서 사용할 수 있게 하는 모듈입니다.
이것은 객체를 **영구히** 저장해 둔다고 합니다.

예제 (io_pickle.py 로 저장하세요):

```
import pickle

# The name of the file where we will store the object
shoplistfile = 'shoplist.data'
# The list of things to buy
shoplist = ['apple', 'mango', 'carrot']

# Write to the file
f = open(shoplistfile, 'wb')
# Dump the object to a file
pickle.dump(shoplist, f)
f.close()

# Destroy the shoplist variable
del shoplist

# Read back from the storage
f = open(shoplistfile, 'rb')
# Load the object from the file
storedlist = pickle.load(f)
print storedlist
```

실행 결과:

```
$ python io_pickle.py
['apple', 'mango', 'carrot']
```

파일에 객체를 저장하기 위해서 먼저
open 문을 이용하여 *쓰기/바이너리* 모드로 파일 을 열어 준 후
pickle 모듈의 dump 함수를 호출하여 줍니다.
이 과정을 *피클링(pickling)* 이 라고 합니다.

다음으로 pickle 모듈의 load 함수를 이용하여
파일에 저장된 객체를 불러옵니다.
이 과정을 *언피클링(unpickling)* 이라고 합니다.

13.4. 유니코드

지금까지 우리가 문자열을 쓰거나 읽고, 또 파일에 쓸 때에 영어 알파벳 문자들만을 주로 이용해 왔습니다.
만약 여러분이 영어가 아닌 다른 언어로 된 문자를 읽고 쓰고 싶을 경우에는 unicode 형식을 이용할 필요가 있으며,
이것은 문자 u 를 앞에 붙여 주어 지정해 줍니다:

```
>>> "hello world"
'hello world'
>>> type("hello world")
<type 'str'>
>>> u"hello world"
u'hello world'
>>> type(u"hello world")
<type 'unicode'>
```

비 영어권 언어를 다룰 때에는 str 대신 unicode 형식을 사용

그러나 여러분이 파일을 읽고 쓰거나 다른 컴퓨터와 교신하려고 할 때에는
유니코드 문자열들을 송수신 가능한 형태로 바꾸어 주어야 하며,
이러한 형식의 이름을 "UTF-8" 이라고 부릅니다.
다음과 같이 open 표준 함수에 간단한 키워드 인수를 넘겨 줌으로써 이 형식을 사용하여 파일을 읽고 쓰게 할 수 있습니다.

```
# encoding=utf-8
import io

f = io.open("abc.txt", "wt", encoding="utf-8")
f.write(u"Imagine non-English language here")
f.close()

text = io.open("abc.txt", encoding="utf-8").read()
print text
```

14장. 예외 처리

14.1. 오류

```
>>> Print "Hello World"
  File "<stdin>", line 1
    Print "Hello World"
                       ^
SyntaxError: invalid syntax
```

14.2. 예외

이번에는 사용자로부터 뭔가를 입력 받는 것을 **시도하는(try)** 경우를 생각해 봅시다.
이 때 `ctrl-d` 를 누르고 어떻게 되는지 살펴봅시다.

```
>>> s = raw_input('Enter something --> ')
Enter something --> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
EOFError
```

그러면 파이썬은 `EOFError` 라고 불리우는 오류를 발생시키는데 이때 EOF란 **파일의 끝(end of file)** 을 의미하며(파일의 끝은 `ctrl-d` 에 의해 표현됩니다), 갑자기 파일의 끝이 올 것을 예상하 지 못했기 때문에 위와 같은 오류가 발생하는 것입니다.

14.3. 예외 처리

예외는 `try..except` 문을 통해 처리할 수 있습니다. 이것은 try 블록 안에 평소와 같이 명령을 입력하고 예외 상황에 해당하는 오류 핸들러를 except 블록에 입력해 주면 됩니다.

예제 (`exceptions_handle.py` 로 저장하세요):

```
try:
    text = raw_input('Enter something --> ')
except EOFError:
    print 'Why did you do an EOF on me?'
except KeyboardInterrupt:
    print 'You cancelled the operation.'
else:
    print 'You entered {}'.format(text)
```

예외가 발생할 수 있는 모든 명령문을 `try` 블록에 넣어 주었으며 오류/ 예외를 적절하게 처리해 줄 핸들러를 `except` 절/블록에 넣어 주었습니다.

실행 결과:

```
# Press ctrl + d
$ python exceptions_handle.py
Enter something --> Why did you do an EOF on me?

# Press ctrl + c
$ python exceptions_handle.py
Enter something --> ^CYou cancelled the operation.

$ python exceptions_handle.py
Enter something --> No exceptions
You entered No exceptions
```

14.4. 예외 발생시키기

raise 문에 오류/예외의 이름을 넘겨 주는 것을 통해 예외를 직접 *발생(raise)* 시킬 수 있습니 다.
그러면 예외 객체가 *throw* 됩니다.

```
class ShortInputException(Exception):
    '''A user-defined exception class.'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast

try:
    text = raw_input('Enter something --> ')
    if len(text) < 3:
        raise ShortInputException(len(text), 3)
    # Other work can continue as usual here
except EOFError:
    print 'Why did you do an EOF on me?'
except ShortInputException as ex:
    print ('ShortInputException: The input was ' + \
           '{0} long, expected at least {1}')\
          .format(ex.length, ex.atleast)
else:
    print 'No exception was raised.'
```

실행 결과:

```
$ python exceptions_raise.py
Enter something --> a
ShortInputException: The input was 1 lo
ng, expected at least 3

$ python exceptions_raise.py

Enter something --> abc
No exception was raised.
```

14.5. Try ··· Finally 문

프로그램이 파일을 읽고 있는 상황을 가정해 봅시다. 이 때 예외가 발생할 경우, 예외의 발생 여부와 상관없이 파일 객체를 항상 닫아 주도록 할 수는 없을까요? 이를 위해 `finally` 블록을 사 용합니다.

아래 프로그램을 `exceptions_finally.py` 로 저장하세요:

```
import sys, time

f = None
try:
    f = open("poem.txt")
    # Our usual file-reading idiom
    while True:
        line = f.readline()
        if len(line) == 0:
            break
        print line,
        sys.stdout.flush()
        print "Press ctrl+c now"
        # To make sure it runs for a while
        time.sleep(2)
except IOError:
    print "Could not find file poem.txt"
except KeyboardInterrupt:
    print "!! You cancelled the reading from the file."
finally:
    if f:
        f.close()
    print "(Cleaning up: Closed the file)"
```

실행 결과:

```
$ python exceptions_finally.py
Programming is fun
Press ctrl+c now
^C!! You cancelled the reading from the file.
(Cleaning up: Closed the file)
```

파일에서 한 줄을 읽어올 때마다 time.sleep 함수를 호출하여 2초씩 멈추게 하는 인위적인 코드를 집어넣어 프로그램이 천천히 실행되도록 해 주었습니다 (파이썬은 원래 굉장히 빠릅니다). 프로그램이 실행중일 때, `ctrl + c` 를 눌러 프로그램을 강제로 중단 시켜 봅시다.

그러면 `KeyboardInterrupt` 예외가 발생되며 프로그램이 종료됩니다. 그러나 프로그램이 종료 되기 전에 finally 절이 실행되므로 파일 객체가 항상 닫히게 됩니다.

여기서 `print` 문 뒤에 `sys.stdout.flush()` 를 사용하여 화면에 결과를 바로바로 출력하도록 해 주었습니다.

14.6. with 문

try 블록에서 시스템 자원을 가져오고 finally 문에서 이를 해제하여 주는 것은 공통된 패턴 입니다. 그렇지만, with 문을 이용하면 이것을 좀 더 깔끔하게 작성해 줄 수 있습니다.

exceptions_using_with.py 로 저장하세요:

```
with open("poem.txt") as f:
    for line in f:
        print line,
```

위 예제는 이전의 예제와 동일한 결과를 출력합니다.
차이점은 open 함수를 사용할 때 with 문을 사용하였다는 것입니다.
그러면 파일을 직접 닫아 주지 않아도 with open 이 자동 으로 파일을 닫아 줍니다.

그러면 with 문은 어떻게 자동으로 이러한 것들을 처리해 주는 것일까요? 우선 with 문은 open 문이 반환해 주는 객체를 받아 오는데, 일단 여기서는 이것을 "thefile" 이라고 해 봅시다.

with 문은 *항상* thefile.enter 함수를 호출한 뒤 해당 블록의 코드를 실행하며,
실행이 끝난 후에는 *항상* thefile.exit 가 호출됩니다.

15장. 표준 라이브러리

15.1. sys 모듈

sys 모듈에는 시스템의 기능을 다루는 여러 함수들이 들어 있습니다.
예를 들어 sys.argv 리스트에는 명령줄 인수들이 들어 있습니다.

```
$ python
>>> import sys
>>> sys.version_info
sys.version_info(major=2, minor=7, micro=6, releaselevel='final', serial=0)
>>> sys.version_info.major == 2
True
```

15.2. os 모듈

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:₩₩Python37'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```