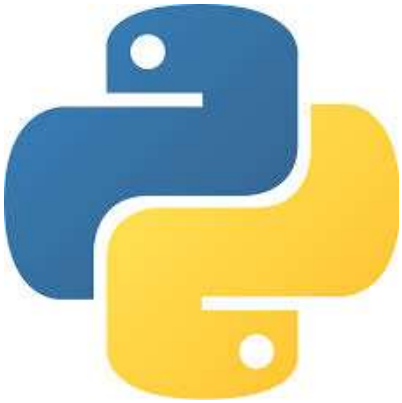


Quick Python Tour

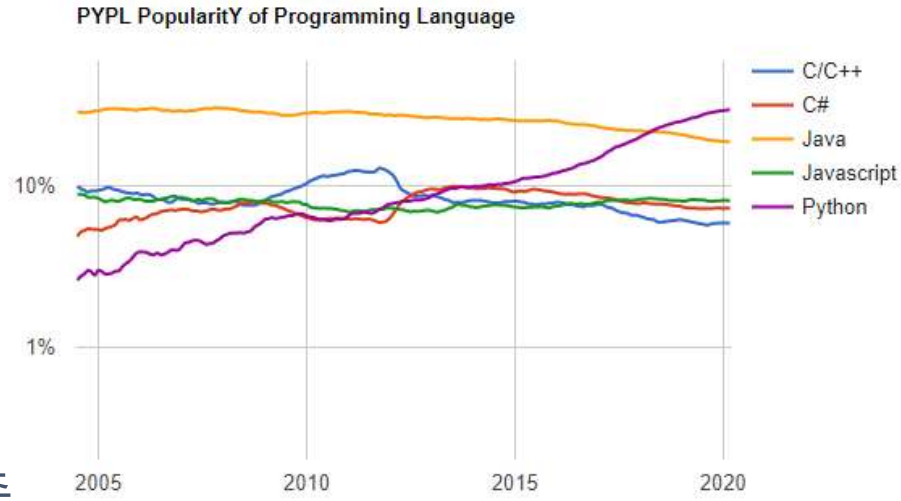
Why Python?

- 초보자가 배우기 쉽고, 베테랑에게는 다양한 응용분야에 적합한 강력한 언어
- 간결한 코드, 가독성 좋아 유지보수 편리
- 20만개 이상의 Python package 활용해 문제 해결 용이

- 특히, 다음 분야에 많이 이용되고 있음
  - Web development
  - IoT
  - Data science
  - AI and deep learning

- 최근 IEEE의 종합 평가에서 100% ranking 점수를 갖춘 유일한 언어

Worldwide, Python is the most popular language,
Python grew the most in the last 5 years (19.0%)
and Java lost the most (-6.8%)



The first program: “Hello, World”

- C

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
   printf(“Hello, World!\n”);
}
```

- Java

```java
public class Hello
{
   public static void main(String argv[])
   {
      System.out.println(“Hello, World!”);
   }
}
```

- now in Python

```python
print("Hello, World!")
```

Printing an Array

```c
void print_array(char* a[], int len)
{
   int i;
   for (i = 0; i < len; i++)
   {
      printf("%s\n", a[i]);
   }
}
```

has to specify `len`,
and only for one type (`char*`)

C

```python
for element in a:
    print(element)
```

only indentations
no { ... } blocks!

for ... in ... :
    ...

Python

no C-style for-loops!

~~`for (i = 0; i < 10; i++)`~~

or even simpler:

```python
print(a)
```

Install Python 3

- Download and install the latest version (Python 3.5 이상)
  - https://www.python.org/downloads
  - Python interpreter, standard library, IDLE IDE 도 함께 설치됨

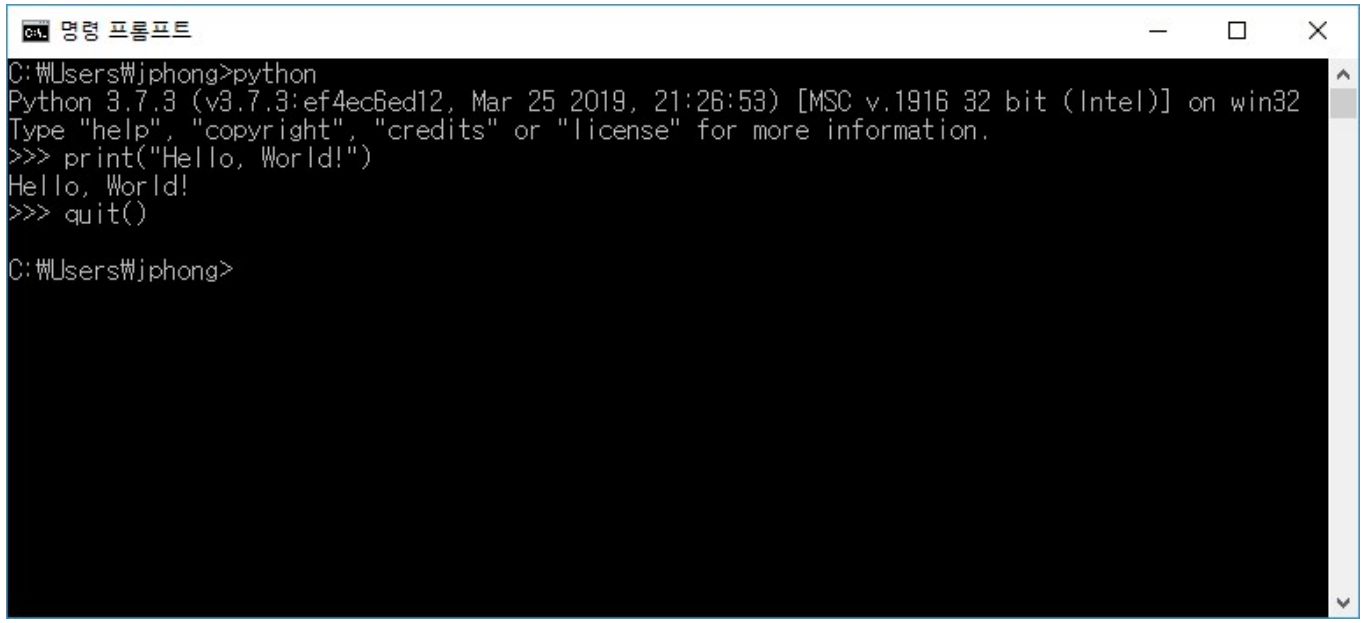
❖Unix 계열에는 보통 이미 설치되어 있다.

- Running Python interpreter
  - Interactive mode

```
$ python
>>> quit()
$
```

  - Script(source file) mode

```
$ python hello.py
$
```



Values, Types, and Expressions

- A data type is defined by:
  - a set of values (e.g. integers)
  - operations (e.g. `+`, `-`, `*`, `/`, `%`, `**`)

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
>>> type('2')
<class 'str'>
>>> type("42.0")
<class 'str'>
>>> type(['kim', 22, 4.3])
<class 'list'>
>>> type(('kim', 22, 4.3))
<class 'tuple'>
>>> type({'name': 'kim', 'age': 22, 'grade': 4.3})
<class 'dict'>
>>> type({'red', 'green', 'yellow'})
<class 'set'>
>>> type(True)
<class 'bool'>
```

- Expressions
  - expression을 실행하면 new value, 즉, object가 생성된다.

```
>>> 40 + 2
42
>>> 20 - 4.5
15.5
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 7 % 3
1
>>> 2 ** 10
1024
>>> (1 + 2) * 3
9
```

Assignment Statements

<name> = <expression>

Typeless Language

Java나 C처럼 변수 이름에 data type을 미리 선언하지 않는다.
이름은 object를 가리킬 뿐이며, 그 object의 type에 달려있다.

<name>

1. evaluate the expression to produce a value(object)
2. create a new name if that name doesn't exist; otherwise, just reuse it.
3. the name refers to the object

```
>>> n = 3
>>> double_n = 2 * n
>>> double_n
6
>>> n = 'n'
>>> type(n)
<class 'str'>
>>> double_n = 2 * n
>>> double_n
'nn'
```

<expression> 속에 포함된 variable 이름은 실행 전에 생성(정의)되어 있어야 한다.

Augmented Assignments

Symbol	Example	Result
+=	x = 7 x += 2	x refers to 9
-=	x = 7 x -= 2	x refers to 5
*=	x = 7 x *= 2	x refers to 14
/=	x = 7 x /= 2	x refers to 3.5
//=	x = 7 x //= 2	x refers to 3
%=	x = 7 x %= 2	x refers to 1
**=	x = 7 x **= 2	x refers to 49

Multiple Assignments

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

swap

```
>>> tmp = a
>>> a = b
>>> b = tmp
>>> a, b
(2, 1)
```

```
>>> a, b = b, a
>>> a, b
(1, 2)
```

packing/unpacking

```
>>> c = 1, 2
>>> c
(1, 2)
>>> c1, c2 = c
>>> c1, c2
(1, 2)
```

Python Built-in Data(Object) Types

- Numbers
  - `int`: `255, 0xff, 0o377, 0b11111111`
  - `float`: `3.14159, 1.2345e-56`
  - `complex`: `1.5 + 2j`

- Sequences: ordered collections

  indexing (예: `a[2]`), slicing (예: `a[2:5]`) 가능
  - `str`: `'abc', "don't", ''` -- immutable
  - `tuple`: `('kim', 22, 4.3)` -- immutable
  - `list`: `['kim', 22, 4.3]` -- mutable

- Mappings: unordered
  - `set`: `{'red', 'green', 'yellow'}` -- mutable
  - `dict`: `{'name': 'kim', 'age': 22,'grade': 4.3}` -- mutable

- Others
  - `bool`: `True, False`
  - `None`
  - file: `open('egg.txt')`
  - Program units:
    - functions
    - modules
    - classes

Numbers

```
>>> a, b = 7, 3
>>> a // b
2
>>> a % b
1
>>> (b * (a // b) + a % b) == a
True
```

```
>>> a, b = 7.0, 3.0
>>> a / b
2.3333333333333335
>>> a // b
2.0
>>> a % b
1.0
>>> (b * (a // b) + a % b) == a
True
```

Symbol	Operator	Example	Result
-	Negation	-5	-5
+	Addition	11 + 3.1	14.1
-	Subtraction	5 - 19	-14
*	Multiplication	8.5 * 4	34.0
/	Division	11 / 2	5.5
//	Integer Division	11 // 2	5
%	Remainder	8.5 % 3.5	1.5
**	Exponentiation	2 ** 5	32

Table 1—Arithmetic Operators

Precedence	Operator	Operation
Highest	**	Exponentiation
	-	Negation
	*, /, //, %	Multiplication, division, integer division, and remainder
Lowest	+, -	Addition and subtraction

Table 2—Arithmetic Operators Listed by Precedence from Highest to Lowest

- 최대 integer
  - 제한이 없다. word 크기 초과 가능
- Floating-point numbers
  - finite precision (due to finite bits used)
  - cannot represent real numbers exactly

```
>>> 123456789.0123456789
123456789.01234567
>>> 0.123456789012345678 9e100
1.2345678901234567e+99
>>> 2 ** 0.5
1.4142135623730951
```

Strings: sequence of characters

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

```
>>> 'Py' 'thon'
'Python'
>>> 3 * 'un' + 'ium'
'unununium'
```

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't c
  ...
SyntaxError: invalid syntax
```

```
>>> prefix + 'thon'
'Python'
```

String: Indexing and Slicing

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Indexing – a chararacter

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
```

Strings are immutable (변경불가)

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object
>>> word[2:] = 'py'
  ...
TypeError: 'str' object
```

Slicing – a substring

```
>>> word[0:2]
'Py'
>>> word[2:5]
'tho'
```

slice도 string이다

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

```
>>> word[4:42]
'on'
>>> word[42:]
''
>>> word[-2:]
'on'
```

error가 아님!!

length of string

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Tuples – immutable ordered collection

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Lists – mutable ordered collection

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Indexing and Slicing

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

New (shallow) copy

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Concatenation

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

nesting

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

modification

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements
>>> letters[:] = []
>>> letters
[]
```

Common Sequence Operations – for string, list, tuple, range, ...

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

String Methods

```
>>> s = " this is  a  python course. \n"
>>> words = s.split()
>>> words
['this', 'is', 'a', 'python', 'course.']
>>> s.strip()
'this is  a  python course.'
>>> " ".join(words)
'this is a python course.'
>>> "; ".join(words).split("; ")
['this', 'is', 'a', 'python', 'course.']
>>> s.upper()
' THIS IS  A  PYTHON COURSE. \n'
```

For more detail, see:
https://docs.python.org/3/library/stdtypes.html#string-methods

Functions that return a new string

- `str.capitalize()` / `str.lower()`. Returns a copy of `str` with all letters converted to uppercase / lowercase.
- `str.strip()`. Returns a copy of `str` with all **whitespace** (spaces/tabs/newlines) from the beginning and end of the string removed.
  Example: `"  test  ".strip() == "test"`.
- `str.replace(old,new)`. Returns a copy of `str` with all instances of `old` within the string replaced with new.
  Example: `"hallo all!".replace("al", "el") == "hello ell!"`.

Functions which return information about a string

- `str.count(substring)`. Returns the number of times `substring` appears within `str`.
- `str.find(substring)` / `str.rfind(substring)`. Returns the position of the *first* instance of `substring` within `str`. rfind returns the position of the *last* instance of `substring`.
- `s.startswith(substring)` / `str.endswith(substring)`. Returns `True` if the string starts with / ends with substring.
  Example: `"Hello".startswith("he") == False`, but `"Hello".endswith("lo") == True`

Functions which transform the string into other types

- `str.split(separator)`. Returns a *list* of words in str, using separator as the delimiter string.
  Example: `"hello world, Mihir here".split(" ")` returns `["hello","world,","Mihir","here"]`.
  Example: `"mississippi".split("s")` returns `["mi", "", "i", "", "ippi"]`.
- `separator.join(seq)`. This one is tricky. It takes a *list* of strings `seq` and combines them into a string. Each element in `seq` is separated by `separator` in the returned string.
  Example: `" ".join(["hello","world"]) == "hello world"`

List Operations

```
>>> a = [1,'python', [2,'4']]
>>> a + [2]
[1, 'python', [2, '4'], 2]
>>> a.extend([2, 3])
>>> a
[1, 'python', [2, '4'], 2, 3]
```

same as `a += [2, 3]`

```
>>> a.append('5')
>>> a
[1, 'python', [2, '4'], 2, 3, '5']
>>> a[2].append('xtra')
>>> a
[1, 'python', [2, '4', 'xtra'], 2, 3, '5']
>>> [1, 2] * 3
[1, 2, 1, 2, 1, 2]
```

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from `s` (same as `del s[:]`)
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i] == x`
`s.reverse()`	reverses the items of *s* in place

`s.sort(key=None, reverse=False)` rearrange s with sorting

Membership Operators

Membership Operators	Examples	Result
in	10 in (10, 20, 30)	True
	'red' in ('red','green','blue')	True
not in	10 not in (10, 20, 30)	False

```
>>> bg_color = 'blue'
>>> colors = ['red', 'blue', 'green']
>>> bg_color in colors
True
>>> bg_color in 'Background color is blue.'   # substring ?
True
>>> colors == []
False
```

Conditions

The result is boolean: `True` or `False`

```
>>> a = b = 0
>>> a == b
True
>>> type(3 == 5)
<class 'bool'>
>>> "my" == 'my'
True
>>> 'abc' < 'abd'
True
>>> [1, 2, 3] < [1, 3, 2]
True
```

```
>>> x, y = 12, 15
>>> x>10 and y>10
True
>>> x>14 or y>14
True
>>> not(x>10 and y>10)
False
>>> not(x>10) or not(y>10)
False
>>> 0<x and x<20
True
>>> 0 < x < 20
True
```

Python Relational Operators

Operator	Name	Example	Result
==	Equal	x==y	True if x is exactly equal to y.
!=	Not equal	x!=y	True if x is exactly not equal to y.
>	Greater than	x>y	True if x (left hand argument) is greater than y (right hand argument).
<	Less than	x<y	True if x (left hand argument) is less than y (right hand argument).
>=	Greater than or equal to	x>=y	True if x (left hand argument) is greater than or equal to y (left hand argument).
<=	Less than or equal to	x<=y	True if x (left hand argument) is less than or equal to y (right hand argument).

Python Logical Operators

x	y		x and y	x or y	not x
False	False		False	False	True
True	False		False	True	False
False	True		False	True	
True	True		True	True	

Truth Value Testing

- False values
  - `None`
  - `False`
  - Zero of any numbers: `0`, `0.0`, `0j`
  - Any empty sequence: `''`, `()`, `[]`
  - Any empty mapping: `{}`
- True values: all other values

```
>>> None
>>> print(None)
None
>>> bool(None)
False
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
```

```
>>> friend = '홍길동'
>>> if friend:
        print('Hello,', friend)
else:
        print('Hello, World!')


Hello, 홍길동
```

```
>>> if friend != '':
```

로 써도 되는데…

Control Flow Statements

https://docs.python.org/3/tutorial/controlflow.html

`if` statements

```
if condition:
    ...
[elif condition:
    ... ]*
[else:
    ... ]
```

`while` statement

```
while condition:
    ...
[else:
    ... ]
```

`for` statements

```
for item in iterable:
    ...
[else:
    ... ]
```

Iterable:

- composite objects such as string, list
- generators: 중지될 때까지 계속 item을 생성/공급(return)하는 function 또는 class object

여기서 `in`은 membership operator가 아니다.
Iterable object에서 next item을 가져온다는 의미

`else` clause in `while` and for `loops`:
loop이 정상 종료될 때 실행된다. (break로 끝날 때는 실행 안됨.)

[...] --> 생략 가능 (0 or 1 times)
[...]* --> 생략 가능, 반복 가능 (0 or more times)

`if` Statements

if … else …

```
>>> a = 'yellow'
>>> if a == 'blue' or a == 'yellow' or a == 'red':
...     print(a, 'is a color')
... else:
...     print(a, 'not a color')
...
yellow is a color
```

rewrite as:

```
if a in ('blue', 'yellow', 'red'):
```

if … elif … else …

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

`while` Statements

Fibonacci sequence:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

`break`, `continue` statements, same as in Java/C

Using Function definition and call:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

for Statements

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

주의:
Loop안에서 list를 변경하면 위험하다.
정 필요하다면, copy를 만들어 변경하라.

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

range function

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

일종의 sequence type object을 return한다.

- for loop에서 사용할 수 있다(iterable)
- list나 tuple로 변환할 수 있다.
- indexing/slicing 가능
- generator이다.

list는 모든 element를 memory에 잡아야 하지만,
range는 필요할 때 하나씩 생성한다. (memory 절약)

Sequence의 index로 iteration하려면,

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

```
>>> range(5, 10)
range(5, 10)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(-10, -100, -20))
[-10, -30, -50, -70, -90]
```

break, continue and else

- break and continue
  borrowed from C/Java
- else in loops
  - loop이 정상적으로 종료될 때 마지막으로 실행된다.
    (break으로 빠져나올 때는 else 절이 수행 안됨)

```
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found a number", num)
```

```
for (n=2; n<10; n++) {
   good = true;
   for (x=2; x<n; x++)
      if (n % x == 0) {
         good = false;
         break;
      }
   if (good)
      printf("%d ", n);
}
```

C/Java

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Sets: unordered collection with no duplicate elements

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Empty set

```
>>> d = {}
>>> type(d)
<class 'dict'>
>>> s = set()
>>> type(s)
<class 'set'>
```

`set` operation

Operation	Equivalent	Result
`len(s)`		cardinality of set *s*
`x in s`		test *x* for membership in *s*
`x not in s`		test *x* for non-membership in *s*
`s.issubset(t)`	`s <= t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s >= t`	test whether every element in *t* is in *s*
`s.union(t)`	`s | t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s & t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s - t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s ^ t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*
`s.update(t)`	`s |= t`	return set *s* with elements added from *t*
`s.intersection_update(t)`	`s &= t`	return set *s* keeping only elements also found in *t*
`s.difference_update(t)`	`s -= t`	return set *s* after removing elements found in *t*
`s.symmetric_difference_update(t)`	`s ^= t`	return set *s* with elements from *s* or *t* but not both
`s.add(x)`		add element *x* to set *s*
`s.remove(x)`		remove *x* from set *s*; raises KeyError if not present
`s.discard(x)`		removes *x* from set *s* if present
`s.pop()`		remove and return an arbitrary element from *s*; rais
`s.clear()`		remove all elements from set *s*

Dictionaries – key→value mapping

- 순서가 없으니 indexing 도 slicing도 할 수 없다.
- Index 대신, key로 해당 item을 찾아 value를 access. (key는 unique해야)
- key는 immutable type이어야 한다.

```
>>> d = {'a': 1, 'b': 2, 'c': 1}
>>> d['b']
2
>>> a = d['b']
>>> a
2
>>> d['b'] = 3
>>> d
{'a': 1, 'c': 1, 'b': 3}
>>> b = d['e']

Traceback (most recent call last):
  File "<pyshell#149>", line 1, in <module>
    b = d['e']
KeyError: 'e'
>>> d['e'] = 5
>>> d
{'a': 1, 'c': 1, 'b': 3, 'e': 5}
```

Update if key exists

Insert if key does not exists

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

`dict` Type Operations

Operation	Result
`len(`*a*`)`	the number of items in *a*
a`[`*k*`]`	the item of *a* with key *k*
a`[`*k*`] = `*v*	set *a*`[`*k*`]` to *v*
`del` *a*`[`*k*`]`	remove *a*`[`*k*`]` from *a*
a`.clear()`	remove all items from `a`
a`.copy()`	a (shallow) copy of `a`
a`.has_key(`*k*`)`	`True` if *a* has a key *k*, else `False`
k `in` *a*	Equivalent to *a*.has_key(*k*)
k `not in` *a*	Equivalent to `not` *a*.has_key(*k*)
a`.items()`	a copy of *a*'s list of (*key*, *value*) pairs
a`.values()`	a copy of *a*'s list of values
a`.get(`*k*`[, `*x*`])`	*a*`[`*k*`]` if *k* `in` *a*, else *x*
a`.setdefault(`*k*`[, `*x*`])`	*a*`[`*k*`]` if *k* `in` *a*, else *x* (also setting it)
a`.pop(`*k*`[, `*x*`])`	*a*`[`*k*`]` if *k* `in` *a*, else *x* (and remove k)

Dictionary construction alternatives:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Counting frequencies:

```
>>> def incr(d, key):
...     if key not in d:
...             d[key] = 1
...     else:
...             d[key] += 1
...
```

```
>>> def incr(d, key):
...     d[key] = d.get(key, 0) + 1
```

```
>>> incr(d, 'z')
>>> d
{'a': 1, 'c': 1, 'b': 2, 'z': 1}
>>> incr(d, 'b')
>>> d
{'a': 1, 'c': 1, 'b': 3, 'z': 1}
```

DB: Dictionary of dictionaries

```
# records
bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
tom = {'name': 'Tom',       'age': 50, 'pay': 0,     'job': None}

# database
db = {}
db['bob'] = bob
db['sue'] = sue
db['tom'] = tom

if __name__ == '__main__':       # when run as a script
    for key, value in db.items():
        print(key, '=>\n  ', value)
```

```
bob =>
   {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
sue =>
   {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
tom =>
   {'name': 'Tom', 'age': 50, 'pay': 0, 'job': None}
```

Performance

- lists, tuples, and strings
  - random access: O(1)
  - insertion/deletion/`in`: O(n)
- dict
  - random access/`in`: O(1)
  - insertion/deletion: O(1)
  - no linear ordering!

Object-oriented programming

person.py:

```
class Person:
    def __init__(self, name, pay=0, job=None):
        self.name = name
        self.pay  = pay
        self.job  = job
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, pay, 'manager')
    def giveRaise(self, percent, bonus=0.1):
        Person.giveRaise(self, percent + bonus)
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', 40000, 'hardware')
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.1)
    print(sue)
    tom = Manager(name='Tom Doe', pay=50000)
    tom.giveRaise(0.1)
    print(tom.lastName())
    print(tom)
```

Output:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 40000]
Smith Jones
[Person: Sue Jones, 44000.0
Doe
[Person: Tom Doe, 60000.0]
>>>
```