Python Programming for Science and Machine Learning

서울대학교 자연과학대학 물리천문학부

2018 가을학기

전산물리 (884.310) & 응용전산물리 (3342.618)

3 주차 강의노트 내용

Python

- Python loops (while & for, terms & objects for iteration, range(), zip(), filter(), map(), enumerate(), iter(), next(), __next__(), …)
- Python function (def …)

공지

- Exercise: 02 차 실습과제 (예습) [9/17(월) - 9/21(금) 24:00, 제출]
- Homework: 02 차 정규과제 [9/17(월) ETL 공지 - 9/28(금) 24:00, 제출]
- 중간고사 일정확인

파이썬 반복문
Loops

파이썬 반복문 (iteration over container) 관련 기능을 이해하기 위한 용어 설명

1. Iterable (순환가능객체):

순환가능한 컨테이너 객체 . 여러가지 데이터 컨테이너들 중에서 원소들을 하나씩 반환가능한 객체들을 지칭하며 다음과 같은 예들로 구분될 수 있다 .

- **' 순차형 ' (sequence type)** 데이터 객체 : str(문자열), list(리스트), tuple(튜플)
- **' 비순차형 ' (non-sequence type)** 데이터 객체 : dict(사전), file(파일)
- __iter__() 혹은 __getitems__() 함수를 통해 정의된 사용자 클래스의 객체

일반적으로 for-loop 이나 순차형 객체가 입력인자로 필요한 함수들 (zip(), filter(),map()..) 과 함께 쓰이며 , 순환가능객체가 자체적으로 가진 __iter__() 멤버함수나 iter() 내장함수를 통하여 그 자신의 요소를 순환할 수 있는 순환자 (Iterator) 객체를 만들 수 있다 .

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, file objects, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements Sequence semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence, and generator.

파이썬 반복문 (iteration over container) 관련 기능을 이해하기 위한 용어 설명

2. Iterator (순환자객체):

순환가능객체가 가진 데이터 전체를 순환할 수 있는 기능을 가진 객체로서, 순환가능객체에 [__iter__(), iter()] 메소드를 작용하여 생성할 수 있다.

- 순환자 자체가 가진 __next__() 멤버함수, 혹은 next() 내장함수 (built-in function) 를 호출할 때 마다, 연결된 순환가능객체가 가지고 있는 데이터 요소들을 처음부터 하나씩 반환한다.
- 이 과정에서 모든 데이터 요소들이 호출되어 더이상 호출할 요소가 남아 있지 않을때, 마지막으로 StopIteration 이라는 예외처리문 (exception) 이 호출되고, 이 순환자 객체는 유효함을 잃어버리고 소진된다 (exhausted). 소진된 이후에는, __next__() 나 next() 함수의 호출시에도 매번 StopIteration 으로 처리될 뿐이며, 따라서 소진된 순환자객체에 다시 list() 나 tuple() 혹은 dict() 와 같은 컨테이너 생성함수를 **사용하면 빈 컨테이너가 생성된다**.

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in Iterator Types.

파이썬 반복문 (iteration over container) 관련 기능을 이해하기 위한 용어 설명

3. Generator :

생성자 함수 . 순환자의 일종인 생성 순환자 (generator iterator) 를 반환 (후에 계속).

generator

A function which returns a generator iterator. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator

An object created by a generator function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression

An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

파이썬 반복문 (iteration over container) 관련 기능을 이해하기 위한 용어 설명

Iterable [순환가능객체]

1. sequence type
[순차형]

list
tuple
str
...

2. non-sequence type
[비순차형]

dict
file
...

iter(<iterable>)

list(<list_itertor>)
tuple(<tuple_iterator>)
str(<str_iterator>)
dict(<zip_iterator>)
dict(<enumerate_iterator>)
...

Iterator [순환자객체]

list_iterator
tuple_iterator
str_iterator
dict_keyiterator
...
zip_iterator
enumerate_iterator
filter_iterator
map_iterator
...
generator_iterator
...

range() / xrange()
...

for .. in .. (*)

- for 문은 순환가능객체 (iterable) 의 순환자 (iterator) 를 직접 인자로 받아서 , __next__() 함수를 통하여 순환가능객체에 속한 데이터 요소들의 값을 하나씩 반환한다 . 순환가능객체를 인자로 직접 받을 경우 , 자동으로 이들의 순환자를 생성시키고 이를 순환에 사용하게된다 .

기본 문법 :

```
for <요소 아이템> in <순환자/순환가능객체>
    <명령구문>
```

- for 문은 인자로 받은 < 순환가능객체 > 의 요소를 순차적으로 < 요소 아이템 > 에 할당 . 할당받은 < 요소 아이템 > 을 가지고 < 명령구문 > 수행 . < 순환가능객체 > 의 모든 아이템을 순회하거나 , break 를 만나면 시점에서 종료 .

리스트객체의 순환

```
l = [True, "dog", 10]
for i in l:
    print (i, type(i))
for i in iter(l):
    print (i, type(i))
```

```
True <class 'bool'>
dog <class 'str'>
10 <class 'int'>
True <class 'bool'>
dog <class 'str'>
10 <class 'int'>
```

사전객체의 순환 1

```
d = {"orange":10, "apple":20, "melon":30}
for k in d:
    print (k)
for k in iter(d):
    print (k)
```

```
orange
apple
melon
orange
apple
melon
```

for .. in .. (*)

사전객체의 순환 2

```
d = {"orange":10, "apple":20, "melon":30}
for k, v in d.items():
    print (k, v)
for k in d.keys():
    print (k)
for k in d.values():
    print (v)
```

```
orange
apple
melon
orange 10
apple 20
melon 30
orange
apple
melon
30
30
30
```

사전객체의 순환 3

```
d = {"orange":10, "apple":20, "melon":30}
for i in d.items():
    print (i, type(i))
```

```
('orange', 10) <class 'tuple'>
('apple', 20) <class 'tuple'>
('melon', 30) <class 'tuple'>
```

문자열객체의 순환

```
s = "y=x"
for i in s:
    print (i, type(i))
for i in iter(s):
    print (i, type(i))
```

```
y <class 'str'>
= <class 'str'>
x <class 'str'>
y <class 'str'>
= <class 'str'>
x <class 'str'>
```

반복문의 중첩

- 반복문은 2 개 이상 중첩하여 사용할 수 있음

예) 반복문의 중첩 : 구구단 테이블

```
for i in [1,2]:
    print("=== %s 단 ===" % (str(i)))
    for j in [1,2,3,4,5,6,7,8,9]:
        print("%s * %s = %s" % (i, j, i*j))
```

```
=== 1 단 ===
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
=== 2 단 ===
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
```

While (*)

- while 문은 조건식이 참 (True) 인 동안 내부명령구문을 반복하여 수행 . 조건식은 내부명령구문을 수행하기 전에 평가되고 , 매번 구문을 수행한 이후에 재평가가 이루어지며 , 조건식이 거짓 (False) 으로 평가되면 , while 문을 정지하고 벗어남

기본문법

```
while <조건식>:
    <명령구문>
```

예) while

```
>>> a = 0
>>> while a < 10:
...     print(a)
...     a += 1
...
```

```
0
1
2
3
4
5
6
7
8
9
```

반복문의 제어 : break & continue (*)

- break : 반복문 수행 도중에 break 문을 만나면 , **반복문의 내부 블록을 벗어남**

- continue : 반복문 수행 도중에 continue 문을 만나면 , continue 문 이후의 내부 블록의 스크립트를 수행하지 않고 , 다음 아이템부터 새로 반복문 시작

예) for - break & continue

```
LIST = [1,2,3,4,5,6,7,8,9,10]

for i in LIST:
    if i > 5:
        break
    print("LIST item (5이하): %i" %(i))

for i in LIST:
    if i % 2 == 1:
        continue
    print("LIST item (짝수): {0}".format(i))
```

```
LIST item (5이하): 1
LIST item (5이하): 2
LIST item (5이하): 3
LIST item (5이하): 4
LIST item (5이하): 5
LIST item (짝수): 2
LIST item (짝수): 4
LIST item (짝수): 6
LIST item (짝수): 8
LIST item (짝수): 10
```

반복문의 제어 : else

- else : for 반복문이 (break 없이) 정상적으로 종료되면, 마지막에 else 블록이 수행될 수 있다.

예) for - else

```
LIST = [1,2,3,4,5,6,7,8,9,10]
for i in LIST:
    if i % 2 == 0:
        continue
    print("LIST item (홀수): {0}".format(i))
else:
    print("for룹 끝.")
```

```
LIST item (홀수): 1
LIST item (홀수): 3
LIST item (홀수): 5
LIST item (홀수): 7
LIST item (홀수): 9
for룹 끝.
```

제어와 반복 관련한 유용한 함수 : range() - 1

- range() : 정수 수열을 생성하는 객체를 반환하며 , 이를 순환자와 비슷하게 (순환자는 아님), for 문에서나 , list() 나 tuple() 과 같은 순환가능객체 생성함수의 입력인자로 사용될 수 있다 .

```
range(<종료값>)

혹은

range(<시작값>, <종료값>, <증가값>)
```

```
?range
```

```
Init signature: range(self, /, *args, **kwargs)
Docstring:
range(stop) -> range object
range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from start (inclusive)
to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
These are exactly the valid indices for a list of 4 elements.
When step is given, it specifies the increment (or decrement).
Type:           type
```

제어와 반복 관련한 유용한 함수 : range() - 2 (*)

range()로 정수 리스트 만들기

```
l1 = list(range(10)) # [0,..,9]
l1
l2 = list(range(5,10)) # [5,..,9]
l2
l3 = list(range(0,10,2)) # 2씩 증가
l3
l4 = list(range(10,0,-1)) # 내림차순으로
l4
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[5, 6, 7, 8, 9]

[0, 2, 4, 6, 8]

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

range()이용한 for문 순환 1

```
for i in range(5):
    print (i)
for i in range(0,10,2):
    print (i)
```

```
0
1
2
3
4
0
2
4
6
8
```

range()이용한 for문 순환 2

```
s = "Awesome Python"
for i in range(len(s)):
    print(" s[%i] = %s"%(i,s[i]))
```

```
 s[0] = A
 s[1] = w
 s[2] = e
 s[3] = s
 s[4] = o
 s[5] = m
 s[6] = e
 s[7] =
 s[8] = P
 s[9] = y
 s[10] = t
 s[11] = h
 s[12] = o
 s[13] = n
```

제어와 반복 관련한 유용한 함수 : enumerate() - 1 (*)

- enumerate() : 순환가능객체의 요소값과 그에 해당하는 인덱스 값을 반환하며 , **반환 값은 쌍을 이룬 튜플 객체의 순환자이다 .**

```
enumerate(<순환가능객체>)
혹은
enumerate(<순환가능객체>, <시작값>)
```

enumerate()이용한 for문 순환

```
s = "Awesome Python"
for i, v in enumerate(s):
    print(" s[%i] = %s"%(i,v))
```

```
 s[0] = A
 s[1] = w
 s[2] = e
 s[3] = s
 s[4] = o
 s[5] = m
 s[6] = e
 s[7] =
 s[8] = P
 s[9] = y
 s[10] = t
 s[11] = h
 s[12] = o
 s[13] = n
```

```
# enumerate()는 순환자를 반환한다
es = enumerate(s)
es.__next__()
es.__next__()
es.__next__()
```

```
(0, 'A')

(1, 'w')

(2, 'e')
```

제어와 반복 관련한 유용한 함수 : enumerate() - 2

예) range() 이용하여 순환가능객체의 인덱스 순환 :

=> *len(< 순환가능객체 >)* 는 그 객체가 가지고 있는 첫 차원의 요소 갯수를 반환한다 . (즉 , 사전의 경우 key 의 갯수)

```
LIST=['Apple','Banana','Orange']

for i in range(len(LIST)):
    print("LIST index: {0}, value: {1}".format(i, LIST[i]))
```

```
LIST index: 0, value: Apple
LIST index: 1, value: Banana
LIST index: 2, value: Orange
```

예) enumerate() 이용 :

```
LIST=['Apple','Banana','Orange']

for i in enumerate(LIST):
    print(i)

for index, value in enumerate(LIST):
    print(index,value)

for index, value in enumerate(LIST,101):
    print(index, value)
```

```
(0, 'Apple')
(1, 'Banana')
(2, 'Orange')
0 Apple
1 Banana
2 Orange
101 Apple
102 Banana
103 Orange
```

리스트 내장 (List Comprehensions) - 1 (*)

- 리스트를 비롯한 기존의 <순환 가능 객체>의 요소값들에 대하여, <조건식>을 만족시키는 <아이템>들의 필터링이나 추가적인 연산을 통해, 새로운 리스트 객체를 쉽게 생성할 수 있다.

기본문법

```
# List Comprehension:
<표현식> for <아이템> in <순환가능객체> (if <조건식>)
```

```
>>> LIST = list(range(1,11))
>>> LIST
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [i**2 for i in LIST]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> TUPLE = ("lion","tiger","cat")
>>> [len(i) for i in TUPLE]
[4, 5, 3]
```

```
>>> DICT = {1:"lion", 2:"tiger", 3:"cat"}
>>> [v.upper() for v in DICT.values()]
['LION', 'TIGER', 'CAT']
```

리스트 내장 (List Comprehensions) - 2 (*)

```
# 길이가 5를 초과하는 문자열만 출력하는 예제

LIST1 = ["apple", "banana", "kiwi", "orange"]
LIST2 = [i for i in LIST1 if len(i) > 5]
print(LIST2)

# 복수개의 리스트를 중첩한 내장도 가능

LIST1 = [1,2]
LIST2 = [1,-2,3,-4]
LIST3 = [x*y for x in LIST1 for y in LIST2]
print(LIST3)
```

```
['banana', 'orange']
[1, -2, 3, -4, 2, -4, 6, -8]
```

filter() - 1 (*)

- filter() : 리스트 내장의 if 문이 조건식을 가지고 필터링하는 것과 비슷한 기능하는 내장함수

```
# filter() 내장함수
filter(<조건문 함수>, <순환가능객체>)
```

  - 조건문함수 : 필터링 방법 / 조건을 제공 , 'None' 을 입력시 필터링 없음
  - 순환가능객체 : 리스트 등의 필터링 할 순환가능객체
  - **반환값이 순환자 (iterator)** 로서 , list() 이나 tuple() 내장함수를 사용하여 , 순환자가 지시하고 있는 메모리에 담긴 정보를 다시 리스트나 튜플 객체로 담아낸다 .

```
>>> LIST = [10, 20, 30, 40]
>>> def BiggerThan20(i):
...     return i > 20
...
>>> Iter = filter(BiggerThan20, LIST)
>>> for i in Iter:
...     print("Iter's item: {0}".format(i))
...
Iter's item: 30
Iter's item: 40
```

```
>>> Iter
<filter object at 0x7f3d7ac627b8>
```

```
>>> LIST2 = list(filter(BiggerThan20, LIST))
>>> LIST2
[30, 40]
```

filter() - 2 (*)

filter()의 활용

```
# 리스트를 정의
l = [1,2,3,4,5]

# 필터에 사용할 필터링함수를 정의
def filter_func(i):
    return i>3

# 필터함수를 리스트(l)에 적용하여 조건(filter_func)을
# 만족시키는 요소에 대한 순환자(iterF)를 정의
iterF = filter(filter_func,l)
iterF # ==> type of iterF = a iterator(한번만 쓰이고 소진)

# 순환자를 사용하여 필터링된 요소를 순환
for i in iterF:
    print("filtered item of l: {0}".format(i))

# 필터링된 요소의 순환자를 통하여 이 요소만의 리스트를 새로 생성 1
# iterF has been exhausted,
# as used in the for-loop once.
l2 = list(iterF)
l2 # ==> empty

# 필터링된 요소의 순환자를 통하여 이 요소만의 리스트를 새로 생성 2
# filtered list created from a new iterator of filter()
l3 = list(filter(filter_func,l))
l3
```

```
<filter at 0x7f952cced240>

filter

filtered item of l: 4
filtered item of l: 5

[]

[4, 5]
```

zip() - 1 (*)

- zip() : 여러가지 순환 가능 객체들의 원소들을 인덱스에 따른 쌍 (2 개 이상도 가능) 으로 결합하여 , 순환자 객체 (iterator) 를 얻는다 .
- (filter() 과 비슷하게) **반환값은 쌍을 이룬 튜플 객체의 순환자**로서 , list() / tuple() / dict() 내장함수를 사용하여 , 순환자가 지시하고 있는 메모리에 담긴 정보를 다시 리스트나 튜플 혹은 사전 객체로 담아낸다 .

zip()으로 순환 가능 객체들을 결합하여 순환자 얻기

```
k = ['a','b','c']
v = [1, 2, 3]
iterZ = zip(k,v)
print (iterZ)
type(iterZ)
for i in iterZ:
    print (i)
```

```
<zip object at 0x7f328ff17888>
zip
('a', 1)
('b', 2)
('c', 3)
```

zip()으로 순환 가능 객체들을 결합하여 리스트로 만들기

```
k = ['a','b','c']
v = [1, 2, 3]
l = list(zip(k,v))
print (l)
```

```
[('a', 1), ('b', 2), ('c', 3)]
```

zip()으로 순환 가능 객체들을 결합하여 튜플로 만들기

```
k = ['a','b','c']
v = [1, 2, 3]
t = tuple(zip(k,v))
print (t)
```

```
(('a', 1), ('b', 2), ('c', 3))
```

zip()으로 순환 가능 객체들을 결합하여 사전으로 만들기

```
k = ['a','b','c']
v = [1, 2, 3]
d = dict(zip(k,v))
print (d)
```

```
{'c': 3, 'b': 2, 'a': 1}
```

zip() - 2 (*)

zip()으로 3개의 리스트를 엮기 1

```
k = ['a','b','c']
v1 = [1, 2, 3]
v2 = [11, 12,13]
l = list(zip(k,v1,v2))
print (l)
```

```
[('a', 1, 11), ('b', 2, 12), ('c', 3, 13)]
```

zip()으로 3개의 리스트를 엮기 2

```
k = ['a','b','c']
v1 = [1, 2, 3]
v2 = [11, 12,13]
d = dict(zip(k,list(zip(v1,v2))))
d
```

```
{'a': (1, 11), 'b': (2, 12), 'c': (3, 13)}
```

unzip a list of tuples

```
k = ['a','b','c']
v = [1, 2, 3]
l = list(zip(k,v))
k2, v2 = zip(*l)
k2
v2
```

```
('a', 'b', 'c')
```

```
(1, 2, 3)
```

zip() - 3

- **Python2** 에서의 zip() 메소드는 순환자로서의 zip 객체가 아닌 **리스트 객체를 직접 반환한다.**

zip() in Python2 returns a list object

```
k = ['a','b','c']
v = [1, 2, 3]
l = zip(k,v)
l
type(l)
```

```
[('a', 1), ('b', 2), ('c', 3)]
```

```
list
```

zip() - 4

- **Python3** 에서의 zip() 메소드가 반환하는 zip 객체는 순환자 (iterator) 로서 , for 문이나 순환가능객체 생성함수 (list()/tuple()/dict()...) 등에서 한 번 사용 후에는 소진된다 .

zip() in Python3 returns an iterator

```
k = ['a','b','c']
v = [1, 2, 3]
iterZ = zip(k,v)
l = list(iterZ)
l
d = dict(iterZ)
d # iterZ has been exhausted, in list() & d is empty.
d2 = dict(zip(k,v))
d2 # dictionary created using a new zip iterator
```

```
[('a', 1), ('b', 2), ('c', 3)]
```

```
{}
```

```
{'a': 1, 'b': 2, 'c': 3}
```

map() (*)

- map() : 순환가능객체의 요소를 순회하면서 , 주어진 매핑 함수에 대하여 각 요소에 대응되는 함수값을 계산하고 결과를 순환자로 반환한다 .

```
# map()

map(<함수이름>, <순환가능객체>)
```

```
>>> LIST = [1, 2, 3, 4, 5]
>>> def add100(i):
...     return i+100
...
>>> IterM = map(add100, LIST)
>>> for i in IterM:
...     print("IterM's item: {0}".format(i))
...
IterM's item: 101
IterM's item: 102
IterM's item: 103
IterM's item: 104
IterM's item: 105
```

```
>>> LIST_added = list(map(add100,LIST))
>>> LIST_added
[101, 102, 103, 104, 105]
>>>
>>> LIST_added2 = list(map(lambda i:i+100, LIST))
>>> LIST_added2
[101, 102, 103, 104, 105]
```

파이썬 함수 (메소드)
Function (Method)

파이썬 함수 (메소드) 의 정의 (*)

- 여러 종류의 인자를 입력받아 프로그램에 필요한 여러가지 일을 하나로 묶어서 , 결과값을 반환한다 . 메소드 (method) 라고도 부른다 .

```
# 파이썬 함수(함수 객체)를 선언하는 문법
def <함수명>(인자1, 인자2, ..., 인자n):
    <명령구문>
    return <반환값>
```

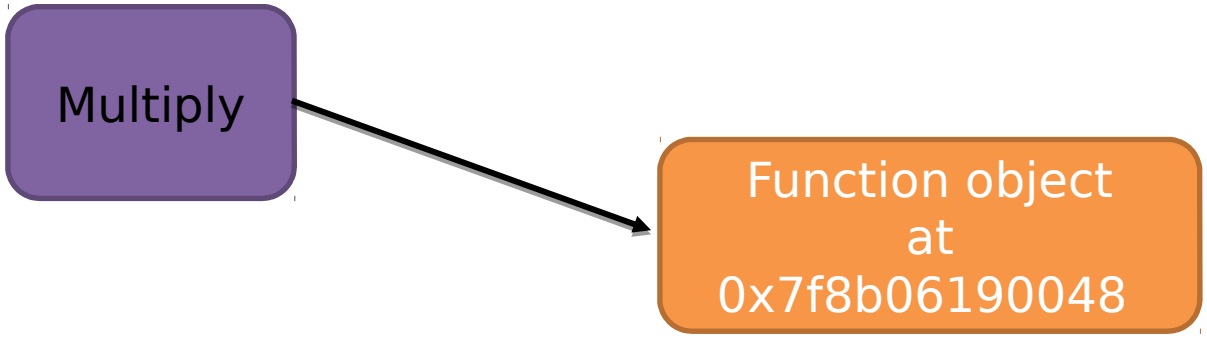
- 함수의 선언은 def 로 시작 , 콜론 (:) 으로 끝나며 , 함수내용의 시작과 끝은 들여쓰기로 구분함 .
- Return 은 함수를 종료하고 < 반환값 > 을 전달하는데 사용 . return 이 없는 경우 None 을 반환 .
- 다른 언어들과 달리 , 함수가 필요할때 바로 선언하고 사용가능 .

```
>>> def Multiply(a, b):
...     return a*b
...
>>> Multiply
<function Multiply at 0x7f8b06190048>
>>> Multiply(2,3)
6
```

파이썬 함수의 정의

```
>>> def Multiply(a, b):
...     return a*b
...
>>> Multiply
<function Multiply at 0x7f8b06190048>
>>> Multiply(2,3)
6
```

- 'Multiply' => def 로 함수 객체 (function object) 가 생성될 때 , 이 객체를 참조하는 레퍼런스 .
- 'function Multiply at 0x7f8b06190048' => 'Multiply' 라는 레퍼런스 이름을 가지는 함수 객체가 메모리 주소 '0x7f8b06190048' 에 생성되었다 .



파이썬 함수의 정의

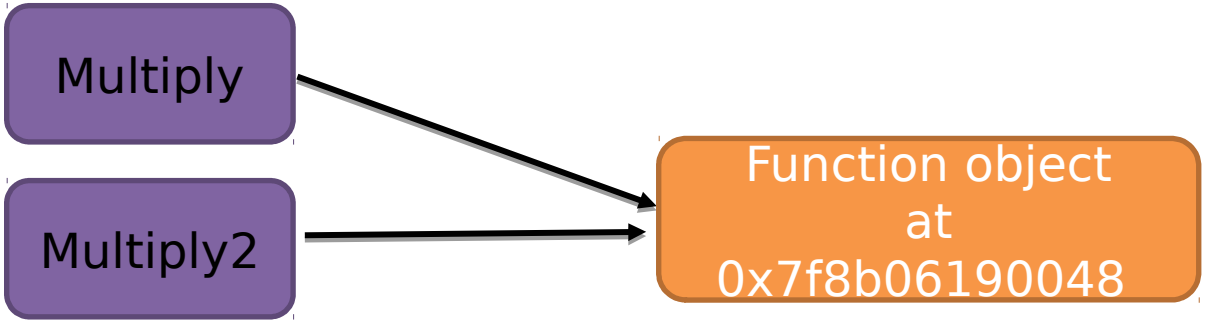
```
...
>>> globals()
```

{... 'Multiply': <function Multiply at 0x7f3d7ac64048>, ... }

```
>>> Multiply2 = Multiply
>>> Multiply2(2,3)
6
>>> globals()
```

{... 'Multiply': <function Multiply at 0x7f3d7ac64048>, ...
'Multiply2': <function Multiply at 0x7f3d7ac64048>, ... }

- Global() 내장 함수 global() 을 사용하면, 현재 생성된 함수 객체들을 볼 수 있다.
- 하나의 함수 레퍼런스는 다른 레퍼런스 (ex> 'Multiply2') 에 할당할 수 있다.
- 이때 또다른 함수 객체로로서 복사되는 것이 아닌, 똑같은 함수객체를 참조하는 또다른 레퍼런스가 된다.



내장함수 목록 확인하기 : dir(__builtins__)

- '__builtins__' 는 내장 영역의 이름이 저장되어 있는 리스트이다 . 이를 dir() 이라는 내장함수를 이용하여 볼 수 있다 .

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

```
>>> LIST = list(range(1,11))
>>> sum(LIST)
55
```

Return (*)

- 함수를 종료하고 해당 함수를 호출한 곳으로 되돌아가게 한다. return 은 어떤 종류의 객체도 반환할 수 있으며 (오직 한 개), 이때 여러가지 값을 튜플 객체로 묶어서 반환할 수도 있다.

```
>>> def swap(a, b):
...     return b, a
...
>>> swap(10, 20)
(20, 10)
>>> c, d = swap(10, 20)
>>> c
20
>>> d
10
```

```
>>> x = swap(10,20)
>>> type(x)
<class 'tuple'>
>>> x
(20, 10)
```

- return 이 없거나, return 만 적었을 때도 함수가 종료되며, None 객체를 돌려준다.

```
>>> def setvalue(x):
...     y = x
...
>>> print(setvalue(10))
None
```

이름공간 (namespace) & 스코핑 룰 (*)

- 이름공간 (namespace)

프로그램에 쓰이는 이름이 저장되는 공간. 예를 들면 ‘ a=[1,2,3]’ 라는 구문을 통해서 리스트 객체가 메모리 공간에 생기는 순간, a 라는 이름이 이름공간에 저장되고, a 라는 이름을 가지고 [1,2,3] 값을 가진 리스트 객체에 접근하게 된다.

- 함수는 별도의 이름공간을 가진다.

함수 내부에서 사용되는 변수는 일단 함수 내부 고유의 이름공간 (지역영역 , local scope) 을 참조한다. 이를 찾지 못하면 상위 이름공간에서 이름을 찾는데, 우선 함수밖 (전역영역 ,global scope) 에서 찾고, 이후 내장영역 (built-in scope) 에서 찾는다

- 지역변수 (local variable):

함수 안에서 정의되어 사용되고 사라지는 변수

- 전역변수 (global variable):

함수 밖에서 정의되어 사용되는 변수

```
>>> x = 1
>>> def func(a):
...     return a+x
...
>>> func(1)
2
```

```
>>> def func2(a):
...     x = 2
...     return a+x
...
>>> func2(1)
3
```

이름공간 (namespace) & 스코핑 룰 (*)

- 스코핑 룰 : 이름공간 (namespace) 검색 순서

Local => Global => Built-in

- 지역에서 전역변수를 사용하고 싶을때 :

지역에서의 global 선언을 통하여, 전역변수에 접근하여 처리.

```
>>> g = 1
>>> def testScope(a):
...     global g
...     g = 2
...     return a+g
...
>>> testScope(1)
3
>>>
>>> g
2
```

인자 전달 (*)

- 파이썬에서는 몇 가지 특별한 인자 전달 모드를 지원한다

1) 기본 인자

```
>>> def Values(a=10, b=20):
...     return a+b
...
>>> Values()
30
>>> Values(1,2)
3
```

2) 키워드 인자

```
>>> def FullName(given_name, family_name):
...     full_name = family_name + given_name
...     return full_name
...
>>> FullName(family_name="Cho", given_name="Wonsang")
'ChoWonsang'
```

인자 전달 (*)

3) 가변 인자 (리스트) : 인자 앞에 * 를 붙이면 정해지지 않은 수의 리스트를 인자로 받을수 있음

```
>>> def myarg(*arg):
...     print(type(arg))
...     print(arg)
...
>>> myarg(1,2,3,4,5)
<class 'tuple'>
(1, 2, 3, 4, 5)
```

4) 정의되지 않은 인자 (사전) : 인자 앞에 ** 를 붙이면 정해지지 않은 수의 사전을 인자로 받을 수 있음

```
>>> def myarg2(**arg):
...     print(type(arg))
...     print(arg)
...
>>> myarg2(a='1', b='2', c='3')
<class 'dict'>
{'c': '3', 'b': '2', 'a': '1'}
```

2 차 (n02) 실습과제 (exercise)

실습목표 :

- 파이썬에서의 여러가지 순환객체와 이를 사용한 반복문의 개념을 익힌다
- 리스트 내장 (List Comprehension) 과 더불어 , 제어와 반복에 관련한 유용한 함수들의 활용방법을 익힌다
- 파이썬 메소드 / 함수의 개념을 익히고 작성해본다 .

실습과제 :

- 본 ppsml_note_03 (3 주차 강의) 슬라이드에서 , 제목에 (*) 가 표시된 항목들의 파이썬 스크립트 예제들을 , 자신만의 Jupyter notebook 으로 정리하여 미리 연습 및 테스트 (변형 가능) 해보고 노트북 파일 (.ipynb) 을 제출한다 .

  (due 9/21(금) 24:00, 파일이름형식 준수)

2 차 (n02) 정규과제 (homework)

정규과제 :

- 9/17(월)~9/21(금) 사이 ETL 공지
- due 9/28(금), 파일이름형식 준수