데이터 사이언스와 파이썬
코드 구조

2020년 2학기
데이터사이언스융합전공

남세진
jordse@gmail.com

Contents

비교(if, if else, if elif else)

루프(while, for)

함수

제너레이터, 네임스페이스 스코프

코멘트 달기

- 프로그램에서 코멘트는 인터프리터에 의해 무시되는 텍스트의 한 부분임.
  - 코드를 설명하거나 나중에 어떤 문제를 고치기 위해 표시를 하는 등 다양한 목적으로 코멘트를 사용할 수 있음.
  - # 문자를 이용해서 코멘트를 표시함.

```
# 이 것은 주석 처리
print('hello world')

# 문자열안에 #을 쓰는 경우, 주석의 시작이 아닌 그냥 문자
print('hello # world')
```

```
hello world
hello # world
```

라인 유지 : ₩

- 한 문장이 길어지는 경우 백슬래시(₩) 문자를 이용하여 다음 라인에 문장까지 연장할 수 있음

```
var_a = 'abcd' + \
    'efg' + \
    'hijklmn'

var_b = '1234' \
    '5678' \
    '90'
print(var_a)
print(var_b)
```

```
abcdefghijklmn
1234567890
```

비교 : if, elif, else

- if와 else는 조건이 참(True)인지 거짓(False)인지 확인하는 python의 선언문
  - if, else문은 콜론으로 끝남
  - if, else문이 참인 경우, 실행되는 문장은 들여쓰기를 이용하여 표현
  - 들여쓰기를 보면 if와 else 부분이 어떻게 짝을 이루는지 알 수 있음
  - 들여쓰기의 크기는 IDE에서 설정 가능(4가 적절)

```
bool_var = True
if bool_var:
    print('True')
else:
    print('False')
```

```
True
```

```
bool_var1 = True
bool_var2 = True

if bool_var1:
    if bool_var2:
        print('var1, var2')
    else:
        print('var1')
else:
    if bool_var2:
        print('var2')
    else:
        print('none')
```

```
var1, var2
```

coding style : PEP 8 Coding Style guide in Python

- https://www.geeksforgeeks.org/pep-8-coding-style-guide-python/

1. Use 4-space indentation and no tabs.

Examples:

```
# Aligned with opening delimiter.
grow = function_name(variable_one, variable_two,
                     variable_three, variable_four)
```

```
# First line contains no argument. Second line onwards
# more indentation included to distinguish this from
# the rest.
def function_name(
        variable_one, variable_two, variable_three,
        variable_four):
    print(variable_one)
```

elif : 두 개 이상의 조건 테스트가 있는 경우

- elif (else if를 의미)는 두 개 이상의 조건 테스트를 하는 경우 사용

```
color = 'red'
if color == 'green':
    print('color : green')
elif color == 'red':
    print('color : red')
elif color == 'blue':
    print('color : blue')
else:
    print('color : n/a')
```

```
color : red
```

if문에서 주로 사용하는 비교 연산자 / False 값

- '==' : 두 값이 같은지 확인할 때 사용('=' 는 변수에 값을 할당할 때 사용)
- 부울 연산자는 비교 연산자의 우선순위가 낮음

비교 연산자	의미
==	같다.
!=	다르다.
<	보다 작다.
<=	보다 작거나 같다.
>	보다 크다.
>=	보다 크거나 같다.
in ...	멤버십

```
>>> x = 7
>>> x == 5
False
>>> x == 7
True
>>> 5 < x
True
>>> x < 10
True
>>> 5 < x and x < 10
True
>>> (5 < x) and (x < 10)
True
>>> 5 < x and x > 10
False
>>> 5 < x and not x > 10
True
>>> 5 < x < 10
True
>>> 5 < x < 10 < 1
False
```

- 다음 예는 모두 False로 간주

요소	False
null	None
정수 0	0
부동소수점수 0	0.0
빈 문자열	''
빈 리스트	[]
빈 튜플	()
빈 딕셔너리	{}
빈 셋	set()

```
>>> bool(empty_list)
False
>>> bool(empty_set)
False
>>> bool(empty_tuple)
False
>>> bool(empty_dict)
False
```

```
>>> empty_list = []
>>> empty_set = set()
>>> empty_tuple = ()
>>> empty_dict = {}
>>> bool(empty_list)
```

반복 : While, for

- While문은 조건이 만족할 때 까지 반복

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

java의 for 문

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

  - 조건문은 콜론으로 끝나야 함
  - 반복되는 명령문의 블록은 tap으로 구분

- Break
  - 반복을 중단할 때 사용
  - 무한 루프 속에 사용하거나, 디버깅시에도 유용하게 사용

```
while True:
    stuff = input('String to capitalize [type q to quit]: ')
    if stuff == "q":
        break
    print(stuff.capitalize())
```

```
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: q
```

continue / break

- continue
  - 반복문을 중단하지 않고, 다음 루프로 건너뛸 때 사용
- break
  - 반복문을 중단할 때 사용
- while ~ else
  - while문이 거짓이 되는 경우, else문이 실행됨

```
while 조건식 :
        실행문1
else:
        실행문2
```

```
while True:
    value = input('Integer, please [q to quit] : ')
    if value == 'q':
        break
    number = int(value)
    if number % 2 == 0:
        continue
    print(number, "squared is ", number * number)
```

```
Integer, please [q to quit] : 3
3 squared is  9
Integer, please [q to quit] : q
```

```
numbers=[1,3,5]
position = 0
while position < len(numbers):
    number = numbers[position]
    if number % 2 == 0:
        print('Found even number', number)
        break
    position += 1
else:
    print("No even number found")
```

for

- for 문은 list, tuple, dictionary, set, 문자열과 같은 순회 가능한(iterable)한 시퀀스를 대상으로 명령문을 반복할 때 사용

```
for [변수] in [list, tuple, dictionary, set,  문자열]:
        실행문
```

- for 문을 이용한 간결한 표현

```
numbers = [1,2,3,4,5]
index = 0
while index < len(numbers):
    print(numbers[index])
    index += 1

for number in numbers:
    print(number)
```

while문을 사용한 numbers(리스트) 출력

for 문을 이용한 간결한 표현

for 문의 다양한 예제

- tuple을 이용한 for 문

```
temp_tuple = (11,22,33,44,55)
for item in temp_tuple:
    print(item)
```

- 문자열을 이용한 for 문

```
for ch in 'abcde':
    print(ch)
```

- 딕셔너리를 이용한 for 문

```
temp_dict = {1:'one', 2:'two', 3:'three'}
for key in temp_dict:
    print(key)
```

- tuple을 이용한 for 문

```
temp_set = set()
temp_set.add(1)
temp_set.add(2)
temp_set.add(3)
for item in temp_set:
    print(item)
```

for ... else / zip

- for 문이 종료되면 else문을 실행
  - 만약 for문의 실행중 break문으로 중단되면, else문이 실행되지 않음

```
numbers = [1,2,3,4,5]
for x in numbers:
  print(x)
else:
  print("Finally finished!")
```

```
numbers = [1,2,3,4,5]
for x in numbers:
  if x == 4:
      break
  print(x)
else:
  print("Finally finished!")
```

- zip 함수를 사용한 시퀀스를 동시에 순회
  - zip() : Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument s equences or iterables.

```
>>> x = [1,2,3]
>>> y = [4,5,6]
>>> list(zip(x,y))
[(1, 4), (2, 5), (3, 6)]
```

```
>>> x = [1,2,3]
>>> y = [4,5]
>>> list(zip(x,y))
[(1, 4), (2, 5)]
```

range() : 숫자 시퀀스 생성

class **range**(*stop*)
class **range**(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, `ValueError` is raised.

- range(start, stop step)
  - step 값을 제공하지 않으면, 기본값은 1
  - start 값을 제공하지 않으면, 기본값은 0
  - 만약 step값이 0이면, 예외 발생

```
>>> for x in range(3):
...     print(x)
...
0
1
2
```

```
>>> for x in range(1,4):
...     print(x)
...
1
2
3
```

```
>>> for x in range(2,-1, -1):
...     print(x)
...
2
1
0
```

Comprehension (함축)

- 하나 이상의 이터레이터로부터 파이썬의 자료구조를 만드는 콤팩트한 방법
  - 반복문과 조건 테스트를 결합한 간결한 표현이 가능
  - 파이썬의 아름다운 모습을 볼 수 있는 문법

- 리스트 Comprehension

```
numbers = []
numbers.append(1)
numbers.append(2)
numbers.append(3)

print(numbers)

numbers = []
for number in range(1,4):
    numbers.append(number)
print(numbers)

numbers = list(range(1,4))
print(numbers)
```

```
numbers = [number for number in range(1,4)]
print(numbers)

numbers = [number*number for number in range(1,4)]
print(numbers)
```

[표현식 for 항목 in 순회 가능한 객체]
[표현식 for 항목 in 순회 가능한 객체 if 조건]

Comprehension (함축)

- Comprehension을 이용한 중첩된 루프
  - [표현식 for 항목1 in 중첩 가능한 객체 for 항목2 in 중첩 가능한 객체]

```
alphas = ['a', 'b', 'c', 'd']
nums = [1, 2, 3, 4]
for ch in alphas:
    for num in nums:
        print(ch, num)
```

```
alphas = ['a', 'b', 'c', 'd']
nums = [1, 2, 3, 4]
result = [(ch, num) for ch in alphas for num in nums]
for ch, num in result:
    print(ch, num)
```

딕셔너리/셋 Comprehension

- 딕셔너리 Comprehension
  - {키표현식 : 값_표현식 for 표현식 in 순회 가능한 객체}

```
word = 'letters'
letter_counts = {letter:word.count(letter) for letter in word}
print(letter_counts)

words = ['ab', 'cd', 'ef', 'cd', 'gg', 'cd', 'cd']
word_counts = {word:words.count(word) for word in words}
print(word_counts)

words = ['ab', 'cd', 'ef', 'cd', 'gg', 'cd', 'cd']
word_counts = {word:words.count(word) for word in set(words)}
print(word_counts)
```

```
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
{'ab': 1, 'cd': 4, 'ef': 1, 'gg': 1}
{'ab': 1, 'gg': 1, 'cd': 4, 'ef': 1}
```

- 셋 Comprehension
  - {표현식 for 항목 in 순회 가능한 객체}

```
even_num = {number for number in range(1,11) if number % 2 == 0}
print(even_num)
print(type(even_num))
```

```
{2, 4, 6, 8, 10}
<class 'set'>
```

함수

- 규모가 있는 프로그램을 작성하는 몇 가지 방법
  - 코드의 재사용을 위한 함수를 사용.
  - 함수는 이름이 붙여진 코드 조각이고, 다른 것으로부터 분리되어 있음.
  - 함수는 입력 매개변수로 모든 타입을 여러 개 취할 수 있음. 그리고 반환값으로 모든 타입을 여러 개 반환할 수 있음.
- 함수의 정의 (define)와 호출(call)

parameter vs argument, 그리고 잠시 생각해볼 문제

- argument
  - 함수로 전달한 값(value)
- parameter
  - 함수의 입력 변수 명(variable)

```
def plusone(val):
    val += 1
    return val

a = 3
plusone(a)
print(a)
```

- 토론하기 : 아래의 결과를 설명해 보자

```
def plusone(val):
    val += 1
    return val

def plusones(vals):
    for index in range(len(vals)):
        vals[index] += 1


a = 3
print (a)
plusone(a)
print(a)

numbers = [1,2,3,4]
print(numbers)
plusones(numbers)
print(numbers)
```

위치 인자 / 키워드 인자 / 기본 매개변수값 지정

- 인자 값 순서대로 매개변수에 복사하는 방식
- 인자의 순서로 인한 혼동을 피하기 위해 매개변수에 상응하는 이름을 지정 가능
- 매개변수에 기본값을 지정할 수 있음

```
def menu(one, two, three, four=4):
    return {'one':one, 'two': two, 'three':three, 'four':four}

print(menu(1, 2, 3))
print(menu(one=1, three=3, two=2))
print(menu(one=1, two=2, three=3, four=5))
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
{'one': 1, 'two': 2, 'three': 3, 'four': 5}
```

기본 매개변수값을 지정할 때 실수하기 쉬운 것

- 기본 인자값은 함수가 실행될 때 계산되지 않고, 함수를 정의할 때 계산됨

```
def buggy(arg, result=[]):
    result.append(arg)
    print(result)

buggy('a')
buggy('b')

def works(arg):
    result = []
    result.append(arg)
    return result

print(works('a'))
print(works('b'))
```

```
['a']
['a', 'b']
['a']
['b']
```

위치 인자 모으기 * / 키워드 인자 모으기 **

- *는 매개변수에서 위치 인자 변수들을 튜플로 묶음
- **는 키워드 인자를 딕셔너리로 묶음
- *, ** 위치매개변수를 같이 사용하려면 순서대로 배치해야 함

```python
def print_args(arg1, arg2, *others):
    print ("arg1 : ", arg1)
    print ("arg2 : ", arg2)
    print ("others : ", others)


print_args('a', 'b', 'c', 'd', 'e')


def print_kwargs(**kwargs):
    print ('kwargs : ', kwargs)

print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
```

```
arg1 :  a
arg2 :  b
others :  ('c', 'd', 'e')
kwargs :  {'wine': 'merlot', 'entree': 'mutton', 'dessert': 'macaroon'}
```

```python
def print_args_kwargs(arg1, *others, **kwargs):
    print ('arg1 : ', arg1)
    print ('others : ', others)
    print ('kwargs : ', kwargs)

print_args_kwargs('1', '2', '3', four='4', five='5')
print_args_kwargs('1', four='4', five='5', '2')
```

```
arg1 :  1
others :  ('2', '3')
kwargs :  {'four': '4', 'five': '5'}
```

```
  File "/Users/rtdatum/PycharmProjects/skku-lecture3/lecture05.py", line 22
    print_args_kwargs('1', four='4', five='5', '2')
                                               ^
SyntaxError: positional argument follows keyword argument
```

docstring

- 함수 몸체 시작 부분에 문자열을 포함시켜 함수 정의에 문서를 붙일 수 있음

```
def echo(anything):
    'echo returns its input argument'
    return anything

def print_if_true(thing, check):
    '''
    Prints the first if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    :param thing:
    :param check:
    :return:
    '''
    if check:
        print(thing)

print(print_if_true.__doc__)
```

```
    Prints the first if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    :param thing:
    :param check:
    :return:
```

First Class Function (일급 함수)

- 함수를 다른 변수와 동일하게 다루는 언어는 **일급 함수**를 가졌다고 표현
  - 함수를 다른 함수에 매개변수로 제공 가능
  - 함수가 함수를 반환
  - 함수를 변수에 할당 가능
- 프로그래밍 언어가 함수를 First class citizen으로 취급

```python
def answer(name):
    print('hello', name)

def run_something(func, str):
    return func(str)

run_something(answer, "sejin")


def sum_args(*args):
    return sum(args)

def run_with_positional_args(func, *args):
    return func(*args)

print(run_with_positional_args(sum_args, 1, 2, 3, 4))
```

```
hello sejin
10
```

내부함수

- 루프나 코드 중복을 피하기 위해 또 다른 함수 내에서 반복적인 작업을 수행할 때 유용

```
def outer(a, b):
    def inner(c, d):
        return c + d
    return inner(a, b)

print (outer(4, 7))

def knights(saying):
    def inner(quote):
        return "We are the knights who say : '%s'" % quote
    return inner(saying)

print(knights("Ni"))
```

```
11
We are the knights who say : 'Ni'
```

클로저

- 프로그래밍 언어에서의 클로저란 퍼스트클래스 함수를 지원하는 언어의 네임 바인딩 기술
- 클로저는 어떤 함수를 함수 자신이 가지고 있는 환경과 함께 저장한 레코드
  - 함수가 가진 프리변수(free variable)를 클로저가 만들어지는 당시의 값과 레퍼런스에 맵핑하여 주는 역할
  - 클로저는 일반 함수와는 다르게, 자신의 영역 밖에서 호출된 함수의 변수 값과 레퍼런스를 복사하고 저장한 뒤, 이 캡처한 값들에 액세스할 수 있게 도와줌

```
def outer_func():
    message = 'Hi'

    def inner_func():
        print(message)

    return inner_func()

outer_func()

def outer_func():
    message = 'Hi'

    def inner_func():
        print(message)

    return inner_func

my_func = outer_func()
my_func()
```

```
Hi
Hi
```

```
def outer_func():
    message = 'Hi'

    def inner_func():
        print(message)

    return inner_func

my_func = outer_func()

print(my_func)
print(dir(my_func))
print(type(my_func.__closure__))
print(my_func.__closure__)
print(my_func.__closure__[0])
print(dir(my_func.__closure__[0]))
print(my_func.__closure__[0].cell_contents)
```

```
<function outer_func.<locals>.inner_func at 0x7fb187a005e0>
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__',
 '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__get__', '__getattribute__', '__globals__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__',
 '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
<class 'tuple'>
(<cell at 0x7fb18799af40: str object at 0x7fb187a03670>,)
<cell at 0x7fb18799af40: str object at 0x7fb187a03670>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'cell_contents']
Hi
```

lambda()

- 람다 함수는 단일문으로 표현되는 익명 함수

```
def inc(n):
    return lambda x: x + n

f = inc(2)
print(f(12))

g = inc(4)
print(g(12))

f2 = lambda x, y: x + y
g2 = lambda x: x**2

print(f2(1,1))
print(g2(10))
```

```
14
16
2
100
```

제너레이터

- 파이썬의 시퀀스를 생성하는 객체
  - 제너레이터로 전체 시퀀스를 한 번에 메모리에 생성하고 정렬할 필요 없이, 큰 규모의 시퀀스를 순회할 수 있음
    - 예) range() 함수 : 일련의 정수를 생성하고 메모리에 제한적인 리스트를 반환
  - 일반 함수는 return문으로 값을 반환하지만, 제너레이터는 yield문으로 값을 반환
  - 마지막으로 호출된 항목을 기억하고 다음 값을 반환

```
def my_range(first=0, last=10, step=2):
    number = first
    while number < last:
        yield number
        number += step

ranger = my_range(1, 10)
type(ranger)
for x in ranger:
    print(x)
```

```
1
3
5
7
9
```

데커레이터(Decorator)

- 데코레이터는 함수를 수정하지 않은 상태에서 추가 기능을 구현할 때 사용
  - @staticmethod, @classmethod, @abstractmethod
  - 데코레이터는 @로 표시

```
def trace(func):
    def wrapper():
        print(func.__name__, '함수 시작')
        func()
        print(func.__name__, '함수 끝')
    return wrapper

def func1():
    print('hello')

def func2():
    print('world')

new_func1 = trace(func1)
new_func2 = trace(func2)

new_func1()
new_func2()
```

```
func1 함수 시작
hello
func1 함수 끝
func2 함수 시작
world
func2 함수 끝
```

```
def document_it(func):
    def new_function(*args, **kwargs):
        print('function : ', func.__name__)
        print('Positional arguments : ', args)
        print('Keyword arguments:', kwargs)
        result = func(*args, **kwargs)
        print('Result:', result)
        return result
    return new_function

def add_ints(a, b):
    return a + b

print(add_ints(3,5))
new_add_ints = document_it(add_ints)
new_add_ints(3,5)
```

```
8
function :  add_ints
Positional arguments :  (3, 5)
Keyword arguments: {}
Result: 8
```

데코레이터(Decorator)

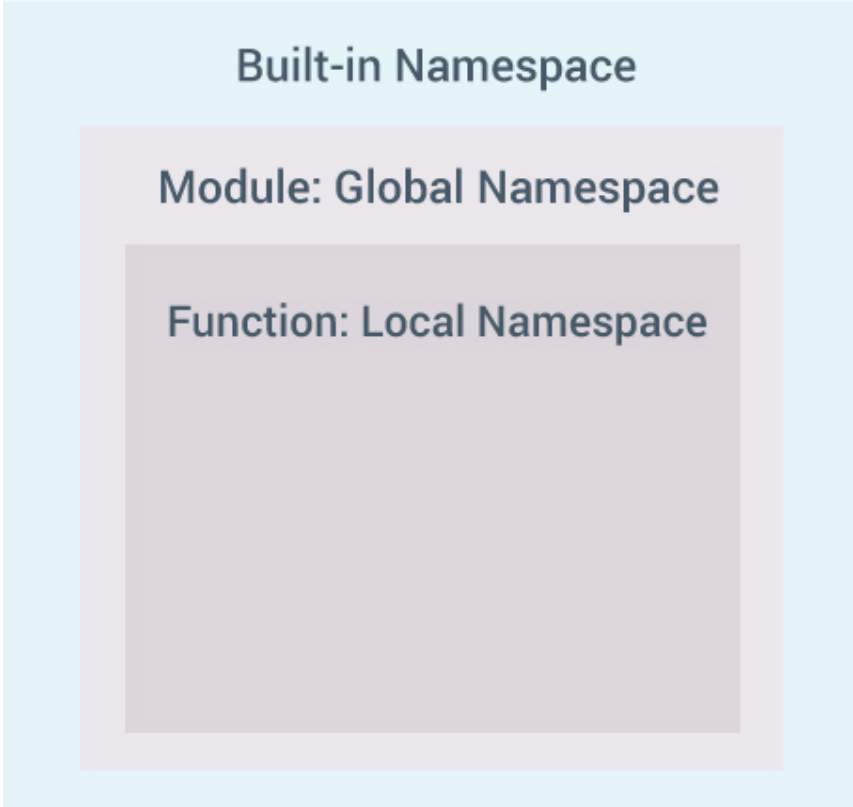
```python
def document_it(func):
    def new_function(*args, **kwargs):
        print('function : ', func.__name__)
        print('Positional arguments : ', args)
        print('Keyword arguments:', kwargs)
        result = func(*args, **kwargs)
        print('Result:', result)
        return result
    return new_function

def square_it(func):
    def new_function(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * result
    return new_function

@document_it
def plus_one(arg1):
    return arg1 + 1

@document_it
@square_it
def plus_two(arg1):
    return arg1 + 2

print(plus_one(1))
print(plus_two(2))
```

```
function :  plus_one
Positional arguments :  (1,)
Keyword arguments: {}
Result: 2
2
function :  new_function
Positional arguments :  (2,)
Keyword arguments: {}
Result: 16
16
```

네임스페이스와 스코프

- 네임스페이스(namespace, 이름공간)란 프로그래밍 언어에서 특정한 객체(Object)를 이름(Name)에 따라 구분할 수 있는 범위
  - 프로그래밍언어에서는 네임스페이스라는 개념을 도입하여, 특정한 하나의 이름이 통용될 수 있는 범위를 제한

- 네임스페이스의 종류
  - 전역 네임스페이스: 모듈별로 존재하며, 모듈 전체에서 통용될 수 있는 이름들이 소속됨
  - 지역 네임스페이스: 함수 및 메서드 별로 존재하며, 함수 내의 지역 변수들의 이름들이 소속
  - 빌트인 네임스페이스: 기본 내장 함수 및 기본 예외들의 이름들이 소속. 파이썬으로 작성된 모든 코드 범위가 포함



네임스페이스와 스코프

```
global_number = 'one'
print("id of global_number", id(global_number))

def print_one():
    print('inside function:', global_number)
    print('id of global_number : ', id(global_number))

print_one()

def change_one():
    global_number = 'two'
    print('inside function:', global_number)
    print('id of global_number : ', id(global_number))

change_one()
print_one()
```

```
id of global_number 140471999379888
inside function: one
id of global_number :  140471999379888
inside function: two
id of global_number :  140471999380144
inside function: one
id of global_number :  140471999379888
```

네임스페이스와 스코프

```
global_number = 'one'
print("id of global_number", id(global_number))

def change_two():
    global global_number
    global_number = 'three'
    print('global function:', global_number)
    print('id of global_number : ', id(global_number))

change_two()
print(global_number)
```

```
id of global_number 140578148858224
global function: three
id of global_number :  140578148858160
three
```

네임스페이스와 스코프

- 네임스페이스의 내용을 접근하기 위해 사용되는 두 가지 함수
  - locals() : 로컬 네임스페이스의 내용이 담긴 딕셔너리를 반환
  - globals() : 글로벌 네임스페이스의 내용이 담긴 딕셔너리를 반환

```
number = 'one'
def change_value():
    number = 'two'
    print('locals:', locals())

change_value()
print('globals:', globals())
```

```
locals: {'number': 'two'}
globals: {'__name__': '__main__',
'__doc__': None,
'__package__': None,
'__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7f99a1eb5fa0>,
'__spec__': None,
'__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>,
'__file__': '/Users/rtdatum/PycharmProjects/skku-lecture3/lecture14.py',
'__cached__': None,
'number': 'one',
'change_value': <function change_value at 0x7f99a1eba160>}
```

네임스페이스와 스코프

- 변수 스코프(Variable Scope)란 접두어(Prefix)없이 어떤 네임스페이스에 직접 접근이 가능한 프로그래밍의 어떤 부분
- 다른 언어와 다르게 다음과 같은 중첩된 변수 스코프를 확인 할 수 있음
  - 지역 이름들을 포함하는 현재 함수의 스코프(지역 네임스페이스)
  - 전역 이름들을 포함하는 현재 모듈의 스코프(전역 네임스페이스)
  - 빌트인 이름들을 포함하는 최외곽의 스코프(빌트인 네임스페이스)

```
def func1():
    number = 'two'

    def inner_func1():
        number = 'three'
        print("number:", number)

    inner_func1()
    print("number:", number)


number = 'one'
func1()
print("number", number)
```

```
number: three
number: two
number one
```

_의 사용

- 두 언더스코어(__)로 시작하고 끝나는 이름은 파이썬 내의 사용을 위해 예약되어 있어, 변수를 선언할 때 두 언더스코어를 사용하지 않는 것이 좋음
  - *function.__name__ : 함수의 이름*
  - *function.__doc__ : 함수의 docstring*
  - *메인 프로그램 : __main__*

```
number = 'one'
def change_value():
    number = 'two'
    print('locals:', locals())

change_value()
print('globals:', globals())

if __name__ == "__main__":
    # execute only if run as a script
    change_value()
```