

# 2022 LSK-시몬스 언어학학교 - 언어연구를 위한 Python 프로그래밍 (초급)

## 02 함수와 클래스

윤태진 교수  
성신여자대학교 영어영문학과



# 강의안내



01

Function

02

이름없는 함수 Lambda

03

Class

04

String operations



01

# 함수(function)란?

# 함수(function)

- 하나의 특별한 목적의 작업을 수행하기 위해 독립적으로 설계된 프로그램 코드의 집합
- 프로그램에서 특정 작업을 여러 번 반복해야 할 때는 해당 작업을 수행하는 함수를 작성

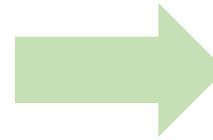


# 반복되는 작업 함수로 만들기

- 재사용
- 가독성
- 유지보수



같은 코드가 반복



함수



함수 정의  
(function  
definition)



함수 호출  
(function call)

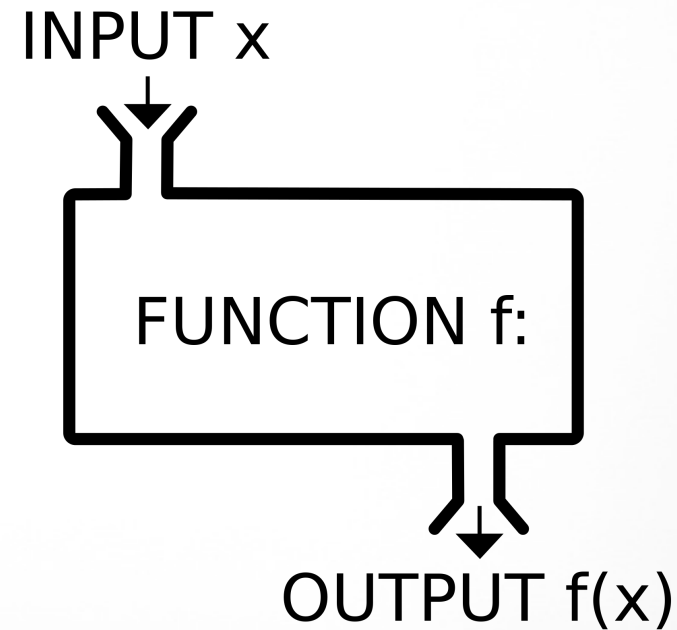


# 함수(function)

$$f(x) = y$$



- 블랙 박스(Black box)
  - 함수는 입력과 출력을 갖는 black box이다.
  - 주어진 입력에 대해서 어떤 과정을 거쳐 출력이 나오는지 숨겨져 있다.



- 함수 이름, 입력, 출력 중요함.

# 함수의 형태

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

```
add(2, 3)
```



함수명 다음 ()는 비어 있기도 하고, 변수(들)이 오기도 하는데, 변수를 매개변수(parameter)라고 함

```
def 함수명 ( 매개변수 ):
```

수행할 명령(들)

...

(return 반환값)

함수 몸체  
(function  
body)

함수 정의  
(function  
definition)

함수 몸체는 반드시 들여쓰기(indentation)를 해 주어야 함.

반환값은 없을 수도 있음



02

# 람다 함수 Lambda Function



# 람다(lambda) 함수 : 이름없는 함수

- 이름없는 한 줄짜리 함수이다.
- 람다 함수는 return 문을 사용하지 않는다.

```
def add(x, y):  
    return x+y  
add(10, 20)
```

30

lambda <인수들>:<반환할 식>

```
lambda x, y: x+y
```

```
<function __main__.<lambda>(x, y)>
```

```
(lambda x, y: x+y)(10, 20)
```

30

# 람다 함수(lambda function)의 모양과 호출

lambda 매개변수1, 매개변수2, ... : 반환값



람다함수    호출

(lambda 매개변수1, 매개변수2, ... : 반환값) (인수1, 인수2, ...)

```
lambda x, y: x+y
```

```
<function __main__.<lambda>(x, y)>
```

```
(lambda x, y: x+y)(10, 20)
```

```
30
```





```
def add(x, y):  
    return x+y  
add(10, 20)
```

30

```
lambda x, y: x+y
```

```
(lambda x, y: x+y)(10, 20)
```

30

```
add = lambda x, y: x+y  
add(10, 20)
```

30

# 왜 람다함수가 필요한거지?

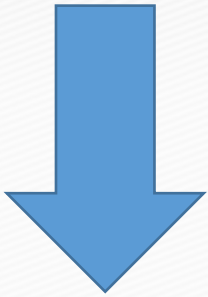
```
print('100과 200의 합 :', (lambda x, y: x + y)(100, 200))
```

100과 200의 합 : 300

```
print('100과 200의 합 :', 100+200)
```

100과 200의 합 : 300

→ 훨씬 쉽고 간단



list 같은 열거 가능한 자료형과 함께 사용하여 함수를 여러 번 호출할 때 사용하면 유용함

# 일반 함수와 람다 함수 비교

## 일반 함수

```
names = ['Alice', 'Paul', 'Bob', 'Tom']  
len_names = []  
for name in names:  
    len_names.append(len(name))  
print(len_names)
```

[5, 4, 3, 3]

## lambda 함수

```
names = ['Alice', 'Paul', 'Bob', 'Tom']  
A = map(len, names)  
print(list(A))
```

[5, 4, 3, 3]





03

# Class

# 객체 지향 프로그래밍(object-oriented programming)

- 잘 설계된 클래스를 이용하여 객체(object)를 만든다.
- 클래스는 속성(instance 변수)과 행위(method)를 가지도록 설계한다.
- 객체는 클래스에서 정의한 속성(attribute)과 행위(behavior)를 수행한다.

상호작용



<https://ac-illust.com/ko/clip-art/704043/붕어빵-먹는-여자>

<http://shopping.interpark.com/product/productInfo.do?prdNo=7780069202&uaTp=1&>



# class와 Instance의 이해

## 클래스(class)

- 속성(instance variable)과 행위(method)들을 모아 놓은 집합체로 객체의 설계도(blueprint)혹은 형틀(template)이다.



class

## 인스턴스(Instance)

- 클래스로부터 만들어진 각각의 객체를 그 클래스의 인스턴스라고 한다.
- 서로 다른 인스턴스는 서로 다른 속성 값을 가질 수 있다.
- 예: int, float, str, list, tuple, set, dict이 모두 class로 구현되었고, 변수에 서로 다른 값들을 저장해서 사용



object; instance

# 클래스(class)

- 객체(object)를 만들기 위한 설계도나 틀
- 클래스의 구성
  - 속성(attribute): 객체를 구성하는 데이터
  - 메소드(method): 속성에 대해 어떤 기능을 수행하는 함수
  - 생성자(Constructor): 객체 생성 시에 자동 호출되는 특별한 메소드
    - 클래스의 `__init__` 메소드를 호출하여 객체를 초기화(initialize)
    - 사용할 클래스의 메모리에 객체를 생성

new

`__init__(self, ... ):`

self: 생성된 객체를 자동으로 참조



# 강아지 클래스 만들기



Attribute

이름: Boo

나이: 12

Method

smiling



Attribute

이름: Buddy

나이: 10

Method

smiling



# 강아지를 클래스를 생성하기 - 속성 추가

```
class Puppy:
    """강아지를 이름과 나이로 표현하는 클래스"""
    """속성은 강아지 객체를 구성하는 데이터임."""
    def __init__(self, name, age): # 생성자
        """강아지 객체를 생성하는 생성자 메소드"""
        self.name = name # self.name과 self.age - 속성
        self.age = age
```

attributes

`__init__(self, ...):`      생성된 Puppy 객체

참조

# 강아지 클래스를 객체로 표현하기

```
boo = Puppy("Boo", 12)
buddy = Puppy("Buddy", 10)
print(boo.name, 'is', boo.age, 'years old.')
print(buddy.name, 'is', buddy.age, 'years old.')
```

```
Boo is 12 years old.
Buddy is 10 years old.
```

서로 다른 인스턴스는 서로  
다른 속성 값을 가질 수 있다.

instances

instance의 속성에 접근

# 강아지 클래스를 객체로 표현하기 - 메소드 추가

```
class Puppy:
    """강아지를 이름과 나이로 표현하는 클래스"""
    """속성은 강아지 객체를 구성하는 데이터임."""
    def __init__(self, name, age): # 생성자
        """강아지 객체를 생성하는 생성자 메소드"""
        self.name = name # self.name과 self.age - 속성
        self.age = age
    def smile(self): # method
        print(self.name, "is smiling.")
```

methods

```
boo = Puppy("Boo", 12)
buddy = Puppy("Buddy", 10)
print(boo.name, 'is', boo.age, 'years old.')
print(buddy.name, 'is', buddy.age, 'years old.')
boo.smile()
buddy.smile()
```

```
Boo is 12 years old.
Buddy is 10 years old.
Boo is smiling..
Buddy is smiling..
```





# 클래스가 포함된 모듈 만들기

# 클래스와 모듈

- 클래스가 저장된 파일을 모듈로 사용할 수 있다.
- 다음의 두 class가 pet.py라는 파일에 저장되어 있다고 설정함.



속성  
이름: kittie  
나이: 5  
메소드:  
wearing a hat





# pet.py 파일 만들어 class가 포함된 모듈 만들기

```
%%writefile pet.py
class Puppy:
    """강아지를 이름과 나이로 표현하는 클래스"""
    """속성은 강아지 객체를 구성하는 데이터임."""
    def __init__(self, name, age): # 생성자
        """강아지 객체를 생성하는 생성자 메소드"""
        self.name = name # self.name과 self.age - 속성
        self.age = age
    def smile(self): # method
        print(self.name, "is smiling.")
class Cat:
    """고양이 이름과 나이로 표현하는 클래스"""
    """속성은 고양이 객체를 구성하는 데이터임."""
    def __init__(self, name, age): # 생성자
        """고양이 객체를 생성하는 생성자 메소드"""
        self.name = name # self.name과 self.age - 속성
        self.age = age
    def wear(self): # method
        print(self.name, "is wearing a hat.")
```



# pet 모듈 import해서 사용하기

```
import pet
doggie = pet.Puppy("Doggie", 8)
kitty = pet.Cat("Kitty", 5)
print(doggie.name, "is", doggie.age, "years old")
doggie.smile()
print(kitty.name, "is", kitty.age, "years old")
kitty.wear()
```

```
Doggie is 8 years old
Doggie is smiling.
Kitty is 5 years old
Kitty is wearing a hat.
```

# class에 저장했던 attributes와 methods가 뭐지?

## help() 이용하기

```
import pet
help(pet)
```

Help on module pet:

NAME

pet

CLASSES

builtins.object

Cat

Puppy

```
class Cat(builtins.object)
```

```
    Cat(name, age)
```

고양이 이름과 나이로 표현하는 클래스

Methods defined here:

```
    __init__(self, name, age)
```

고양이 객체를 생성하는 생성자 메소드

```
    wear(self)
```

docstring의 중요성

## dir() 이용하기

```
print(dir(pet))
print("=="*20)
print(dir(pet.Puppy))
print("=="*20)
print(dir(pet.Cat))
```

```
['Cat', 'Puppy', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']
=====
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'smile']
=====
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'wear']
```





04

**string**

# + : 문자열 연결 (concatenate)



문자열 연결하기 (+)

```
a = 'hello'  
b = 'world'
```

```
c = a + b  
print(c)
```

```
d = a + ' ' + b  
print(d)
```



# string과 int의 연결?

str()  
int()  
float()



```
score = 95  
x = 'I got ' + score + ' in the exam.' # 에러
```

```
x = 'I got ' + str(score) + ' in the exam.'  
print(x)
```

# 문자열 길이 len()

- 문자열의 길이가 가변적
- 문자열의 길이가 얼마나 있는지, 특정 문자(들)이 있는지 알아보고 싶은데...

문자열 길이 - len() 내장함수

```
subject = "programming"  
len(subject)
```

```
subject[-1:-len(subject)-1:-1]
```



# 문자열 포함 관계 in & not in

문자열 포함 - in, not in 연산자

```
subject = "programming"
```

```
'r' in subject
```

```
'gram' in subject
```

```
'abcd' not in subject
```





# 문자열 인덱싱 (indexing)/슬라이싱(slicing)

문자열 객체의 특징

- immutable하다
- 순서가 있는 자료형으로 인덱싱을 이용할 수 있다.

```
greeting = 'hello world'
```



index

negative index

0	1	2	3	4	5	6	7	8	9
h	e	l	l	o		w	o	r	l
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2

# 문자열 indexing

```
print(greeting[7])
```

```
print(greeting[0])
```

```
greeting[0] = 'H' # TypeError 발생
```

- 문자열 객체는 **immutable**하기 때문에 생성된 후 수정할 수 없다.

Immutable: An object with a fixed value.

numbers, strings and tuples

a constant hash value is needed, for key in a dictionary.



# 문자열 슬라이싱

문자열 슬라이싱

- 범위를 이용하여 문자열 일부분에 접근한다.

```
lang = "python programming"
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
p	y	t	h	o	n		p	r	o	g	r	a	m	m	i	n	g
-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



# indexing & slicing

- `lang[a]` : 인덱스 `a`의 문자
- `lang[a:b]` : 인덱스 `a`부터 `b-1`까지의 문자열 ( $a < b$ )
- `lang[a:b:c]`
  - $a < b, c > 0$  : `a`부터 `b-1`까지 `c`간격의 문자열
  - $a > b, c < 0$  : `a`부터 `b+1`까지 `c`간격의 문자열
- `lang[:]` : `lang` 문자열 전체
- `lang[::]` : `lang` 문자열 전체

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
p	y	t	h	o	n		p	r	o	g	r	a	m	m	i	n	g
-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



**str methods**

# 문자열 메소드 (method) 사용 하기

문자열 객체에서 사용할 수 있는 method

• 문자열.메소드() 형태로 사용한다.

메소드 - 클래스에서 정의된 함수

```
print(dir(str))
```





# upper() lower()

upper()

- 문자열을 모두 대문자로 바꾼다

```
name = 'steve jobs'
```

```
name.upper() # 'STEVE JOBS'.upper()
```

lower()

- 문자열을 모두 소문자로 바꾼다.

```
school = "SUNGSHIN WOMEN'S University"
```

```
school.lower()
```



# count()

count()

•부분 문자열을 센다.

```
state = 'mississippi'
```

```
state.count('s')
```

```
state.count('ssi')
```

```
state.count('s', 5)
```

```
# [5:]으로 잘라서 count한 결과
```

```
state.count('s', 1, 5)
```

```
# [1:5] 범위
```



# find()

find()

- 부분 문자열을 찾아서 첫 index를 알려준다.

```
state.find('s')
```

```
state.find('i', 5) # [5:] 범위에서 'i'를  
찾는다
```





# replace()

문자의 일부분을 다른 문자열로 대체할 때 사용

```
text = "Ice cream you scream we all scream"  
text.replace("Ice cream", "I scream")
```



```
tyoon — python3 — 57x5  
[>>> text = "Ice cream you scream we all scream"  
[>>> text.replace("Ice cream", "I scream")  
'I scream you scream we all scream'  
[>>>  
[>>>
```

# join(리스트) split()

join(리스트)

- 문자열.join(리스트) - 리스트에 있는 자료들을 문자열로 연결한다.

split()

- 문자열을 space를 기준으로 잘라서 리스트에 저장한다.



```
friends = ['alice', 'bob', 'cindy']
```

```
dash = '-'
```

```
dash.join(friends) # '-'.join(friends)
```

```
texts = 'alice bob cindy david'
```

```
texts.split(' ')
```

```
date = '2020/09/20'
```

```
date.split('/')  
  

```



# split()

공백(space), \t, \n, \r

split(): 문자열 양쪽에 불필요한 문자를 떼어 내어 리스트(list)를 만들

split('기준문자열'): 기준문자열을 기준으로 하여 문자열을 분리하여 리스트를 만

```
>>> '사과 배 포도 오렌지 감'.split()  
['사과', '배', '포도', '오렌지', '감']
```

```
>>> '사과, 배, 포도, 오렌지, 감'.split()  
['사과,', '배,', '포도,', '오렌지,', '감']
```

```
>>> '사과, 배, 포도, 오렌지, 감'.split(',')  
['사과', ' 배', ' 포도', ' 오렌지', ' 감']
```



# join()

‘구분자문자열’.join(리스트): 구분자 문자열과 리스트의 요소를 연결하여 문자열

```
>>> ' '.join(['사과', '배', '포도', '오렌지', '감'])  
'사과 배 포도 오렌지 감'
```

```
>>> ', '.join(['사과', '배', '포도', '오렌지', '감'])  
'사과,배,포도,오렌지,감'
```

```
>>> ', '.join(['사과', '배', '포도', '오렌지', '감'])  
'사과, 배, 포도, 오렌지, 감'
```



# Divide and Conquer (분할 정복 전략)



Battle of Austerlitz

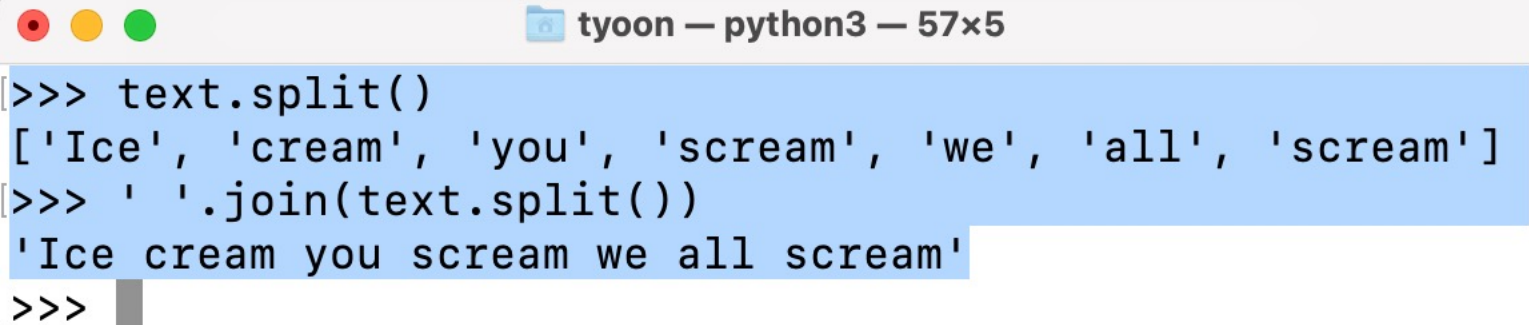
1805년 겨울에 프랑스의 나폴레옹의 군대는 아우스터리츠(Austerlitz, 현재는 체코의 남동쪽에 위치한 지역)라는 곳에서 오스트리아와 러시아 연합군과 대치하고 전투가 임박한 상태였습니다. 당시에 나폴레옹은 자신의 병력이 연합군보다 약 2만 명이 적은 상황에서, 연합군이 나폴레옹 군대의 옆쪽으로 공격하도록 유도한 후에, 연합군의 중앙을 공격하였습니다.

두개의 무리로 나뉘어진 연합군은 당황하여 도망가느라 바빴고 나폴레옹의 군대는 각각의 남은 무리들을 공격하여서 대승을 거두었습니다. 약 3주 후에 오스트리아는 나폴레옹과 조약을 맺고 프랑스에게 많은 땅을 내줌과 동시에 거액의 전쟁 보상금 또한 지불하였다고 합니다.



# split() & join() 전략

```
>>> text
'Ice cream you scream we all scream'
>>> text.split()
['Ice', 'cream', 'you', 'scream', 'we', 'all', 'scream']
>>> ' '.join(text.split())
'Ice cream you scream we all scream'
```

A screenshot of a terminal window titled 'tyoon — python3 — 57x5'. The window has a light blue background and shows the same Python code as the previous block. The text is highlighted in a darker blue.

```
>>> text.split()
['Ice', 'cream', 'you', 'scream', 'we', 'all', 'scream']
>>> ' '.join(text.split())
'Ice cream you scream we all scream'
>>>
```

# THANK YOU

