파이썬 입문

한국폴리텍대학

2023.03.17

1. 100부터 1000까지 더하기

100 + 101 + 102 + 103 + ... + 999 + 1000

while, for, if

list, set, tuple, dict

2. 면적 구하기. x = 2~10, y = 2x

3. 면적 구하기. x = 2~10, y = x**2, delta = 1

$$\int_a^b f(x)dx = \lim_{\Delta x \to 0} \sum_{x=a}^{b} f(x)\Delta x$$

4. 기울기 구하기. x = 2~3, y = 2x

sin, cos, tan

기울기 = y/x = tan

tan0 = 0, tan45 = 1, tan90 = ∞

모듈

: 함수, 변수, 클래스를 모아 놓은 파이썬 파일

사용법

```
import math
from math import *
```

만들기

1. 모듈 파일 만들기 mod1.py

2. 모듈 내용 만들기

```
#mod1.py
pp = 900

def add(x1, x2):
    return x1 + x2

def sub(x1, x2):
    return x1 - x2
```

사용하기

```
#test.py

import mod1
#import mod1 as md
#from mod1 import *
#from mod1 import add, sub

print(mod1.add(100, 200))  # 300
#print(md.add(100, 200))  # 300
#print(add(100, 200))  # 300
#print(sub(100, 200))  # -100
#print(pp)  # 900
```

람다는 익명함수(이름이 없음, anonymous function)

```
s1 = lambda x: x**2
s2 = lambda x, y: x + y
s3 = lambda : “Hello world”
s4 = lambda s: s[::-1].upper()
```

```
print(s1(10))
print(s2(10, 5))
print(s3())
print(s4(“school”))    # CH
```

```
# s4 = lambda s: s[::-1][::-1][1:3].upper()
```

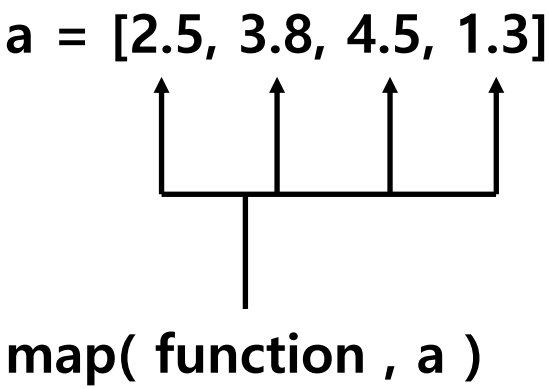
```
def func_name (매개변수)
      식

Lambda 매개변수: 식
```

람다(Lambda)

```
def lm_func(x, fn):
    return fn(x)


print(lm_func(5, lambda x: x**2))  # 25
print(lm_func('School', lambda x: x[2:] * 2))    # 'hoolhool'


def fn(x):
    return x[2:] * 2

print(lm_func('School', fn)    # 'hoolhool'
```

map()

```
a = [2.5, 3.8, 4.5, 1.3]

for i in range(len(a)):
    a[i] = int(a[i])

print(a)

print(list(map(int, a)))
print(set(map(int, a)))
print(tuple(map(int, a)))

def func(x):
    return int(x)

print(list(map(func, a)))
```



enumerate()

```
a = [2.5, 3.8, 4.5, 1.3]

for i, s in enumerate(a):
    a[i] = int(s)

print(a)


for i in enumerate(a):
    print(i, type(i))
    a[i[0]] = int(i[1])    # (0, 2.5)

print(a)
```

클래스 (class)

클래스

1. 클래스는 속성과 메소드로 구성

- 속성(attribute): 변수 — person.age
- 메소드(method): 함수 — person.study()

2. 4가지 특성 (OOP)

- 추상화 (Abstraction)
  : 사용자에게 불필요한 정보를 숨겨서 복잡한 내용을 단순하게 다루는 과정
- 캡슐화 (Encapsulation)
- 상속 (Inheritance)
- 다형성 (Polymorphism)

순차적 프로그래밍
(Sequential Programming)

객체지향 프로그래밍
(Object Oriented Programming, OOP)

클래스(class) : 설계도
객체(object) : 설계도로 만들 것, 만든 것
인스턴스(instance): 설계도로 실제 만든 것
"A객체는 B클래스의 인스턴스"

클래스 (class)

```
# 클래스 만들기, 사용하기

class Person:
        def __init__(self, name, age):
       #def __init__(self, name, age = 20):
                self.name = name
                self.age = age
                self.job = “student”

        def add(self, num):
                self.age += num
                return self.age

        def __del__(self):
                print(“destructor”)

p1 = Person("John",36)
print(p1.name, p1.age, p1.job)
print(p1.add(30))
```

생성자(Constructor) 함수 __init__()

self는 인스턴스(instance)
자기 자신을 참조하는 매개변수

클래스의 함수는 메소드(method)
person.method()
--> 클래스명(또는 인스턴스명).함수명()

소멸자(Destructor) 함수 __del__()

클래스 (class)

```
# 속성(Attribute)


class avatar:
        def __init__(self, b, c = 'test'):
                self.a = 100            # static (정적 정의)
                self.b = b              # dynamic (동적 정의)
                self.c = c              # default (기본값 정의)
```

클래스 (class)

속성, 메소드

비공개 속성: __속성 ← private 변수, 함수
(관행)언더바 1개 사용

```
class avatar:
        __special_item = 2

        def check_item(self):
                print(self.__special_item)

s = avatar()
s.check_item()        # 2

print(s.__special_item)     # error
print(avatar.__special_item)     # error
print(s._avatar__special_item)   # 2
```

print(dir(s)) →

1. 캡슐화 (encapsulation)

- 외부 접근 금지
  클래스 내부에서만 접근가능

- 이름 충돌(name conflict),
  오버라이딩 금지

- 완벽한 캡슐화는 안됨

name mangling (다른 이름으로 바꾸기)

언더바 2개 __ 를 변수나 함수 앞에 붙이면
이름이 "_클래스명__변수, 함수" 로 변경됨

클래스 (class)

```
# self   < - - - - - 관행적으로 self사용

class Person:
        def add1(a, b):
                return a - b

        def add2(self, a, b):
                return a + b


p1 = Person()

print(p1.add2(100, 50))  # 150
#print(p1.add1(100, 50))  # error

print(Person.add1(100, 50))  # 50
#print(Person.add2(100, 50))  # error
#print(Person.add2(p1, 100, 50))  # 150
```

```
# 변수(Variables) – 클래스 변수, 인스턴스 변수

class Person:
        s = 100
        def add1(a, b):
                return a - b

p1 = Person()
p2 = Person()

#Person.s = 200
#p1.s = 200
print(Person.s, p1.s, p2.s)  # 100, 100, 100
```

클래스 (class)

```
# __call__

class Person:
        def __init__(self):
                print("start")

        def __call__(self):
                print("call")


s = Person()

s()                 # start
                    # call
```

Instance를 함수처럼 호출 가능

```
# __call__()이 없다면

  TypeError: 'person' object is not callable
```

```
# 상속 (Inheritance)

class Person:
        def add1(self, a, b):
                return a - b

class Student(Person):
        pass

p1 = Person()
p2 = Student()

print(p1.add1(10, 5), p2.add1(10, 5))  # 100, 100
```

2. 상속 (Inheritance)

- 반복 코드를 줄일 수 있다

기초 클래스(base class)
= 부모 클래스(parent class)
= 상위 클래스(super class)

파생 클래스(derived class)
= 자식 클래스(child class)
= 하위 클래스(sub class)

클래스 (class)

```
# 다중 상속 (multi-Inheritance)

class tiger:
        def cry(self):
                print("tiger cry")
        def eat(self):
                print("tiger eat")

class lion:
        def cry(self):
                print("lion cry")
        def move(self):
                print("lion move")

Class liger(tiger, lion):
        pass
```

```
liger1 = liger()

liger1.cry()        # tiger cry   : 먼저 상속받은
                    #               부모 클래스
liger1.eat()        # tiger eat
liger1.move()       # lion move
```

메소드 호출에서 혼동이 발생할 수 있으니 가급적 사용 자제

```
# 메소드 오버라이딩 (overriding, 덮어쓰기)

class Person:
        def add1(self, a, b):
                return a - b

class Student(Person):
        def add1(self, a, b):
                return a * b

p1 = Person()
p2 = Student()

print(p1.add1(10, 5), p2.add1(10, 5))
```

3. 다형성 (Polymorphism)

- 같은 형태의 코드가 서로 다른 동작을 하는 것

```
# 추상 클래스 (abstract class)

from abc import *      < - - - abstract base class

class 추상클래스이름(metaclass=ABCMeta):
        @abstractmethod
        def 메소드이름(self):
            코드
```

4. 추상화 (Abstraction)

- 사용자에게 불필요한 정보를 숨겨서 복잡성을 낮추고, 효율을 높이는 과정

추상 클래스

- 메소드의 목록만 가진 클래스. 상속받는 클래스에서 메소드 구현을 강제하기 위해 사용

클래스 (class)

```
from abc import *      < - - - abstract base class

class studentBase(metaclass=ABCMeta):
        @abstractmethod
        def study(self):
            pass

        @abstractmethod
        def go_to_school(self):
            pass

class student(studentBase):
        def study(self):
            print("study")

kim = student()
kim.study()
```

```
def go_to_school(self):
    print("go")
```

파생클래스에서 구현하지 않으면 에러

4. 추상화 (Abstraction)

- 사용자에게 불필요한 정보를 숨겨서 복잡성을 낮추고, 효율을 높이는 과정

추상 클래스

- 메소드의 목록만 가진 클래스. 상속받는 클래스에서 메소드 구현을 강제하기 위해 사용

- 파생클래스에서 반드시 구현해야함