웹프로그래밍의 기초

Week6

fuction; class

function

function

- A function is **a block of code** which only runs when it is called.
- You can pass data, known as **parameters**, into a function.
  - Parameters or Arguments?
    - The terms parameter and argument can be used for the same thing: information that are passed into a function.
  - From a function's perspective:
    - A parameter is the variable listed inside the parentheses in the function definition.
    - An argument is the value that is sent to the function when it is called.
- A function can **return** data as a result.

General

- **Calling** a function
  - To call a function, use the function name followed by parenthesis:

```
def my_function():
    """ This is doc string, and the function name must have
    only lower case characters and underscore character for its readability."""
    print("Hello from a function")

my_function()
----------------------------------------
Hello from a function
```

Positional Arguments

- An argument is a variable, value or object passed to a function or method as input. Positional arguments are arguments that need to be included in the proper position or order. The first positional argument always needs to be listed first when the function is called. The second positional argument needs to be listed second and the third positional argument listed third, etc.

```
def describe_pet(animal_type, pet_name):
    """Display pet information"""
    print(f"\nI have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name.title()}")

describe_pet('hamster', 'harry')
describe_pet('harry', 'hamster')
```

```
----------------------------------------
I have a hamster
My hamster's name is Harry

I have a harry
My harry's name is Hamster
```

Keyword Arguments

- You can also send arguments with the key = value syntax. This way, the order of the arguments does not matter.

```
def describe_pet(animal_type, pet_name):
    """Display pet information"""
    print(f"\nI have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name.title()}")

describe_pet(animal_type = 'cat', pet_name = 'ali')
describe_pet(pet_name = 'ali', animal_type = 'cat')
```

```
----------------------------------------
I have a cat
My cat's name is Ali

I have a cat
My cat's name is Ali
```

default Arguments

- Python can also set the default value for the function, that is, when defining the function, the formal parameter will be assigned. After that, if the user does not specify a new argument value, the function will always default to the default value.

```
def describe_pet(pet_name, animal_type='dog'):
    """Display pet information"""
    print(f"\nI have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name.title()}")

describe_pet(pet_name = 'ali', animal_type = 'cat')
describe_pet(animal_type = 'cat', pet_name = 'ali')
describe_pet(pet_name = 'ali')
describe_pet()

----------------------------------------
I have a cat
My cat's name is Ali

I have a cat
My cat's name is Ali

I have a dog
My dog's name is Ali
Traceback (most recent call last):
  File "<string>", line 9, in <module>
TypeError: describe_pet() missing 1 required positional argument:
'pet_name'
```

Return values

- To let a function return a value, use the return statement.

```
def get_formatted_name():
    print("Enter 'q' at any time to quit.")
    while True:
            first = input("\nPlease give me a first name: ")
            if first == 'q':
                        break
            last = input("Please give me a last name: ")
            if last == 'q':
                        break
            full_name = first + ' ' + last
            return full_name.title()

formatted_name = get_formatted_name()
print("\tNeatly formatted name: " + formatted_name + '.’)
```

```
-----------------------------------------
Enter 'q' at any time to quit.
Please give me a first name: tony
Please give me a last name: stark
Neatly formatted name: Tony Stark.
```

Module

- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module.
- A module is a file containing Python definitions and statements. The file name is the module name with **.py appended**. Within a module, the module's name (as a string) is available as the value of the global variable __name__.

Calling your module, example 1

```
fibo.py

def fib(n):
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

```
import fibo

print(fibo.fib(100))
print(fibo.__name__)
----------------------------------------

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
'fibo'
```

Calling your module, example 2

```
pizza.py

def make_pizza(size, *toppings):
    """ Overview of the pizza to be made """
    print("\nMaking a "+str(size)+
          "-inch pizza with the following toppings:")
    for topping in toppings:
        print("- "+topping)
```

```
import pizza
pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
-----------------------------------------
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni
Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

when module is called in your program

- Use as Assign an alias to a module

  import module_name as mn

```
import pizza as p
p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

- Import specific functions

  from module_name import function_name

```
from pizza import make_pizza
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

- Use as Give the function an alias

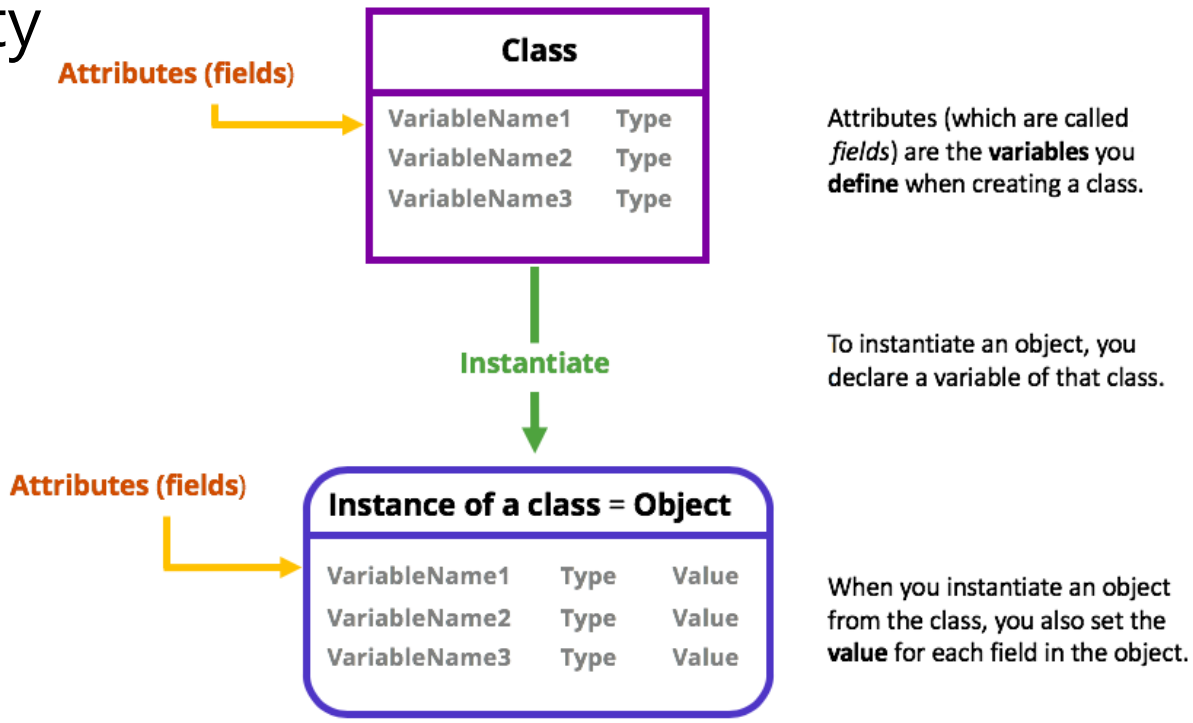
  from module_name import function_name as fn

```
from pizza import make_pizza as mp
mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

class

generals

- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new type of object, allowing new instances of that type to be made.
- Each class instance can have attributes attached to it for maintaining its state.
- Class instances can also have methods (defined by its class) for modifying its state.



Methods & attributes

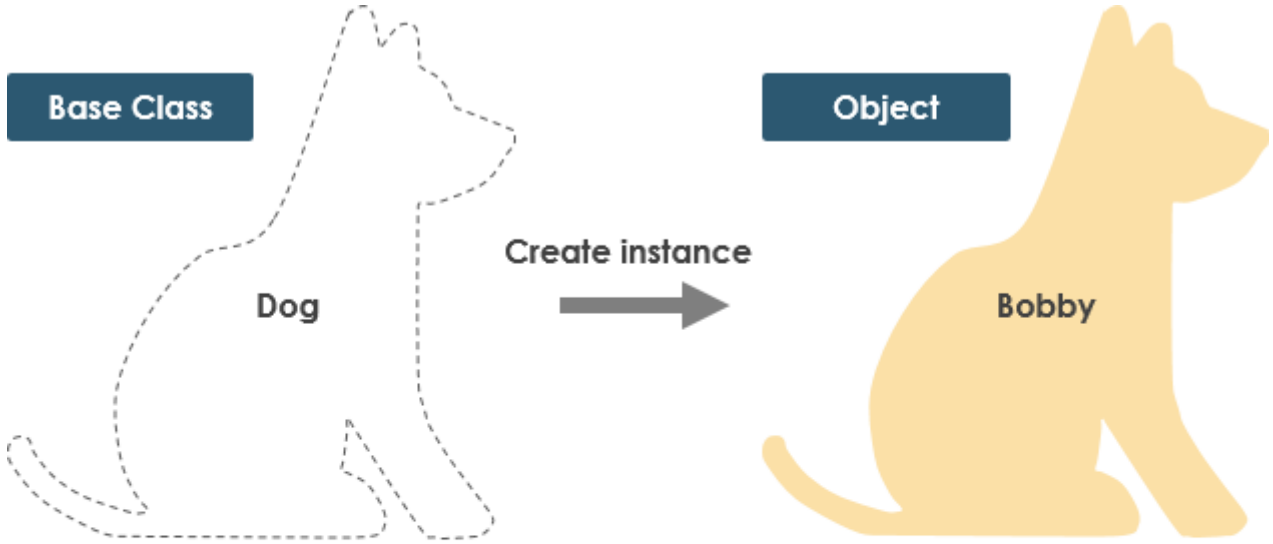
- Once an instance is created, it has two kinds of valid copoment names: attributes and methods.
  - attributes correspond to “instance variables”, and can be accessed via its name in the form of <object_name>.<속성이름>
    - *my_car.make*
  - A method is a function that “belongs to” an object, and can be called via its name in the form of <object_name>.<메소드이름>
    - *my_car. get_descriptive_name()*

```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = maket
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
```

Data Attributes (속성)

Methods (메소드)

Instantiation: a class into the specific object



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

Instantiation: Dog to my/your dogs

```
class Dog:
    """A simple attempt to model a dog."""

    def __init__(self, name, age):
        """Initialize name and age attributes."""
        self.name = name
        self.age = age

    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        """Simulate rolling over in response to a command."""
        print(f"{self.name} rolled over!")
```

```
my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

```
--------------------------------------------------
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Changing 속성 in Object: Direct

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")
```

```
my_new_car = Car('BMW', 'M340i', 2022)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 10000
print(my_new_car.read_odometer())
```

```
2022 Bmw M340I
This car has 10000 miles on it.
```

Changing 속성 in Object: Method for update

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
```

```
my_new_car = Car('BMW', 'M340i', 2022)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 10000
my_new_car.update_odometer(23333)
print(my_new_car.read_odometer())
-------------------------------------------------

2022 Bmw M340I
This car has 23333 miles on it.
```

Changing 속성 in Object: Method for a unit increment

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles
```

```
my_new_car = Car('BMW', 'M340i', 2022)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 10000
my_new_car.increment_odometer(100)
print(my_new_car.read_odometer())
```

```
2022 Bmw M340I
This car has 10100 miles on it.
```

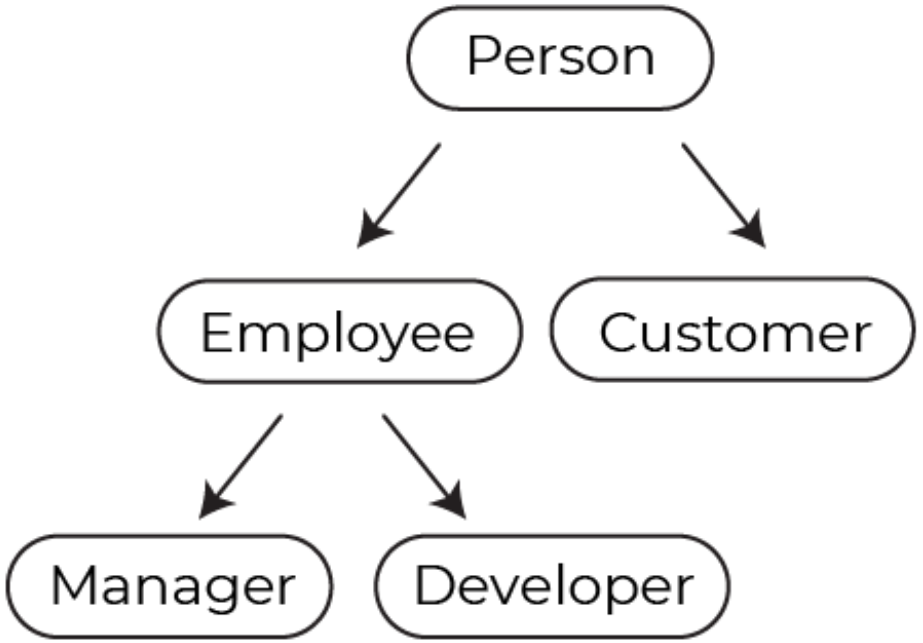
Inheritance (상속)

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
  - Parent class is the class being inherited from, also called base class.
  - Child class is the class that inherits from another class, also called derived class.

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  pass

x = Student("Steve", "Jobs")
x.printname()
```

```
-----------------------------------
```

```
Steve Jobs
```



Method override bt parent-child classes

```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def fill_gas_tank(self):
        pass

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery_size = 75

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def fill_gas_tank(self):
        """obsolte method for EV."""
        print(f"This car doesn't need a gas tank.")
```

```
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.fill_gas_tank()
```

```
----------------------------------------------------

2019 Tesla Model S
This car doesn't need a gas tank.
```

import

car.py

```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles
```

my_car.py

```
Import car import Car

my_new_car = Car('subaru', 'outback', 2015)
print(my_new_car.get_descriptive_name())

my_new_car.increment_odometer(100)
my_new_car.read_odometer()
```

```
2015 Subaru Outback
This car has 100 miles on it.
```

other ways to import class(es) from another file

- import entire module

```
import <module_name>
ie.
    import car
```

- import ONE class

```
from <module_name> import <class_name>
ie.
    from car import Car
```

- import multiple classes

```
from <module_name> import <class_name1>, <class_name1>, ...
ie.
    from car import Car, ElectricCar
```