Python DTS PRoA 2022

Python Object-Oriented Programming

Muhammad Ogin Hasanuddin

KK Teknik Komputer
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

In this course, you will learn about:

- Create objects in Python by defining classes and methods
- Extend classes using inheritance
- Principles in object-oriented programming

Introduction to Python Object-oriented Programming

- Everything in Python is an object.
- An object has a state and behaviors.
- To create an object, you define a class first. And then, from the class, you can create one or more objects.
- The objects are instances of a class.

Define a class

- To define a class, you use the class keyword followed by the class name. For example, the following defines a Person class:

```
class Person:
    pass
```

- To create an object from the Person class, you use the class name followed by parentheses (), like calling a function:

```
person = Person()
```

- In this example, the person is an instance of the Person class. Classes are callable.

Define instance attributes

- Python is dynamic. It means that you can add an attribute to an instance of a class dynamically at runtime.
- For example, the following adds the name attribute to the person object:

```
person.name = 'John'
```

- However, if you create another Person object, the new object won't have the name attribute.
- To define and initialize an attribute for all instances of a class, you use the `__init__` method. The following defines the Person class with two instance attributes name and age:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- When you create a Person object, Python automatically calls the __init__ method to initialize the instance attributes. In the __init__ method, the self is the instance of the Person class.

Define instance attributes

- The following creates a Person object named person:

```
person = Person('John', 25)
```

- The person object now has the name and age attributes. To access an instance attribute, you use the dot notation. For example, the following returns the value of the name attribute of the person object:

```
person.name
```

Define instance methods

- The following adds an instance method called greet() to the Person class:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hi, it's {self.name}."
```

- To call an instance method, you also use the dot notation. For example:

```python
person = Person('John', 25)
print(person.greet())
```

- Output:

```
Hi, it's John
```

Define class attributes

- Unlike instance attributes, class attributes are shared by all instances of the class. They are helpful if you want to define class constants or variables that keep track of the number of instances of a class.
- For example, the following defines the counter class attribute in the Person class:

```python
class Person:
    counter = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hi, it's {self.name}."
```

- You can access the counter attribute from the Person class:

```python
Person.counter
```

Define class attributes

- Or from any instances of the Person class:

```
person = Person('John',25)
person.counter
```

- To make the counter variable more useful, you can increase its value by one once an object is created. To do it, you increase the counter class attribute in the __init__ method:

```
class Person:
    counter = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.counter += 1

    def greet(self):
        return f"Hi, it's {self.name}."

p1 = Person('John', 25)
p2 = Person('Jane', 22)
print(Person.counter)
```

Define class method

- Like a class attribute, a class method is shared by all instances of the class. The first argument of a class method is the class itself. By convention, its name is cls. Python automatically passes this argument to the class method. Also, you use the @classmethod decorator to decorate a class method.
- The following example defines a class method that returns an anonymous Person object:

```
class Person:
    counter = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.counter += 1

    def greet(self):
        return f"Hi, it's {self.name}."

    @classmethod
    def create_anonymous(cls):
        return Person('Anonymous', 22)
```

Define static method

- A static method is not bound to a class or any instances of the class. In Python, you use static methods to group logically related functions in a class. To define a static method, you use the @staticmethod decorator.
- For example, the following defines a class TemperatureConverter that has two static methods that convert from celsius to Fahrenheit and vice versa:

```
class TemperatureConverter:
    @staticmethod
    def celsius_to_fahrenheit(c):
        return 9 * c / 5 + 32

    @staticmethod
    def fahrenheit_to_celsius(f):
        return 5 * (f - 32) / 9
```

Define static method

- To call a static method, you use the ClassName.static_method_name() syntax. For example:

```
f = TemperatureConverter.celsius_to_fahrenheit(30)
print(f)  # 86
```

- Notice that Python doesn’t implicitly pass an instance (self) as well as class (cls) as the first argument of a static method.

Single inheritance

- A class can reuse another class by inheriting it. When a child class inherits from a parent class, the child class can access the attributes and methods of the parent class.
- For example, you can define an Employee class that inherits from the Person class:

```
class Employee(Person):
    def __init__(self, name, age, job_title):
        super().__init__(name, age)
        self.job_title = job_title
```

- Inside the __init__ method of the Employee class calls the __init__method of the Person class to initialize the name and age attributes. The super() allows a child class to access a method of the parent class.
- The Employee class extends the Person class by adding one more attribute called job_title.

Single inheritance

- The Person is the parent class while the Employee is a child class. To override the greet() method in the Person class, you can define the greet() method in the Employee class as follows:

```
class Employee(Person):
    def __init__(self, name, age, job_title):
        super().__init__(name, age)
        self.job_title = job_title

    def greet(self):
        return super().greet() + f" I'm a 
{self.job_title}."
```

- The greet() method in the Employee is also called the greet() method of the Person class. In other words, it delegates to a method of the parent class.

Terimakasih!