PHOT 110: Introduction to programming

LECTURE 11: Object Oriented Programming: Classes (Ch. 7 & 9)

Michaël Barbier, Spring semester (2023-2024)

CLASSES

WHAT IS A CLASS ?

- Part of object-oriented programming
- A class combines
  - **Attributes** (parameters of the class), and
  - **Methods** (functions of the class)
- An **object** is an **instance** of a **class**

```
1 text = "This is an object of class string"
2 print(type(text))
```

```
<class 'str'>
```

WHAT IS A CLASS ?

- Part of object-oriented programming
- A class combines
  - **Attributes** (parameters of the class), and
  - **Methods** (functions of the class)
- An **object** is an **instance** of a **class**

```
1 a_list = ["Green", "Yellow", "Orange"]
2 print(a_list.__doc__)
```

```
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty
list.
The argument must be an iterable if specified.
```

CLASS DEFINITION

- **Attributes** (parameters of the class), and
- **Methods** (functions of the class)

```
1  class SimpleClass:
2
3      # An attribute
4      secret = "4567"
5
6      # A method
7      def tell_the_secret(self):
8          return "The code = " + self.secret
```

CLASSES & OBJECTS

An **object** is an **instance** of a **class**

```
1  class SimpleClass:
2
3      secret = "4567"
4
5      def tell_the_secret(self):
6          return "The code = " + self.secret
```

```
1  c = SimpleClass()
2  print(c.secret)
3  print(c.tell_the_secret())
```

```
4567
The code = 4567
```

CONSTRUCTORS

```
1  class SimpleClass:
2
3      def __init__(self, code):
4          self.secret = str(code)
5
6      def tell_the_secret(self):
7          return "The code = " + self.secret
```

A constructor creates an **instance** of a **class** with **parameters**

```
1  c = SimpleClass(123456)
2  print(c.tell_the_secret())
```

```
The code = 123456
```

A SLIGHTLY MORE COMPLEX EXAMPLE CLASS

```
1  class spaceship:
2
3    def __init__(self, pos_0, orient_0, image):
4      self.position = pos_0
5      self.orientation = orientation_0
6      self.image = image
7
8    def teleport(self, displacement):
9      self.position = self.position + displacement
10
11   def test_collision(pos, R):
12     if (pos - self.position)**2  < R ** 2:
13       return True
14     return False
```

EXTRA ATTRIBUTES

```
1 c = SimpleClass(512)
2 print(c.tell_the_secret())
3
4 # Define attributes after creation
5 c.color = "Green"
6
7 # Test the attribute is there
8 print(c.color)
```

```
The code = 512
Green
```

OBJECTS OF A CLASS

Adding attributes per object

```
1  c1 = SimpleClass(123)
2  c1.color = "Yellow"
3
4  c2 = SimpleClass(456)
5  c2.name = "Mehmet"
```

```
1  print("c1 says: " + c1.tell_the_secret())
```

```
c1 says: The code = 123
```

```
1  print("c2 says: " + c2.tell_the_secret())
```

```
c2 says: The code = 456
```

OBJECTS OF A CLASS

Adding attributes per object

```
1  c1 = SimpleClass(123)
2  c1.color = "Yellow"
3
4  c2 = SimpleClass(456)
5  c2.name = "Mehmet"
```

```
1  print(c1.color)
```

```
Yellow
```

```
1  print(c2.color)
```

```
AttributeError: 'SimpleClass' object has no attribute
'color'
```

METHODS VERSUS FUNCTIONS

- Methods act on objects: You need an object!

```
1 a_list = [1, 2, 3]
2 a_list.reverse()
3 print(a_list)
```

```
[3, 2, 1]
```

- functions are not bound to an object

```
1 a_list = [1, 2, 3]
2 print(list(reversed(a_list)))
```

```
[3, 2, 1]
```

SPECIAL METHODS

SPECIAL METHODS OF A CLASS

Special methods use double underscores in their name:

- Constructor: `__init__()`
- Callable: `__call__()`
- Printing a class instance: `__str__()`

Operator overloading: Using operators `+`, `-`, `*` between objects

- Not equal sign: `__ne__()`
- Plus operator: `__add__()`

CALLABLES

- Especially for classes defining a formula
- Call a class instance like a function

```
1  class Formula:
2
3      def __init__(self, a):
4          self.a = a
5
6      def __call__(self, a, x):
7          return self.a * x
```

CALLABLES

```
1  class Formula:
2
3      def __init__(self, a):
4          self.a = a
5
6      def __call__(self, a, x):
7          return self.a * x
```

Formula is a class:

```
1  f = Formula(6)
2  print(f)
```

```
<__main__.Formula object at 0x0000023543DEEF30>
```

CALLABLES

```
1  class Formula:
2
3      def __init__(self, a):
4          self.a = a
5
6      def __call__(self, a, x):
7          return self.a * x
```

Formula can be called like a function:

```
1  f_called = f(6, 4)
2  print(f_called)
```

```
24
```

TELLING HOW TO PRINT AN INSTANCE: STR()

```
1  class Formula:
2
3      def __init__(self, a):
4          self.a = a
5
6      def __str__(self):
7          return f"My special format: {self.a} + 5"
```

```
1  f = Formula(6)
2  print(f)
```

```
My special format: 6 + 5
```

OPERATOR OVERLOADING

Using for example “+” between custom class objects

```
1  class Figure:
2
3      def __init__(self, a):
4          self.a = a
5
6      def __add__(self, a, x):
7          return self.a * x
```

CLASS HIERARCHY

EXAMPLE CLASS

```
1  class Vehicle:
2
3      location = None
4      city_list = ["Izmir", "Istanbul", "Bursa"]
5
6      def __init__(self, location):
7          self.location = str(location)
8
9      def drive_to_city(self, city):
10         self.location = city
11
12     def tell_current_location(self, city):
13         return self.location
```

IF YOU NEED SOMETHING SIMILAR BUT NOT EXACTLY THE SAME

Cargo delivery service: We want to transport goods

- A list of stock of goods in each city ? as an **attribute** ?
- A method `deliver()` ?

Bus transport of persons

- time tables
- number of seats
- ticket price per person

IF YOU NEED SOMETHING SIMILAR BUT NOT EXACTLY THE SAME

Do we need to write code for a whole new class ?

- How to add functionality to the vehicle class ?
- Change methods ?

We can derive a class from another class

- Keeping the old functionality
- Extending with new methods/attributes
- Adapting methods

FAMILY OF CLASSES

```
1 class DerivedClass(BaseClass):
2     <statement>
3     ...
4     <statement>
```

EXAMPLE CLASS

```
 1 class Vehicle:
 2 
 3     city_list = ["Izmir", "Istanbul", "Bursa"]
 4 
 5     def __init__(self, location):
 6         self.city_list = ["Izmir", "Istanbul", "Bursa"]
 7         self.location = location
 8 
 9     def drive_to_city(self, city):
10         print(f"Driving from {self.location} to {city}")
11         self.location = city
```

```
1 car = Vehicle("Izmir")
2 car.drive_to_city("Istanbul")
```

```
Driving from Izmir to Istanbul
```

EXAMPLE CLASS

```
 1 class Vehicle:
 2 
 3     city_list = ["Izmir", "Istanbul", "Bursa"]
 4 
 5     def __init__(self, location):
 6         self.city_list = ["Izmir", "Istanbul", "Bursa"]
 7         self.location = location
 8 
 9     def drive_to_city(self, city):
10         print(f"Driving from {self.location} to {city}")
11         self.location = city
```

```
1 bus = Vehicle("Izmir")
2 bus.drive_to_city("Istanbul")
```

```
Driving from Izmir to Istanbul
```

DERIVED CLASSES

Make a derived class (subclass)

- Use `super()` to access BaseClass
- Use `super().__init__()` to have BaseClass constructor

```
1  class DerivedClass(BaseClass):
2
3      <attribute>
4      <attribute>
5
6      <method>
7      <method>
8      ...
```

DERIVED CLASSES

```
class Vehicle:

    def __init__(self, location):
        self.city_list = ["Izmir", "Istanbul", "Bursa"]
        self.location = location

    def drive_to_city(self, city):
        print(f"Driving from {self.location} to {city}")
        self.location = city
```

```
class Bus(Vehicle):

    def __init__(self, location, passengers):
        super().__init__(location)
        self.max_passengers = 4
        self.passengers = passengers
```

DERIVED CLASSES

```
1  class Bus(Vehicle):
2
3      def __init__(self, location, passengers):
4          super().__init__(location)
5          self.max_passengers = 4
6          self.passengers = passengers
```

```
1  bus = Bus("Izmir", ["Ahmet", "Zeynep"])
2  bus.drive_to_city("Istanbul")
```

```
Driving from Izmir to Istanbul
```

DERIVED CLASSES, USE SUPERCLASS METHOD

```
1  class Bus(Vehicle):
2
3      def __init__(self, location, passengers):
4          super().__init__(location)
5          self.max_passengers = 4
6          self.passengers = passengers
7
8      def accept_passenger(self, name):
9          if len(self.passengers) < self.max_passengers:
10             self.passengers.append(name)
11         else:
12             print("Sorry, the bus is full !")
```

```
1  bus = Bus("Izmir", ["Ahmet", "Zeynep"])
2  bus.drive_to_city("Istanbul")
```

```
Driving from Izmir to Istanbul
```

DERIVED CLASSES, USE SUBCLASS METHOD

```
class Bus(Vehicle):

    def __init__(self, location, passengers):
        super().__init__(location)
        self.max_passengers = 4
        self.passengers = passengers

    def accept_passenger(self, name):
        if len(self.passengers) < self.max_passengers:
            self.passengers.append(name)
        else:
            print("Sorry, the bus is full !")
```

```
bus = Bus("Izmir", ["Ahmet", "Zeynep"])
bus.accept_passenger("Kemal"); print(bus.passengers)
```

```
['Ahmet', 'Zeynep', 'Kemal']
```

DERIVED CLASSES, USE SUBCLASS METHOD

```
1 bus = Bus("Izmir", ["Ahmet", "Zeynep"])
2 bus.accept_passenger("Mehmet")
3 print(bus.passengers)
```

```
['Ahmet', 'Zeynep', 'Mehmet']
```

```
1 bus.accept_passenger("Rani")
2 print(bus.passengers)
```

```
['Ahmet', 'Zeynep', 'Mehmet', 'Rani']
```

```
1 bus.accept_passenger("Bob")
2 print(bus.passengers)
```

```
Sorry, the bus is full !
['Ahmet', 'Zeynep', 'Mehmet', 'Rani']
```

CLASSES

- For a more comprehensive overview of the possibilities of classes, see e.g. the video of the OpenCourseWare class: CS50 course on Object Oriented Programming

Lecture 11: Object Oriented Programming: Classes (Ch. 7 & 9)