LAiMM

ModulE 08: More on class & OBJECTS(2)

Instructor: Jonny C.H Yu

Review

Food for Thought: Managing Complexity

- **Programming is about managing complexity.**
- **There are two powerful mechanisms, to accomplish this: decomposition and abstraction.**
- **Decomposition creates structure.**
  - It allows us to break a program into parts that are self-contained.
  - **Functions** are the major facilitator.
- **Abstraction hides details.**
  - It allows us to use a piece of code as if it were a black box. For example, **lists**.
  - **Abstract data types** are the major facilitator.
  - **Class** allows us to create abstract data types of our own.

Review

Important Class Mechanisms

- **Class definitions start with the class keyword followed by the name of class and a colon.**
- **A class often (~99%) consists of instance attributes and instance methods.**
- **Instance attributes**
  - Instance attributes: attributes that vary from one object to another.
  - You access these instance attributes inside the class using **self**.
- **Instance methods**
  - The first parameter of instance methods is always **self**.
  - Python has a few magic methods for instance methods (__init__, __str__, __repr__, __add__, __eq__, …)
- **A class sometimes (~5%) consists of class attributes and class methods.**

Review

(Exercise) Who is Who?

- **Identify the instance attributes, instance methods, class attributes and class methods in the class Dog.**

```python
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # An instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
    # Replace description() with __str__()
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Review

(Exercise) Who is Who?

- **Identify the instance attributes, instance methods, class attributes and class methods in the class Dog.**

```python
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # An instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
    # Replace description() with __str__()
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

- Instance attributes: self.name, self.age
- Instance methods: __init__, speak, __str__
- Class attribute: species
- Class method: N/A

Live exercise

Exercise1: Sorted List

Review

```
[2]: a = Sorted_list([5, 4, 3])

[3]: a

[3]: Sorted_list: [3, 4, 5]

[4]: b = Sorted_list([6, 2, 3])

[5]: b

[5]: Sorted_list: [2, 3, 6]

[6]: c = a + b
     c

[6]: Sorted_list: [2, 3, 3, 4, 5, 6]

[7]: d = a + b + c
     d

[7]: Sorted_list: [2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6, 6]

[8]: print(c)

     [2, 3, 3, 4, 5, 6]
```

- **Define a Sorted_list class that implements some behavior of sorting lists as specified by the client codes.**

Exercise1: Sorted List (Ans)

Review

[1]:
```
class Sorted_list:
    def __init__(self, lst):
        self.slst = sorted(lst)
    def __repr__(self):
        return f'Sorted_list: {self.slst}'
    def __str__(self):
        return f'{self.slst}'
    def __add__(self, other):
        return Sorted_list(self.slst + other.slst)
```

[2]:
```
a = Sorted_list([5, 4, 3])
```

[3]:
```
a
```

[3]:
```
Sorted_list: [3, 4, 5]
```

[4]:
```
b = Sorted_list([6, 2, 3])
```

[5]:
```
b
```

[5]:
```
Sorted_list: [2, 3, 6]
```

Sorted_List.ipynb

Exercise1: Sorted List (Ans)

Review

[1]:
```python
class Sorted_list:
    def __init__(self, lst):
        self.slst = sorted(lst)
    def __repr__(self):
        return f'Sorted_list: {self.slst}'
    def __str__(self):
        return f'{self.slst}'
    def __add__(self, other):
        return Sorted_list(self.slst + other.slst)
```

[6]:
```python
c = a + b
c
```

[6]:
```
Sorted_list: [2, 3, 3, 4, 5, 6]
```

[7]:
```python
d = a + b + c
d
```

[7]:
```
Sorted_list: [2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6, 6]
```

[8]:
```python
print(c)
```

```
[2, 3, 3, 4, 5, 6]
```

Sorted_List.ipynb

Review

Class Creator

```python
class Sorted_list:
    def __init__(self, lst):
        self.slst = sorted(lst)
    def __repr__(self):
        return f'Sorted_list: {self.slst}'
    def __str__(self):
        return f'{self.slst}'
    def __add__(self, other):
        return Sorted_list(self.slst + other.slst)
```

Class User (Client)

```
[2]: a = Sorted_list([5, 4, 3])

[3]: a

[3]: Sorted_list: [3, 4, 5]

[4]: b = Sorted_list([6, 2, 3])

[5]: b

[5]: Sorted_list: [2, 3, 6]

[6]: c = a + b
     c

[6]: Sorted_list: [2, 3, 3, 4, 5, 6]

[7]: d = a + b + c
     d

[7]: Sorted_list: [2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6, 6]

[8]: print(c)

     [2, 3, 3, 4, 5, 6]
```

Exercise2: Point and Circle

(1) Create a class `Point` in the first cell that represents an (x, y) coordinate pair. Include `__init__` and `__repr__` methods and a `move` method. The `move` method receives a `Point` object to set a new location.

(2) Create a class `Circle` in the second cell that has its attribute `radius` and `point` (a `Point` object that represents the `Circle`'s center location). Include `__init__` and `__repr__` methods and a `move` method. The `move` method receives a `Point` object and set the center to the new location by calling Point's `move` method.

Exercise2: Point and Circle

(1) Create a class `Point` in the first cell that represents an (x, y) coordinate pair. Include `__init__` and `__repr__` methods and a `move` method. The `move` method receives a `Point` object to set a new location.

```
[3]: p = Point(2, 5)
```

```
[4]: p
```

```
[4]: Point(x=2, y=5)
```

```
[5]: p.move(Point(10,10))
```

```
[6]: p
```

```
[6]: Point(x=10, y=10)
```

```
[1]: class Point:
         """Point class for maintaining an x-y location."""
         def __init__(self, x, y):
             """Initialize a Point object."""
             self.x = x
             self.y = y
         def __repr__(self):
             return f'Point(x={self.x}, y={self.y})'

         def move(self, p):
             self.x = p.x
             self.y = p.y
```

Exercise2: Point and Circle

(2) Create a class `Circle` in the second cell that has its attribute `radius` and `point` (a `Point` object that represents the `Circle`'s center location). Include `__init__` and `__repr__` methods and a `move` method. The `move` method receives a `Point` object and set the center to the new location by calling Point's `move` method.

```
[7]: circle = Circle(p, 25)

[8]: circle

[8]: Circle(center=Point(x=10, y=10), radius=25)

[9]: circle.move(Point(-10, 20))

[10]: circle

[10]: Circle(center=Point(x=-10, y=20), radius=25)
```

Exercise2: Point and Circle

```
[7]: circle = Circle(p, 25)

[8]: circle

[8]: Circle(center=Point(x=10, y=10), radius=25)

[9]: circle.move(Point(-10, 20))

[10]: circle

[10]: Circle(center=Point(x=-10, y=20), radius=25)
```

Circle_Point.ipynb

```
[2]: class Circle:
         """Circle class for maintaining a point and radius."""

         def __init__(self, point, radius):
             """Initialize a Point object."""
             self.point = point
             self.radius = radius

         def __repr__(self):
             return f'Circle(center={self.point}, radius={self.radius})'

         def move(self, new_location):
             self.point.move(new_location)
```

Review

Class Creator

```
[1]: class Point:
         """Point class for maintaining an x-y location."""
         def __init__(self, x, y):
             """Initialize a Point object."""
             self.x = x
             self.y = y
         def __repr__(self):
             return f'Point(x={self.x}, y={self.y})'

         def move(self, p):
             self.x = p.x
             self.y = p.y
```

```
[2]: class Circle:
         """Circle class for maintaining a point and radius."""

         def __init__(self, point, radius):
             """Initialize a Point object."""
             self.point = point
             self.radius = radius

         def __repr__(self):
             return f'Circle(center={self.point}, radius={self.radius})'

         def move(self, new_location):
             self.point.move(new_location)
```

Class User (Client)

```
[3]: p = Point(2, 5)

[4]: p

[4]: Point(x=2, y=5)

[5]: p.move(Point(10,10))

[6]: p

[6]: Point(x=10, y=10)

[7]: circle = Circle(p, 25)

[8]: circle

[8]: Circle(center=Point(x=10, y=10), radius=25)

[9]: circle.move(Point(-10, 20))

[10]: circle

[10]: Circle(center=Point(x=-10, y=20), radius=25)
```

Putting Classes in Modules

- A well-defined class or set of classes provides useful abstractions that can be leveraged in many different programs.
- Modularizing classes in python follow what we have learned from functions

Storing Functions in Modules

- **Module is a file that contains Python code**
  - Contains function definition but does not contain calls to the functions
    - Importing programs will call the functions
- **Rules for module names:**
  - File name should end in `.py`
  - Cannot be the same as a Python keyword
- **Import module using `import` statement**

Example: write your own modules for functions

circle.py

```python
# The circle module has functions that perform
# calculations related to circles.
import math

# The area function accepts a circle's radius as an
# argument and returns the area of the circle.
def area(radius):
    return math.pi * radius**2

# The circumference function accepts a circle's
# radius and returns the circle's circumference.
def circumference(radius):
    return 2 * math.pi * radius
```

[1]:

```python
import circle
```

[2]:

```python
radius = float(input('Enter radius:'))
my_area = circle.area(radius)
my_circum = circle.circumference(radius)
print('The area is:', format(my_area, '.4f'))
print('The circumference is:', format(my_circum, '.4f'))
```

```
Enter radius: 10
The area is: 314.1593
The circumference is: 62.8319
```

Example: write your own modules for classes

dog.py

```python
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # Instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
    # Replace description with __str__
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

dog.py

L2_Dog_Client.ipynb

```
[1]: from dog import Dog

[2]: miles = Dog("Miles", 4)

[3]: print(miles)

Miles is 4 years old
```

- You first write your class definition in a separate .py file. In this case dog.py and you have the dog module.
- [1] You can now import the Dog class from the dog module.
- [2] – [3] You can now use the Dog class as usual.

Exercise: Number Class Using Magic Methods

- Create a foo module and implement a Number class that supports the operators specified by the client codes.

```
[1]: from foo import Number

[2]: a = Number(20)

[3]: a

[3]: Number: 20

[4]: b = Number(10)

[5]: b

[5]: Number: 10

[6]: c = Number(5)

[7]: print(a + b)

30

[8]: print(a + b + c)

35
```

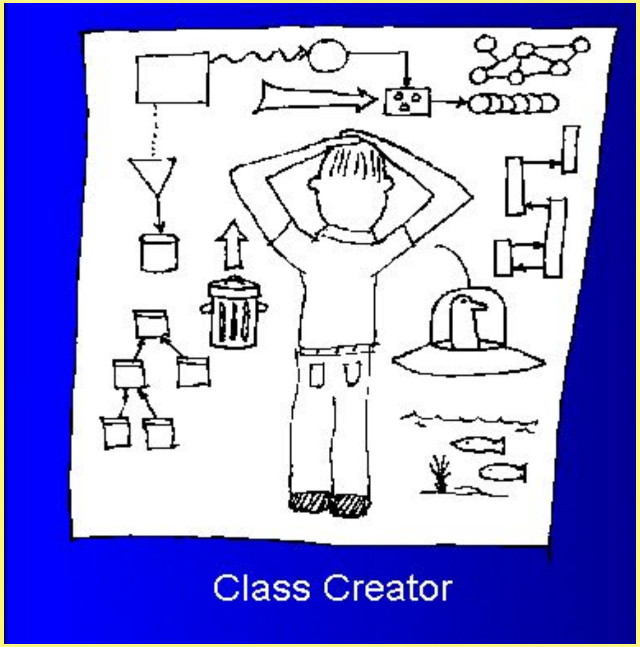
- Create a foo module and implement a Number class that supports the operators specified by the client codes.

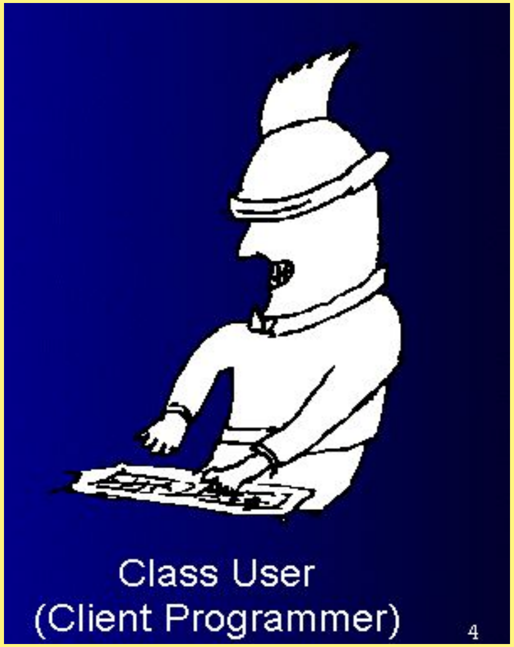
foo.py

```python
class Number:
    def __init__(self, num):
        self.num = num
    def __add__(self, other):
        return Number(self.num + other.num)
    def __repr__(self):
        return f'Number: {self.num}'
    def __str__(self):
        return f'{self.num}'
```

foo.py
L2_Number_Client.ipynb

```
[1]: from foo import Number

[2]: a = Number(20)

[3]: a

[3]: Number: 20

[4]: b = Number(10)

[5]: b

[5]: Number: 10

[6]: c = Number(5)

[7]: print(a + b)

30

[8]: print(a + b + c)

35
```





Object-Oriented Design

- The essence of object-oriented design is describing a system in terms of **a set of cooperating classes** and their interfaces. Each class provides a set of services through its interface. Other components are users or clients of the services.

Guidelines for Object-Oriented Design

1. **Look for object candidates.**
2. **Identify instance variables.**
3. **Think about interfaces.**
4. **Refine nontrivial methods.**
5. **Design iteratively.**
6. **Try out alternatives.**
7. **Keep it simple.**

- **Like all design, object-oriented design is part art and part science**
- **The best way to learn about design is to do it. The more you design, the better you will get.**

Summary

This module covered:

- The important mechanisms of class.
- Putting classes into modules for reuse in other client codes.
- Guidelines and a case study for object-oriented design.