Week 5

Python OOP II

Agenda

- Copying
  - Objects are mutable
  - Shallow copy and deep copy
- Matplotlib & plotting
  - Installation
  - Basic syntax
- Inheritance
- More syntaxes of Python OOP (optional)

Copying

Objects are mutable

```
class Point:
    """A class to represent a two-dimensional point"""
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def distance(self, point2):
        dx = self.x - point2.x
        dy = self.y - point2.y
        return (dx**2 + dy**2)**0.5

    def equals(self, point2):
        return (self.x == point2.x) and (self.y == point2.y)
```

```
In [3]: p1 = Point(2,3)

In [4]: p2 = p1

In [5]: p2.x = 4

In [6]: p2.y = 5

In [7]: p1.x
Out[7]: 4

In [8]: p1.y
Out[8]: 5
```

Objects are mutable. Since p1 and p2 refer to the same object, changes of p2 will also occurred on p1.

Shallow copy and deep copy

```
class Polygon:
    """
    A polygon class with a list of points
    """

    def __init__(self):
        self.points = []

    def add_point(self, x, y):
        self.points.append(Point(x, y))

    def get_point(self, index):
        #check that the index is valid
        if 0 < index < len(self.points):
            return self.points[index-1]
        else:
            return
```

We define a Polygon class.

Each point in the Polygon is an instance of the Point class.

Shallow copy

```
p1 = Polygon()

p1.add_point(1, 2)

p1.add_point(3, 3)

p1.add_point(0, 0)

p2 = copy.copy(p1)

p1
<__main__.Polygon at 0x109183160>

p2
<__main__.Polygon at 0x10917cb70>

p1 == p2
False
```

p1 and p2 have different memory addresses. Also, id(p1) and id(p2) are different.

But this is **shallow copy**. The points of p1 and p2 are the same. They have the same ids.

```
p2.get_point(1)
<Point.Point at 0x109183390>

p1.get_point(1)
<Point.Point at 0x109183390>

id(p1.get_point(1))
4447548304

id(p2.get_point(1))
4447548304
```

Deep copy

```
p2 = copy.deepcopy(p1)

p1
<__main__.Polygon at 0x109183160>

p2
<__main__.Polygon at 0x10918bc18>

p1.get_point(1)
<Point.Point at 0x109183390>

p2.get_point(1)
<Point.Point at 0x10918bf28>

id(p2.get_point(1))
4447584040

id(p1.get_point(1))
4447548304
```

By using the deep copy, p1 and p2 are completely different.

id(p1) and id(p2) are different and the points they refer to are also different.

Inheritance

Inheritance

- Inheritance allows a new class to extend an existing class.
- The new class inherits the data attributes and methods of the class it extends.

Triangle class

We want a triangle class.

- Instead of writing everything from zero, we can take use of the Polygon class.
- We can let the Triangle inherits methods from the Polygon.

The syntax:

```
class Triangle(Polygon):
    """
    A triangle class with a list of points
    """
    def __init__(self):
        super().__init__()
```

Specify the name of the superclass

If you want to use the `__init__()` method of the superclass when you initializing the child class, you should call the method by `super().__init__()`

Triangle class

```
class Triangle(Polygon):
    """
    A triangle class with a list of points
    """
    def __init__(self):
        super().__init__()

    def remove_point(self, index):
        #check that the index is valid
        if 0 < index < len(self.points):
            del self.points[index]
```

We can define methods in the child class for its own.

`dir(Triangle)` shows all the methods belong to the Triangle class.

```
In [6]: dir(Triangle)
 'add_point',
 'get_point',
 'plot',
 'remove_point']
```

You can see some methods are inherited from Polygon while some are defined by itself.

```
In [8]: dir(Polygon)
 'add_point',
 'get_point',
 'plot']
```

Function overloading

```
class Triangle(Polygon):
    """
    A triangle class with a list of points
    """
    def __init__(self):
        super().__init__()


    def remove_point(self, index):
        #check that the index is valid
        if 0 < index < len(self.points):
            del self.points[index]

    def add_point(self, x, y):
        #check the number of points
        if len(self.points) < 3:
            self.points.append(Point(x, y))
        else:
            print("No more points. It is an triangle.")
```

Child class can change the methods that inherited from the superclass simply by redefining it.

We check if the number of points is less than 3 before adding new points since it is a triangle.

```
In [38]: triang_1 = Triangle()

In [39]: triang_1.add_point(0, 0)

In [40]: triang_1.add_point(0, 3)

In [41]: triang_1.add_point(4, 0)

In [42]: triang_1.add_point(4, 2)
No more points. It is an triangle.
```

built-in functions related to inheritance

- `isinstance(instance, class_name)`
- `issubclass(subclass_name, superclass_name)`

```
In [43]: isinstance(triang_1, Triangle)
Out[43]: True

In [44]: issubclass(Triangle, Polygon)
Out[44]: True
```

Terminology

- superclass, also called base class, parent class
- subclass, also called derived class, child class
- polymorphism: subclass can re-implement superclass' functions.

"Is a" relationship: indicates the inheritance exists among the classes

When you define a new class, you might think whether to inherit something from a existing class. You can use this "is a" relationship to find the superclass.

Inheritance

```
class Employee:

    raise_amt = 1.04

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@nyu.edu"
        self.pay = pay

    def fullname(self):
        return "{} {}".format(self.first, self.last)

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)


developer_1 = Employee("Nishant", "Mohanchandra", 50000)
developer_2 = Employee("Bruce", "Xie", 40000)

print(developer_1.pay)
developer_1.apply_raise()
print(developer_1.pay)
print(developer_1.email)
print(developer_2.email)
```

```
50000
52000
nishant.mohanchandra@nyu.edu
bruce.xie@nyu.edu
```

Inheritance: Examp

Two Subclasses of Employee

```
class Developer(Employee):

    raise_amt = amt = 1.10

    def __init__(self, first, last, pay, prog_lang):
        super().__init__(first, last, pay)
        #alternatively: Employee.__init__(self, first, last, pay)
        self.prog_lang = prog_lang

class Manager(Employee):

    def __init__(self, first, last, pay, employees = None):
        super().__init__(first, last, pay)
        if employees is None:
            self.employees = []
        else:
            self.employees = employees

    def add_employee(self, employee):
        if employee not in self.employees:
            self.employees.append(employee)

    def remove_employee(self, employee):
        if employee in self.employees:
            self.employees.remove(employee)

    def print_employees(self):
        for emp in self.employees:
            print("-->", emp.fullname())
```

Inheritance: Example

Sample input:

```
developer_1 = Developer("Nishant", "Mohanchandra", 50000, "Java")
developer_2 = Developer("Bruce", "Xie", 40000, "Python")

manager_1 = Manager("Keith", "Ross", 200000, [developer_1, developer_2])

print(manager_1.email)
manager_1.print_employees()
manager_1.remove_employee(developer_2)
print("After change")
manager_1.print_employees()
```

Sample output:

```
keith.ross@nyu.edu
--> Nishant Mohanchandra
--> Bruce Xie
After change
--> Nishant Mohanchandra
```

Inheritance: Example

- Use of `isinstance` Function

```
print(isinstance(manager_1, Manager))
print(isinstance(manager_1, Developer))
print(isinstance(manager_1, Employee))
print(isinstance(developer_1, Manager))
print(isinstance(developer_1, Developer))
print(isinstance(developer_1, Employee))
```

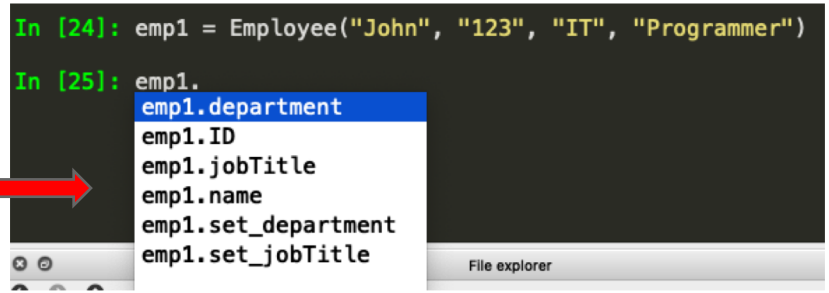
```
True
False
True
False
True
True
```

More about Python OOP

Hide the attributes (optional)

An attribute can be hidden by **starting its name** with:
"_" (**weakly** hidden, for indicating it is a private attribute)
"__"(**strongly** hidden, can **not** be inherited)

```
class Employee:
```
Not hide
```
    def __init__(self, name, ID, dept, jobTitle):
        self.name = name
        self.ID = ID
        self.department = dept
        self.jobTitle = jobTitle
```




```
class Employee:
```
hide
```
    def __init__(self, name, ID, dept, jobTitle):
        self._name = name
        self._ID = ID
        self._department = dept
        self._jobTitle = jobTitle
```

By hiding the attributes, it is harder for accessing the attributes directly. So, it can reduce the occurrence of unwanted modifications on attributes.

Decorator(optional)

Decorator:

`@property`: let a method be “accessed” like an attribute (decorator for getters)

```python
class Employee:

    def __init__(self, name, ID, dept, jobTitle):
        self._name = name
        self._ID = ID
        self._department = dept
        self._jobTitle = jobTitle

    #getter
    @property
    def name(self):
        return self._name

    @property
    def ID(self):
        return self._ID

    def get_department(self):
        return self._department

    def get_jobTitle(self):
        return self._jobTitle
```

```
In [33]: emp1 = Employee("John", "123", "IT", "Programmer")

In [34]: emp1.name
Out[34]: 'John'

In [35]: emp1.ID
Out[35]: '123'

In [36]: emp1.get_department()
Out[36]: 'IT'

In [37]: emp1.get_jobTitle()
Out[37]: 'Programmer'
```

No need for the parenthesis, just like calling the attributes directly.

Decorators (optional)

Decorator:

`@some_property.setter`: define a setter for a defined property.

```
class Employee:

    def __init__(self, name, ID, dept, jobTitle):
        self._name = name
        self._ID = ID
        self._department = dept
        self._jobTitle = jobTitle

    #getter
    @property
    def name(self):
        return self._name
     #setter
    @name.setter
    def name(self, new_name):
        self._name = new_name
    @property
    def ID(self):
        return self._ID
     #setter
    @ID.setter
    def ID(self, new_ID):
        self._ID = new_ID
```

It defines a setter for the property name we defined above.

```
In [39]: emp1 = Employee("John", "123", "IT", "Programmer")

In [40]: emp1.name
Out[40]: 'John'

In [41]: emp1.name = "Mary"

In [42]: emp1.name
Out[42]: 'Mary'
```

Now, we can use the getter and setter like directly calling the attributes.

Iterator Class(optional)

Python **iterator object** must implement two special methods `__iter__()` and `__next__()` collectively called the **iterator protocol**.

```
class Polygon:
    """
    A polygon class with a list of points
    """

    def __init__(self):
        self.points = []
        self.idx = 0
```

Returns an iterator object.
Is implicitly called at start of loops

```
    def __iter__(self):
        self.idx = 0
        return self

    def __next__(self):
        if self.idx <= len(self.points)-1:
            result = self.points[self.idx]
            self.idx += 1
            return result
        else:
            raise StopIteration
```

Returns the next value.
Is implicitly called at each loop increment

Reference

Python OOP Tutorial: https://www.youtube.com/playlist?list=PL-osiE80TeTsqhIuOqKhwlXsIBIdSeYtc

You should watch the above tutorial 1-4; the 5th and 6th are optional.