CIS 112

Intro to Programming Using Python

Module 2 Part 2

Agenda for the Day!

- More in depth on Classes and OOP
  - class and instance data;
  - shallow and deep operations;
  - inheritance and polymorphism;
  - composition and inheritance;
  - copying object data;

Advanced Class and OOP Topics

Class vs. Instance Data

Class vs. Instance Data

- **Instance variables**
  - This kind of variable exists when and only when it is explicitly created and added to an object. This can be done during the object's initialization, performed by the __init__ method, or later at any moment of the object's life. Furthermore, any existing property can be removed at any time.
  - Each object carries its own set of variables - they don't interfere with one another in any way. The word instance suggests that they are closely connected to the objects (which are class instances), not to the classes themselves.

Class-Level Data

- **Class variables**
  - Class variables are defined within the class construction, so these variables are available before any class instance is created. To get access to a class variable, simply access it using the class name, and then provide the variable name.
  - Class variables and instance variables are often utilized at the same time, but for different purposes. As mentioned before, class variables can refer to some meta information or common information shared amongst instances of the same class.
- In Python, a class variable is a variable that is defined within a class and outside of any class method. It is a variable that is shared by all instances of the class, meaning that if the variable's value is changed, the change will be reflected in all instances of the class. Class variables help store data common to all instances of a class.

```
class Employee:
    # Class variable
    office_name = 'XYZ Private Limited'

    # Constructor
    def __init__(self, employee_name, employee_ID):
        self.employee_name = employee_name
        self.employee_ID = employee_ID

    def show(self):
        print("Name:", self.employee_name)
        print("ID:", self.employee_ID)
        print("Office name:", Employee.office_name)

# create Object
e1= Employee('Ram', 'T0166')
print('Before')
e1.show()

# Modify class variable
Employee.office_name = 'PQR Private Limited'
print('After')
e1.show()
```

Copying and shallow vs. deep operations

Shallow and Deep Operations

```
new_list = list(original_list)
new_dict = dict(original_dict)
new_set = set(original_set)
```

```
import copy
xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
zs = copy.deepcopy(xs)
```

- Assignment statements in Python **do not create copies of objects**, they only **bind names to an object**. For immutable objects, that usually doesn't make a difference.
- But for working with mutable objects or collections of mutable objects, you might be looking for a way to create **"real copies"** or **"clones"** of these objects.
  - Essentially, you'll sometimes want copies that you can modify withoutautomatically modifying the original at the same time.
- A **shallow copy** means constructing a new collection object and then populating it with **references** to the child objects found in the original.
  - In essence, a shallow copy is only one level deep. The copying process does not recurse and therefore won't create copies of the child objects themselves.
- A **deep copy** makes the copying process recursive. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original.
  - Copying an object this way walks the whole object tree to create a fully independent clone of the original object and all of its children.

Shallow and Deep Cont.

```
class Rectangle:
    def __init__(self, topleft, bottomright):
        self.topleft = topleft
        self.bottomright = bottomright
```

- Shallow and deep copy operations are equally applicable to all python classes, including custom classes, so we need to be mindful of that behavior when utilizing copy operations

```
rect = Rectangle(Point(0, 1), Point(5, 6))
srect = copy.copy(rect)
```

```
>>> rect.topleft.x = 999
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

```
>>> drect = copy.deepcopy(srect)
>>> drect.topleft.x = 222
>>> drect
Rectangle(Point(222, 1), Point(5, 6))
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

Inheritance, Extension, and Polymorphism

Inheritance

- Inheritance allows us to define a new class that inherits all the methods and properties from another class while also allowing us to extend the behavior of the new 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)

#Use the Person class to create an object, and then execute the
printname method:

x = Person("John", "Doe")
x.printname()
```

```
class Student(Person):
  pass

x = Student("Mike", "Olsen")
x.printname()
```

Extending

- ‘Pass’ing a class through will default construct the prior attributes.
  - Setting a constructor for the child class will override the parent’s attributes
- To inherit while extending, we can leverage the super() function.
  - This allows us to maintain all the constructed elements of the parent class while still allowing to add new attributes
- We can also add additional functionality in terms of child class methods.

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class
of", self.graduationyear)
```

Polymorphism

- “I **bow** to no man!” I shouted from the **bow** of the ship, as I readied my **bow** and arrow and let sail an arrow, carrying a rope, that flew perfectly through a loop, tying a **bow.**”
- The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.
  - len(), +

Inheritance and Class Polymorphism

- Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.
  - student.display()/teacher.display()
- Inheritance rules also apply to polymorphism
  - Children inherit both attributes and methods from parents
  - Children also have the ability to overwrite/customize methods
    - You can almost think of an inherited method as a default value. It can be overwritten but exists automatically if untouched.