OBJECT ORIENTED PROGRAMMING

CS 3080: Python Programming

UCCS University of Colorado Colorado Springs

What Is Object-Oriented Programming (OOP)?

- Is a **programming paradigm** which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
  - *For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running.*
- OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Classes in Python

- Each thing or object is an instance of some class.
- Think of a class as the idea of how something should be defined. A class provides structure! **It is a blueprint**.

- Let's say you want to track a number of different animals and their age.
  - *List?* *['Deer',8]* *What about more animals?*
    What about adding other properties?
    This lacks organization! And it is where we need classes.

- The Animal() class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal's name or age is.

Python Objects (Instances)

- While the class is the blueprint, an **instance** is a copy of the class with actual values, literally an object belonging to a specific class.
- It’s not an idea anymore; it’s an actual animal, like a dog named Roger who’s eight years old.

Defining a class

```
class Dog:      # CamelCase notation starting with a capital letter
   pass         # pass is used to create empty classes, methods,
                # whiles, if, else, etc
```

Instance attributes

```
class Dog:

    # Initializer / Instance Attributes
    def __init__(self, name, age):  # Similar to a constructor
            self.name = name    # Init default values/state
            self.age = age      # Self references to an instance of
                                # this class
                                # Instance attributes are specific to
                                # each object
```

Class attributes

```
class Dog:

    species = 'mammal'          # Class attributes are the same for all
                                # instances when creating them


    # Initializer / Instance Attributes
    def __init__(self, name, age):
            self.name = name
            self.age = age

# So while each dog has a unique name and age, every dog will be a
mammal.
```

Instantiating Objects

```
# Instantiate the Dog object
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 6)

# Is Philo a mammal?
if philo.species == "mammal":
      print("{} is a {}!".format(philo.name, philo.species))
```

Instantiating Objects

```
# Instantiate the Dog object
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 6)

# Is Philo a mammal?
if philo.species == "mammal":
      print("{} is a {}!".format(philo.name, philo.species))

# Philo is a mammal
```

Instantiating Objects

```
# Instantiate the Dog object
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 6)

# Is Philo a mammal?
if mikey.species == "mammal":
      print("{} is a {}!".format(mikey.name, mikey.species))
```

Instantiating Objects

```
# Instantiate the Dog object
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 6)

# Is Philo a mammal?
if mikey.species == "mammal":
      print("{} is a {}!".format(mikey.name, mikey.species))

# Mikey is a mammal
```

Instantiating Objects

```
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 2)
bella = Dog(”Bella", 9)
lucy = Dog(”Lucy", 4)
```

Write a function that prints information of the oldest dog:

“The oldest dog is Bella who’s 9 years old.”

Instance methods

- Instance methods are defined inside a class and are used to **get the contents of an instance**. They can also be used to perform operations with the attributes of our objects.

```
class Dog:

        …

        def description(self):
                return "{} is {} years old".format(self.name, self.age)
```

Instance methods

- Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to **perform operations with the attributes of our objects**.

```
class Dog:

        …

        def addLastname(self, lastname):
                self.name += ‘ ‘ + lastname
```

Python Object Inheritance

- Inheritance is the process by which one class takes on the attributes and methods of another.
- Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.

Python Object Inheritance

```
class Person:

    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def description(self):
        return self.firstname + " " + self.lastname
```

Python Object Inheritance

```
class Employee(Person):

    def __init__(self, first, last, staffnum):
        Person.__init__(self, first, last)
        self.staffnumber = staffnum

    def employeeDescription(self):
        return self.description() + ", " +  self.staffnumber
```

Python Object Inheritance

```
marge = Person("Marge", "Simpson")
homer = Employee("Homer", "Simpson", "1007")

print(marge.description())              # Marge Simpson
print(homer.description())       # Homer Simpson
print(homer.employeeDescription())      # Homer Simpson, 1007
```

Python Object Inheritance

We can override methods from the parent class:

```
class Employee(Person):

    def __init__(self, first, last, staffnum):
        Person.__init__(self, first, last)
        self.staffnumber = staffnum

    def description(self):
        return super().description() + ", " +  self.staffnumber
```

Python Object Inheritance

```
marge = Person("Marge", "Simpson")
homer = Employee("Homer", "Simpson", "1007")

print(marge.description())              # Marge Simpson
print(homer.description())              # Homer Simpson, 1007
```

Python Object Inheritance

Two ways to calling parent methods from a child class:

- *ClassName.method(self, arg1, arg2) == super().method(arg1, arg2)*

```
class Employee(Person):

    def __init__(self, first, last, staffnum):
        Person.__init__(self, first, last)
        self.staffnumber = staffnum

    def description(self):
        return super().description() + ", " +  self.staffnumber
```

issubclass()

```
class Base(object):
    pass

class Derived(Base):
    pass

print(issubclass(Derived, Base))    # True
print(issubclass(Base, Derived))    # False
```

isinstance()

```
d = Derived()
b = Base()

print(isinstance(d, Derived))          # True
print(isinstance(b, Base))             # True

# b is not an instance of Derived
print(isinstance(b, Derived))          # False

# But d is an instance of Base
print(isinstance(d, Base))             # True
```

Multiple inheritance

```
class Base1(object):
    def __init__(self):
        self.str1 = ”String 1"

class Base2(object):
    def __init__(self):
        self.str2 = ”String 2"

class Derived(Base1, Base2):
    def __init__(self):
        Base1.__init__(self)
        Base2.__init__(self)

    def printStrs(self):
        print(self.str1, self.str2)

ob = Derived()
ob.printStrs()                  # String 1 String 2
```

Multiple inheritance

```
class Base1(object):
    ...
    def method(self):
        print("Base1 is used")


class Base2(object):
    ...
    def method(self):
        print("Base2 is used")
                        # When same methods, the
ob = Derived()          # first base inherited is called
ob.method()             # Base1 is called
```

Python looks for the methods from parent class's listed left to right.

For attributes, it will be the last __init__ called.

Printing objects

```
class Test:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return ”Test: a is {}”.format(self.a)

t = Test(1234)
print(t)                    # Test: a is 1234

# If no __str__ defined the output would be
# <__main__.Test instance at 0x7fa079da6710>
```

Encapsulation

- Encapsulation restrict access to methods and variables to prevent data from direct modification.
  - *Denote **private attributes** using single or double underscore as prefix.*
    "_" or "__".

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.__maxprice = 1000             # This has no effect!
c._Computer__maxprice = 1000    # But we can modify it by this tricky syntax
```

Polymorphism

- Polymorphism means that different types respond to the same methods and attributes.
- Polymorphism is a fancy word that just means the same function or attribute is defined on objects of different types.
- This permits your code to use entities of different types at different times.
- Polymorphism can be carried out through:
  - ***inheritance**, with subclasses making use of base class methods or overriding them.*
  - *Having the same method names in several classes or subclasses with different implementations*

Polymorphism with class methods

```
class Shark:
    def swim(self):
        print("The shark is swimming.")

class Clownfish:
    def swim(self):
        print("The clownfish is swimming.")

sammy = Shark()
casey = Clownfish()

for fish in (sammy, casey):
    fish.swim()
```

```
# The shark is swimming.
# The clownfish is swimming.
```

Polymorphism with a function

```
def inThePacific(fish):
    fish.swim()

sammy = Shark()
casey = Clownfish()

inThePacific(sammy)
inThePacific(casey)

# The shark is swimming.
# The clownfish is swimming.
```