Object-oriented programming (OOP)

One of the most popular programming paradigms is OOP. Number of programming languages support OOP, e.g., Java, SmallTalk, C++,..etc. In this lecture, we show how we can use Python to carry out some OOP.

Created by John C. S. Lui on May 31, 2018.

Concept of Classes

One important concept in OOP is the use of classes. Class is a way to combine data and behavior of data within a construct. For example, let's consider what you'd need to do if you were creating a rocket ship in a physics simulation. One of the first things you'd want to track are the x and y coordinates of the rocket. Here is what a simple rocket ship class looks like in code:

In [ ]:
```python
# Rocket is a class which simulates a rocket ship
class Rocket():

    def __init__(self):      # Each rocket has an (x,y) position, and these are the *data* of this class
        self.x = 0
        self.y = 0

    def move_up(self):       # a method which moves the rocket ship up by 1 unit in the y-coordinate
        self.y = self.y + 1


# One can *instantiate* an instance of the Rocket class
my_rocket = Rocket()       # instantiate an instance

# my_rocket has a copy of each of the class's variables,
# and it can do any action that is defined for the class.

print(my_rocket)           # shows that my_rocket is stored at a particular location in memory
print('my_rocket x value is = ', my_rocket.x, ", my rocket y value is = ", my_rocket.y) # access instance's
```

In [ ]:

```
# Let's see how we can invoke the class's method

# Rocket is a class which simulates a rocket ship
class Rocket():

    def __init__(self):      # Each rocket has an (x,y) position, and these are the *data* of this class
        self.x = 0
        self.y = 0

    def move_up(self):       # a method which moves the rocket ship up by 1 unit in the y-coordinate
        self.y = self.y + 1


# One can *instantiate* an instance of the Rocket class
my_rocket = Rocket()       # instantiate an instance

for counter in range(3):
    my_rocket.move_up()    # invoke class function

print('my_rocket x value is = ', my_rocket.x, ", my rocket y value is = ", my_rocket.y)
```

In [ ]:
```python
#  The following code shows that one can create **multiple instances** of the Rocket class

class Rocket():
    def __init__(self):       # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):        # Increment the y-position of the rocket.
        self.y += 1

# Create a fleet of 5 rockets, and store them in a list.
my_rockets = []
for x in range(0,5):    # go through the loop 5 times
    new_rocket = Rocket()
    my_rockets.append(new_rocket)    # my_rocket contains 5 Rocket class instances

# Show that each rocket is a separate object.
for rocket in my_rockets:
    print(rocket)
```

In [ ]:

```
#  The following code shows a more elegant way to create **multiple instances** of the Rocket class
#  using **list comprehension**

class Rocket():
    def __init__(self):      # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):       # Increment the y-position of the rocket.
        self.y += 1

# Create a fleet of 5 rockets, and store them in a list.
my_rockets = [Rocket() for x in range(0,5)]  #   This is a more elegant way to create via list comprehensio

# Show that each rocket is a separate object.
for rocket in my_rockets:
    print(rocket)

my_rockets[0].move_up()
my_rockets[1].move_up()
my_rockets[1].move_up()
my_rockets[3].move_up()
my_rockets[3].move_up()
my_rockets[3].move_up()
my_rockets[4].move_up()
my_rockets[4].move_up()

for rocket in my_rockets:    # Let's display their y-values
    print('For rocket in memory:', rocket, ', its altitude is: ', rocket.y)
```

OOP

Classes are the core component of OOP. When you want to use a class in one of your programs,
you make an object (or instance) from that class, which is where the phrase "object-oriented" comes from.
Python itself is not tied to OOP, but can also be used as a functional programming language (see this later).
Let's introduce some *terminologies* for OOP

- **attribute** is a piece of information or data within a class. E.g., x and y in Rocket().
- **behavior** is an action that defined within a class. E.g., mov_up() in Rocket().
- **object** is a particular instance of a class. E.g., my_rocket

Let's us try to refine our Rocket class.

Accepting parameters for the init() method

We can generalize the **init**() method by adding a couple keyword arguments so that new rockets
can be initialized at any position.

In [ ]:
```python
class Rocket():

    def __init__(self, x=0, y=0):     # using keywords with default values
        self.x = x
        self.y = y

    def move_up(self):                # Increment the y-position of the rocket.
        self.y += 1

# Make a series of rockets at different starting places.
rockets = []
rockets.append(Rocket())
rockets.append(Rocket(0,10))
rockets.append(Rocket(100,0))

# Show where each rocket is.
for index, rocket in enumerate(rockets):  # let's look up the documentation of enumerate() function
    print("Rocket %d is at (%d, %d)." % (index, rocket.x, rocket.y))
```

Accepting parameters in a method

Let's now generalize it to any method.

In [ ]:
```
class Rocket():

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

# a new method move_rocket() which we can move in x and y coordinates
    def move_rocket(self, x_increment=0, y_increment=1):  # note that parameters are initialized
        self.x += x_increment
        self.y += y_increment

# Create three rockets.
rockets = [Rocket() for x in range(0,3)]   # use list comprehension

# Move each rocket a different amount.
rockets[0].move_rocket()
rockets[1].move_rocket(10,10)
rockets[2].move_rocket(-10,0)

# Show where each rocket is.
for index, rocket in enumerate(rockets):
    print("Rocket %d is at (%d, %d)." % (index, rocket.x, rocket.y))
```

Adding a new method

Let's try to add a new method so that we can compute the Euclidean distance between to rockets.

In [ ]:
```
# We need the math library so that we can use various functions, e.g., sqrt()
from math import sqrt

class Rocket():

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

# Make two rockets, at different places.
rocket_0 = Rocket()
rocket_1 = Rocket(10,5)

# Show the distance between them.
distance = rocket_0.get_distance(rocket_0)
print("The rockets are %f units apart." % distance)
distance = rocket_0.get_distance(rocket_1)
print("The rockets are %f units apart." % distance)
```

Concept of Inheritance in OOP

One of the most important goals of the OOP is the creation of stable, reliable, reusable code.
If you had to create a new class for every kind of object you wanted to model, you would hardly have any reusable code. In Python (which supports OOP_, one class can **inherit** from another class. This means you can create a new class on an existing class; the new class inherits all of the attributes and behavior of the class it is based on. A new class can **override** any undesirable attributes or behavior of the class it inherits from, and it can add any new attributes or behavior that are appropriate.

The original class is called the **parent class**, and the new class is a **child** of the parent class.
The parent class is also called a **superclass**, and the child class is also called a **subclass**.

The child class inherits *all attributes and behavior from the parent class*, but any attributes that are defined in the child class are *not available to the parent class*. The child class can also override behavior of the parent class. If a child class defines a method that also appears in the parent class, objects of the child class will use the new method rather than the parent class method.

Let's illustrate this concept.

In [ ]:
```python
from math import sqrt

class Rocket():
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):      # Shuttle is a child of the Rocket() class, flights_completed states # of compl

    def __init__(self, x=0, y=0, flights_completed=0):  # this is the init() function for the Shuttle() cla
        super().__init__(x, y)                           # it first calls its parent's __init__() method
        self.flights_completed = flights_completed      # it also does its own initialization to its variab

shuttle = Shuttle(10,0,3)
print(shuttle)
print("This shuttle has x = ", shuttle.x, "y = ", shuttle.y, "# of completed flights = ", shuttle.flights_c
```

Let's generate some rockets and shuttles and compute their distance

In [ ]:
```python
from math import sqrt
from random import randint   # this generates a random integer between a lower and upper bound

class Rocket():

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
```

```
    def move_rocket(self, x_increment=0, y_increment=1):
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed


# Create several shuttles and rockets, with random positions.
#  Shuttles have a random number of flights completed.
shuttles = []
for index in range(0,3):
    x = randint(0,100)  # generate a random number for x
    y = randint(1,100)  # generate a random number for y
    flights_completed = randint(0,10)  # generate a random # for number of completed flight
    shuttles.append(Shuttle(x, y, flights_completed))

rockets = []
for index in range(0,3):
    x = randint(0,100)
    y = randint(1,100)
    rockets.append(Rocket(x, y))

# Show the number of flights completed for each shuttle.
for index, shuttle in enumerate(shuttles):
    print("Shuttle %d has completed %d flights." % (index, shuttle.flights_completed))

print("\n")
# Show the distance from the first shuttle to all other shuttles.
first_shuttle = shuttles[0]
for index, shuttle in enumerate(shuttles):
    distance = first_shuttle.get_distance(shuttle)
    print("The first shuttle is %f units away from shuttle %d." % (distance, index))
print("\n")
# Show the distance from the first shuttle to all other rockets.
for index, rocket in enumerate(rockets):
    distance = first_shuttle.get_distance(rocket)
    print("The first shuttle is %f units away from rocket %d." % (distance, index))
```

More examples

Let's illustrate more on OOP.

In [ ]:
```
class BankAccount(object):
    def __init__(self, balance=0):
        self.balance = balance
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance

my_account1 = BankAccount (200)
# what is the balance in my_account1 ?
print ('my_account 1 balance: ', my_account1.getBalance())

my_account2 = BankAccount ()
# what is the balance in my_account2 ?
print ('my_account 2 balance: ', my_account2.getBalance())
```

In [ ]:
```
## Let's examine the class and dictionary of this class
print(my_account1.__class__)
print (my_account1.__dict__)
```

Objects are referecens

Objects are internally stored as references. So assigning an object only means its reference being copies. Let's illustrate.

In [ ]:
```
husband_account = BankAccount(500)
wife_account = husband_account
wife_account.withdraw(300)

print("hushand account's balance = ", husband_account.balance)
print("wife account's balance = ", wife_account.balance)
```

In [ ]:
```python
## Let's illustrate again the concept of inheritance

class BankAccount(object):
    def __init__(self, balance=0):
        self.balance = balance
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance


class CheckAccount(BankAccount):
    def __init__(self, initBal=0):
        BankAccount.__init__(self, initBal)
        self.checkRecord = {}
    def processCheck(self, number, toWho, amount):
        self.withdraw(amount)
        self.checkRecord[number]= (toWho, amount)
    def checkInfo(self, number):
         if number in self.checkRecord:
            return self.checkRecord[number]

ca = CheckAccount (1000)
ca.processCheck(100, 'CUHK', 328.)
ca.processCheck(101, 'HK Electric', 452.)
print('Check 101 has information of: ', ca.checkInfo(101))
print ('The current balance is: ', ca.getBalance())
ca.deposit(100)
print('The current balance is: ', ca.getBalance())
```

More example about OOP

Let's try to build a calculator which uses reverse polish notation (RPN).

In [ ]:
```python
## First, let us define a stack class
class Stack(object):
    def __init__(self):
        self.storage = []
    def push (self, newValue):
       self.storage.append( newValue )
    def top(self):
        return self.storage[len(self.storage) - 1]
    def pop(self):
        result = self.top()
        self.storage.pop()
        return result
    def isEmpty(self):
        return len(self.storage) == 0

# let's run some program
stackOne = Stack()
stackTwo = Stack()
stackOne.push( 12 )
stackTwo.push( 'abc' )
stackOne.push( 23 )

print('top element for stackOne is: ', stackOne.top())

stackOne.pop()
print('top element for stackOne is: ', stackOne.top())
print('top element for stackTwo is: ', stackTwo.top())
```

In [ ]:

```
## Now define a calculator class
class CalculatorEngine(object):
    def __init__(self):
        self.dataStack = Stack()
    def pushOperand (self, value):
        self.dataStack.push( value )
    def currentOperand ( self ):
        return self.dataStack.top()
    def performBinary (self, fun ):
        right = self.dataStack.pop()
        left = self.dataStack.pop()
        self.dataStack.push( fun(left, right))
    def doAddition (self):
        self.performBinary(lambda x, y: x + y)
    def doSubtraction (self):
        self.performBinary(lambda x, y: x - y)
    def doMultiplication (self):
        self.performBinary(lambda x, y: x * y)
    def doDivision (self):
        self.performBinary(lambda x, y: x / y)
    def doTextOp (self, op):
        if (op == '+'): self.doAddition()
        elif (op == '-'): self.doSubtraction()
        elif (op == '*'): self.doMultiplication()
        elif (op == '/'): self.doDivision()

# Now let's test our program
calc = CalculatorEngine()  # instantiate a calculator
calc.pushOperand( 3 )      # push operand on the stack
calc.pushOperand( 4 )      # push operand on the stack
calc.doTextOp ( '*' )      # do multiplication
print ('The result is = ', calc.currentOperand())
```

In [ ]:
```python
## Now define an RPNCalculator
class RPNCalculator(object):
    def __init__(self):
        self.calcEngine = CalculatorEngine()
    def eval (self, line):
        words = line.split(" ")
        for item in words:
            if item in '+-*/':
                self.calcEngine.doTextOp( item )
            else:
                self.calcEngine.pushOperand( int (item))
        return self.calcEngine.currentOperand()
    def run(self):
        while True:
            line =  input("type an expression: ")
            if len(line) == 0:
                break
            print (self.eval( line ))

# Now let's instantiate an RPN calculator and do some testing
calc = RPNCalculator()
calc.run()
```

In [ ]:

Modules and classes

Python allows you to **save your classes in another file** and then **import** them into the program you are working on. This has the added advantage of isolating your classes into files that can be used in any number of different programs. As you use your classes repeatedly, the classes become more reliable and complete overall.

When you save a class into a separate file, that file is called a **module**. You can have any number of classes in a single module. There are a number of ways you can then import the class you are interested in. Let's illustrate.

In [ ]:
```
# load from rocket.py
from rocket import Rocket    # only import the Rocket() class

rocket = Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))
```

In [ ]:
```
from rocket import Rocket, Shuttle   # load both classes !!!

rocket = Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))

shuttle = Shuttle(1,2,3)
print("\nThe shuttle is at (%d, %d)." % (shuttle.x, shuttle.y))
print("The shuttle has completed %d flights." % shuttle.flights_completed)
```

Several ways to import modules and classes

The general syntax is:

from module_name *import* ClassName

Several ways to import:

- import rocket
  This will load all classes within the rocket module (or file)
    After this, classes are accessed using **dot notation**, or module_name.ClassName (e.g., rocket.Rocket()
- import module_name as local_module_name

Let's see an example

```
In [ ]: # Save as rocket_game.py
        import rocket as rocket_module

        rocket = rocket_module.Rocket()
        print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))

        shuttle = rocket_module.Shuttle()
        print("\nThe shuttle is at (%d, %d)." % (shuttle.x, shuttle.y))
        print("The shuttle has completed %d flights." % shuttle.flights_completed)
```

A module of functions

Let's create a file in the current directory as multiplying.py

```
In [ ]: # Save as multiplying.py
        def double(x):
            return 2*x

        def triple(x):
            return 3*x

        def quadruple(x):
            return 4*x
```

Using the: from *module_name* import *function_name* syntax:

```
In [ ]: from multiplying import double, triple, quadruple

        print(double(5))
        print(triple(5))
        print(quadruple(5))
```

Using the *import module_name* syntax:

```
In [ ]: import multiplying

        print(multiplying.double(5))
        print(multiplying.triple(5))
        print(multiplying.quadruple(5))
```

Using the *import* module_name *as* local_module_name syntax:

```
In [ ]: import multiplying as m

        print(m.double(5))
        print(m.triple(5))
        print(m.quadruple(5))
```

Using the *from* module_name import * syntax:

```
In [ ]: from multiplying import *

        print(double(5))
        print(triple(5))
        print(quadruple(5))
```

```
In [ ]:
```