Object-Oriented Programming

Computer Science & Engineering Department

The Chinese University of Hong Kong

Content

- OOP Concept
- Class, Object, Instance, Inheritance …
- More on Scope

Problem of Bugs

- According to some studies[1]
  - Industry average: about 15-50 errors happened per 1000 lines of code(LOC)
  - Microsoft Applications: about 10-20 defects per 1000 LOC during in house testing, & 0.5 defect per 1000 LOC
- In order to reduce number of bugs in our software, one way is to develop based on established, thoroughly tested and debugged code

1. **Code Complete: A Practical Handbook of Software Construction, Second Edition** 2nd Edition
Steve McConnelMicrosoft Press; 2nd edition (June 19, 2004)

Object-Oriented Programming (OOP)

- One of the most popular programming paradigms
- OOP emphasize the concept of software reuse -
  once a certain program is being written, later others can build a new prototype base on current one with minimal modification

- Or think of building many identical flats with basic functions such as water, electricity
- New tenants can customize each flat into their own preferred style with different paints, furniture, carpets etc.

Object-Oriented Programming (OOP)

- Most important concept is "classes"
- Class : a way to combine both data(variables) and behaviour(functions) within a construct, think of template/blueprint
- We use an example of creating a rocket ship in a physics simulation
- Let say we want to track (x,y) coordinates of the rocket

```python
# Rocket is a class which simulates a rocket ship
class Rocket():
    def __init__(self): # Each rocket has an (x,y) position, and init to 0
        self.x = 0
        self.y = 0

    def move_up(self): # a method which moves the rocket ship up by 1 unit
        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
print('my_rocket x is = ', my_rocket.x, ", my rocket y is = ", my_rocket.y)
```

```
<__main__.Rocket object at 0x0000023D65B964E0>
my_rocket x value is =  0 , my rocket y value is =  0
```

OOP Concepts

- Classes are the core component of OOP
- When we want to use a class in one of your programs, we create an object (or instance) from that class
- OOP Terms
  - **Attribute**
    a piece of information or data within a class. E.g., x and y in Rocket().
  - **Behavior/Method**
    an action that defined within a class. E.g., mov_up() in Rocket().
  - **Object :** a particular instance of a class. E.g., my_rocket

```python
# Rocket is a class which simulates a rocket ship
class Rocket():
    def __init__(self): # Each rocket has an (x,y) position, and init to 0
        self.x = 0
        self.y = 0

    def move_up(self): # a method which moves the rocket ship up by 1 unit
        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
print('my_rocket x is = ', my_rocket.x, ", my rocket y is = ", my_rocket.y)
```

Behavior/Method declarations

```python
# Rocket is a class which simulates a rocket ship
class Rocket():
    def __init__(self): # Each rocket has an (x,y) position, and init to 0
        self.x = 0
        self.y = 0

    def move_up(self): # a method which moves the rocket ship up by 1 unit
        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
print('my_rocket x is = ', my_rocket.x, ", my rocket y is = ", my_rocket.y)
```

Attributes declarations

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

my_rocket = Rocket() # instantiate an instance
for counter in range(3):
    my_rocket.move_up() # invoke class function

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

```
my_rocket x is =  0 , my rocket y is =  3
```

```python
# The following code shows that one can create **multiple instances** of
# 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 instances

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

```
<__main__.Rocket object at 0x000001F7ECBB64E0>
<__main__.Rocket object at 0x000001F7ECBB66D8>
<__main__.Rocket object at 0x000001F7ECC16358>
<__main__.Rocket object at 0x000001F7ECC16390>
<__main__.Rocket object at 0x000001F7ECC163C8>
```

```python
# The following code shows a more elegant way to create **multiple instances** # using **list
comprehension**
# 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 #
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)
```

```
For rocket in memory: <__main__.Rocket object at 0x0000018B732A6588> , its altitude is:  1
For rocket in memory: <__main__.Rocket object at 0x0000018B732A6780> , its altitude is:  2
For rocket in memory: <__main__.Rocket object at 0x0000018B73306400> , its altitude is:  0
For rocket in memory: <__main__.Rocket object at 0x0000018B73306438> , its altitude is:  3
For rocket in memory: <__main__.Rocket object at 0x0000018B73306470> , its altitude is:  2
```

Object-Oriented Programming (OOP)

- We will introduce more concepts through further refine our Rocket class
- When we create our rocket instance, we cannot set the initial value
- __init__ method can be changed to tailor this need

- We change it so that new rockets can be initialized at any position.

```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
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
    print("Rocket {} is at ({}, {})." .format (index, rocket.x, rocket.y))
```

```
Rocket 0 is at (0, 0).
Rocket 1 is at (0, 10).
Rocket 2 is at (100, 0).
```

```
from math import sqrt
    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)
```

```
The rockets are 0.000000 units apart.
The rockets are 11.180340 units apart.
```

Class variables

- Variables defined at class level are shared by all instances
- Initialization only once
- Variables defined using self are unique to each instance

```
class Rocket:
    count = 0
    def __init__(self):
        Rocket.count = Rocket.count + 1
>>>a = Rocket()
>>>b = Rocket()
>>>print (Rocket.count)
2
```

Inheritance (OOP)

- We wish to save our programming effort by using existing reliable program
- OOP allows us to reuse code by creating a new class by inheriting from an existing class
- the new class inherits *all* of the *attributes* and *behavior* of old class

- Can override undesirable features as well as create new ones
- but any attributes that are defined in the child class are *not available to the parent class*

```
from rocket import Rocket
class Shuttle(Rocket):     # Shuttle is a child of the Rocket() class,
                           # x and y are inherited attributes
                           # flights_completed is new attribute
    def __init__(self, x=0, y=0, flights_completed=0): # this is new init()
        super().__init__(x, y)                             # it first calls its super class init()
        self.flights_completed = flights_completed         # it also does its own initialization

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

```
<__main__.Shuttle object at 0x000002AEE18776A0>
This shuttle has x =  10 y =  0 # of completed flights  3
```

More OOP Concepts

- We further illustrate the application of OOP in daily lives
- Consider banking operation, we have accounts opened in bank
- We want to keep track of the balance of the account and be able to withdraw and deposit money into it'

```python
class BankAccount(object): # BankAccount is a child of object - base class for all Python class
    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())
```

```
my_account 1 balance:  200
my_account 2 balance:  0
```

More OOP Concepts

- Objects are internally stored as references
- assigning an object only means its reference being copies

```
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)
```

```
hushand account's balance =  200
wife account's balance =  200
```

```python
class CheckAccount(BankAccount): # CheckAccount, a child of BankAccount, inherits balance
    def __init__(self, initBal=0):
        BankAccount.__init__(self, initBal) # it first calls its super class init() to set balance
        self.checkRecord = {}               # checkRecord is a new dict attribute
    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)    # newly created CheckAccount object got balance of 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())
```

```
Check 101 has information of:  ('HK Electric', 452.0)
The current balance is:  220.0
The current balance is:  320.0
```

```python
class CheckAccount(BankAccount):
    def __init__(self, initBal=0):
        BankAccount.__init__(self, initBal)
        self.checkRecord = {}
    def processCheck(self, number, toWho, amount): # processCheck() is a new behavior
        self.withdraw(amount)        # withdraw() is an inherited behavior on reducing balance
        self.checkRecord[number]= (toWho, amount)
    def checkInfo(self, number):          # checkInfo() is also a new behavior
        if number in self.checkRecord:
            return self.checkRecord[number]
ca = CheckAccount (1000)
ca.processCheck(100, 'CUHK', 328.)          # balance is reduced by 328, and
ca.processCheck(101, 'HK Electric', 452.)     #  then by 452, so balance = 220
print('Check 101 has information of: ', ca.checkInfo(101))
print ('The current balance is: ', ca.getBalance())
ca.deposit(100)                         # deposit() is also an inherited behavior for being called
print('The current balance is: ', ca.getBalance())
```

```
Check 101 has information of:  ('HK Electric', 452.0)
The current balance is:  220.0
The current balance is:  320.0
```

Overriding

- Sometimes necessary for child class to modify or replace the behavior inherited from parent class
- Child class redefines the function using the same name and arguments
- To invoke original parent class function, class name must be explicitly provided (or using super() in most of the cases)

```
class CheckAccount( BankAccount):
    :
    def withdraw(self, amount):
        print ('withdrawing ', amount)
        BankAccount.withdraw(self, amount)
```

Types & Tests

- Each class definition creates a new type

```
>>> print (type(myAccount))
<class '__main__.BankAccount'>
>>> print (type(BankAccount))
<type 'type'>
```

- Test for membership in a class

```
>>> newAccount = CheckAccount(4000)
>>> sndAccount = BankAccount(100)
>>> print (isinstance(newAccount, BankAccount))
True
>>> print (isinstance(newAccount, CheckAccount))
True
>>> print (isinstance(sndAccount, CheckAccount))
False
```

Types & Tests

- issubclass(A,B) returns true if class A is a subclass of B

```
>>> print (issubclass(CheckAccount, BankAccount))
True
```

- Can also perform type checking for built-in types

```
>>> isinstance(3, int)
True
>>> import types
>>> def f():
        pass
>>> isinstance(f, types.FunctionType)
True
```

OOP Application

- Let's try to build a calculator using reverse polish notation (RPN)
- In order for RPN be used, we first need to have a stack
- A stack is a last-in-first-out data structure
- Imagine you want to take a dish from a pile of dishes
- To take the top data from stack is a "pop operation
- To add new data to stack, we have the "push" operation

- We also need to implement some housekeeping operations

Stack in OOP

```
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
```

Stack implementation in Python

```
stackOne = Stack()
stackTwo = Stack()
stackOne.push( 12 )
stackTwo.push( 'abc' )
stackOne.push( 23 )
print(stackOne.top())
>>> 23
stackOne.pop()
print(stackOne.top())
>>>12
print(stackTwo.top())
>>>'abc'
```

Test code

Using stack to build a calculator

```
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 doAddition (self ):
        right = self.dataStack.pop()
        left = self.dataStack.pop()
        self.dataStack.push(left + right)
# …
# similarly for doSubtraction, doMultiplication and doDivision
```

Using stack to build a calculator

```
def doTextOp (self, op):
        if (op == '+'): self.doAddition()
        elif (op == '-'): self.doSubtraction()
        elif (op == '*'): self.doMultiplication()
        elif (op == '/'): self.doDivision()

calc = CalculatorEngine()
calc.pushOperand( 3 )
calc.pushOperand( 4 )
calc.doTextOp ( '*' )
print (calc.currentOperand() )

>>> 12
```

Revisit

Identifiers in Program

- Names of variables, functions, modules can collide with others – same name used unintentionally (Python allows this)
- Managed using name spaces
- Encapsulation of names through levels of abstraction
- Three levels of encapsulation
  - LEGB rule for simple variables
  - Qualified names
  - modules

Scopes, Names, and References

- Scope is property of a name, not a property of a value
- Two names can refer to the same value, and they have different scopes

```
class Box(object):
    def __init__( self, v):
        self.value = v

def newScope(x):
    y = x
    y.value = 42

a = Box(3)
newScope(a)
print (a.value)
>>>42
```

Qualified Names

- A period following a base e.g. **`object.attribute`**
- Base is first determined using LEGB rule
- Names can be qualified include
  - Classes
  - Instances or objects
  - Instances of built-in types e.g. list, dictionary
  - Modules

Qualified Names

- Names resolution are performed using dictionaries
- locals() and globals() return the current scope through dictionary
- Classes store their name space in a field __dict__
- Can be accessed (or modified!) by programmer

```
>>> Rocket.__dict__
{'count': 2, '__module__': '__main__', '__doc__':
 None, '__init__': <function __init__ at
 0x00FDE4F0>}
```

Module

- Just like library
- Can have two ways when used:
  - **`import modName`**
  - **`from modName import attribute`**

- For second way of using module
  - Means construct the module dictionary, the given attribute is then copied into local dictionary
  - Thus the attribute can be used without qualification in local space

Module

- Suppose we want to use bar method in module *foo*
- **`import foo`**
- .. Then use foo.bar to use it
- Two run-time lookups needed in this case :
  1. locate foo,
  2. locate bar

- **`from foo import bar`**
- bar is called directly without qualification
- Only One search required
- More execution efficiency

Avoid Name Space collision

- Can use wild card '*' to import - from mod import *
- Has risk of name collision

```
>>> from math import *
>>> print(e)
2.71828182846
```

- Can Use as clause to avoid

```
>>> e = 42
>>> from math import e as eConst
>>> e
42
```

Modules

- Simply a Python file
- Only the handling of names in modules differs
- Import statement scans a file and execute each statement in program
- Names of all values in module are stored in its own dictionary
- Thus the qualified name modName.x is actually just modName.__dict__['x']

```
>>> import math
>>> print( type(math) )
<class 'module'>
>>> print( math.__dict__['sqrt'] )
<built-in function sqrt>
```

Creating your own module

- Just another Python program
- Only difference is it is being loaded by *import* statement
- Normally contains only classes and function definitions
- Can also have statements inside be executed
- Name of current module is held in internal variable called **`__name__`**
- Top level program executed by Python interpreter is of the name **`__main__`**
- Can use the following to conditionally executing those statements

```
if __name__ == '__main__':
    .. statements
```

supplementary

Appendix

Class scope

- Class has its scope, but not part of LEGB
- A class method can see their surrounding scope, but cannot see the class scope
- Normally it's okay as classes defined at top level

```
def silly():
    x = 12
    class A:
        x = 42  # class variable
        def foo(self):
            print (x)
            print (self.x)   #
    return A()
anA = silly()
anA.foo()
>>> 12
42
```

Multiple Inheritance

- Class definition specify inheritance from more than one class
- Not recommended

```
class A(object):
    def doa(self):
        print ("I'm a")
class B (object):
    def dob(self):
        print ("I'm b")
class C(A,B):
    def doc(self):
        print ("I'm c")
>>> v=C()
>>> v.doc()
I'm c
>>> v.doa()
I'm a
>>> v.dob()
I'm b
```