Python Programming

Classes and Objects

Aug/Sep 2019

Dr. Ram P Rustagi
Professor, CSE Dept
KRP, KSGI
rprustagi@ksit.edu.in

Resources and Acknowledgements

- Intro to Programming with C++
  - Abhiram Ranade, Prof CSE, IIT Bombay
- A first course in programming
  - https://introcs.cs.princeton.edu/python/home/
  - https://introcs.cs.princeton.edu/java/home/
- Python for everybody
  - https://www.py4e.com
- Web Applications for everybody
  - https://www.wa4e.com
- https://education.pythoninstitute.org/course_datas
- https://www.w3schools.com/python/
  - Basic Python Tutorial

Overview

- Overview of programming style
- Basic classes
- Encapsulation
- Inheritance
- Inheritance and composition
- Method Resolution Order (MRO)
- Summary

Programming Exercises:

- Ex 00:
  - Define a class for carrying bitwise operations.
  - The class should support the following.
    - Initialize number of bits (limited to 8, 16, 32 and 64)
      - Any other value, override to 64.
    - `reset()`: Reset all bits to zero.
    - `setbit(n)`: Set the n^{th} bit to `1`.
    - `chkbit(n)`: Check if n^{th} bit is `1`.
      - Returns `True` or `False`
- Time: `7` minutes

Exercise 00 Template

```
class Binary:
  def __init__(self,n):
    # ??

  def reset(self):
    # reset all the bits

  def setbit(self,k):
    # set kth bit

  def chkbit(self,k):
   # return True or False
```

Programming Style

- Procedural
  - Majority of software is developed using it
  - A sequential flow
- Object Oriented
  - Quite young compared to procedural
  - Usefule when used by large team of developers
- Python supports both

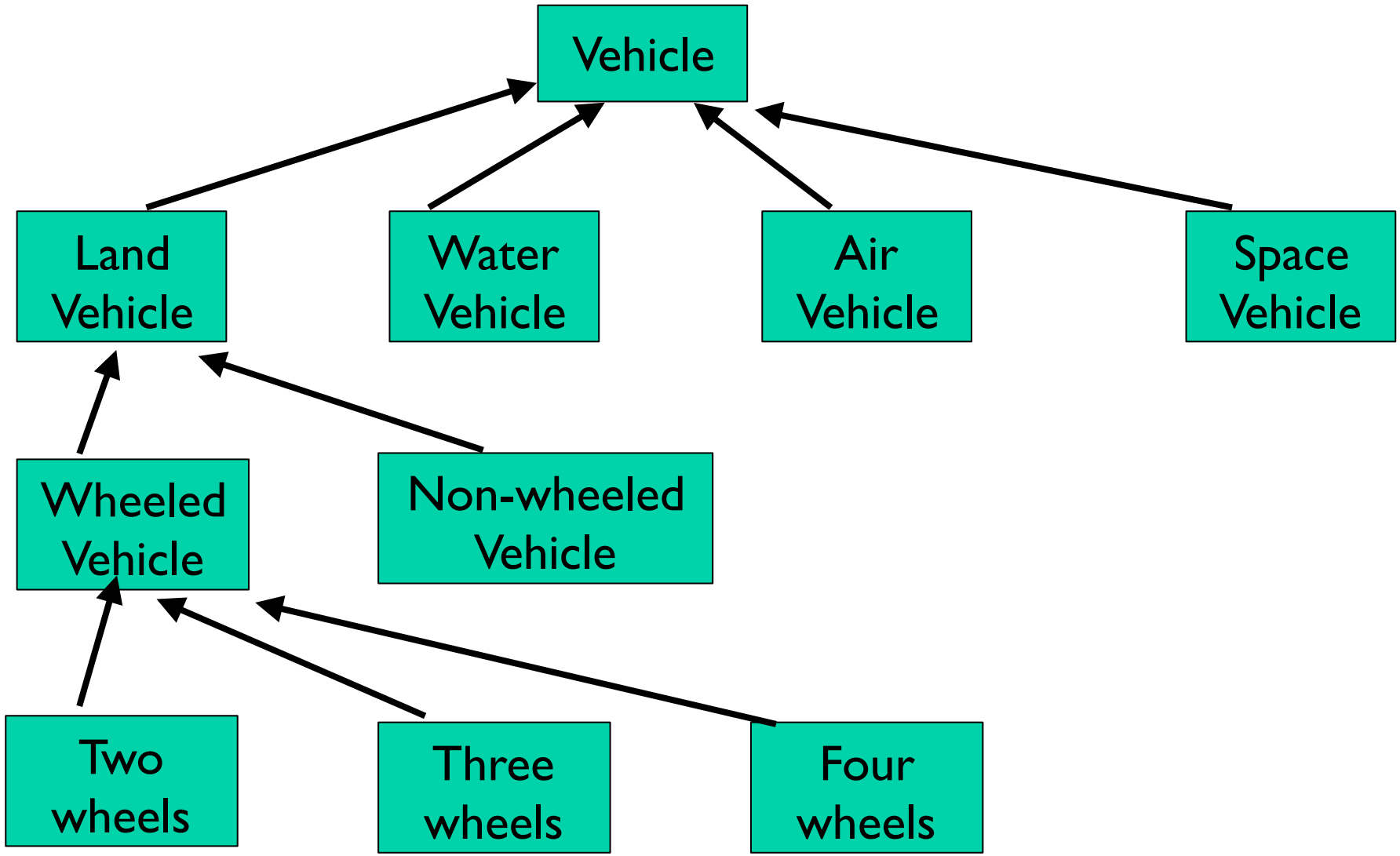
Basics of OOP

- Each object
  - Has set of traits, called properties or attributes
  - Performs a set of activities called methods
- Object
  - An incarnation of idea, expressed in class
  - Reflect real facts, relationships and circumstances
- Objects interact with each other
  - By exchanging data
  - By activating methods
- An object can protect its sensible data
  - Can hide it from unauthorised modifications
- In object, both code and data live together
  - No clear border between data and code

Objects

- Consider an entity (created by human not by nature)
  - Used for transportation
  - Driven by human
  - Moves by obeying the laws of physics
- Vehicle represents such an entity
- Dog or cow does not fit this entity
- Vehicle
  - A broad classification
- Can we define more specific classification further
  - Land vehicles
  - Water vehicles
  - Air vehicles
  - Space vehicles

Vehicle: classification



Direction of arrow points to superclass

Class Inheritance

- Classes form a hierarchy
- An object belonging to a subclass
  - Belongs to all of its superclasses
  - i.e. it inherits all the properties, methods
- An object belonging to a superclass
  - May not belong to any of it subclass
- Defining an object
  - Has a name (defined by noun)
  - Has properties (defined by adjectives)
  - Has methods or activities (defined by verbs)

Example: Stack

- Stack: procdural approach

```
stack = []
def push(val):
   stack.append(val)
def pop():
   val = stack[-1] # the last element
   del stack[-1]
   return val
#-----------------
 push(5)
 push(3)
 push(1)
 print(pop())
 print(pop())
 print(pop())
```

Stack: Procedural approach

- Vulnerable to direct manipulation of `stack[]` data structure
  - A programmar can directly write
  - `stack[0] = 1`
- Managing multiple stacks
  - Need to create another variable `stack2[]`
  - With its own `pop()` and `push()` methods

Stack: Objective Approach

- Ability to hide selected properties, e.g.
  - `stack[]` is not directly accessible.
  - Essentially, provides **encapsulation**
  - Encaspulated values can't be directly accessed
    - if so desired
- Can create multiple stacks without writing new code
- Ability to enrich the stack with new functions
  - Enabled by **inheritance**

Stack: Objective Approach

```
class Stack:
  def __init__(self):
    self.__stk = []

  def push(self, val):
    self.__stk.append(val)

  def pop(self):
    val = self.__stk[-1]
    del(self.__stk[-1])
    return val
```

- Any component starting with `__` becomes private
  - Can be accessed only from within the class
  - Implementation of python's encapsulation concept
  - `pop()` and `push()` are public methods

Stack: Objective Approach

- Using `Stack` class
- Instantiating objects

```
stack1 = Stack()
stack2 = Stack()
```

- Working with created objects

```
stack1.push(5)
stack2.push(3)
stack1.push(4)
stack2.push(2)
print(stack1.pop())
print(stack2.pop())
```

example: `stack2.py`

Stack

- Src: https://education.pythoninstitute.org/course_datas/display/97/837#
- What is the output of following program

```
a_stack = Stack()
b_stack = Stack()
c_stack = Stack()
a_stack.push(1)
b_stack.push(a_stack.pop() + 1)
c_stack.push(b_stack.pop() - 2)
print(c_stack.pop())
```

example: `stack3.py`

Object: Adding Properties

Define a new stack which maintains sum of all the values in it

```
class AddingStack(Stack):
   def __init__(self):
      Stack.__init__(self)
      self.__sum = 0

   def getSum(self):
      return self.__sum

   def push(self, val):
      self.__sum += val
      Stack.push(self,val)

   def pop(self):
      val = Stack.pop(self)
      self.__sum -= val
      return val
```

Using Inherited Stack

- src: https://education.pythoninstitute.org/course_datas/display/97/837#
- Example use of inherited stack
- What will be the output of following?

```
stack = AddingStack()

for i in range(5):
  stack.push(i)

print(stack.getSum())

for i in range(5):
  print(stack.pop())

# example: AddStack.py
```

Attribute's Existence

- What is the output of following code

```
class Class:
  def __init__(self,val):
    if val % 2 != 0:
      self.a = 1
    else:
      self.b = 1


obj = Class(1)
print(obj.a)
print(obj.b)
```

example: `class1.py`

Attribute's Existence

- Correcting the error

```
class Class:
  def __init__(self,val):
    if val % 2 != 0:
      self.a = 1
    else:
      self.b = 1

obj = Class(1)
print(obj.a)
if hasattr(obj, 'b'):
  print(obj.b)
# example: class2.py
```

Attribute's Existence

- What is the output of following

```
class Class:
   a = 1
   def __init__(self, val):
      self.b = val

obj = Class()

print(hasattr(obj, 'b'))
print(hasattr(obj, 'a'))
print(hasattr(Class, 'b'))
print(hasattr(Class, 'a'))
```

example: `classa1.py`

Hidden Methods

- Output of following snippet

```
class Hidden:
  def visible(self):
    print("visible")

  def __hidden(self):
    print("hidden")

obj = Hidden()
obj.visible()
try:
  obj.__hidden()
except:
  print("failed")
```

example: hidden1.py

Inheritance

- Output of following Code

```
class Vhcl:
  pass
class LandVhcl(Vhcl):
  pass
class TrackedVhcl(LandVhcl):
  pass
for v1 in [Vhcl, LandVhcl, TrackedVhcl]:
  for v2 in [Vhcl, LandVhcl, TrackedVhcl]:
    print(issubclass(v1,v2),end='\t')
  print()
```

Ans:

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

example: `vehicle1.py`

Inheritance

- Output of following Code

```
class Vhcl:
  pass
class LandVhcl(Vhcl):
  pass
class TrackedVhcl(LandVhcl):
  pass
vh = Vhcl()
lv = LandVhcl()
tv = TrackedVhcl()

for vr in [vh, lv, tv]:
  for cl in [TrackedVhcl, LandVhcl, Vhcl]:
    print(isinstance(vr, cl),end='\t')
  print()
```

Ans:

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

example: `vehicle2.py`

Inheritance

- Output of following code

```
class Level0:
  Var0 = 100
  def __init__(self):
    self.var0 = 101
  def fun0(self):
    return 102

class Level1(Level0):
  Var1 = 200
  def __init__(self):
    super().__init__()
    self.var1 = 201
  def fun1(self):
    return 202

class Level2(Level1):
  Var2 = 300
  def __init__(self):
    super().__init__()
    self.var2 = 301
  def fun2(self):
    return 302

obj = Level2()
print(obj.Var0, obj.var0,
      obj.fun0())
print(obj.Var1, obj.var1,
      obj.fun1())
print(obj.Var2, obj.var2,
      obj.fun2())
```

example: `levels.py`

MRO

- Method Resolution Order
- Consider following inheritance, composition

```
class O:
  pass
class A(O):
  pass
class B(O):
  pass
class X(A,B):
  pass
class Y(B,A):
  pass
class Z(X,Y) #gives error
```

why?

In X: A comes before B,
in Y: B comes before A
In Z, can't determine order of A and B

example: `mro_error1.py`

MRO

- Following works fine

```
class O:
   pass
class A(O):
   pass
class B(O):
   pass
class X(A,B):
   pass
class Y(X,A):
   pass
```

\# example: `mro_ok.py`

- Following fails

```
class O:
   pass
class A(O):
   pass
class B(O):
   pass
class X(A,B):
   pass
class Y(A,X):
   pass
```

Fails because in definition of X,
X comes before A,
but in Y: A comes before X.

\# example: `mro_error2.py`

No Overloading Support

```
def product(a, b):
 res = a * b
 return res

def product(a, b, c):
 res = a * b *c
 return res
```

- The last definition overrides earlier definitions.
- Indirectly can be supported via class inheritance

- Example: `nooverload.py`

Polymorphism

```
def product(a, b, c=10)
  res = a * b *c
  return res

print(product(2,3))
print(product(2,3,4)
```

- Polymorphism support is via default value
  - assigned to parameters
- Class inhertance can define different function signatures.
  - However, object in hierarchy needs to pass correct number of arguments

Programming Exercises:

- Ex 01:
  - Define a class for conducting bitwise operations. The class should support the following.
    - Initialize number of bits (limited to 8, 16, 32 and 64)
    - `Reset()`: Reset all bits to zero.
    - `SetBit(n)`: Set the n^{th} bit to `1`.
    - `ChkBit(n)`: Check if n^{th} bit to `1`. Returns `True` or `False`
    - `Not()`: implement 1's complement on bits
    - `And(o)`: Perform bitwise AND operation with bits of object `o` and return the result.
    - `Or(o)`: Perform bitwise OR operation with bits of object `o` and return the result.

Programming Exercises:

- Ex 02:
  - Define a class for complex number arithmetic and perform following operations
    - Add another complex number to it
    - Subtract another complex number from it
    - Multiply it by another complex numbers
    - Divide it by another complex numbers
    - Conjugate this complex number
    - Compare this complex number with other
    - Return absolute value

Programming Exercises:

- Ex 03:
  - Define a class for representing fractions as rational number P/Q, Q!=0, and P and Q are relatively prime.
  - Define following operations
    - Add another rational number to it
    - Subtract another rational number from it
    - Multiply it by another rational numbers
    - Divide it by another rational numbers
    - Compare this rational number with other

Programming Exercises:

- Ex 04:
  - Using the rational number class as programmed in exercise 02, do the following,
  - Read a text file where each line contains two rational numbers with some mathematical operation
    - e.g. `5/6 + 3/4`
  - Read line and create a rational number for it.
  - Find all the rational numbers which occur more than once e.g.
    - `5/6 + 3/4` and `2/3 + 11/12` represent same rational number

Questions