Introduction to Computer Science:

python programming 3

April 2020

Honguk Woo

OOP (object oriented programming) in Python

- 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*
  - Modeling real world things e.g, cars, employees, companies
  - Using data and functions of an object

```
# many variables for two cookies

a = “cookie1‟
a_width = 3
a_height = 5
a_area = a_width*a_height
print("{0} area is {1}".format(a, a_area))

b = “cookie2‟
b_width = 4
b_height = 6
b_area = b_width*b_height
print("{0} area is {1}".format(b, b_area))
```

Need a type for cookies objects

class

- Class : type of objects, blueprint or prototype for the object

- Defining a class

```
class Animal:
    pass
```

Object

- An object (instance) is an instantiation of a class
  - When a class is defined, only the description for the object is defined; no memory or storage is allocated

```
mypet = Animal(“cutie”)
```

Class : define a new type of object, e.g., *Rectangle*

- defining a new **class** creates a new *type* of object, allowing new *instances* of that type to be made
- Recall that *everything is an object in python* (instance of class)

```
>>> a = 3
>>> type(a)
<class 'int'>
>>> a = "x"
>>> type(a)
<class 'str'>
>>> a = [1, 2, "x"]
>>> type(a)
<class 'list’>
>>> def f():
          print("hello world")
>>> type(f)
<class 'function'>
```

Class definition

- Class variable
- Method
- Instance variable

```
class Rect:
  c = 0

  def __init__(self, width, height):
    self.width = width
    self.height  = height
    Rect.c += 1

  def calcArea(self):
    area = self.width * self.height
    return area
```

Creating instance

- **__init__** is a special reseved method that is automatically called when memory is allocated for a new object

```
class Rect:
  c = 0

  def __init__(self, width, height):
    self.width = width
    self.height  = height
    Rect.c += 1

  def calcArea(self):
    area = self.width * self.height
    return area

#

a = Rect()
Traceback (most recent call last):
 File "python", line 1, in <module>
TypeError: __init__() missing 2 required positional
arguments: 'width' and 'height'

a = Rect (10, 20)
a.calcArea() => 200
```

Built-in class vs user-defined class

```
a = 3
print(a)

a = int(4)
print(a)
```

- **int** type (class):
  - 1, 2, 3 ... objects are instances of int type
  - a = 1

- **Rect** type (class)
  - a = Rec(10, 20) # *a* is object, an instance of Rect type

class variable, instance variable, method

- Class variable : shared by all instances
- Instance variable : unique to each instance
- Method : function

```
class Rect:
  c = 0

  def __init__(self, width, height):
    self.width = width
    self.height = height
    Rect.c += 1

  def calcArea(self):
    area = self.width * self.height
    return area
```

Class variable

method

Instance variable

method

class variable, instance variable, method

```
class Rect:
  c = 0

  def __init__(self, width, height):
    self.width = width
    self.height = height
    Rect.c += 1

  def calcArea(self):
    area = self.width * self.height
    return area
```

```
r1 = Rect(10, 20)   # class instantiation
r2 = Rect(20, 40)


print(r1.calcArea())  # method call
# ?
print(r2.calcArea())
# ?
print(r1.c)       # class variable shared by all
# ?
print(r2.c)
# ?
print(Rect.c)
# ?
```

class variable, instance variable, method

```
class Rect:
  c = 0

  def __init__(self, width, height):
    self.width = width
    self.height = height
    Rect.c += 1

  def calcArea(self):
    area = self.width * self.height
    return area
```

```
r1 = Rect(10, 20)   # class instantiation
r2 = Rect(20, 40)
r1.c = -1     # immutable int variable is bound to object -1
          # is it local or global ? Recall the local assignment in a function
print(r1.calcArea()) # method call
# ?
print(r2.calcArea())
# ?
print(r1.c)      # now, instance variable, not shared
# ?
print(r2.c)      # ?
# ?
print(Rect.c)     # class variable
# ?
```

OOP (object oriented programming) concept

- OOP can be characterized by :
  ① Abstraction
  ② Encapsulation
  ③ Inheritance
  ④ Polymorphism

- Python has built-in classes; int, str, tuple, list….
  - *What if we want to represent more complex data (user-defined data structure) ?*

OOP – abstraction, encapsulation

```
a = “cookie1“
a_width = 3
a_height = 5
a_area = a_width*a_height
print("{0} area is {1}".format(a, a_area))

b = “cookie2“
b_width = 4
b_height = 6
b_area = b_width*b_height
print("{0} area is {1}".format(b, b_area))
#############################################
#############################################

class Rect:
          c = 0
          def __init__(self, name, width, height):
                    self.name = name
                    self.width = width
                    self.height = height
                    Rect.c += 1

          def calcArea(self):
                    area = self.width * self.height
                    return area

r1 = Rect(“cookie1", 3, 5)
print(r1.calcArea())
r2 = Rect(“cookie2", 4, 6)
print(r2.calcArea())
```

Abstraction : hide details, keep important features (I would say “generalization”)

Encapsulation : use methods to integrate data and function; data can be accessed by methods

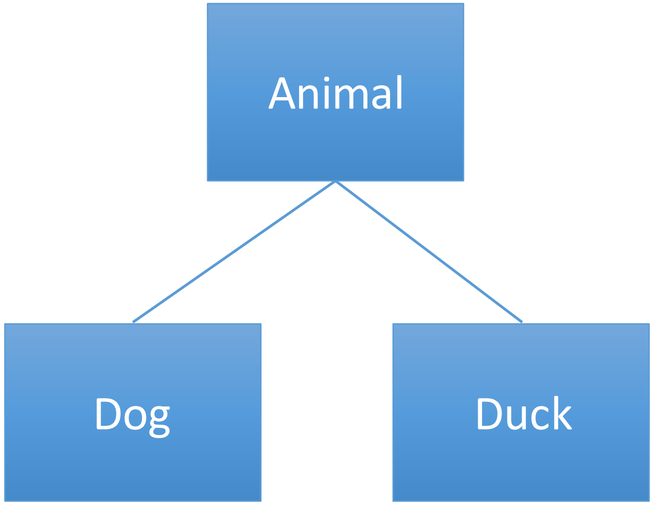
OOP - Inheritance

- Inheritance enables new classes to receive *(or inherit)* the data properties and methods of existing classes
    - Dag and Duck inherit some common properties (e.g., move) from Animal
        - Parent, base, super class
        - Child, derived, sub class

```
class Animal:
    def __init__(self, name):
        self.name = name
    def move(self):
        print("move")
    def speak(self):
        pass

class Dog (Animal):
    def speak(self):   #override
        print("bark")

class Duck (Animal):
    def speak(self):   #override
        print("quack")
```



OOP - Inheritance

```
class Animal:
    def __init__(self, name):
        self.name = name
    def move(self):
        print("move")
    def speak(self):
        pass

class Dog (Animal):
    def speak(self):
        print("bark")

class Duck (Animal):
    def speak(self):
        print("quack")
```

```
a = Animal("mydog")
b = Dog("mydog2")
c = Duck("myduck")
a.move()
?
b.move()
?
c.move()
?
b.speak()
?
c.speak()
?
a.speak()
?
```

OOP - Polymorphism

- Polymorphism enables to process objects differently depending on their types (classes)

```
class Animal:
    def __init__(self, name):
        self.name = name
    def move(self):
        print("move")
    def speak(self):
        pass

class Dog (Animal):
    def speak(self):
        print("bark")

class Duck (Animal):
    def speak(self):
        print("quack")
```

```
animals = [Dog('doggy'), Duck('duck'), Duck('duck2‘)]

for a in animals:
    a.speak()
```

Polymorphism :
same code but
differently executed

Discussion : what does *self* do?

- The self parameter is a reference to the current instance of the class
  - is used to access variables that belongs to the class

```
>>> Rect.calcArea()
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    Rect.calcArea()
TypeError: calcArea() missing 1 required
positional argument: 'self'
>>> Rect.calcArea(a)
```

```
class Rect:
  c = 0

  def __init__(self, width, height):
    self.width = width
    self.height  = height
    Rect.c += 1

  def calcArea(self):
    area = self.width * self.height
    return area

>>> a = Rect(10, 20)
>>> a.calcArea()
200
>>> Rect.calcArea()
?
>>> Rect.calcArea(a)
?
```

Discussion

- What’s the output?

```
class Rect:
  c = 0

  def __init__(self, width, height):
    self.width = width
    self.height  = height
    Rect.c += 1

  def calcArea(self):
    area = self.width * self.height
    return area

print(type(Rect.calcArea))

print(type(a.calcArea))
```

Discussion : string concat

- C programming
  - What’s the output ?

```c
#include <stdio.h>

int main() {
  char s1[20] = "first string";
  char s2[20] = "second string";
  char *s3 = s1 + s2;
  printf("%s", s3);
  return 0;
}
```

- Python programming

```python
s1 = "first string"
s2 = "second string"
s3 = s1 + s2
print(s3)
```

In C

```c
#include <stdio.h>

int main() {
  char s1[20] = "first string";
  char s2[20] = "second string";
  char *s3 = s1 + s2;
  printf("%s", s3);
  return 0;
}
```

```
hong@Ubuntu-V:~/myLec/2020/2020.1/cs$ gcc str.c
str.c: In function ‘main’:
str.c:7:17: error: invalid operands to binary + (have ‘char *’ and ‘char *’)
  char *s3 = s1 + s2;
```

```c
#include <stdio.h>
#include <string.h>

int main() {
  char s1[40] = "first string ";
  char s2[20] = "second string";
  strcat(s1, s2);
  printf("%s", s1);
  return 0;
}
```

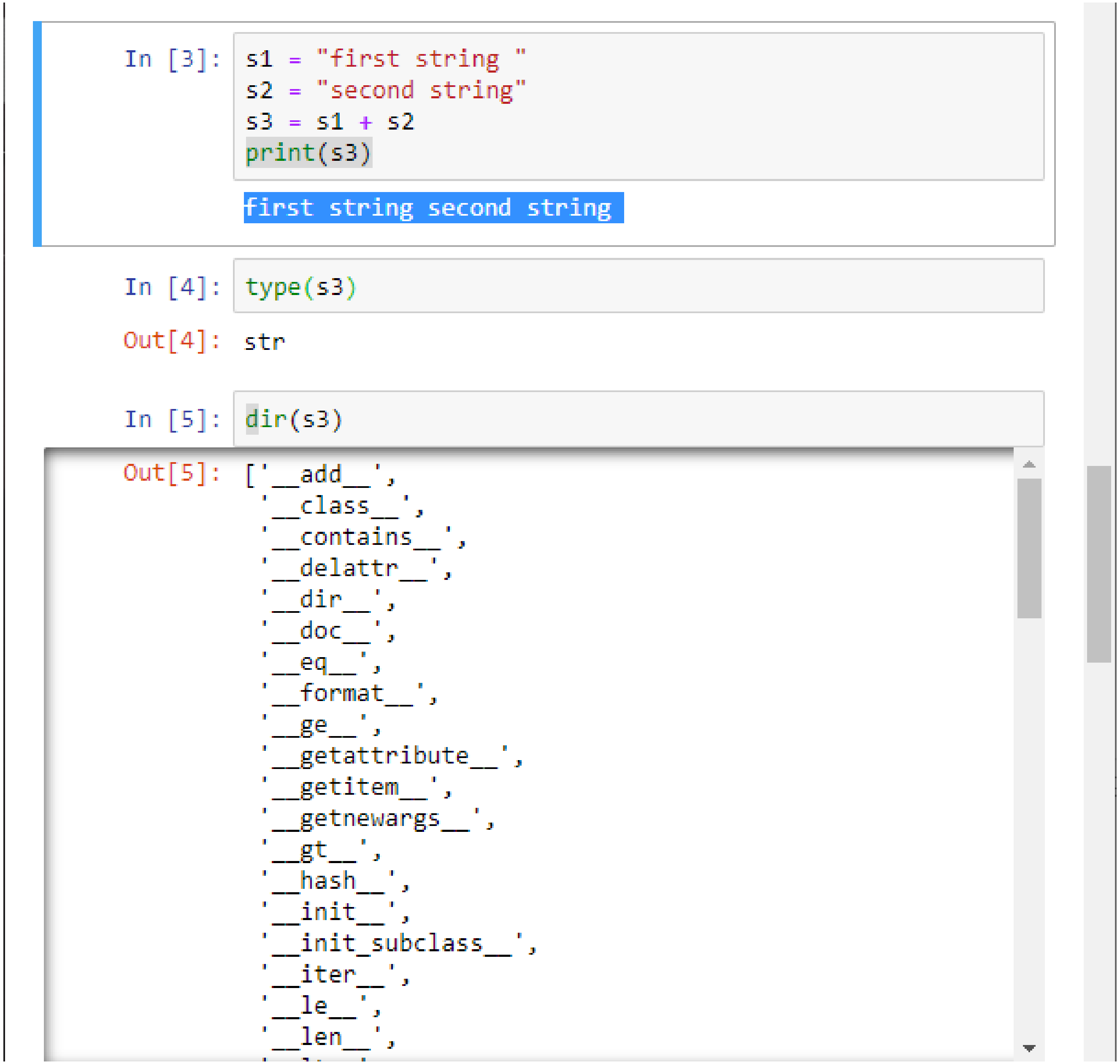
In python, operator overloading

```
>>> # Adds the two numbers
>>> 1 + 2
3

>>> # Concatenates the two strings
>>> 'Real' + 'Python'
'RealPython'


>>> # Gives the product
>>> 3 * 2
6

>>> # Repeats the string
>>> 'Python' * 3
'PythonPythonPython'
```



```
In [3]: s1 = "first string "
        s2 = "second string"
        s3 = s1 + s2
        print(s3)

        first string second string

In [4]: type(s3)

Out[4]: str

In [5]: dir(s3)

Out[5]: ['__add__',
         '__class__',
         '__contains__',
         '__delattr__',
         '__dir__',
         '__doc__',
         '__eq__',
         '__format__',
         '__ge__',
         '__getattribute__',
         '__getitem__',
         '__getnewargs__',
         '__gt__',
         '__hash__',
         '__init__',
         '__init_subclass__',
         '__iter__',
         '__le__',
         '__len__',
```

Quiz : *sharable & mutable, sharable & immutable*

- Try this code : e.g., tricks is class variable shared by instances

```
class Dog:
        tricks = []

        def __init__(self, name):
                self.name = name

        def add_trick(self, trick):
                self.tricks.append(trick)

>>> d = Dog('Fido’)
>>> e = Dog('Buddy’)
>>> d.add_trick('roll over’)
>>> e.add_trick('play dead’)
>>> print(d.tricks)
?
```