ITI 1120

Lab # 10

Objects

Starting Lab 10

- Open a browser and log into Blackboard Learn
- On the left hand side under Labs tab, find lab6 material contained in lab10-students.zip file
- Download that file to the Desktop and unzip it.

Most of the programming exercises in this lab are from your 3rd recommended textbook:

Ljubomir Perkovic. *Introduction to Computing Using Python: An Application Development Focus, 2nd Edition*

Before starting, always make sure you are running Python 3

This slide is applicable to all labs, exercises, assignments … etc

ALWAYS MAKE SURE FIRST that you are running Python 3.4 (3.5 is fine too)

That is, when you click on IDLE (or start python any other way) look at the first line that the Python shell displays. It should say Python 3.4 or 3.5 (and then some extra digits)

If you do not know how to do this, read the material provided with Lab 1. It explains it step by step

Do all the exercises labeled as Task in your head i.e. on a paper

Later on if you wish, you can type them into a computer (or copy/paste from the solutions once I poste them)

Task 1: “Reference” Variables and Objects

(a) What does the program below print? (Class Point is a simplified version of the class we designed in class). IMPORTANT: notice that the function riddle is outside of Point class (pay attention to what is lined up)

(b) http://www.pythontutor.com/visualize.html#mode

Open file t1.py and copy it in Python Vizualizer. Make sure you understand what is happening with variables as you step through the execution of the program.

```
class Point:
    def __init__(self, xcoord=0, ycoord=0):
        self.x = xcoord
        self.y = ycoord


def riddle(x, p):
    x=x+7
    return x + p.x + p.y

x = 5
blank = Point(1, 2)
t =riddle(x, blank)
print(x, t, blank.x, blank.y)
```

Task 2: "Reference" Variables and Objects

(a) What does the program below print? (Class Point is a simplified version of the class we designed in class)

(b) http://www.pythontutor.com/visualize.html#mode

Open file t2.py and copy it in Python Vizualizer. Make sure you understand what is happening with variables as you step through the execution of the program.

```
class Point:
    def __init__(self, xcoord=0, ycoord=0):
        self.x = xcoord
        self.y = ycoord


a = Point(-1, 1)
b = Point(3, 3)
a=b
a.x = 1

print(a.x, a.y, b.x, b.y)
```

Task 3: “Reference” Variables and Objects

```
class Point:
    def __init__(self, xcoord=0, ycoord=0):
        self.x = xcoord
        self.y = ycoord
    def __repr__(self):
        return 'Point('+str(self.x)+','+str(self.y)+')'

def increment_all(x,p):
    x=x+1
    p.x=p.x+1
    p.y=p.y+1

p1=Point(1,10)
y=1
print(y,p1)
increment_all(y,p1)
print(y, p1)
```

(a) What does the program on the left prints? The objects you design are MUTABLE.

(b) Open file t3.py and copy it in Python Vizualizer. Make sure you understand what is happening with variables as you step through the execution of the program.

Task 4: “Reference” Variables and Objects

```
class Point:
    def __init__(self, xcoord=0, ycoord=0):
        self.x = xcoord
        self.y = ycoord
    def __repr__(self):
        return 'Point('+str(self.x)+','+str(self.y)+')'

def riddle(a,b):
    a=b
    a.x=1000
    a.y=1000

p1=Point(1,2)
p2=Point(10,20)
print(p1,p2)
riddle(p1,p2)
print(p1, p2)
```

(a) What does the program on the left prints?

(b) Open file t3.py and copy it in Python Vizualizer. Make sure you understand what is happening with variables as you step through the execution of the program.

Class methods (REVIEW)

A class method is really a function defined in the class namespace; when Python executes

```
instance.method(arg1, arg2, …)
```

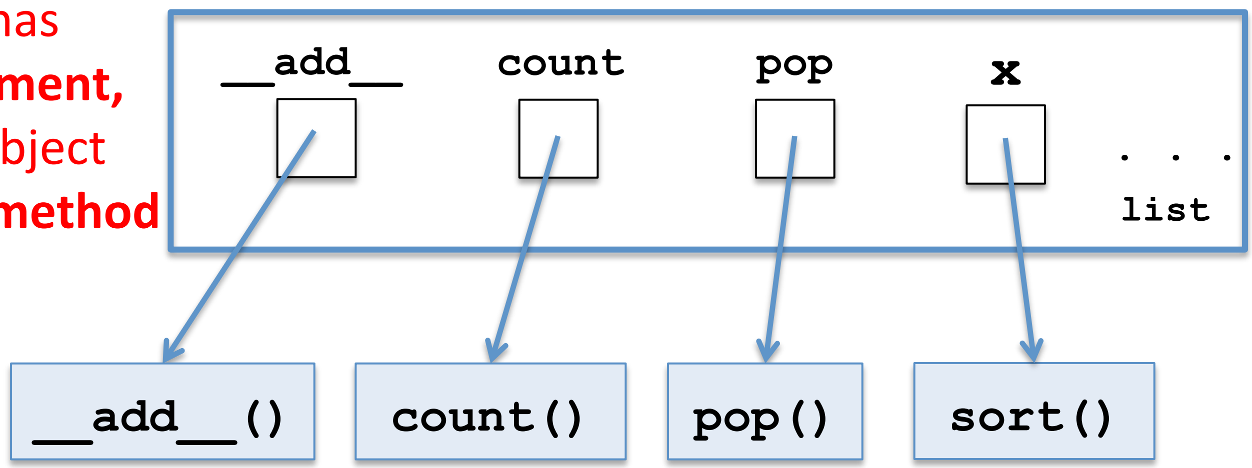
it first translates it to

```
class.method(self, arg1, arg2, …)
```

and actually executes this last statement

```
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> lst.sort()
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> list.sort(lst)
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst.append(6)
>>> lst
[1, 2, 3, 7, 8, 9, 6]
>>> list.append(lst, 5)
>>> lst
[1, 2, 3, 7, 8, 9, 6, 5]
```

The function has **an extra argument,** which is the object **invoking the method**



Task 5: Methods class translation to Function calls

- Open the file called t5.py and translate all the indicated method calls to equivalent function calls

- Once done run those function calls in Python shell to observe that they indeed are equivalent to their method call couterparts

Programming exercise 1:

Open file lab10.py It contains the classes we developed during lectures last week. Do the following exercises:

A) Add method `distance` to the class Point. It takes another Point object as input and returns the distance to that point (from the point invoking the method). (Recall that you need to import math to get sqrt function)

```
>>> c = Point(0,1)
>>> d = Point(1,0)
>>> c.distance(d)
1.4142135623730951
```

B) Add to class Point methods `up`, `down`, `left`, and `right` that move the Point object by 1 unit in the appropriate direction. The implementation of each should not modify instance variables x and y directly but rather indirectly by calling existing method move().

```
>>> a = Point(3, 4)
>>> a.left()
>>> a.get()
(2, 4)
```

C) In to class Animal modify the constructor to set age of the Animal object. Also add method `getAge` to retrieve the age of the Animal object.

```
>>> flipper = Animal('dolphin', '?', 3)
>>> flipper.getAge()
3
```

Python operators

In Python, all expressions involving operators are translated into method calls

```
>> '!'.__mul__(10)
'!!!!!!!!!!'
>>> [1,2,3].__eq__([2,3,4])
False
>>> int(2).__lt__(5)
True
>>> 'a'.__le__('a')
True
>>> [1,1,2,3,5,8].__len__()
6
```

```
>>> [1,2,3].__repr__()
'[1, 2, 3]'
>>> int(193).__repr__()
'193’
>>> set().__repr__()
'set()'
```

Operator	Method
x + y	x.__add__(y)
x - y	x.__sub__(y)
x * y	x.__mul__(y)
x / y	x.__truediv__(y)
x // y	x.__floordiv__(y)
x % y	x.__mod__(y)
x == y	x.__eq__(y)
x != y	x.__ne__(y)
x > y	x.__gt__(y)
x >= y	x.__ge__(y)
x < y	x.__lt__(y)
x <= y	x.__le__(y)
repr(x)	x.__repr__()
str(x)	x.__str__()
len(x)	x.__len__()
<type>(x)	<type>.__init__(x)

Overloading repr()

In Python, operators are translated into method calls

To add an overloaded operator to a user-defined class, the corresponding method must be implemented

To get this behavior

```
>>> a = Point(3, 4)
>>> a
Point(3, 4)
```

```
>>> a = Point(3, 4)
>>> a.__repr__()
Point(3, 4)
```

method **`__repr__()`** must be implemented and added to class `Point`

`__repr__()` should return the (canonical) string representation **of the point**

```
class Point:

    # other Point methods here

    def __repr__(self):
        'canonical string representation Point(x, y)'
        return 'Point({}, {})'.format(self.x, self.y)
```

Overloading operator +

To get this behavior

```
>>> a = Point(3,4)
>>> b = Point(1,2)
>>> a+b
Point(4, 6)
```

```
>>> a = Point(3,4)
>>> b = Point(1,2)
>>> a.__add__(b)
Point(4, 6)
```

method `__add__()` must be implemented and added to class `Point`

`__add__()` should return a new `Point` object whose coordinates are the sum of the coordinates of `a` and `b`

Also, method `__repr__()` should be implemented to achieve the desired display of the result in the shell

```
class Point:

    # other Point methods here

    def __add__(self, point):
        return Point(self.x+point.x, self.y+point.y)

    def __repr__(self):
        'canonical string representation Point(x, y)'
        return 'Point({}, {})'.format(self.x, self.y)
```

str() vs repr()

Built-in function __repr()__ returns the canonical string representation of an object

- **This is the representation printed by the shell when evaluating the object**
- **The string returned by __repr__ method must look like the statement that creates that object**

Built-in function __str()__ returns the **"pretty"** string representation of an object

- **This is the representation printed by the `print()` statement and is meant to be readable by humans**

```
>>> str([1,2,3])
'[1, 2, 3]'
>>> str(193)
'193'
>>> str(set())
'set()'
```

Operator	Method
`x + y`	`x.__add__(y)`
`x - y`	`x.__sub__(y)`
`x * y`	`x.__mul__(y)`
`x / y`	`x.__truediv__(y)`
`x // y`	`x.__floordiv__(y)`
`x % y`	`x.__mod__(y)`
`x == y`	`x.__eq__(y)`
`x != y`	`x.__ne__(y)`
`x > y`	`x.__gt__(y)`
`x >= y`	`x.__ge__(y)`
`x < y`	`x.__lt__(y)`
`x <= y`	`x.__le__(y)`
`repr(x)`	`x.__repr__()`
`str(x)`	`x.__str__()`
`len(x)`	`x.__len__()`
`<type>(x)`	`<type>.__init__(x)`

Programming exercise 2:

Open file lab10.py. It contains the classes we developed during lectures last week. Do the following exercises:

Overload appropriate operators for class Card so that you can compare cards based on rank. In particular override `__gt__` , `__ge__` , `__lt__` and `__le__`

```
>>> c1=Card('3','\u2660')
>>> c2=Card('5','\u2662')
>>> c1
Card(3,♠)
>>> c2
Card(5,♢)
>>> c1 < c2
True
>>> c1 > c2
False
>>> c1<=c2
True
```

Programming exercise 3: Bank Account

Develop a class BankAccount that supports these methods:

- `__init__`:Initializes the bank account balance to the value of the input argument, or to 0 if no input argument is given
- `withdraw`: Takes an amount as input and withdraws it from the balance
- `deposit`: Takes an amount as input and adds it to the balance
- `balance`: Returns the balance on the account
- `__repr__`:

```
>>> x = BankAccount(700)
>>> x.balance()
700.00
>>> x.withdraw(70)
>>> x.balance()
630.00
>>> x.deposit(7)
>>> x.balance()
637.00
>>> x
BankAccount(637.00)
```

Programming exercise 4: Ping Pong

Write a class named PingPong that has a method `next` that alternates between printing 'PING' and 'PONG' as shown below.

```
>>> ball = PingPong()
>>> ball.next()
PING
>>> ball.next()
PONG
>>> ball.next()
PING
>>> ball.next()
PONG
```

Programming exercise 5a: Class Queue

Goal: develop a class Queue , an ordered collection of objects that restricts insertions to the rear of the queue and removal from the front of the queue

- The class Queue should support methods:
  - `Queue()` :   Constructor that initializes the queue to an empty queue
  - `enqueue(item)` : Add item to the end of the queue
  - `dequeue()` : Remove and return the element at the front of the queue
  - `isEmpty()` : Returns True if the queue is empty, False otherwise

```
>>> appts = Queue()
>>> appts.enqueue('John')
>>> appts.enqueue('Annie')
>>> appts.enqueue('Sandy')
>>> appts.dequeue()
'John'
>>> appts.dequeue()
'Annie'
>>> appts.dequeue()
'Sandy'
>>> appts.isEmpty()
True
```

Class Queue: example


```
>>> appts = Queue()
>>> appts.enqueue('John')
>>> appts.enqueue('Annie')
>>> appts.enqueue('Sandy')
>>> appts.dequeue()
'John'
>>> appts.dequeue()
'Annie'
>>> appts.dequeue()
'Sandy'
>>> appts.isEmpty()
True
```

Programming exercise 5b: Class Queue

Make your class Queue user friendly by adding to it `__eq__`, `__repr__` and `__len__`

Example:

```
>>> q1=Queue()
>>> q1.enqueue('kiwi')
>>> q1.enqueue('apple')
>>>  print(q1)
>>> print(q1)
Queue(['kiwi', 'apple'])
>>> len(q1)
2
>>> q2=Queue()
>>> q2.enqueue('apple')
>>> q1==q2
False
>>> q1.dequeue()
'kiwi'
>>> q1==q2
True
```

Programming exercise (Inheritance) 6: Class Vector

Implement a class Vector that supports the same methods as the class Point we developed in class. In other words it inherits all atributes (data and methods) from class Point. (Revisit class Animal and Bird to see a simple example)

The class Vector should also support vector addition and product operations.

The addition of two vectors

```
>>> v1 = Vector(1, 3)
>>> v2 = Vector(-2, 4)
```

is a new vector whose coordinates are the sum of the corresponding coordinates of v1 and v2:

```
>>> v1 + v2
Vector(-1, 7)
```

The product of v1 and v2 is the sum of the products of the corresponding coordinates:

```
>>> v1 * v2
10
```

In order for a Vector object to be displayed as `Vector(., .)` instead of `Point(., .)`, you will need to override method `__repr__()`.

Programming exercise 7a: Class Marsupial

Write a class named Marsupial that can be used as shown below:

```
>>> m=Marsupial("red")
>>> m.put_in_pouch('doll')
>>> m.put_in_pouch('firetruck')
>>> m.put_in_pouch('kitten')
>>> m.pouch_contents()
['doll', 'firetruck', 'kitten']
>>> print(m)
I am a red Marsupial.
```

Programming exercise 7b (Inheritance) : Class Kangaroo

Write a class named Kangaroo as a subclass of Marsupial that inherits all the attributes of Marsupial and also:

- *extends* the Marsupial `__init__` constructor to take, as input, the coordinates x and y of the Kangaroo object,
- *has* method `jump` that takes number values dx and dy as input and moves the kangaroo by dx units along the x-axis and by dy units along the y-axis, and
- *overloads* the `__str__` operator so it behaves as shown below.

```
>>> k = Kangaroo("blue", 0,0)
>>> print(k)
I am a blue Kangaroo located at
coordinates (0,0)
>>> k.put_in_pouch('doll')
>>> k.put_in_pouch('firetruck')
>>> k.put_in_pouch('kitten')
>>> k.pouch_contents()
['doll', 'firetruck', 'kitten']
>>> k.jump(1,0)
>>> k.jump(1,0)
>>> k.jump(1,0)
>>> print(k)
I am a blue Kangaroo located at
coordinates (3,0)
```

Programming exercise 8 : Class Points

Write a class named Points (that represents points in the plane). The class has a list containing elements that are objects of type Point.

`__init__` constructor creates an empty list if no input list is given. Otherwise it sets the list to the given list.

- *has* method `add(x,y)` that adds an object Point with coordinates x and y to points
- has method `left_most_point` that returns the left most point in the set. If there is more than one left most it returns the bottom left most
- Has method `len` and overrides `__repr__`

```
>>> a=[Point(1,1), Point(1,2), Point(2,20), Point(1.5, -20)]
>>> mypoints=Points(a)
>>> mypoints.add(1,-1)
>>> mypoints.left_most_point()
Point(1,-1)
>>> len(mypoints)
5
>>> mypoints
Points([Point(1,1), Point(1,2), Point(2,20), Point(1.5,-20),Point(1,-1)])
```