INTRODUCTION TO PYTHON

LECTURE 4: Object Oriented Programming

Asem Elshimi

Announcements

Assignment 2 published tonight.

- Due on Feb 25th

Project proposal due on

- Due on Feb 25th

Project proposal

Specify team members (groups of 2 to 3)

Clear statement of goals of your project.

Initial statement of plan of execution.

"The Zen of Python" – Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

Outline

Programs

Object oriented programming

Functions

Data types

Conditionals

Loops

Input/output

Strings

Boolean

Arithmetic

Syntax basics

Everything is an object

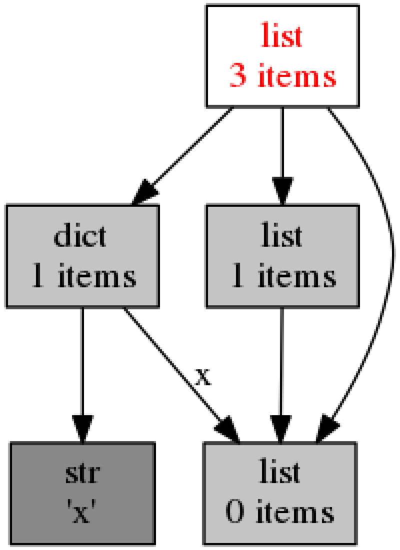
- can create new objects of some type
- can manipulate objects
- can destroy objects
  - explicitly using del or just “forget” about them
  - python system will reclaim destroyed or inaccessible objects – called “garbage collection”

Garbage collection

Examples, where the reference count increases:

- assignment operator
- argument passing
- appending an object to a list (object's reference count will be increased).

```
x = []
y = [x, [x], dict(x=x)]
```



https://rushter.com/blog/python-garbage-collector/

Objects

```
1234 3.14159 "Hello" [1, 5, 7, 11, 13]

{"CA": "California", "MA": "Massachusetts"}
```

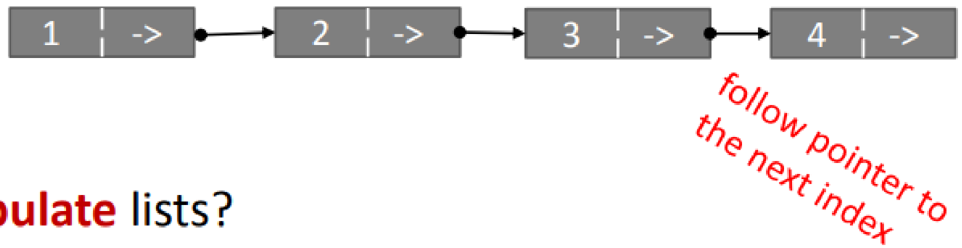
Each is an object, and every object has:
- a type
- data
- procedures

An object is an instance of a type:
- 1234 is an instance of an int
- "hello" is an instance of a string

EXAMPLE:
[1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells

L =



- how to **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

THE POWER OF OOP

- **bundle together objects** that share
  - common attributes and
  - procedures that operate on those attributes
- use **abstraction** to make a distinction between how to implement an object vs how to use the object
- build **layers** of object abstractions that inherit behaviors from other classes of objects
- create our **own classes of objects** on top of Python's basic classes

Define your own type

keyword Name/type Parent

```
class Coordinate(object):
    #define attributes here
```

- **Coordinate** is a subclass of `object`
- `object` is a superclass of **Coordinate**

Class attributes

Data attributes:

- Other objects contained within.

Procedural attributes:

- Methods
- Ex: distance()

__init__

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Data for initializing

Special method for initialization of instances

Parameter referring to the object being defined

Data attributes for coordinate objects

IMPLEMENTING THE CLASS vs USING THE CLASS

- write code from two different perspectives

implementing a new object type with a class

- **define** the class
- define **data attributes** (WHAT IS the object)
- define **methods** (HOW TO use the object)

using the new object type in code

- create **instances** of the object type
- do **operations** with them

Actually initializing an instance

```
c = Coordinate(3,4) #create a new object of type
#coordinate and pass 3,4 to its __init__

origin = Coordinate(0,0)


print(c.x)

print(origin.x)
```

```
__init__(self, x, y)
```

No need to pass self

Defining a method

```
class Coordinate(object):
    def __init__(self, x, y):
        .x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
#returns a number! not a coordinate object
```

Another argument

Accessing attribute using dot notation

Function vs method

method

Acts on objects of a class.

Takes self as an argument

Uses dot notation

function

Infers a type of objects

Functions can be defined in any scope

- In global scope, as we've seen in the past
- Inside other functions, as we've seen in the past
- Inside class objects

(take params, do operations, return)

Using method

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

Self object Other object

```
#==>5
```

```
print(Coordinate.distance(c, zero))
```

Print

```
c = Coordinate(3,4)
print(c)
```

<__main__.Coordinate object at 0x7fa918510488>

<3,4>

__str__

Special method

```
def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

Must return a string

Types

```
#can ask for the type of an object instance
c = Coordinate(3,4)
print(c) # runs __str__
#=> <3,4>
print(type(c))
#=> <class __main__.Coordinate>
print(Coordinate)
#=> <class __main__.Coordinate>
print(type(Coordinate))
#=> <type 'type'>
print(isinstance(c, Coordinate))
#=>True
```

SPECIAL OPERATORS

- +, -, ==, <, >, len(), print, and many others

https://docs.python.org/3/reference/datamodel.html#basic-customization

- like `print`, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)   →   self + other
__sub__(self, other)   →   self - other
__eq__(self, other)    →   self == other
__lt__(self, other)    →   self < other
__len__(self)          →   len(self)
__str__(self)          →   print self
```

... and others

str1+str2

CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

- class name is the **type**

  `class Coordinate(object)`

- class is defined generically
  - use `self` to refer to some instance while defining the class

    `(self.x – self.y)**2`
  - `self` is a parameter to methods in class definition

- class defines data and methods **common across all instances**

- instance is **one specific** object

  `coord = Coordinate(1,2)`

- data attribute values vary between instances

  `c1 = Coordinate(1,2)`

  `c2 = Coordinate(3,4)`
  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects

- instance has the **structure of the class**

OOP example: fractions

create a new type to represent a number as a fraction:

- internal representation is two integers
  - Numerator
  - Denominator
- interface a.k.a. methods a.k.a how to interact with fraction objects:
  - add, subtract
  - print representation, convert to a float.
  - invert the fraction.

```python
class Fraction(object):
    """
    A number represented as a fraction
    """
    def __init__(self, num, denom):
        """ num and denom are integers """
        assert type(num) == int and type(denom) == int, "ints not used"
        self.num = num
        self.denom = denom
    def __str__(self):
        """ Retunrs a string representation of self """
        return str(self.num) + "/" + str(self.denom)
    def __add__(self, other):
        """ Returns a new fraction representing the addition """
        top = self.num*other.denom + self.denom*other.num
        bott = self.denom*other.denom
        return Fraction(top, bott)
    def __sub__(self, other):
        """ Returns a new fraction representing the subtraction """
        top = self.num*other.denom - self.denom*other.num
        bott = self.denom*other.denom
        return Fraction(top, bott)
    def __float__(self):
        """ Returns a float value of the fraction """
        return self.num/self.denom
    def inverse(self):
        """ Returns a new fraction representing 1/self """
        return Fraction(self.denom, self.num)
```

`Fraction()` examples

```
a = Fraction(1,4)
b = Fraction(3,4)
c = a + b # c is a Fraction object
print(c)
print(float(c))
print(Fraction.__float__(c))
print(float(b.inverse()))
c = Fraction(3.14, 2.7)
print a*b
```

16/16

1.0

1.0

1.3333333333333333

Assertion error

error, did not define how to multiply two Fraction objects

intSet() code example

Getters and setters

getters

setters

```
class Animal(object):
       def __init__(self, age):
              self.age = age
              self.name = None
       def get_age(self):
              return self.age
       def get_name(self):
              return self.name
       def set_age(self, newage):
              self.age = newage
       def set_name(self, newname=""):
              self.name = newname
       def __str__(self):
              return"animal:"+str(self.name)+":"+str(self.age)
```

Information hiding

```
a = Animal(3)

a.age #not recommended

a.get_age()
```

Why not?

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

outside of class, use getters and setters instead:
good style, easy to maintain code, prevents bugs

Inheritance

child classes override *or* extend the functionality (e.g., attributes and procedures) of parent classes.

```
# Parent class
class Dog(object):
    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)
```

```python
# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

```
# Child classes inherit attributes and
# behaviors from the parent class
jim = Bulldog("Jim", 12)
print(jim.description())
```

Jim is 12 years old

```
# Child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))
```

Jim runs slowly

```python
# Is jim an instance of Dog()?
print(isinstance(jim, Dog))


# Is julie an instance of Dog()?
julie = Dog("Julie", 100)
print(isinstance(julie, Dog))


# Is johnny walker an instance of Bulldog()
johnnywalker = RussellTerrier("Johnny Walker", 4)
print(isinstance(johnnywalker, Bulldog))


# Is julie and instance of jim?
print(isinstance(julie, jim))
```

```
True
True
False
ERROR!
```

Building a **range** class

```
for i in range(3):
    print(i, sep=', ‘)          # => 0
                                     1
                                     2
```

Building a range class

```
def range(n):
    i = 0
    while i < n:
        yield i
        i += 1

for i in range(3):
    print(i)            # => 0
                              1
                              2
```

LBE: Building a range class

```python
class range:
    def __init__(self, n):
        self.n = n
        self.i = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            tmp = self.i
            self.i += 1
            return tmp
        else:
            raise StopIteration
```

```python
for i in range(3):
    print(i, sep=', ')    # => 0, 1, 2
```

Any generator can be written as a class

Generators are much more concise though!

This is how you notify
the caller that the iterator is expended

OBJECT ORIENTED PROGRAMMING

- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

Consider when to use OOP

Previous slides just contain buzzwords

- Buzzwords let you communicate with others

General rule of thumb: if it's a small project or only a few people are working on it, OOP may not be necessary

Good OOP is hard, bad OOP gets in the way

Classification of Circuit Components

When one thinks of circuit elements, a myriad of objects immediately comes to mind. These devices include wires, AND gates, transistors, input and output ports, RS-latches, HIGH and LOW signals and so on. For the purpose of this report, objects which are at the digital level and above will only be considered. Therefore, switch level devices such as transistors will not be considered.

One possible way to classify these numerous objects is to consider all the circuit entities at their highest level of abstraction and attempt to group objects which have similar properties under the same base class. From this, three very general classes are formed:

- `Component` -- All elements derived from this class process input signals and generate output signals to the objects to which they are connected. It is possible for one component to be *part of* another component. Circuit elements which can be considered as *kind of* components would include AND gates, RS-latches and random functional blocks.
- `Connector` -- All elements derived from this class would be responsible for connecting components with other components or with the external world. Each connector is *part of* a component at some level of abstraction. Since a connector can ``feed'' one or more components via fan-out, a list of components can be considered *part of* a connector. Some circuit elements which are *kind of* connectors include wires, and I/O ports.
- `Signals` -- Objects instantiated from this class are passed from component to component via the connectors. As will be shown later, a list of signals can be considered as *part of* a wire. This report will consider signals as two entities: a signal value (such as HIGH, LOW or X) and an associated unit of time. Since signals are used almost exclusively during the simulation of circuitry, a detailed analysis of this class will be postponed until the next chapter.

As alluded to above, linked list classes are required for components and signals. As will be shown in the next section, the need will also arise for a linked list class for I/O ports. Due to the current lack of parameterized types in the C++ language, some duplication of code is necessary to create the three linked list classes. Fortunately, the replication of code is relatively small.

---

http://www.cs.mun.ca/~donald/bsc/node17.html#SECTION00510000000000000000

Lab 4

More inheritance

Jupyter notebook (final report.)

Ungraded quiz.

Help with project/assignment.

Questions?

THANK YOU