CS303E: Elements of Computers and Programming

Objects and Classes

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

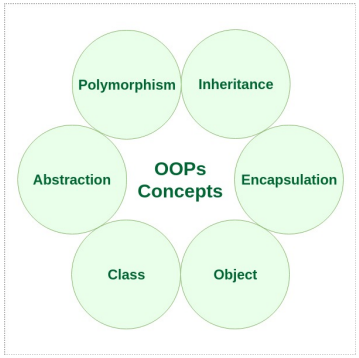
Last updated: April 12, 2021 at 09:15

Python and OO

Python is an *object-oriented* (OO) language. That implies a certain approach to thinking about problems.

Basic idea: conceptualize any problem in terms of a collection of "objects"—data structures consisting of data fields and methods together with their interactions.

Programming techniques may include: data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance. *We'll talk about some of these later.*



Object Orientation

The basic idea of object oriented programming (OOP) is to view your problem as a *collection of objects*, each of which has certain state and can perform certain actions.

Each object has:

- some *data* that it maintains characterizing its current state;
- a set of actions (*methods*) that it can perform.

A user interacts with an object by calling its methods; this is called *method invocation*. That should be the *only way* that a user interacts with an object.

Significant object-oriented languages include Python, Java, C++, C#, Perl, JavaScript, Objective C, and others.

OO Paradigm: An Example

Example: A soda machine has:

Data: products inside, change available, amount previously deposited, etc.

Methods: accept a coin, select a product, dispense a soda, provide change after purchase, return money deposited, etc.

Object Orientation

The programmer interacts with objects by invoking their methods, which may:

- update the state of the object,
- query the object about its current state,
- compute some function of the state and externally provided values,
- some combination of these.

Name potential instances of each of these for our Soda Machine example.

Another OO Example: A Simple Calculator

Imagine that you're trying to do some simple arithmetic. You need a Calculator application, programmed in an OO manner. It will have:

Some data: the current value of its *accumulator* (the value stored and displayed on the screen).

Some methods: things that you can ask it to do: add a number to the accumulator, subtract a number, multiply by a number, divide by a number, zero out the accumulator value, etc.

Calculator Specification

In Python, you implement a particular type of object (soda machine, calculator, etc.) with a `class`.

Let's define a class for our simple interactive calculator.

Data: the current value of the accumulator.

Methods: any of the following.

clear: zero the accumulator

print: display the accumulator value

add k: add k to the accumulator

sub k: subtract k from the accumulator

mult k: multiply accumulator by k

div k: divide accumulator by k

A Calculator Class

Below is a (partial) Python implementation of the Calculator class:

In file `Calc.py`:

```
class Calc:
    """This is a simple calculator class. It stores and
    displays a single number in the accumulator.  To that
    number, you can add, subtract, multiply or divide."""

    def __init__(self):
        """Constructor for new Calc objects,
           with display 0."""
        self.__accumulator = 0

    def __str__(self):
        """Allows print to display accumulator value
           in a nice string format."""
        return "Displaying: " + str(self.__accumulator)

    def getAccumulator(self):
        return self.__accumulator
```

Definition of class `Calc` continues on the next slide.

A Calculator Class

Continuation of the `Calc` class:

```
    def clear(self):
        self.__accumulator = 0

    def add(self, num):
        self.__accumulator += num

    def sub(self, num):
        ...

    def mult(self, num):
        ...

    def div(self, num):
        ...
```


Using Our Class

```
>>> from Calc import *    # import from Calc.py
>>> c = Calc()            # create a calculator object
>>> print( c )            # show its current value
Displaying: 0
>>> c.add( 10 )           # add 10
>>> print( c )
Displaying: 10
>>> c.div( 0 )            # try to divide by 0
Error: division by 0 not allowed.
>>> c.div( 2 )            # divide by 2
>>> print( c )
Displaying: 5.0
>>> c.mult( 4 )           # multiply by 4
>>> print( c )
Displaying: 20.0
>>> c.clear()             # clear the state
>>> print( c )
Displaying: 0
```


Let's Take a Break

Defining Classes

General Form:

```
class ClassName:
    initializer
    methods
```

This defines a new class (type), which you can *instantiate* to create as many objects (instances) as you like.

The Circle Class

In file `Circle.py`:

```
import math

class Circle:
    def __init__(self, rad = 1):
        """ Construct a Circle object with radius
            rad (defaults to 1). """
        self.radius = rad

    def getRadius(self):          # getter
        return self.radius

    def setRadius(self, rad):     # setter
        self.radius = rad

    def getPerimeter(self):
        return 2 * math.pi * self.radius

    def getArea(self):
        return math.pi * ( self.radius ** 2 )
```


Using the Circle Class

```
>>> from Circle import *
>>> c1 = Circle()          # create a new Circle, radius 1
>>> c1.getRadius()
1
>>> c1.setRadius(5)        # reset c1's radius to 5
>>> c1.getRadius()
5
>>> c1.getArea()           # compute its area
78.53981633974483
>>> c1.getPerimeter()      # compute its perimeter
31.41592653589793
>>> c2 = Circle(10)        # make a new Circle, radius 10
>>> c2.getArea()           # get its area
314.1592653589793
```


Creating a New Object

Use the class name to create a new object of that class.

```
class Circle:

    def __init__(self, rad = 1):
        """ Construct a Circle object with radius
            rad (defaults to 1). """
        self.radius = rad

    ...
```

```
>>> c1 = Circle()
>>> c2 = Circle( 5 )
```

The function `__init__` is automatically called to initialize the object and define its *data members*.

Creating a New Object

```
class Circle:
    def __init__(self, rad = 1):
        """ Construct a Circle object with radius
            rad (defaults to 1). """
        self.radius = rad

    ...
```

Notice that `__init__` has two parameters:

- self : refers to the object just created. It is used within the class definition, but not outside it.
- rad : it wouldn't make any sense to define a circle without a radius. It's a **data member** of the class.

The Other Methods

```
    ...
    def getRadius(self):
        # Return the radius
        return self.radius

    def getPerimeter(self):
        # Compute the perimeter
        return 2 * math.pi * self.radius
    ...
```

The other methods can refer to the class data members using the dot notation.

They have `self` as a parameter at definition. When they are called on a class instance (object), self is an *implicit parameter* referring to the object itself.

```
>>> c1.getRadius()                # self references c1
5
>>> c1.getPerimeter()
31.41592653589793
```


Accessing Data Members

It is (sometimes) possible to directly access the data members of a class:

```
c1 = Circle()
>>> c1.radius           # bad practice
1
>>> c1.getRadius()      # better
1
```

But it's a bad idea, for two reasons:

1. Anyone can tamper with your class data, including setting it to illegal values.
2. The class becomes difficult to maintain. Suppose some user sets the Circle radius to a negative value.

Accessing Data Members

Better to deny direct access to data members; instead define *setters* (or mutators) and *getters* (or accessors).

```
    def getRadius(self):           # getter
        return self.radius

    def setRadius(self, radius):   # setter
        self.radius = radius
```

Even with setters and getters, there's nothing to prevent code from accessing data members directly, unless you make the data member *private*.

A data member beginning with two underscores is private to the class.

The Circle Class with Private Data

```
import math

class Circle:
    # Construct a circle object, with radius
    # a private data member.
    def __init__(self, rad = 1):
        self.__radius = rad

    def getRadius(self):
        return self.__radius

    def setRadius(self, rad):
        self.__radius = rad

    def getPerimeter(self):
        return 2 * math.pi * self.__radius

    def getArea(self):
        return math.pi * ( self.__radius ** 2 )
```

The only access to `__radius` outside the class is via the getter and setter methods.

Private Data Members

```
>>> from Circle import *
>>> c = Circle( 10 )
>>> c.getRadius()
10
>>> c.__radius      # violates privacy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Circle' object has no
   attribute '__radius'
>>> print( c )      # didn't define __str__
<Circle.Circle object at 0x7ff32a47e470>
```


Let's Take a Break

Everything's an Object

In Python, everything is an object, even numbers and strings. Every object has a unique id, accessed with the function `id()`.

You can access the class of any object with the function `type()`.

```
>>> from Circle import *
>>> c1 = Circle()
>>> type(c1)
<class 'Circle.Circle'>
>>> id(c1)
140162312889400
>>> type(7)
<class 'int'>
>>> id(7)
10914688
>>> type("xyz")
<class 'str'>
>>> id("xyz")
140162312889488
>>> id(4 + 1)
10914624
>>> id(5)
10914624
```


Printing a Class

If you want to print a class instance, you need to tell Python how to print it. Do that by defining a class method `__str__` that returns a `str`.

```
class Rectangle:
   def __init__(self, width = 2, height = 1):
       self.__width = width
       self.__height = height

   def __str__(self):
      return "Rectangle with width " + str(self.__width) + \
          " and height " + str(self.__height)
```

```
>>> from Rectangle import *
>>> r1 = Rectangle()
>>> print( r1 )
Rectangle with width 2 and height 1
>>> r2 = Rectangle( 3, 5 )
>>> print( r2 )
Rectangle with width 3 and height 5
```

`print` knows to call the `__str__` function on each object.

Mutable vs. Immutable

Remember that integers and strings are *immutable* meaning that you can't change them.
Classes you define are mutable. For an immutable object, there is only one copy, which is why you can't change it.

```
>>> from Circle import *
>>> x = 7
>>> id(x)
10914688
>>> y = 7
>>> id(y)
10914688
>>> c1 = Circle()
>>> c2 = Circle()
>>> id(c1)
140497298719856
>>> id(c2)
140497298720920
>>> x is y              # are x, y the same object
True
>>> c1 is c2            # are c1, c2 the same object
False
```


Putting It All Together

Suppose you want to write a Python program to play Poker. What is the *object oriented* way of thinking about this problem?

First question: What are the *objects* involved in a game of Poker?

Putting It All Together

Suppose you want to write a Python program to play Poker. What is the *object oriented* way of thinking about this problem?

First question: What are the *objects* involved in a game of Poker?

- Card (rank and suit)
- Deck of Cards (an ordered collection of cards)
- Hand (a collection of 5 cards dealt from a Deck)
- Player (an entity that makes decisions about its hand)
- Table (several Players competing against each other)

There are probably other ways to conceptualize this problem. It's good practice to put each class into its own file.

Designing a Class: Card

Let's start at the bottom. Suppose we want to design a representation in Python of a playing Card.

- What data is associated with a Card?
- What actions are associated with a Card?

Designing a Class: Card

Let's start at the bottom. Suppose we want to design a representation in Python of a playing Card.

- What data is associated with a Card?
- What actions are associated with a Card?

Data:

- Rank: ["Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"]
- Suit: ['Spades', 'Diamonds', 'Hearts', 'Clubs']

Methods:

- Tell me your rank.
- Tell me your suit.
- How would you like to be printed?

Designing a Class

We'll define a `Card` class with those attributes and methods.

Notice that there are:

- a *class* definition (defines the type of an arbitrary playing card),
- *instances* of that class (particular cards).

Card Class

In file/module Card.py

```
class Card:
    """A card object with a suit and rank."""
    # These are class attributes, not instance attributes
    RANKS = ["Ace", "2", "3", "4", "5", "6", "7", "8", \
             "9", "10", "Jack", "Queen", "King"]
    SUITS = ['Spades', 'Diamonds', 'Hearts', 'Clubs']

    def __init__(self, rank, suit):
        """Create a Card object with the given rank
        and suit."""
        if (    not rank in Card.RANKS \
             or not suit in Card.SUITS ):
            print ("Not a legal card specification.")
            return
        self.__rank = rank
        self.__suit = suit

    def getRank(self):
        return self.__rank

    def getSuit(self):
        return self.__suit
```


Poker: Card Class

```
    # This is the continuation of the Card class.

    def __str__(self):
        """Return a string that is the print representation
        of this Card's value."""
        return self.__rank + ' of ' + self.__suit
```

This tells `print` what string to display if you ask to print a Card object.

Poker: Card Class

```
>>> from Card import *
>>> print (Card.RANKS)
['Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10',
  'Jack', 'Queen', 'King']
>>> print (Card.SUITS)
['Spades', 'Diamonds', 'Hearts', 'Clubs']
>>> c1 = Card('2', 'Spades')
>>> c1.getRank()
'2'
>>> c1.getSuit()
'Spades'
>>> c1
<Card.Card object at 0xb763d4ec>
>>> print(c1)
2 of Spades
>>> c2 = Card('Queen', 'Hearts')
>>> print(c2)
Queen of Hearts
>>> (c1 < c2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Card() < Card()
```


Something Cool

Suppose we add the following functions ourside our `Card` class.

```
def cardRankToIndex (rank):
    # Converts a str rank to an index into RANKS
    return Card.RANKS.index( rank )
```

And add the following method to our `Card` class:

```
    def __lt__(self, other):
        return (   cardRankToIndex( self.__rank )  \
                 < cardRankToIndex( other.getRank() ))
```

This assumes that `other` is another Card object; if we're being very careful, we could check that in our code.

Something Cool

Now we can compare two cards using a convenient notation:

```
>>> from Card import *
>>> c1 = Card('2', 'Spades')
>>> c2 = Card('5', 'Diamonds')
>>> c1 < c2
True
>>> c2 < c1
False
>>> c1 > c2
False
```

Notice that we're comparing cards only according to rank, and Ace is less than 2. Think how you'd define a more robust test.

Comparing Class Instances

You can use all of the standard relational operators assuming you have defined `__lt__` and `__le__` so Python can figure out what you mean. You can always do equality comparison $X == Y$, which will be structural equality unless you define `__eq__`.

You can also define `__gt__` and `__ge__` but be careful that your definitions form a consistent collection.

You *shouldn't* define all of those functions, just enough to get it to work. That is, if you have `__lt__`, you don't need `__ge__` because that's just the negation.

Aside: Equality Comparisons

`(X == Y)` tests for structural equivalence of values. `(X is Y)` tests whether two objects are in fact the same object. Sometimes those are not the same thing

```
>>> x = [1, 2, 3]
>>> y = x
>>> z = [1, 2, 3]
>>> x == y
True
>>> x is y
True
>>> x == z
True
>>> x is z
False
```


Abstraction

Notice that we defined the Card class abstractly. There's nothing about it that indicates we're going to be playing Poker. *That's why it's good to start at the bottom!*

It would work as well for blackjack or canasta. It wouldn't work for Uno or another game using a specialty deck. What would you do for such cases?

Now the *interface* to the Card class is the methods: getSuit(), getRank(), print, and the relational comparisons. *Any other way of manipulating a Card object "violates the abstraction."*

Aside: Those Funny Names

In general, any method name in Python of the form `__xyz__` is probably not intended to be called directly. These are called "magic methods" and have associated functional syntax ("syntactic sugar"):

`__init__`	`ClassName()`
`__len__`	`len()`
`__str__`	`str()`
`__lt__`	`<`
`__eq__`	`==`
`__add__`	`+`

However, you often can call them directly if you want.

```
>> "abc".__add__("def")
'abcdef'
>> l = [1, 2, 3, 4, 5]
>>> len(l)
5
>>> l.__len__()
5
```