ENGR 298: Engineering Analysis and Decision Making - Classes

Dr. Jason Forsyth
Department of Engineering
James Madison University

On to More Complex Storage

- So far we have stored information as single variables (int, string, float..etc) or in simple containers (lists). These basic structures break down when the *object* we are modeling becomes more complicated.

- Consider you work for a bank and which to automate operations related to banking and bank accounts. Would need to track: user name, location, balance, transaction history….

- Not only does a bank account store information, we may wish to perform *operations* on that account (deposit() withdraw()) that modifies its internal data.

A program is a set of instructions for **manipulating**, **moving/storing**, and **making decisions** on **data** in your computer.

Classes: Or How I Learned to Love Object-Oriented Programming

- Every piece of data in a program is an **object**. All objects are unique **instances** of a particular **class**.

- A class is an organized collection of information (**attributes**) and functions (**methods**). These items are grouped together for logical/organization reasons to ensure common types and operations “stay together” and are in correct scope.

- All modern programming languages directly support classes. A programming paradigm that heavily utilizes classes is called “object-oriented programming” (OOP)

Interfacing with Classes – Part 1

- A class is invoked/created by calling its **Constructor**. This may be done explicitly or part of the built-in language. For each *instance* of a class, various class *methods* can be called.

```
# directly declare a list
declared_list = [1, 2, 3, 4]

# call the list constructor to create the list
constructed_list = list()

# use the list class method extend() to copy the lists
constructed_list.extend(declared_list)

if declared_list==constructed_list:
    print("Lists are the same!")
else:
    print("Lists are not the same!")
```

```
Lists are the same!
```

Interfacing with Classes – Part 2

- All interfacing with an object utilizes the . operator. This can call *methods* or access data/*attributes*. Consider a simple class Point that represents a 2D point.

```
# set (1,1) for A
a = Point()
a.x = 1
a.y = 1
print("Point A contains",a.x,a.y)
```

```
Point A contains 1 1
```

Interfacing with Classes – Part 3

- For existing classes, all the methods and fields should be well documented. Your interaction should only be through the constructor(), and the various attributes/methods of the class.

isprime(p: int) True if {p} is prime.

```
primality.isprime(13)
>> True
primality.isprime(20)
>> False
primality.isprime(5163490735091213 11)
>> True
```

nthprime(nth: int) Returns the {nth} prime, starting from n = 0, returning 2.

```
primality.nthprime(0)
>> 2
primality.nthprime(100)
>> 547
primality.nthprime(9999)
>> 104729
```

- For a well-designed class, you should not care what happens “inside” the class.

Creating Classes

- When creating your own classes, you greatly care what happens “inside” the class. All attributes and methods should be safe/logical such that someone utilizing the class “outside” has a clean interface.

- Class should:
  - Have sufficient complexity to represent/model an object but no more
  - Have reasonable attributes and methods to allow “readable” code
  - Generally, be independent for each instance to contain/operate on their own data
  - Improve code maintenance by encapsulating common data and functions.

Creating Your First Class

- The class keyword indicates to the program that a class is being defined. All attributes and methods are tabbed within the class scope

```
# create a basic class to represent points
# yes, we can do this in the middle of the program because this is Python
class Point:
    # give the class some default values
    x = 0
    y = 0

    # my stuff list
    my_stuff = []
```

- Generates a class called Point with three attributes. X and Y (with default values of 0) and my_stuff (an empty list).

- **Important**: Are x, y, and my_stuff shared by all instances of Point or unique to each one?

What do you expect the output of this code to be?

```
# Instantiate a new Point A and set values to (1,1)
a = Point()
a.x = 1
a.y = 1
print("Point A contains", a.x, a.y)

# now make B. See what it's default value is
b = Point()
print("Point B contains", b.x, b.y)
```

Ok, seems reasonable…they each have their own x and y values?

```
# Instantiate a new Point A and set values to (1,1)
a = Point()
a.x = 1
a.y = 1
print("Point A contains", a.x, a.y)

# now make B. See what it's default value is
b = Point()
print("Point B contains", b.x, b.y)
```

```
Point A contains 1 1
Point B contains 0 0
```

What about adding ‘stuff’ to that list…

```
# give A some objects, should stay with A
a.my_stuff.append("10 bitcoins")
print("A has",a.my_stuff)

# B shouldn't have anything, right?
print("B has",b.my_stuff)
```

Wait, they both have ‘10 bitcoins’? I thought each class had its own attributes??

```
# give A some objects, should stay with A
a.my_stuff.append("10 bitcoins")
print("A has",a.my_stuff)

# B shouldn't have anything, right?
print("B has",b.my_stuff)
```

```
A has ['10 bitcoins']
B has ['10 bitcoins']
```

Scope and Classes in Python

- Variable ‘scope’ is complex and dynamic within Python. Other languages (C++ and Java) are much more explicit about variables are available to classes, methods..etc.

- A **static** attribute is shared by all instances of a class. All attributes are static unless assigned to *self* within the constructor.


```
class Point:
    # give the class some default values
    x = 0
    y = 0

    # my stuff list
    my_stuff = []
```

Since “my_stuff” is a static variable, that list is shared amongst all instances of Point.

```
# give A some objects, should stay with A
a.my_stuff.append("10 bitcoins")
print("A has",a.my_stuff)

# B shouldn't have anything, right?
print("B has",b.my_stuff)
```

```
A has ['10 bitcoins']
B has ['10 bitcoins']
```

Then why did this example work…? X and Y weren't shared.

```
# Instantiate a new Point A and set values to (1,1)
a = Point()
a.x = 1
a.y = 1
print("Point A contains", a.x, a.y)

# now make B. See what it's default value is
b = Point()
print("Point B contains", b.x, b.y)
```

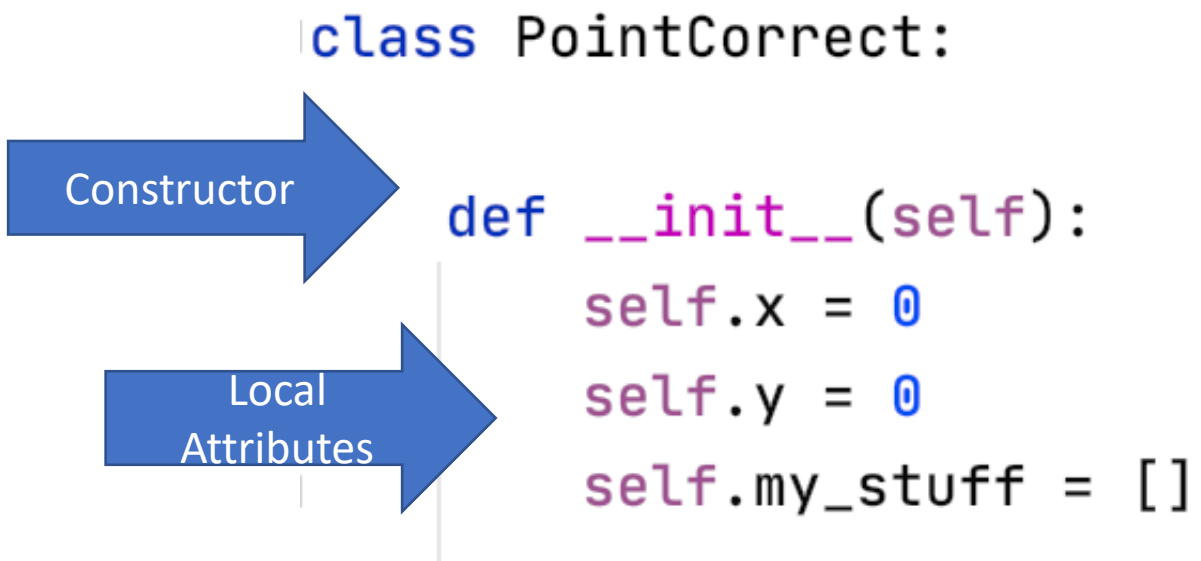
```
Point A contains 1 1
Point B contains 0 0
```

This is Python making subtle distinctions between 'basic' (int, float, str) and 'complex' types (lists). We will avoid this entirely as it is very confusing.

A Correct Implementation for Point()

- To avoid confusion of scope and namespaces, will generally avoid static attributes and declare them locally.

- Will utilize the keyword **self** to bind attributes uniquely to an object instance within the constructor **__init__**



This should fix things…

```
# Instantiate a new Point A and set values to (1,1)
a = PointCorrect()
a.x = 1
a.y = 1
print("Point A contains", a.x, a.y)

# now make B. See what it's default value is
b = PointCorrect()
print("Point B contains", b.x, b.y)

# give A some objects, should stay with A
a.my_stuff.append("10 bitcoins")
print("A has", a.my_stuff)

# B shouldn't have anything, right?
print("B has", b.my_stuff)
```

```
Point A contains 1 1
Point B contains 0 0
A has ['10 bitcoins']
B has []
```

__init__ Constructor – Part 1

- All Python classes have a built-in constructor method __init__
- This method is called whenever a new class object is instantiated. Should never be called directly.
- Takes “self” as the first parameter but this is automatically inserted by Python. You never need to add “self” to any class method calls.

__init__ Constructor – Part 2

- The constructor may behave differently based upon what it passed as argument. Both codes below result in equivalent lists.

```
# directly declare a list
declared_list = [1, 2, 3, 4]

# call the list constructor to create the list
constructed_list = list()

# use the list class method extend() to copy the lists
constructed_list.extend(declared_list)
```

Create an empty list and then add new elements

```
# directly declare a list
declared_list = [1, 2, 3, 4]

# call the list constructor to create the list
constructed_list = list(declared_list)
```

Create a new list with the copy constructor

__init__ Constructor – Part 3

- Would be useful to initialize a point to a single value at the start. Will need to modify __init__ to accommodate multiple parameters. I utilize _<name> to indicate parameters versus attributes with similar names.

```
# An implementation of point class that has two constructors
class PointClassConstructed:

    # advanced constructor with initial parameters
    def __init__(self, _x, _y):
        self.x = _x
        self.y = _y
        self.my_stuff = []
```

__init__ Constructor – Part 3

- But now, this code will not work… cannot match constructor calls since the __init__ signature has changed.

```
# now make B. See what it's default value is
b = PointClassConstructed()
print("Point B contains", b.x, b.y)
```

```
Traceback (most recent call last):
  File "/Users/jforsyth/Documents/GitHub/ENGR298-2022-Private/lecture-examples/week4-classes.py", line 87, in <module>
    b = PointClassConstructed()
TypeError: __init__() missing 2 required positional arguments: '_x' and '_y'
```

Two potential solutions: (1) change the code to explicitly provide value or (2) provide default values for x and y in the constructor

__init__ Constructor – Part 4 – Default Values

- Can resolve the previous error by providing **default values** for constructor parameters _x and _y

```
# An implementation of point class that has two constructors
class PointClassConstructed:

    # advanced constructor with initial parameters and
    # now with default values
    def __init__(self, _x=0, _y=0):
        self.x = _x
        self.y = _y
        self.my_stuff = []
```

Other Class Methods

- A class may have many methods/functions attached to it. Each of these functions can access the internal/local values that were setup in the constructor.

- Basic methods will set() or get() internal values. Others may perform some modification on data or compare against other values.

- Following examples will consider a Vector class similar to the Point

```python
class Vector:
    """
    A simple class to hold a 3D points
    """

    def __init__(self, _x=0, _y=0, _z=0):
        """
        Constructor a vector with (x,y,z) values. (0,0,0) unless specified.

        :param _x: Initial x value is 0 unless specified
        :param _y: Initial y value is 0 unless specified
        :param _z: Initial z value is 0 unless specified
        """
        self.x = _x
        self.y = _y
        self.z = _z
```

Implementation of dot product and distance as a function of the object/class

```
def dot(self, v):
    """
    Return the dot product of this object and the passed vector v1
    :param v: Another vector in three-space
    :return: Dot product of two vectors
    """

    return self.x * v.x + self.y * v.y + self.z * v.z
```

```
# use class methods to compute the product a * b.
dot_product = a.dot(b)
```

Inside the method, self will refer to A and v will refer to B.

Summary

- Classes provide a mechanism to organize data and functions into a single object.

- Many complex rules/situations regarding scope and sharing information across classes. Much larger topics include inheritance, operator overloading…etc.

- Will mainly use/call classes rather than create our own. If we do create classes, then they will be simpler objects.

Weekly Assignments. Use templates on GitHub.

- **Vector Attributes**: Complete the program operating on the Vector class to compute distance and dot product on various 3D objects. Directly access the Vector attributes using the **.** operator.

- **Vector Methods**: Complete the program operating on the Vector class to compute distance and dot product on various 3D objects. Use the class methods provided with Vector to complete the operations.

- **Counter Class**: Complete the implementation for the Class Counter. A test program will run through various operations to check the implementation of the constructor Counter() and various methods inc() and dec(). A counter is a simple object that is incremented (+1) or decremented (-1) once at a time.