# Topic: Classes and Object Oriented Programming

*Goals: By the end of this topic, we will discuss…*

- *Attributes and Methods*
- *Object Oriented Programming*
- *Inheritance*

Acknowledgements: These class notes build on the content of my previous courses as well as the work of R. Jordan Crouser, and Jeffrey S. Castrucci.

---

## From the Archives...

The first Smith College campus computer appeared in 1967.

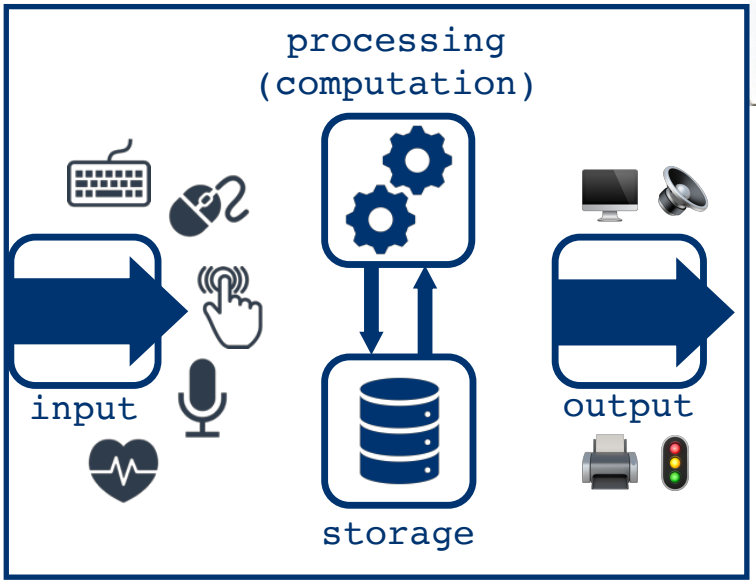
(see picture to right)

From: Nanci A. Young, College Archivist


---

## Recall: 4 Elements of a System

Same in 1967 as now...



## Recall: Lists/Dictionaries

In the following example, we created a list of dictionaries (called library to store our books) and passed it into each function.

```
def addBook(library):
    # Initialize an empty Book
    book = {}

    # Populate with user input
    book["title"] = input("Book title: ")
    book["author"] = input("author: ")

    # Append book to library
    library.append(book)

def printBooks(library):
    counter = 0
    for book in library:
        counter += 1
        print(str(counter) + ". '" + book["title"] + "' by",
              book["author"])

def removeBook(library, index):
    print("Removing book #" + str(index) +"...", end = "")
    library.pop(index - 1)

def main():
     library = []
     ... #calling code here.
```

*This feels odd to always be passing library.... is there a better way??*

Discussion: Compare this with other operations we can perform on lists and dictionaries; what do you notice?

```
animals.append('guinea pig')
vowels.insert(3, 'o')
animals.remove('rabbit')
pets.count('dog')
```

## Recall: Playlist (from Lab 5)

- We'd like to be able to ask a Book to print() or read() itself.
- That way we don't have to waste time passing everything from function to function.
- To do this, we'll need a way to combine functions (methods) and variables (attributes).

Solution: Classes... you will do this week and in Lab 7.

## Part 1: Functions Vs. Methods & Attributes

A **function** returns a value, but a **procedure** does not.
A **method** is similar to a function, but is internal to part of a class. The term method is used almost exclusively in object-oriented programming.

Put another way....
**Functions** (in general) are things that are **done BY a program TO an object.**
**Methods** (in general) are things that are **done BY an OBJECT TO or ABOUT itself.**

Examples:
- describe what someone is wearing (function)
- tell me what's in someone's pocket (method)
- make a circle around someone (function)
- move someone's chair somewhere else in the room (method)
- get someone to sing the ABC's (method)
- class: put yourselves in order from tallest to shortest (could be either!)

Discussion: Can you come up with other things that are probably better as methods than functions?

Are you starting to get a sense for an object's "personal space"?

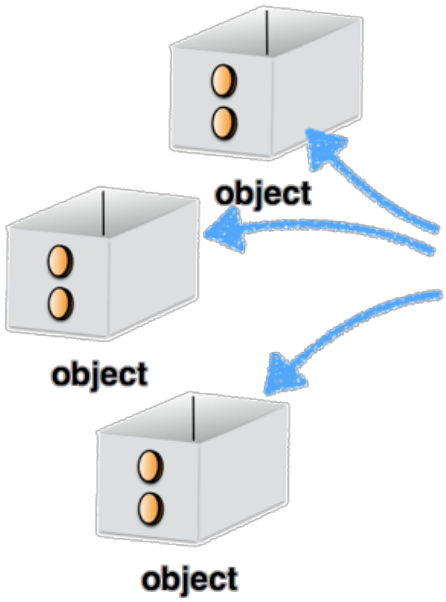
Good guidelines for understanding Methods:
- If it's something an object could reasonably do for itself, it should.
- If it changes anything about the object.
- Methods belong to the object, and they can access everything inside the object.
- When that thing needs to be done, you call the method.

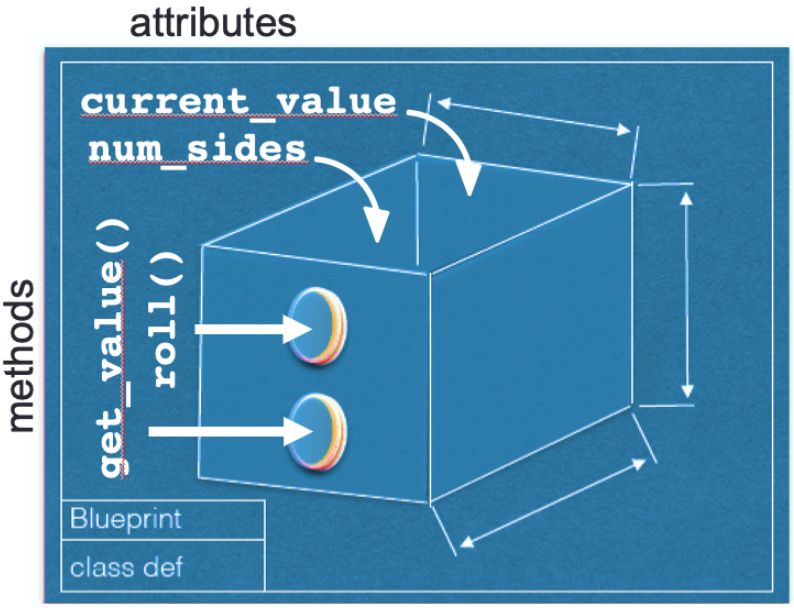
## Building a Die class

```
def main():
     d6 = Die(6)
     d8 = Die(8)

     d6.roll()
     print( "Value of d6:",
          d6.getValue() )

     d8.roll()
     print( "Value of d8:",
          d8.getValue() )
```

By default, python doesn't know how to do this.
1. a way to build a Die given # sides
2. to be able to .roll() them
3. to be able to .getValue()

Blueprint (Class) for a Die




**Die Class**

```
# libraries
from random import randrange

# a class for a die
class Die:
    def __init__(self, n):  #    def __init__(self):
        self.num_sides = n
        self.value = 1

    def roll(self):
        self.value = randrange(1, self.num_sides + 1)

    def getValue(self):
        return self.value
```

- Classes are defined using `class`.
  - Start class names with a capital letter.
- All classes need a constructor.
  - Python constructors are always called `__init__`
  - Attribute values get initialized here.
- Methods are `def`ined inside the class and indented.
- Self:
  - When attached to a variable, self makes the variable a "member" of the object.
  - Every method in a class automatically gets passed a reference to the object as its first parameter. (Python specialty. 😉)
  - **BUT** We don't put the self reference into any of the method calls.

# Programming Paradigms


object-oriented


functional


imperative


declarative

**Imperative ("procedural") programming:**
Program is structured as a set of steps (functions and code blocks) that flow sequentially to complete a task

**Object-oriented programming ("OOP") :**
Program is structured as a set of objects (with attributes and methods) that group together data and actions


Discussion: What do you think the pros and cons are for each?

| | **Object-oriented** (a.k.a. "OOP") | **Imperative** (a.k.a. "procedural") |
|---|---|---|
| PROS | + more organized (logically)<br>+ matches the real world<br>+ easier to test / debug<br>+ easier to reuse code | + easy to learn and implement<br>+ only need to think a few steps ahead<br>+ much more straightforward |
| CONS | - more "overhead" (need to plan out further in advance)<br>- harder to learn<br>- overkill for small tasks | - can be hard to follow returns<br>- have to pass stuff around<br>- gets "unwieldy" / "clunky"<br>- hard to test / debug |

Review the Die Class:

```
# libraries
from random import randrange

# a class for a die
class Die:
    def __init__(self, n):
        self.num_sides = n
        self.current_value = 1

    def roll(self):
        self.current_value = randrange(1, self.num_sides + 1)

    def getValue(self):
        return self.current_value
```

Identify the constructor, attributes, methods...

What happens if we run this program?

Nothing! We can in instantiate it...creating objects.
Each die object has different attributes.

```
d6 = Die(6)     # Calling the Constructor -> six sided die
d8 = Die(8)     # Calling the Constructor -> eight sided die
```

---

## Accessing Attributes

Question: Why do we have this method? Why don't we access the attribute directly?

```
    def getValue(self):
        return self.current_value
```

DO NOT DO...

```
print(d6.current_value)  # RUDE: access Song's
attributes directly
d6.current_value = 2     # WORSE: change it without
permission
```


Think back to our ATM example...
Can you imagine any attributes/methods you might want to be private?
print(account.pin)

---

## public vs. private

- python methods/attributes are **public** by default
  - this means that they can be accessed from outside the instance… by anyone (for better or for worse)
- To make a method/attribute private (i.e. accessible only within the instance itself),
  - prefix it with a double underscore (_ _)

```
def __init__(self, n_sides):
        self.__num_sides = n_sides
        self.__current_value = 1
```

## Big takeaways

- Object-oriented programming is a powerful paradigm
- It's also very common (and therefore useful to learn)
- The more complex your problem, the more it makes sense to organize your code this way
- In Python, it isn't all or nothing: some parts of your program might be object-oriented, others might be procedural
- The important part is that your code makes sense

## Cash Register Example Con't

```
class CashRegister:
   def __init__(self, ones, twos, fives, tens, twenties):
          ...

   def add(self, count, denomination):
       self.cash[denomination] = self.cash[denomination] + count
```

Activity: Now write delete?

```
   def remove(self, count, denomination):
       self.cash[denomination] = self.cash[denomination] - count
```

OR

```
   def remove(self, count, denomination):
       self.add(-count,denomination)
```

## Printing Objects

Have you tried to print an object... you get something like:

```
<__main__.CashRegister object at 0x10f0342b0>
```

Not terribly helpful. But, we can override this result by introducing our own response to the print command. Remember, print() must be passed strings, so what we really need to do is give beaker a way to present itself as a string.

We will used the double underscores to indicate we are overriding the default string response when it is applied to our object.

Inside the class definition, we write:

```
   def __str__(self):
        """Print cash register contents"""
        return "The cash register holds: " \
            + "\n20's: " + str(self.cash["twenties"]) \
            + "\n10's: " + str(self.cash["tens"]) \
            + "\n5's: " + str(self.cash["fives"]) \
            + "\n2's: " + str(self.cash["twos"]) \
            + "\n1's: " + str(self.cash["ones"])

register = CashRegister(4, 2, 4, 5, 5)
print(register)
```

Now:

```
>>> register = CashRegister(4, 2, 4, 5, 5)
The cash register holds:
20's: 5
10's: 5
5's: 4
2's: 2
1's: 4
```

That's better! All we have done is renamed the method. When we call print(obj), then obj.__str__() is called to find out what string to print.

## Getters

How can we make accessing properties easier? We can create getter methods.

By creating getter methods at the same time we create attributes, we make it easier to modify the program later by just changing the getter function, instead of each instance of an attribute call.

Good programming practice to name all your getter methods with the same structure:

```
def get_attribute(self):
     return self.attribute location
```

The question came up last class as to why we would use getter methods.

We noted how accessing an object property deep within a complex data structure like a database (or that is a composite of several object attributes) is much easier with a getter.

This is an example for accessing properties of a cash register object:

```
    def get_fives(self):
        return self.fives
```

An additional reason that we have introduced getters is because in other programming languages you use, object attributes are not as easily accessed, so you may encounter getter structures in other programming languages in the future.
They are a very standard part of object oriented programming.

Also, Setters -> like getters only they set the value of an attribute.

---

## Encapsulation (Definition)

The core of object-oriented programming (OOP) is the organization of the program by encapsulating related data and functions together in an object.

To encapsulate something means to enclose it in some kind of container. In programming, encapsulation means keeping data and the code that uses it in one place and hiding the details of exactly how they work together. For example, each instance of class file keeps track of what file on the disk it is reading or writing and where it currently is in that file. The class hides the details of how this is done so that programmers can use it without needing to know the details of how it was implemented.

---

## Activity: Write a Dog Class

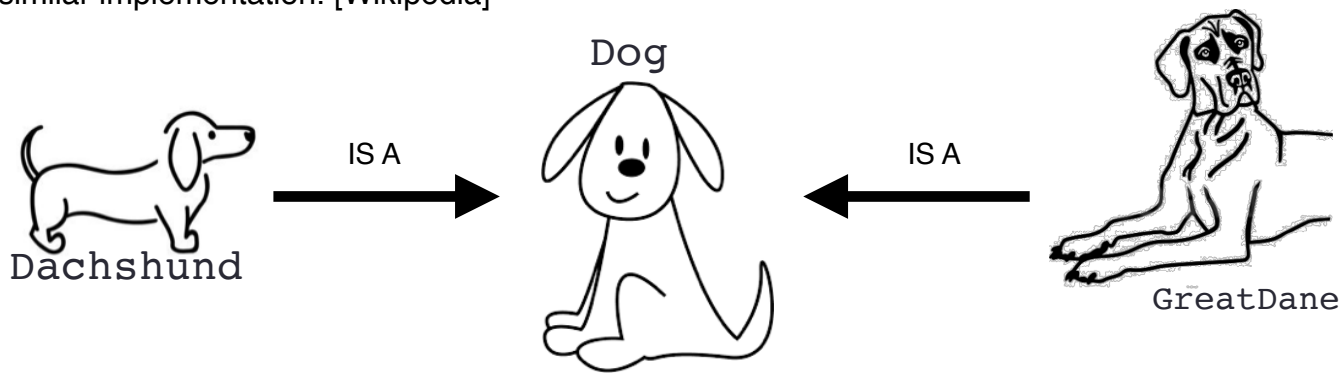
Write a class called Dog, with a constructor that takes in the following parameters:

- name (the dog's name)
- age (the dog's age in years)

Write a method called 'bark', where the dog says something.

## Part 3: Inheritance

In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. [Wikipedia]




```
class Dog:
    # A class attribute (every Dog has the same value).
    species = "CANINE"

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

    def bark(self):
        print("Woof! I am a", Dog.species)

class Dachshund(Dog):
    def runLowToGround(self):
        print("I'm runnin' so lowwwww...")

class GreatDane(Dog):
    def leapOver(self, something):
        print("I'm leaping over", something)
```

Subclasses "inherit" all the attributes and methods from their parent class.
They can also have their own attributes and methods separate from their parent.

If necessary, they can "override" attributes and methods from their parent.

```
class RobotDog(Dog):
    species = "ROBOT"
    def bark(self):
        print("Woof! I am a", RobotDog.species)
```

## Discussion

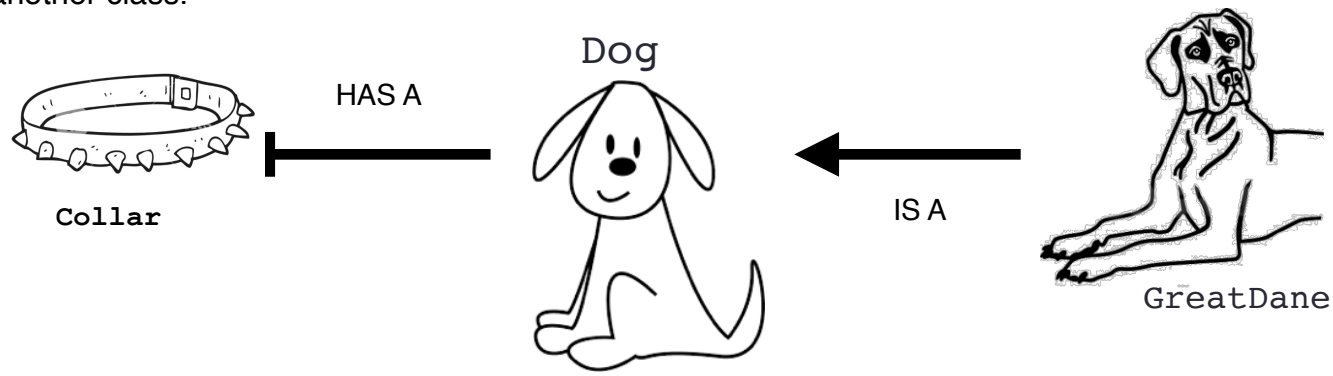
Why is this "inheritance" idea useful?

Let's practice what abstraction...what are the common property of all dogs, people, items.

Pick two items, try to come up with common properties between them.

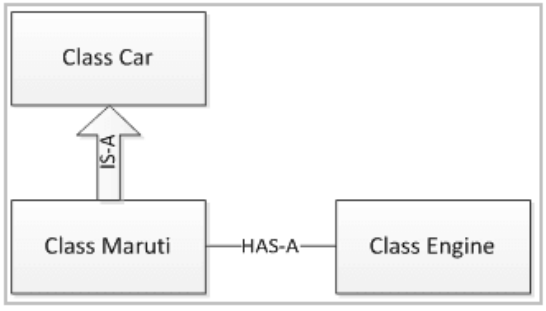
## Inheritance vs. Composition

In object-oriented programming, composition means that an instance variable is an object of another class.




```
class Collar:
    def __init__(self, color):
        self.color = color
    def getColor(self):
        return self.color

class Dog:
    def __init__(self, name, age, collar:Collar):
        self.name = name
        self.age = age
        self.collar = collar

    def bark(self):
        print("Woof! My collar is:", self.collar.getColor())

my_collar = Collar("red")
my_dog = Dog("Baxter", 2, my_collar)
my_dog.bark()
```

When working with classes make yourself a diagram with your "is a" and "has a" relationships.




[https://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php]

Review: Inheritance (point.py)

```
import math
class Point:
    def __init__ (self, x, y):
        self.x = x
        self.y = y
        self.numPoint = 2

    def sum_squares (self, otherPoint):
        return (self.x - otherPoint.getX())**2 +
          (self.y - otherPoint.getY())**2

    def distance_to (self, otherPoint):
        # using distance formula: sqrt((x2-x1)^2 + (y2-y1)^2)
        return math.sqrt( self.sum_squares(otherPoint) )

    def __str__ (self):
        return "({},{})".format(self.x, self.y)
    def getNumPoint(self):
        return self.numPoint
    def getX (self):
        return self.x
    def getY (self):
        return self.y

class Point3D (Point):
    def __init__ (self, x, y, z):
        super().__init__(x,y)   #Always call super first!
        self.z = z
        self.numPoint = 3

    def sum_squares (self, otherPoint3D):
        return super().sum_squares(otherPoint3D) +
          (self.z - otherPoint3D.getZ())**2

    def __str__ (self):
        return "({},{},{})".format(self.x, self.y, self.z)
    def getZ (self):
        return self.z

p1 = Point(0,0)
p2 = Point(4,4)
d = p1.distance_to( p2 )
print("distance from", p1, "to", p2, "=", d)
print("NP2:", p1.getNumPoint())

p3 = Point3D(0,0,0)
p4 = Point3D(4,4,4)
d = p3.distance_to( p4)
print("distance from", p3, "to", p4, "=", d)
print("NP3:", p3.getNumPoint())
```

## Assert

Assert allow you to establish that a condition must be `True` at a point in the code. Python will produce an error if the *assertion* is `False`.

```
p1 = Point(0,0)
p2 = Point(4,4)
d = p1.distance_to( p2 )
print("distance from", p1, "to", p2, "=", d)
print("NP2:", p1.getNumPoint())

# distance should be 4*sqrt(2) = 5.66
assert round(d,2) == 5.66

p3 = Point3D(0,0,0)
p4 = Point3D(4,4,4)
d = p3.distance_to( p4)
print("distance from", p3, "to", p4, "=", d)
print("NP3:", p3.getNumPoint())

# distance should be sqrt( 4^2 + 4^2 + 4^2 ) = sqrt(48) = 6.93
assert round(d,2) == 6.93
print("Everything is working!")
```