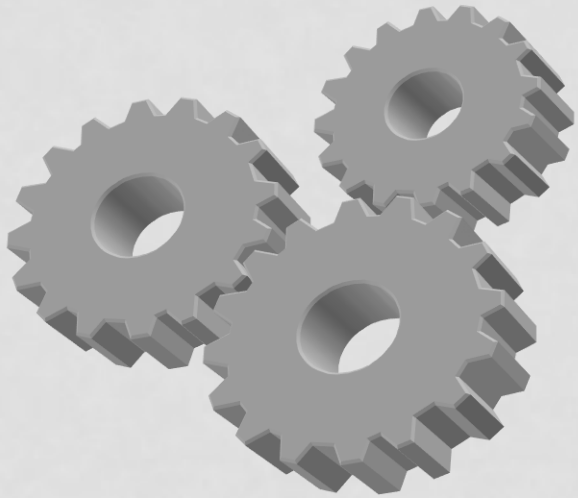

(PYHTONIC) OBJECT-ORIENTED PROGRAMMING (OOP) IN PYTHON

REFERENCES: CHAPTER 8

OOP IN GENERAL

- Break your program into types of Objects
  - Player
  - Enemy
  - Map
  - Application
  - …
- Collect all **data** and **functions** for that type into a class.
  - Example (Player):
    - Data: position, health, inventory, …
    - Functions: draw, handleInput, update, hitDetect, …
- Most modern software engineers use this approach to programming

CLASSES AND OBJECTS

- Classes.
  - A blueprint for a new python type
  - Includes:
    - Variables (**attributes**)
    - Functions (**methods**)
- Objects (instances)
  - A variable built using a class "blueprint"
  - All python variables are objects.
  - Creating an object creates a new set of attributes for that object.
  - You call methods of the class *through* an instance.

EXAMPLE

```
class Player(object):
      """This is the docstring for the player class."""

      def attack(self):
            """ This is the docstring for a method."""
            print("HIII-YA!“)


P = Player()     # This creates a new instance of Player
Q = Player()     # Another instance

P.attack()       # This calls the attack method of P.
Q.attack()       # self will refer to Q here.
```

Self: Note when we call P's attack method, we don't pass an argument for self.
self is a reference (alias) to the calling object (python handles this)
So…while we're in the attack method called from line 10, self refers to P.
But…while we're in the attack method called from line 11, self refers to Q.
Both calls use the same method definition, but self refers to different things.

All "normal" methods will have self as their first parameter.

ADDING ATTRIBUTES.

- To really do something useful, the class needs data (attributes).
- The first (really bad) way of doing it would be:

```
class Player(object):
        """This is the docstring for the player class."""

        def attack(self):
                """ This is the docstring for a method."""
                print("HIII-YA!")
                #print("HIII-YA!")    ays 'HIII-YA!'")

P = Player()     # This creates a new instance of Player
Q = Player()     # Another instance

P.name = "Bob"   # Associates the attribute (variable)
                    # name with P
P.attack()       # This calls the attack method of P.
Q.attack()       # Error!
```

Problem: we have to "manually" add attributes to all instances (error-pr

ADDING ATTRIBUTES THE "RIGHT" WAY (CONSTRUCTORS)

- Python allows you to write a special method which it will automatically call (if defined).
  - Called the **constructor**.
  - In python it is __init__
- Example:

```
class Player(object):
        def __init__(self):
                print("Hi! I'm a new player")

P = Player()   # Calls __init__ (passing P) automatically.
```

ADDING ATTRIBUTES THE "RIGHT" WAY (CONSTRUCTORS)

- Pass arguments to __init__ and creating attributes.
- Example:

```
class Player(object):
      def __init__(self, newName):
            self.name = newName    # A new attribute

      def attack(self):
            print(self.name + " says 'HIII-YA!'")

#P = Player()         # Error.  __init__ requires an argument.
P = Player("Bob")     # Calls __init__ (passing P and "bob") automatically.
Q = Player("Sue")
P.attack()
Q.attack()
```

Now we are guaranteed every player has a name attribute.

TEMPORARY VARIABLES IN METHODS

- Not all variables in a class method need self.
- Example:

```
class Player(object):
      def __init__(self, name):
            self.name = name
            self.hp = 100

      def defend(self):
            dmg = random.randint(1,10)
            self.hp -= dmg
```

- dmg is a **temporary** variable. It goes away after the defend method is done
  - Just like in a function.
  - Adding a self.dmg attribute would just clutter the class. It could potentially cause errors too.
  - **Only make something an attribute if you need to have it in multiple methods

STRING CONVERSION OF OBJECTS

- Normally, if you do this:

```
P = Player("Bob")
x = str(P)
print(x)
```

  - you get output similar to:

```
<__main.Player object at 0x00A0BA90>
```

- This is the "default" way of converting an object to a string
  - Just like the "default" way of initializing an object is to do nothing.
- Python allows us to decide how to do it.

STRING CONVERSION OF OBJECTS

- Example

```
class Player(object):
        def __init__(self, newName):
                self.name = newName
        def __str__(self):
                return "Hi, I'm " + self.name


P = Player("Bob")
x = str(P)
print(x)    # Prints 'Hi, I'm Bob'
```

A common mis-conception: The __str__ method needs to return a string, not directly print it.

EXAMPLE

- [Bubble-popper]

ADVANCED: INHERITANCE

- Not for this lab, but used in Lab9
- Basic idea:
  - Create a new (child) class from an *existing* class (parent)
  - Add new methods or modify existing.
  - Why?
    - Eliminate redundant code.
    - Indicates the child is a specialization of parent.
    - (not always a good thing…)
- Example:
  - Missile class: moves, draws, explodes
  - derived HomingMissile class: does all that, but modifies movement.

BASIC SYNTAX

```
class Shape(object):
      def __init__(self, pos, size, color):
            self.mColor = color
            self.mPos = pos
            self.mSize = size
            self.mColor = color

class Circle(Shape):
      # Note: no constructor!  We inherited it!
      def draw(self, surf):
            # Note2: we inherited all these attributes!
            pygame.draw.circle(surf, self.mColor,
                  self.mPos, self.mSize)
```

SUBTELTY: CALLING BASE-CLASS

- Challenge:
  - Sometimes we want to modify how a base-class method works.
  - Choice 1: just replace it
    - easy – just write it again.
  - Choice 2: do the base-class method and then add additional code.
    - But…we don't want to copy-paste (error-prone)

```
class Base(object):
    def __init__(self, x):
        # Do a bunch of stuff, including making self.mVal (setting it to x)
    def foo1(self):
        # Do a bunch of stuff.
    def foo2(self):
        # Do a bunch of stuff

class Derived(Base):
    def foo1(self):
        # This just replaces the inherited version

    def foo2(self):
        super().foo2()      # This calls the base-class version
        # Now we can do additional operations here.

    def __init__(self, x, y):
        super().__init__(x)     # This calls the base-class constructor, doing the
                                #    'stuff' (including making self.mVal)
        self.mVal2 = y          # A new attribute specific to Derived.
        # Other code, if necessary.
```

TEST-YOUR-KNOWLEDGE

- (Quiz-like problem)
- Create a class called Foo which:
  - Has these attributes:
    - mValue: an integer, passed to the constructor
    - mName: a string made up of 1-5 a's, b's, or c's (at random
- Produces this output when used in a main program

```
F = Foo(5) # 'aaa' was chosen for mName
G = Foo(3) # 'cc' was chosen for mName
print(F)          # output: aaa-aaa-aaa-aaa-aaa
print(G)          # output: cc-cc-cc
print(F.test())    # output: 25.0
print(G.test())    # output: 9.0
```