

# Python crash course

## Classes with python

with Jongbin Jung

All material publicly available [here \(https://github.com/jongbinjung/mse231-workshop\)](https://github.com/jongbinjung/mse231-workshop)

## 2. python programming - 3 of 3

The final topic of python programming (for the crash course, but by no means for python in general!) is classes.

### Classes

If you've worked with/written classes before in another language, then dealing with classes in python should be a breeze, once you get some syntax cleared. (But on the other hand, if you are experienced in programming to that degree, this introductory workshop was probably pretty boring for you...)

If you've never heard of object-oriented programming or classes in a programming context, keep in mind that while we'll try to cover some of the core concepts, this is by no means an exhaustive treatment of the topic. If you want to learn more about the fundamentals of object-oriented programming, there are tons of books and courses focusing on exactly that.

Here, let's try to focus on what's useful in getting work done with python, even if you don't quite understand precisely what's going on under-the-hood.

In python, you can define a class using the `class` keyword, followed by a class name and indented statements that belong to the class (one python naming convention is to use CamelCases for class names):

```
In [1]: class MyFirstClass:
        """A simple example class"""
        def __init__(self, name='Jongbin'):
            self.name = name
```

Let's pause there for a second.

So, what *is* a class? I find it useful (not always correct, but useful) to think of a class as a blueprint for some kind of entity. For example, a Human would be a class. A class itself doesn't necessarily do anything, just like the concept of a Human doesn't do much. However, there can be *instances* (copies, actual realizations) of a class, just like we are all instances of the Human class.

The class definition, therefore, defines what that class looks like.

- What **attribute** (data, values) does it have? For example, the Human class typically has a head, two arms, and two legs.
- What **methods** does the class have, i.e., what can the class do? For example, the Human class can think(), walk(), run(), type(), etc., each involving some kind of maneuvering of its attributes (head, arms, legs, ...)

While we're at it, let's try and define a simple Human class:

```
In [2]: class Human:
        """Basic Human class"""
        def __init__(self):
            """Initialize the Human"""
            self.head = True

        def think(self):
            """Makes the Human think"""
            if self.head:
                print "I'm thinking..."
            else:
                print "Pretty hard to think without a head..."
```

Remember, this definition is just a blueprint. If we actually want to use a human of the class Human, we need to create an *instance* of Human (note another convention: instances will usually be lower case). Creating an instance in python is as simple as pretending the class is a function and *calling* it:

```
In [3]: jongbin = Human()
```

Once we've created an instance, that instance has all the attributes and methods described in the blueprint:

```
In [4]: print jongbin.head # print the value of jongbin's head
        jongbin.think() # make the Human jongbin think
        jongbin.head = False # set the value of jongbin's head to False
        jongbin.think() # make the Human jongbin think, again

True
I'm thinking...
Pretty hard to think without a head...
```

Some notes on syntax:

- `self`: The `self` keyword in class definitions refer to the instance of the class that is making the calls. All class methods (functions defined inside a class definition) are required to take `self` as the first argument. For now, let's think of this as a syntactic requirement. More on this later ...
- `__init__`: The `__init__` method is called by default when a class instance is created. If the class definition doesn't explicitly define an `__init__` method, an empty class object is created. In other words, you don't **have to** define an `__init__` method, but it's usually useful to have one so that you can declare some initial values for instances of your class.

The `__init__` method can also take arguments (other than `self`). So, for example, if we wanted our humans to have a name, and let users define the human's name when the human is created, we can modify the `__init__` method to:

```
In [5]: class Human:
        """Basic Human class"""
        def __init__(self, name):
            """Initialize the Human"""
            self.head = True
            self.name = name

        def think(self):
            """Makes the Human think"""
            if self.head:
                print 'My name is', self.name, "and I'm thinking..."
            else:
                print "Pretty hard to think without a head..."
```

```
In [6]: jongbin = Human('Jongbin')
        jongbin.think()
```

My name is Jongbin and I'm thinking...

## The `self`

Now let's talk a bit more about that `self`. It is important to have a clear distinction between the class as a class definition (or class object), and an actual **instance** of said class. Attributes in a class definition that are not prepended with `self` will belong to the class object, and anything prepended with a `self` will belong to the instance of that class that is currently making the call.

This can be mildly confusing, but becomes quite important if you have mutable attributes. Let's add a list of tools for our Human class:

```
In [7]: class Human:
        """Basic Human class"""
        tools = [] # all Humans have a list of tools

        def __init__(self, name):
            """Initialize the Human"""
            self.head = True
            self.name = name

        def think(self):
            """Makes the Human think"""
            if self.head:
                print 'My name is', self.name, "and I'm thinking..."
            else:
                print "Pretty hard to think without a head..."

        def add_tool(self, tool):
            """Add a tool to the list of tools"""
            self.tools.append(tool)

        def get_tools(self):
            """Print the tools that the human has"""
            print self.name, 'has:',
            for tool in self.tools:
                print tool, '|',
            print # create newline after last tool
```

Now, lets create some humans, and see how the list of tools work:

```
In [8]: luke = Human('Luke')
        anakin = Human('Anakin')
        luke.add_tool('light saber') # add light saber to Luke's tools
        anakin.add_tool('death star') # add death star to Anakin's tools
        luke.get_tools() # print Luke's tools
        anakin.get_tools() # print Anakin's tools

Luke has: light saber | death star |
Anakin has: light saber | death star |
```

See that even though two Human instances added their own tool, both ended up having all the tools, since the tools list was declared without the self prepended, making it an attribute of the class object, instead of an attribute of each instance.

If we wanted each Human to have ther own list of tools, a better way to define the class would have been:

```
In [9]: class Human:
        """Basic Human class"""
        def __init__(self, name):
            """Initialize the Human"""
            self.head = True
            self.name = name

            self.tools = [] # initiate a specific human's list of tools

        def think(self):
            """Makes the Human think"""
            if self.head:
                print 'My name is', self.name, "and I'm thinking..."
            else:
                print "Pretty hard to think without a head..."

        def add_tool(self, tool):
            """Add a tool to the list of tools"""
            self.tools.append(tool)

        def get_tools(self):
            """Print the tools that the human has"""
            print self.name, 'has:',
            for tool in self.tools:
                print tool, '|',
            print # create newline after last tool
```

```
In [10]: luke = Human('Luke')
          anakin = Human('Anakin')
          luke.add_tool('light saber') # add light saber to Luke's tools
          anakin.add_tool('death star') # add death star to Anakin's tools
          luke.get_tools() # print Luke's tools
          anakin.get_tools() # print Anakin's tools

Luke has: light saber |
Anakin has: death star |
```

Note that there's no right or wrong way to define attributes. There actually might be cases where you **want** different instances to share a single attribute.

For example, think of a `Bar` class that has a `bathroom` attribute. All bars share a single bathroom which can only hold a single customer. We want to print a message if any bar tries to use *the* bathroom while it's occupied. We can define such a `Bar` class as:

```
In [11]: class Bar:
        """A fictional Bar"""
        bathroom_occupancy = []
        def use_bathroom(self):
            if len(self.bathroom_occupancy) < 1:
                print 'use bathroom'
                self.bathroom_occupancy.append('in use')
            else:
                print 'bathroom in use!'

        # instantiate two bars
        coolBar = Bar()
        hotBar = Bar()

        print 'coolBar:',
        coolBar.use_bathroom()
        print 'hotBar:',
        hotBar.use_bathroom()
```

```
coolBar: use bathroom
hotBar: bathroom in use!
```

## Inheritance

A very useful concept related to classes is inheritance. Basically, classes can 'inherit' other classes, and build upon them. Inheritance is especially useful when you want to build upon a class that you didn't write, which happens sometimes when using 3rd party modules. You can inherit a class by specifying the parent class in parentheses when defining the class name.

For example, let's create the SuperHuman, which is a child of the Human class, except that a SuperHuman can think even without a head!

```
In [12]: class SuperHuman(Human): # the Human in parens states that we're inheriting the Human class
        """SuperHuman that thinks without a head"""
        def think(self):
            """Makes the Human think"""
            if self.head:
                print 'My name is', self.name, "and I'm thinking!!!"
            else:
                print "Pretty hard to think without a head...BUT I'M STILL THINKING!!!"

        clark = SuperHuman('Clark Kent')
        clark.think()
        clark.head = False
        clark.think()

        clark.add_tool('laser vision')
        clark.add_tool('spandex')
        clark.get_tools()
```

```
My name is Clark Kent and I'm thinking!!!
Pretty hard to think without a head...BUT I'M STILL THINKING!!!
Clark Kent has: laser vision | spandex |
```

Note how a child class, by default, has all the attributes and methods of its parent, without explicitly defining them. However, when the child wants to change the behavior of an inherited parent, it simply redefines a method of the same name, which effectively overwrites the inherited method.

Sometimes, instead of replacing the inherited method, you might want to build upon it. In such cases, it's common to write the new method such that the child first calls the parent's method with required parameters, using the syntax `ParentClassName.method_name(self, arguments)`, and then work from there.

For example, let's modify our SuperHuman's `__init__` behavior. We still want to initialize the head, name, and tools, but we also want to add a new `home_planet` attribute:

```
In [13]: class SuperHuman(Human): # the Human in parens states that we're inheriting the Human class
        """SuperHuman that thinks without a head"""
        def __init__(self, name, home_planet='Krypton'):
            """SuperHuman's augmented initialization (with default values)"""
            Human.__init__(self, name) # call the parent's __init__ with required arguments
            self.home_planet = home_planet
            # say hello when instantiated
            print 'Hi, my name is', self.name, "and I'm from", self.home_planet

        def think(self):
            """Makes the Human think"""
            if self.head:
                print "I'm thinking!!!"
            else:
                print "Pretty hard to think without a head...BUT I'M STILL THINKING!!!"

clark = SuperHuman('Clark Kent', 'Krypton')
```

Hi, my name is Clark Kent and I'm from Krypton

### One last thing

A class method can use one or more of its own methods by specifying `self` before the method call.

For example, let's add a `think_hard(self, n)` method to the `SuperHuman`, which makes the `SuperHuman` `think()` `n` times, lose its head and think one last time:



```
In [14]: class SuperHuman(Human): # the Human in parens states that we're inheriting the Human class
        """SuperHuman that thinks without a head"""
        def __init__(self, name, home_planet='Krypton'):
            """SuperHuman's augmented initialization (with default values)"""
            Human.__init__(self, name) # call the parent's __init__ with required arguments
            self.home_planet = home_planet
            # say hello when instantiated
            print 'Hi, my name is', self.name, "and I'm from", self.home_planet

        def think(self):
            """Makes the Human think"""
            if self.head:
                print "I'm thinking!!!"
            else:
                print "Pretty hard to think without a head...BUT I'M STILL THINKING!!!"

        def think_hard(self, n):
            """Makes the Human think harder (n times)"""
            print "Oh, I'm gonna think SO hard!"
            for i in range(n): # loop n times
                self.think()
            self.head = False # lose head
            self.think() # one last think()!

clark = SuperHuman('Clark Kent', 'Krypton')
clark.think_hard(5)
```

Hi, my name is Clark Kent and I'm from Krypton  
Oh, I'm gonna think SO hard!  
I'm thinking!!!  
I'm thinking!!!  
I'm thinking!!!  
I'm thinking!!!  
I'm thinking!!!  
Pretty hard to think without a head...BUT I'M STILL THINKING!!!

## Exercise 4.

Continuing from the previous exercises (kind of) ...

1. Define a class **BookReader**, which instantiates with a filename as an argument. The **BookReader** has a single method, `count_words()`, which returns a dictionary of word occurrence counts in the text specified by filename.
2. Define another class, **OnlineBookReader**, which inherits the **BookReader** class. The only difference between **BookReader** and **OnlineBookReader** is that **OnlineBookReader** takes a URL as a single argument when instantiated. The **OnlineBookReader** then downloads the specified URL locally, and uses the downloaded file when `count_words()` is called (all implemented in the `__init__` method)