Classes in Python – An Introduction

Though not always explicitly stated, Python provides several types of variables to store data in: int, float, boolean and string. However, not all information can be easily stored in one of these types. One solution that we've seen is using a list. A list can store an arbitrary amount of information. Imagine storing the following information about a Person:

1. Name
2. Age
3. Phone Number
4. Birthday

We could store this information in a list:

```
myperson = ["Sylvia", 25, 4078231000, 1213]
```

In this system, both the phone number and birthday are integers, where the latter is equal to 100 times the month (1 to 12) plus the day. To refer to someone's name we would do:

myperson[0]

To refer to someone's age we would do:

myperson[1].

The main difficulty of this system is that for the programmer, it might be hard to remember that index 0 is name, index 1 is age, index 2 is phone number and index 3 is birthday. In some sense, it would be better to refer to these things in a manner similar to this:

myperson[name], myperson[age], myperson[phonenumber], myperson[bday]

Python allows for something similar to this, but because there is a strong convention that the items in subscripts can not be variables themselves (you can put non-integers if you are using a dictionary), Python allows for this sort of nomenclature via different syntax.

In addition, not only might someone want to define their own type of variable, they may want to create special functions that operate on variables of that type only.

Creating your own type in a programming language AND creating special functions that operate on variables of that type has its own name for the programming paradigm: Object-Oriented Programming. The definition for creating the type is called creating a class. To use a class, one must create an object (an instance) of that class. This is essentially declaring and initializing a variable of the type you just created.

All classes in Python must have the following components:

1. List of instance variables – these are the components that make up the object. (In our previous example these are name, age, phone number and birthday.)

2. A constructor – this is a method (functions that are inside of classes are called methods) that describes how to build or construct an instance of the object.

Most classes should also have the following method:

```
def __str__(self):
```

The job of this method is to return a string representation of the object. It automatically gets invoked anytime one tries to print an object via the print statement.

Most classes also have methods other than the constructor. All methods in a class are required to list at least one parameter, (an object of the class itself), because it's understood that these methods can only be called on objects of the class. (For example, if we had a Cat class which had a meow method, it wouldn't make any sense to call that meow method on a Dog object or Human object.) For Python, the convention is that we name this parameter self.

Perhaps the easiest way to start learning how to define a class is to see a full class definition, and then take it apart, piece by piece. Here is the Contact class, which stores information about a person:

```
# Written By: Arup Guha
# Translated By: Tyler Woodhull
# 6/1/13
# Contact.py

class Contact:

    name = "" # Stores name of Contact
    age = 0 # Stores age of Contact
    phonenumber = 0 # Stores phone number of Contact
    bday = 0 # Stores birthday in an int

    # Creates Contact object based on parameters.
    def __init__(self, n, a, p, month, day):
        self.name = n
        self.age = a
        self.phonenumber = p
        self.bday = 100 * month + day

    # Changes phone number of contact
    def changeNumber(self, newnum):
        self.phonenumber = newnum
    # Implements the passing of the Contact's birthday.
    def Birthday(self):
        self.age += 1

    # Returns the name of the Contact
    def getName(self):
        return self.name

    # Returns the age of a Contact
    def getAge(self):
        return self.age

    # Returns the phone number of a Contact
    def getNumber(self):
        return self.phonenumber

    # Returns month of Contact's birthday
    def getBdayMonth(self):
        return self.bday//100

    # Returns day of the month of Contact's birthday
    def getBdayDay(self):
        return self.bday%100

    # Contact has a birthday.
    def celebrateBirthday(self):
        self.age += 1

    # Prints all information about a Contact out.
    def printContact(self):
        print("In printContact")
        print("Name: " + self.name + " Age: " + str(self.age))
        print("Phone#: " + str(self.phonenumber))
        print("Birthday: " + str(self.getBdayMonth()) + "/" +
str(self.getBdayDay()))
        print()

    # Prints all information about a Contact out.
    def __str__(self):
        ans = "Name: " + self.name + " Age: " + str(self.age) + "\n"
        ans = ans + "Phone#: " + str(self.phonenumber) + "\n"
        ans = ans + "Birthday: " + str(self.getBdayMonth()) + "/" +
str(self.getBdayDay())+"\n"
        return ans
```

The convention is to list each of the instance variables, the pieces that make up the object, first. Each of these must be set to a default value. Here is this portion of the Contact class:

```
name = "" # Stores name of Contact
age = 0 # Stores age of Contact
phonenumber = 0 # Stores phone number of Contact
bday = 0 # Stores birthday in an int
```

These are placeholder values that are never meant to be used. The constructor is what builds the object. In order to build a Contact, we need to know the following information:

name, age, phone number, birth month, birth day

Thus, when we write the constructor, we must have each of these items as a parameter, listed **AFTER** self. The name of the constructor is always `__init__`. Let's take a closer look at constructor for the Contact class:

```
def __init__(self, n, a, p, month, day):
    self.name = n
    self.age = a
    self.phonenumber = p
    self.bday = 100 * month + day
```

The formal parameters are: self, n, a, p, month, day. self is the object that is being built. n is the name of the person given to the function, a is their age, p their phone number, month the month they are born in and day the day they were born in. Most constructors are quite straight forward. In this case, most of the code is setting each component of the object to the information passed to the constructor. Constructors could have more complicated logic in them, but they typically do not. For this constructor, since we are storing the birthday as a single integer, we can assign the value to `self.bday` by doing a little mathematical calculation with both month and day.

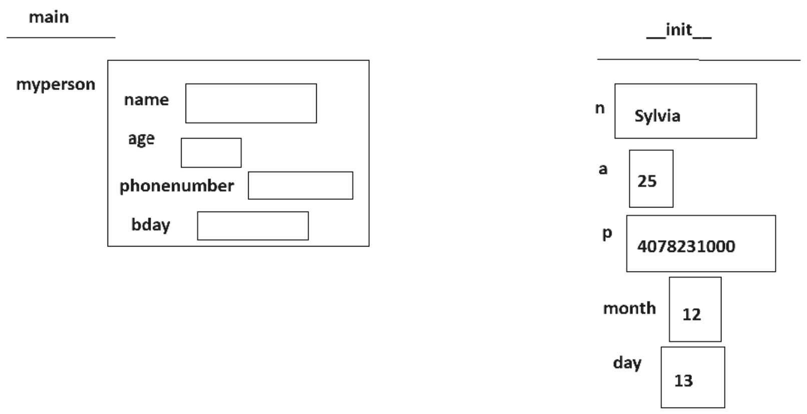
Thus, anytime we are inside of a method in a class and want to refer to one of the components of the object the method was called upon, we always use the dot operator to the right of self (the object the method was called upon.) This is the proper syntax, instead of using brackets.

Notice that this is much easier to read than self[0] or self[1] and having to remember what 0 or 1 stands for.

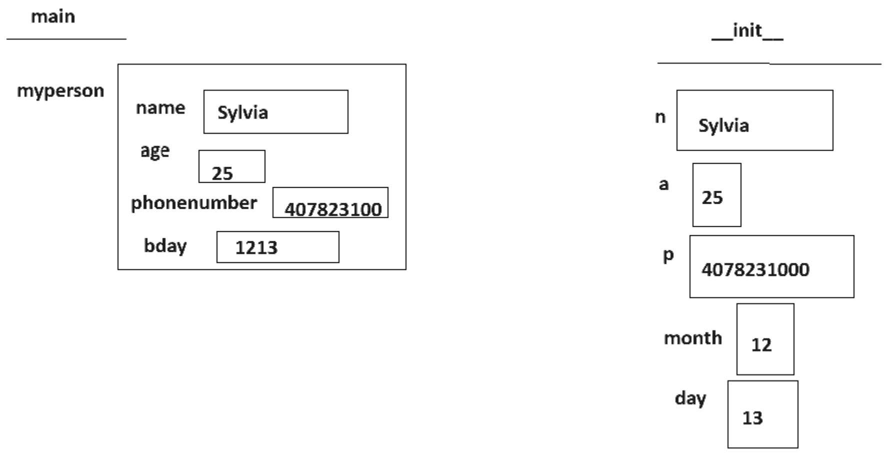
Now that we've seen the constructor definition, we need to know how to call the constructor. Namely, how can we build an object of this class? Here is a line of code that creates a single Contact object:

```
myperson = Contact("Sylvia", 25, 4078231000, 12, 13)
```

Although there is no reference to __init__, it is understood that when you use a class name followed by parentheses, that this is actually a call to the __init__ function. Here is a full picture of memory as this line of code is being called:



Thus, right when the constructor is called, each of the actual parameter values from the method call get copied into the formal parameters (n, a, p, month and day). After this, the constructor executes its lines of code. After the four lines of code in the constructor are executed, the picture looks like this:



Then, once the constructor completes, the memory for it disappears and we're just left with the fully initialized Contact object in main.

For our contact, other than basic methods that retrieve information about the Contact (these are often called getters), we have two ways to manipulate the Contact object via methods:

1. changeNumber
2. celebrateBirthday

The first allows you to change the phone number of a Contact object. The second allows a contact to age by one year. Notice that by convention, if a programmer creates a contact object and only uses the methods provided in the class, that a few things are true about the object:

1. Name can't be changed after its set initially.
2. Age can not decrease after it is set initially.
3. Only the phone number can be changed to something completely different.

From a design perspective, the idea is that whoever designs the class determines which behaviors are acceptable for objects of that class. Then, whenever another programmer wants to create an object of that class, they are limited to use the methods that the designer has provided to access and update the object. This allows the designer to worry about the fine details of how to get the implementation of various functions to work and the users only need to understand the method definitions in the class and what each method does at a high level (what information it needs to do its job and the job the method performs.)

Here is a function that creates a Contact object and calls some of the methods:

```
def testContact():

    # I have his bobblehead on my desk at home!
    lionel = Contact("LionelMessi", 36, 9999999999, 6, 24)

    # Invokes the __str__ method
    print(lionel)

    # This method is unnecessary, but I'm showing the difference syntactically.
    # Whereas __str__ is special, printContact is a regular void method.
    lionel.printContact()

    # Give him a birthday.
    lionel.celebrateBirthday()
    print("After his birthday, here is Leo's information:")
    print(lionel)

    # Leo gets a Miami phone number!
    lionel.changeNumber(3059999999)
    print("Leo avoids being found by changing his number:")
    print(lionel)
```

Thus, in general, except for the constructor and __str__ methods, the way to call any method in a class is through the dot operator. The other peculiar thing is that the object itself isn't passed in the parameter list, even though it appears there it the method definition. Basically, the object the method was called upon is referenced by self once any of the methods start running. So, for example, when we say `lionel.celebrateBirthday()`, self will refer to the same object that lionel does and add 1 to its age instance variable (component). Thus, when we print lionel afterwards, he is 37 years old. Similarly, after changing his phone number, print afterwards verifies that his number was changed.

In the second example in the class notes, we create a CellPhone class that manages a Cell Phone plan with data and minutes. A CellPhone object can be created, and once its created, one can use it to either talk (using minutes) or use data. If there's not enough of either left, you're not allowed to execute the call or use of data. The last piece of functionality in the class is receiving the bill.

Here are the instance variables of the class and two of the more interesting methods:

```
class CellPhone:

    minPerMonth = 500
    dataPerMonth = 2000
    data = 2000
    cost = 79
    overMinCost = 0.25
    curMin = 0

    def bill(self):

        # Clear out the object.
        storeMin = self.curMin
        self.curMin = 0
        self.data = self.dataPerMonth

        # Regular bill.
        if storeMin <= self.minPerMonth:
            return self.cost

        # Overage bill
        return self.cost + self.overMinCost*(storeMin - self.minPerMonth)

    # Use data.
    def goOnline(self, info):

        # Typical case.
        if self.data >= info:
            print("You have successfully used",info,"megs of data.")
            self.data -= info

        # Ooops, you've run out of data.
        else:
            print("You    do    not    have    enough    data,    you    only
get",self.data,"megs.")
            self.data = 0
```

In the bill method, we use the information in self to calculate the total bill and return that cost. If you haven't gone over in minutes, you just pay the regular cost. Otherwise, you have to pay for the extra minutes. Namely, the minutes you've talked minus your minutes per month is how many overtime minutes you have to pay for. For each of these minutes, you pay overMinCost dollars.

In the goOnline method, since your data just stops once you use it up, if you request to use more than you have, the method doesn't allow it. Instead, it just gives you as much data as it can until you run out.