American Concrete Institute (ACI)-UIUC Student Chapter

Python Workshop (Basic and Advanced)

Sushobhan Sen

Doctoral Candidate

April 6, 2019

Urbana, IL

Contents

1 Introduction

1.1 Why Python?

The Python programming language was conceived in the 1980s and the first implementation was deployed in 1991 by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands. Python is designed to be a high-level, interpreted, general-purpose computer programming language with exception-handling and the ability to be extended by users. Crucially, Python is an open-source language, meaning anybody can see the code behind it. Python today is a very popular language is a broad variety of disciplines: machine learning, web development, Geographical Information Systems (GIS), engineering, database management, high performance computing, etc.

Python 2.0, which truly launched the popularity of the language, was released in 2000, with the last version in the series, Python 2.7, set to be phased out in 2020. The current recommended standard as of when I wrote this is Python 3.7.2, which is crucially **not backwards compatible** with Python 2.x. All subversions of Python 3.x are backwards compatible though, so feel free to update regularly.

(The above information was taken from Wikipedia)

The beauty of Python is its ease - the syntax is extremely simple as compared to many other popular programming languages. It is often said, not entirely without truth, that the difference between Python code and pseudocode is merely the indentation. Thus, the language is very easy to learn for programming novices. However, its ease should not be misconstrued to mean that it is only for easy or trivial applications: Python is an extremely powerful language with a wide variety of applications, it has extensive documentation, and its open-source nature ensures that new features are being developed continuously while bugs are also being fixed.

1.2 Learning Objectives

This workshop is broken up into two sessions:

1. A Basic Python session for beginners with zero knowledge of programming in any language

2. An Advanced Python session for those with prior knowledge of Python programming

At the end of the **Basic Python** session, participants will be able to:

1. List the types of Python variables and define them
2. Add control statements (`if-then-else`) and `for` loops to their program
3. Define and use functions
4. Define and use object oriented programming

At the end of the **Advanced Python** session, participants will be able to:

1. Use the `numpy` library to define and use matrices, and read and process data from files
2. Use the `pandas` library to read and process data
3. Use the `scipy` library to solve a system if linear equations and implement other useful scientific functions
4. Use the `matplotlib` library to create publication-quality plots

The lists above ware what I will try to accomplish in our two two-hour long sessions. Depending on how quickly the workshop goes, I may or may not be able to meet all the learning objectives. However, I will get you along far enough so that you can complete any remaining items on your own.

1.3 Installing Python

The easiest and **recommended** way to get all popular Python libraries and IDEs is by downloading and installing the latest Anaconda distribution for your computer. Make sure to download the latest version corresponding to Python 3.x.

Alternatively, you can install and use Python using the terminal:

- On Windows 10, activate Windows Subsystem for Linux

- On Mac OS or any UNIX-like OS (such as Linux), just run your favorite Terminal application

Then install Jupyter with `pip`. With any other version of Windows (which you should not be using for too long on your personal computers anyway for security reasons), Anaconda is your best option.

Once installed, you can now create a new notebook. If you installed Anaconda, open the Anaconda Prompt, navigate to your directory, and use the `jupyter notebook` command. If you choose to use the terminal, follow the same steps on the terminal. Both methods will launch Jupyter Notebooks in your browser. From there, you can create a new Notebook.

If you prefer not to install anything on your computer but would rather run Python remotely from your browser, you can use the Online IDE from repl.it. This doesn't always work very well though. Any other online IDE that you find should be OK too.

2 Variables in Python

2.1 Data Types

Python defines the following data types for variables:

Data Type	**Syntax**	**Description**	**Comments**
Integer	`x = 5`	Signed integer	Use `int` to typecast, if valid
Float	`x = 5.`	IEEE floating point number	Use `float` to typecast, if valid
Complex	`x = 3.1+4.6j`	Complex number	Use `complex` to typecast, if valid
String	`x = 'Hello World!'`	Strings are always enclosed within quotation marks	Use `str` to typecast, slicing operator valid
List	`x = [1, 2.2, 'otter']`	**Mutable** list of variables of any type	Use `list()` to typecast, if valid
Tuple	`x = (1, 2.2, 'otter')`	Immutable tuple of variables of any tupe	Use `tuple()` to typecast, if valid
Dictionary	`x = {'one':1, 'two':2}`	Key-value pairs	Use `get()` to get `value` from `key`, use `dict()` to typecast, if valid
Bool	`x = True`	Boolean value	Python uses keywords `True` and `False`, not 1 and 0

2.2 Creating a Variable

Variables are usually given a name, which is any string of characters you use to call it. Any string of characters is a valid name, although there are a couple of rules to follow:

1. The name **cannot** contain spaces - consider using an underscore character or chamelCase instead for readability
2. The name cannot start with a number, but can contain a number anywhere
3. The name is case-sensitive, so `temp` and `Temp` are different variables
4. Python has a number of reserved keywords (see documentation, of just follow along and you'll get the hang of it), which cannot be used as names

Creating a variable is as simple as giving it a name, and assigning a value to it. For example, run this piece of code:

```
1 x = 1.0
2 y = 'Hello, World!'
3 z = (x==y)
4 print(type(x),type(y),type(z),z)
```

Here, we defined three variables: `x` of type `int`, `y` of type `str`, and `z` of type `bool`. Note that the `==` is a comparison operator that compares two values, while `=` is an assignment operator that assigns a value to a variable. The `print()` function, as the name suggests, prints the comma-separated inputs as a string , while the `type()` function returns the data type of the input. Note that every new line of code in Python starts on a new line, and **there is no end of line character** (such as a semi-colon in many languages).

2.3 Mutability

Some Python data types are **immutable**. This means that, once defined, ibjects of those types cannot be changed without creating a new object entirely. Most objects are immutable: `int, float, complex, str, bool, tuple` are immutable. If you try to change their values, you will either get an error or a new object containing the new value will be created. For example, run the following code, where `id()` is a function that returns the memory location of the object passed to it:

```
x = 1
print(id(x))
x = 2.0
print(id(x))
```

You'll see that the memory location of `x` has changed entirely, instead of just the value being changed. This is because `x` was defined as type `int`, which is immutable. Similarly, try changing the value inside a tuple and see what happens (note that `x[0]` is a way to reference the first element of `x` - **Python is zero-indexed**):

```
x = (1,2,3)
x[0] = 10
print(x)
```

Lists and dictionaries are mutable, which means their values can be changed at any time. Try running the following code and compare it to the last one:

```
x = [1,2,3]
print(id(x))
x[0] = 10
print(id(x))
print(x)
```

2.4 Operations on Variables

Python provides a number of operations that can be performed between variables:

- **Arithmetic operators:** + (add), - (subtract), * (multiply), / (divide), % (modulus or remainder), // (floor division), ** (exponent)
- **Comparison operators:** > (grater than), < (less than), == (equal to), != (not equal to), >= (greater than or equal to), <= (less than or equal to)
- **Logical operators:** `and`, `or`, and `not`

- **Bitwise operators:** & (bitwise AND), | (bitwise OR), ∼ (bitwise NOT), (bitwise XOR), >> (bitwise right shift), << (bitwise left shift)
- **Assignment operators:** A combination of the basic assignment operator (=) and an optional arithmetic or bitwise operator. For example, `x **= 5` is the same as `x = x**5`
- **Special operators:** `is` or `is not` check if two variables have the same memory address, `in` or `not in` check if a variable is in a sequences of variables

Operators are mostly self-explanatory, but their behavior can be different based on the data type of the input variables. Consider adding two integers, adding an integer to a float, adding two strings, and adding an integer to a string, all of which use the same + operator:

```
1 x = 2
2 print(x+2)
3 print(x+2.5)
4 print('Goodbye'+'Hello')
5 print(x+'Hello')
```

The first operation is as expected and returns an integer. In the second operation, `x` is first typecast to `float` *implicitly* (the program does it by itself) and then added to another float to return a float. In the third operation, two strings are simply concatenated together to return another string.

However, the last operation of adding an integer to a string returns an error, because Python is unable to figure out which variable to cast to which type. Of course, casting a string to an integer makes no sense, but Python doesn't even try, because it's unsure about what to do. You can help it by *explicitly* casting `x` from an integer to string, and then see what happens:

```
1 x = 2
2 print(str(x)+'Hello')
```

3 Control Statements

Control statements are blocks of code that are used to control the flow of the program - that could mean skipping some lines, or repeating some lines a certain number of times. There are two main types of control statements:

1. **Conditional statements**: Better known as if-else blocks, these test a condition before executing a line
2. **Loops**: These repeatedly execute a line for a certain number of times or till a break condition is met

We'll look at each of these below.

3.1 Conditional Statements

The most basic control statement is an `if` statement, which checks a condition and executes the code after it only if the condition evaluates `True`. Consider the following block of code:

```
1 x = 6
2 if x<5:
3     print('x is less than 5')
4     print('Finished evaluating if block')
5 print('Finished this code block')
```

First, notice the syntax: the `if` statement starts with the word itself, followed by a space, and followed by the condition (`x<5`). This line then ends with a colon (:), signifying that things after the colon are to be executed if the condition returns `True`. But how do you tell Python what lines of code are part of the `if`-statement to be executed, and what are not? In other words, how do you signify the scope of the `if`-statement? In this case, we want the two `print()` statements after the `if`-statement to be executed only if the condition return `True`, while the third `print()` statement to be executed regardless. How do we tell it to do that?

That's where indentation (a tab character, usually four spaces long) comes into play. In many computer languages, a whitespace like a tab character is simply ignored. However, in Python, it is an *integral part* of the code - **this**

is a major different between Python and other languages! Mistakes made with indentation can and will return an error, so be careful.

Now, coming back to the code. We want the first two `print()` statements to be executed only if the condition is True, so we indent them by one level (one tab character) with respect to the indentation level of the `if`-statement. The additional indentation level makes these lines associated with the parent line. Whereas, we don't want to last `print()` statement to be associated with the `if`-statement, so we keep it at the same indentation level as the statement, signifying that there is no association or dependency between them. This can seem confusing at first, but it is a very elegant way of writing code and is indeed, almost the same as writing pseudo-code.

To drive home the point, let's put an `if`-statement within another `if`-statement:

```
1 x = 6
2 if x<5:
3     print('x is less than 5')
4     print('Finished evaluating outer if block')
5 
6     if x>2:
7         print('but x is greater than 2')
8         print('Finished evaluating inner if block')
9 print('Finished this code block')
```

Now see the indentation here. The inner `if`-statement is one indentation level after the outer one, indicating that it will be executed only if the outer condition is true. And the `print()` statements associated with the inner statement is at yet another indentation level (two tab characters) or one character after the inner `if`-statement, indicating that they will be executed only if the inner statement is true. This indentation, unlike a lot of languages, is **not optional**. Python has no other way of knowing which lines of code are associated with which control statement without proper indentation.

The code above is simple enough, but can actually be combined by using a logical operator to combine both `if`-statements:

```
1 x = 6
2 if x<5 and x>2:
3     print('x is less than 5 and greater than 2')
```

```
4       print('Finished evaluating  if block')
5   print('Finished this code block')
```

Thus, logical operators can be used to write more concise conditional statements, and they can even be nested together into increasingly complex conditions.

It is however common for conditional evaluations to be binary in nature - if something is true, do this thing, or else do this other thing. In principle, this could be achieved by using two `if`-statements one after the other:

```
1   x = 6
2   if x<5:
3       print('x is less than 5')
4   if x>5:
5       print('x is greater than 5')
6   print('Finished this code block')
```

However, this is an unnecessary evaluation: if `x<5` is `False`, it automatically follows that `x>5` is `True` (except for one case, which we'll see in a bit), so there's no need to check again. Thus, Python has an `if-else` statement that implements just that idea:

```
1   x = 6
2   if x<5:
3       print('x is less than 5')
4   else:
5       print('x is greater than 5')
6   print('Finished this code block')
```

Note again how the line after `else` is indented after the colon, signifying an association between the two.

If you're alert, you'll nice that `x==5` is a third possibility. We could add another `if` statement for it, or we could group the three conditions together into an `if-elif-else` block:

```
1   x = 6
2   if x<5:
3       print('x is less than 5')
4   elif x>5:
```

```
5       print('x is greater than 5')
6   else:
7       print('x is equal to 5')
8   print('Finished this code block')
```

Here, "elif" is short for "else if". It is not necessary to use `elif` statements, but it is recommended because it makes the code more readable. Once again, notice the indentation and which lines are associated with which.

Note that the last else statement could've been replaced with an `elif x==5:` statement. However, it is usually good practice to end the `if-elif-else` block with an `else` statement as a fail-safe case to catch all other possible outcomes, both expected and unexpected.

3.2 Loops

Loops are blocks of code that are executed repeatedly either for a fixed number of times, or till a break condition is satisfied. Loop statements offer a convenient way to execute code repeatedly without having to write very long, unwieldy programs. Python offers two types of loops: `while` and `for` loops. Let's look at these individually.

`While` loops execute a block of statements (grouped by indentation) as long as a condition is true. Consider the following code, which prints a number and increments it until it reaches a threshold:

```
1   x = 3
2   while(x<6):
3       print('The value of x is ',x)
4       x+=1
```

Here, `x` is set to 3. When the `while` loop begins, the condition `x<6` is evaluated and if it is true, the associated lines of code (which are indented by one level more than the `while` statement) are executed. In this case, the condition is true, so first the value of `x` is printed, and then it is incremented by 1. At this point, control returns (or 'loops back') to the `while` statement, and the condition is evaluated again. This is why it's called a loop. The loop continues to iterate until the condition is `False`, at which point any code after the loop and its associated lines are executed till the program ends.

`While` loops are used when the number of times the code has to run is unknown. However, a more common case is that the lines of code are

executed for a fixed number of times. The `While` loop could be 'hacked' by coming up a with a condition that runs for a fixed number of times (like the example above), but there's a better way: `for` loops. These types of loops are extremely common, and most programming languages offer them. Compared to many other popular languages though, Python's `for` loop works a little differently: it is an iterator-based loop, similar to the `foreach` loop provided in C# and VBA, or a version of the `for` loop in Java. It is not the same as the `for` loop used in C/C++.

An iterator is a specific type of Python object that allows the program to traverse through all the elements in a sequence (tuples, lists, or dictionaries), regardless of how that sequence is implemented. And since sequences have a fixed length at any point of time in a code, iterating through them is equivalent to running a loop for a fixed number of times. Consider the following block of code:

```
1 x = range(3)
2 print(list(x))
3 for i in x:
4     print('The value of i is ',i)
```

Here, `range()` is a function that returns a fixed number of *integers* (*not floats*), *starting from 0 by default*, although the default behavior can be modified (see documentation). The return type is an iterable `range` object, which can be cast into a list type to see its contents - in this case, it's a list `[0,1,2]`.

Next, the `for` loop defines a variable `i` which is in `x`. This means that the variable `i` traverses through each the value in `x` in each iteration, until it runs out of values to traverse. Execute the function to print `i` and see this for yourself.

The beauty of this approach in Python and perhaps the most startling difference between it and the `for` loop in C/C++ is that the iterator can have any implementation - it can contain anything, not just numbers! Consider this example:

```
1 for i in ['Harry',(0,1,2),1]:
2     print(i)
```

Here, `i` iterates over a list, which itself is composed of three items of three different data types: a string, a tuple (yes, a list can hold a tuple, or

even another list), and an integer. And it does this naturally, without any additional code or typecasting necessary. This is a really big deal!

And it gets better - a single `for` loop can have more than one loop variable. Consider the following example:

```
1 x = [0,1,2]
2 y = [10,11,12]
3 z = [21,22,23]
4 print(list(zip(x,y,z)))
5 for i,j,k in zip(x,y,z):
6     print(i,'+',j,'+',k,'=',i+j)
```

Here, the `zip()` function (see documentation) takes three lists (or technically, iterable objects) *of the same length* (or it truncates to the shortest length) and combines them into an iterable object of tuples, with each tuple having three members. Then, the `for` loop **unpacks** the tuples into the constituent members, which can then be iterated over *simultaneously*. Python also provides several functions to efficiently create complex iterable objects through the itertools package (see documentation). We'll see how to import a package later.

Finally, because the `for` loop is based on an iterator, it can actually be used anywhere in a line of code, and not just in a formal loop. Consider the following piece of code, which creates a list of the squares of the first five whole numbers in just one line:

```
1 x = [i**2 for i in range(5)]
2 print(x)
```

This feature makes writing Python loops particularly elegant and simple.

4 Functions

In just a few pages, we've already learned all the basic building blocks of any Python program. Now, we can move to higher abstractions. These abstractions are useful in making code more efficient, readable, and modifiable, and while they are strictly speaking not necessary write a program, they are used universally to write a *good* program.

The first abstraction is functions. Functions are bits of code that are separated from the main function, but can be called by the main function to perform a specific task. Functions have their own scope, which means that they cannot access variables in the main program, unless those variables explicitly passed to the function by the program. Here's a simple function that simply prints whatever input is passed to it:

```
1  def print_input(x):
2      print(x)
3
4  k = 'Hello'
5  print_input(k)
6
7  j = 'How are you'
8  print_input(j)
9
10 x = 'I am good'
11 print_input(x)
```

Let's break this down. A function is defined with a `def` statement, followed by the name of the function (which generally follows the same rules as variable names), followed by any inputs within parentheses, and finally a colon. The colon, like with loops, implies that items indented below it are part of the function. In this case, the print statement is the only piece of code in the function.

The main program comes after the function definition, without an indentation. This is not necessary - functions can be defined anywhere in the code, but should be defined before their first call. However, it is good practice to define all functions at the beginning of the program. In the main program, a string variable is defined and passed to the function (variables are passed by object reference, which means the value of its memory address is passed, but changing it involves the same principles as mutability).

4.1 Passing Variables

Note that the name of the variable passed to the function (`k, j, m` here), and the name the function uses for it internally (`x` here) could be the same or different. It doesn't matter: the function has its own scope, which means within the function, `x` is the name of the variable, irrespective of whether another variable outside the code has the same name or not. Think of `x` as a local alias for whatever variable is passed to the function. Thus, when the function is called, it receives a copy of variables passed to it, executes its code, and then returns control to the main program. The function itself only has to be defined once and can be called repeatedly. Furthermore, if the function is changes, it only has to be changed once, instead of having to change every instance of the same code.

However, the data type of the variable passed to the function may be important. Consider the following piece of code:

```
1 def add5(x):
2     y = []
3     for i in x:
4         y.append(i+5)
5     return y
6
7 a = ['one','two']
8 b = add5(a)
9 print(b)
```

Here, a function `add5(x)` is defined, which takes in a collection called `x`. Within the function, an empty list `y` is initialized. Then, every value in `x` is iterated over, with a value of 5 added to it, and the new value appended to `y`. Once all values are exhausted, the function returns `y` to the main program with a `return` statement. In the main program, a list of stings `a` is created and passed to the `add5()` function, and the value returned from it is stored in the variable `b`. This won't work however: a string cannot be added to an integer, and so the function will return an error. Thus, in this case, it is important to check what type of data is passed to the function. Change the main program to define `a = range(3)` and see what it returns.

4.2 Multiple Inputs and Outputs

A function can take in any number of variables and also return any number. Consider a function `solve_eqs()` to solve the system of linear equations $ax + by = e$ and $cx + dy = f$, whose solution is $x = (de - bf)/(ad - bc)$ and $y = (af - ce)/(ad - bc)$. The function takes in the constants `a,b,c,d,e,f` and returns the value of `x,y`:

```
1 def solve_eqs(a,b,c,d,e,f):
2     if (a*d-b*c)==0:
3         return 'Error'
4     x = (d*e-b*f)/(a*d-b*c)
5     y = (a*f-c*e)/(a*d-b*c)
6     return (x,y)
7
8 a = solve_eqs(3.2,3.2,7.8,7.8,1.3,2.4)
9 print(a)
```

The function first checks whether a unique solution exists, and if it doesn't, it returns 'Error' to both the expected variables `x,y` (there are better ways to handle this, but it's good enough to get the point). After the first `return` statement is executed, control returns to the main program, irrespective of whether any more lines could have been executed in the function. If a unique solution does exist, the `if` statement's condition is false and the first return statement is not executed. Then, the values of `x,y` are evaluated and returned as a tuple. In the main program, the output from the function is stored and printed in `a`. Try changing the inputs to the function call and see the results.

4.3 Keyword and Default Arguments

There are two more things to keep in mind: keyword arguments and default arguments. Consider the `solve_eqs()` function above: it has a lot of inputs, and the programmer has to input them in exactly the right order. This can get confusing. Fortunately, Python allows the function call to include the the name of the variable (also called an **argument**) being passed. In the code above, change line 8 to the following and see this in action:

```
a = solve_eqs(a=3.2,f=3.2,d=7.8,b=7.8,c=1.3,e=2.4)
```

This method of passing variables is called keyword arguments (where the name of the variable inside the function becomes its keyword in the function call). With this, arguments can be passed in any order without confusion.

Python can also assume some default values for arguments, which will be used if the user does not pass that particular argument. Consider this code:

```
def addnumber(x,n=5):
    y = []
    for i in x:
        y.append(i+n)
    return y

a = range(3)
print(addnumber(a))
print(addnumber(a,n=2))
```

This code defines a function `addnumber()`, which is similar to the `add5()` function defined previously, but takes an extra argument: the number to add to the first variable `n`, which is set to 5 in the function definition. Consider the first function call in the main program: only one variable is passed, so the function will use the default value of `n`. In the second call, both variables are passed, and the function uses the value of `n` passed in the call. Note that variables with default values are usually defined at the end of a function definition, and it is good practice to pass them using keywords to avoid confusion, as shown above.

5 Classes and Objects

The last abstraction that is useful but not necessary is classes, and objects created from those classes. Think of a class as a set of related variables and functions that are logically grouped together, and **an object as an instance of a class**. Let's take a concrete example: a person. What are the characteristics of a person? Their name, age, and height. Let's say the height is in centimeters, and we'd like a function to tell us what it is in feet and inches. These variables and functions can be grouped together into a logical class, and then we can create objects from those classes for every person we have data for. Let's put this into practice below.

5.1 Defining a Class

The `Person` class described above is constructed in the following code, after which an object is created from it:

```
1  class Person:
2      def __init__(self,name,age,height_cm):
3          self.name = name
4          self.age = age
5          self.height_cm = height_cm #in centimeters
6
7      def print_name(self):
8          print(self.name)
9
10     def print_age(self):
11         print(self.age)
12
13     def print_height_cm(self):
14         print(self.height_cm)
15
16     def height_ftin(self):
17         inches = self.height_cm/2.54
18         feet = inches//12
19         inches = inches%12
20         return feet,inches
21
22 adam = Person('Adam Levine',38,190)
```

```
23 adam.print_name()
24 adam.print_age()
25 adam.print_height_cm()
26
27 feet,inches = adam.height_ftin()
28 print('Height is ',feet,' ft and ',
29       "{:.1f}".format(inches),' in')
```

Like a function, a class is defined by prefixing it with a keyword `class` followed by the name of the class and colon. All indented lines after the colon represent the member variables and functions of that class. Within the class, member variables like `gender` and member functions can be defined. A class definition usually contains a special function `__init__()`, which is called a **constructor**. A constructor, as the name suggests, can be used to create an object of a class by creating variables and setting their values (unlike other object-oriented languages, a Python class can have only one constructor). Note that any member function, including a constructor, is defined just like any other Python function, with appropriate indentation for lines associated with that function. However, unlike ordinary Python functions, member functions of a class **must** be passed a `self` argument at the very beginning, as this is necessary for the function to be associated with the object.

In the `Person` class above, the constructor has four arguments: `self, name, age, height_cm`. The last three are self-explanatory, while `self` is a reference to the class itself (or strictly speaking, the object, similar to a `this` pointer in C++). The `self` argument is usually provided to all functions in a class so that they can access the variables in the associated object using the dot (.) operator. Consider the constructor above: it takes the value of the `name` variable passed to it and stores it in a **local** variable, also called `name` (it could've been called something else too). It knows that this is a local variable because it is defined using `self.name`, which implies that it is a member of the object itself. Thus, the dot operator is used to access member variables and functions. A similar operation is performed for the other member variables, `age, height_cm`. Then, four more member functions are defined, which are also self-explanatory. Remember, all members functions must be passed a `self` argument in addition to any other optional arguments, *even if it never uses any local variables*. Similar to regular Python functions, member functions can also return values, like the `height_ftin()` function defined above.

Now that we've defined the class, we can create an object, which is an instance of the class. This means that the object is a specific implementation of a class. In the code above, this is defined similar to how a new variable is defined. An object called `adam` is created and the `Person` class is assigned to it. In this assignment, the constructor of the `Person` class is called, which needs the `name, age, height_cm` variables, which are thus provided. Below that, members functions are used to print the objects's name, age, and height in cm, and to obtain the height in feet and inches. These member functions are accessed using the dot operator on the object, similar to how we used the dot operator on `self` to access members in the definition of the class.

5.2 Inheritance

Classes can seem unwieldy at first, but they are actually very useful in writing well-organized and efficient code. One of the most useful features of classes is called inheritance: the ability of a class to inherit all the members of a parent or *base* class while adding more to itself. This allows for a more granular level of abstraction. Let's see this with a concrete example: consider a student. A student is also a person (and thus has all the features of a person), but they have more features unique to them: GPA and level of study. We can define a class `Student` that inherits all the members of the `Parent` class, but also has these extra members:

```
1  class Student(Person):
2      def __init__(self,name,age,height_cm,GPA,level):
3          Person.__init__(self,name,age,height_cm)
4          self.GPA = GPA
5          self.level = level
6
7      def print_name(self,status='regular'):
8          print(self.name,' STUDENT: ',status)
9
10     def print_GPA(self):
11         print(self.GPA)
12
13     def print_level(self):
14         print(self.level)
15
```

```
16 nicole = Student('Nicole Kidman',19,170,3.6,'freshman')
17
18 nicole.print_name('in absentia')
19 nicole.print_age()
20 nicole.print_height_cm()
21 nicole.print_GPA()
22 nicole.print_level()
23
24 feet,inches = nicole.height_ftin()
25 print('Height is ',feet,' ft and ',
26       "{:.1f}".format(inches),' in')
```

Once again, a class `Student` is defined with the `class` keyword, but this time, the class is also passed the `Person` class so that it inherits from it. The inherited class has all the members of the base class, and can change them if necessary. The constructor is defined as usual, with the necessary variables passed to it and stored in local member variables. However, within the constructor, variables that were inherited from the base class can be initialized using the constructor of the base class, which can be accessed with the dot operator. This avoids having to rewrite lines of code that were already written for the base class' constructor, with code being written only for the variables unique to the child class.

After the constructor, the `Student` class has access to all the member functions of its base class, and thus, they do not have to be rewritten. Three new member functions are created: `print_GPA()` and `print_level()`, which are self-explanatory, and a new `print_name()` that overwrites the function of the same name in the base class by appending the name of the person with the word STUDENT, and accepting a new argument `status` with a specified default value.

Thus, the child class not only has access to the members functions of the base class, but can *also* change them by changing the implementation of the function and also the number of arguments. This ability of functions with the same name to do different things in different situations is called **polymorphism**. Indeed, it would've also been possible to define the `print_name()` function in the base `Person` class *without any implementation at all*, and then allow all child classes to have their own specific implementations of it. Such a class, which only has the name of a function but no implementation, is called an **Abstract Base Class (ABC)**. Polymorphism and ABCs form the basis of modern object-oriented programming.

The definition of the child class is followed by defining an object in the main program, and then all its member functions are called. Try to do this yourself to see the result.

6 Challenge: Basic Python