Introduction to Python, Part 1

Introduction:
Python is an Interpreted language. It uses the CPython Interpreter to compile the Python code to byte code. For a beginner, you don't need to know much about CPython, but you must be aware of how Python works internally.
The philosophy behind Python is that code must be readable. It achieves this with the help of indentation. It supports many programming paradigms like Functional and Object-Oriented programming. You will understand more about these as you read through the article.

The basic question that most beginners have in mind is what a language can do. Here are some of the use-cases of Python:

- Server-side development (Django, Flask)
- Data Science (Pytorch, Tensor-flow)
- Data analysis / Visualisation (Matplotlib)
- Scripting (Beautiful Soup)
- Embedded development

Installation
The first step of learning any programming language is installing it. Python comes bundled with most operating systems nowadays. Use the following command in your terminal to check if Python is available:

```
python3 --version
```

You'll see the following output:

```
Python 3.7.0
```

Note that your version of Python might be different. If you have Python installed and the version is above 3.5.2 then you can skip this section.

For those who don't have Python installed, follow the steps below:

Windows User:

- Go to Python's official website.
- Click on the download button (Download Python 3.8.2) [**Note:** The version may differ based on when you are reading this article]
- Go to the path where the package is downloaded and double-click the installer.
- Check the box indicating to "Add Python 3.x to PATH" and then click on "Install Now".

- Once done you'll get a prompt that "Setup was successful". Check again if python is configured properly using the above command.
- To confirm if Python is installed and configured properly, use the command python3 --version.

Mac User:

- First install xcode from the app store.
- If you want to install Xcode using the terminal, then use the following command:

```
xcode-select --install
```

- After that, we will use the brew package manager to install Python. To install and configure brew, use the following command:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

- Once brew setup is done, use the following command to update any outdated packages:

```
brew update
```

- Use the following command to install Python:

```
brew install python3
```

- To confirm if Python is installed and configured properly, use the command python3 --version.

Linux User:

- To install Python using apt, use the following command:

```
sudo apt install python3
```

- To install the Python using yum, use the following command:

```
sudo yum install python3
```

- To confirm if Python is installed and configured properly, use the command python3 --version.

Python shell:

The shell is one of the most useful tools you'll come across. The Python shell gives us the power to quickly test any concept before integrating it into our application.

Go to the terminal or command line prompt. Enter python3 command and you'll get the following output:

```
➜ python3.7
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26, 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In this tutorial, we will learn some concepts with the help of the python3 shell which you can see above. From now on, whenever I mention **go to the Python shell**, it means that you must use the python3 command.

To learn the remaining concepts, we will create a file called "testing" with the extension .py. To run this file, we will use the following command:

```
python3 testing.py
```

Let's go to the Python shell. Type 10 + 12 after the >>> mark. You'll get the output 22:

```
>>> 10 + 12
22
```

Commenting:

Comments make it easy to write code as they help us (and others) understand why a particular piece of code was written. Another awesome thing about comments is that they help improve the readability of the code.

```
# Stay Safe
```

When you add the above syntax, the Python interpreter understands that it is a comment. Everything after # is not executed.
You may be wondering why you should use comments. Imagine you are a developer and you have been assigned to a huge project. The project has more than a thousand lines of code. To understand how everything works you'll need to go line by line and read through all the code.

What's a better solution than that? Ah-ha! Comments. Comments help us understand why a particular piece of code was written and what it returns or does. Consider it as documentation for every piece of code.

Print:

Other than debugging tools from the editor, the thing which helps developers solve problems most often is a print statement. The print statement is one of the most underrated pieces of syntax in all of programming.

So how does it help in debugging an issue? Well, consider that you have a module, and you want to check the flow of execution to understand or debug it. There are two options. Either you can use a debugger or add a print statement.

It's not always possible to use a debugger. For example, if you are using the Python shell, then a debugger is not available. In such a scenario, print helps us. Another scenario is when your application is running. You can add a print statement that will display in the logs of your application and monitor them in runtime.

Python provides an inbuilt print method with the following syntax:

```
print ("Stay safe...")
```

Indentation:

Another interesting part of this language is indentation. Why? Well, the answer is simple: It makes the code readable and well-formatted. It is compulsory in Python to follow the rules of indentation. If proper indentation is not followed, you'll get the following error:

```
IndentationError: unexpected indent
```

See, even the errors in Python are so readable and easy to understand. At the start, you may be annoyed by the compulsion of indentation. But with the time you'll understand that indentation is a developer's friend.

Variables:
As the name implies, a variable is something that can change. A variable is a way of referring to a memory location used by a computer program.

Well in most programming languages you need to assign the type to a variable. But in Python, you don't need to. For example, to declare an integer in C, the following syntax is used: int num = 5; In Python it's num = 5.
Go to the Python shell and perform the operation step by step:

- Integer: Numerical values that can be positive, negative, or zero without a decimal point.

```
>>> num = 5
>>> print(num)
5
>>> type(num)
<class 'int'>
```

As you can see here, we have declared a num variable and assigned 5 as a value. Python's inbuilt type method can be used to check the type of variable. When we check the type of num we see the output <class 'int'>. For now, just focus on the int in that output. int represents an integer.

- Float: Similar an integer but with one slight difference – floats are a numerical value with a decimal place.

```
>>> num = 5.0
>>> print(num)
5.0
>>> type(num)
<class 'float'>
```

Here we have assigned a number with a single decimal to the num. When we check the type of num we can see it is float.

- String: A formation of characters or integers. They can be represented using double or single quotes.

```
>>> greet = "Hello user"
>>> print(greet)
Hello user
>>> type(greet)
<class 'str'>
```

Here we have assigned a string to greet. The type of greet is a string as you can see from the output.

- Boolean: A binary operator with a True or False value.

```
>>> is_available = True
>>> print(is_available)
True
>>> type(is_available)
<class 'bool'>
```

Here we have assigned a True value to is_available. The type of this variable is boolean. You can only assign **True** or **False**. Remember **T** and **F** should be capital, or it will give an error as follows:

```
>>> is_available = true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

- NoneType: This is used when we don't have the value of the variable.

```
>>> num = None
>>> print(num)
None
>>> type(num)
<class 'NoneType'>
```

Operators:

Look at the image below to see all the arithmetic operators available in Python:

Operator	Name	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus	a % b
**	Exponentiation	a ** b
//	Floor division	a // b
=	Assignment	a = 10
+=	Addition shorthand	a += 2 (same as a = a + 2)
-=	Subtraction shorthand	a -= 2 (same as a = a - 2)
*=	Multiplication shorthand	a *= 2 (same as a = a * 2)
/=	Division shorthand	a /= 2 (same as a = a / 2)
==	Equal	a == b
!=	Not equal	a != b
>	Greater than	a > b
<	Less than	a < b
> =	Greater than or equal to	a >= b
< =	Less than or equal to	a <= b
and	True if both statements are true	a < 5 and b > 10
or	True if either statement is true	a < 5 or b > 10
not	Reverses the result. If result is true then false and vice versa	not(a < 5)

Arithmetic operators

These include addition, subtraction, deletion, exponentiation, modulus, and floor division. Also, the shorthand syntax for some operators.

First, we will declare two variables, a, and b.

```
>>> a = 6 # Assignment
>>> b = 2
```

Let's try our basic arithmetic operations:

```
>>> a + b # Addition
8
>>> a - b # Subtraction
4
>>> a * b # Multiplication
12
>>> a / b # Division
3.0
>>> a ** b # Exponentiation
36
```

To test for other arithmetic operations let's change the value of a and b.

```
>>> a = 7
>>> b = 3
>>> a % b # Modulus
1
>>> a // b # Floor division
2
```

Shorthand arithmetic operations are also available in Python. Refer to the image above to test them out. To print the output of the shorthand operations, use the print statement.

Comparison operators

These include equal to, greater than, and less than.

```
>>> a = 5 # Assign
>>> b = 2 # Assign
>>> a > b # Greater than
True
>>> a < b # less then
False
>>> a == b # Equal to
```

```
False
>>> a >= 5 # Greater than or equal to
True
>>> b <= 1 # Less than or equal to
False
```

Logical operators
These operators include not, and, & or.

```
>>> a = 10
>>> b = 2
>>> a == 2 and b == 10 # and
False
>>> a == 10 or b == 10 # or
True
>>> not(a == 10) # not
False
>>> not(a == 2)
True
```

Conditional Statements:
As the name suggests, conditional statements are used to evaluate if a condition is true or false.

Many times, when you are developing an application you need to check a certain condition and do different things depending on the outcome. In such scenarios conditional statements are useful. If, elif and else are the conditional statements used in Python.

We can compare variables, check if the variable has any value or if it's a Boolean, then check if it's true or false. Go to the Python shell and perform the operation step by step:

Condition Number 1: We have an integer and 3 conditions here. The first one is the if condition. It checks if the number is equal to 10.
The second one is the elif condition. Here we are checking if the number is less than 10.
The last condition is else. This condition executes when none of the above conditions match.

```
>>> number = 5
>>> if number == 10:
...     print("Number is 10")
... elif number < 10:
...     print("Number is less than 10")
... else:
...     print("Number is more than 10")
...
```
Output:

```
Number is less than 10
```
Note: It is not compulsory to check that two conditions are equal in the if condition. You can do it in the elif also.

Condition Number 2: We have a boolean and 2 conditions here. Have you noticed how we are checking if the condition is true? If is_available, then print "Yes, it is available", else print "Not available".

```
>>> is_available = True
>>> if is_available:
...     print("Yes, it is available")
... else:
...     print("Not available")
...
```
Output:

```
Yes, it is available
```
Condition Number 3: Here we have reversed condition number 2 with the help of the not operator.

```
>>> is_available = True
>>> if not is_available:
...     print("Not available")
... else:
...     print("Yes, it is available")
...
```
Output:

```
Yes, it is available
```
Condition Number 4: Here we are declaring the data as None and checking if the data is available or not.

```
>>> data = None
>>> if data:
...     print("data is not none")
... else:
...     print("data is none")
...
```
Output:

```
data is none
```

Condition Number 5: You can also use an inline if in Python. The syntax to achieve this is the following:

```
>>> num_a = 10
>>> num_b = 5
>>> if num_a > num_b: print("num_a is greater than num_b")
...
```

Output:

```
num_a is greater than num_b
```

Condition Number 6: You can also use an inline if else in Python. The syntax to achieve this is the following:

```
expression_if_true if condition else expression_if_false
```

Example:

```
>>> num = 5
>>> print("Number is five") if num == 5 else print("Number is not five")
```

Output:

```
Number is five
```

Conditional Number 7: You can also use nested if-else statements. The syntax to achieve this is the following:

```
>>> num = 25
>>> if num > 10:
...     print("Number is greater than 10")
...     if num > 20:
...         print("Number is greater than 20")
...     if num > 30:
...         print("Number is greater than 30")
... else:
...     print("Number is smaller than 10")
...
```

Output:

```
Number is greater than 10
Number is greater than 20
```

Condition Number 8: You can also use the and operator in a conditional statement. It states if condition1 and condition2 both are true then execute it.

```
>>> num = 10
>>> if num > 5 and num < 15:
...     print(num)
... else:
...     print("Number may be small than 5 or larger than 15")
...
Output:

10
```

As our number is between 5 and 15, we get the output of 10.

Condition Number 9: You can also use the or operator in a conditional statement. It states that if either condition1 or condition2 is true then execute it.

```
>>> num = 10
>>> if num > 5 or num < 7:
...     print(num)
...
Output:

10
```

Are you confused because the value of num is 10 and our second condition states that num is less than 7? So why do we get the output as 10? It's because of the or condition. As one of the conditions matches, it will execute it.

For Loops:

Another useful method in any programming language is an iterator. If you must implement something multiple times, what will you do?

```
print("Hello")
print("Hello")
print("Hello")
```

Well, that's one way to do it. But imagine you must do it a hundred or a thousand times. Well, that's a lot of print statements we must write. There's a better way called iterators or loops. We can either use a for or while loop.

Here we are using the range method. It specifies the range until which the loop should be repeated. By default, the starting point is 0.

```
>>> for i in range(3):
...     print("Hello")
...
Output:
```

```
Hello
Hello
Hello
```

You can also specify the range in this way range(1,3).

```
>>> for i in range(1,3):
...     print("Hello")
...
```

Output:

```
Hello
Hello
```

"Hello" is only printed two times as we have specified the range here. Think of the range as Number on right - Number on left.
Well, you can also add an else statement in the for loop.

```
>>> for i in range(3):
...     print("Hello")
... else:
...     print("Finished")
```

Output:

```
Hello
Hello
Hello
Finished
```

See our loop iterated 3 times (3 - 0) and once that is done it executed the else statement.

We can also nest a for loop inside another for loop.

```
>>> for i in range(3):
...     for j in range(2):
...             print("Inner loop")
...     print("Outer loop")
...
```

Output:

```
Inner loop
Inner loop
```

```
Outer loop
Inner loop
Inner loop
Outer loop
Inner loop
Inner loop
Outer loop
```

As you can see the inner loop print statement executed two times. After that outer loop print statement executed. Again, the inner loop executed two times. So, what is happening here? If you are confused, then consider this to solve it:

- Our Interpreter comes and sees that there is a for loop. It goes down again and checks there is another for loop.
- So now it will execute the inner for loop two times and exit. Once it's finished it knows that outer for loop has instructed it to repeat two more times.
- It starts again and sees the inner for loop and repeats.

Well, you can also choose to pass a certain for loop condition. What does pass mean here? Well, whenever that for loop will occur and the Interpreter sees the pass statement it won't execute it and will move to the next line.

```
>>> for i in range(3):
...     pass
...
```

You will not get any output on the shell.

While loops:

Another loop or iterator available in Python is the while loop. We can achieve some of the same results with the help of a while loop as we achieved with the for loop.

```
>>> i = 0
>>> while i < 5:
...     print("Number", i)
...     i += 1
...
```

Output:

```
Number 0
Number 1
Number 2
Number 3
Number 4
```
Remember whenever you use a while loop it's important that you add an increment statement or a statement that will end the while loop at some point. If not, then the while loop will execute forever.

Another option is to add a break statement in a while loop. This will break the loop.

```
>>> i = 0
>>> while i < 5:
...     if i == 4:
...             break
...     print("Number", i)
...     i += 1
...
```
Output:

```
Number 0
Number 1
Number 2
Number 3
```
Here we are breaking the while loop if we find the value of i to be 4.
Another option is to add an else statement in while loop. The statement will be executed after the while loop is completed.

```
>>> i = 0
>>> while i < 5:
...     print("Number", i)
...     i += 1
... else:
...     print("Number is greater than 4")
...
```
Output:

```
Number 0
Number 1
Number 2
Number 3
Number 4
Number is greater than 4
```
The continue statement can be used to skip the current execution and to proceed to the next.

```
>>> i = 0
>>> while i < 6:
...     i += 1
...     if i == 2:
...             continue
...     print("number", i)
...
```

Output:

```
number 1
number 3
number 4
number 5
number 6
```

User Input:

Imagine you are building a command-line application. Now you must take the user input and act accordingly. To do that you can use Python's inbuilt input method.
The syntax to achieve this is as follows:

```
variable = input("....")
```

Example:

```
>>> name = input("Enter your name: ")
Enter your name: DASA
```

When you use the input method and press enter, you'll be prompted with the text that you enter in the input method. Let's check if our assignment is working or not:

```
>>> print(name)
DASA
```

There it is! It is working perfectly. Here DASA is of the type string.

```
>>> type(name)
<class 'str'>
```

Let's try one more example where we will assign an integer rather than a string and check the type.

```
>>> date = input("Today's date: ")
Today's date: 12
>>> type(date)
<class 'str'>
```

Are you confused? We entered an integer 12 and it's still giving us its type as a string. It's not a bug. It's how input is intended to work. To convert the string to integer we will use typecasting.

Typecasting:

We saw that the input method returns a string for the integer also. Now if we want to compare this output with another integer then we need a way to convert it back to an integer.

```
>>> date_to_int = int(date)
>>> type(date_to_int)
<class 'int'>
```

Here we took the date that we have declared above in the User input section and converted it into the integer using the Python's inbuilt int method. This is called typecasting.
Basically, you can do the following conversion with the help of typecasting:

- integer to string: str()
- string to integer: int()
- integer to float: float()

Note: Conversion from float to integer is also possible.

```
>>> type(date)
<class 'str'>

# Converting from string to float
>>> date_to_float = float(date)
>>> type(date_to_float)
<class 'float'>

# Converting from float to string
>>> date_to_string = str(date_to_float)
>>> type(date_to_string)
<class 'str'>

# Converting from float to integer
>>> date_to_int = int(date_to_float)
>>> type(date_to_int)
<class 'int'>
```

Dictionaries:

Imagine you want to store some user details. So how can you store these details? Yes, we can use variable to store them as follows:

```
>>> fname = "Goodman"
>>> lname = "DASA"
>>> profession = "Club"
```

To access this value, we can do the following:

```
>>> print(fname)
Goodman
```

But is this an elegant and optimized way to access it? The answer is no. To make it more friendly, let's store the data in a key-value dictionary.

What is a dictionary? A dictionary is a collection that is unordered and mutable (i.e., it can be updated).

Following is the format of the dictionary:

```
data = {
        "key" : "value"
}
```

Let's understand the dictionary further by an example:

```
>>> user_details = {
...     "fname": "Goodman",
...     "lname": "DASA",
...     "profession": "Club"
... }
```

How to access a value in a dictionary

We can access the value inside a dictionary in two ways. We will look at both and then debug them to find out which is better.

Method 1: To access the value of fname key from user_details dictionary we can use the following syntax:

```
>>> user_details["fname"]
'Goodman'
```

Method 2: We can also access the value of fname key from user_details dictionary using get.

```
>>> user_details.get("fname")
'Goodman'
```

I know method 1 looks easier to understand. The problem with it occurs when we try to access the data that is not available in our dictionary.

```
>>> user_details["age"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
KeyError: 'age'
```

We get a KeyError which indicates that the key is not available. Let's try the same scenario with method 2.

```
>>> user_details.get("age")
```

We do not get anything printed in our console. Let's debug it further to know why this happened. Assign a variable age to our get operation and we will print it in our console.

```
>>> age = user_details.get("age")
>>> print(age)
None
```

So, when get doesn't find the key it sets the value to None. Because of this, we do not get any error. Now you may be wondering which one is right. Most of the time using method 2 makes more sense, but for some strict checking conditions, we need to use method 1.

How to check if a key exists

You may be wondering how to check if the dictionary has a particular key or not in it. Python provides the built-in method keys() to solve this issue.

```
>>> if "age" in user_details.keys():
...     print("Yes, it is present")
... else:
...     print("Not present")
...
```

We will get the following output:

```
Not present
```

What if we want to check if the dictionary is empty or not? To understand this let's declare an empty dictionary as follows:

```
>>> user_details = {}
```

When we use if-else on a dictionary directly it either returns true if data is present or false if empty.

```
>>> if user_details:
...     print("Not empty")
... else:
...     print("Empty")
...
```

Output:

```
Empty
```

We can also use Python's inbuilt method bool to check if the dictionary is empty or not. Remember bool returns False if the dictionary is empty and True if it is filled.

```
>>> bool(user_details)
False

>>> user_details = {
...     "fname" : "Goodman"
... }
>>> bool(user_details)
True
```

How to update the value of an existing key

So now we know how to get a particular key and find if it exists – but how do you update it in the dictionary?

Declare a dictionary as follows:

```
>>> user_details = {
...     "fname":"Goodman",
...     "lname": "DASA",
...     "profession": "Club"
... }
```

To update the value, use the following syntax:

```
>>> user_details["profession"] = "Club"
>>> print(user_details)
{'fname': 'Goodman', 'lname': 'DASA', 'profession': 'Club'}
```

Updating a value of key in dictionary is same as assigning a value to the variable.

How to add a key-value pair

The next question is how to add a new value to the dictionary? Let's add an age key with a value of 100.

```
>>> user_details["age"] = "100"
>>> print(user_details)
{'fname': 'Goodman', 'lname': 'DASA', 'profession': 'Club', 'age': '100'}
```

As you can see a new key-value is added in our dictionary.

How to remove a key-value pair

To remove a key-value from the dictionary, Python provides an inbuilt method called pop.

```
>>> user_details.pop("age")
'100'
```

```
>>> print(user_details)
{'fname': 'Goodman', 'lname': 'DASA', 'profession': 'Club'}}
```

This removes the age key-value pair from the user_details dictionary. We can also use a del operator to delete the value.

```
>>> del user_details["age"]
>>> print(user_details)
{'fname': 'Goodman', 'lname': 'DASA', 'profession': 'Club'}
```

The del method can also be used to **delete complete dictionary**. Use the following syntax to delete complete dictionary del user_details.

How to copy a dictionary

A dictionary cannot be copied in a traditional way. For example, you cannot copy value of dictA to dictB as follows:

```
dictA = dictB
```

To copy the values, you need to use the copy method.

```
>>> dictB = user_details.copy()
>>> print(dictB)
{'fname': 'Goodman', 'lname': 'DASA', 'profession': 'Club'}
```

Lists:

Imagine you have a bunch of data that is not labeled. In other words, each piece of data doesn't have a key that defines it. So how will you store it? Lists to the rescue. They are defined as follows:

```
data = [ 1, 5, "xyz", True ]
```

A list is a collection of random, ordered, and mutable data (i.e., it can be updated).

How to access list elements

Let's try to access the first element:

```
>>> data[1]
5
```

Wait what happened here? We are trying to access the first element, but we are getting the second element. Why?

Indexing of the list begins from zero. So, what do I mean by this? The indexing of the position of the elements begins from zero. The syntax to access an element is as follows:

```
list[position_in_list]
```

To access the first element, we need to access it as follows:

```
>>> data[0]
1
```

You can also specify a range to access the element between those positions.

```
>>> data[2:4]
['xyz', True]
```

Here, the first value represents the start while the last value represents the position until which we want the value.

How to add an item to a list

To add an item in the list we need to use the append method provided by python.

```
>>> data.append("Hello")
>>> data
[1, 5, 'abc', True, 'Hello']
```

How to change the value of an item

To change the value of an item, use the following syntax:

```
>>> data[2] = "abc"
>>> data
[1, 5, 'abc', True]
```

How to remove an item from a list

To remove an item from a list we can use the Python's inbuilt remove method.

```
>>> data.remove("Hello")
>>> data
[1, 5, 'abc', True]
```

How to loop through a list

We can also loop through the list to find a certain element and operate on it.

```
>>> for i in data:
...     print(i)
...
Output:

1
5
```

```
abc
True
```

How to check if an item exists or not

To check if a particular item exists or not in list we can use if loop as follows:

```
>>> if 'abc' in data:
...     print("yess..")
...
yess..
```

How to copy list data

To copy list data from one list to another we need to use copy method.

```
>>> List2 = data.copy()
>>> List2
[1, 5, 'abc', True]
```

How to check the length of a list

We can also check the length of list using Python's inbuilt len method.

```
>>> len(data)
4
```

How to join two lists

To join two list, we can use the + operator.

```
>>> list1 = [1, 4, 6, "hello"]
>>> list2 = [2, 8, "bye"]
>>> list1 + list2
[1, 4, 6, 'hello', 2, 8, 'bye']
```

What happens if we try to access an element position which is not available in the list? We get a list index out of range error in such a condition.

```
>>> list1[6]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Tuples:

The tuple is a data type which is ordered and immutable (i.e., data cannot be changed).

Let's create a tuple:

```
>>> data = ( 1, 3 , 5, "bye")
>>> data
```

```
(1, 3, 5, 'bye')
```

How to access a tuple element
We can access elements in the tuple the same way as we access them in a list:

```
>>> data[3]
'bye'
```

We can access the index range as follows:

```
>>> data[2:4]
(5, 'bye')
```

How to change a tuple's value
If you are thinking wait – how can we change the value of tuple, then you are right my friend. We cannot change the value of tuple as it is immutable. We get the following error if we try to change the value of a tuple:

```
>>> data[1] = 8
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

There's a workaround available to change the value of a tuple:

```
>>> data = ( 1, 3 , 5, "bye")
>>> data_two = list(data) # Convert data to list
>>> data_two[1] = 8 # Update value as list is mutable
>>> data = tuple(data_two) # Convert again to tuple
>>> data
(1, 8, 5, 'bye')
```

All other methods that we have seen in the list are applicable for the tuple also.

[Note: Once a tuple is created a new value cannot be added in it.].

Sets:
Sets are another data type in Python which are unordered and unindexed. Sets are declared as follows:

```
>>> data = { "hello", "bye", 10, 15 }
>>> data
{10, 15, 'hello', 'bye'}
```

How to access a value

As sets are unindexed, we cannot directly access the value in a set. Thus, to access the value in the set you need to use a for loop.

```
>>> for i in data:
...     print(i)
...

10
15
hello
bye
```

How to change a value

Once the set is created, values cannot be changed.

How to add an item

To add an item to the set python provides an inbuilt method called add.

```
>>> data.add("test")
>>> data
{10, 'bye', 'hello', 15, 'test'}
```

How to check length

To check the length of the set we use the len method.

```
>>> len(data)
5
```

How to remove an item

To remove an item, use the remove method:

```
>>> data.remove("test")
>>> data
{10, 'bye', 'hello', 15}
```

Functions and Arguments:

Functions are a handy way to declare an operation that we want to perform. With the help of functions, you can separate logic according to the operation.

Functions are a block of code that helps us in the reusability of the repetitive logic. Functions can be both inbuilt and user defined.

To declare a function, we use the def keyword. Following is the syntax of the functions:

```
>>> def hello_world():
...     print("Hello world")
```

```
...
```
Here we are declaring a function which, when called, prints a "Hello world" statement. To call a function we use the following syntax:

```
>>> hello_world()
```
We will get the following output:

```
Hello world
```
Remember that the () brackets in a function call means to execute it. Remove those round brackets and try the call again.

```
>>> hello_world
```
You'll get the following output:

```
<function hello_world at 0x1083eb510>
```
When we remove the round brackets from the function call then it gives us a function reference. Here above as you can see the reference of function hello_world points to this memory address 0x1083eb510.

Consider you must perform an addition operation. You can do it by declaring a and b and then performing the addition.

```
>>> a = 5
>>> b = 10
>>> a + b
15
```
This is one way to go. But now consider that the value of a and b have changed, and you need to do it again.

```
>>> a = 5
>>> b = 10
>>> a + b
15
>>> a = 2
>>> b = 11
>>> a + b
13
```
This still looks doable. Now imagine we need to add a set of two numbers a hundred times. The numbers within the set are different for every calculation. That's a lot to do. Don't worry – we have a function at our disposal to solve this issue.

```
>>> def add(a,b):
...     print(a+b)
...
```
Here we are adding a and b as a compulsory argument to the add function. To call this function we will use the following syntax:

```
>>> add(10,5)
Output:
```

```
15
```

See how easy it is to define a function and use it? So, what happens if we don't pass an argument?

```
>>> add()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() missing 2 required positional arguments: 'a' and 'b'
```

Python throws a TypeError and informs us that the function requires two arguments.

Can you guess what will happen if we pass a third argument?

```
>>> add(10,5,1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
        TypeError: add() takes 2 positional arguments but 3 were given
```

Well, Python will inform us that we have passed 3 arguments but there are only 2 positional arguments.

So, what can we do when we don't know how many arguments a function can take? To solve this issue, we use args and kwargs – which we will go through in part 2.