What is Python

- Python is an interpreted, object-oriented, high-level programming language.
- Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.
- Python supports modules and packages, which encourages program modularity and code reuse.
- The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Download Python

- https://www.python.org/downloads/

Pre-requisites

- Indentation
- Declaration of variable not required (automatically derive the data type)

> Python is by default installed on all Linux Servers. For any program to work in your Local Machine/Server, the Runtime Programming Language environment should be installed. As of January 1st, 2020 no new bug reports, fixes, or changes will be made to Python 2, and Python 2 is no longer supported.

Python Shell

- We can use any one of the below Python Shell for understanding of this lannguage
  - Python Online Shell
    - Navigate to Python Online Shell in browser : https://www.python.org/shell/
  - Login to EC2 and Install `python3` using `yum`

```
sudo yum install python3
```

    - To Open Python Interpretor enter

```
python3
```

Values and types

- A value is one of the basic things a program works with, like a letter or a number.
- To get the `type` of the `value`

```
>>> type('Hello, World!')
<class 'str'>

>>> type(17)
<class 'int'>

>>> type('17')
<class 'str'>

>>> type(3.2)
<class 'float'>

>>> type('3.2')
<class 'str'>
```

```
# this is similar to echo in bash
print("Hello world!")
```

- These values belong to different types: `2` is an integer, and 'Hello, World!' is a string, so-called because it contains a “string” of letters.
- You (and the interpreter) can identify strings because they are enclosed in quotation marks.
- A string is a series of characters, surrounded by single or double quotes.

Variables

- Variables are used to store values.
- An assignment statement creates new variables and gives them values:

```
msg = "Hello world!"
type(msg)
print(msg)
```

Concatenation (combining strings)

```
first = 'AWS'
last = 'Devops'
full_name = first + ' ' + last
print(full_name)
```

Comments

- Comments in Python start with the `#` symbol:
- This comment contains useful information that is not in the code:

```
>>> v = 5 # velocity in meters/second.
>>> print(v)
```

Boolean expressions

- A boolean expression is an expression that is either true or false. The following examples use the operator `==` which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

- `True` and `False` are special values that belong to the type bool; they are not strings:

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

- The `==` operator is one of the relational operators; the others are:
    - `x != y`

        - x is not equal to y
    - `x > y`
        - x is greater than y
    - `x < y`
        - x is less than y
    - `x >= y`
        - x is greater than or equal to y
    - `x <= y`
        - x is less than or equal to y

Interpreter Vs Compiler

Interpreter	**Compiler**
Translates program one statement at a time	Scans the entire program and translates it as a whole into machine code.
Take less amount of time to analyze the source code, while the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code, while the overall execution time is comparatively faster than interpreters.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Programming languages like `Python, Javascript, Ruby` use `interpreters`.	Programming languages like `C, C++, Java` use `compilers`.

Python and Java

- Java

```
// Your First Program

class HelloWorld {
    public static void main(String[] args) {
System.out.println("Hello, World inside main function");

    }
    System.out.println("Hello, World! outside main function");
}
```

- To install java use : `sudo yum install java-devel`
- If you have Java installed in you machine/server, you can execute `javac HelloWorld.java` to compile the Java Code.
- This will create a `HelloWorld.class` file the program can be executed with `java HelloWorld`.
- Python follows `indentation` approach

- No Curly brackets in Python

```
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")

print("x is greater than 5 outside if")
```

Conditional Execution

- Conditional statements give us this ability to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly.
- The simplest form is the if statement:

```
x=5
if x > 0:
    print('x is positive')
```

- The boolean expression after if is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens.
- Another Example of `if`

```
age = 21
if age >= 18:
    print("You can vote!")
```

- If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15

print("ticket_price value is",ticket_price)
```

- Create ticket_price.py file
- Write above code in file
- Run it with `python ticket_price.py`

Lists

- A list stores a series of items in a particular order. You access items using an index, or within a loop.
- List are `mutable`
- In a list, first value is at index `0`

```
bikes = ['Apache', 'Suzuki', 'Pulsar']
bikes[1] = 'bmw'
type(bikes)
#Get the first item in a list

first_bike = bikes[0]
first_bike = bikes[1]

bikes[3]
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#IndexError: list index out of range

last_bike = bikes[-1]

len(bikes)

for bike in bikes:
    print(bike)
```

- Adding items to a list

```
bikes = []
bikes.append('Apache')
bikes.append('Suzuki')
bikes.append('Pulsar')

len(bikes)
```

- Slicing a list

```
aws_topics = ['ec2', 's3', 'rds', 'lambda']
first_two = aws_topics[:2]

aws_topics_copy = aws_topics[:]

>>> aws_topics[:]
['ec2', 's3', 'rds', 'lambda']
>>> aws_topics[0:1]
['ec2']
>>> aws_topics[0:2]
['ec2', 's3']
>>> aws_topics[:2]
['ec2', 's3']
```

```
>>> aws_topics[1:3]
['s3', 'rds']

list_var[:2]
```

Tuples

- Tuples are `immutable`
- A tuple is a sequence of values.
- The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists.
- Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

- Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

- Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print(t)
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
>>> print(t[1:3])
```

- But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: 'tuple' object does not support item assignment
```

Dictionaries in Python

- It stores connections between pieces of information. Each item in a dictionary is a `key:value` pair #A simple dictionary

```
s3buckets = {'name': 'mys3bucket', 'numOfObj': 10}
```

- Accessing a value

```
print("The S3 Bucket name is color is " + s3buckets['name'])
```

- Adding a new key-value pair

```
s3buckets['size'] = 0
print(s3bucket)
```

Dictionaries and tuples

- Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair.

```
>>> d = {'name':'mys3bucket', 'numOfObj':10, 'totalSize':200}
>>> list_d = d.items()
>>> type(list_d)
>>> print(list_d)
```

- Combining items, tuple assignment and for, you get the item for traversing the keys and values of a dictionary

```
>>> for key, val in d.items():
...     print(key,val)
```

- Looping through all key-value pairs

```
s3buckets = {'name': 'mys3bucket', 'numOfObj': 10}
for key, value in s3buckets.items():
    print("key is ",key)
    print("value is ",value)
```

- Looping through all keys

```
s3buckets = {'name': 'mys3bucket', 'numOfObj': 10}
for s3key in s3buckets.keys():
    print(s3key)
```

- Looping through all the values

```
s3buckets = {'name': 'mys3bucket', 'numOfObj': 10}
for s3value in s3buckets.values():
    print(s3value)
```

- Your programs can prompt the user for input. All input is stored as a string i.e `<class str>`.

```
name = input("What's your name? ")
print("Hello, " + name + "!")
type(name)
count = input("How many Data Centers are present in NV region?")
count = int(count)

pi = input("What's the value of pi? ")
pi = float(pi)
```

- String in the above input is converted into specific data type using `int()` and `float()` methods.
- A while loop repeats a block of code as long as a certain condition is True.
- A simple while loop

```
current_value = 1
while current_value <= 5:
print(current_value)
current_value += 1
```

- Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
msg = input("Type your message? ")
print(msg)
```

Functions

- Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an `argument`, and information received by a function is called a `parameter`.

Defining a Function

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the structure of a function is finalized, you can execute it by calling it from another function or directly in the Python Script.
- A simple function

```
def greet_user():
"""Display a simple greeting."""
    print("Hello!")
greet_user()
```

Function Arguments

- `Required arguments` are the arguments passed to a function in correct positional order.

```
def greet_user(username):
    print("Hello, " + username + "!")

greet_user('jesse')
```

- Default values for parameters

```
def create_s3_bucket(bktname='mys3bucket'):
    print("Creating a " + bktname + " S3 bucket!")

create_s3_bucket()
create_s3_bucket('news3bucket')
```

- Returning a value

```
def add_numbers(x, y):
    return x + y

sum1 = add_numbers(3, 5)
sum2 = add_numbers(10, 15)
print(sum1)
print(sum2)
```

Math functions

- Python has a math module that provides most of the familiar mathematical functions. A `module` is a file that contains a collection of related functions.
- Before we can use the module, we have to import it:

```
>>> import math
```

- This statement creates a module object named math. If you print the module object, you get some information about it:

```
>>> print(math)
<module 'math' (built-in)>
```

- The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called `dot notation`.

- If you `import math`, you get a module object named `math`. The module object contains constants like `pi` and `functions like sin and exp`.

Python provides two ways to import modules;

- using `dot notation`

```
>>> import math
>>> print(math)
<module 'math' (built-in)>
>>> print(math.pi)
3.14159265359
```

- But if you try to access `pi` directly, you get an error.

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

- As an alternative, you can import an object from a module like this:

```
>>> from math import pi
>>> print(pi)
3.141592653589793
```

- Execute a file in python

```
python filename.py
```

Json module in Python

- **json objects** are surrounded by curly braces { }. They are written in key and value pairs.
- **json.loads()** takes in a string and returns a json object.
- **json.dumps()** takes in a json object and returns a string.

String (JSON) ----------> `json.loads()` -----> json Object (dict)

json Object (dict) -----> `json.dumps()` -----> String

- **Example : json.loads()**

```
import json
x =  '{ "name":"John", "age":30, "city":"New York"}'
print("Type of x is ",type(x))
y = json.loads(x)
print("Type of y is ",type(y))
print(y["age"])
```

- **Example : json.dumps()**

```
import json
a = {"GroupName": "default","GroupId": "sg-32ef414c"}
print("Type of a is ",type(a))
b = json.dumps(a)
print("Type of b is ",type(b))
print (b)
```

Python sys.argv

- The `sys` module provides functions and variables used to manipulate different parts of the Python runtime environment.
- This module provides access to some variables used or maintained by the interpreter, one such variable is `sys.argv` which is a simple list structure. It's main purpose are:
    - It is a list of command line arguments.
    - len(sys.argv) provides the number of command line arguments.
    - sys.argv[0] is the name of the current Python script.

```
import sys

# total arguments
```

```
print(type(sys.argv))
n = len(sys.argv)
print("Total arguments passed:", n)

# Arguments passed
print("\nName of Python script:", sys.argv[0])
# print("\nArguments passed:", end = " ")
print("\nArguments passed:", sys.argv )

for i in range(1, n):
    print(sys.argv[i])
    # print(sys.argv[i], end = " ")

# Addition of numbers
sum_val = 0
# Using argparse module
for i in range(1, n):
    sum_val = sum_val + int(sys.argv[i])

print("\n\nResult:", sum_val)
```

Execute the above code as:

```
python3 add.py 2 4 6 8
```

Reference

> https://docs.python.org/3/py-modindex.html