**Python**

**How python is interpreted?**

**Ans.** Interpreted in simple terms means running code line by line. It also means that the instruction is executed without earlier compiling the whole program into machine language.

**Python is both compiled as well as an interpreted language**. This means when we run a python code, it is first compiled and then interpreted line by line. The compilation part is mostly hidden from the user. While running the code, Python generates a byte code internally, this byte code is then converted using a python virtual machine (p.v.m) to generate the output.

**Q2 What is if \_\_name\_\_ == "\_\_main\_\_"?**

Like other programming languages, Python too has an **execution entry point** i.e. main. A module is a file containing Python definitions and statements. Every module in python has a special attribute called **\_\_name\_\_** . The value of \_\_name\_\_ attribute is set to **'\_\_main\_\_'** when module run as main program.

When you execute a **Python script** , it is treated as the main and its \_\_name\_\_ attribute is set to **"\_\_main\_\_"** . If you import this script as a module in another script, the \_\_name\_\_ is set to the name of the script/module.

By doing the main check, you can have that code only **execute** when you want to run the module as a program and not have it execute when someone just wants to import your **module** and call your functions themselves. Consider the following code for better understanding, it checks if a module is being imported or not.

**Python Example**

print "program started"

if \_\_name\_\_ == "\_\_main\_\_":

print "This is from main module"

else:

print "This is from imported module"

**What is lambda in Python?**

Lambda, the 11th letter of the **Greek alphabet** , is the symbol for wavelength . Lambda comes from the **Lambda Calculus** and refers to anonymous functions in programming.

In Python, Lambda is an **expression** . Lambda's body is a single expression, not a block of statements. Because it is limited to an expression, a **lambda** is less general than a **def** you can only squeeze so much logic into a lambda body without using statements such as if. This is not exactly the same as lambda in functional programming languages, but it is a very powerful concept that's well integrated into **Python** and is often used in conjunction with typical functional concepts like **map()** , **filter()** and **reduce()** . Moreover, Lambda can be used wherever function objects are required.

The general syntax of a lambda function is quite simple:

**lambda argument\_list: expression**

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments.

**def square\_root(x): return math.sqrt(x)**

**square\_root = lambda x: math.sqrt(x)**

**Lambda in Conditional expressions:**

result = lambda x: "Bigger than 100" if x > 100 else "Smaller than 100"

print(result(99))

**Map example using lambda**

My\_list = [2,3,4,5]

Square = map(lambda x: x\*x, my\_list)

print(list(squared))

[4,9,16,25]

### Filter example using lambda

my\_list = [1, 3,5, 7, 9, 11, 13, 15]

new\_list = list(filter(lambda x: (x%3 == 0) , my\_list))

print(new\_list)

### Reduce example using lambda

from functools import reduce

result = reduce((lambda x, y: x \* y), [1, 2, 3, 4,5])

print(result)

ans = 120

**Runtime vs Compile time**

Runtime and compile time are **programming terms** that refer to different stages of software program development. **Compile-time** is the instance where the code you entered is converted to executable while **Run-time** is the instance where the executable is running. The terms "runtime" and "compile time" are often used by programmers to refer to different types of errors too.

Compile-time checking occurs during the compile time. **Compile time errors** are error occurred due to typing mistake, if we do not follow the proper **syntax** and **semantics** of any programming language then compile time errors are thrown by the compiler. They wont let your program to execute a single line until you remove all the syntax errors or until you debug the compile time errors. The following are usual compile time errors:

* Syntax errors
* Typechecking errors
* Compiler crashes (Rarely)

Run-time type checking happens during run time of programs. **Runtime errors** are the errors that are generated when the program is in running state. These types of errors will cause your program to behave unexpectedly or may even kill your program. They are often referred as **Exceptions** . The following are some usual runtime errors:

* Division by zero
* Dereferencing a null pointer
* Running out of memory

**'self' in Python**

The **self** in Python represents the instance of the class. Unlike **this** in C++, "self" is not a **keyword** , it's only a **coding convention** . Often, the first argument of a method is called self. You could give the first parameter of your method any name you want, but you are strongly advised to stick to the convention of calling it self. It binds the **attributes** with the given arguments. The use of self makes it easier to distinguish between **instance attributes** (and methods) from local variables.

You could declare variables within a class without using the **self reference** , but then those variables would be shared by all instances of that class, which may not be what you intended. In the example above, self.age = age and self.name = name are declaring **"instance variables"** (as opposed to "class variables"), the values of which would be unique to the instantiated objects of that class. Otherwise, all students would have the same name and age.

**Debugger**

Python has a **debugger** , which is available as a module called **pdb** . It supports setting conditional **breakpoints** , stepping through the source code one line at a time, stack inspection, and more.

import pdb

msg = "this is a test"

pdb.set\_trace()

print(msg)

Insert **pdb.set\_trace()** anywhere and it will function as a **breakpoint** . When you execute the script by python test.py, you will in the debug mode.

Some useful **debugging** commands:

* b: set a breakpoint
* c: continue debugging until you hit a breakpoint
* s: step through the code
* n: to go to next line of code
* l: list source code for the current file
* u: navigate up a stack frame
* d: navigate down a stack frame
* p: to print the value of an expression in the current context
* q: quit

Running from command line of **python** interpreter.

python -m pdb scriptName.py

If you don't like spending time in debuggers, you can **dump execution** trace and analyse it later.

python -m trace -t scriptName.py install > debug.log

# With statement in Python

In python the **with** keyword is used when working with unmanaged resources (like file streams). The **with statement** simplifies exception handling by encapsulating common preparation and cleanup tasks. This allows common **try..except..finally** usage patterns to be encapsulated for convenient reuse.



It's handy when you have two **related operations** which you'd like to execute as a pair, with a block of code in between. The classic example is **opening a file** , manipulating the file, then closing it:



Using **with statement** , to open a file, process its contents, and make sure to close it, you can simply do:

The **with statement** clarifies code that previously would use try...finally blocks to ensure that clean-up code is executed. The advantage of using a with statement is that it is guaranteed to close the file no matter how the nested block exits. If the **\_\_enter\_\_()** method returns without an error, then **\_\_exit\_\_()** will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. If an **exception** occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler. If the nested block were to contain a return statement, or a continue or **break statement** , the with statement would automatically close the file in those cases, too. Use **with statement** , when ever you acquire resources in your application that must be explicitly relinquished such as files, network connections, locks and the like.

## Mutable Vs. Immutable

The key difference is that tuples are immutable. This means that you cannot change the values in a tuple once you have created it. This is a good feature to have in some data structures where you intend to not make any changes to certain parts. As a list is mutable, it can't be used as a key in a dictionary because dictionaries can use any immutable object as a key. Thus, tuples can be used as dictionary keys if needed. If you try to modify a tuple in a permitted way it becomes two Tuples: the original, which remains for every scope other than yours, and your modified copy for your scope.

## Homogeneous Vs Heterogeneous

There's a strong culture of tuples being for heterogeneous collections, similar to what you'd use structs for in C, and lists being for homogeneous collections, similar to what you'd use arrays for. In other words, different data can be stored in single tuple while same type of data is stored in lists.

## Other Differnces:

* Lists are for variable length , tuples are for fixed length .
* Tuples show structure whereas lists show order .
* Tuples have O(N) append, insert, and delete performance whereas Lists have O(1) append, insert and delete performance.