Python

- `Python` is a **Dynamically Typed** as well as **Strongly Typed** Language.
- `Dynamically Typed` : *Type* tracking is done automatically instead of manually typing the object types (like in `C++`)
- `Strongly Typed` : An object of a particular *type* can only do operations linked to that type (a *string* can only perform string-operations not *list*-operations)
- `Modules` are just packages of additional tools that we `import` to use.
- `strings` are sequences of one-character strings.
- Other more general sequence types include: `lists` and `tuples`.
- In Python, we can also index backwards from end. Positive indexes counts from the left while Negative indexes counts backwards from the right.
- **Immutable Objects** : `numbers`, `strings`, `tuples`
- **Mutable Objects** : `lists`, `dictionaries`, `sets`.

Numeric Literals:

- In **Python 2.X** :
  - Two `integer` types :
    - `normal` : 32 bits
    - `long` : unlimited precison
  - Python automatically converts up to `long` integer type when extra precision is needed
- In **Python 3.X** :
  - Only 1 `integer` type: unlimited precision
- `is` operator tests object identity (i.e. address in memory)
- `lambda` creates un-named functions.

```
# Python Expression Operators and Precedence

yield x             # Generator function send protocol
lambda args : expression     # Anonymous function generation

x if y else z            # Ternary selection (x is evaluated only if y
is True)

x or y                # Logical OR (y is evaluated only if x is False)
x and y             # Logical AND (y is evaluated only if x is True)
not x                 # Logical Negation

x in y                # Membership (iterables, sets)
x not in y            # Membership (iterables, sets)

x is y                # Object identity tests
x is not y            # Object identity test
```

- `Polymorphism` : Meaning of an operation depends on the type of the `object` being operated on.
- **Variables** :
  - Variables are created when they are first assigned values
  - Variables are replaced with values when they are first used in expressions
  - Variables must be assigned before they can be used in expressions
  - **Variables refer to objects and are never declared ahead of time**
- `Classic Division ( X / Y )` : In `Python 2.X`, Truncating results for `integers` and keeping the remainders (i.e. fractional parts) for the `floating-point numbers`
- `True Division ( X / Y )` : In `Python 3.X`, ALWAYS keeping remainders in floating-point results regardless of the types.
- `Floor Division ( X // Y )` : In `Python 2.X & 3.X`, Always truncates fractional remainders down to their floors regardless of the types. Its result type depends on the type of its operands
- `Decimal` are *fixed-precision* floating point values
- For optimization, Python internally *caches* and reuses certain kinds f unchangeable objects, such as small integers and strings.

- Each `object` has two standard header fields : 1. `type designator` & 2. `reference counter`.
- Names have no types. Types live with `objects` not names.
- `object` know what they are; each object contains a header field (refernece counter) that points to the `type` of the object. B'coz objects know what type they are, variables don't have to. Variables just point to the objects.

```
import copy
X1 = copy.copy(Y)         # Makes top-level "shallow" copy of the object Y
X2 = copy.deepcopy(Y)         # MAkes deep-copy of the object Y : copies all nested parts


X == Y                 # tests whether the values in objects referenced by X, Y are same or not
X is Y                 # tests whether the objects(not just values) referenced by X, Y are same or not
```

- `weakref` : Weak-refernce is a reference to an object that doesnot by itself prevent the object from **garbage collected**. If the last remaining reference to an object are weak-reference then the object is automatically garbage collected and the weak-references are deleted (or otherwise notified).
- Python's `string` serve the same role as character-arrays in languages like C/C++, but they
- Objects that are `iterable` return results one at a time, not in a physical list.
- `List comprehension` is not the same as `for` loops because it makes new list object.
- `List comprehensions` run much faster than equivalent `for` loop statements b'coz their iterations are performed at `C`-language speed inside the interpreter rather than with manual python code.
- `readlines()` method loads the `file` object into a list of line strings all at once.
- Any tool that employs the iteration protocol will automatically work on any built-in type or user-defined class that provides it.
- Every built-in tool that scans from left-to-right across objects, uses the iteration protocol.

Command	Details	`return` value
`sorted`	Sorts items in an iterable	**3.X**: An actual `list`
`zip`	Combines items from iterable	**3.X**: Iterable objects
`enumerate`	Pairs items in an iterable with their relative positions	**3.X**: Iterable objects
`filter`	Selects items for which a function is `True`	**3.X**: Iterable objects
`reduce`	Runs pair of items in an iterable through a function	**3.X**: Iterable objects

```
A = zip(*zip(X,Y))          # unzip a zip
```

- Fundamental changes in 3.X than in 2.X :
  - 3.X puts stronger emphasis on `iterators`
  - Unicode model
  - 3.x's mandated new-style classes
- Two ways to make functions:
  - `def`
  - `lambda`
- Two ways to manage `scope` visibility:
  - `global`
  - `nonlocal`
- Two ways to send results back to callers:
  - `return`
  - `yield`
- Functions behave very differently in Python than they do in compiled languages like C.

- Unlike in compiled languages like C; Python `functions` do not need to be fully defined before the program runs.
- `def`s are not evaluated until they are reached and run.
- Code inside `def` is not evaluated until the function is called later.
- Like everything else in Python, functions are just objects.
- Besides calls, functions allow arbitrary attributes to be attached to record information for later use.
- Local variables are removed from memory when the function call exits; and the objects they refrence may be *garbage-collected* if not refrenced elsewhere.
- Each `module` is a self contained `namespace`.
- Functions are objects in Python like everything else and hence can be passed back as `return` values from other functions.
- **`Forward Referencing`**: Its OK to call a function defined after a function that calls it; as long as the seconf `def` runs before the first function is actually called.

```
def f1():
    m_value = 88
    f2(m_value)            #Forward Reference: OK

f1()                    # ERROR: f1 is called before `f2` is defined (as
f2 is called inside f1)

def f2(x):
    print(x)

f1()                    # OK: f1 is called after both f1, f2 are defined
```

- Scopes may nest arbitrarily, but only enclosing function `def` statements(not `class`) are searched when names are referenced.
- Unlike `global`; `nonlocal` names mus already exist in enclosing function's scope when declared -- they can only exist in enclosing `def`s and cannot be created by first assignment in a nested `def`.
- `nonlocal` statements have meaning only inside *functions* ```python def func1(): nonlocal var1, var2, var3 # OK

nonlocal var4 # ERROR (nonlocal statements are valid only inside a function def or lambda)
```
```

- In Python 2.X, references to enclosing `def` scope names are allowed, but not assignment.
- Function attributes allow the state variables to be accessed outside the nested function like class attributes. But with `nonlocal`, state variables can be seen directly only within the nested `def`.