**Functions:** Are objects that can be defined or assigned to a variable name.

def func\_1() – def in the starting defines a function; which is like defining a variable name. Python interpreter does not care about what is inside the function. i.e.

def func\_1():

return func\_2()

def func\_2():

print “running func\_2()”

the above code will work irrespective of func\_2() is defined after it’s been called.

my\_func = func\_4() – my\_func() can be used to call the function func\_4()

try:

try:

<condition>

Except errorcondition:

<do this>

Finally:

<executes always>

Enumerate: first element of enumerate is the index and second is the value. E.g.

S = ‘hello’

For I,c in enumerate (s):

Print(I,c)

h

e

l

l

o

**Reference Counting: How many variables are pointing to the same memory address – hex address**

sys.getrefcount(a) – reference count will be one more than original cause it creates a reference by itself

def ref\_count(address: int)

return ctypes.c\_long.from\_address(address).value

ref\_count(id(a))

This function uses the memory address itself to get the reference count so the overhead of one more reference on the same location is reduced.

**Garbage Collector**

can be controlled programmatically using the gc module

by default, it is turned on

you may turn it off if you’re sure your code does not create circular references – but beware!!

Runs periodically on its own (if turned on)

You can call it manually, and even do your own cleanup

. Variables in python do not have an inherent static type; instead when we call type(my\_var) – python looks up the object my\_var is referencing (pointing to) and return the type of the object at the memory location.

**Object Mutated**

When the internal state of the data pointing to one memory location.

An object whose internal state can be changed, is called Mutable. Eg lists, sets, dictionaries, user-defined classes

An object whose internal state cannot be changed, is known as immutable. Eg numbers (int, float, Booleans, etc), strings, tuples, frozen sets, user-defined classes

**Shared Reference**

Concept of 2 variables referencing the same object in memory (ie having the same memory address)

Python’s memory manager decides to automatically re-use the memory references.

With mutable objects, python memory manager will never create a shared memory reference. Eg list, sets, dictionaries can be only possible for int, string, float, Booleans

**Variable equality**

* is – used for memory reference for variable equality
* == - used for data reference for variable equality; content comparison

**None Object**

* The none object can be assigned to variables to indicate that they are not set (in the way we would expect them to be), i.e. an ‘empty’ value (or null pointer)
* But the none object is a real object that is managed by the python memory manager
* Furthermore, the memory manager will always use a shared reference when assigning a variable as none
* So we can test if a variable is ‘not set’ or ‘empty’ by comparing it’s memory address to memory address of none using ‘is’ operator

**Everything is an object**

* Functions
* Classes
* Types

This means they all have a memory address!

Consequently

* Any object can be assigned to a variable including functions
* Any object can be passed to a function including functions; decorators
* Any object can be returned from a function including functions

**Interning**

Numerical

Reusing objects on-demand

At startup, Python (CPython), pre-loads (caches) a global list of integers in the range [-5,256]

Any time an integer is referenced in that range, Python will use the cached version of that object

Singletons – optimization strategy – small integers show up often

i.e.

a = 10; b = int(10); c = int(‘10’); d=int (‘1001’, base2) – will all point to the same memory address

String

As python code is compiled, identifiers are interned

* Variable names
* Functions names
* Class names
* Etc

Sys.intern() method – force strings to be interned

A= sys.intern(‘quick brown fox’)

B= sys.intern(‘quick brown fox’)

* Dealing with the large number of strings that could have high repetition e.g. tokenizing a large corupus of text (NLP)
* Lots of string comparison (us a is b instead of a == b)

Optimization that can occur at compile time

Constant expression

Numerical expression: eg. 24\*60 = 1440; python will actually precalculate

Short sequences length < 20

(1,2) \* 5 🡪 (1,2,1,2,1,2,1,2,1,2)

‘abc’ \* 5 🡪 abcabcabc

‘hello’ + ‘ world’ 🡪 hello world

But not ‘the quick brown fox’ \* 10 – more than 20 characters

**Integers**

Using 8 bits we can represent all the integers in the range [-127, 127] as 0 does not required sign we end up with [-128, 127]

In a 32-bit OS: memory spaces (bytes) are limited by their address number -> 32 bits

4,294,967,296 bytes of addressable memory = 4 GB

Int object uses a variable number of bits – seamless to us – 16,32,64 bits etc

**Floor Division**

First define the floor of a real number

The floor of a real number a is the largest (in the standard number order) integer <= a

Floor (3.14) = 3

Floor (1.999) = 1

Floor (2) = 2

Negative number:

Floor (-3.1) = -4

**So, floor is not quite the same as truncation!**

A // b = floor (a / b)

a = b \* (a//b) + a%b

Negative numbers:

a//b, is not the integer portion of a/b, it is the florr of a/b; for a>0 and b>0

a -135; b=4

floor = -34 (-135 // 4) remainder = 1 as it should be equal to the equation

**Constructors and bases**

An integer number is an object – an instance of the int class

A = int (10)

A = int (-10)

A = int (10.9) 🡪 truncation a=10

A = int (-10.9) 🡪 truncation a=-10

A = int (True) 🡪 a=1; false a=0

A = int(Decimal(“10.9”)) 🡪 truncation a=10

A = int(“10”) 🡪 string a=10

Number Base

Int(“123”) 🡪 (123)10

2 <=base <=36

Base 10 default

Int (“1010”, 2) 🡪 (10)10

Int(“A12F”, 16) 🡪 (41263)10

Built-in function:

Bin(), Oct(), Hex()