Chapter 1: Getting started with Python Language

Introduction

Reset your brain.

Objects

All Python objects have 3 things:

1. a unique identity (an integer, returned by *id(x)*)
2. a type (returned by *type(x)*)
3. some content

✓ **You cannot change the identity.**

✓ **You cannot change the type.**

✓ **Some objects allow you to change their *content* (without changing the identity or the type, that is).**

✓ **Some objects don't allow you to change their content (more below).** The type is represented by a type object, which knows more about objects of this type (how many bytes of memory they usually occupy, what methods they have, etc).

Section 1.4: Datatypes

Collection data types

List

- `list`: An ordered collection of n values (`n >= 0`)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # allowed
```

Not hashable; mutable.

```
list_of_banks = ["SBI", "ICICI", "CANARA BANK", "SBI", True, 600034, 45525.52,
"SBI"] # print(type(list_of_banks))
print(id(list_of_banks))
print(list_of_banks)
```

Set

- `set`: An unordered collection of unique values. Items must be hashable.

```
a = {1, 2, 'a'}
```

```
set1 = {'name', 'marks'}
```

```
print(set1)
output
 {'marks', 'name'}

set2 = {'name', 'marks',[]}
print(set2)
--------------
set_of_banks = {"SBI", "ICICI", "CANARA BANK", "SBI"}
print(type(set_of_banks))
print(id(set_of_banks))
print(set_of_banks)

----------
```

Tuples

A tuple is similar to a list except that it is fixed-length and immutable. So the values in the tuple cannot be changed nor the values be added to or removed from the tuple. Tuples are commonly used for **small collections of values** that will not need to change, such as an IP address and port. Tuples are represented with parentheses instead of square brackets:

ip_address = ('10.20.30.40', 8080, ‘door number’, ‘pincode’, ‘bankIFSC code’)

```
tuple_of_banks = ("SBI", "ICICI", "CANARA BANK", "SBI", 56.20)
print(type(tuple_of_banks))
```

```
print(tuple_of_banks)
print(id(tuple_of_banks))
```

Dict

- `dict`: An unordered collection of unique key-value pairs; keys must be hashable.

```
a = {1: 'one',
     2: 'two'}

b = {'a': [1, 2, 3],
     'b': 'a string'}
```

> An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equality must have the same hash value.

```
dic = {1:10, 'Name':"John", 'marks':[20,30], ('Marks1',
'Marks2'):(80,90)}
print(dic)
{1: 10, 'Name': 'John', 'marks': [20, 30], ('Marks1',
'Marks2'): (80, 90)}
```

bank_IFSC= {"Name": "SBI", "Branch": "Ngm", "Pincode":600034, "Locker": True, 10 :

1000} item
key
value

_ _ _ _ _

Built-in constants

Built-in constants

In conjunction with the built-in datatypes there are a small number of built-in constants in the built-in namespace:

- `True`: The true value of the built-in type `bool`
- `False`: The false value of the built-in type `bool`
- `None`: A singleton object used to signal that a value is absent.
- `Ellipsis` or `...`: used in core Python3+ anywhere and limited usage in Python2.7+ as part of array notation. `numpy` and related packages use this as a 'include everything' reference in arrays.
- `NotImplemented`: a singleton used to indicate to Python that a special method doesn't support the specific arguments, and Python will try alternatives if available.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x `Version ≥ 3.0`

`None` doesn't have any natural ordering. Using ordering comparison operators (`<`, `<=`, `>=`, `>`) isn't supported anymore and will raise a `TypeError`.

Python 2.x `Version ≤ 2.7`

`None` is always less than any number (`None < -32` evaluates to `True`).

Testing the type of variables

```
a = '123'
print(type(a))
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

Section 1.5: Collection Types

There are a number of collection types in Python. While types such as **int** and **str** hold a single value, **collection types hold multiple values**

Lists

The list type is probably the most commonly used collection type in Python. Despite its name, a list is more like an array in other languages, mostly JavaScript. In Python, a list is merely an **ordered collection of valid Python values**. A list can be created by enclosing values, separated by commas, in square brackets:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

A list can be empty:

```
empty_list = []
```

The elements of a list are not restricted to a single data type, which makes sense given that

Python is a dynamic language:

mixed_list = [1, 'abc', True, 2.34, None]

A list can contain another list as its element:
nested_list = [['a', 'b', 'c'], [1, 2, 3]]

The elements of a list can be accessed via an index, or numeric representation of their position. Lists in Python are zero-indexed meaning that the first element in the list is at index 0, the second element is at index 1 and so on:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

Indices can also be negative which means counting from the end of the list (-1 being the index of the last element).
So, using the list from the above example:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Lists are mutable, so you can change the values in a list:

```
names[0] = 'Ann'
```

```
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

Besides, it is possible to add and/or remove elements from a list:
Append object to end of list with L.append(object), returns None

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Add a new element to list at a **specific index**. L.insert(index, object)

```
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

```
L = ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric',
'Sia'] print(L)
L.remove('Bob')
print(L)
```

Output

```
['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

Remove the first occurrence of a value with L.remove(value), returns

None
```
L = ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Bob','Sia']
print(L)
L.remove('Bob')
print(L)
```
Output
['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Bob', 'Sia']
['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Bob', 'Sia']

Get the index in the list of the first item whose value is x. **It will show an error if there is no such item.**
name.index("Alice")
0

Count length of list
len(names)
6

count occurrence of any item in list
a = [1, 1, 1, 2, 3, 4]
a.count(1)
3

=========

Tuples

A tuple is similar to a list except that it is fixed-length and immutable. So the values in the tuple cannot be changed nor the values be added to or removed from the tuple. Tuples are commonly used for **small collections of values** that will not need to change, such as an IP address and port. Tuples are represented with parentheses instead of square brackets:

ip_address = ('10.20.30.40', 8080)
The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid
Python valid.

A tuple with only one member must be defined (note the comma) this
way: one_member_tuple = ('Only member',)

or

one_member_tuple = 'Only member', # No brackets
or just using **tuple** syntax

one_member_tuple = tuple(['Only member'])

Dictionaries

A dictionary in Python is a collection of key-value pairs. The dictionary is surrounded by curly braces. Each pair is separated by a comma and the key and value are separated by a colon. Here is an example:

state_capitals = {
'Arkansas': 'Little Rock',
'Colorado': 'Denver',
'California': 'Sacramento',
'Georgia': 'Atlanta'
}

To get a value, refer to it by its key:
ca_capital = state_capitals['California']
You can also get all of the keys in a dictionary and then iterate over them

```
state_capitals = {
  'Arkansas': 'Little Rock',
  'Colorado': 'Denver',
  'California': 'Sacramento',
  'Georgia': 'Atlanta',
 'TN': 'Chennai'
}

for k in state_capitals.keys():
 print('{} is the capital of {}' .format(state_capitals[k], k))
```

output
Little Rock is the capital of Arkansas

Denver is the capital of Colorado
Sacramento is the capital of California
Atlanta is the capital of Georgia
Chennai is the capital of TN

Dictionaries strongly resemble JSON syntax. The native json module in the Python standard library can be used to convert between JSON and dictionaries --------

Set

A set is a collection of elements with no repeats and without insertion order. **A set is an unordered collection with no duplicate elements**.. They are used in situations where it is only important that some things are grouped together, and not what order they were
included. For large groups of data, it is much faster to check whether or not an element is in a set than it is to do
the same for a list.

Defining a set is very similar to defining a dictionary:
first_names = {'Adam', 'Beth', 'Charlie'}

Or you can build a set using an existing list:

my_list = [1,2,3]

my_set = set(my_list)

Check membership of the set using in:
if name in first_names:
 print(name)

You can iterate over a set exactly like a list, but remember: the values will be in an arbitrary, implementation defined order

Section 1.7: User Input

To get input from the user, use the input function (note: in Python 2.x, the function is called **raw_input** instead, although Python 2.x has its own version of input that is completely different):

```
name = input("What is your name? ")
# Out: What is your name? Bob
print(name) # Out: Bob
```

Note that the input is always of type str, which is important if you want the user to enter numbers. Therefore, you need to convert the str before trying to use it as a number:

Section 1.8: Built in Modules and Functions

A module is a file containing Python definitions and statements. Function is a piece of code which execute some logic

To check the built in function in python we can use dir(). If called **without an**

argument, return the names in the current scope. Else, return an alphabetized list of names comprising (some of) the attribute of the given object, and of attributes reachable from it.

```
print(dir())# gives the current scope
```

output

['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']

```
print(dir(__annotations__))
print(dir(__builtins__))
```

execute the above and check the answers

To know the functionality of any function, we can use built in function help .

```
help(max)
```

output

Help on built-in function max in **module builtins:**

max(...)

max(iterable, *[, default=obj, key=func]) -> value

max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if

the provided iterable is empty.

With two or more arguments, return the largest argument.

None

Built in modules contains extra functionalities. For example to get square root of a number we need to include math module.

```
import math
print (math.sqrt(4.154))
```

2.0381364036786156

```
import math
print(dir(math))
```

output

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

it seems __doc__ is useful to provide some documentation in, say, functions

```
import math
print(math.__doc__)
output
This module provides access to the mathematical
functions defined by the C standard.
-------
```