PYTHON PROGRAMMING

Customised for **Capital Group**

July, 2022

Trainer: SHOBHIT NIGAM

Email: Mr.Nigam@gmail.com; Ph: (+1) 331-800-1299

This document, codes and exercises discussed in the sessions can also be downloaded from GitHub: https://github.com/shobhit-nigam/capital_candies

Table of Contents

Contents

1. Fun with Python

Let us get started with writing codes in the IDE. The examples discussed here have been executed in anaconda's spyder console, however we will get similar output in any python based IDE/interpreter/framework.

Python prompt as a calculator:
In the cell of spyder console, type the following:

```
4 + 3
```

After pressing shift+enter, the output is visible as follows:

```
7
```

Comments:

Comments in Python are single line comments and start with the hash character.

```
# this is a comment
```

A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Try out the following examples to get a feel of python language:
Type the following in a input cell of console, and press shift+enter to see the output:
input:

```
a = 10
print(a)
```

output:

```
10
```

print() is a function used to print value of any variable
It can also be used to print a string
input:

```
print("hello")
```

output:

```
hello
```

Let us further explore strings & numbers subsequently.

Numbers

Integer numbers (e.g. 8, 10, 22) have type **int**.
Floating point types (fractions, e.g. 5.4, 1.98) have type **float**.
We will see more about numeric data types later.
Operators `+`, `-`, `*` and `/` work just like in most other languages with selective differences

Some examples:
input:

```
20 + 10 * 2
```

output:

```
40
```

input:

```
20/6      #division with float result
```

output:

```
3.3333333333333335
```

input:

```
20//6     #division with int result
          #this is called floor division
```

output:

```
3
```

input:

```
17 % 3
# the % operator returns the remainder of the division
```

output:

```
2
```

With Python, it is possible to use the ** operator to calculate powers
input:

```
9 ** 2  # 9 squared
```

output:

```
81
```

variables

Variables in python can be created anywhere.
Data types are not required to be mentioned.
Data type is picked up based on value assigned to it.

input:

```
var_x = 20                #data type becomes int
var_y = "hyderabad" #data type becomes string
var_z = 4.56         #data type becomes float
```

can check the inherent data type by using the type() function
input:

```
type(var_y)
```

output:

```
str
```

Variable names can start with an underscore or a letter.
Variable names can have an underscore, letter or a number.
Undefined variables (unassigned) will throw an error when used.
'_' is a name of a variable which stores the latest output (whatever has been displayed as output last)
input:

```
var_x + var_z
```

output:

```
24.56
```

input:

```
_
```

output:

```
24.56
```

if '_' is overwritten (assigned a value explicitly) then it loses its default behaviour and becomes a named variable in the current context.

Multiple Assignment

Multiple variables can be assigned different values in a single statement.

input:

```
a, b, c = 4, 5, 7
print(a)
print(b)
print(c)
```

output:

```
4
5
7
```

Multiple variables can be assigned same values in a single statement.

input:

```
d = e = f = 10
print(d)
print(e)
print(f)
```

output:

```
10
10
10
```

Strings

Strings can be enclosed in single quotes ('...') or double quotes ("...") with the same result. Use either single or double quotes, the examples below will give a further clarity.

\ can be used to escape sequences like in other languages

The following are valid examples:

input:

```
'Hyderabad'
```

output:

```
'Hyderabad'
```

input:

```
"Secunderabad"
```

output:

```
'Secunderabad'
```

input:

```
"hyderabad doesn't have a port"
#single quote within double quote
```

output:

```
"hyderabad doesn't have a port"
```

input:

```
'hyderabad doesn\'t have a port' #see the usage of \
```

output:

```
"hyderabad doesn't have a port"
```

input:

```
'"vizag" is a port city'
```

output:

```
'"vizag" is a port city'
```

input:

```
"hyderabad has a \"metro\" service though"
```

output:

```
'hyderabad has a "metro" service though
```

input:

```
'hyd' "era" 'bad'   #auto concatenation
```

output:

```
'hyderabad'
```

As seen in above example, strings written adjacent to each other gets concatenated.
‘+’ can be used to concatenate explicitly and ‘*’ can be used to multiply

input:

```
str_a = 'hyder'
str_b = 'abad'
```

input:

```
str_a + str_b
```

output:

```
'hyderabad'
```

input:

```
str_a + str_b * 3
```

output:

```
'hyderabadabadabad'
```

print() function reads & prints string appropriately
It also attaches a newline at the end by default.
input:

```
vijayawada = 'hot city \n but beautiful'
```

input:

```
vijayawada
```

output:

```
'hot city \n but beautiful'
```

input:

```
print(vijayawada)
```

output:

```
hot city
 but beautiful
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:
input:

```
print("\telangana\nizamabad")
```

output:

```
     elangana
izamabad
```

input:

```
print(r"\telangana\nizamabad")
```

output:

```
\telengana\nizamabad
```

docstrings
String literals can span multiple lines by using triple-quotes: `"""..."""` or `'''...'''`
internally its stored using `\n`
input:

```
docstring1 = """this line
spans
multiple lines """
```

input:

```
docstring1
```

output:

```
'this line \nspans \nmultiple lines '
```

input:

```
print(docstring1)
```

output:

```
this line
spans
multiple lines
```

Docstrings are used for adding descriptions about classes & packages
Some programmers also tend to use it like multi line comments

string index

Strings can be *indexed* (subscripted), with the first character having index 0.
There is no separate character type; a character is simply a string of size one.
input:

```
var_c = "cgroup"
```

input:

```
var_c[0]  # character in position 0
```

output:

```
'c'
```

input:

```
var_c[1]  # character in position 1
```

output:

```
'g'
```

input:

```
var_c[9]  # character in position 9
#this should throw us an error
```

output:

```
---------------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-7-4446c6562fdf>", line 1, in <module>
    var_c[9]

IndexError: string index out of range
```

Indices may also be negative numbers, it counts from right.
(Note that since -0 is the same as 0, negative indices start from -1)

c	g	r	o	u	p
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

input:

```
var_c[-1]
```

output:

```
'p'
```

input:

```
var_c[-4]
```

output:

```
'r'
```

slicing

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

input:

```
var_c[0:2]
# characters from position 0 (included) to 2 (excluded)
```

output:

```
'cg'
```

input:

```
var_c[2:5]
# characters from position 2 (included) to 5 (excluded)
```

output:

```
'rou'
```

input:

```
var_c[:2]
# character from the beginning to position 2 (excluded)
```

output:

```
'cg'
```

input:

```
var_c[4:]
# characters from position 4 (included) to the end
```

output:

```
'up'
```

input:

```
var_c[-2:]
# characters from the second-last (included) to the end
```

output:

```
'up'
```

Earlier we observed that attempting to use an index that is too large resulted in an error; However, out of range slice indexes are handled gracefully when used for slicing:

input:

```
var_c[2:50]
```

output:

```
'roup'
```

input:

```
var_c[-2:-24]
```

output:

```
''
```

input:

```
var_c[10:14]
```

output:

```
''
```

Python strings cannot be changed — they are **immutable**. It means that individual letters of a string can not be replaced.
Therefore, assigning to an indexed position in the string results in an error:
input:

```
var_c[2] = 't'
```

output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-78a8ecf8cb2e> in <module>
----> 1 var_c[2] = 't'

TypeError: 'str' object does not support item assignment
```

The built-in function `len()` returns the length of a string:
input:

```
len(var_c)
```

output:

```
6
```

List

There are a number of *compound* data types or data structures in Python, which are used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

input:

```
list_a = [34, 6, 9, 12, 65, 23, 78]
list_a
```

output:

```
[34, 6, 9, 12, 65, 23, 78]
```

input:

```
print(list_a)
```

output:

```
[34, 6, 9, 12, 65, 23, 78]
```

Like strings, lists can be indexed and sliced.

The data type of list is list, where as the type of individual elements may be different

All slice operations return a new list (shallow copy) containing the requested elements

input:

```
type(list_a)
```

output:

```
list
```

input:

```
type(list_a[2])
```

output:

```
int
```

input:

```
list_a[3]
```

output:

```
12
```

input:

```
list_a[2:4]
```

output:

```
[9, 12]
```

input:

```
list_a[-4:]
```

output:

```
[12, 65, 23, 78]
```

Like strings, lists also support operations like concatenation:
input:

```
list_b = list_a + [11,22,33]
list_b
```

output:

```
[34, 6, 9, 12, 65, 23, 78, 11, 22, 33]
```

Unlike strings, which are immutable, lists are a **mutable** type, i.e. it is possible to change their content:
input:

```
list_c = ['hi', 'hello', 'nice','dp']
```

input:

```
list_c[3] = 'display pic'
list_c
```

output:

```
['hi', 'hello', 'nice', 'display pic']
```

We can also add new items at the end of the list, by using the `append()` *method*. (we will see more about methods later)
The built-in function `len()` also applies to lists.
input:

```
list_c.append('goodday')
list_c
```

output:

```
['hi', 'hello', 'nice', 'display pic', 'goodday']
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:
input:

```
list_d = ['h', 'y', 'd', 'e', 'r', 'a', 'b', 'a', 'd']
list_d
```

output:

```
['h', 'y', 'd', 'e', 'r', 'a', 'b', 'a', 'd']
```

input:

```
list_d[2:5] = ['D', 'E', 'R']
list_d
```

output:

```
['h', 'y', 'D', 'E', 'R', 'a', 'b', 'a', 'd']
```

input:

```
list_d[2:5] = []
list_d
```

output:

```
['h', 'y', 'a', 'b', 'a', 'd']
```

Nesting of lists

Lists can have different data types including lists.
In other words lists can be nested too.

input:

```
list_e = [49 , 'centuries']
list_f = [44.8 , 'average']
player = ['sachin', list_e, list_f]
player
```

output:

```
['sachin', [49, 'centuries'], [44.8, 'average']]
```

input:

```
player[1]
```

output:

```
[49, 'centuries']
```

input:

```
player[1][1]
```

output:

```
'centuries'
```

input:

```
player[1][1][1]
```

output:

```
'e'
```

Dynamic data types

The type of data in python is dynamic in nature and can be changed based on the value that is assigned to it.

input:

```
var_a = 'hello'
type(var_a)
```

output:

```
str
```

input:

```
var_a = 34
type(var_a)
```

output:

```
int
```

input:

```
var_a = [4,7]
type(var_a)
```

output:

```
list
```

So here we have seen var_a taking different data types based on what is stored in it.

So the question here is, does the value keep getting updated in the same location?
No it doesn't, different values (‘hello’, 34, [4,7]) are stored at different locations, however var_a represents different locations based on its assignment.
Unlike C/C++ memory locations can not be accessed & printed, so the above concept can be understood with the id() function.

object and **id()**

Everything in python is internally treated as an object. Integers that we create are actually objects of the class int. Strings are objects of the class str. Even functions are treated as objects, belonging to an inbuilt class ‘function’.
Each of these objects are stored at unique locations, but the memory addresses can’t be displayed. Hence id() function helps us look at the unique identity attached to any object. objects which same value get same id.

input:

```
var_a = 10
id(var_a)
```

output:

```
4371596688
```

input:

```
id(10)
```

output:

```
4371596688
```

input:

```
var_b = 20
id(var_b)
```

output:

```
4371597008
```

As expected var_b has a different id as it's a different variable with a different value. However when var_b is equated to same value as var_a (value 10), var_b gets the same id as var_a as shown below:

input:

```
var_b = 10
id(var_b)
```

output:

```
4371596688
```

So all the objects with the value 10 will have same id in the system.

conversion functions

int() function converts any data to integer

input:

```
var_i = '5678923'
type(var_i)
```

output:

```
str
```

input:

```
var_k = int(var_i)
type(var_k)
```

output:

```
int
```

Likewise str() converts data to string, float() converts data to float.
list() will generate a list.

del

the ***del*** statement is used to delete an object(variable, list, string, function, class…. etc.)
del var_a
will delete the variable named var_a if it exists.
An entire list, or an element in a list can be deleted using ***del***

None

A value exists in the Python interpreter as ***None*** . It means nothing and is similar to void values in C/C++ programming.

When a variable is equated to None, it means that the variable exists but has nothing in it yet. It is different from ***del*** (which deletes the variable's identity and clears the memory for the variable)

Functions which return no values are equivalent to saying functions return ***None***.

Bool and Logical Operator

A boolean expression (or logical expression) evaluates to one of two states true or false. A bool data type exists in python having two values:

True

False

Many functions and operations returns boolean objects. Logical operators can be used anywhere in the code.

input:

```
var_a = 0
var_b = (var_a == 0)
print(var_b)
```

output:

```
True
```

input:

```
type(var_b)
```

output:

```
bool
```

input:

```
type(True)
```

output:

```
bool
```

The not keyword can also be used to inverse a boolean type

The following elements are false:

None

False

0 (whatever type from integer, float to complex)

Empty data structures: "", (), [], {}

Objects from classes that have the special method __nonzero__

Comparison operators

The <, <=, >, >=, ==, != operators compare the values of 2 objects and returns True or False. Comparison depends on the type of the objects.

Comparison operators can be chained

input:

```
var_x = 20
1 < var_x < 40
```

output:

```
True
```

input:

```
20 == var_x < 50
```

output:

```
True
```

input:

```
10 < var_x < 20
```

output:

```
False
```

Logical operators

and

or

not

Membership operators

in evaluates to True if it finds a variable in a specified sequence and false otherwise.

not in evaluates to False if it finds a variable in a sequence, True otherwise.

input:

```
'h' in 'hyderabad'
```

output:

```
True
```

input:

```
'h' not in 'hyderabad'
```

output:

```
False
```

Identity operators

is evaluates to True if the variables on either side of the operator point to the same object and False otherwise

is not evaluates to False if the variables on either side of the operator point to the same object and True otherwise

Bitwise operators

Bitwise operators are used to work on integers in their binary formats.

For example, the and operation between 2 the values 5 and 4 is actually the and operations between 0101 and 0100 binaries. It is therefore equal to 0100

input:

```
4 & 5
```

output:

```
4
```

>>	bitwise left shift
<<	bitwise right shift
&	bitwise and
^	bitwise xor
|	bitwise or
~	bitwise not

2. Flow control

Like many other languages, python has features for controlling the flow of a program and to implement algorithms. Some of these features are:
while
if elif
for
break and continue
else clause on loops

In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false.
The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).

while
The *while loop* is similar in behaviour as in other programming languages.
Consider the following example:
input:

```
var_a = 'india'
i = 0
while i < 5:
    print(var_a[i])
    i = i+1
```

output:

```
i
n
d
i
a
```

The test condition used in the example is a simple comparison (i < 5)

The body of the loop is *indented*: indentation is Python's way of grouping statements.
The indentation was done automatically after a colon (:)
When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

As discussed earlier, print() function adds a new line by default. The keyword argument *end* can be used to avoid the newline after the output. In the following example, newline is replaced by a comma operator using *end* keyword.
input:

```
var_a = 'india'
i = 0
    while i < 5:
        print(var_a[i], end=',')
        i = i+1
```

output:

```
    i,n,d,i,a,
```

if

`if` statements are plaintively simple
input:

```
    var_a = 30

    if var_a == 20:
        print('value is 20')
    elif var_a < 20:
        print('value is greater than 20')
    elif var_a > 20:
        print('value is greater than 20')
    else:
        print('something is not right')
```

output:

```
    value is greater than 20
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation.

An `if` … `elif` … `elif` … sequence is a substitute for the `switch` or `case` statements found in other languages.

for

The `for` statement in Python differs a bit from what you may be used to in other languages. Rather than always iterating over an arithmetic progression of numbers, or giving the user the ability to define both the iteration step and halting condition, Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

input:

```
cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru',
'chennai']

for var_c in cities:
    print(var_c, len(var_c))
```

output:

```
hyderabad 9
delhi 5
mumbai 6
bengaluru 9
chennai 7
```

The for loop has iterated over the entire list, equating var_c to each item during the iteration.

Slicing can also be used to iterate from a particular start point to a particular end point.

input:

```
for var_c in cities[1:4]:
    print(var_c, len(var_c))
```

output:

```
delhi 5
mumbai 6
bengaluru 9
```

range

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions.

input:

```
for i in range(5):
    print(i)
```

output:

```
0
1
2
3
4
```

Range can have a start & end point. The given end point is never part of the generated sequence. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

input:

```
for i in range(3,9):
    print(i)
```

output:

```
3
4
5
6
7
8
```

input:

```
for i in range(3,9,2):
    print(i)
```

output:

```
3
5
7
```

input:

```
for i in range(-1,-10,-3):
    print(i)
```

output:

```
-1
-4
-7
```

To iterate over the indices of a sequence, you can combine range() and len() as follows:

input:

```
cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru',
          'chennai']
for i in range(len(cities)):
    print(i, cities[i])
```

output:

```
0 hyderabad
1 delhi
2 mumbai
3 bengaluru
4 chennai
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

input:

```
cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru',
          'chennai']
for i, var_d in enumerate(cities):
    print(i, var_d)
```

output:

```
0 hyderabad
1 delhi
2 mumbai
3 bengaluru
4 chennai
```

In many ways the object returned by range() behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

iterable

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the *for* statement is such an *iterator*.
The function list() as we say previously is another such iterator; it creates lists from iterables.

input:

```
list_i = list(range(2,6))
list_i
```

output:

```
[2, 3, 4, 5]
```

break, continue, else

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

input:

```
for i in range(0,3):
for j in range(0,3):
          if(i==j):
               print("tea break")
               break
          else:
               print("no tea break")
```

output:

```
tea break
no tea break
tea break
no tea break
no tea break
tea break
```

Unlike other languages, loop statements may have an **else** clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:
input:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
    # loop completed without finding a factor
        print(n, 'is a prime number')
```

output:

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

The ***else*** clause here belongs to the ***for*** loop, not the ***if*** statement

pass

The ***pass*** statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example in below code, ***pass*** does nothing:

input:

```
var_a = 30

if var_a == 20:
    pass
elif var_a < 20:
    pass
elif var_a > 20:
    pass
else:
    print('something is not right')
```

pass is used more for readability rather than any algorithm.

It can be used as a placeholder for a function, class or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level.

3. Functions

The keyword ***def*** introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

input:

```
def func_a():
    print('hello world')
```

Now func_a() can be called anywhere in the program

input:

```
func_a()
```

output:

```
hello world
```

Functions can also return values using keyword return. Unlike most other languages, multiple values can also be returned from a function. ***None*** is returned when nothing is mentioned.

input:

```
def func_b():
    print('returns a float value')
    return 44.55
```

input:

```
var_b = func_b()
print(var_b)
```

output:

```
returns a float value
44.55
```

input:

```
def func_c():
    print('returns multiple values')
    return 3, 5, 9
```

input:

```
var_a, var_b, var_c = func_c()
print(var_b)
```

output:

```
returns multiple values
5
```

The name of the function itself is an object type.

input:

```
def funch():
    print('hello world')
```

input:

```
funch():
```

output:

```
hello world
```

input:

```
funch
```

output:

```
<function funch at 0x10f56c158>
```

input:

```
type (funch)
```

output:

```
<class 'function'>
```

input:

```
f=funch
f()
```

output:

```
hello world
```

input:

```
id(f)
```

output:

```
4552311128
```

input:

```
id(funch)
```

output:

```
4552311128
```

function nesting

Function definitions can be enclosed within another function. The enclosed function can not be called outside but only within the enclosing function.

input:

```
def func_o():
    print("outer 1st statement")
    #some code here
    #-----------------
    def func_i():
        print("in inner function")
    #-----------------
    print("outer 2nd statement")
    #some more code here
    #
    #
    #
    func_i()
    print("outer 3rd statement")
```

input:

```
func_o()
```

output:

```
outer 1st statement
outer 2nd statement
in inner function
outer 3rd statement
```

input:

```
func_i()
```

output:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-32-a5da79d35acf> in <module>
----> 1 func_i()

NameError: name 'func_i' is not defined
```

symbol table

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a ***global*** statement), although they may be referenced.

input:

```
x = 'global'
def outer():
    x = "local"

    def inner():
#         nonlocal x
#         global x
        x = "nonlocal"
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print(x)
```

output:

```
inner: nonlocal
outer: local
global
```

Remove the comments alternatively in the above code to see the behaviour of global, local & nonlocal.

Default arguments

A function can take values from the user, and also have default values for the arguments if the user fails to give any values.

input:

```
def default(var_a = 20, var_b = 30):
    print('var_a = ', var_a)
    print('var_b = ', var_b)
```

input:

```
default()
```

output:

```
var_a =  20
var_b =  30
```

input:

```
default(11)
```

output:

```
var_a =  11
var_b =  30
```

input:

```
default(13,14)
```

output:

```
var_a =  13
var_b =  14
```

Selective arguments can be given values while calling, these are called as keyword arguments. For example var_b becomes the keyword argument here

input:

```
default(var_b = 100)
```

output:

```
var_a =  20
var_b =  100
```

Keyword arguments can be given in any order provided all the keyword arguments passed must match one of the arguments accepted by the function

For example

`default(var_b = 100, var_a = 200)` and

`default(var_a = 200, var_b = 100)` are same.

Please note that positional argument can not follow keyword argument, the following will throw an error

input:

```
default(var_b = 100, 30)
```

output:

```
 File "<ipython-input-70-638a5c2ff929>", line 1
   default(var_b = 100, 30)
                ^
SyntaxError: positional argument follows keyword argument
```

Documentation strings

If a string is written just below the function definition then it automatically becomes a doc string (documentation string). Any string can be used but conventionally a string with triple quotes is used and most programmers tend to call only the triple quote string as document string. It can be accessed by an inbuilt variable called __doc__

input:

```
def func_d():
    """function is used to do something important
    What it does can be documented here
    """
    pass
```

input:

```
print(func_d.__doc__)
```

output:

```
function is used to do something important
    What it does can be documented here
```

4. Data Structures

Python is enriched with multiple data structures (or sequences) in various forms. They are similar to the ‘collections’ of java or ‘STL’ of C++
We already have discussed lists, let us deeper into it and look at some of the functions used with lists.

Lists, extended

list.append(x)
    Add an item to the end of the list. Equivalent to a[len(a):] = [x].
list.extend(iterable)
    Extend the list by appending all the items from the iterable.
    Equivalent to a[len(a):] =iterable.
list.insert(i, x)
    Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x)is equivalent to a.append(x).
list.remove(x)
    Remove the first item from the list whose value is equal to x.
    It raises a ValueError if there is no such item.
list.pop([i])
    Remove the item at the given position in the list, and return it.
    If no index is specified, a.pop() removes and returns the last item in the list.
list.clear()
    Remove all items from the list. Equivalent to del a[:].
list.index(x)
    Return zero-based index in the list of the first item whose value is equal to x.
    Raises a ValueError if there is no such item.
list.count(x)
    Return the number of times x appears in the list.
list.sort()
    Sort the items of the list in place
list.sort(reverse=True) will sort in reverse order
list.reverse()
    Reverse the elements of the list in place.
list.copy()
    Return a shallow copy of the list. Equivalent to a[:].

A few of the functions have been demonstrated below:

input:

```
cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru',
'chennai']
northeast = ['agartala', 'gangtok', 'guwahati', 'shillong']
```

input:

```
cities.append('kolkata')
print(cities)
```

output:

```
['hyderabad', 'delhi', 'mumbai', 'bengaluru', 'chennai',
'kolkata']
```

input:

```
cities.extend(northeast)
print(cities)
```

output:

```
['hyderabad', 'delhi', 'mumbai', 'bengaluru', 'chennai',
'agartala', 'gangtok', 'guwahati', 'shillong']
```

input:

```
cities.index('chennai')
```

output:

```
4
```

input:

```
cities.count('chennai')
```

output:

```
1
```

input:

```
cities.sort(reverse = True)
print(cities)
```

output:

```
['shillong', 'mumbai', 'hyderabad', 'guwahati', 'gangtok',
'delhi', 'chennai', 'bengaluru', 'agartala']
```

List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:
input:

```
squares = []
for x in range(10):
    squares.append(x**2)
```

input:

```
print(squares)
```

output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This can be written in a concise manner using list comprehension:
input:

```
squares = [x**2 for x in range(10)]
```

input:

```
print(squares)
```

output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

For example, this listcomp combines the elements of two lists if they are not equal:
input:

```
[(x, y) for x in [3,4,5] for y in [2,3,4] if x != y]
```

output:

```
[(3, 2), (3, 4), (4, 2), (4, 3), (5, 2), (5, 3), (5, 4)]
```

Tuple

Tuples are similar to list, barring a few differences like:

- declared using parentheses
- immutable
- faster

input:

```
telangana =('hyderabad', 'warangal', 'khammam',
'karimnagar', 'nizamabad')
type(telangana)
```

output:

```
tuple
```

Since tuples are immutable, the tuple telangana can't be appended. The elements in telangana can't be changed, popped etc.
Tuples can also be declared without using parentheses by using comma operator

The following is a valid declaration:
input:

```
telangana ='hyderabad', 'warangal', 'khammam',
'karimnagar', 'nizamabad'
```

However a tuple with a single element can not be created using parentheses, instead has to be created using a trailing comma
input:

```
t1 = ('one')        #this creates a string
type(t1)
```

output:

```
str
```

input:

```
t2 = 'one',         #this creates a tuple
```

output:

```
type(t2)
```

Important functions for a tuple:

all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Return the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.
count(x)	Return the number of items that is equal to x
index(x)	Return index of first item that is equal to x

Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it.

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
A set is created by placing all the items (elements) inside curly braces {}, separated by comma

input:

```
set_a = {1, 2, 3}
print(set_a)
```

output:

```
{1, 2, 3}
```

As mentioned earlier sets can have different data types, but cant have mutable type (example it can not have lists)

input:

```
set_b = {1, 2, 3, 'some string'}
print(set_b)
```

output:

```
{'some string', 1, 2, 3}
```

input:

```
cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru',
'chennai']
set_b = {1, 2, 3, cities}
print(set_b)
```

output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-116-f9c88584d41b> in <module>
----> 1 set_b = {1, 2, 3, cities}
      2 print(set_b)

TypeError: unhashable type: 'list'
```

We get an error in the above code cause it is trying to have a list in a set.

We can add single element using the add() method and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

A particular item can be removed from set using methods, discard() and remove(). The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove() will raise an error in such condition.

Set Operations:

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Union of A and B is a set of all elements from both sets.
Union is performed using | operator or using the method union().
input:

```
set_c = {1, 2, 3, 4, 5}
set_d = {4, 5, 6, 7, 8}
print(set_c|set_d)
```

output:

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Similarly intersection, difference etc can be performed over sets. Following is a list of important functions associated with set.

add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitrary set element. Raise KeyError if the set is empty
remove()	Removes an element from the set. If the element is not a member, raise a KeyError
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others
all()	Return True if all elements of the set are true (or if the set is empty).
any()	Return True if any element of the set is true. If the set is empty, return False.
len()	Return the length (the number of items) in the set.
max()	Return the largest item in the set.
min()	Return the smallest item in the set.
sorted()	Return a new sorted list from elements in the set (does not sort the set itself).
sum()	Return the sum of all elements in the set.

Dictionary

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

Dictionaries are optimized to retrieve values when the key is known.
An item has a key and the corresponding value expressed as a pair, key: value.
While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

input:

```
dict_a = {'thor':'hammer', 'captain':'shield',
'ironman':'suit', 'hawkeye':'arrow'}
```

input:

```
print(dict_a['thor'])
print(dict_a['hawkeye'])
```

output:

```
hammer
arrow
```

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the clear() method.We can also use the del keyword to remove individual items or the entire dictionary itself.

clear()	Remove all items from the dictionary.
copy()	Return a shallow copy of the dictionary.
get(key[,d])	Return the value of key. If key does not exist, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitrary item (key, value). Raises KeyError if the dictionary is empty.
all()	Return True if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries.
sorted()	Return a new sorted list of keys in the dictionary.

5. Files

In Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

open

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

input:

```
f = open("test.txt")
          # open file in current directory
f = open("C:/Python/sample.txt")   # specifying full path
     #full path varies in linux, mac & windows
```

we specify whether we want to read `'r'`, write `'w'` or append `'a'` to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

'r' Open a file for reading. (default)
'w' Open a file for writing.
Creates a new file if it does not exist or truncates the file if it exists.
'x' Open a file for exclusive creation. If the file already exists, the operation fails.
'a' Open for appending at the end of the file without truncating it.
Creates a new file if it does not exist.
't' Open in text mode. (default)
'b' Open in binary mode.
'+' Open a file for updating (reading and writing)

input:

```
f = open("test.txt")       # equivalent to 'r' or 'rt'
f = open("test.txt",'w')   # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

The default encoding differs from platform to platform. open function assumes default text formatting. To be on safer side, including the encoding standard while opening the file is a better way to program.
input:

```
f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

Closing a file will free up the resources that were tied with the file and is done using Python close() method.
input:

```
f.close()
```

write

In order to write into a file in Python, we need to open it in write `'w'`, append `'a'` or exclusive creation `'x'` mode. We need to be careful with the `'w'` mode as it will overwrite into the file if it already exists. All previous data are erased.

```
f = open("test.txt",'w',encoding = 'utf-8')
f.write("This is my first file\n")
f.write("This file\n\n")
f.write("contains three lines\n")
```

read

There are various methods available for reading from a file. We can use the read(size) method to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file.
input:

```
f = open("test.txt",'r',encoding = 'utf-8')
f.read(4) # read the first 4 data
```

output:

```
'This'
```

input:

```
f.read(4)    # read the next 4 data
```

output:

```
' is '
```

input:

```
f.read()     # read in the rest till end of file
```

output:

```
'my first file\nThis file\ncontains three lines\n'
```

input:

```
f.read()  # further reading returns empty string
```

output:

```
''
```

We can change our current file cursor (position) using the seek() method.
Similarly, the tell() method returns our current position (in number of bytes).
input:

```
f.tell()    # get the current file position
```

output:

```
56
```

input:

```
f.seek(0)   # bring file cursor to initial position
```

seek will jump certain bytes (*offset*) from certain (*from*)location (start, current, end)
seek(*offset*,*from*=SEEK_SET)
from is understood as:

0: means your reference point is the **beginning** of the file
1: means your reference point is the **current** file position
2: means your reference point is the **end** of the file

input:

```
print(f.read())  # read the entire file
```

output:

```
This is my first file
This file
contains three lines
```

One can also store the contents of a file into a string.
The readline() function will read only a line of the file.
readlines() method returns a list of remaining lines of the entire file.
input:

```
f.readlines()
```

output:

```
['This is my first file\n', 'This file\n', 'contains three
lines\n']
```

The following are some of the important methods:

close() Close an open file. It has no effect if the file is already closed.

fileno() Return an integer number (file descriptor) of the file.

flush() Flush the write buffer of the file stream.

isatty() Return True if the file stream is interactive.

read(n) Read atmost n characters from the file.
Reads till end of file if it is negative or None.

readable() Returns True if the file stream can be read from.

readline() Read and return one line from the file.
Reads in at most n bytes if specified.

readlines() Read and return a list of lines from the file.
Reads in at most n bytes/characters if specified.

seek(offset,from=SEEK_SET)
Change the file position to offset bytes, in reference to from
(start, current, end).

tell() Returns the current cursor location in the file.

truncate() Resize the file stream to size bytes.
If size is not specified, resize to current cursor location.

write(s) Write string s to the file and return the number of characters written.

writelines() Write a list of lines to the file.

Appendix A: Getting Help

Getting help

Help on python functions can be gained from the console environment by using **help** function. For example to get help on **print** :

input:

```
help(print)
```

output:

```
Help on built-in function print in module builtins:
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
          flush=False)

    Prints the values to a stream, or to sys.stdout by
          default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the
          current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a
          newline.
    flush: whether to forcibly flush the stream.
```

dir() will list all the functions within the modules (inbuilt & user defined)

just typing help() will take us to the inbuilt ‘help console’

input:

```
help()
Welcome to Python 3.7's help utility!
```

output:

```
---
---
---
help>
```

within this console we can type the name of function/identifier or keyword for getting details about it

input:

```
help> print
Help on built-in function print in module builtins:
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
          flush=False)

    Prints the values to a stream, or to sys.stdout by
          default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the
          current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a
          newline.
    flush: whether to forcibly flush the stream.
```

to get help for functions within a module, its required to mention the function name with the module name. For example if we need help for the function ‘getcwd’ from os module then the right way to get help is os.getcwd()

input:

```
help> getcwd
```

output:

```
No Python documentation found for 'getcwd'.
Use help() to get the interactive help utility.
Use help(str) for help on the str class.
```

input:

```
help> os.getcwd
Help on built-in function getcwd in os:
```

output:

```
os.getcwd = getcwd()
    Return a unicode string representing the current
working directory.
```

quit will exit the help console

```
help> quit
```

Help can be used to get help for the customised modules created by the programmers too; the __**doc**__ string is printed when help is called for