Python Getting Started - Learning Notes.

Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby, Scheme, or Java.
This is basic documentation for getting started with python.

Python's features

- Easy syntax. [1]
- easy-to-use language.
- Python interative mode helps test code snippets quickly.
- Can be extended by adding new modules.
- Can embedded into application.
- Runs on most of the platforms `Mac`, `Win`, `Linux`, `Unix`.
- Free to use and include with applications.

Basics Getting Started

Please use the code in the document and execute it along for better understanding.

`print`

The `print()` function prints the given object to the standard output device (screen) or to the text stream file.
Ofcourse the infamous `Hello world`.

```
print('Hello world')
```

`int`, `float`, `str`, `bytes`

```
# int i
# float f
# bool b
# bytes by
# str "string" "Hello!", "what’s up!"

i = 1
f = 2.2222
b = True
by = b"fe"
strs = "string"

print("Int {}, Float {}, bool {}, bytes {}, string {}".format(str(i), str(f), str(b), str(by), strs))
```

Comments

Code comments in `python` are done using the `#` symbol.
For docstring for function documentation we can use `"""`.

```
#
# This is a comment for the string below
#
my_string = "This is a string" # We can add here as well
one_more_string = "This is another string"

#
# Below is an example for docstrings for python functions.
#
def some_functions(parameter1, parameter2, parameter3):
    """[summary]

    Args:
        parameter1 ([type]): [description]
        parameter2 ([type]): [description]
        parameter3 ([type]): [description]

    Returns:
        [type]: [description]
    """
    if parameter1 == parameter3:
        raise ("We have an exception")
    else:
        return parameter1
```

Strings.

Strings in python are created using `"` or multiple line strings with `"""`.

Basic strings

```
my_string = "This is a string"
one_more_string = "This is another string"
long_string_multiline = """We can add multiple lines to a string
like we did here, we user the \"\"\" for this """ # we are using `\` to escape the `"`

print(my_string)                        #>>> This is a string
print(long_string_multiline)            #>>> We can add multiple lines to a string
                                        #    like we did here, we user the """ for this

print("Let me put some string here")    #>>> Let me put some string here
```

String Concatenation

- `+` operator will concat 2 string.
- Different datatypes **cannot** be concatenated ex: `str` and `int`. You will get a `TypeError: Can't convert 'int' object to str implicitly`

```
my_string = "This is a string"
one_more_string = "This is another string"

new_string = my_string + one_more_string
print(new_string) #>>> This is a string This is another string
```

String replication

```
my_string = "This" * 5
print(my_string) #>>> ThisThisThisThisThis
```

Accessing by Index.

String can be traversed similar to `list`. Generic notation is as below.

```
string_process[<start>:<end>:step]
```

- `start`: Index start point (default starts at `0`) ex: `string[::] == string`, `string[::1] == string` [Inclusive od the index]
- `end`: End of the index (default can nothing, which will take it as end of string) ex: string[0::1] == string [Not inclusive of the index mentioned]
- `step`: should be a `int`, can be negative or postive.
    - negative traverses the string backwards.
    - positive traverses the string forward.

```
strings1 = "This is a string, which we can access using index"

# string[<start>:<end>:step]
print(strings1)  # >>> This is a string, which we can access using index
print(strings1[::1])  # >>> This is a string, which we can access using index
print(strings1[0::1])  # >>> This is a string, which we can access using index
# >>> This is a string, which we can access using index
print(strings1[0:len(strings1):1])

# 10th char
print(strings1[10])     # >>> 5

# // 1 to 10 char including spaces [index starts at 0]
print(strings1[1:10])  # >>> his is a

# Last Char
print(strings1[-1])  # >>> x

# Char 10 to 1 in reverse
print(strings1[10:0:-1])  # >>> s a si sih

# Reversing a string is very simple now
print(strings1[::-1])  # >>> xedni gnisu ssecca nac ew hcihw ,gnirts a si sihT

# Start from the end of the string to the start.
# >>> xedni gnisu ssecca nac ew hcihw ,gnirts a si sihT
print(strings1[len(strings1)::-1])
```

Execute on the `python` commandline.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> strings1 = "This is a string, which we can access using index"
>>> # string[<start>:<end>:step]
>>> strings1  # >>> This is a string, which we can access using index
'This is a string, which we can access using index'
>>> strings1[::1]  # >>> This is a string, which we can access using index
'This is a string, which we can access using index'
>>> strings1[0::1]  # >>> This is a string, which we can access using index
'This is a string, which we can access using index'
>>> # >>> This is a string, which we can access using index
>>> strings1[0:len(strings1):1]
'This is a string, which we can access using index'
>>>
>>> # 10th char
>>> strings1[10]     # >>> s
's'
>>>
>>> # // 1 to 10 char including spaces [index starts at 0]
>>> strings1[1:10]  # >>> his is a
'his is a '
>>>
>>> # Last Char
>>> strings1[-1]  # >>> x
'x'
>>>
>>> # Char 10 to 1 in reverse [index 0 not inclusive]
>>> strings1[10:0:-1]  # >>> s a si sih
's a si sih'
>>>
>>> # Reversing a string is very simple now
>>> strings1[::-1]  # >>> xedni gnisu ssecca nac ew hcihw ,gnirts a si sihT
'xedni gnisu ssecca nac ew hcihw ,gnirts a si sihT'
>>>
>>> # Start from the end of the string to the start.
>>> # >>> xedni gnisu ssecca nac ew hcihw ,gnirts a si sihT
>>> strings1[len(strings1)::-1]
'xedni gnisu ssecca nac ew hcihw ,gnirts a si sihT'
```

String methods

String methods let you perform specific tasks for strings. String Methods

- `len()` - Gives length of the string.
- `lower()` - String to all lowercase.
- `upper()` - String to all uppercase.
- `capitalize()` - Capitalize String.
- `title()` - Creates a `title` from string.
- `split()` - default split on `space`, but we can pass demiliter to this. ex: `str.split(',')` will split `str` based on `,`

```
# String methods

string2 = "this is a SECOND string for methods."

print(string2.upper())  # >> THIS IS A SECOND STRING FOR METHODS.
print(string2.lower())  # >> this is a second string for methods.
print(string2.capitalize())  # >> This is a second string for methods.

# Create a Title for the string
print(string2.title())  # >> This Is A Second String For Methods.

# Converts string to list split over `space`
# >> ['this', 'is', 'a', 'SECOND', 'string', 'for', 'methods.']
print(string2.split())

print(len(string2)) #>> 32
```

Execute on the `python` commandline.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└──▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> # String methods
>>> string2 = "this is a SECOND string for methods."
>>>
>>> string2.upper()  # >> THIS IS A SECOND STRING FOR METHODS.
'THIS IS A SECOND STRING FOR METHODS.'
>>> string2.lower()  # >> this is a second string for methods.
'this is a second string for methods.'
>>> string2.capitalize()  # >> This is a second string for methods.
'This is a second string for methods.'
>>>
>>> # Create a Title for the string
>>> string2.title()  # >> This Is A Second String For Methods.
'This Is A Second String For Methods.'
>>>
>>> # Converts string to list split over `space`
>>> # >> ['this', 'is', 'a', 'SECOND', 'string', 'for', 'methods.']
>>> string2.split()
['this', 'is', 'a', 'SECOND', 'string', 'for', 'methods.']
>>>
>>> len(string2) #>> 32
36
>>>
```

Variables assignments

Examples.

- `x = 1.2 + 8 + math.sin(90)` All values are added (since there are int and float values, result `x` is converted to float)

```
import math  # importing math for math.sin()
x = 1.2 + 8 + math.sin(90)
print(x)   #>>> 10.093996663600556
```

- `a = b = c = 0` All variables are set to `0`.

```
a = b = c = 0
print(a, b, c) #>>> 0 0 0
```

- `x, y, z = 1, 2, 3` Variables are assigned in sequence.

```
x, y, z = 1, 2, 3
print(x, y, z) #>>> 1 2 3
```

- `x, y = y, x` swapping values.

```
x, y = 1, 2
print(x, y)  # >>> 1 2
x, y = y, x
print(x, y)  # >>> 2 1
```

Math Operations

Operators	Operation	Example
**	Exponent	2 ** 3 = 8
%	Modulus/Remainder	22 % 8 = 6
//	Integer division	22 // 8 = 2
/	Division	22 / 8 = 2.75
*	Multiplication	3 * 3 = 9
-	Subtraction	5 - 2 = 3
+	Addition	2 + 2 = 4

```python
# Math Operations

a = 2
b = 4

exp = a ** b
print("exp ", exp) #>>> exp  16

add = a + b
print("add ", add) #>>> add  6

sub = a - b
print("sub ", sub) #>>> sub  -2

mul = a * b
print("mul ", mul) #>>> mul  8

int_div = 23 // a
print("int_div ", int_div) #>>> int_div  11

div = 23 / a
print("div ", div) #>>> div  11.5

mod = 23 % a
print("mod ", mod) #>>> mod  1
```

Execute on the `python` commandline.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└──▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> # Math Operations
>>>
>>> a = 2
>>> b = 4
>>>
>>> exp = a ** b
>>> print("exp ", exp) #>>> exp  16
exp  16
>>>
>>> add = a + b
>>> print("add ", add) #>>> add  6
add  6
>>>
>>> sub = a - b
>>> print("sub ", sub) #>>> sub  -2
sub  -2
>>>
>>> mul = a * b
>>> print("mul ", mul) #>>> mul  8
mul  8
>>>
>>> int_div = 23 // a
>>> print("int_div ", int_div) #>>> int_div  11
int_div  11
>>>
>>> div = 23 / a
>>> print("div ", div) #>>> div  11.5
div  11.5
>>>
>>> mod = 23 % a
>>> print("mod ", mod) #>>> mod  1
mod  1
>>>
```

Conversions

- `int("15")` string to int convertion 15
- `int("3f",16)` 63 can specify integer number base in 2nd parameter
- `int(15.56)` float to int, 15 truncate decimal part
- `float("-11.24e8")` string to float `-1124000000.0`
- `round(15.56,1)` `15.6` rounding to 1 decimal (0 decimal integer number)
- `bool(x)` False for null `x`, empty container `x` , None or False `x` ; True for other `x`
- `str(x)` representation string of `x` for display
- `chr(64)` converts to charater '@'
- `ord('@')` 64 code char
  - `ord()` function returns the number representing the unicode code of a specified character
- `repr(x)` literal representation string of `x`

    - The `repr()` function returns a printable representation of the given object.
    - Return a string containing a printable representation of an object.
    - For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object.
    - A class can control what this function returns for its instances by defining a `__repr__` method.
- `bytes([72,9,64])` converts to `b'H\t@'`
- `list("abc")` list convertion `['a','b','c']`
- `dict([(3,"three"),(1,"one")])` converts to dictionary `{1:'one',3:'three'}`
- `set(["one","two"])` convert to set `{'one','two'}`
- separator `str` and sequence of `str` assembled `str`
    - `':'.join(['toto','12','pswd'])` joins all strings in list with `:` delimiter, `'toto:12:pswd'`.
- `str` splitted on whitespaces `list` of `str`
    - `"words with spaces".split()` split all words into list with space separator by default ['words','with','spaces']
    - `"words,with,spaces".split(',')` split all words into list with `,` separater as passed on ['words','with','spaces']
- sequence of one type `list` of another type (via `list comprehension`)
    - `[int(x) for x in ('1','29','-3')]` convert from tuple to list [1,29,-3]

Execute on the `python` commandline.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> # `int("15")`  string to int convertion 15
>>> int("15")
15
>>>
>>> # `int("3f",16)`  63 can specify integer number base in 2nd parameter
>>> int("3f",16)
63
>>>
>>> # `int(15.56)` float to int, 15 truncate decimal part
>>> int(15.56)
15
>>>
>>> # `float("-11.24e8")` string to float `-1124000000.0`
>>> float("-11.24e8")
-1124000000.0
>>>
>>> # `round(15.56,1)`  `15.6` rounding to 1 decimal (0 decimal  integer number)
>>> round(15.56,1)
15.6
>>>
>>> # `bool(x)` False for null `x`, empty container `x` , None or False `x` ; True for other `x`
>>> x = 1
>>> bool(x)
True
>>>
>>> # `str(x)` representation string of `x` for display
>>> str(x)
'1'
>>>
>>> # `chr(64)` converts to charater '@'
>>> chr(64)
'@'
>>>
>>> # `ord('@')` 64 code char / `ord()` function returns the number representing the unicode code of a specified character
>>> ord('@')
64
>>>
>>> # `repr(x)` literal representation string of `x`
>>> #   The `repr()` function returns a printable representation of the given object.
>>> #   Return a string containing a printable representation of an object.
>>> #   For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval, otherwise
>>> #   A class can control what this function returns for its instances by defining a `__repr__` method.
>>> repr(x)
'1'
>>>
>>> # `bytes([72,9,64])` converts to `b'H\t@'`
>>> bytes([72,9,64])
b'H\t@'
>>>
>>> # `list("abc")` list convertion `['a','b','c']`
>>> list("abc")
['a', 'b', 'c']
>>>
>>> # `dict([(3,"three"),(1,"one")])` converts to dictionary `{1:'one',3:'three'}`
>>> dict([(3,"three"),(1,"one")])
{3: 'three', 1: 'one'}
>>>
>>> # `set(["one","two"])` convert to set  `{'one','two'}`
>>> set(["one","two"])
{'two', 'one'}
>>>
>>> # separator `str` and sequence of `str`  assembled `str`
>>> # `':'.join(['toto','12','pswd'])` joins all strings in list with `:` delimiter,  `'toto:12:pswd'`.
>>> ':'.join(['toto','12','pswd'])
'toto:12:pswd'
>>>
>>> # `str` splitted on whitespaces  `list` of `str`
>>> # `"words with spaces".split()` split all words into list with space separator by default ['words','with','spaces']
>>> # `"words,with,spaces".split(',')` split all words into list with `,` separater as passed on ['words','with','spaces']
>>> "words with spaces".split()
['words', 'with', 'spaces']
>>> "words,with,spaces".split(',')
['words', 'with', 'spaces']
>>>
>>> # sequence of one type `list` of another type (via `list comprehension`)
>>> # `[int(x) for x in ('1','29','-3')]`  convert from tuple to list [1,29,-3]
>>> [int(x) for x in ('1','29','-3')]
[1, 29, -3]
>>>
```

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available.
Few of the common getting starte functions are listed below. Complete list of built-in functions is here.

`print()`

The `print()` method prints the given object to the console or to the text stream file.

```
print("Hello World")
```

`input()`

`input()` function is a simple way to prompt the user for some input

```
# The input funciton allows you to prompt the user for a value
# You need to declare a variable to hold the value entered by the user
name = input('What is your name? ')
print("Welcome to Python {}!".format(name))
```

`len()`

Returns length of `string`, `list`, `tuple`, `dictionary`, or any other datatype.

```
my_string = "This"
print(len(my_string)) #>>> 4
```

`filter()`

- Used to filter any contents in an `iterable` object.
- `iterable` object are one which we can be traverse over like `list`, `tuples`, `dictionaries` etc.

`range()`

The `range()` function has two sets of parameters, as follows:

`range(stop)`

- stop: Generate numbers up to, but not including this number starts from `0`.
- `range(3) == [0, 1, 2]`.

`range([start], stop[, step])`

- start: Starting number of the sequence.
- stop: Generate numbers up to, but not including this number.
- step: Difference between each number in the sequence.

```
# list of letters
letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

# Method
def filter_func(letter):
    if letter in ['a', 'i', 'c']:  # Compare `char` with contents of `list`
        return True  # boolean
    else:
        return False  # boolean

filtered_letters = filter(filter_func, letters)
for item in filtered_letters: # looping over iterable object
    print(item)               #>>> a
                              #>>> i
```

Control Flow.

`if` Statements

`if` statement is used for conditional execution or branching.
`if` statement checks for a condition to be true or false and then evaluates it.

```
if <condition>:
    <do something>
```

```
# Since we have set the condition to `True`
# It will always go the first branch.
if True:
    print("I am always true")
```

Note: Indentation in python is very important.
In the below example here the state in the `tab` are part of the `if` evaluation. `statement4` is outside the block.

```
if <condition>:
    <statement1>
    <statement2>
    <statement3>

<statement4>  # not in block
```

`if` - `elif` - `else` statements

```
input_age = input('What is your age? ')
age = int(input_age)

if age < 4:
    ticket_price = 0
    print("FREE!!")
elif age < 18:
    ticket_price = 10
    print("£10 per Ticket")
else:
    ticket_price = 15
    print("£15 per Ticket")
```

Comparators

Comparators check if a value is (or `is not`) `equal to`, `greater than` (or `equal to`), or `less than` (or `equal to`) another value.

Note:

- `==` compares whether two variables are equal.
- `=` is a ssignment operator and it assigns a value to a variable.

Comparators	Description	Results
`==`	Equal to	`2 == 2 >>> True` / `2 == 5 >>> False`
`!=`	Not Equal to	`3 != 2 >>> True` / `2 != 2 >>> False`
`<`	less than	`2 < 5 >>> True` / `5 < 2 >>> False`
`<=`	less than or equal	`2 <= 2 >>> True` / `5 <= 2 >>> False`
`>`	greater than	`5 > 2 >>> True` / `2 > 3 >>> False`
`>=`	greater than or equal	`5 >= 5 >>> True` / `5 >= 6 >>> False`

```
str1 = "One"

if str1 == "One":  # True
    print(True)  # >>> True
else:
    print(False)
```

Boolean Operators

Boolean operators compare statements and result in boolean values. There are three boolean operators:

- `and`, which checks `if` both the statements are `True`;
- `or`, which checks `if` at least one of the statements is `True`;
- `not`, which gives the `opposite` of the statement.

```
str1 = "One"
str2 = "Two"
str11 = "One"
```

```
# Boolean

str1 = "One"
str2 = "Two"
str11 = "One"

if str1 == "One" and str11 == "One":  # True and True
    print(True)  # >>> True
else:
    print(False)

if str1 == "One" or str2 == "One":   # True or False
    print(True)  # >>> True
else:
    print(False)

if not str2 == "One":                # not False ==> True
    print(True)  # >>> True
else:
    print(False)
```

Execute on the `python` commandline.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> str1 = "One"
>>> str2 = "Two"
>>> str11 = "One"
>>>
>>> str1 == "One" and str11 == "One" # True and True
True
>>> str1 == "One" or str2 == "One"   # True or False
True
>>> not str2 == "One"                # not False ==> True
True
>>>
```

Data structure / Containers

- **List** - Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.
- **Tuple** - Tuples are **immutable**, and usually contain a **heterogeneous sequence of elements** that are accessed via unpacking or indexing (or even by attribute in the case of `namedtuples`.
- **Sets** - A set is an unordered collection with no duplicate elements.
- **Dictionaries** - dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

`list`

Lists are used to store multiple items in a single variable.

usage.

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
```

Similar to string we can also process list using their indexes.
Execute on the `python` commandline.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> # fruits[<start>:<end>:step]
>>>
>>> fruits # List all the items in the list
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>>
>>> fruits[::1] # Traverse the list one at a time till the end same as `fruits`
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>>
>>> fruits[0::1] # From index 0 to end, 1 step at a time
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>>
>>> fruits[0::2]  # From index 0 to end, 2 step at a time
['orange', 'pear', 'kiwi', 'banana']
>>>
>>> fruits[0:len(fruits):1] # Start to finish single step
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>>
>>> fruits[4] # centents in index 4, index starts at 0.
'kiwi'
>>>
>>> fruits[1:4] # Index 1 to 4 (non inclusive index 4), default step of 1.
['apple', 'pear', 'banana']
>>>
>>> fruits[-1] # Last item in the list
'banana'
>>>
>>> fruits[5:0:-1] # Traverse reverse based on step -1, from item index 5(inclusive) to 0 (not inclusive).
['apple', 'kiwi', 'banana', 'pear', 'apple']
>>>
>>> fruits[::-1] # Reverse the list
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>>
>>> fruits[len(fruits)::-1] # reverse the string from end of the list (we get this by len(list)) to start (empty), step -1.
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>>
```

Conditional test with lists

Lets says we have list of `motorcycles` = `['yamaha', 'ducati', 'kawasaki', 'ktm']`
we can check if the string `item` is present in the list, we can use the `in` or `not in` to check it.

```
motorcycles = ['yamaha', 'ducati', 'kawasaki', 'ktm']
if 'ducati' in motorcycles:     #>>> This statement is `True` as the item is present in the list.
    print("We have it in the list")

if 'indian' not in motorcycles: #>>> This statement is `True` (rather `!False`) as the item is not present in the list.
    print("Sorry we dont.")
```

Operations on `list`

- **`list.append(x)`**

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

```
# list.append(x) # Add an item to the end of the list.
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.append('pineapple')
print(fruits)

# list.append(x) # Equivalent to a[len(a):] = [x].
fruits[len(fruits):] = ['pen-pineapple-apple-pen']
print(fruits)
```

- **`list.extend(iterable)`**

Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.extend(['another-pineapple'])
print(fruits)

# Output
>>> ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'pineapple', 'another-pineapple']
```

- **`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)`.

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.insert(0, 'another-pineapple')
print(fruits)

# Output
# >>> [ 'another-pineapple', 'orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'pineapple']

fruits.insert(len(fruits), 'another-pineapple')
print(fruits)

# Output
# >>> [ 'another-pineapple', 'orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'pineapple', 'another-pineapple']
```

- **`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

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.remove('orange')
print(fruits)

# Output
# >>> [ 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'pineapple']
```

- **`list.pop(i)`** / **`list.pop()`**

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**.

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.pop(0)
print(fruits)

# Output
# >>> [ 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'pineapple']

fruits.pop()
print(fruits)

# Output
# >>> [ 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
```

- **`list.clear()`**

Remove all items from the list. Equivalent to del `a[:]`.

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.clear()
print(fruits)

# Output
# >>> []
```

- **`list.index(x[, start[, end]])`**

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.

The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.

```python
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.index('apple'))
# Output
# >>> 1

print(fruits.index('pear'))
# Output
# >>> 2

print(fruits.index('banana'))
# Output
# >>> 3

print(fruits.index('banana', 4)) # Find next banana starting a position 4
# Output
# >>> 3
```

- `list.count(x)`

Return the number of times x appears in the list.

```python
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple'))
# Output
# >>> 2

print(fruits.count('pear'))
# Output
# >>> 1

print(fruits.count('banana'))
# Output
# >>> 2
```

- `list.sort(*, key=None, reverse=False)`

Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

Note: not all types can be sorted. `[ None, 1, 'some string']` as we cannot compare `None`, `string` and `int`.

```python
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.sort()
print(fruits)

# Output
# >>> ['apple', 'apple', 'banana', 'banana', 'kiwi', 'orange', 'pear']
```

- `list.reverse()`

Reverse the elements of the list in place.

```python
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.reverse()
print(fruits)

# Output
# >>> ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
```

- `list.copy()`

Return a shallow copy of the list. Equivalent to `a[:]`.

```python
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
new_fruits = fruits.copy()
print(new_fruits)

# Output
# >>> ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
```

`tuple`

Tuples are immutable, and usually contain a heterogeneous sequence of elements.

```
t = 12345, 54321, 'hello!'
print(t[0])
print(t)
u = t, (1, 2, 3, 4) # Tuples may be nested:
print(u)

# Output
#>>>  12345
#>>>  (12345, 54321, 'hello!')
#>>>  ((12345, 54321, 'hello!'), (1, 2, 3, 4))
```

A special problem is the construction of tuples containing 0 or 1 items.

- Empty tuples are constructed by an empty pair of parentheses.
- a tuple with one item is constructed by following a value with a comma.

For example:

```
empty = ()
singleton = 'hello',
print(len(empty))
print(len(singleton))
print(singleton)

# output
#>>> 0
#>>> 1
#>>> ('hello',)
```

Convert Tuple to a List

```
x = ("apple", "orange", "pear")
y = list(x)   # tuple to list Conversions
print(y)      #>>> ['apple', 'orange', 'pear']
x = tuple(y)  # list to tuple Conversions
print(x)      #>>> ("apple", "orange", "pear")
```

`set`

A set is an **unordered** collection with **no duplicate** elements.

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)             #>>> {'apple', 'orange', 'banana', 'pear'}
print("orange" in basket)  #>>> True
```

`dict` - Dictionaries

Dictionaries are key value pairs (`maps`).

- `countries["GB"]` - Return the value from the dictionary d that has the key `"GB"`
- `countries.get("AU","Sorry")` - Return the value from the dictionary d that has the key `"AU"`, or the string `"Sorry"` if the key - "AU" is not found in `countries`
- `countries.keys()` - Return a `list` of the keys from `countries`
- `countries.values()` - Return a `list` of the values from `countries`
- `countries.items()` - Return a `list` of `(key, value)` pairs

```
# Create a dictionary `countries` with keys of "CA", "GB", and "IN"
#   and corresponding values of of "Canada", "Great Britain", and "India"
countries = {"CA": "Canada", "GB": "Great Britain", "IN": "India"}
print(countries["GB"])               #>>> Great Britain
print(countries.get("AU", "Sorry"))  #>>> Sorry
print(countries.keys())              #>>> dict_keys(['CA', 'GB', 'IN'])
print(countries.items())             #>>> dict_items([('CA', 'Canada'), ('GB', 'Great Britain'), ('IN', 'India')])
print(countries.values())            #>>> dict_values(['Canada', 'Great Britain', 'India'])
```

Few more examples.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Looping Technique for Dictionaries

We are taking both key and value at the same time.

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for key, value in knights.items():
...     print(key, value)
...
gallahad the pure
robin the brave
```

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for key in knights:   # This will only give you keys
...     print(key)        # Prints keys
...
gallahad
robin
```

Loops

There are two types of loops in Python, `for` and `while`.

`for`

The Python `for` statement iterates over the members of a sequence in order, executing the block each time.
Example: Traversing a `list`.

```
list_a = ['a', 'b', 'c', 'd']
for item in list_a:
    print(item)

# output
#>>> a
#>>> b
#>>> c
#>>> d
```

Example: Traversing a `dict`

```
┌─[Zubair AHMED][AHMEDZBYR-WRK-HORSE][/d/python/]
└─▪ python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Example: Traversing a 2d `list`.

```
# [
#   [1, 2, 3],
#   [4, 5, 6],
#   [7, 8, 9]
# ]
#
list_of_lists = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list in list_of_lists:
    for x in list:
        print(x)
```

`while`

Printing sequence of numbers.

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1

# output
#>>> 1
#>>> 2
#>>> 3
#>>> 4
#>>> 5
```

Message till we enter `quit`.

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Modules / Imports

- import os
- import numpy as nm
- from `<package>` import `<module>` as `user_defined_name`

```
from os import mkdir as create_directory
# windows
create_directory("C:\\Users\\ahmedzbyr\\test")

# linux
create_directory("/home/ahmedzbyr/test")
```

Exception Errors Handling

The idea of the `try-except` block is this: more details

- **try**: the code with the exception(s) to catch. If an exception is raised, it jumps straight into the except block.
- **except**: this code is only executed if an exception occured in the try block. The except block is required with a try block, even if it contains only the pass statement.

It may be combined with the `else` and `finally` keywords.

- **else**: Code in the else block is only executed if no exceptions were raised in the try block.
- **finally**: The code in the finally block is always executed, regardless of if a an exception was raised or not.

`try except`

```
try:
    <do something>
except Exception:
    <handle the error>
```

Example:

```
try:
    1 / 0
except ZeroDivisionError:
    print('Divided by zero')

print('Should reach here')
```

Multi-level

```
try:
    # your code here
except FileNotFoundError:
    # handle exception
except IsADirectoryError:
    # handle exception
except:
    # all other types of exceptions

print('Should reach here')
```

`try Finally`

```
try:
    <do something>
except Exception:
    <handle the error>
finally:
    <cleanup>
```

If a file is opened then we keep all the cleanup in the `finally`.

```
try:
    f = open("test.txt")
except:
    # When an exception occurs
    print('Could not open file')
finally:
    # This is called all the time regardless of the exception
    f.close()

print('Program continue')
```

`try else`

The else clause is executed if and only if no exception is raised. This is different from the finally clause that’s always executed.

```
try:
    x = 1
except:
    print('Failed to set x')
else:
    print('No exception occured')
finally:
    print('We always do this')
```

`raise` Exception

Exceptions are raised when an error occurs. But in Python you can also force an exception to occur with the keyword `raise`.

```
>>> raise MemoryError("Out of memory")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError: Out of memory
```

Built-in Exceptions.

A list of Python's Built-in Exceptions is shown below. This list shows the Exception and why it is thrown (raised).
Few of the exeception are listed here for more detailed list go here: `https://docs.python.org/3/library/exceptions.html`

Exception	Cause of Error
AssertionError	Raised when an `assert` statement fails.fails.
AttributeError	Raised when an attribute reference or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)fails.
EOFError	Raised when the input() function hits an end-of-file condition (EOF) without reading any data.
GeneratorExit	Raised when a `generator` or `coroutine` is closed;
ImportError	Raised when the import statement has troubles trying to load a module.
IndexError	Raised when a sequence subscript is out of range.
KeyError	Raised when a mapping (dictionary) key is not found in the set of existing keys.
KeyboardInterrupt	Raised when the user hits the interrupt key (normally `Control-C` or `Delete`).
MemoryError	Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects).
NameError	Raised when a `local` or `global` name is not found.
NotImplementedError	by abstract methods.
OSError	if system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
RuntimeError	if an `error` does not fall under any other category.
StopIteration	by `next()` function to indicate that there is no further item to be returned by iterator.
SyntaxError	by parser if syntax `error` is encountered.
IndentationError	if there is incorrect indentation.
TabError	if indentation consists of inconsistent tabs and spaces.
SystemError	if interpreter detects internal error.
SystemExit	by sys.exit() function.
TypeError	if a function or operation is applied to an object of incorrect type.
ZeroDivisionError	if second operand of `division` or `modulo` operation is zero.

Files Operations

File operation are used to process files in python. `open()` is one of the build in functions which is used to open files.

`open()`

Open file and return a corresponding file object. If the file cannot be opened, an OSError is raised. See Reading and Writing Files for more examples of how to use this function.

Character	Meaning
`r`	open for reading (default)
`w`	open for writing, truncating the file first
`x`	open for exclusive creation, failing if the file already exists
`a`	open for writing, appending to the end of the file if it exists
`b`	binary mode
`t`	text mode (default)
`+`	open for updating (reading and writing)

```
# Open a file in write/update mode
# if the file is not present it will create one.
file_descriptor = open("foo.txt", "w+")
print("filename: ", file_descriptor.name)
file_descriptor.write("I am writting into this file")
# Close opend file
file_descriptor.close()
```

Using `try` `except` `finally`

```
try:
    # Open a file in write/update mode
    # if the file is not present it will create one.
    file_descriptor = open("foo.txt", "w+")
    print("filename: ", file_descriptor.name)
    file_descriptor.write("I am writting into this file")
except:
    print("We have an exception")
finally:
    # Close opend file
    file_descriptor.close()
```

Methods in File Objects

`file_descriptor.read()`

Reads some quantity of data and returns it as a `string` (in text mode) or `bytes` object (in binary mode).

- `read()` all contents are output.
- `read(size)` will output the size mentioned.

`read()`

```
try:
    file_descriptor = open("foo.txt", "r+")
    print(file_descriptor.read())
except:
    print("We have an exception")
finally:
    # Close opend file
    file_descriptor.close()

 #Output
 #>>>  I am writting into this file
```

`read(size)`

```
try:
    file_descriptor = open("foo.txt", "r+")
    print(file_descriptor.read(10))
except:
    print("We have an exception")
finally:
    # Close opend file
    file_descriptor.close()

 #Output 10 chars
 #>>>  I am writt
```

`file_descriptor.readline()`

The `readline()` method returns the line from the file given.

`readline()`

```
try:
    with open("foo.txt", "r+") as file_descriptor:
        # get the first line.
        line = file_descriptor.readline()
        # Check if we have a line if so print it.
        while line:
            # Print the line
            print(line)
            # Move on to next line
            line = file_descriptor.readline()
except:
    print("We have an exception")
finally:
    # Close opend file
    file_descriptor.close()

# Output
# 1. I am writting into this file
# 2. I am writting into this file
# 3. I am writting into this file
# 4. I am writting into this file
```

`readline(size)`

```
try:
    with open("foo.txt", "r+") as file_descriptor:
        # get the first line.
        line = file_descriptor.readline(10)
        # Check if we have a line if so print it.
        while line:
            # Print the line
            print(line)
            # Move on to next line
            line = file_descriptor.readline(10)
except:
    print("We have an exception")
finally:
    # Close opend file
    file_descriptor.close()

# Output 10 char per line
#
# 1. I am wr
# itting int
# o this fil
# e
#
# 2. I am wr
# itting int
# o this fil
# e
#
# 3. I am wr
# itting int
# o this fil
# e
#
# 4. I am wr
# itting int
# o this fil
# e
```

`file_descriptor.write()`

`f.write(string)` writes the contents of string to the file, returning the number of characters written.

```
try:
    # Open a file in write/update mode
    # if the file is not present it will create one.
    file_descriptor = open("foo.txt", "w+")
    print("filename: ", file_descriptor.name)
    char_written = file_descriptor.write("I am writting into this file")
    print("We wrote {} chars to the file".format(char_written))
except:
    print("We have an exception")
finally:
    # Close opend file
    file_descriptor.close()

# Output
# We wrote 28 chars to the file
```

`file_descriptor.seek()` and `file_descriptor.tell()`

- `f.tell(`) returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- To change the file object's position, use `f.seek(offset, whence)`.
- The position is computed from adding offset to a reference point;
    - the reference point is selected by the `whence` argument.
    - A `whence` value of `0` measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point.
    - `whence` can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Saving data to `json`.

Python data hierarchies, and convert them to `string` representations; this process is called `serializing`. Reconstructing the data from the string representation is called `deserializing`. Between `serializing` and `deserializing`, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

If you have an object x, you can view its JSON string representation with a simple line of code:

Convert `dict` to `JSON` using `json.dumps()` function.

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Convert `JSON` to `dict` using `json.load()` function.

File: `my_json_file.json`

```
{
    "book": [
        {
            "id": "444",
            "language": "C",
            "edition": "First",
            "author": "Dennis Ritchie "
        },
        {
            "id": "555",
            "language": "C++",
            "edition": "second",
            "author": " Bjarne Stroustrup "
        }
    ]
}
```

```
import json
import pprint
try:
    with open("my_json_file.json", "r+") as jsonfile_descriptor:
        json_dictionary = json.load(jsonfile_descriptor)
        pprint.pprint(json_dictionary)
except Exception as e:
    print("We have an exception {}".format(e))
finally:
    # Close opend file
    jsonfile_descriptor.close()

# Output
#{'book': [{'author': 'Dennis Ritchie ',
#           'edition': 'First',
#           'id': '444',
#           'language': 'C'},
#          {'author': ' Bjarne Stroustrup ',
#           'edition': 'second',
#           'id': '555',
#           'language': 'C++'}]}
```

Help

- Beginners Guide
- Python Loops
- Control Statemetns
- Built-in Functions
- String Methods
- Collections
- Exception Handling
- Python's Built-in Exceptions