Chapter 2
Python Basics

Chapter Content

- Keyword - Syntax
- Variables - Operators
- Fundamental Data types
- Control flow statements
- Loop control statements
- Function
- File Handling
- Exception Handling

Keyword

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

- Not use keyword to name for: class, object, function, variable, const…

- Rules for variable name in Python:
  - Must start with a letter or the underscore character
  - Cannot start with a number
  - Can only contain alpha, numeric characters and underscores (A-z, 0-9, and _)
  - Don't use Python's keyword to name variables
  - Are case-sensitive (age, Age and AGE are three different variables)

- Example:
  - Correct variable name: `Hello_1` `_Hello`
  - Variables are different : `spam` `Spam` `SPAM`
  - Incorrect variable name:
    - `1_Hello`: start with a number character
    - `He llo`: contains spaces
    - `print`: Python's keyword

Variables

- Python has no command for declaring a variable.
- Assignment operator: **=**
- Multiple data types can be assigned to a variable

```
x = 4            # x is of type int
x = "Sally"      # x is now of type str
```

- Variables can also specific to the particular data type with casting

```
x = str(3)       # x will be '3'
y = int(3)       # y will be 3
```

- One values can be assigned to multiple variables

```
a, b, c = 1
```

- Many values can be assigned to multiple variables

```
x, y, z = "Orange", "Banana", "Cherry"
```

- It is possible to query that data through the variable name

```
x = "Orange"
print(x)      ➔ Orange
```

- The **input()** function allows user input

```
x = input('Enter your name:')
print('Hello, ' + x)
```

- Constant is a quantity with constant value

```
a, b, c = 1
```

Constants

```
x, y, z = "Orange", "Banana", "Cherry"
```

Operators

- Arithmetic Operators: `+, -, *, /, %, **, //`
- Comparison Operators: `==, !=, >=, <=, >, <`
- Logical Operators: `and, or, not`
- Identity Operators: `is, is not`
- Membership Operators: `in, not in`
- Bitwise Operators: `&, |, ^, ~, >>, <<`
- Assignment Operators: `=, +=, -=, *=, /=, %=, **=, //=, |=, &=, >>=, <<=`

Expression

- An expression is used to calculate and return a value
- An expression consisting of a sequence of operands (values) linked by operators
- Ví dụ:
  - `3 + 2 + 6` ⇒ `11`
  - `"Python is a" + Programming language"` ⇒ `Python is a Programming language`
  - `(3 + 4) * 2 - (2 + 3) / 5` ⇒ `14`

Statement

- End of a statement on line break
- A statement can be extended over multiple lines by character (\)
- Example:

```
sum = 1+3+5 + \
      3+2+4
```

⇔ `sum = 1+3+5+3+2+4`

- Or ( ); [ ] ; { }
- Example: The commands below are the same

```
sum = {1+3+5 +
       3+2+4}
```

⟺

```
sum = (1+3+5 +
       3+2+4)
```

- Multiple commands can be written on one line, but separated by semicolons (;)

Command Block

- Command blocks will be recognized by indents
- A Command blocks begins with an Indentation and ends with the first line without Indentation
- Indentation spacing is arbitrary but should be consistent within a program (4 space key).
- Example:

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

```
if True :
  print("Hello")
  print("True")
else:
  print("False")
```

- Comments for a line starts with a #, and Python will ignore them

```
#This is a comment
print("Hello, World!")
```

- Comments for a paragraph use """

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Numeric Types

- Int (integer) is a whole number, positive or negative, without decimals, of unlimited length
- Float is a number, positive or negative, containing one or more decimals. It can also be scientific numbers with an "e" to indicate the power of 10.
- Complex numbers are written with a "j" as the imaginary part.

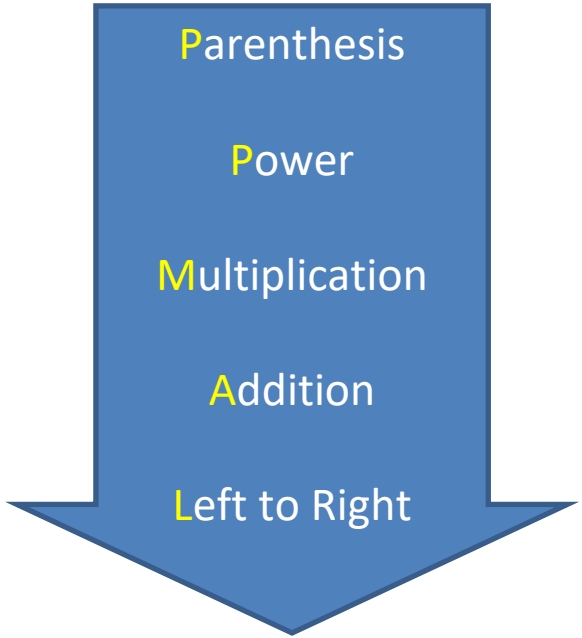
```
x = 1     # int
y = 2.8   # float
z = 1j    # complex
```

- Convert from one type to another with the int(), float(), and complex() methods

- To uses the following symbols to perform operations on numeric data:
  - + : Addition
  - - : Subtraction
  - / : Division. When two integers (int) are divided, the result is a real number (float)
  - * : Multiplication
  - **: Exponential
  - %: Remainder Division
  - //: Integer division
- Example:

```
5 + 2    ⇒ 7      5 - 2    ⇒ 3      5 // 2   ⇒ 2
5 * 2    ⇒ 10     5 / 2    ⇒ 2.5    5 ** 2   ⇒ 25
5 % 2    ⇒ 1
```

- The precedence's order of operations according to PPMAL ruler




```
x = 1 + 2 ** 3 / 4 * 5
print(x)                    ⇒ 11.0
```

```
1 + 2 ** 3 / 4 * 5
    1 + 8 / 4 * 5
      1 + 2 * 5
        1 + 10
           11
```

- Some common Math Functions
- Import functions of math class: `from math import *`

Function Name	Describe	Example	Rusult
ceil(float value)	Round up	ceil(102.4)	103
cos(radian value)	Cosin of an angle	cos(0)	1
floor(float value)	Round down	floor(102.4)	102
log(value, base)	logarit with base	log(6,2)	2.6
log10(value)	logarit with base 10	log10(6) =log(6,10)	0.78
Max(value 1,value 2 ...)	Return value maximum	max(3,5,6)	6
min(value 1, value 2, ...)	Return value minimum	min(3,5,6)	3
round(float value)	Rounding	round(102.4) round(102.7)	102 103
sin(gradian value)	Sin of an angle	sin(0)	0
sqrt(value)	Calculate square root 2	sqrt(25)	5
abs(value)	Return the absolute value	abs(-100)	100

pi = 3.141592653589793 e = 2.718281828459045

String Type

- Data string is surrounded by either single quotation marks, or double quotation marks
- Example: `'hello'` is the same as `"hello"`
- Assign a multiline string to a variable by using three quotes.

```
longer = " " " This string has
         multiple lines " " "
```

- Convert from one type to another with the `int()`, `float()`, and `complex()` methods.

- String operators `str1= "Hello"` `str2= "world"`

+	Concatenation operator	`Str1 + str2` ➔ `"Helloworld"`
*	Repetition operator	`str1* 3` ➔ `"HelloHelloHello"`
[]	Slice operator	`str1[4]` ➔ `'o'`
[:]	Range Slice operator	`str1[6:10]` ➔ `'world'`
in	Membership operator (in)	`'w' in str2` ➔ **True**
not in	Membership operator (not in)	`'e' not in str1` ➔ **False**

- String Functions

capitalize()	expandtabs()	isalnum()	upper()	partition()
casefold()	find()	isalpha()	title()	replace()
center()	format()	isdecimal()	join()	rfind()
count()	format_map()	isdigit()	ljust()	rindex()
encode()	format_map()	islower()	lower()	rjust()
endswith()	index()	isnumeric()	lstrip()	….

- Truy cập vào một ký tự trong chuỗi: StringName[index]
  - Index value is a integer number and first element is 0
  - The index value of the last element can be represented by the value -1
  - Index value can be the value of the expression

b	a	n	a	n	a
0	1	2	3	4	5

```
fruit = "banana"
letter_5 = fruit[5]
print(letter_5)          ⇒    a
letter_end = fruit[-1]
print(letter_end)        ⇒    a
x = 3
w = fruit[x - 1]
print(w)                 ⇒    n
```

- Extract substring: StringName[index 1: index 2]
- Extract the substring, starting at the character with "index 1" to the adjacent preceding character of the character with "index 2“

Value ⇒	b	a	n	a	n	a
Index ⇒	0	1	2	3	4	5

```
fruit = 'banana'
letter = fruit[1:3]
print(letter)          ⇒ an
```

- Extract the substring, starting from the first character to the immediately preceding character of the specified character with "index": StringName[: chỉ số]
- Extract the substring, starting at the character with "index" to the last character: StringName[chỉ số :]
- Get the whole string : StringName[:]

Value ⇒	b	a	n	a	n	a
Index ⇒	0	1	2	3	4	5

```
fruit = 'banana'
letter = fruit[:3]
print(letter) ⇒ ban

letter = fruit[1:]
print(letter) ⇒ anana

letter = fruit[:]
print(letter) ⇒ banana
```

Value ⇒	M	o	n	t	y		P	y	t	h	o	n
Index ⇒	0	1	2	3	4	5	6	7	8	9	10	11

```
s = "Monty
Python"
print(s[:2])        ⇒ Mo
print(s[8:])        ⇒ thon
print(s[:])         ⇒ Monty Python
print(s[0:4])       ⇒ Mont
print(s[6:7])       ⇒ P
print(s[6:20])      ⇒ Python
```

- Method: is an action that Python can perform on an object. Syntax to use the method: ObjectName.MethodName()
- Some methods for string data:
  - StringName.title(): convert the data of StringNmae to data with the first character of the words in the string to uppercase.
  - StringName.upper(): Convert data to uppercase string.
  - StringName.lower(): convert data to lowercase string.
- Example:

```
name= "ngon ngu python"
print(name.title())     ⇒          Ngon Ngu Python
print(name.upper())     ⇒          NGON NGU PYTHON
print(name.lower())     ⇒          ngon ngu python
```

- Some methods for string data:
  - StringName.rstrip(): Remove the spaces on the right of the string
  - StringName.lstrip(): Remove the spaces on the left of the string
  - StringName.strip(): Remove spaces on both sides of the string.
- Example:

```
name = "□ □Tran□"
print(name.lstrip()) ⇒        "Tran□"
print(name.rstrip()) ⇒        "□ □Tran"
print(name.strip()) ⇒         "Tran"
```

- Some methods for string data:
  - StringName.replace(SubStr 1, Subtr 2): Generates a new string from StringName. This new string is replaced by SubStr 1 with SubStr 2.
  - Example:

```
name = "Hello Python 2.0"
nameNew = name.replace("2.0", "3.7.14")
print(nameNew)    ⇒    "Hello Python 3.7.14"
```

  - StringName.find(string to find): Return a Integer number which the index of String to find. If not found, return -1. Starting position is index 0.
  - Example:

```
name = "Hello Panana"
print(name.find("na"))        ⇒    8
print(name.find("Python"))    ⇒    -1
```

- Some methods for string data:
  - StringName.find(string to find, location): Return a Integer number which the index of String to find. If not found, return -1. Starting position is index location.
  - Example:

```
name = "Hello□Panana□Panana"
print(name.find("□",6))     ⇒     12
print(name.find("□"))       ⇒      5
```

- Some methods for string data:
  - StringName.isupper(): Return True, if all character is Uppercase character.
  - StringName.islower(): Return True, if all character is Lowercase character.
  - Example:

```
name1 = "HELLO"
name2 = "hello"
print(name1.isupper())      ⇒      True
print(name2.islower())      ⇒      True
```

- Some methods for string data:
  - Concatenating: Use the symbol (+)
  - Example:

```
Ten     = "trung"
Ho_lot = "van"
Ho      = "phan"
Ho_va_ten = Ho + "□" + Ho_lot + "□"+ Ten
print(Ho_va_ten)   ⇒        "phan□van□trung"
```

  - Repeat number of times string value: Use the symbol (*)
  - Example:

```
st="Hello□"
st = 4 * st
print(st)          ⇒          Hello□Hello□Hello□Hello
```

- Some methods for string data:
  - Use the symbol (in) to check if a string is in another string
  - Example:

```
st1 = "Hello Python"
st2 = "Hello"
st3 = "Pyth"
st2 in st1              ⇒ True
st3 in st1              ⇒ False
```

  - Use the symbol (\n) to insert a newline into the string
  - Example: `print("Xin\nChao!")` ⇒

```
Xin
Chao!
```

  - Use the symbol (\t) to insert a Tab
  - Example: `print("Xin\tChao!")` ⇒ `Xin	Chao!`

- Some Functions for string data:
  - **len**(StringName): Returns an integer indicating the length of the string.
  - Example:

```
name = "□ □Tran□"
print(len(name))      ⇒      7
```

  - int(IntegerStringName): Returns an integer number
  - float(FloatStringName): Returns an float number
  - Example:

```
st1, st2 = "20", "30"
   Sum1 = st1 + st2                    ⇒    "2030"
   Sum2 = int(st1) + int(st2)          ⇒    50
   st1, st2 ="20.5", "30.5"
   Sum1 = st1 + st2                    ⇒    "20.530.5"
   Sum2 = float(st1) + float(st2)      ⇒    51.0
```

Boolean Type

- Represents one of two values: True or False.
- The `bool()` function is used to evaluate any value, and return True or False in the result.
- Almost any value is evaluated to True if it has some sort of content; any string is True, except empty strings; any number is True, except 0; any list, tuple, set, and dictionary are True, except empty ones.
- Not many values are evaluated to False, except empty values, such as (), [], {}, "", the number 0, and the value None. And of course the value False evaluates to False.

- Comparison operator : Return True or False

Operator	Example	Result
==	`1 + 1 == 2`	`True`
!=	`3.2 != 2.5`	`True`
<	`10 < 5`	`False`
>	`10 > 5`	`True`
<=	`126 <= 100`	`False`
>=	`5.0 >= 5.0`	`True`

- Logic operator : Return True or False

Operator	Mean	Result	Example	Result
and	True **and** True True **and** False False **and** False	True False False	(2 < 3) **and** (-1 < 5) (2 == 3) **and** (-1 < 5) (2 == 3) **and** (-1 > 5)	True False False
or	True **or** False	True	(2 == 3) **or** (-1 < 5)	True
not	**Not** True **Not** False	False True	**not** (2 == 3)	True

Practice and exercises
Part 1

Structured Data

- List
- Set
- Tuple
- Dictionary

- are like dynamically sized arrays used to store multiple items
- Properties of a list: mutable, ordered, heterogeneous, duplicates.

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [['tiger', 'cat'], ['fish']]
list4 = ["abc", 34, True, 40, "abc"]
```

List Initialization

- Using square brackets []

```
# an empty list
L1= list[]
# a list of 3 items
L2= list['banana','apple', 'kiwi']
```

- Using list() constructor

```
# an empty list
L1 =list()
# a list of 3 items
L2= list((banana','apple', 'kiwi'))
```

- Using list multiplication

```
# a list of 10 items of ' '
L1= list[' ']*10
```

- Using list comprehension

```
# a list of 10 items of ' '
L2 = [' ' for i in range(10)]
```

Operations on Lists

- Modify list items
- Insert list items
- Append items
- Extend the list
- Remove list items

List Methods

Append()	Add an element to the end of the list
Extend()	Add all elements of a list to another list
Insert()	Insert an item at the defined index
Remove()	Removes an item from the list
Pop()	Removes and returns an element at the given index
Clear()	Removes all items from the list

Index()	Returns the index of the first matched item
Count()	Returns the count of the number of items passed as an argument
Sort()	Sort items in a list in ascending order
Reverse()	Reverse the order of items in the list
copy()	Returns a copy of the list

- Example:

(1) `L1 = [ "a", "b", "c", "d"]`

(2) `L1.append("e")` ⇒ `["a", "b", "c", "d", "e"]`

(3) `L1.insert(0, "X")` ⇒ `["X", "a", "b", "c", "d", "e"]`

(4) `L1.pop()` ⇒ `["X", "a", "b", "c", "d"]`

(5) `L1.pop(0)` ⇒ `["a", "b", "c", "d"]`

(6) `L1.remove("a")` ⇒ `["b", "c", "d"]`

(7) `del L1[2]` ⇒ `["b", "c"]`

(8) `L1.clear()` ⇒ `[ ]`

- Example:
- Method 1:

```
L2 = [ "a", "d", "c", "b"]
     L2.sort()          ⇒          [ "a", "b", "c", "d"]
     L2.sort(reverse=True) ⇒       [ "d", "c", "b", "a"]
```

- Method 1:

```
L2 = [ "a", "d", "c", "b"]
print(danh_sach_ky_tu)          ⇒          [ "a", "d", "c", "b"]
print(sorted(danh_sach_ky_tu)) ⇒          [ "a", "b", "c", "d"]
print(danh_sach_ky_tu)          ⇒          [ "a", "d", "c", "b"]
```

- Example:

```
(1) cars = ["bmw", "audi", "toyota", "subaru"]
(2) print(cars)          ⇒ ["bmw", "audi", "toyota", "subaru"]
(3) cars.reverse()
(4) print(cars)          ⇒ ["subaru", "toyota", "audi", "bmw"]
```

- Example:

```
(1) cars = ["bmw", "audi", "toyota", "subaru"]
(2) len(cars)      ⇒  4
```

- Convert a string to a list of characters (character separator): Used to list() function
- Syntax:

SourceString = "String Value"

ResultList =**list**(SourceString)

- Example:

```
(1) St = "hello"
(2) DS = list(St)
(3) print(DS)          ⇒ ["h", "e", "l", "l", "o"]
```

- Split string into elements of a list : Used to split() menthod
- Syntax: StringName.**split()**
- Each element is identified through the space character (space key) in the sentence.
- Example:

```
(1) Sentence = "subaru□toyota□audi□bmw"
(2) ResultList = Sentence.split()
(3) print(Sentence)          ⇒ "subaru□toyota□audi□bmw"
(4) print(ResultList)    ⇒ ["subaru","toyota","audi", "bmw"]
```

- To split a string into the elements list of a comma-defined (,):

  StringName.**split(",")**

- Concatenating strings and list: Used to **join()** menthod
- Append a string to each element of the list to produce a string.
- Syntax:

```
st = "String"
L1 = [item1, item2, …, itemLast]
st.join(L1)
```

Result: Return 1 string: `item1Stringitem2String … itemLast`

- Example:

```
(1) cars = ["bmw", "audi", "toyota", "subaru"]
(2) print(cars)    ⇒    ["bmw", "audi", "toyota", "subaru"]
(3) space = "□"
(4) print(space.join(cars)) ⇒ bmw□audi□toyota□subaru
```

- Merge 02 lists: Used to **extend()** menthod
- Syntax: List1.**extend**(List2)
- Append List2 into List1
- Example:

```
(1) cars = ["bmw", "audi"]
(2) cars_new  = ["toyota", "subaru"]
(3) print(cars)                ⇒  ["bmw", "audi"]
(4) cars.extend(cars_new)
(5) print(cars)                ⇒  ["bmw", "audi", "toyota", "subaru"]
```

- Example:

```
(1) ds=[1,5,6,7,9,7,6,7,20]
(2) max(ds)                    ⇒  20
(3) min(ds)                    ⇒  1
(4) ds.count(7)                ⇒  3
```

Tuples

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and unchangeable
- are written with round brackets.
- Example:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)    ⇒ ("apple", "banana", "cherry")
```

Tuple Initialization

- One item tuple, remember the comma:
- Example:

```
thistuple = ("apple",)
print(type(thistuple))     ⇒ <class 'tuple'>

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))     ⇒ <class 'str'>
```

Operations on Tuple

- Access Tuples
- Unpacked Tuples
- Loop Tuples
- Join Tuples

- Access Tuple: can access tuple items by referring to the index number, inside square brackets:
  - Ex1:
    ```
    thistuple = ("apple", "banana", "cherry")
    print(thistuple[1])
    ```
  - Ex2:
    ```
    thistuple = ("apple", "banana", "cherry")
    print(thistuple[-1])
    ```
  - Ex3:
    ```
    thistuple = ("apple", "banana", "cherry", "orange")
    print(thistuple[2:3])
    ```

- Update Tuples:
  - Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.
  - Convert the tuple into a list, change the list, and convert the list back into a tuple.
- Example:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

- Unpacked Tuples: extract the values back into variables
- Example:

```
# Packed Tupples
fruits = ("apple", "banana", "cherry")
# Unpacked Tupples
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)
```

- Join Tuples:
- Ex 1:

```
# use "+" operator
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

- Ex 1:

```
# use "*" operator
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

- Unpacked Tuples: extract the values back into variables

Index()	Searches the tuple for a specified value and returns the position of where it was found
Count()	Returns the number of times a specified value occurs in a tuple

Set

- Used to store multiple items in a single variable.
- is a collection which is unordered, unchangeable, and unindexed.
- Sets are written with curly brackets.
- Set Initialization
  - Ex1:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

  - Ex2: Sets cannot have two items with the same value.

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

Operations on Set

- Access Set Items
- Add Set Items
- Remove Set Items
- Loop Set Items
- Join

- Access Sets Items:
  - Cannot access items in a set by referring to an index or a key.
  - But we can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.
  - Example:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
print(x)
```

- Remove Set Items: to remove using **remove()** method or **discard()** method.
  - Example 1:

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

  - Example 2:

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

- Loop: through the set items by using a for loop:
  - Example:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
print(x)
```

- Join: using **union()** method or **update()** method.
  - Example:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)

Set Methods

Method	Description
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Dictionary

- Are used to store data values in <key> : <value> pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- written with curly brackets, and have keys and values
- Example:

```
thisdict = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964
            }
print(thisdict)
```

Dictionary Initialization

- Example:

```
# Duplicate values will overwrite existing values:
thisdict = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964,
            "year": 2020
            }
 print(thisdict)
```

Operations on Dictionary

- Access Items
- Change Items
- Add Items
- Remove Items
- Loop Items
- Copy Dictionaries

- **Access Items**: access the items of a dictionary by referring to its key name, inside square brackets:
- Example:

```
thisdict = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964
            }
x = thisdict["model"]
```

- **Change Items:**
  - Update dictionary: **update()** method will update the dictionary with the items from the given argument
  - Example

```
thisdict = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964
           }
thisdict.update({"year": 2020})
```

- **Add Items**: using a new index key and assigning a value to it
- Example:

```
thisdict = {
              "brand": "Ford",
              "model": "Mustang",
              "year": 1964
            }
thisdict["color"] = "red"
print(thisdict)
```

- **Remove Items:**
  - **Pop()** method: removes the item with the specified key name
  - Example:

```
thisdict = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964
           }
thisdict.pop("model")
print(thisdict)
```

- **Remove Items:**
  - **popitem()** method: removes the last inserted item (in versions before 3.7, a random item is removed instead)
  - Example:

```
thisdict = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964
           }
thisdict.popitem()
print(thisdict)
```

- **Remove Items:**
  - **del :** delete the dictionary completely
  - **Clear()** method: empties the dictionary
  - Example:

```
thisdict = {
                "brand": "Ford",
                "model": "Mustang",
                "year": 1964
            }
del thisdict
print(thisdict)
thisdict.clear()
print(thisdict)
```

- **Loop dictionary**: the return value are the keys of the dictionary, but there are methods to return the values as well.
- Ex1: Print all key names in the dictionary, one by one:

```
for x in thisdict:
print(x)
```

- Ex2: Print all values in the dictionary, one by one:

```
for x in thisdict:
print(thisdict[x])
```

- **Loop dictionary**: the return value are the keys of the dictionary, but there are methods to return the values as well.
- Ex3: **values()** method to return values of a dictionary:

```
for x in thisdict.values():
print(x)
```

- Ex4: **keys()** method to return the keys of a dictionary:

```
for x in thisdict.keys():
print(x)
```

- Ex5: **items()** method to through both keys and values

```
for x, y in thisdict.items():
print(x, y)
```

Dictionary Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys

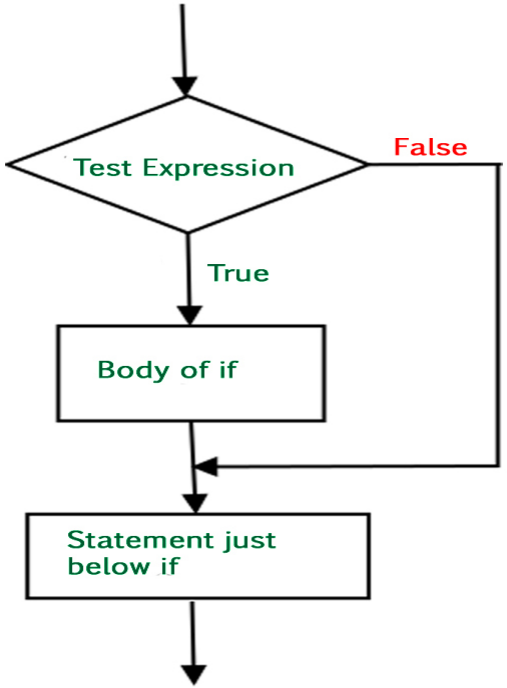
Dictionary Methods

Method	Description
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Conditional Control Statements

- If statement
- If… else statement
- If… elif… else statement
- Nested If statement
- Short- hand if & if…else statements

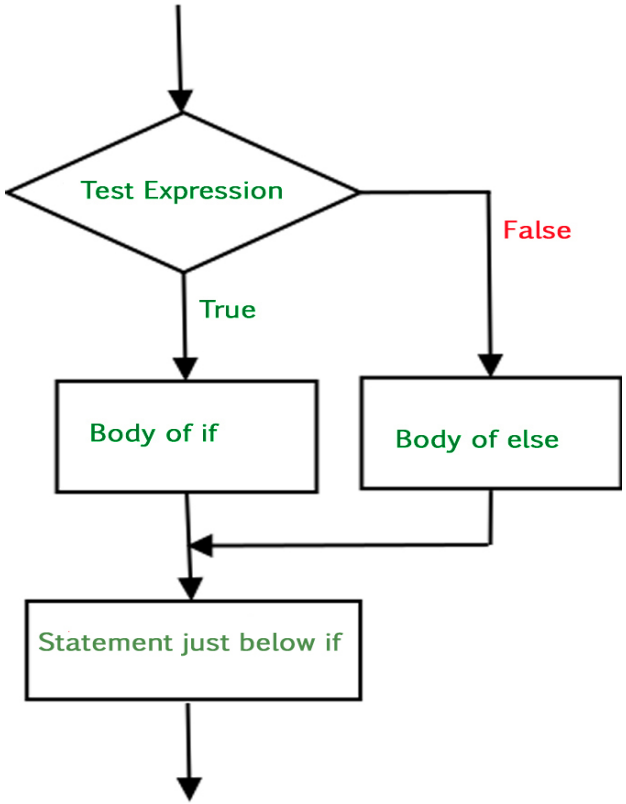
if statement




```
if (condition):
    # Statements to execute if condition is true
```

```
i = 10
if (i > 15):
    print("10 is less than 15")
print("I am Not in if")
```

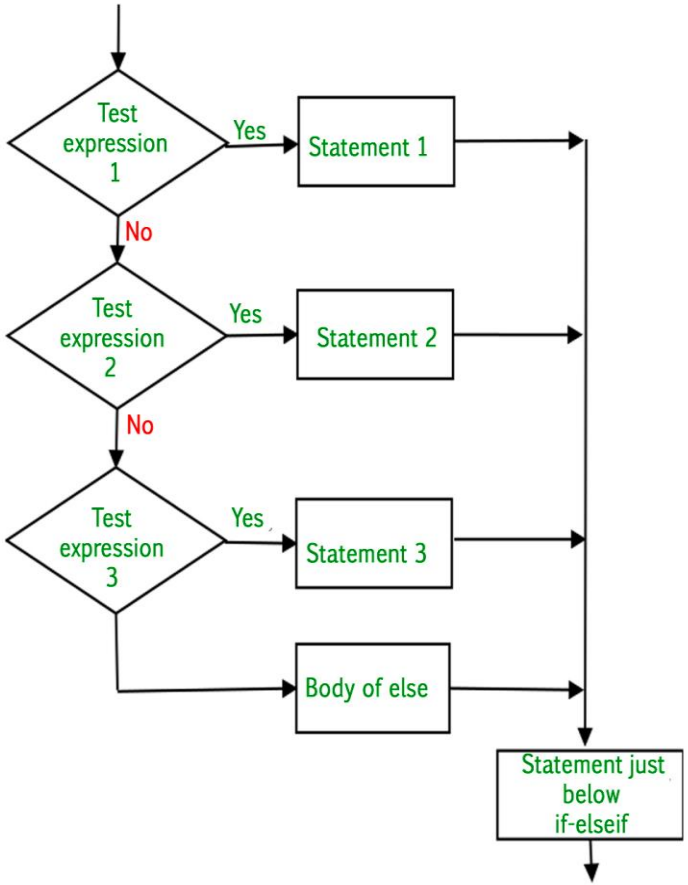
if…else statement




```
if (condition):
    # Executes this block if  condition is true
else:
    # Executes this block if condition is false
```

```
i = 20
if (i < 15):
    print("i is smaller than 15")
    print("in if Block")
else:
    print("i is greater than 15")
    print("in else Block")
print("not in if and not in else Block")
```

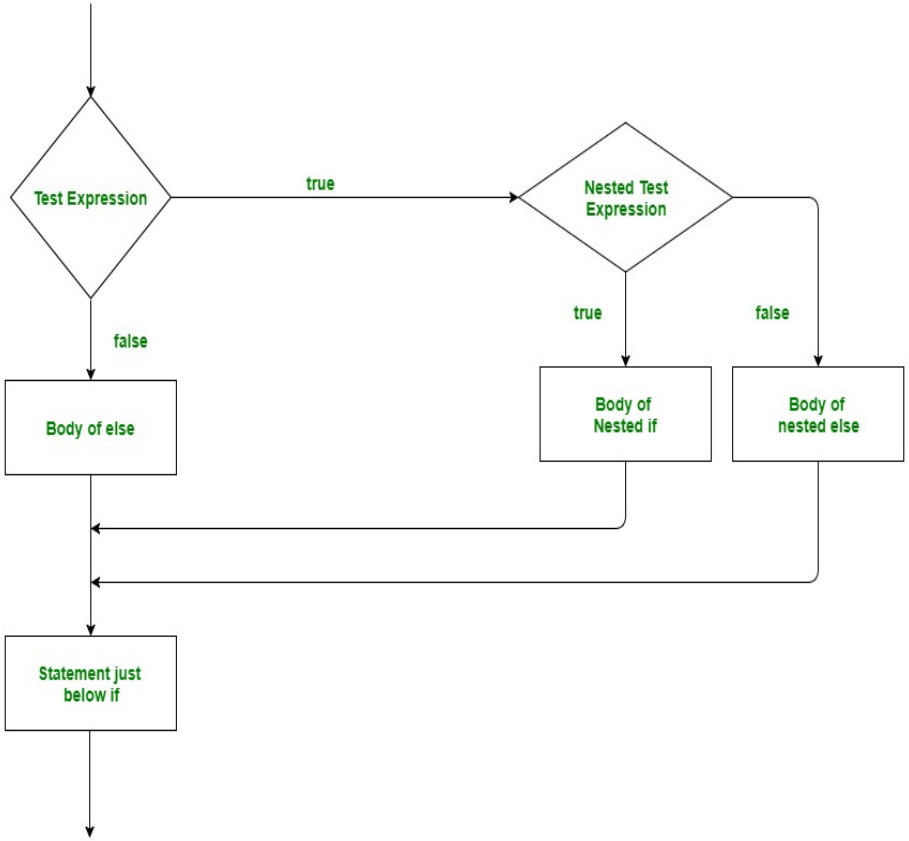
If... elif... else Statement




```
if (condition):
    statement
elif (condition):
    statement
⋮
else:
    statement
```

```
i = 20
if (i == 10):
    print("i is 10")
elif (i == 15):
    print("i is 15")
elif (i == 20):
    print("i is 20")
else:
    print("i is not present")
```

Nested If Statement




```
if (condition1):
  # Executes when condition1 is true
  if (condition2):
    # Executes when condition2 is true
  # if Block is end here
# if Block is end here
```

```
i = 10
if (i == 10):
    if (i < 15):
        print("smaller than 15")
    if (i < 12):
        print("smaller than 12")
    else:
        print("greater than 15")
```

Short- hand if & if…else statements

- If there is only one statement to execute, the If & If … else statements can be put on the same line

if condition: Statement

```
i = 10
if (i > 15): print("10 is less than 15")
```

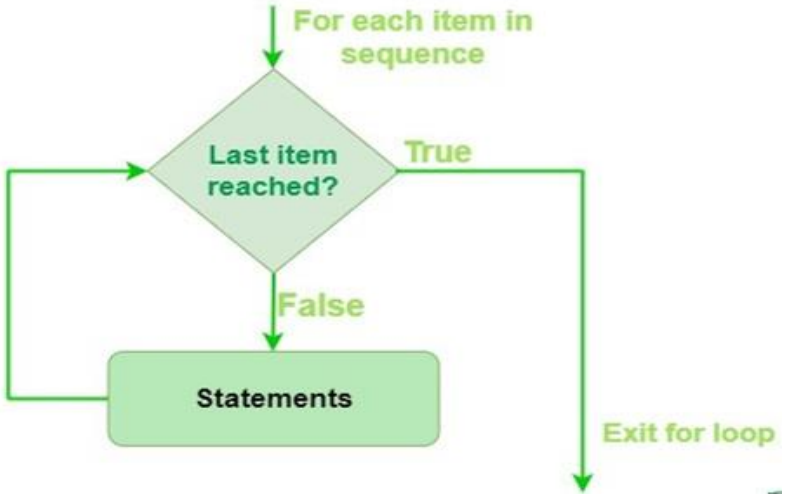
Statement_when True **if** (condition) **else** statement_when False

```
i = 10
print(True)  if (i < 15) else print(False)
```

Loop Control Statements

- **for** loop statements
- **while** loop statements
- The **range()** function
- Loops with **break** statement
- Loops with **continue** statement
- Loops with **else** statement
- Loops with **pass** statement

- is used for sequential traversals, i.e. iterate over the items of squensence like list, string, tuple, etc.
- In Python, for loops only implements the collection-based iteration.

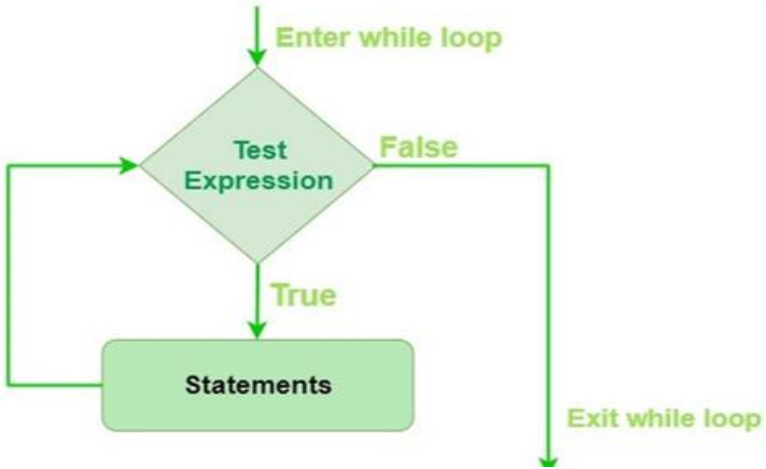



```
for variable_name in sequence :
    statement_1
    statement_2
        ....
```

```
L = ["red", "blue", "green"]
for i in L:
    print(i)
```

while Loop Statements

- is used to execute a block of statements repeatedly until a given condition is satisfied.
- can fall under the category of indefinite iteration when the number of times the loop is executed isn't specified explicitly in advance.




```
while expression:
        statement(s)
```

```
count = 0
while (count < 10):
    count = count + 1
    print(count)
```

The range() function

- is used to specific number of times whereby a set of code in the for loop is executed.
- returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

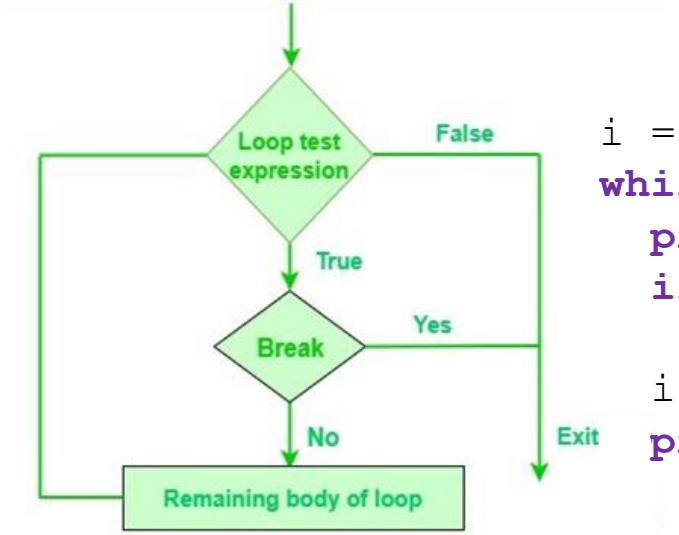
range(start_number, last_number, increment_value)

```
for x in range(2, 6):
  print(x)
```

```
for x in range(2, 30, 3):
  print(x)
```

Loops with break statement

- The break keyword in a for/while loop specifies the loop to be ended immediately even if the while condition is true or before through all the items in for loop.




```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
  print(i)
```

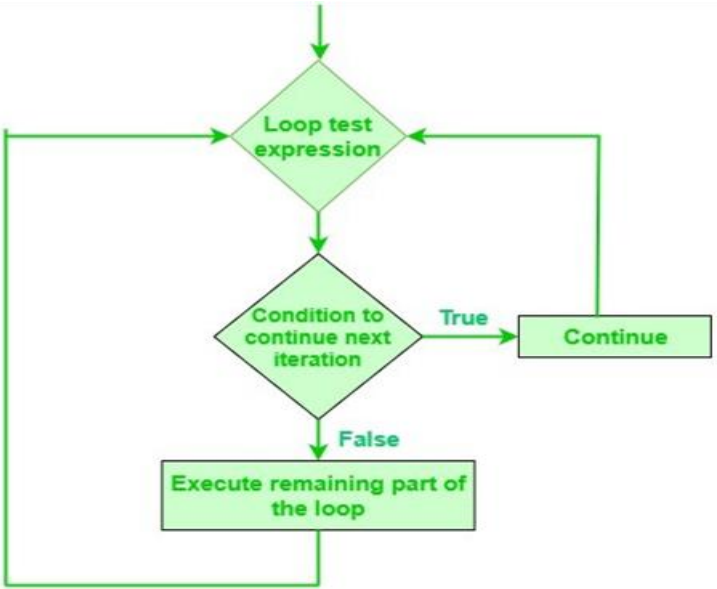
```
colors = ["blue", "green", "red"]
for x in colors:
  print(x)
  if x == "green":
    break
  print(x)
```

Loops with continue statement

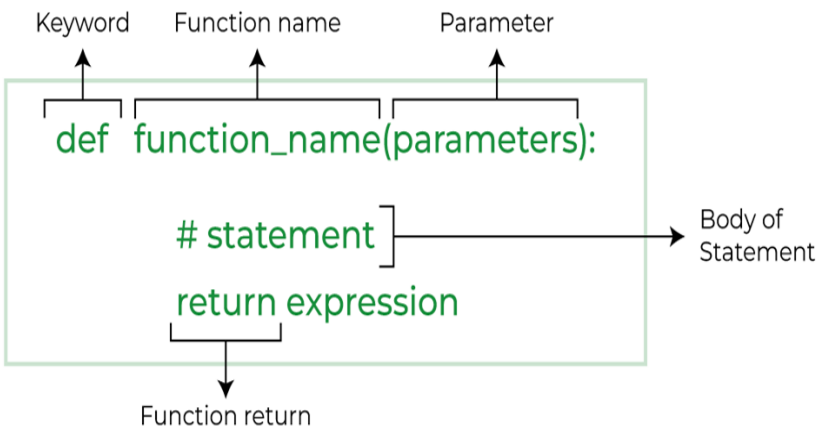
- The continue statement in a for/while loop is used to force to execute the next iteration of the loop while skipping the rest of the code inside the loop for the current iteration only.

```
i = 0
while i < 7:
    i += 1
    if i == 4:
        continue
    print(i)
```

```
for x in range(7):
    if (x == 4):
        continue
    print(x)
```



- Definition syntax:



Example:

```
# A function to check
# whether n is even or odd
def CheckEvenOdd(n):
    if (n % 2 == 0):
        print("even")
    else:
        print("odd")
```

- Calling a Python Function by using the name of the function followed by parenthesis containing parameters of that particular function.

Example:

```
# Driver code to call the function
CheckEvenOdd(2)
```

Types of Arguments

- A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.
- A keyword argument allows the caller to specify the argument name with values so that caller does not need to remember the order of parameters.

Example:

```
# default arguments
def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)
```

Example:

```
# a Python function
def student(firstname, lastname):
    print(firstname, lastname)
# Keyword arguments
student(firstname='Van A', lastname='Nguyen')
student(lastname='Nguyen', firstname='Van A')
```

- A variable length argument pass a variable number of arguments to a function using special symbols:
  - *args (Non-Keyword Arguments)

```
Example:
def myFun(*args):
    for arg in args:
        print(arg)
myFun('Welcome', 'to', 'VKU')
```

```
Welcome
to
VKU
```

  - **kwargs (Keyword Arguments)

```
Example:
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))
myFun(first='Welcome', second='to', last='VKU')
```

```
first  Welcome
second to
last   VKU
```

Organize storage and use functions

- In large projects, for easy management:
- Method 1:
  - Step 1: Create functions and save in a separate file (Module File)
  - Step 2: Import Module File into main program by used to import command

    import Module
  - Step 3: Used to function according to the syntax : Module.FunctionName

Organize storage and use functions

- Step 1: Create **Tong.py**

```
def Sum_list(ds):
        sum=0
    for d in ds
        sum = sum + d
    return sum
```

- Step 2: Create Test_tong.py and used to function

```
import Tong
list_odd = [ 11, 13, 15, 17, 19, 21]
sum1 = Tong.Sum_list(ds_le)
print("Sum of items in List : ", sum1)
```

Organize storage and use functions

- Method 2:
  - Step 1: Create functions and save in a separate file (Module File)
  - Step 2: Import Module File into main program by used to import command
    - Type 1: from Module import FunctionName
    - Type 2: from Module import FunctionName_1, FunctionName_2, …
    - Type 3: from Module import *
  - Step 3: Used to function according to the syntax: FunctionName

Organize storage and use functions

- Step 1: Create **Tong.py**

```
def Sum_List(ds):
        sum=0
    for d in ds
        sum = sum + d
    return sum
```

- Step 2: Create Test_tong.py and used to function

```
from Tong import *
List_odd = [ 11, 13, 15, 17, 19, 21]
sum1 = Sum_List(List_odd)
print("Sum of items in List: ", sum1)
```

- Method 1: Used to an alias
  - Step 1: Create functions and save in a separate file (Module File)
  - Step 2: Import Module File into main program by used to import command, and Used to an alias

    from Module import FunctionName as alias

  - Step 3: Sử dụng hàm gọi hàm theo cú pháp: alias

Organize storage and use functions

- Step 1: Create **Tong.py**

```
def Sum_List(ds):
        sum=0
    for d in ds
        sum = sum + d
    return sum
```

- Step 2: Create Test_tong.py and used to function

```
from Tong import Sum_List as Sum
List_odd = [ 11, 13, 15, 17, 19, 21]
sum1 = Sum(List_odd)
print(" Sum of items in List: ", sum1)
```

File Handling

- Opening file
- Reading file
- Writing to file
- Appending file
- With statement

- Using the **open()** function : `File_object=open(filename, mode)`
  - *filename*: the name of file
  - *mode*: represents the purpose of the opening file with one of the following values:
    - **r:** open an existing file for a read operation.
    - **w:** open an existing file for a write operation.
    - **a:** open an existing file for append operation.
    - **r+:** to read and write data into the file. The previous data in the file will be overridden.
    - **w+:** to write and read data. It will override existing data.
    - **a+:** to append and read data from the file. It won't override existing data.

Example:

```
# a file named "sample.txt",
will be opened with the
reading mode.
file = open('sample.txt', 'r')
# This will print every line
one by one in the file
for each in file:
    print(each)
```

- Using the **read()** method: `File_object.read(size)`
  - size <=0: returning a string that contains all characters in the file

```
# read() mode
file = open("sample.txt", "r")
print(file.read())
```

  - size>0: return a string that contains a certain number of characters size

```
# read() mode character wise
file = open("sample.txt", "r")
print(file.read(3))
```

Closing File

- Using **close()** method to close the file and to free the memory space acquired by that file
- Used at the time when the file is no longer needed or if it is to be opened in a different file mode.

```
File_object.close()
```

Writing to file

- Using the **write()** method to insert a string in a single line in the text file and the **writelines()** mwthod to insert multiple strings in the text file at a once time. Note: the file is opened in write mode

```
File_object.write/writelines(text)
```

- Example:

```
file = open('sample.txt', 'w')
L = ["VKU \n", "Python Programming \n", "Computer Science \n"]
S = "Welcome\n"
# Writing a string to file
file.write(S)
# Writing multiple strings at a time
file.writelines(L)
file.close()
```

Appending File

- Using the **write/writelines()** method to insert the data at the end of the file, after the existing data. Note: the file is opened in append mode
- Example:

```
file = open('sample.txt', 'w') # Write mode
S = "Welcome\n"
# Writing a string to file
file.write(S)
file.close()
# Append-adds at last
file = open('sample.txt', 'a') # Append mode
L = ["VKU \n", "Python Programming \n", "Computer Science \n"]
file.writelines(L)
file.close()
```

With statement

- used in exception handling to make the code cleaner and to ensure proper acquisition and release of resources.
- using **with** statement replaces calling the **close()** method

```
# To write data to a file using with statement
L = ["VKU \n", "Python Programming \n", "Computer Science \n"]
# Writing to file
with open("sample.txt", "w") as file1:
    # Writing data to a file
    file1.write("Hello \n")
    file1.writelines(L)
# Reading from file
with open("sample.txt", "r+") as file1:
# Reading form a file
    print(file1.read())
```

- **Try** and **Except** Statement – Catching Exceptions
- **Try** and **Except** Statement – Catching Specific Exceptions
- **Try** with **Else** and **Finally** Clauses

- **Try** and **except** statements are used to catch and handle exceptions in Python.

```
try :
    #statements
except :
    #executed when error in try block
```

- Example:

```
try:
        a=5
        b='0'
        print(a/b)
except:
        print('Some error occurred.')
print("Out of try except blocks.")
```

Try with Else and Finally Clauses

- The **else** block gets processed if the **try** block is found to be exception free (no exception).
- The **finally** block always executes after normal termination of **try** block or after **try** block terminates due to some exception

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if no exception
finally:
   #executed irrespective of exception occured or not
```

Try with Else and Finally Clauses

- Example:

```
try:
    print('try block')
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
 print("Out of try, except, else and finally blocks.")
```

Practice and exercises
Part 2