ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Chapter 2
Python Basics

Thu Huong Nguyen, PhD
Si Thin Nguyen, PhD

KHMT
Computer Science

http://vku.udn.vn/

About Authors

Thu Huong Nguyen

PhD in Computer Science at Université Côte d’Azur, France
Email: *nthuong@vku.udn.vn*
Address: Faculty of Computer Science, VKU

Si Thin Nguyen

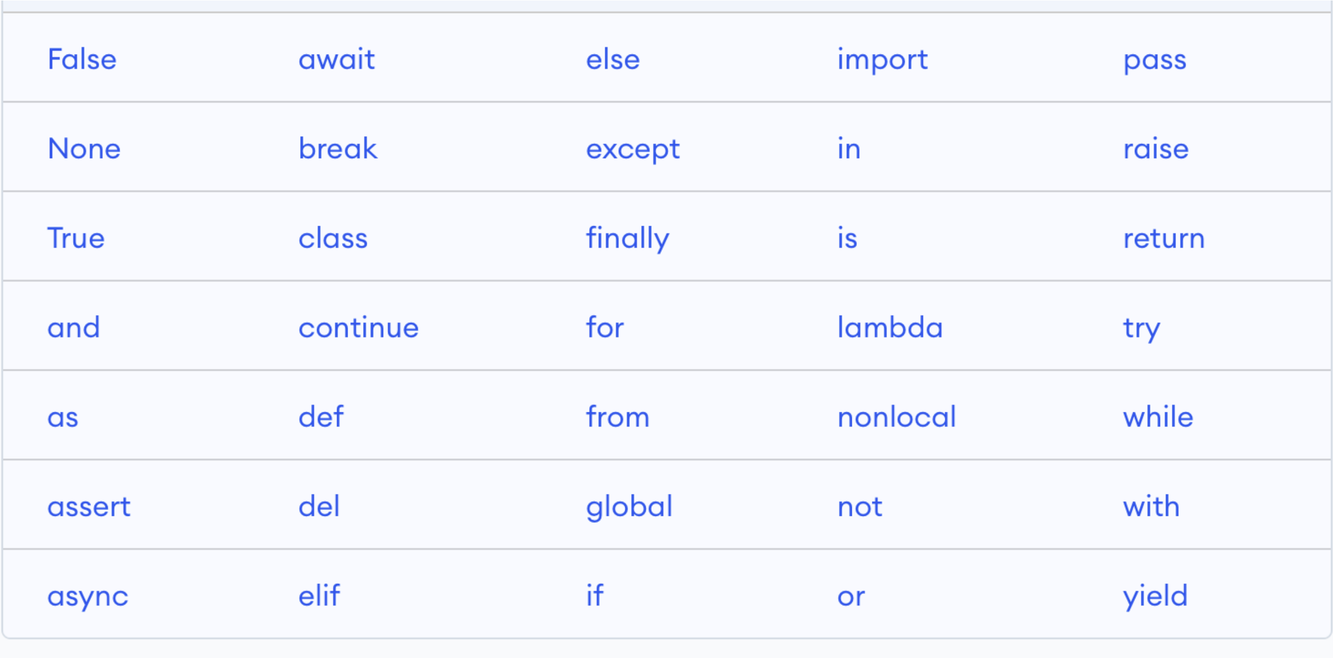
PhD in Computer Science at Soongsil University, Korea
Email: *nsthin@vku.udn.vn*
Address: Faculty of Computer Science, VKU

Chapter Content

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

Syntax

➔ ***Keywords*** in Python



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

Syntax

➔ **Indentation** which refers to the spaces at the beginning of a code line is obligated to use to indicate a block of code in control flows, classes or functions

Indentation

```
if 6 > 3 :
 print("Six is greater than three!")
```

➔ Indentations are the same for all statements in a block of code

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

Syntax

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

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

- ➔ ***Comments for a paragraph*** use """

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

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

Syntax

➔ ***Multi-line commands*** can use with multiple \

```
total = item_one + \
         item_two + \
         item_three
```

➔ ***Multi-line commands*** can also use [], {}, () and need not use \

```
   days = ['Monday', 'Tuesday',
'Wednesday', 'Thursday', 'Friday']
```

➔ ***Multiple commands in a line*** is splitted with ;

```
import sys; x = 'Hello'; sys.stdout.write(x + '\n')
```

Variables

- ➔ Python has no command for declaring a variable.
- ➔ A variable is created the moment a value is assigned to it in the first time.
- ➔ Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

```
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
```

Variables

➔ Many values can be assigned to multiple variables

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

➔ 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 _)

★ Are case-sensitive (age, Age and AGE are three different variables)

Operators

➔ Arithmetic Operators: +, -, *, /, %, **, //

➔ Assignment Operators: =, +=, -=, *=, /=, %=, **=, //=, |=, &=, >>=, <<=

➔ Comparison Operators: ==, !=, >=, <=, >, <

➔ Logical Operators: and, or, not

➔ Identity Operators: is, is not

➔ Membership Operators: in, not in

➔ Bitwise Operators: &, |, ^, ~, >>, <<

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

String

- ➔ are surrounded by either *single* quotation marks, or *double* quotation marks.

  Ex: `'hello'` is the same as `"hello"`

- ➔ Convert from one type to another with the `int()`, `float()`, and `complex()` methods.
- ➔ Assign a multiline string to a variable by using *three* quotes.

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

String

➔ String operators

```
>>> str1= “Hello”
>>> str2= “world”
```

+	Concatenation operator	>>> str1+ str2 >>> “Hello world”
*	Repetition operator	>>> str1* 3 >>> “Hello Hello Hello”
[]	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

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()	…..

➔ For more information about **string built-in functions** in

Boolean

- ➔ 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.

Data Structures

- Lists
- Sets
- Tuples
- List
    - List Initialization
    - Operations on Lists
    - List methods
- Dictionary

Lists

- ➔ 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

Data Structures

- Lists
- Tuples
  - Tuples Initialization
  - Operations on Tuples
  - Tuples methods
- Dictionary
- Sets

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)
```

Tuple Initialization

➔ One item tuple, ***remember the comma***:

Ex:

```
thistuple = ("apple",)
print(type(thistuple))
#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Operations on Tuple

➔ Access Tuples

➔ Unpacked Tuples

➔ Loop Tuples

➔ Join Tuples

Operations on Tuple

- **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])
```

Operations on Tuple

➔ **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.

⍰ Ex:

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

Operations on Tuple

➔ **Unpacked Tuples:** extract the values back into variables

⍰ **Ex:**

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

Operations on Tuple

➔ **Join Tuples:**

➔ **Ex 1:**

```
# use “+” operator

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

⍰ **Ex 2:**

```
# use “*” operator

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

Tuples Methods

➔**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

Data Structures

- Lists
- Tuples
- Sets
  - Sets Initialization
  - Operations on Sets
  - Sets methods
- Dictionary

Sets

- ➔ used to store multiple items in a single variable.
- ➔ is a collection which is ***unordered***, ***unchangeable***, and ***unindexed***.
- ➔ Sets are written with ***curly brackets.***

Sets 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 Sets

- ➔ Access Sets Items
- ➔ Add Sets Items
- ➔ Remove Sets Items
- ➔ Loop Sets Items
- ➔ Join

Operations on Sets

□ **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.

- **Ex1:**

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

Operations on Sets

- **Remove Sets Items:** to remove using `remove()` method or `discard()` method.

  ○ **Ex:**

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

Operations on Sets

➔ **Loop:** through the set items by using a for loop:

○ **Ex:**

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

Operations on Sets

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

  ○ **Ex:**

```
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)
```

Sets 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)

Sets 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

Data Structures

- Lists
- Tuples
- Sets
- Dictionary
  - Dictionary Initialization
  - Operations on Dictionary
  - Dictionary methods

Dictionaries

- ➔ 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:

Dictionaries Initialization

➔ Ex1:

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

Dictionaries Initialization

- Ex2:

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

Operations on Dictionaries

➔ Access Items

➔ Change Items

➔ Add Items

➔ Remove Items

➔ Loop Items

➔ Copy Dictionaries

Operations on Dictionaries

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

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

Operations on Dictionaries

➔ Change Items:

- **Update dictionary:** `update()` method will update the dictionary with the items from the given argument
- **Ex:**

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

Operations on Dictionaries

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

➔ **Ex:**

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

Operations on Dictionaries

➔ Remove Items:

- Pop() method: removes the item with the specified key name:
- Ex:

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

Operations on Dictionaries

➔ Remove Items:

- popitem() : removes the last inserted item (in versions before 3.7, a random item is removed instead):
- Ex:

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

Operations on Dictionaries

➔ **Remove Items:**

- del : delete the dictionary completely
- Clear(): empties the dictionary:
- Ex:

```
thisdict = {
      "brand": "Ford",
      "model": "Mustang",
      "year": 1964
}
del thisdict
print(thisdict) #this will cause an error
because "thisdict" no longer exists.
thisdict.clear()
print(thisdict)
```

Operations on Dictionaries

➔ **Loop dictionaries:** 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])
```

Operations on Dictionaries

→ **Loop dictionaries:** 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:** use the `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)
```

Dictionaries 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

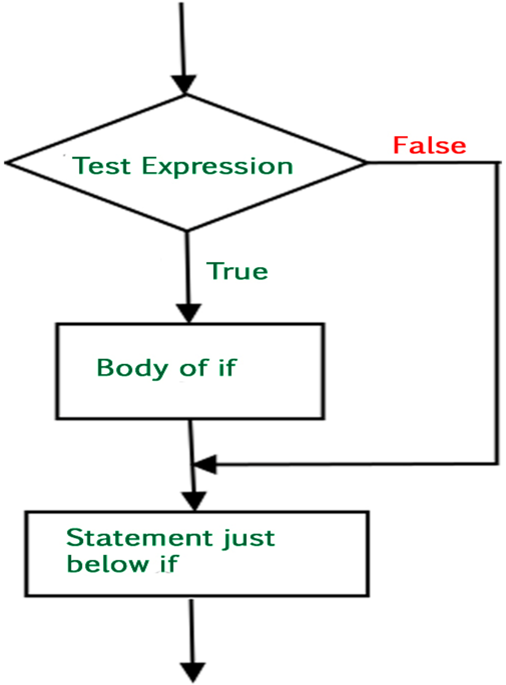
Dictionaries Methods

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

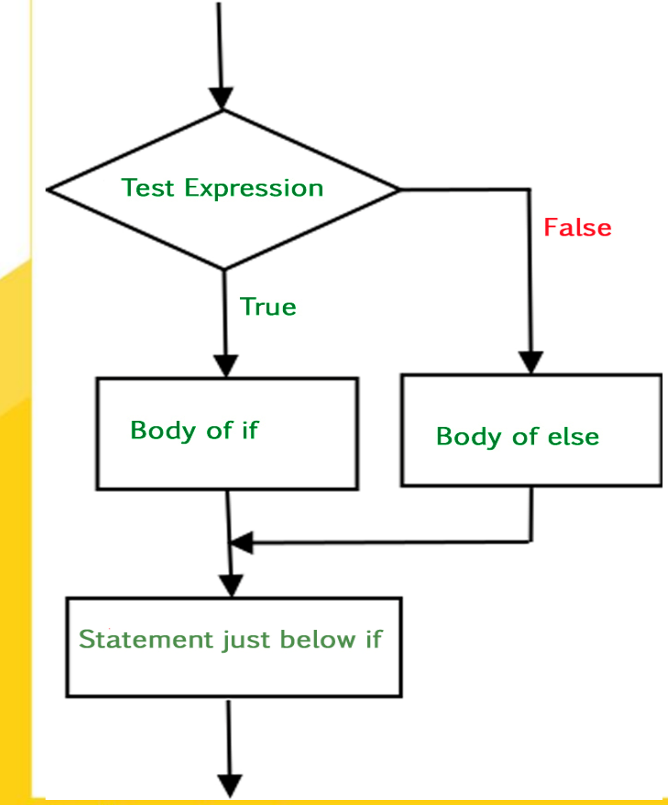


```
if condition:
        # Statements to execute if condition is true
```

```
i = 10

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

If…else statement



if (*condition*):

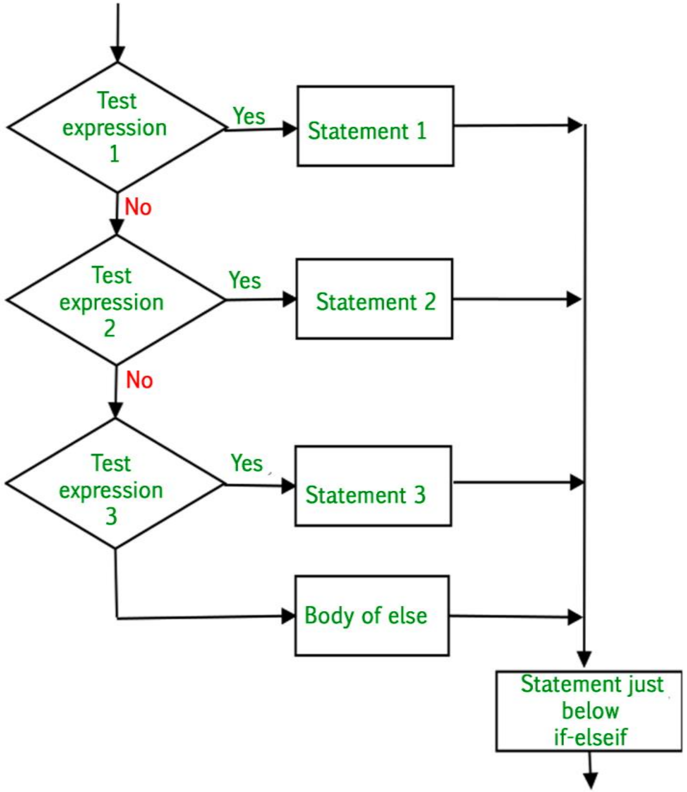
# Executes this block if condition is true

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")
```

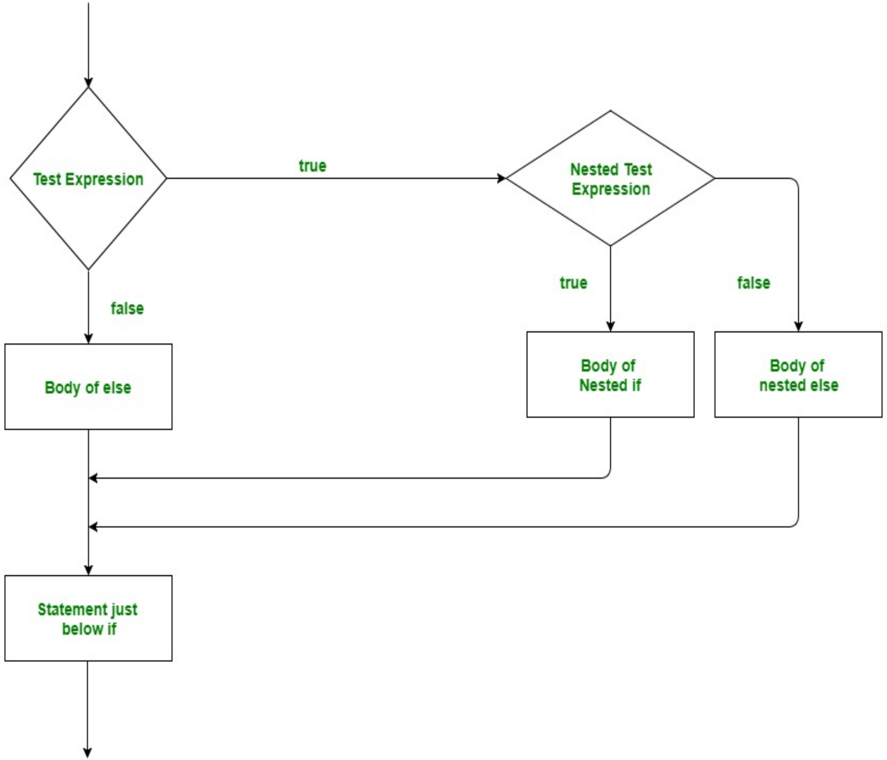
If… elif… else Statement



```
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)
```

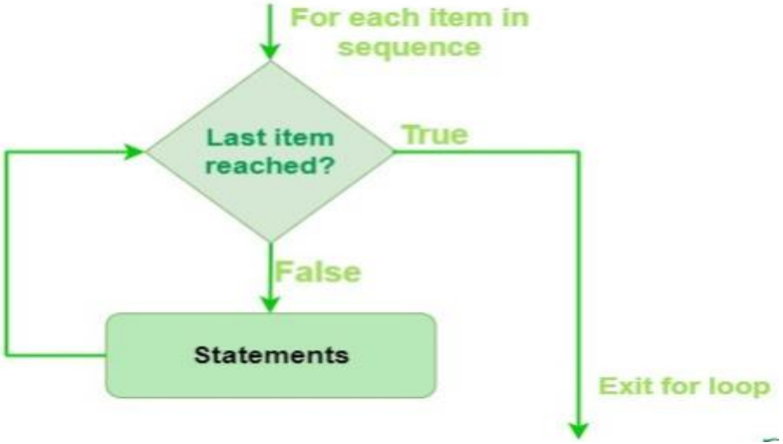
Loop Control Statements

- *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

for Loop Statements

➔ 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*.



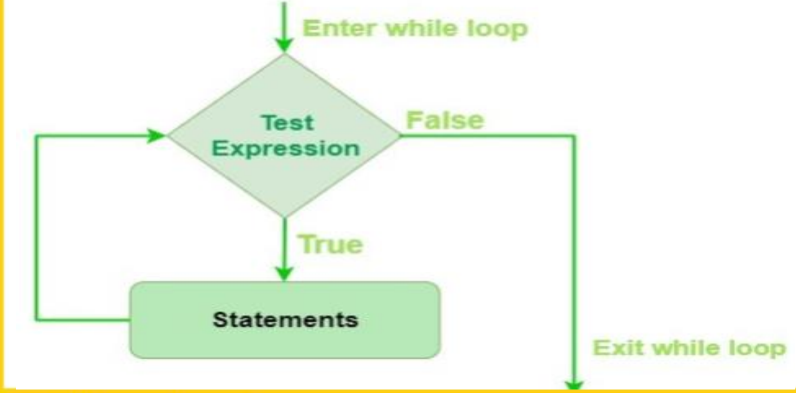
```
for variable_name in sequence :
        statement_1
        statement_2
        ....
```

```
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.

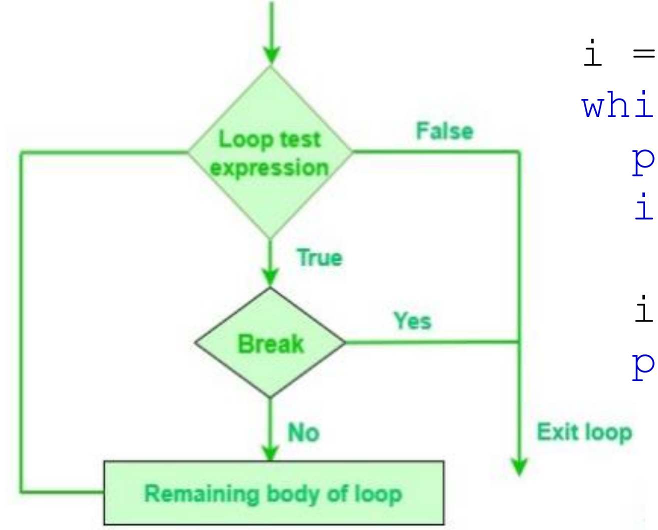
```
range(start_number, last_number, increment_value)
```

```
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 <u>*ended immediately*</u> 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)
```

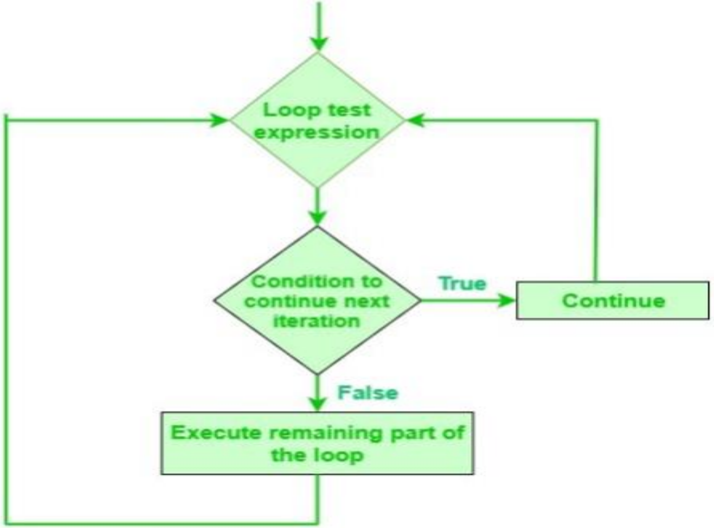
```
colors = ["blue", "green", "red"]
for x in colors:
  print(x)
  if x == "green":
    break
  print(x)
```

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.

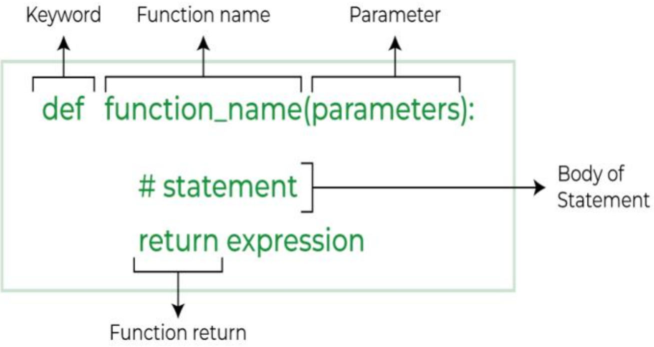
```
i = 0
while i < 7:
  i += 1
  if i == 4:
    continue
  print(i)
```

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


Loop test expression
Condition to continue next iteration
True
Continue
False
Execute remaining part of the loop

Function

➔ Definition syntax:



```
def  function_name(parameters):

      # statement

      return expression
```

Ex:

```
# 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.

Ex:

```
# 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.

Ex:

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

Ex:

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

Types of Arguments

➔ A ***variable length argument*** pass a variable number of arguments to a function using special symbols:

- ◆ *args (Non-Keyword Arguments)

Ex:

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

```
Welcome
to
VKU
```

- ◆ **kwargs (Keyword Arguments)

Ex:

```
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
```

File Handling

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

Opening file

```
File_object=open(filename, mode)
```

➔ Using the function `open():`

- *`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.

Ex:

```
# 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)
```

Reading file

```
File_object.read(size)
```

→ Using the function `read()`:

- `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()` function 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 function `write()` to insert a string in a single line in the text file and the function `writelines()` 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)
```

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

```
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 function close()

```
# 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())
```

Exception Handling

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

Try and Except Statement

Catching Exceptions

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

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

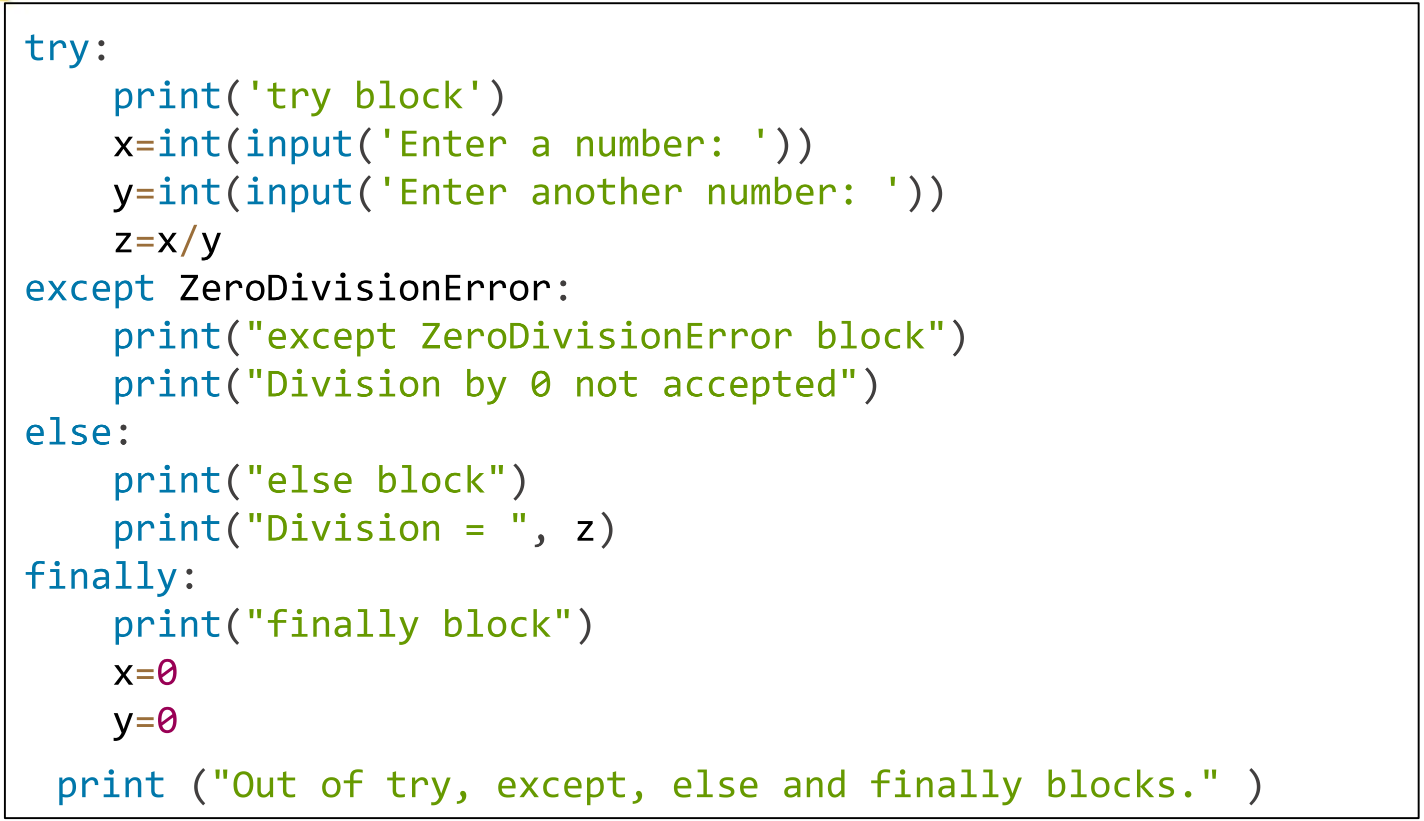
```
Ex:
try:
    a=5
    b='0'
    print(a/b)
except:
    print('Some error occurred.')
 print("Out of try except blocks.")
```

Try with Else and Finally Clauses

- ➔ The else block gets processed if the try block is found to be exception free (no exception).
- ➔ The final 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
```

An Example



```python
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." )
```