Chapter 2.
Python Basics

Chapter Content

- 1. Keywords
- 2. Variables - Constants
- 3. Operators - Expression
- 4. Statements - Statement Block - Comments
- 5. Simple Data types
- 6. Structured Data Types
- 7. Control flow statements
- 8. Loop control statements
- 9. Functions
- 10. File Handling
- 11. Exception Handling

1. Keywords

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 (a→z), numeric characters (0-9) and underscores (_)
  - Don't use Python's keyword to name variables
  - 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

2. Variables - Constants

- 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

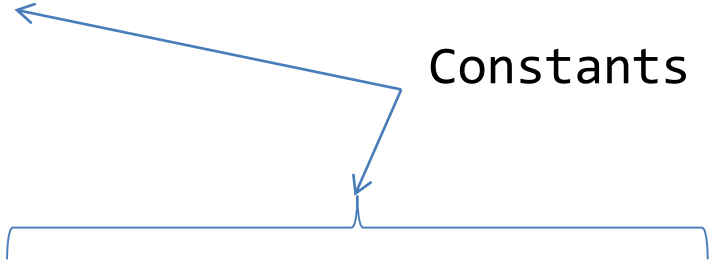
```
x = "Orange"
print(x)
```
➔ Orange

- 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
```




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

3. Operators - Expression

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

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

4. Statement - Command Block - Comments

- 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 (;)

4. Statement - Command Block - Comments

- 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")
```

4. Statement - Command Block - Comments

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

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

- Comments for a paragraph use the """

```
# 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!")
```

5. Simple Data types: 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
```

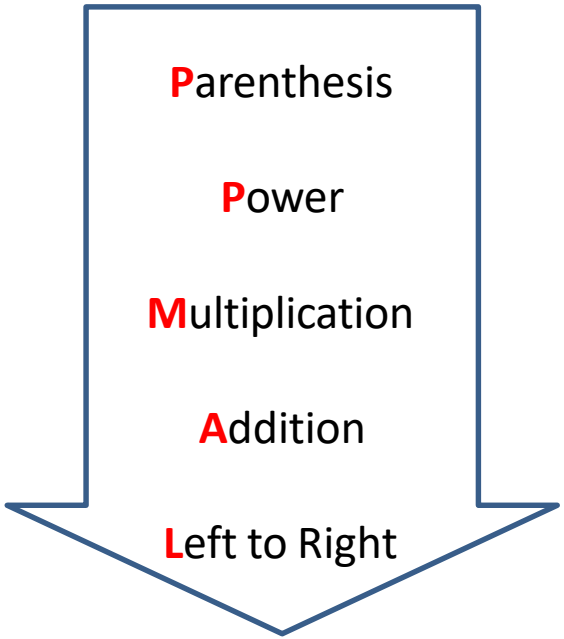
- Convert from one type to another with the **int()**, **float()** and **complex()** methods

5. Simple Data types: Numeric Types

- 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
```

5. Simple Data types: Numeric Types

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

Function Name	Describe	Example	Result
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 = 3.1415926535897 93 e = 2.718281828459045

5. Simple Data types: 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.

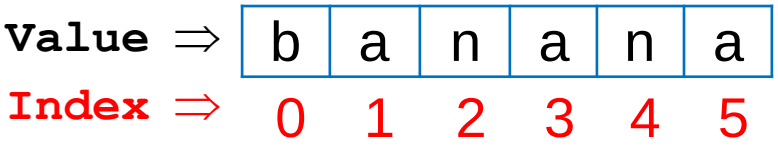
```
longer = " " " This string has
          multiple lines " " "
```

5. Simple Data types: String Type

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

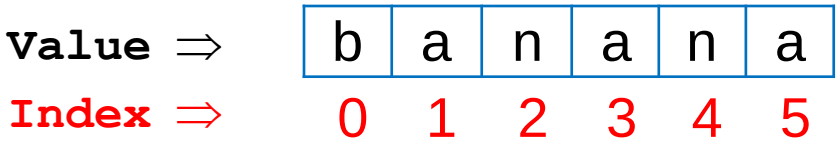
+	Concatenation operator	`Str1 + str2` ➔ `"Helloworld"`
*	Repetition operator	`str1* 3` ➔ `"HelloHelloHello"`
[]	Slice operator	`str1[4]` ➔ `"o"`
[:]	Range Slice operator	`Str3 = Str1 + str2` `str3[5:10]` ➔ `"world"`
in	Membership operator (in)	`"w" in str2` ➔ `True`
not in	Membership operator (not in)	`"e" not in str1` ➔ `False`

- Access to string's character: StringName[index]
  - The 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
  - The Index value can be the value of the expression




```
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“




```
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[: index]

- Extract the substring, starting at the character with "index" to the last character: StringName[index:]

- Get the whole string : StringName[:]

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

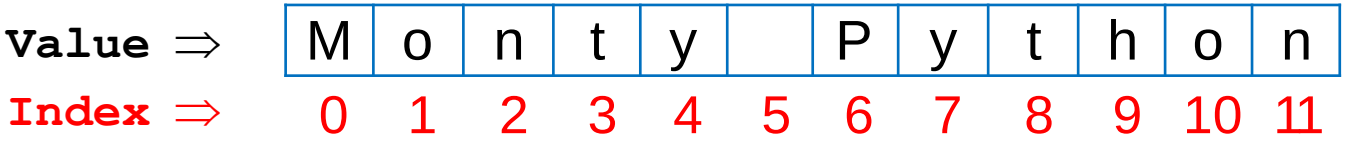
```
fruit = 'banana'

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

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

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

5. Simple Data types: String Type




```
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
```

5. Simple Data types: String Type

- 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
```

5. Simple Data types: String Type

- 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**(SubStr1, SubStr2):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
```

5. Simple Data types: String Type

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

5. Simple Data types: String Type

- 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          ⇒ True
```

  - Use the symbol (**\n**) to insert a newline into the string
  - Example:

```
print("Hello\nJoin!")      Hello
⇒                          Join!
```

  - Use the symbol (**\t**) to insert a Tab
  - Example:

```
print("Hello\tJoin!") ⇒ Hello   Join!
```

- 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): Convert string to an integer number
  - **float**(FloatStringName): Convert string to 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
```

- More 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()	….

5. Simple Data types: String Type - Summary

- Some methods
  - `StringName.title()`
  - `StringName.upper()`
  - `StringName.lower()`
  - `StringName.rstrip()`
  - `StringName.lstrip()`
  - `StringName.strip()`
  - `StringName.replace(SubStr 1, SubStr 2)`
  - `StringName.find(string to find)` ⇒ Return index
  - `StringName.find(string to find, location)` ⇒ Return index
  - `StringName.isupper()` ⇒ Return True/False
  - `StringName.islower()` ⇒ Return True/False

5. Simple Data types: String Type - Summary

- Some Operators:
  - Concatenating: (+)
  - Repeat number of times string value : (*)
  - Check if a string is in another string : (in) ⇒ Return True/False
  - Insert a newline into the string : (\n)
  - Insert a Tab : (\t)

5. Simple Data types: String Type - Summary

- Some functions:
  - `len`(StringName)
  - `int`(IntegerStringName)
  - `float`(FloatStringName):
- Access to string's character/ Extract the substring:
  - StringName[index]
  - StringName[index1 : index2]
  - StringName[: index]
  - StringName[index số :]
  - StringName[:]

5. Simple Data types: 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

5. Simple Data types: Boolean Type

- 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

- List
- Set
- Tuple
- Dictionary

- A list is like a dynamically sized array used to store multiple items.
- Items are separated by commas and enclosed in square brackets **[]**.

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

- Key Properties of a List
  - Mutable: Elements can be changed after creation.
  - Ordered: Maintains insertion order.
  - Heterogeneous: Can store elements of different data types.
  - Allows Duplicates: Repeated values are allowed.

6. Structured Data Types: List Initialization

- Using square brackets []

```
# an empty list
L1= list[]
L4= []
# a list of 3 items
L2= list['banana','apple', 'kiwi']
L3= [2,3,5]
```

- Using list multiplication

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

- Using list() constructor

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

- Using list comprehension

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

6. Structured Data Types: Operations on List

- Accessing List Items
- Updating List Items
- Modify List Items
- Insert Item into a List
- Append Item to a List
- Removing Items from a List
- Extending a List
- Sorting a List
- Statistics on Lists
- …

- Accessing List Items:
  - You can access elements in a list by referring to their index number inside square brackets `[]`.
  - This is similar to accessing characters in a string.
  - Example:

```
list1=["apple", "banana", "cherry"]
print(list1[0])          ➔ "apple"
print(list1[-1])         ➔ "cherry"
print(list1[0:2])        ➔ ["apple", "banana"]
```

6. Structured Data Types: Operations on List

- Updating List Items:
  - List items can be modified by referring to their index number and assigning a new value.
  - Syntax: ListName[index] = new_value
  - ➔ Lists are mutable, meaning their contents can be changed after creation.
  - Example:

```
list1=["apple", "banana", "cherry"]
print(list1)      ➔ ["apple", "banana", "cherry"]
List1[0] = "Newapple"
print(list1)      ➔ ["NewApple", "banana", "cherry"]
```

- Modify List Items
  - Lists in Python are mutable ➔ you can change the value of an item after the list is created.
  - Use the index to access and modify a specific element.
  - Example:

```
my_list = ["apple", "banana", "cherry"]
my_list[1] = "blueberry"

print(my_list)     ➔  ['apple', 'blueberry', 'cherry']
```

  - You can also modify multiple items using slicing:

```
my_list[0:2] = ["kiwi", "mango"]
print(my_list)      ➔ = ["kiwi", "mango ", "cherry"]
```

- Insert Item into a List
  - Use the **`insert()`** method to add an item at a specific index in the list.
  - Syntax: list.**`insert`**(index, item)
  - Example:

```
my_list = ["apple", "banana", "cherry"]
my_list.insert(1, "orange")

print(my_list)  ➔ ['apple', 'orange', 'banana', 'cherry']
```

  - ➔ Existing elements are shifted to the right.
  - ➔ Does not replace the item at the index

- Append Item to a List
  - Use the **append()** method to add a single item to the end of the list.
  - Syntax: list.**append**(item)
  - Example:

```
my_list = ["apple", "banana"]
my_list.append("cherry")

print(my_list)    ➔ ['apple', 'banana', 'cherry']
```

  - ➔ Adds one item only.
  - ➔ Use **append()** when adding a single element.

6. Structured Data Types: Operations on List

- Removing Items from a List: Python provides several ways to remove items from a list:
  - **remove**(value): Removes the first occurrence of the specified value➔Raises an error if the value is not found.
  - Example:

```
my_list = ["apple", "banana", "cherry"]
my_list.remove("banana")
print(my_list )    ➔['apple', 'cherry']
```

  - **pop**(index): Removes the item at the specified index ➔ If no index is provided, removes the **last item**.
  - Example:

```
my_list.pop(1)
print(my_list)   ➔ ['apple', 'cherry']
```

  - **clear**(): Removes **all items** from the list.
  - Example:

```
my_list.clear()
print(my_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) L1.clear()              ⇒      [ ]
```

Del ListName[index] ➔ Remove item at index

Example:

```
L1 = [10,20,30]
del L1[2]        ➔   [10,20]
```

6. Structured Data Types: Operations on List

- Some methods and function:

• sort() method: Sort items in a List in ascending order • reverse() method: Reverse the order of items in the List • extend() method: Add all elements of a List to another list • index() method: Returns the index of the first matched item • count() method: Returns the count of the number of items passed as an argument • copy() method: Returns a copy of the list • split() method • join() method	• max() function • min() function • len() function • list() function • sorted() function

- Example:
  - Method 1:

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

  - Method 2:

```
L2 = [ "a", "d", "c", "b"]
print(L2)          ⇒          [ "a", "d", "c", "b"]
print(sorted(L2)) ⇒           [ "a", "b", "c", "d"]
print(L2)          ⇒          [ "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 (character separator) to a list Characters : 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"]
```

6. Structured Data Types: Operations on Lists

- 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(",")**

6. Structured Data Types: Operations on Lists

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

```
StringName = StringValue
ListName = [item1, item2, …, itemLast]
StringName.join(ListName)
```

➔ Return 1 string: `item1StringValueitem2StringValue … 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
```

6. Structured Data Types: Operations on Lists

- Merge 02 lists: Used to **extend()** menthod
- Syntax: ListName1.**extend**(ListName2)
- Append ListName2 into ListName1
- 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"]
```

6. Structured Data Types: Operations on Lists

- Statistics on Lists:
  - **max**(ListName)
  - **min**(ListName)
  - ListName.**count**(Item)
- 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
```

6. Structured Data Types: List - Summary

- List Initialization
  - NameList =[]
  - NameList = list()
  - NameList = [item1, item2, …]
- Access to List Items
  - NameList[index]
- Convert a Item
  - NameList[index].`upper()`
  - NameList[index].`lower()`
  - NameList[index].`title()`
- Append/Insert new item:
  - NameList.`append`(Item)
  - NameList.`insert`(Index, Item)
- Delete to List Items:
  - NameList.`pop`(index)
  - NameList.`pop`()
  - NameList.`remove`(item)
  - NameList.`clear`()
  - `del` NameList[index]
- List arrangement:
  - NameList.`sort`()
  - NameList.`sort`(reverse=True)
  - `sorted`(NameList)
- 7. Reverse List:
  - NameList.`reverse`():

6. Structured Data Types: List - Summary

- Return list's length
  - `len`(ListName)
- Split list:
  - ListName = StringName.`split`()
  - ListName =StringName.`split`(",")
- Convert a string to a list of character:
  - ListName =`list`(StringName)
- Concatenate a string into a list to create a string:
  - StringName.`join`(ListName)
- extend 2 lists:
  - ListName1.`extend`(ListName2)
- Statistics:
  - `max`(ListName)
  - `min`(ListName)
  - ListName.`count`(item)
- List Comparison :
  - <, >, ==, !=, <=, >=

6. Structured Data Types: Tuple

- Tuples are used to store multiple items.
- Items are separated by commas and enclosed in parentheses ( )
- A tuple is a collection which is ordered and unchangeable
- Example:

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

6. Structured Data Types: Tuples 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'>
```

6. Structured Data Types: Operations on Tuple

- Access Tuple
- Update Tuple
- Unpacked
- Loop through a Tuple
- Join Tuples

- Access Tuple:
  - You can access items in a tuple by referring to their index number, inside square brackets [ ]
  - Tuples are **ordered**, so the position of elements does not change.

Ex1:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```
➔ banana

Ex2:

```
thistuple = ("apple","banana","cherry")
print(thistuple[-1])
```
➔ cherry

Ex3:

```
thistuple = ("apple","banana","cherry","orange")
print(thistuple[2:3])
```
➔ cherry

- Update Tuple:
  - Tuples are immutable, meaning that once they are created, their elements cannot be modified, added or removed
  - ➔ Since tuples are immutable, you can convert a tuple into a list, modify the list, and then convert it back into a tuple..
- Example:

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

- Unpacked Tuple: You can extract individual values from a tuple by assigning them to variables in a single statement
- Example:

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

- Loop through a tuple
  - To loop through a tuple, you can use a for loop.
  - Example:

```
my_tuple = ("apple", "banana", "cherry")
for item in my_tuple:
    print(item)
```

apple
banana
cherry

  - You can also use **range()** and indexing
  - Example:

```
for i in range(len(my_tuple)):
    print(my_tuple[i])
```

6. Structured Data Types: Operations on Tuple

- Join tuples:
- Ex 1:

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

- Ex 1:

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

- Note: We can used the "**+**" and "*****" operator for list data type

- Tuple Methods

Method	Description
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

6. Structured Data Types: Set

- Used to store multiple items in a single variable.
- A set is an unordered, unchangeable (immutable), and unindexed collection.
- Items are separated by commas and enclosed in curly braces {}.
- Sets Initialization:
  - Ex1:

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

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

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

6. Structured Data Types: Operations on Set

- Accessing Items in a Set
- Adding Elements to a Set
- Removing Items from a Set
- Joining Sets
- Other Set Operations

- Accessing Items in a Set:
  - You cannot access items in a set by referring to an index, because sets are unordered and unindexed.
  - However, you can:
    - Loop through set items using a `for` loop.
    - Check for membership using the `in` keyword.
  - Example:

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

```
apple
Banana
cherry
```

- Adding Elements to a Set
  - Use the **add**() method to insert a single element into a set.
  - If the element already exists, the set remains unchanged ➔ You cannot add duplicate values to a set.
  - Example 1:

```
thisset = {"apple", "banana"}
thisset.add("banana")
print(thisset) ➔{'cherry', 'apple', 'banana'}
```

- Removing Items from a Set
  - You can remove elements from a set using either the **remove**() or **discard**() method:
  - Example 1:

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

  - Example 2:

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

6. Structured Data Types: Operations on Set

- Joining Sets: You can combine two or more sets using:
  - **union()** method
    - Returns a new set containing all unique elements from both sets.
    - ➔ Original sets remain unchanged.
  - **update()** method
    - Adds all items from one set into another.
    - ➔ The original set is modified in place.
  - Example:

```
set1 = {"a","b","c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
➔ {"a","b","c", 1, 2, 3}
```

```
set1 = {"a","b","c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)
➔ {"a","b","c", 1, 2, 3}
```

- Other Set Operations

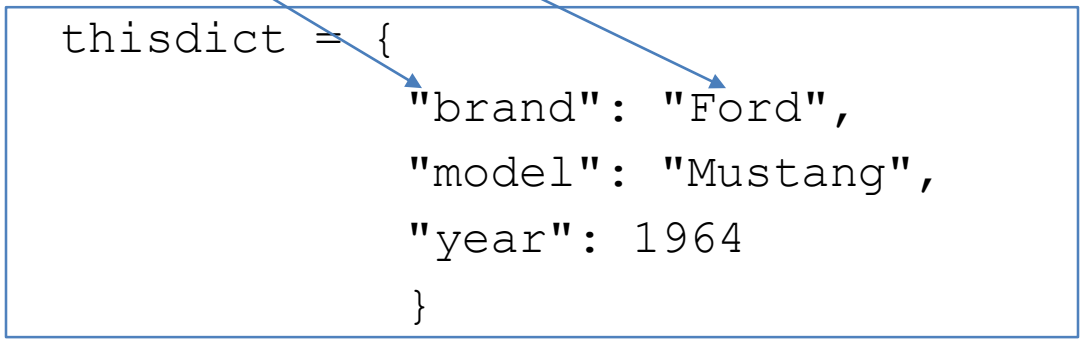
Method	Description
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified 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)

- Other Set Operations

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
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
update()	Update the set with the union of this set and others

6. Structured Data Types: Dictionary

- A dictionary is a collection items which is ordered*, changeable and do not allow duplicates.
- Items have <key> : <value> pairs
- Items are separated by commas and written with curly brackets { }
- Example:



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

6. Structured Data Types: Dictionary Initialization

- Example:

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

➔ {"brand": "Ford","model":"Mustang", "year": 1964}

6. Structured Data Types: Operations on Dictionary

- Accessing Dictionary Items
- Changing Dictionary Items
- Adding Items to a Dictionary
- Removing Items from a Dictionary
- Loop Dictionary
- Some other methods

- Accessing Dictionary Items
  - You can access the value of a dictionary item by referring to its key name inside square brackets []
- Example:

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

- Changing Dictionary Items
  - You can change the value of a specific item by referring to its key name.
  - Used to **update()** method will update the dictionary with the items from the given argument

```
student = {
            "name":"Alice",
            "age": 21
            }

student["age"] = 22
print(student)

➔ {'name': 'Alice', 'age': 22}
```

```
thisdict = {
             "brand": "Ford",
              "year":  1964
             }

thisdict.update({"year": 2020})
print(thisdict)

 ➔ {'brand': 'Ford', 'year': 2020}
```

- Adding Items to a Dictionary
  - To add a new item, use a new key and assign a value to it using square brackets [ ].
  - Example:

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

➔ {'brand':'Ford','model':'Mustang','year':1964, 'color':'red'}

- Removing Items from a Dictionary
  - **Pop()** method: removes the item with the specified key name
  - Example:

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

➔ {'brand': 'Ford', 'year': 1964}

- Removing Items from a Dictionary
  - **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)
```

➔ {'brand': 'Ford', 'model': 'Mustang'}

- Removing Items from a Dictionary
  - **del** command:delete the dictionary completely
  - **clear()** method: empties the dictionary
  - Example:

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

➔ NameError: name 'thisdict' is not defined

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

➔ { }

6. Structured Data Types: Operations on Dictionary

- Loop dictionary: the return value are the keys of the dictionary, but there are methods to return the values as well.
- Example 1: Print all key names in the dictionary, one by one:

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

→
brand
model
year

- Example 2: Print all values in the dictionary, one by one:

```
thisdict = {"brand": "Ford","model": "Mustang","year": 1964}
for x in thisdict:
   print(thisdict[x])
```

→
Ford
Mustang
1964

6. Structured Data Types: Operations on Dictionary

- Example 3: **values()** method to return values of a dictionary:

```
thisdict ={"brand":"Ford","model":"Mustang","year":1964}
for x in thisdict.values():
   print(x)
```

→ Ford
Mustang
1964

- Example 4: **keys()** method to return the keys of a dictionary:

```
thisdict ={"brand":"Ford","model":"Mustang","year": 1964}
for x in thisdict.keys():
   print(x)
```

→ brand
model
year

- Example 5: **items()** method to through both keys and values

```
thisdict ={"brand":"Ford","model":"Mustang","year": 1964}
for x,y in thisdict.items():
   print(x,y)
```

→ brand Ford
model Mustang
year 1964

6. Structured Data Types: Operations on Dictionary

- Some other 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

6. Structured Data Types: Operations on Dictionary

- Some other methods

Method	Description
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

6. Structured Data Types: Dictionary List

- A list whose elements are dictionaries is called a Dictionary List.

DictionaryName_A = { key A1 : value A1, key A2 : value A2, …}

DictionaryName_B = { key B1 : value B1, key B2 : value B2, …}

DictionaryName_C = { key C1 : value C1, key C2 : value C2, …}

- Then:

DictionaryList = [DictionaryName_A, DictionaryName_B, DictionaryName_C]

- Example:

```
bong_0 = {"color": "green", "ban_kinh": 5}
bong_1 = {"color": "yellow", "ban_kinh": 10}
Danh_Sach_Bong = [bong_0, bong_1]
for bong in Danh_Sach_Bong:
    print(bong)
```

{"color": "green", "points": 5}

{"color": "yellow", "points": 10}

6. Structured Data Types: List Dictionary

- A Dictionary where each <value> is a list is called a List Dictionary or List in Dictionary.

List_A = [item A1, item A2, item A3, …]

List_B = [item B1, item B2, item B3, …]

- Then

DictionaryName = {key 1: List_A, key 2: List_B}

- Ví dụ:

```
NNLT = {"hung": ["python", "ruby"], "dung": ["C"]}
for name, language in NNLT.items():
    print("\n" + name.title() + "use:")
    for NN in language:
        print("\t" + NN.title())
```

Hung use:
	Python
	Ruby
Dung use:
	C

6. Structured Data Types: Dictionary of Dictionary

- A dictionary where each value is a dictionary is called a Dictionary of Dictionaries.

DictionaryName = {Key 1: DictionaryName A, Key 2: DictionaryName B, …}

- Example

```
users = {
          "user1": {"name": "Hung", "password": "01234"},
          "user2": {"name": "Dung", "password": "56789"}
        }
for user, user_infor in users.items():
    print("\nUser: " + user)
    Name = user_infor["name"]
    Pass = user_infor["password"]
    print("\tName: " + name.title())
    print("\tPassword: " + Pass.title())
```

User: user1
Name: Hung
Password: 012345

User: user2
Name: Dung
Password: 56789

Dictionary – Summary

- 1. Initializing a Dictionary
  - `dict_name = {}` ➔ Empty dictionary
  - `dict_name = dict()`
  - `dict_name = {"key1": value1, "key2": value2, ...}`
- 2. Accessing a Single Value
  - `dict_name[key]`
  - `dict_name.get(key, default_value)`
- 3. Accessing All Items (key-value pairs)
  - `dict_name.items()` ➔ Returns all (key, value) pairs as tuples
- 4. Accessing All Keys
  - `dict_name.keys()` ➔ Returns a view of all keys
- 5. Accessing All Values
  - `dict_name.values()` ➔ Returns a view of all values

Dictionary – Summary

- 6. Add an element:
  - dictionary_name[new_key_to_add] = value
- 7. Update the value of an existing key:
  - dictionary_name[key] = new_value
- 8. Delete an element:
  - del dictionary_name[key_to_delete]
- 9. Check if a key exists in the dictionary:
  - key in dictionary_name.keys()
- 10. Sort the keys in the dictionary:
  - sorted(dictionary_name.keys())
- 11. Create a set of unique values from the dictionary:
  - set(dictionary_name.values())

Dictionary – Summary

- 12. List of Dictionaries:
  - Dictionary_A = { key_A1: value_A1, key_A2: value_A2, ... }
  - Dictionary_B = { key_B1: value_B1, key_B2: value_B2, ... }
  - Dictionary_C = { key_C1: value_C1, key_C2: value_C2, ... }
  - List_of_Dictionaries = [Dictionary_A, Dictionary_B, Dictionary_C]
- 13. Dictionary of Lists:
  - List_A = [Element_A1, Element_A2, Element_A3, ...]
  - List_B = [Element_B1, Element_B2, Element_B3, ...]
  - Dictionary = { Key_1: List_A, Key_2: List_B }
- 14. Dictionary of Dictionaries:
  - Dict_A = { key_A1: value_A1, key_A2: value_A2, ... }
  - Dict_B = { key_B1: value_B1, key_B2: value_B2, ... }
  - Dictionary = { Key_1: Dict_A, Key_2: Dict_B, ... }

Tuple – Summary

- 1. Initialize Tuples
  - tuple_name = () tuple_name = (element1, element2, ...)
- 2. Operations on Tuples
  - Most List operations can be applied to tuples, except:
    - Once a tuple is created, its contents cannot be changed.
    - The following methods cannot be used with tuples: sort(), append(), reverse(), ...
- 3. Note: When using the items() method to access the elements of a dictionary, the result is a list of tuples➔ So, to sort them, use the sorted() function as follows:
    - First, use the items() method on the dictionary.
    - Then apply the sorted() function to the result.
- 4. Convert a list to a tuple
  - tuple_name = tuple(list_name)

Zip() Function

- Creating dictionaries from two lists: one for keys, one for values.

```
keys = ['name', 'age', 'city']
values = ['Alice', 25, 'New York']

dictionary = dict(zip(keys, values))
print(dictionary)
```

➔ {'name': 'Alice', 'age': 25, 'city': 'New York'}

- To iterate through multiple lists at the same time

```
subjs = ['Math', 'Science', 'English']
scores = [88, 92, 85]

for subject, score in zip(subjs,scores):
    print(subject, score)
```

➔Math 88
➔Science 92
➔English 85

Zip () Function

- Merging the lists

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
combined = list(zip(names, ages))
print(combined)
```

➔ `[('Alice',25),('Bob',30),('Charlie',35)]`

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
list3 = [True, False, True]
combined = list(zip(list1, list2, list3))
print(combined)
```

➔ `[(1, 'a', True), (2, 'b', False), (3, 'c', True)]`

```
numbers = range(1, 4)
letters = ['a', 'b', 'c']
result = list(zip(numbers, letters))
print(result)
```

➔ `[(1, 'a'), (2, 'b'), (3, 'c')]`

```
list1 = [1, 2, 3]
list2 = ['a', 'b']
combined = list(zip(list1, list2))
print(combined)
```

➔ `[(1, 'a'), (2, 'b')]`

Zip () Function

- To compare elements of Lists.

```
list1 = [1, 2, 3]
list2 = [1, 4, 3]
comparison = [a == b for a, b in zip(list1, list2)]
print(comparison)   ➔[True, False, True]
```

- Splitting paired data into separate lists

```
combined = [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
names, ages = zip(*combined)
print(names)   ➔('Alice', 'Bob', 'Charlie')
print(ages)    ➔(25, 30, 35)
```

Câu 1: Python là một ngôn ngữ lập trình thuộc loại nào?
A. Ngôn ngữ lập trình biên dịch
B. Ngôn ngữ lập trình thông dịch
C. Ngôn ngữ lập trình lập trình hướng đối tượng
D. Ngôn ngữ lập trình chức năng

Đáp án: B. Ngôn ngữ lập trình thông dịch

Câu 2: Câu lệnh nào sau đây là cách khai báo
một danh sách trong Python?
A. list = {}
B. list = []
C. list = ()
D. list = <>

Đáp án: B. list = []

Câu 3: Đoạn mã Python sau sẽ in ra kết quả gì?

```
x = 5
y = 10
print(x + y)
```

A. 15
B. 510
C. Error
D. 5 10

Đáp án: A. 15

Câu 4: Hàm nào sau đây được sử dụng để đọc dữ liệu từ người dùng trong Python?

A. input()
B. print()
C. scan()
D. get()

Đáp án: A. input()

Câu 5: Câu lệnh nào sau đây sẽ tạo ra một tuple trong Python?

A. t = [1, 2, 3]
B. t = (1, 2, 3)
C. t = {1, 2, 3}
D. t = <1, 2, 3>

Đáp án: B. t = (1, 2, 3)

Câu 6: Trong Python, cách nào sau đây là đúng để kiểm tra kiểu dữ liệu của một biến?

A. type(variable)
B. typeof(variable)
C. datatype(variable)
D. type of variable

Đáp án: A. type(variable)

Câu 7: Hàm len() trong Python dùng để làm gì?
A. Trả về chiều dài của một chuỗi
B. Trả về số phần tử trong một danh sách, tuple, hoặc chuỗi
C. Kiểm tra xem chuỗi có rỗng hay không
D. Tất cả các đáp án trên

Đáp án: B. Trả về số phần tử trong một danh sách, tuple, hoặc chuỗi

Câu 8: Câu lệnh nào sau đây tạo ra một dictionary trong Python?
A. d = {1: 'a', 2: 'b'}
B. d = [1: 'a', 2: 'b']
C. d = (1: 'a', 2: 'b')
D. d = <1: 'a', 2: 'b'>

Đáp án: A. d = {1: 'a', 2: 'b'}

Câu 9: Đoạn mã sau sẽ trả về kết quả gì?

```
x = [1, 2, 3]
x.append(4)
print(x)
```

A. [1, 2, 3]
B. [1, 2, 3, 4]
C. Error
D. [4, 1, 2, 3]

Đáp án: B. [1, 2, 3, 4]

Câu 10: Hàm nào trong Python có thể kiểm tra xem một chuỗi có phải là một chuỗi số nguyên không (chỉ chứa các chữ số)?

A. isnumber()
B. isdigit()
C. isnumeric()
D. isdecimal()

Đáp án: B. isdigit()

Câu 11: Đoạn mã sau sẽ in ra kết quả gì?

```
s = "Hello, World!"
print(s.find("World"))
```

A. 0
B. 5
C. 7
D. -1

Đáp án: C. 7

Câu 12: Kết quả của đoạn mã sau là gì?

```
s = "Python Programming"
print(s.replace("o", "O"))
```

A. PythOn PrOgramming
B. PythOn Programming
C. Python PrOgramming
D. Tất cả đều sai

Đáp án: A. PythOn PrOgramming

Câu 13: Hàm nào sau đây kiểm tra xem tất cả các ký tự trong chuỗi có phải là chữ cái (không tính khoảng trắng và các ký tự đặc biệt)?

A. isalpha()
B. isletter()
C. isdigit()
D. isalphaonly()

Đáp án: A. isalpha()

Câu 14: Kết quả của đoạn mã sau là gì?

```
x = 4
y = 2
print(x / y)
```

A. 2
B. 2.5
C. 2.0
D. Error

Đáp án: C. 2.0

Câu 15: Đoạn mã sau sẽ in ra kết quả gì?

```
x = 9
print(x ** 0.5)
```

A. 81
B. 3
C. 9.0
D. 0.5

Đáp án: B. 3
Giải thích: x ** 0.5 tính căn bậc hai của x, tức là sqrt(9) = 3

Câu 16: Kết quả của đoạn mã sau là gì?

```
x = 7
y = 3
print(x // y)
```

A. 2.33
B. 2
C. 3
D. 0.33

Đáp án: B. 2
Giải thích: Phép toán // thực hiện phép chia lấy phần nguyên.

Câu 17: Đoạn mã sau sẽ in ra kết quả gì?

```
x = 15
y = 4
print(x % y)
```

A. 3
B. 2
C. 4
D. 1

Đáp án: A. 3
Giải thích: Phép toán % trả về phần dư của phép chia 15 % 4 = 3.

Câu 18: Đoạn mã sau sẽ in ra kết quả gì?

```
x = -10
y = 3
print(abs(x / y))
```

A. 3.33
B. 10
C. 3.33
D. 3.33

Đáp án: A. 3.33
Giải thích: Hàm abs() trả về giá trị tuyệt đối. -10 / 3 = -3.33 và abs(-3.33) trả về 3.33.

Câu 19: Đoạn mã sau sẽ in ra kết quả gì?

```
x = 0
y = 1
print(not x)
```

A. True
B. False
C. Error
D. 1

Đáp án: A. True
Giải thích: Toán tử not đảo ngược giá trị Boolean. Vì x = 0 (được coi là False trong Python), not x sẽ trả về True.

Câu 20: Kết quả của đoạn mã sau là gì?

```
x = 10
y = 20
print(x > y or x == 10)
```

A. True
B. False
C. Error
D. 0

Đáp án: A. True
Giải thích: Phép toán or trả về True nếu một trong các giá trị của biểu thức là True. x > y là False, nhưng x == 10 là True, do đó kết quả là True.

Câu 21: Đoạn mã sau sẽ in ra kết quả gì?

```
x = False
y = True
print(x != y)
```

A. True
B. False
C. Error
D. 0

Đáp án: A. True

Giải thích: Toán tử != kiểm tra sự khác biệt. Vì x = False và y = True, nên x != y sẽ trả về True.

Câu 22: Kết quả của đoạn mã sau là gì?

```
x = True
y = False
print(x or y)
```

A. True
B. False
C. Error
D. 0

Đáp án: A. True

Giải thích: Phép toán or trả về True nếu một trong các giá trị là True. Vì x = True, kết quả là True.

Câu 23: Đoạn mã sau sẽ in ra kết quả gì?

```
x = 0
y = -1
print(bool(x) and bool(y))
```

A. True
B. False
C. Error
D. 0

Đáp án: B. False

Giải thích: Hàm bool() chuyển đổi giá trị thành kiểu Boolean. Vì x = 0 (được coi là False), và y = -1 (được coi là True), nhưng phép toán and yêu cầu cả hai giá trị phải là True mới trả về True.

Câu 24: Kết quả của đoạn mã sau là gì?

```
x = 7
y = 7
print(x is y)
```

A. True
B. False
C. Error
D. 0

Đáp án: A. True

Giải thích: Toán tử is kiểm tra xem hai đối tượng có phải là một đối tượng duy nhất hay không. Trong trường hợp này, cả x và y đều là 7, do đó x is y trả về True.

Câu 25: Đoạn mã sau sẽ in ra kết quả gì?

```
x = 10
y = 5
print(x > 5 and y < 10)
```

A. True
B. False
C. Error
D. 0

Đáp án: A. True
Giải thích: Cả hai điều kiện x > 5 và y < 10 đều là True, do đó True and True trả về True

Câu 26: Đoạn mã sau sẽ in ra kết quả gì?

```
lst = [1, 2, 3, 4]
print(lst[2])
```

A. 2
B. 3
C. 4
D. 1

Đáp án: B. 3
Giải thích: Trong Python, chỉ số của danh sách bắt đầu từ 0, do đó lst[2] là phần tử thứ 3 của danh sách, có giá trị là 3.

Câu 27: Kết quả của đoạn mã sau là gì?

```
lst = [1, 2, 3, 4]
lst.append(5)
print(lst)
```

A. [1, 2, 3, 4, 5]
B. [1, 2, 3, 5, 4]
C. [5, 1, 2, 3, 4]
D. [1, 2, 5, 3, 4]

Đáp án: A. [1, 2, 3, 4, 5]
Giải thích: Phương thức append() thêm phần tử vào cuối danh sách, vì vậy sau khi gọi lst.append(5), danh sách sẽ là [1, 2, 3, 4, 5].

Câu 28: Đoạn mã sau sẽ in ra kết quả gì?

```
lst = [1, 2, 3, 4]
lst[1:3] = [7, 8, 9]
print(lst)
```

A. [1, 7, 8, 9, 4]
B. Lỗi
C. [7, 8, 9]
D. [1, 7, 8, 9, 2, 3, 4]

Đáp án: A. [1, 7, 8, 9, 4]
Giải thích: Câu lệnh lst[1:3] = [7, 8, 9] thay thế các phần tử tại vị trí chỉ số 1 và 2 bằng [7, 8, 9]. Danh sách mới sẽ là [1, 7, 8, 9, 4].

Câu 29: Kết quả của đoạn mã sau là gì?

```
lst = [1, 2, 3, 4]
lst.remove(3)
print(lst)
```

A. [1, 2, 4]
B. [1, 3, 4]
C. [2, 3, 4]
D. [1, 2, 3]

Đáp án: A. [1, 2, 4]
Giải thích: Phương thức remove() xóa phần tử đầu tiên trong danh sách có giá trị bằng đối số (ở đây là 3). Sau khi xóa, danh sách còn lại là [1, 2, 4].

Câu 30: Đoạn mã sau sẽ in ra kết quả gì?

```
lst = [1, 2, 3, 4]
lst.insert(2, 5)
print(lst)
```

A. [1, 2, 5, 3, 4]
B. [5, 1, 2, 3, 4]
C. [1, 2, 3, 5, 4]
D. lỗi

Đáp án: A. [1, 2, 5, 3, 4]
Giải thích: Phương thức insert(index, value) chèn giá trị vào vị trí chỉ số index. Ở đây, lst.insert(2, 5) chèn giá trị 5 vào chỉ số 2, làm danh sách trở thành [1, 2, 5, 3, 4].

Câu 31: Kết quả của đoạn mã sau là gì?

```
lst = [1, 2, 3, 4]
lst.sort()
print(lst)
```

A. [1, 2, 3, 4]
B. [4, 3, 2, 1]
C. [1, 3, 2, 4]
D. [1, 2, 4, 3]

Đáp án: A. [1, 2, 3, 4]

Giải thích: Phương thức sort() sắp xếp các phần tử trong danh sách theo thứ tự tăng dần. Danh sách ban đầu là [1, 2, 3, 4], và sau khi sắp xếp, nó vẫn giữ nguyên.

Câu 32: Đoạn mã sau sẽ in ra kết quả gì?

```
lst = [1, 2, 3, 4]
print(lst[-1])
```

A. 4
B. 3
C. 2
D. 1

Đáp án: A. 4

Giải thích: Chỉ số -1 được sử dụng để truy cập phần tử cuối danh sách. lst[-1] sẽ trả về phần tử cuối cùng, có giá trị là 4.

Câu 33: Kết quả của đoạn mã sau là gì?

```
lst = [1, 2, 3, 4]
lst.pop(2)
print(lst)
```

A. [1, 2, 3]
B. [1, 2, 4]
C. [2, 3, 4]
D. [1, 2, 4]

Đáp án: B. [1, 2, 4]

Giải thích: Phương thức pop() lấy ra phần tử tại chỉ số chỉ định (ở đây là chỉ số 2, phần tử có giá trị 3), và danh sách còn lại là [1, 2, 4].

Câu 34: Đoạn mã sau sẽ in ra kết quả gì?

```
lst = [1, 2, 3]
lst.extend([4, 5, 6])
print(lst)
```

A. [1, 2, 3, 4, 5, 6]
B. [1, 2, 3, [4, 5, 6]]
C. [4, 5, 6, 1, 2, 3]
D. [1, 2, 3]

Đáp án: A. [1, 2, 3, 4, 5, 6]

Giải thích: Phương thức extend() thêm tất cả các phần tử của danh sách vào danh sách ban đầu. Sau khi gọi lst.extend([4, 5, 6]), danh sách trở thành [1, 2, 3, 4, 5, 6].

Câu 35: Đoạn mã sau sẽ in ra kết quả gì?

```
t = (1, 2, 3, 4)
print(t[1])
```

A. 1
B. 2
C. 3
D. 4

Đáp án: B. 2

Giải thích: Chỉ số trong tuple bắt đầu từ 0, vì vậy t[1] trả về phần tử thứ 2 trong tuple, có giá trị là 2

Câu 36: Kết quả của đoạn mã sau là gì?

```
t = (1, 2, 3, 4)
t[2] = 5
```

A. (1, 2, 3, 4)
B. (1, 2, 5, 4)
C. Error
D. (1, 5, 3, 4)

Đáp án: C. Error

Giải thích: không thể thay đổi giá trị của các phần tử sau khi tạo tuple

Câu 37: Đoạn mã sau sẽ in ra kết quả gì?

```
t = (1, 2, 3, 4)
print(t[-2])
```

A. 1
B. 3
C. 2
D. 4

Đáp án: C. 2

Giải thích: Chỉ số âm dùng để truy cập từ cuối tuple. t[-2] sẽ trả về phần tử thứ 2 từ cuối, có giá trị là 2.

Tương tự như kiểu dữ liệu danh sách, tập hợp

Câu 38: Kết quả của đoạn mã sau là gì?

```
t = (1, 2, 3, 4)
t = t + (5, 6)
print(t)
```

A. (1, 2, 3, 4, 5, 6)
B. (1, 2, 3, 5, 6, 4)
C. (1, 2, 3, 4)
D. Error

Đáp án: A. (1, 2, 3, 4, 5, 6)

Giải thích: Tuple có thể được nối với nhau bằng cách sử dụng toán tử +. Sau khi nối, tuple t trở thành (1, 2, 3, 4, 5, 6).
Tương tự như danh sách, tập hợp

Câu 39: Đoạn mã sau sẽ in ra kết quả gì?

```
t = (1, 2, 3, 4)
print(t.index(3))
```

A. 0
B. 1
C. 2
D. 3

Đáp án: C. 2

Giải thích: Phương thức index() trả về chỉ số đầu tiên của phần tử trong tuple. Phần tử 3 có chỉ số là 2.

Tương tự như danh sách, tập hợp

Câu 40: Đoạn mã sau sẽ in ra kết quả gì?

```
t = (1, 2, 3, 4)
print(t.count(2))
```

A. 1
B. 2
C. 3
D. 0

Đáp án: A. 1

Giải thích: Phương thức count() trả về số lần xuất hiện của phần tử trong tuple. Phần tử 2 xuất hiện 1 lần trong tuple t.

Tương tự như danh sách, tập hợp

Câu 41: Kết quả của đoạn mã sau là gì?

```
t = (1, 2, 3, 4)
t = t * 2
print(t)
```

A. (1, 2, 3, 4, 1, 2, 3, 4)
B. (1, 1, 1, 1, 1, 1, 1, 1)
C. (2, 3, 4, 1, 2, 3, 4)
D. (1, 2, 3, 4)

Đáp án: A. (1, 2, 3, 4, 1, 2, 3, 4)
Giải thích: Phép toán * trên tuple nhân đôi các phần tử của tuple, kết quả là (1, 2, 3, 4, 1, 2, 3, 4).
Tương tự như chuỗi, danh sách, tập hợp

Câu 42: Đoạn mã sau sẽ in ra kết quả gì?

```
t = (1, 2, 3, 4)
print(t[1:3])
```

A. (1, 2)
B. (2, 3)
C. (1, 2, 3)
D. (2, 3, 4)

Đáp án: B. (2, 3)
Giải thích: Khi sử dụng slicing t[1:3], chúng ta lấy các phần tử từ chỉ số 1 đến 2 (không bao gồm chỉ số 3), vì vậy kết quả là (2, 3).

Tương tự như danh sách, tập hợp

Câu 43: Đoạn mã sau sẽ in ra kết quả gì?

```
t = (1, 2, 3, 4)
a, b, c, d = t
print(a, b, c, d)
```

A. (1, 2, 3, 4)
B. 1 2 3 4
C. [1, 2, 3, 4]
D. Error

Đáp án: B. 1 2 3 4

Giải thích: Tuple có thể được giải nén (unpacking) vào các biến. Các giá trị trong tuple t được gán lần lượt vào các biến a, b, c, d, và in ra dưới dạng 1 2 3 4.

Tương tự như dùng hàm zip() cho danh sách

Câu 44: Trong Python, tập hợp có những tính chất nào?

A. Không có thứ tự, chứa phần tử trùng lặp
B. Chứa phần tử trùng lặp, giá trị phần tử có thể thay đổi được
C. Giá trị phần tử không thay đổi được
D. Không có thứ tự, không có các phần tử trùng lặp và giá trị phần tử có thể thay đổi được

Đáp án: D

Giải thích: Tập hợp trong Python là tập hợp không có thứ tự và không chứa các phần tử trùng lặp (set tự động loại bỏ các phần tử trùng). Giá trị phần tử có thể thay đổi.

Câu 45: Nếu tạo một tập hợp từ danh sách sau: [1, 2, 2, 3, 4, 4, 5]. Kết quả sẽ là gì?
A. {1, 2, 3, 4, 5}
B. {2, 3, 4, 5}
C. {1, 2, 3, 4}
D. [1, 2, 3, 4, 5]

Đáp án: A
Giải thích: Khi tạo tập hợp từ danh sách có phần tử trùng lặp, Python tự động loại bỏ các phần tử trùng và chỉ giữ lại các phần tử duy nhất.

Câu 46: Tập hợp nào dưới đây sẽ trả về kết quả là True khi thực hiện phép toán issubset() trong Python?
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}

A. A.issubset(B)
B. B.issubset(A)
C. A.issubset({4, 5, 6})
D. B.issubset({1, 2, 3})

Đáp án: A
Giải thích: issubset() kiểm tra xem tập hợp A có phải là một phần con của tập hợp B hay không. Vì tất cả phần tử của A đều có trong B, nên kết quả là True.

Câu 47: Kết quả của phép toán A - B khi A = {1, 2, 3} và B = {3, 4, 5} là gì?

A. {1, 2, 3, 4, 5}
B. {1, 2}
C. {3, 4, 5}
D. {}

Đáp án: B
Giải thích: Phép toán A - B trả về các phần tử có trong A nhưng không có trong B. Kết quả là {1, 2}.

Câu 48: Phép toán intersection() trong Python trả về gì khi áp dụng lên các tập hợp sau?
A = {1, 2, 3}
B = {3, 4, 5}

A. {1, 2, 3, 4, 5}
B. {3}
C. {1, 2}
D. {4, 5}

Đáp án: B
Giải thích: Phép toán intersection() trả về tập hợp chứa các phần tử chung giữa hai tập hợp. Ở đây chỉ có phần tử 3 là chung giữa A và B.

Câu 49: Phép toán nào dưới đây có thể giúp bạn kiểm tra xem một tập hợp có chứa phần tử cụ thể hay không?

A. in
B. contains()
C. has()
D. is

Đáp án: A
Giải thích: Phép toán in có thể kiểm tra xem một phần tử có nằm trong tập hợp hay không.

Câu 50: Khi nào phép toán A.isdisjoint(B) trả về True?

A. Khi A và B có phần tử giống nhau
B. Khi A và B không có phần tử chung
C. Khi A là tập con của B
D. Khi A là tập hợp con của B và có phần tử giống nhau

Đáp án: B
Giải thích: Phép toán isdisjoint() trả về True khi hai tập hợp không có phần tử chung.

Câu 51: Tạo một tập hợp A = {1, 2, 3} và thực hiện lệnh A.pop(). Kết quả sẽ là gì?

A. Tập hợp A không thay đổi
B. Lỗi TypeError
C. Một phần tử bất kỳ trong tập hợp A sẽ bị loại bỏ và trả về
D. Tập hợp A bị xóa hoàn toàn

Đáp án: C
Giải thích: Phương thức pop() sẽ loại bỏ và trả về một phần tử bất kỳ trong tập hợp (vì tập hợp không có thứ tự). Tập hợp còn lại sẽ thiếu phần tử đó.

Câu 52: Trong Python, nếu bạn cố gắng truy cập một khóa không tồn tại trong từ điển, điều gì sẽ xảy ra?

A. Trả về giá trị None
B. Lỗi KeyError
C. Trả về giá trị mặc định
D. Trả về một từ điển rỗng

Đáp án: B
Giải thích: Nếu truy cập một khóa không tồn tại trong từ điển, Python sẽ gây ra lỗi KeyError.

Câu 53: Kết quả của đoạn mã sau sẽ là gì?

```
d = {'a': 1, 'b': 2, 'c': 3}
d['d'] = 4
del d['a']
Print(d)
```

A. {'b': 2, 'c': 3, 'd': 4}
B. {'a': 1, 'b': 2, 'c': 3, 'd': 4}
C. {'b': 2, 'c': 3}
D. {'a': 1, 'c': 3, 'd': 4}

Đáp án: A
Giải thích: Sau khi thêm khóa 'd' và xóa khóa 'a', từ điển sẽ có các phần tử còn lại là 'b': 2, 'c': 3, 'd': 4.

Câu 54: Phương thức nào dưới đây sẽ trả về một danh sách các khóa trong từ điển?

A. dict.keys()
B. dict.items()
C. dict.values()
D. dict.get()

Đáp án: A
Giải thích: Phương thức keys() trả về một "view" chứa tất cả các khóa trong từ điển.

Câu 55: Câu lệnh nào sẽ làm thay đổi giá trị của khóa 'key' trong từ điển
d = {'key': 'old_value'} thành 'new_value'?

A. d['key'] = 'new_value'
B. d.set('key', 'new_value')
C. d.update('key', 'new_value')
D. d['key'].update('new_value')

Đáp án: A
Giải thích: Để thay đổi giá trị của một khóa trong từ điển là sử dụng cú pháp d['key'] = 'new_value'.

Câu 56: Phương thức nào dưới đây sẽ giúp xóa tất cả các phần tử trong từ điển mà không xóa chính từ điển đó?

A. clear()
B. remove()
C. pop()
D. delete()

Đáp án: A
Giải thích: Phương thức clear() sẽ xóa tất cả các phần tử trong từ điển, nhưng không xóa chính từ điển.

Câu 57: Phép toán nào sẽ cho kết quả là một từ điển chứa tất cả các phần tử từ hai từ điển d1 và d2?
d1 = {'a': 1, 'b': 2}
d2 = {'b': 3, 'c': 4}

A. d1 | d2
B. d1 & d2
C. d1 + d2
D. d1.update(d2)

Đáp án: A
Giải thích: Phép toán | (hoặc union) giữa hai từ điển sẽ trả về một từ điển mới chứa tất cả các phần tử

Câu 58: Kết quả của đoạn mã sau sẽ là gì?
d = {'a': 1, 'b': 2}
d.update({'b': 3, 'c': 4})

A. d == {'a': 1, 'b': 2, 'c': 4}
B. d == {'a': 1, 'b': 3, 'c': 4}
C. d == {'a': 1, 'b': 2, 'c': 4}
D. d == {'a': 1, 'b': 3}

Đáp án: B
Giải thích: Phương thức update() sẽ cập nhật các giá trị của các khóa trong từ điển. Nếu khóa đã tồn tại, giá trị của khóa đó sẽ được thay đổi. Do đó, 'b' sẽ có giá trị mới là 3, và 'c' sẽ được thêm vào từ điển.

Câu 59: Kết quả của đoạn mã sau sẽ là gì?
x = '123'
y = int(x)

A. '123'
B. 123.0
C. 123
D. None

Đáp án: C
Giải thích: Hàm int() chuyển đổi chuỗi '123' thành số nguyên 123.

Câu 60: Phép chuyển đổi nào dưới đây sẽ gây ra lỗi khi áp dụng cho chuỗi "12.34"?

A. int("12.34")
B. float("12.34")
C. str(12.34)
D. bool("12.34")

Đáp án: A
Giải thích: Hàm int() không thể chuyển đổi chuỗi "12.34" (là một số thực) thành kiểu dữ liệu int. Còn các phép chuyển đổi khác đều hợp lệ.

Câu 61: Kết quả của đoạn mã sau là gì?

```
x = [1, 2, 3]
y = tuple(x)
```

A. y == [1, 2, 3]
B. y == (1, 2, 3)
C. y == '1, 2, 3'
D. y == (1, 2, 3, 4)

Đáp án: B

Giải thích: Hàm tuple() chuyển danh sách [1, 2, 3] thành tuple (1, 2, 3).

Câu 62: Kết quả của đoạn mã sau là gì?

```
x = 5.75
y = str(x)
print(y)
```

A. 5.75
B. "5.75"
C. '575'
D. 5

Đáp án: B

Giải thích: Hàm str() chuyển số thực 5.75 thành chuỗi "5.75".

Câu 63: Kết quả của đoạn mã sau là gì?

```
x = [1, 2, 3,3]
print(set(x))
```

A. [1, 2, 3]
B. {1, 2, 3}
C. {1, 2, 3, 3}
D. (1, 2, 3)

Đáp án: B

Giải thích: Hàm set() chuyển danh sách [1, 2, 3] thành tập hợp {1, 2, 3}. Các phần tử trùng lặp sẽ bị loại bỏ (nếu có)

Câu 64: Kết quả của đoạn mã sau là gì?

```
x = {1, 2, 3}
print(list(x))
```

A. {1, 2, 3}
B. [1, 2, 3]
C. [1, 2, 2, 3]
D. (1, 2, 3)

Đáp án: B

Giải thích: Hàm list() chuyển tập hợp {1, 2, 3} thành danh sách [1, 2, 3].

Câu 65: Phép toán nào dưới đây sẽ chuyển một kiểu dữ liệu int thành kiểu boolean?

A. bool(int)
B. int(bool)
C. str(bool)
D. bool(str)

Đáp án: A

Giải thích: Hàm bool() chuyển đổi giá trị int thành kiểu boolean. Nếu giá trị là 0, kết quả là False, nếu khác 0 thì kết quả là True.

Câu 66: Kết quả của đoạn mã sau sẽ là gì?

```
x = "True"
print(bool(x))
```

A. True
B. False
C. 'True'
D. None

Đáp án: A

Giải thích: Hàm bool() trả về True khi chuỗi không rỗng, dù chuỗi đó có phải là "True" hay không.

Câu 67: Kết quả của đoạn mã sau là gì?

```
x = '123.45'
print(float(x))
```

A. 123
B. 123.45
C. '123.45'
D. 123.0

Đáp án: B

Giải thích: Hàm float() chuyển chuỗi '123.45' thành số thực 123.45

Câu 68: Kết quả của đoạn mã sau là gì?

```
x = 10
print(str(float(x)))
```

A. '10.0'
B. '10'
C. 10.0
D. 10

Đáp án: A

Giải thích: Đầu tiên, float(x) chuyển x thành số thực 10.0. Sau đó, str() chuyển 10.0 thành chuỗi '10.0'.

Câu 69: Kết quả của đoạn mã sau là gì?

```
lst = [1, 2, 3, 4, 5, 6, 7]
print(lst[2:5])
```

A. [1, 3, 5, 7]
B. [1, 2, 3, 4, 5, 6, 7]
C. [2, 4, 6]
D. [3,4,5]

Đáp án: A

Giải thích: lấy các phần tử từ phần tử có chỉ số 2 đến phần tử có chỉ số (5-1), do đó kết quả là [3, 4, 5].

Tương tự cho tuple, set

Câu 70: Kết quả của đoạn mã sau là gì?

```
lst = [1, 2, 3, 4, 5, 6, 7]
print(lst[::2])
```

A. [1, 3, 5, 7]
B. [1, 2, 3, 4, 5, 6, 7]
C. [2, 4, 6]
D. [7, 5, 3, 1]

Đáp án: A

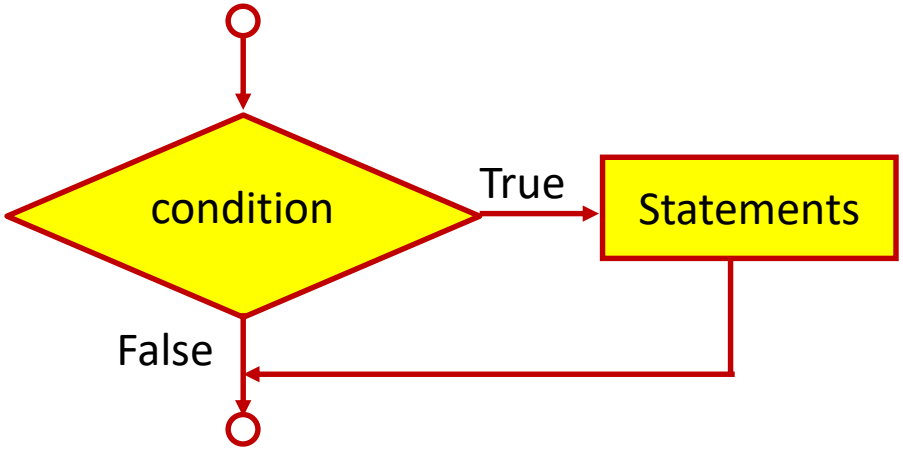
Giải thích: Slicing [::2] lấy các phần tử với bước nhảy 2, tức là phần tử đầu tiên, sau đó cứ 2 phần tử sẽ được lấy, do đó kết quả là [1, 3, 5, 7].

Tương tự cho tuple, set

7.Conditional Control Statements

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

7.Conditional Control Statements: if statement

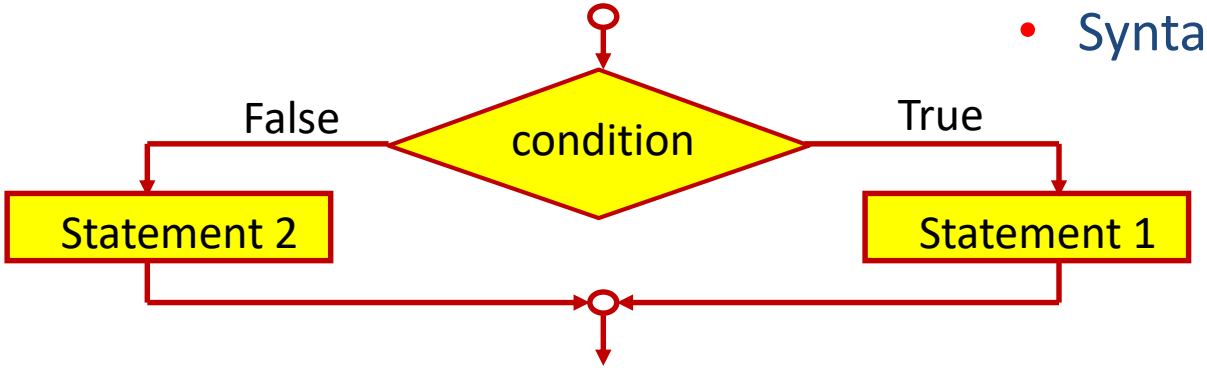


- Syntax

```
if (condition):
    Statements
```

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

7.Conditional Control Statements: if…else statement

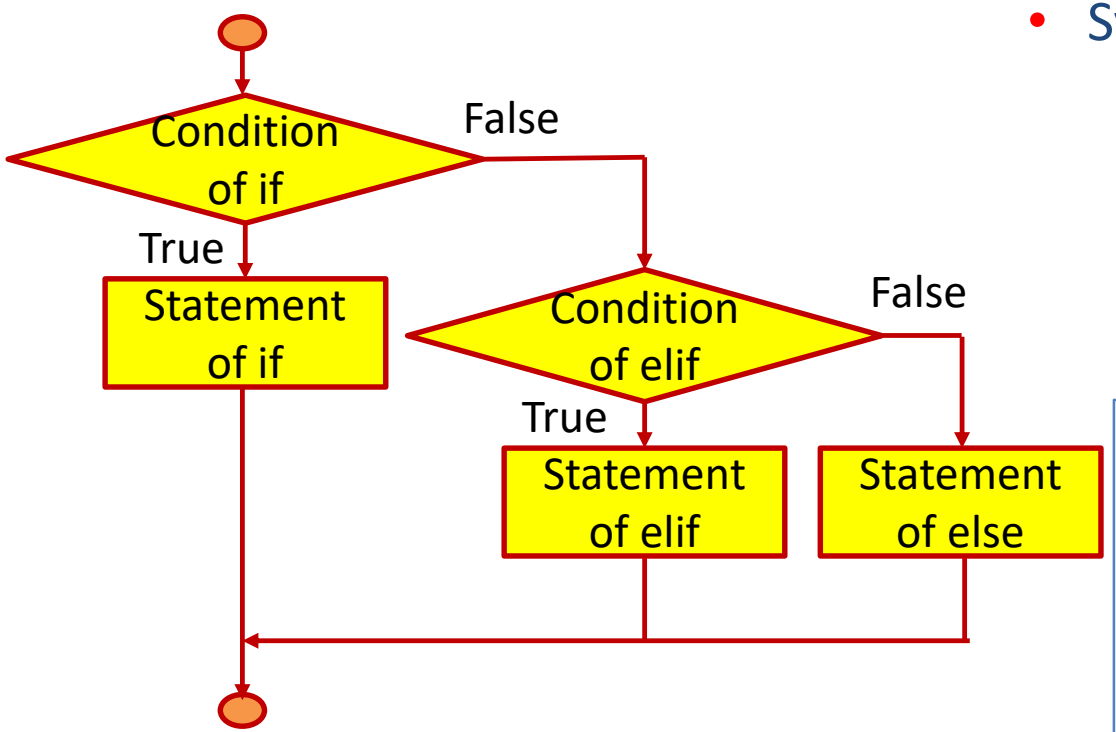


- Syntax

```
if (condition):
    Statement 1
else:
    Statement 2
```

```
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")
```

7.Conditional Control Statements: if... elif... else Statement



- Syntax

```
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")
```

7.Conditional Control Statements: Nested If Statement

- Syntax

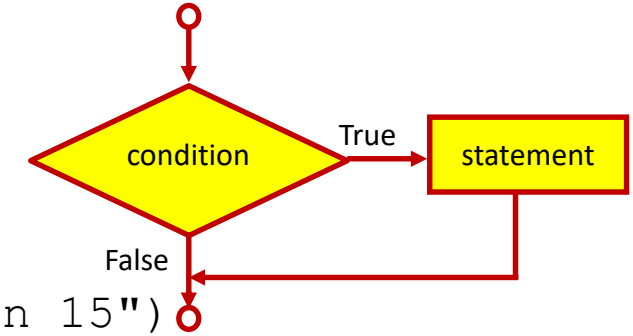
```
if (condition1):
    # Executes when condition1 is true
    if (condition2):
    # Executes when condition2 is true
  # if Block is end here
# if Block is end here
```

- Example

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

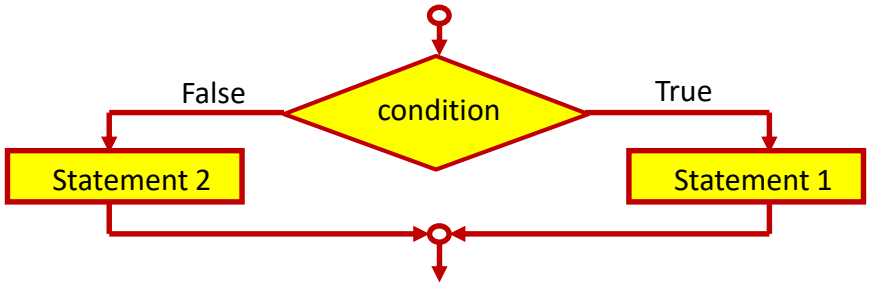
- If there is only one statement to execute, the if & if … else statements can be put on the same line

if condition: statement




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

Statement 1 **if** (condition) **else** statement 2



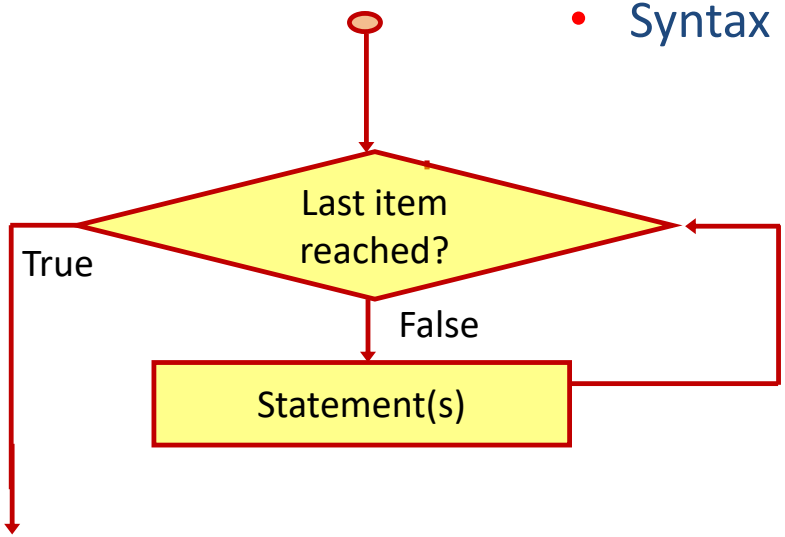

```
i = 10
print("A") if (i < 15) else print("B")
```

8. 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

8. Loop Control Statements: for Loop Statements

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



- Syntax

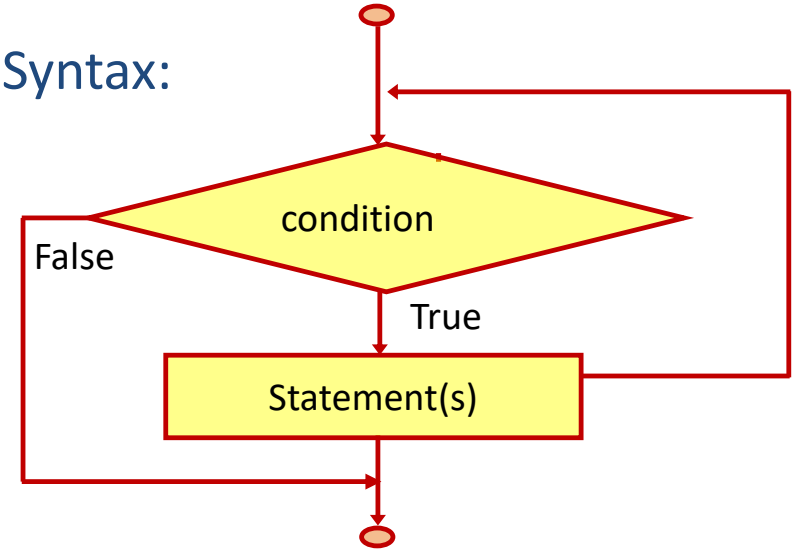
```
for variable_name in sequence :
    statement_1
    statement_2
        ....
```

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

8. Loop Control Statements: `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.

- Syntax:




```
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)
```

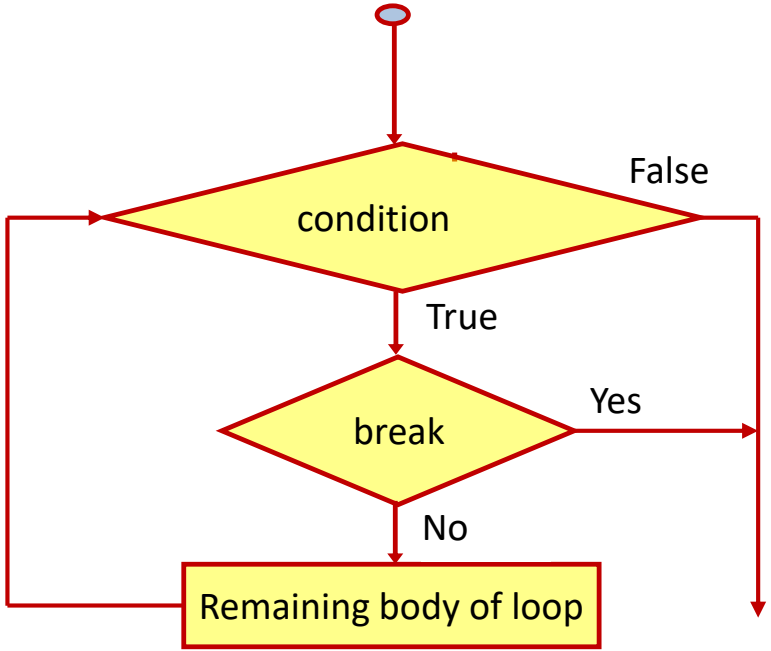
➔ 2 3 3 4 5

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

➔ 2 5 8 11 14 17
20 23 26 29

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)        ➔ 1  2  2  3  3
```

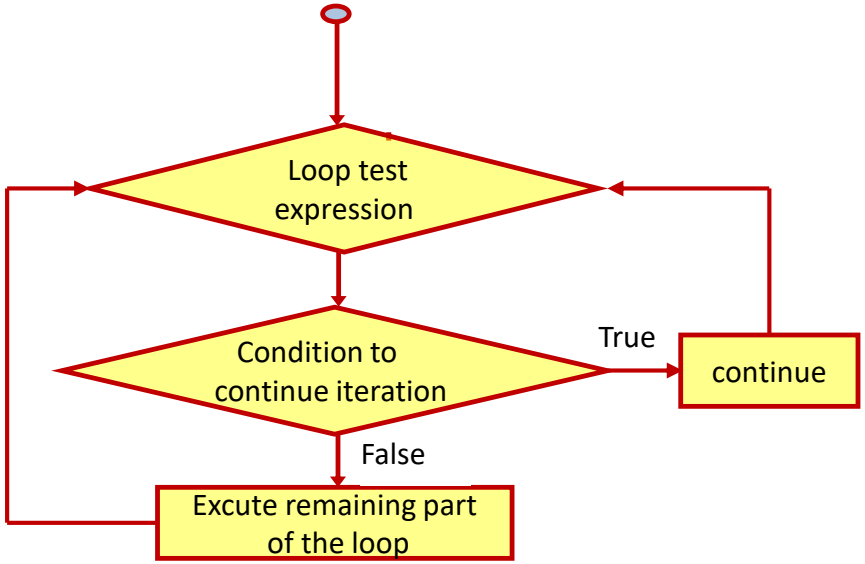
```
colors = ["blue", "green", "red"]
for x in colors:
  print(x)
  if x == "green":
    break
  print(x) ➔ blue  blue  green
```

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)➔ 1 2 3 5 6 7
```

```
for x in range(7):
    if (x==4):
        continue
    print(x)➔ 0 1 2 3 5 6
```



- Introduction to Function in python
- Types of functions
  - A) Function has no parameter and no return value
  - B) Function has no parameter and has a return value
  - C) Function has parameter and no return value
  - D) Function with many parameters
  - E) Function with variable number of parameters
  - F) Functions with parameter have default value
- Organize management and use of functions

- Functions in python have 2 types
  - Built – in function:
    - Functions created by Python developers
    - When needed, call the function to use
    - Example:
      - `len()`…
      - `float(), int(), …`
      - `input()`
      - `sorted(), min(), max(),..,`
  - Functions created by the user (called a user-defined function)

- Example: The sum of the list's elements

```
ds_le = [ 11, 13, 15, 17, 19, 21]
ds_chan = [ 10, 12, 14, 16, 18, 20]
sum1=0
for d1 in ds_le:
    sum1 = sum1 + d1
print("Sum of ds_le:", sum1)

sum2=0
for d2 in ds_chan:
    sum2 = sum2 + d2
print("Sum of ds_chan:", sum2)
```

```
Sum of ds_le: 96
Sum of ds_chan: 90
```

- Replace with the following Code

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

define function

```
ds1 = [ 11, 13, 15, 17, 19, 21]
ds2 = [ 10, 12, 14, 16, 18, 20]
sum1 = Tong_DS(ds1)
sum2 = Tong_DS(ds2)
print(" The sum of the list of odd numbers is: ", sum1)
print(" The sum of the list of even numbers is : ", sum2)
```

Call function

- Function in python
  - A function is a special block of code, it is named and can be called for use in different places in the program.
  - Meaning: A function is a reusable block of commands.

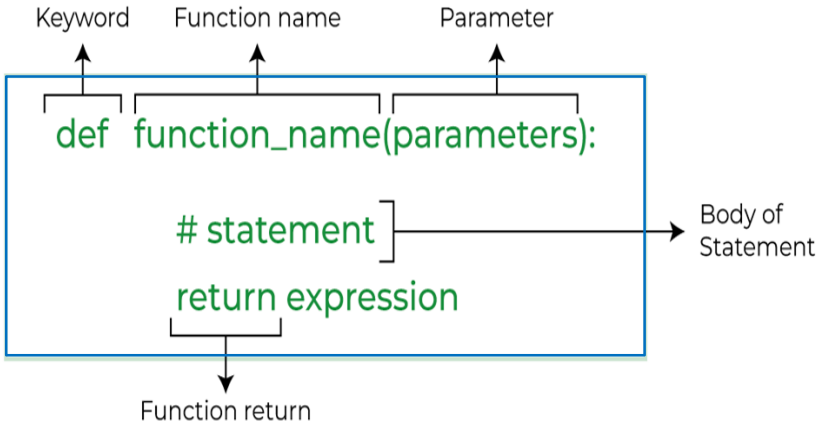
Input:
Parameters, possible or not

output:
- Return value
- Or done some task

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

Function: Tong_DS(ds)
Input: Paramete is ds
Output: return value is sum

- 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 Arguments of that particular function.
- Example:

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

- A) Function has no parameter and no return value
  - Define (create) a function:

```
def  Name_of_Function():
     " Short description of the function " # not required
     Body_of_Function
```

    - def: is a keyword.
    - Name_of_Function: The function's identifier is named by the programmer.
    - Body_of_Function : Contains Python statements
  - Call Function: Name_of_Function()

Define (create) a function

Call Function

- Example:

```
def greet_user():
    "This is a example about function"
    print("Hello!")
greet_user()
```

- B) Function has no parameter and has return value
  - Define (create) a function:

```
def  Name_of_Function():
     " Short description of the function "
     Body_of_Function
     return  value
```

  - Example:

Define (create) a function

```
def   greet_user():
    "This is a example about function"
    x = 5 + 6
    return x

x= greet_user()
```

Call Function

- C) Function has parameter and no return value
  - Define (create) a function:

```
def Name_of_Function(Parameter):
    " Short description of the function "
    Body_of_Function
```

  - Call function: Name_of_Function(Argument)
  - When calling a function, the value of the argument will be passed into the parameters of the called function
  - Example:

Define (create) a function

"Ten" is a parameter

```
def greet_user(Ten):
    print("Hello!" + Ten.title())
```

"name" is a argument

Call Function:

```
name="dũng"
greet_user(name)
```

- D) Function with many parameters
  - Define (create) a function:

```
def Name_of_Function(Parameter1, Parameter1…):
    " Short description of the function "  # not required
    Body_of_Function
```

  - Call function: Name Function(Argument1, Argument2…)
  - When calling the function, then: Value of argument1 will be passed into parameter1 , value of argument2 will be passed into parameter2…
  - Example:

Define (create) a function:

```
def sum_02_so(a, b):
    print("Sum of 02 number is:", a+b)
```

Call Function:

```
x1, x2 = 10, 20
sum_02_so(x1, x2)
```

- E) Functions with parameter have default value
  - Define (create) a function:

```
def Name_of_Function(Parameter1, Parameter2= default value,…):
    " Short description of the function "
    Body_of_Function
```

  - Call function: Name_of_Function(Argument1)
  - Value of argument1 will be passed into parameter1
  - Value of Parameter2 is default value
  - Example: Define (create) a function

```
def greet_user(name, age="22"):
    print("My name is" + name.title())
    print("My age is" + age)
```

  - Call fuction:

```
name="dũng"
greet_user(name)
```

My name is Dũng
My age is 22

- F) Function with special parameter
  - Example:
    - Define a function:

```
def sum_03_so(a, b, c):
    print("Sum 03 number is: ", a+b+c)
```

    - Call Function:

```
x1, x2, x3 = 10, 20, 30
sum_03_so(x1, x2, x3)
```

⇒ **60**

    - Replay with:

```
def sum_many_number(*a):
    for so in a:
        sum = sum + a
    print("Sum is: ",sum)
```

    - Call function:

```
x1, x2, x3 = 10, 20, 30
sum_many_number(x1, x2, x3)
```

⇒ **60**

```
x1, x2, x3, x4, x5 = 10, 20, 30, 40, 50
sum_many_number(x1, x2, x3, x4, x5)
```

⇒ **150**

- F) Function with special parameter
  - Define (create) a function:

```
def Name_of_Function(*Parameter):
    " Short description of the function "
    Body_of_Function
```

  - Call function: Name_of_Function(argument1, argument2…)
  - When calling the function, then: Value of argument1 will be passed into parameter , value of argument2 will be passed into parameter…
  - Use the structure **`for …in …`** to access all arguments

- Example of Function with special parameter :

```
def myFun(*a):
    for i in a:
        print(i)
myFun('Welcome', 'to', 'VKU')
```

```
Welcome
to
VKU
```

```
def myFun(**a):
    for key, value in a.items():
        print(key, value))
myFun(first='Welcome', second='to', last='VKU')
```

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

9. Function: Organize storage and use functions

- In large projects, for easy management:
- Method 1:
  - Step 1: Define (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 (Call function) according to the syntax :

    Module.FunctionName

- Step 1: Create sum.py

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

Define (create) a function

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

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

Call Function

- Method 2:
  - Step 1: Define (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 (call function) according to the syntax: FunctionName

- Step 1: Create **sum.py** file

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

Define (create) a function

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

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

Call Function

- Method 3: 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

- Step 1: Create a **sum.py** file

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

Define (create) a function

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

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

Call Function

Practice and exercises
Part 2

- 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:

- 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)
```

- **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.

- Using the **read()** method:

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

  - size <=0: return 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))
```

10. File Handling: Close 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.
- Syntax

```
File_object.close()
```

10. File Handling: Writing to file

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

```
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()
```

10. File Handling: 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()
```

- 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` … `except` Statement
- `try` … `except`… `else` … `finally` Statement

11. Exception Handling: try ... except Statement

- **Try** and **except** statements are used to catch and handle exceptions.
- Syntax:

```
try:
   statements in try - except block
except [exception error code]:
   executed when error in try - excep
   block
```

- Example:

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

```
filename = 'alice.txt'
try:
    with open filename as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "File " + filename + "File Not Found."
    print(msg)
```

11. Exception Handling: try … except… else … finally Statement

- 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 – except block ➔ Run this Code
except exception error code:
   Executed when error in try – except block ➔ Execute
   this code when there is an exception
else:
   Executed if no exception ➔ No exceptions, run this code
finally:
   Executed irrespective of exception occured or not ➔ Always
   run this code
```

11. Exception Handling: try … except… else … finally Statement

- Example:

```
try:
    print('try block')
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError: # exception error code
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")

print("Out of try, except, else and finally blocks.")
```

- Example

```
filename = 'alice.txt'
try:
    with open (filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "File " + filename + " File Not Found."
    print(msg)
else:
    words = contents.split()
    num_words = len(words)
    print("File " + filename + "have" + str(num_words) + " words.")
finally:
   print("finish!!!")
```

11. Exception Handling: Python Built-in Exceptions

Exception	Cause of Error
AssertionError	Raised when an assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() function hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or Delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.

11. Exception Handling: Python Built-in Exceptions

Exception	Cause of Error
OSError	Raised when system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.

11. Exception Handling: Python Built-in Exceptions

Exception	Cause of Error
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.

11. Exception Handling: User-defined exception

- Sometimes, we may need to create our own exceptions that serve our purpose.
- We can define own exceptions by creating a new class that is derived from the built-in **Exception** class
- Syntax to define custom exceptions:

```
class CustomError(Exception):
        ...
        pass
```

Define a "CustomError" user-defined error which inherits from the Exception class

```
try:
        raise CustomError
        ...
except CustomError:
        ...
```

Use a "CustomError" user-defined error to exception Handling

- Example: define user-defined exceptions

```
class InvalidAgeException(Exception):
      pass


number = 18
try:
    input_num = int(input("Enter a number: "))
    if input_num < number:
        raise InvalidAgeException
    else:
        print("Eligible to Vote")
except InvalidAgeException:
    print("Exception occurred: Invalid Age")
```

Define a user-defined error

exception Handling

Practice and exercises
Part 3

The end of Chapter