DLCV-01-Python_Basics

August 31, 2024

0.1 Python

Python is a general-purpose, high-level and interpreted programming language and dynamically typed semantics. It is also an object-oriented programming languge. To run a simple Python program, you can simply run the following command in your Python interpreter.

```
print('Hello, World!')
```

Bingo!!! Once you can run that simple program, you are ready to go!!!

Example

[ ]:
```
print('Hello, World!')
```

```
Hello, World!
```

0.1.1 Data Types

1. String
2. Integer
3. Float
4. Complex
5. Boolean
6. List
7. Set
8. Tuple
9. Dictionary
10. None

1. String A string can be a single character or a sequence of characters.

```
x = "a"
x = 'b'
y = "cat"
x = 'This is a sentence.'
x = """This is docstrings or multi-line comments"""
```

All the string values are assigned in `x` or `y` variables.

Examples

[ ]:
```
x = "a"
print("First x: ", x)

x = 'b'
print("Second x: ", x)

y = "cat"
print(y)

x = 'This is a sentence.'
print("Third x: ", x)

x = """This is docstrings or multi-line comments"""
print("Fourth x: ", x)
```

```
First x:  a
Second x:  b
cat
Third x:  This is a sentence.
Fourth x:  This is docstrings or multi-line comments
```

2. Integer An integer is a numeric data type that represents whole numbers. In other words, integers are numbers that have no fractional or decimal part. Integers can be positive, negative or zero.

```
x = 1
negative_integer = -10
y = 1009
zeroValue = 0
```

When you combine an integer with any string or when you put an integer into `"-"` quotes, the data type is always changed to `string` value.

`x = "24"` « this is a string
`friday13 = "Friday the 13."`
`x = 13, y = "Friday ", z = x + y` « this will output `Friday 13` which is also a string.
`concat_string = 'Friday' + " " + str(13)` « this is also a string

Examples

[ ]:
```
x = 1
print("First x: ", x)
print(type(x))

negative_integer = -10
y = 1009
zeroValue = 0

x = "24"  # this is a string
print("Second x: ", x)
```

```
print("Type of x before: ", type(x))
print("Type of x after: ", type(str(x)))

friday13 = "Friday the 13."
print("Type of friday13: ", type(friday13))

x = 13
y = "Friday "
z = str(x) + y # this will output Friday 13 which is also a string.
print(x)

concat_string = 'Friday' + " " + str(13) # this is also a string
print(type(str(13)))
print(type("13"))
```

```
First x:  1
<class 'int'>
Second x:  24
Type of x before:  <class 'str'>
Type of x after:  <class 'str'>
Type of friday13:  <class 'str'>
13
<class 'str'>
<class 'str'>
```

3. Float A float data type is also a numeric type that has fractional or decimal part. Unlike other programming languages, Python float data type represents any precision number.

PS. This is a different scenario when you use python library packages such as numpy or pytorch where we have p

```
x = 3.14
x = 3.1416
pi = 3.1415926535
npower = 1.7e-3 « this is equal to 1.7x10^-3 or 0.0017
ppower = 1.7e3 « this is equal to 1.7x10^3 or 1700
```

Examples

[ ]:
```
npower = 1.7e-3
print(npower)
```

```
0.0017
```

4. Complex A complex number is specified as real part and imginary part where the imaginary part is written with a `j` or `J`

```
x = 5j
y = 3 + 5j
```

Examples

[ ]:
```
y = 3 + 5j
print(y.imag)
print(y.real)
```

```
5.0
3.0
```

5. Boolean A boolean data type represents one of the two values from True and False.

```
x = True
y = False
```

Examples

[ ]:
```
x = True
y = False

print(y)
print(x)
```

```
False
True
```

6. List A list in Python includes zero or more elements between square brackets. The elements can be of different data types such as strings, integers, booleans, objects or even lists. A list has some useful properties: - A list is ordered - Elements in a list can be accessed by index - A list can have any type of object - A list is mutable

```
empty_list = []
x = ['a', 'cat', 12, 3.14]
listy = [['a', 'cat', 12, 3.14], ["This is a nested list", 1995, True]]
```

Examples

[ ]:
```
empty_list = []
print("Empty list: ", empty_list)

x = ['a', 'cat', 12, 3.14]
listy = [['a', 'cat', 12, 3.14], ["This is a nested list", 1995, '3.14',␣
 ↪'0952345676', True]]

print("Nested list: ", listy)
```

```
Empty list:  []
Nested list:  [['a', 'cat', 12, 3.14], ['This is a nested list', 1995, '3.14',
'0952345676', True]]
```

Indexing To extract a value from the list, you can access from the index number of the respective element.

`x = ['a', 'cat', 12, 3.14]`
`x[0]` « return 'a'
`x[1]` « return 'cat'
`x[2]` « return 12
`x[3]` « return 3.14

When you are indexing nested list, you can use indexes inside multiple square brackets where the earlier one represents the outer list while the later one(s) represent(s) inner list(s) sequentially.

You can also use the negative indexing in a list or nested list.

Examples

[ ]:
```
x = ['a', 'cat', 12, 3.14]
print("Index 0: ", x[0])
print("Index 1: ", x[1])
print("Index 2: ", x[2])
print("Index 3: ", x[3])
print("Reverse Index -3: ", x[-3])
```

```
Index 0:  a
Index 1:  cat
Index 2:  12
Index 3:  3.14
Reverse Index -3:  cat
```

Replacing values in a list You can replace any value in a list by accessing the index location and assign a new value. The replaced data type can be any type.

`x = ['a', 'cat', 12, 3.14]`
`x[0] = 'b'` « this replace the first element with a string value `'b'`
`x[0] = 1` « this replace the first element with an integer value `1`

Examples

[ ]:
```
x = ['a', 'cat', 12, 3.14]
x[0] = 'b'

print("Replaced value of x at index 0: ", x[0])
x[1] = 11
print("Replaced value of x at index 1: ", x[1])
print(x)
```

```
Replaced value of x at index 0:  b
Replaced value of x at index 1:  11
['b', 11, 12, 3.14]
```

Slicing a list `l = ['a', 'b', 'c', 'd', 'e', 'f']`
`l[1:3]` « returns ['b', 'c']
`l[:3]` « returns ['a', 'b', 'c']

`l[3:5]` « returns [‘d’, ‘e’]
`l[3:6]` « returns [‘d’, ‘e’, ‘f’]
`l[3:]` « returns [‘d’, ‘e’, ‘f’]
`l[::2]` « returns [‘a’, ‘c’, ‘e’] which is step=2 slicing

Examples

[ ]:
```
l = ['a', 'b', 'c', 'd', 'e', 'f']

print("Slice l[1:3]: ", l[1:3]) # returns ['b', 'c']
print("Slice l[:3]: ", l[:3]) # returns ['a', 'b', 'c']
print("Slice l[3:5]: ", l[3:5]) # returns ['d', 'e']
print("Slice l[3:6]: ", l[3:6]) # returns ['d', 'e', 'f']
print("Slice l[3:]: ", l[3:]) # returns ['d', 'e', 'f']
print("Slice l[::2]: ", l[::2]) # returns ['a', 'c', 'e'] which is step=2
 ↪slicing

print("Slice l[::2]: ", l[::2]) # l[starts:ends:steps]
```

```
Slice l[1:3]:  ['b', 'c']
Slice l[:3]:  ['a', 'b', 'c']
Slice l[3:5]:  ['d', 'e']
Slice l[3:6]:  ['d', 'e', 'f']
Slice l[3:]:  ['d', 'e', 'f']
Slice l[::2]:  ['a', 'c', 'e']
Slice l[::2]:  ['a', 'c', 'e']
```

Constructors and methods that can be applied in a list

- `len()`
- `append()`
- `insert()`
- `extend()`
- `pop()`
- `remove()`
- `sorted()`
- `sort()`
- `reverse()`
- `index()`
- `count()`
- `copy()`
- `sum()`
- `min()`, `max()`

Examples

[ ]:
```
l = ['a', 'b', 'c', 'd', 'e', 'f']
s = 'Tanawin Two'
print("Length of list l: ", len(l))
print("Length of string s: ", len(s))
```

```
l.append(s)
print("List after appending s string", l)

l.insert(1, s)
print("List after inserting s string at index 1", l)
```

```
Length of list l:  6
Length of string s:  11
List after appending s string ['a', 'b', 'c', 'd', 'e', 'f', 'Tanawin Two']
List after inserting s string at index 1 ['a', 'Tanawin Two', 'b', 'c', 'd',
'e', 'f', 'Tanawin Two']
```

7. Set A set in Python is similar to a list however they have some different properties. - A set is written in a curly bracket
- A set is unordered - Elements in a set are unique - A set is mutable

Examples

[ ]:
```
set_x = {1, 2, 3}
print("Integer set_x: ", set_x)

set_x = {"a", "b", "d", "c", 1, 3, 2, 4}
print("Unordered set_x: ", set_x)

set_x = {"a", "b", "a", "a", "c", 1, 1, 2, 3, 4}
print("Unique set_x: ", set_x)

print("########################")
print("Error while indexing set")
set_x[0]
```

```
Integer set_x:  {1, 2, 3}
Unordered set_x:  {1, 2, 3, 4, 'c', 'b', 'a', 'd'}
Unique set_x:  {1, 2, 3, 4, 'b', 'a', 'c'}
########################
Error while indexing set
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[13], line 12
     10 print("########################")
     11 print("Error while indexing set")
---> 12 set_x[0]

TypeError: 'set' object is not subscriptable
```

Constructors and methods that can be applied in a set

- union()
- update()
- intersection()
- difference()
- sum()

Examples

[ ]:
```
set_x = {1, 2, 3}
set_y = {1, 2, 4, 5}

print("Union of two sets option 1: ", set_x.union(set_y))
print("Union of two sets option 2: ", set_y.union(set_x))

print("Intersection of two sets option 1: ", set_x.intersection(set_y))
print("Intersection of two sets option 2: ", set_y.intersection(set_x))

print("Update a set returns: ", set_x.update(set_y))
print("Updated set_x", set_x)
```

```
Union of two sets option 1:  {1, 2, 3, 4, 5}
Union of two sets option 2:  {1, 2, 3, 4, 5}
Intersection of two sets option 1:  {1, 2}
Intersection of two sets option 2:  {1, 2}
Update a set returns:  None
Updated set_x {1, 2, 3, 4, 5}
```

8. Tuple A tuple is also similar to a list and ordered collection of values put inside parentheses or sometimes no parentheses at all. - A tuple is ordered - A tuple is assessible by index - A tuple is immutable

Examples

[ ]:
```
tuple_x = (1, 2, 3, 5, 4)
print("tuple_x: ", tuple_x)

print("Indexing a tuple", tuple_x[0])
print("Slicing a tuple", tuple_x[0:3])

print("################################")
print("Immutable error while editing a tuple: ")
tuple_x[0] = 100
```

```
tuple_x:  (1, 2, 3, 5, 4)
Indexing a tuple 1
Slicing a tuple (1, 2, 3)
################################
Immutable error while editing a tuple:
```

```
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
Cell In[15], line 9
      7 print("################################")
      8 print("Immutable error while editing a tuple: ")
----> 9 tuple_x[0] = 100

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

Constructors and methods that can be applied in a tuple

- sorted()
- sum()
- min()
- max()

An iterable can be converted to a list or set or tuple and vice versa.

`tuple(list)` « convert from list to tuple
`list(tuple)` « convert from tuple to list
`set(list)` « make a set from a list
`set(tuple)` « make a set from a tuple

Examples

[ ]:
```
list_x = ['a', 'b', 'c', 'e', 'd', 'd']
tuple_x = ('a', 'b', 'c', 'e', 'd', 'd')
set_x = {'a', 'b', 'c', 'e', 'd', 'd'}

print("From list to set", type(set(list_x)))
print("From tuple to set", type(set(tuple_x)))

print("From set to list", type(list(set_x)))
print("From tuple to list", type(list(tuple_x)))

print("From list to tuple", type(tuple(list_x)))
print("From set to tuple", type(tuple(set_x)))
```

```
From list to set <class 'set'>
From tuple to set <class 'set'>
From set to list <class 'list'>
From tuple to list <class 'list'>
From list to tuple <class 'tuple'>
From set to tuple <class 'tuple'>
```

9. Dictionary A dictionary in Python refers to key:value pairs or items. This type of data structure is generally called associative arrays, hashes or hashmaps.

The syntax of a dictionary consists of a comma-separated list of `key:value` pairs in curly braces `{}`. A single `key:value` pair is also called an item.

```
d = {"name": "David",       "age": 25,       "nationality": "English",
"mobile": "0912345678",       "is_student": True,       "courses":
["mathematics", "DLCV", "statistics"],       "started_year": 2023}
```

We can created a dictionary by different ways:
1. simply assigning the `key:value` pairs
2. `key:value` pairs as two-item lists in a tuple
3. `key:value` pairs as two-item tuples in a list
4. zipping keys/values lists
5. and so on

Important properties of a dictionary:
1. Keys are unique
2. Keys are immutable type
3. Value can be of any type

Examples

[ ]:
```
d = {"name": "David",
      "age": 25,
      "nationality": "English",
      "mobile": "0912345678",
      "is_student": True,
      "courses": ["mathematics", "DLCV", "statistics"],
      "started_year": 2023}

print("Dictionary d: ", d)

tuple_1 = ("key_1", "value_1")
tuple_2 = ("key_2", "value_2")
tuple_3 = ("key_3", "value_3")
tuple_4 = ("key_4", "value_4")
dict_1 = dict([tuple_1, tuple_2, tuple_3, tuple_4])
print("Dictionary from two-item tuples list", dict_1)

list_1 = ["key_1", "value_1"]
list_2 = ["key_2", "value_2"]
list_3 = ["key_3", "value_3"]
list_4 = ["key_4", "value_4"]
dict_2 = dict([list_1, list_2, list_3, list_4])
print("Dictionary from two-item lists tuple", dict_2)
```

```
Dictionary d:  {'name': 'David', 'age': 25, 'nationality': 'English', 'mobile':
'0912345678', 'is_student': True, 'courses': ['mathematics', 'DLCV',
'statistics'], 'started_year': 2023}
Dictionary from two-item tuples list {'key_1': 'value_1', 'key_2': 'value_2',
'key_3': 'value_3', 'key_4': 'value_4'}
Dictionary from two-item lists tuple {'key_1': 'value_1', 'key_2': 'value_2',
'key_3': 'value_3', 'key_4': 'value_4'}
```

Accessing the dictionary `d = {"name": "David", "age": 25, "nationality": "English", "mobile": "0912345678", "is_student": True, "courses": ["mathematics", "DLCV", "statistics"], "started_year": 2023}`

`d['name']` « returns 'David'
`d['age']` « returns 25
`d['mobile']` « returns 0912345678
`d['program']` « returns KeyError

`d.get('courses')` « returns ["mathematics", "DLCV", "statistics"]
`d.get('program')` « returns None

Examples

[ ]:
```
d = {"name": "David",
      "age": 25,
      "nationality": "English",
      "mobile": "0912345678",
      "is_student": True,
      "courses": ["mathematics", "DLCV", "statistics"],
      "started_year": 2023}
print("Dictionary d: ", d)

print(d.get('courses')) # returns ["mathematics", "DLCV", "statistics"]
print(d.get('program')) # returns None

print(d['name']) # returns 'David'
print(d['age']) # returns 25
print(d['mobile']) # returns 0912345678

print("################################")
print("Error no key name program")
print(d['program']) # returns KeyError
```

```
Dictionary d:  {'name': 'David', 'age': 25, 'nationality': 'English', 'mobile':
'0912345678', 'is_student': True, 'courses': ['mathematics', 'DLCV',
'statistics'], 'started_year': 2023}
['mathematics', 'DLCV', 'statistics']
None
David
25
0912345678
################################
Error no key name program
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[18], line 19
     17 print("################################")
     18 print("Error no key name program")
```

```
---> 19 print(d['program']) # returns KeyError

KeyError: 'program'
```

Manipulating the dictionary `d = {"name": "David", "age": 25, "nationality": "English", "mobile": "0912345678", "is_student": True, "courses": ["mathematics", "DLCV", "statistics"], "started_year": 2023}`

1. Update a value
   `d['name'] = 'John'`
2. Add a key:value pair
   `d['program'] = 'Mechatronic'`
3. Remove an item from the dictionary
   `d.pop('is_student')` « this method will return a removed value
   `removed = d.pop('is_student')` and `print(removed)` « this printing will return `True`
   `del d['is_student']` « this will removed the item `'is_student': True`
4. Clear all items in the dictionary
   `d.clear()`
5. Get the keys or values from the dictionary
   `d.keys()` « this will return all keys in the dictionary
   `d.values` « this will return all values in the dictionary

Examples

[ ]:
```
d = {"name": "David",
      "age": 25,
      "nationality": "English",
      "mobile": "0912345678",
      "is_student": True,
      "courses": ["mathematics", "DLCV", "statistics"],
      "started_year": 2023}

print("name before updating: ", d['name'])
d['name'] = 'John'
print("name after updating: ", d['name'])

print("############################")
print("Added a new key 'program':")
d['program'] = "Mechatronic"
print("Call the key 'program': ", d['program'])

print("########################################")
print("pop removes a key:value pair and return the value")
d.pop('is_student') # this method will return a removed value

d = {"name": "David",
```

```
      "age": 25,
      "nationality": "English",
      "mobile": "0912345678",
      "is_student": True,
      "courses": ["mathematics", "DLCV", "statistics"],
      "started_year": 2023}

removed = d.pop('is_student')
print("The value returned from pop()", removed) # this printing will return True
print("Dictionary after removed 'is_student': ", d)

del d['started_year'] # this will removed the item 'started_year': 2023
print("Dictionary after removing 'started_year': ", d)
```

```
name before updating:  David
name after updating:  John
############################
Added a new key 'program':
Call the key 'program':  Mechatronic
########################################
pop removes a key:value pair and return the value
The value returned from pop() True
Dictionary after removed 'is_student':  {'name': 'David', 'age': 25,
'nationality': 'English', 'mobile': '0912345678', 'courses': ['mathematics',
'DLCV', 'statistics'], 'started_year': 2023}
Dictionary after removing 'started_year':  {'name': 'David', 'age': 25,
'nationality': 'English', 'mobile': '0912345678', 'courses': ['mathematics',
'DLCV', 'statistics']}
```

0.1.2 Operators

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators

Examples

[ ]:
```
print("########################################")
# Arithmetic Operators
print("Addition: ", 1 + 1)
print("Subtraction: ", 1 - 1)
print("Multiplication: ", 1 * 1)
print("Division: ",4 / 2)
print("Modulus: ", 5 % 2)
print("Exponentiation: ", 2**3)
print("Floor Division: ", 5//3)

print("########################################")
```

```
# Assignment Operators
x = 1 # Assignment
print("Before additional assignment: x =", x)
x+= 1
print("After additional assignment: x =", x)

print("########################################")
# Comparison Operators
x = 1
y = 2
print("Compare two variables whether equal or not which return boolean: ",␣
 ↪x==y) # != represents not equal to
```

```
########################################
Addition:  2
Subtraction:  0
Multiplication:  1
Division:  2.0
Modulus:  1
Exponentiation:  8
Floor Division:  1
########################################
Before additional assignment: x = 1
After additional assignment: x = 2
########################################
Compare two variables whether equal or not which return boolean:  False
```

[ ]:
```
x = 1
y = 0

print(x==0)
print(x==1)

print((x==1) and (y==1))
print(((x==1) and (y==1)))
# !
# ~
```

```
False
True
False
False
```

0.1.3 Conditional Statement

The conditional statement is a very useful for setting up the rules for specific action mapped with it's respective condition. The statement can be single condition or multiple conditions. Similarly, the action can be single or multiple.
Notice that the indentation in Python is important. You will get an error if the indentation is

wrong.

1. **if statement**

   if `condition`: action

2. **elif statement**

   if `condition 1`: action 1 elif `condition 2`: action 2 elif `condition 3`: action 3

3. **else statement**

   if `only one student`: cancel class elif `two students`: wait else : continue class

Examples Let's set some rules as an example: 1. If the weather is sunny and hot, we will bring umbrellas+water and go out for dinner. 2. If the weather is cold, we will bring coats and go out for dinner. 3. If the weather is rain or other than above two conditions, we will not go out and dinner at home.

[ ]:
```
# First define variables
weather_list = ['sunny and hot', 'cold', 'rain']
bring_items = ['umbrellas and water', 'coats']
go_out_status = [True, False]

weather = 'rain' ### Modify any of the element from weather_list

if weather == weather_list[0]:
    bring = bring_items[0]
    print(f"Today is {weather}, and we are bringing {bring}.")
    print("Go out: ", go_out_status[0])
elif weather == weather_list[1]:
    bring = bring_items[1]
    print(f"Today is {weather}, and we are bringing {bring}.")
    print("Go out: ", go_out_status[0])
elif weather == weather_list[2]:
    bring = None
    print(f"Today is {weather}, and we are bringing {bring}.")
    print("Go out: ", go_out_status[1])
```

```
Today is rain, and we are bringing None.
Go out:  False
```

0.1.4 The loops

The loops are used in iterating over the items from an iterable or a collection (list, tuple, dictionary, set or string).

1. `while` loop

   while `boolean`: action
   optional: stopping criteria

2. `for` loop for item in iterable: action

Examples Let's create some scenarios: While a person is alive, his age increase 1 year every year. However, when he died at an age, the age increasing process stops. Thus, one while loop is assumed to be 1 year.

[ ]:
```
### Define variables:
alive = True
age = 12
age_of_death = 70

while alive: ### infinit loop
    age+=1

    if age == 36:
        continue
    print('The age of a person is {}'.format(age))

    if age == age_of_death:
        alive=False
```

```
The age of a person is 13
The age of a person is 14
The age of a person is 15
The age of a person is 16
The age of a person is 17
The age of a person is 18
The age of a person is 19
The age of a person is 20
The age of a person is 21
The age of a person is 22
The age of a person is 23
The age of a person is 24
The age of a person is 25
The age of a person is 26
The age of a person is 27
The age of a person is 28
The age of a person is 29
The age of a person is 30
The age of a person is 31
The age of a person is 32
The age of a person is 33
The age of a person is 34
The age of a person is 35
The age of a person is 37
The age of a person is 38
The age of a person is 39
The age of a person is 40
The age of a person is 41
The age of a person is 42
```

```
The age of a person is 43
The age of a person is 44
The age of a person is 45
The age of a person is 46
The age of a person is 47
The age of a person is 48
The age of a person is 49
The age of a person is 50
The age of a person is 51
The age of a person is 52
The age of a person is 53
The age of a person is 54
The age of a person is 55
The age of a person is 56
The age of a person is 57
The age of a person is 58
The age of a person is 59
The age of a person is 60
The age of a person is 61
The age of a person is 62
The age of a person is 63
The age of a person is 64
The age of a person is 65
The age of a person is 66
The age of a person is 67
The age of a person is 68
The age of a person is 69
The age of a person is 70
```

break `break` will prematurely terminates the loop.

`for item in iterable:  break`

continue `continue` will stop executing the current loop and advance into next iteration. The remaining lines of codes after `continue` will not execute for that loop.

`for item in iterable:  action 1  continue  action 2` « this action for current loop will be skipped

pass `pass` will do nothing and it is just acting as a placeholder.

[ ]:
```
def passing():
    pass
```

Examples Let's use for loop to count 5 and each count again counts a-e strings.

[ ]:
```
string_list = ['a', 'b', 'c', 'd', 'e']
for i in range(5):
```

```
    print("Number Count", i)
    for j in string_list:
        print("At Number Count {} String Value is {}".format(i, j))
```

```
Number Count 0
At Number Count 0 String Value is a
At Number Count 0 String Value is b
At Number Count 0 String Value is c
At Number Count 0 String Value is d
At Number Count 0 String Value is e
Number Count 1
At Number Count 1 String Value is a
At Number Count 1 String Value is b
At Number Count 1 String Value is c
At Number Count 1 String Value is d
At Number Count 1 String Value is e
Number Count 2
At Number Count 2 String Value is a
At Number Count 2 String Value is b
At Number Count 2 String Value is c
At Number Count 2 String Value is d
At Number Count 2 String Value is e
Number Count 3
At Number Count 3 String Value is a
At Number Count 3 String Value is b
At Number Count 3 String Value is c
At Number Count 3 String Value is d
At Number Count 3 String Value is e
Number Count 4
At Number Count 4 String Value is a
At Number Count 4 String Value is b
At Number Count 4 String Value is c
At Number Count 4 String Value is d
At Number Count 4 String Value is e
```

[ ]:
```
### pass, break, continue
string_list = ['a', 'b', 'c', 'd', 'e']
for i in range(5):
    print("######################")
    print("Number Count", i)
    if i % 2 == 0:
        continue
    for j in string_list:
        print("At Number Count {} String Value is {}".format(i, j))
        if j == 'c':
            break
    if i == 3:
```

```
        break

print("############################")
### Do nothing for pass statement
for i in range(5):
    pass
```

```
#####################
Number Count 0
#####################
Number Count 1
At Number Count 1 String Value is a
At Number Count 1 String Value is b
At Number Count 1 String Value is c
#####################
Number Count 2
#####################
Number Count 3
At Number Count 3 String Value is a
At Number Count 3 String Value is b
At Number Count 3 String Value is c
############################
```

List Comprehension

[ ]:
```
myEvenList = [0, 2, 4, 6, 8, 10, 12]

myOddList = [(number-1) for number in myEvenList if (number-1) > 0]
# myOddList.pop(0)
# abs(myOddList)
print("myEvenList: ", myEvenList)
print("myOddList: ", myOddList)
```

```
myEvenList:  [0, 2, 4, 6, 8, 10, 12]
myOddList:  [1, 3, 5, 7, 9, 11]
```

0.1.5 Function

A function is a block of statement that is reusable in a program. Normally, the codes inside a function have specific tasks to perform and return appropriate output(s).

Built-in functions

1. `print()`
2. `len()`
3. `type()`

[ ]:
```
def myfunction(x, y, z, a, b):
    pass
```

```
def myFunction():
    pass

def my_function():
    pass
```

Examples Notice: all the functions and loops consist of indentation. Python is indentation sensative and it will pop up errors if you use wrong indentation.

let's start with simple function that does some operation and return the result.

x = 1
y = 2

get the sum, min and max values and return from a function

[ ]:
```
a = 1 # Can change any value
b = 2

def operator(x, y):

    sum_results = x+y
    min_results = min(x, y)
    max_results = max(x, y) # min() and max() functions are built-in functions␣
 ↪of Python

    return sum_results, min_results, max_results # The results of operator()␣
 ↪function returns a tuple

sum_results, min_results, max_results = operator(y=b, x=a) # We can extract the␣
 ↪tuple result by assigning into variables
print('sum_results: ', sum_results)
print('min_results: ', min_results)
print('max_results: ', max_results)
```

```
sum_results:  3
min_results:  1
max_results:  2
```

Python Keywords Python has a list of built-in keywords which are specific and ready to use in advance. You should avoid assigning these keywords as your variable names. You can look at the keywords by simply using the following function call `help('keywords')`.

[ ]:
```
help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               break               for                 not
```

```
None                __peg_parser__      and                 as
assert              async               await               class
continue            def                 del                 elif
else                except              finally             from
global              if                  import              in
is                  lambda              nonlocal            or
pass                raise               return              try
while               with                yield
```

Variable Scope Python has three different variable scopes: - Local - Global - Enclosing

Local Scope A variable inside a function possesses a local scope property or called local variable. It can only be accessed within the function that defines.

```
def local_scope():     x = 11     print(x)
```

Example

[ ]:
```
def local_scope():
    local_x = 11
    print(local_x)

local_scope()
print(local_x)
```

```
11
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[31], line 6
      3     print(local_x)
      5 local_scope()
----> 6 print(local_x)

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

Global Scope A variable that is declared outside all functions possesses a global scope or global variable. It can be accessed from anywhere within or outside of the function.

```
global_x = 12 def function_x():     print(global_x)
```

Example

[ ]:
```
global_x = 12
def function_x():
    print(global_x)

function_x()
```

```
print(global_x)
```

```
12
12
```

Let's look at the differences between following functions and variables. How will you comment them.

[ ]:
```
global_var = 27

def global_function():
#     global global_var
    global_var = 72
    print(global_var)

print(global_var)
global_function()
print(global_var)
```

```
27
72
27
```

[ ]:
```
global_var2 = 27

def global_function2():
    global global_var2
    global_var2 = 72
    print(global_var2)

print(global_var2)
global_function2()
print(global_var2)
```

```
27
72
72
```

[ ]:
```
global_var = 27

def global_function():
#     global global_var
    print("Before modification: ", global_var)
    global_var = 72
    print(global_var)

print(global_var)
global_function()
print(global_var)
```

27

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[35], line 10
      7     print(global_var)
      9 print(global_var)
---> 10 global_function()
     11 print(global_var)

Cell In[35], line 5, in global_function()
      3 def global_function():
      4 #     global global_var
----> 5     print("Before modification: ", global_var)
      6     global_var = 72
      7     print(global_var)

UnboundLocalError: local variable 'global_var' referenced before assignment
```

[ ]:
```
global_var4 = 30

def global_function3():
    global_var4 = global_var4 + 1
    print(global_var4)

global_function3()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[36], line 7
      4     global_var4 = global_var4 + 1
      5     print(global_var4)
----> 7 global_function3()

Cell In[36], line 4, in global_function3()
      3 def global_function3():
----> 4     global_var4 = global_var4 + 1
      5     print(global_var4)

UnboundLocalError: local variable 'global_var4' referenced before assignment
```

[ ]:
```
global_var4 = 30

def global_function3():
    global global_var4
```

```
    global_var4 = global_var4 + 1
    print(global_var4)

global_function3()
```

```
31
```

Lambda Function Lambda function does not use `def` keyword to define a function but simply uses `lambda` keyword. Here is a function how `lambda` defines.

[ ]:
```
### Normal function
def square(x):
    return x**2

print(square(2))

### Lambda function
square2 = lambda x: x**2 # lambda function is defined as >> lambda variable:␣
 ↪expression/operation
print(square2(2))
```

```
4
4
```

0.1.6 Class and Object

Python is an object-oriented programming language. Everything in Python is an object.

In Python, a class is a blueprint for creating objects (instances) that share common attributes (data) and behaviors (methods). A class defines a new data type, allowing you to create multiple instances of that type with consistent properties and behaviors.

A class contains attributes (variables) and methods (functions) that define the behavior of objects created from the class. The attributes store data related to the class, while the methods define the actions or operations that can be performed on the objects.

[ ]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} is barking!")

# Creating instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Accessing attributes and calling methods
```

```
print(f"{dog1.name} is {dog1.age} years old.")
dog1.bark()

print(f"{dog2.name} is {dog2.age} years old.")
dog2.bark()
```

```
Buddy is 3 years old.
Buddy is barking!
Max is 5 years old.
Max is barking!
```

[ ]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating instances of the Person class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

print(person1.name, person1.age)  # Output: Alice 30
print(person2.name, person2.age)  # Output: Bob 25
```

```
Alice 30
Bob 25
```

[ ]:
```
class InitCall:
    def __init__(self): ## dunder function or magic function
        print("This is init call.")

#     def yourName(self, name):
#         return name

initCall = InitCall()
```

```
This is init call.
```

Inheritance An inheritance is the ability of a class carrying all the properties of an old or existing class. The inheriting class is known as a child class, and the class being inherited is known as parent or base class.

Example

[ ]:
```
# base class
class Vehicle():
    def description(self):
        print("I'm a Vehicle!")

# subclass
```

```
class Car(Vehicle):
    pass

# create an object from each class
v = Vehicle()
c = Car()

v.description()
# Prints I'm a Vehicle!
c.description()
# Prints I'm a Vehicle!
```

```
I'm a Vehicle!
I'm a Vehicle!
```

We can overide a method from the parent class as well as we can add more methods inside the child class.

Example

[ ]:
```
### Overiding
# base class
class Vehicle():
    def description(self):
        print("I'm a Vehicle!")

# subclass
class Car(Vehicle):
    def description(self):
        print("I'm a Car!")

# create an object from each class
v = Vehicle()
c = Car()

v.description()
# Prints I'm a Vehicle!
c.description()
# Prints I'm a Car!
```

```
I'm a Vehicle!
I'm a Car!
```

[ ]:
```
# a parent class
class Vehicle():
    def description(self):
        print("I'm a", self.color, "Vehicle")

# subclass
```

```
class Car(Vehicle):
    def description(self):
        print("I'm a", self.color, self.style)
    def setSpeed(self, speed):
        print("Now traveling at", speed,"miles per hour")

# create an object from each class
v = Vehicle()
c = Car()

c.setSpeed(25)
# Prints Now traveling at 25 miles per hour
v.setSpeed(25)
# Triggers AttributeError: 'Vehicle' object has no attribute 'setSpeed'
```

```
Now traveling at 25 miles per hour
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[44], line 19
     17 c.setSpeed(25)
     18 # Prints Now traveling at 25 miles per hour
---> 19 v.setSpeed(25)
     20 # Triggers AttributeError: 'Vehicle' object has no attribute 'setSpeed'

AttributeError: 'Vehicle' object has no attribute 'setSpeed'
```

`super()` function is used to add more parameter(s) inside the `__init__()` function of the child class. Those parameter(s) are not present inside the `__init__()` of the parent class.

[ ]:
```
# base class
class Vehicle():
    def __init__(self, color):
        self.color = color
    def description(self):
        print("I'm a", self.color, "Vehicle")

# subclass
class Car(Vehicle):
    def __init__(self, color, style):
        super().__init__(color)    # invoke Vehicle's __init__() method
        self.style = style
    def description(self):
        print("I'm a", self.color, self.style)

# create an object from each class
v = Vehicle('Red')
```

```
c = Car('Black', 'SUV')

v.description()
# Prints I'm a Red Vehicle
c.description()
# Prints I'm a Black SUV
```

```
I'm a Red Vehicle
I'm a Black SUV
```

0.1.7 File Handling

Read/Write .txt files

[ ]:
```
f = open('data/sample.txt')
print(f.read())
```

```
This is the lab session of deep learning for computer vision course. This course
is offering on Mondays and Fridays.
We are now having a lab session for Python basics.
```

[ ]:
```
with open('data/sample.txt', 'r') as f:
    data = f.read()
f.close()

print(data)
```

```
This is the lab session of deep learning for computer vision course. This course
is offering on Mondays and Fridays.
We are now having a lab session for Python basics.
```

[ ]:
```
text = 'I am happy with Python.'
with open('data/sample2.txt', 'a') as f: # if you want to append, you can use␣
 ↪'a'
    f.write('\n'+text) # you can skip with '\n'
f.close()
```

0.1.8 Other useful Python operations you can explore

1. Regular Expression
2. Decorators
3. @Property
4. Exception Handling
5. Read/write .csv and other binary files
   and many more

0.1.9 Assignment

Exercise 1 We have Aug25 to Aug31 and each day has the following properties:

1. date number (25 to 31)
2. Monday to Sunday as days (strings)
3. is_weekend (boolean)
4. have_class (boolean)

if the day is Monday and Friday, we have class.

Define the seven variables in the dictionary format and print Aug25's properties using keys.

Output example:
`Date = August 25, Day = Friday, Weekend = False, Have class = True.`

[ ]:
```
d = {
    "dates": [num for num in range(25, 32)],
    "days": ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",␣
 ↪"Saturday", "Sunday"],
    "is_weekend": [0, 0, 0, 0, 0, 1, 1],
    "have_class": [1, 0, 0, 0, 1, 0, 0]
}
d
```

[ ]:
```
{'dates': [25, 26, 27, 28, 29, 30, 31],
 'days': ['Monday',
  'Tuesday',
  'Wednesday',
  'Thursday',
  'Friday',
  'Saturday',
  'Sunday'],
 'is_weekend': [0, 0, 0, 0, 0, 1, 1],
 'have_class': [1, 0, 0, 0, 1, 0, 0]}
```

Exercise 2 Print out all the days (Monday to Sunday) with it's four statuses using `for loop` and `if statement`.

hints: you need to make all the seven dictionaries iterable in order to loop.

output example:
`August 25, Weekday, Have class.`
`August 26, Weekend, No class.` and so on.

[ ]:
```
for i in range(7):
    s = f"August {d['dates'][i]}, {d['days'][i]}, {'Weekday' if␣
 ↪d['is_weekend'][i] else 'Weekend'}, {'Have class' if d['have_class'][i] else␣
 ↪'No class'}."
    print(s)
```

```
August 25, Monday, Weekend, Have class.
August 26, Tuesday, Weekend, No class.
August 27, Wednesday, Weekend, No class.
August 28, Thursday, Weekend, No class.
```

```
August 29, Friday, Weekend, Have class.
August 30, Saturday, Weekday, No class.
August 31, Sunday, Weekday, No class.
```

Exercise 3 Write a function that receives a dictionary variable (assume it's today) you defined in exercise 1 and returns a statement:

Output example:
`Today is {Friday}. It is a {weekday} and we {have} class.`

run the function two times with arguments Aug26 and Aug28 and get the output for each.

[ ]:
```python
def printToday(d, i):
    day = d.get("days")[i]
    weekday = "weekday" if d.get("is_weekend")[i] else "weekend"
    have = "have" if d.get("have_class")[i] else "have no"
    return f"Today is {day}. It is a {weekday}, and we {have} class."

print(printToday(d, 1))
print(printToday(d, 3))
```

```
Today is Tuesday. It is a weekend, and we have no class.
Today is Thursday. It is a weekend, and we have no class.
```

Exercise 4 Creat a new class called `MyDay`, inherit the `Day` class as a parent class and add the following.
1. By using `super()` function, the `__init__()` receive additional argument named `date` 2. Write a function(any name) that prints the same output as `Exercise 3` by using the `day` and `date` parameters from the class you define. Notice that you have to filter a condition whether it has `have_class` or not on the day you pass. 3. Finally, you create an instance and print the output that you want.

[ ]:
```python
### Please do not modify the given class
class Day:
    def __init__(self, day): # 'day' can be any day between "Monday" to "Sunday"
        self.day = day
        self.dayOfWeek = self.whichDay(self.day)

    def whichDay(self, day):
        if self.day == 'Saturday' or self.day == 'Sunday':
            return 'weekend'
        else:
            return 'weekday'
```

[ ]:
```python
class MyDay(Day):
    def __init__(self, day):
        super().__init__(day)

    def printToday(self):
```

```
        class_status = "have" if self.day in ["Monday", "Friday"] else "have no"
        return f"Today is {self.day}. It is a {self.dayOfWeek}, and we␣
 ↪{class_status} class."


for day in ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",␣
 ↪"Sunday"]:
    print(MyDay(day).printToday())
```

```
Today is Monday. It is a weekday, and we have class.
Today is Tuesday. It is a weekday, and we have no class.
Today is Wednesday. It is a weekday, and we have no class.
Today is Thursday. It is a weekday, and we have no class.
Today is Friday. It is a weekday, and we have class.
Today is Saturday. It is a weekend, and we have no class.
Today is Sunday. It is a weekend, and we have no class.
```

Exercise 5 By using the given `text` docstring variable, extract every word without period and write them into a .txt file with tab spacings. Then re-read the file into a variable called `my_list` and print out the first and second index names 'Lorem' and 'Ipsum'.

[ ]:
```
text = """Lorem Ipsum is simply dummy text of the printing and typesetting␣
 ↪industry. Lorem Ipsum has been the industry's standard dummy text ever since␣
 ↪the 1500s, when an unknown printer took a galley of type and scrambled it to␣
 ↪make a type specimen book. It has survived not only five centuries, but also␣
 ↪the leap into electronic typesetting, remaining essentially unchanged. It␣
 ↪was popularised in the 1960s with the release of Letraset sheets containing␣
 ↪Lorem Ipsum passages, and more recently with desktop publishing software␣
 ↪like Aldus PageMaker including versions of Lorem Ipsum."""
```

[ ]:
```
words = text.replace('.', '').split()

with open('output.txt', 'w') as f:
    f.writelines('\n'.join(words))

with open('output.txt', 'rt') as f:
    txt = f.read()
    data = txt.split('\n')

print(data[0])
print(data[1])
```

```
Lorem
Ipsum
```