

## A python interpreter :

- ❖ is a program that converts the code written in python language by the user to the language which computer hardware or system can understand

### Function vs Method:

- ◆ Function is doing something General like ... `Len()`
- ◆ Method is doing something related to specific thing like ... methods manipulating strings

We can see methods related to string by writing (name of string then dot ).

```
name1 = 'mohamed'
name1.
```

The screenshot shows a code editor with the line `name1 = 'mohamed'`. Below it, the string `name1` is followed by a dot, which triggers a dropdown menu of methods. The methods listed are: capitalize, casefold, center, count, encode, endswith, expandtabs, find, format, format\_map, index, and isalnum. The method `capitalize` is highlighted with a blue background.

These Methods doesn't affect our string , it does create another one and manipulate it

```
name1 = 'mohamed'
print(name1.capitalize())
print(name1)
```

Mohamed  
mohamed

### Find() :

- search for the index of the a character in a string and return the first occurrence
- General purpose Function in python

```
course = 'Python for Beginners'
print(course.find('P'))
```

The screenshot shows a code editor with the line `course = 'Python for Beginners'`. Below it, the line `print(course.find('P'))` is shown. A blue arrow points from the code to a white box containing the text `Output = 0`.

### Replace() :

- replace a group of character with another one
- This method is Case sensitive , if we typed 'M' it wont find it

```
name1 = 'mohamed'
print(name1.replace('m', 'y'))
```

The screenshot shows a code editor with the line `name1 = 'mohamed'`. Below it, the line `print(name1.replace('m', 'y'))` is shown. A blue arrow points from the code to the word `yohayed`, which is the result of replacing all 'm's with 'y's in the string.

### In :

- we use it to check the existence of a string
- Its Case sensitive

### Example :

Will search for 'mohamed' and return boolean value

```
print('mohamed' in name1)
```

### **input()**:

- the function takes a prompt (an optional string) as an argument, displays it to the user, and then waits for the user to enter some text.
- it returns a string

Example:

```
age = input("Enter your age: ")
```

- ◆ You can provide a default value to the `input()` function by adding a keyword argument `default` with the desired default value inside the function call.
- ◆ If the user presses Enter without typing anything, the default value will be used instead.

Example:

```
# Using input() with a default value and distinguishing between empty and default
name = input("Enter your name (or press Enter for default): ")
name = name if name else "Guest"

# Displaying the entered value or the default value
print("Hello, " + name + "!")
```

### **upper() Method:**

- Purpose: Converts all the characters in a string to uppercase.

### **capitalize() Method:**

- Purpose: Capitalizes the first character of a string while converting the rest to lowercase.

### **int() : syntax ----> int(value, base [optional])**

- The `int()` function converts the specified value into an integer number.

```
● ● ●
1 print("int(123) is:", int('123'))
2
3 # converting a string (in binary) to integer
4 print("For 0b101, int is:", int("0b101", 2))      # int is: 5
5
6 # converting a string (in octal) to integer
7 print("For 0o16, int is:", int("0o16", 8))        # output : int is: 14
8
9 # converting a string (in Hex) to integer
10 print("For 0xA, int is:", int("0xA", 16))         # output : int is: 10
```

```
● ● ●
1 # Given hexadecimal string
2 number = '270188'
3
4 # Step 1: Separate every 2 bytes in a list
5 number_list = [number[i:i+2] for i in range(0, len(number), 2)]
6 print(number_list) # Output: ['27', '01', '88']
7
8 # Step 2: Convert each hexadecimal string to its integer value
9 integer_values = [int(hex_str, 16) for hex_str in number_list]
10 print(integer_values) # Output: [39, 1, 136]
```

# String

- ❖ In Python, a string is a sequence of characters, and it is one of the basic data types.  
Strings can be created using single ('') or double ("") quotes

## Formatted Strings ([f-strings](#))

- In Python, f-strings provide a concise and convenient way to embed expressions inside string literals.
- Inside the f-string, you can include expressions enclosed in curly braces {}.
- These expressions are evaluated at runtime and then formatted into the string.

```
● ● ●
1 name = "Alice"
2 age = 30
3
4 # Using f-string to format a string
5 formatted_string = f"My name is {name} and I am {age} years old. "
```

### Split():

- Purpose: split the string based on the input (" ") --> split string when find space
- return : a List
- note : default delimiter (white-space)

```
csv_data = "apple,orange,banana,grape"
fruits = csv_data.split(',')
```

### Example :

- We can use split with index to return the specific value we want

```
● ● ●
1 name = r"C:\Users\user\Desktop\Python OS.docx"
2
3 directory= name.split("\\")[0]
4 file_name= name.split("\\")[-1]
5
6 print(directory) # output --> C:
7 print(file_name) # output --> Python OS.docx
```

join(): syntax ----> string.join(iterable)

- it join a separator inside any iterable like (list,string,tuple) ----> more like combine something between items
- return string

```
● ● ●
1 """ example 1 """
2 words = ["Hello", "world"]
3 sentence = " ".join(words) # use a space as the separator
4 print(sentence)           # prints "Hello world"
5
6
7 """ example 2 """
8 number = '270188' # the string to be modified
9 number = " ".join ([number [i:i+2] for i in range (0, len (number), 2)]) # use the join method and a list comprehension to insert spaces
10 print(number)        # '27 01 88'
```

# Arithmetic Operations

## Division (/):

Example : `result = 5 / 3`

- This will assign the value 1.6666666666666667 to the variable result.
- In Python 3, the division operator (/) always returns a float.

## Floor Division (//):

Example : `result = 5 // 3`

- This will assign the value 1 to the variable result.
- The // operator performs division and rounds down to the nearest integer.

## Exponentiation (\*\*):

Example : `result = 5 ** 3`

- This will assign the value 125 to the variable result.
- The \*\* operator is used for exponentiation.

## augmented assignment operator

- ❖ They provide a concise way to perform an operation and update the value of a variable in a single step.

```
x = 5
x += 3 # Equivalent to x = x + 3

y = 10
y -= 4 # Equivalent to y = y - 4

z = 3
z *= 2 # Equivalent to z = z * 2

w = 8
w /= 2 # Equivalent to w = w / 2

v = 7
v //= 3 # Equivalent to v = v // 3

u = 11
u %= 4 # Equivalent to u = u % 4

t = 2
t **= 3 # Equivalent to t = t ** 3
```

## operator precedence

- ❖ It determines the order in which operations are performed in an expression.  
The expression is evaluated based on the priority of operators.
- The operator precedence in Python is listed in the following table.  
It is in descending order (upper group has higher precedence than the lower ones).

Operators	Meaning
<code>( )</code>	Parentheses
<code>**</code>	Exponent
<code>+X</code> , <code>-X</code> , <code>~X</code>	Unary plus, Unary minus, Bitwise NOT
<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>	Multiplication, Division, Floor division, Modulus
<code>+</code> , <code>-</code>	Addition, Subtraction
<code>&lt;&lt;</code> , <code>&gt;&gt;</code>	Bitwise shift operators
<code>&amp;</code>	Bitwise AND
<code>^</code>	Bitwise XOR
<code> </code>	Bitwise OR
<code>==</code> , <code>!=</code> , <code>&gt;</code> , <code>&gt;=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>is</code> , <code>is not</code> , <code>in</code> , <code>not in</code>	Comparisons, Identity, Membership operators
<code>not</code>	Logical NOT
<code>and</code>	Logical AND
<code>or</code>	Logical OR

# Mathematical Functions

- ❖ In Python, you can perform various mathematical operations using built-in functions and operators.

Here are some common mathematical functions and operations:

## 1- Absolute Value

```
result = abs(-5) # Output: 5
```

## 2- Round

- the function will return the nearest integer.

```
result = round(5.67) # Output: 6
```

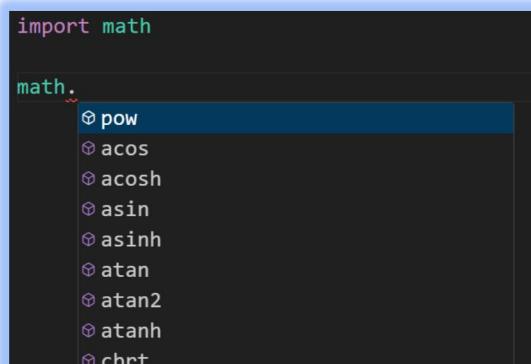
## 3- Math Module

- Python also provides a math module for more advanced mathematical functions:

You can import Math Library and type (math.) ,you will see all supported methods

```
import math

math.
```



```
# Square Root
result = math.sqrt(25) # Output: 5.0

# Trigonometric Functions
sin_value = math.sin(math.radians(30)) # Output: 0.4999999999999994

# Logarithmic Functions
log_value = math.log(10, 2) # Output: 3.3219280948873626
```

## Python If Statement

- ❖ Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

```
if condition:  
    # Code to execute when the condition is true
```

Example :

```
x = 5  
  
if x > 5:  
    print("x is greater than 5")  
elif x == 5:  
    print("x is equal to 5")  
else:  
    print("x is less than 5")
```

## Logical Operators

- ❖ In Python, logical operators are used to perform logical operations on boolean values.
- ❖ These operators allow you to combine or manipulate boolean expressions, resulting in a new boolean value.
- ❖ There are three main logical operators in Python: `and`, `or`, and `not`.

```
# Sample variable
x = 10
y = 5
is_sunny = True

# Using logical operators in an if condition
if x > 5 and y < 8:
    print("Both conditions are True.")
elif x > 5 or y > 10:
    print("At least one condition is True.")

if not is_sunny:
    print("It's not sunny today.")
else:
    print("It's sunny today.")
```

# While Loop

- ❖ With the while loop we can execute a set of statements as long as a condition is true.

**Example:**

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

## 1- The break Statement :

With the **break** statement we can stop the loop even if the while condition is true:

**Example:**

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

## 2- The continue Statement :

With the continue statement we can stop the current iteration, and continue with the next:

**Example:**

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

## 3- The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

**Example:**

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

# Python For Loops

- ❖ A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- ❖ This is less like the **for** keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

```
for variable in iterable:  
    # Code to be executed for each iteration
```

## Looping Through a List

Even strings are iterable objects, they contain a sequence of characters:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
  
apple  
banana  
cherry
```

## Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

```
for x in "banana":  
    print(x)  
  
b  
a  
n  
a  
n  
a
```

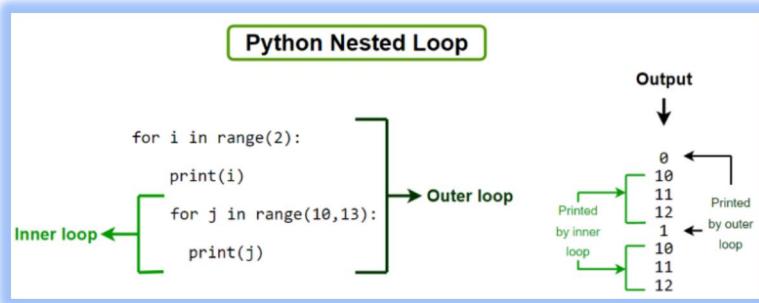
### range()

- The range() function in Python is used to generate a sequence of numbers. It's commonly used in for loops to iterate over a sequence of numbers
- It starts from index until the last index-1

```
for i in range(2, 8):  
    print(i)
```

```
for i in range(1, 10, 2):  
    print(i)  
  
# In this example, the range() function generates a sequence starting from 1,  
# up to (but not including) 10, with a step of 2.
```

## Nested for loop

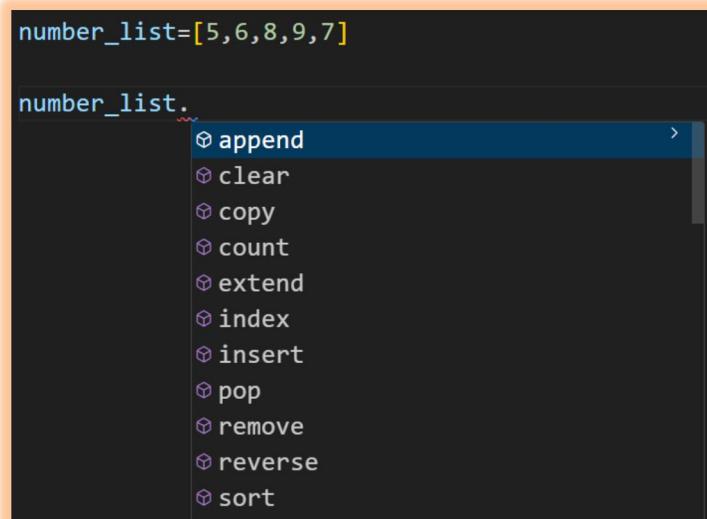


## List

- Python Lists are just like dynamically sized arrays, declared in other languages (vector in C++).
- List items can be of any data type
- A list can contain different data types

```
list1 = ["abc", 34, True, 40, "male"]
```

## List methods



### **append()** :

- add new item to the end of the List

### **pop()** :

- remove item from the end of the List

### **insert()** :

- add new item to the list in specific location

### **remove()** :

- remove the first occurrence of item in the List

### **clear()** :

- empty the list

### **index()** :

- find if a number exist or not (return error) and its first occurrence index  
- So its safer to use print(50 in number\_list) .. return false if not found but no error

### **count()** :

- find number of occurrence in list

### **sort()** :

- sort list in ascending order and has no return

### **sort(reverse=True)** :

- sort list in descending order and has no return

### **reverse()** :

- you can use it after sort to descending order your list

### **copy()** :

- take a copy from list and return list

### **sorted()** :

- take a copy from list and return new sorted list , doesn't modify our list like sort()

# Tuple

- ❖ Tuples are used to store multiple items in a single variable.
- ❖ Unlike list , we cant add items or remove or modify them

```
tuple = ("apple", "banana", "cherry")
```

- ❖ We can only get information about it not change it
- ❖ Use it when you don't want anything to change your data

A screenshot of a Python code editor showing the following code:

```
numbers = (1,2,3)
numbers.
```

The cursor is at the end of the word "numbers" after the dot. A dropdown menu is open, showing two options:

- ↪ count
- ↪ index

# Lambda function

```
1 lambda arguments: expression
```

- ❖ Lambda is a keyword that allows you to create small anonymous functions, which are functions that don't have a name.
  - You can use lambda to define a function in one line, without using the def keyword.
  - It returns a function object that has the same type as a regular function

## Usage:

- You can also use lambda functions as arguments to other functions that take a callable object, such as sorted, sum, map, filter, etc.

## List sorting

- ❖ When using sort(), you can use the **key parameter** to perform more customized sorting operations.
  - The value assigned to the key parameter needs to be something callable.
  - Callable is something that can be called, which means it can be invoked like methods and functions.

### that's why we use lambda function

- If you want to sort the list by specific value in the tuple, like the number, You should use lambda function to return a key that you use.

In this example, the key parameter is a lambda function that takes a tuple as an argument and returns the second element of the

```
1 items = [
2     ("product2", 25),
3     ("product1", 15),
4     ("product0", 24),
5     ("product3", 22)
6 ]
7
8 items.sort(key=lambda x: x[1])
9
10 print(items)
11
```

x = is the argument that represents the tuple  
x[1] = is the index that returns the second value of the tuple

Lambda is way better than doing this

```
1 items = [
2     ("product2", 25),
3     ("product1", 15),
4     ("product0", 24),
5     ("product3", 22)
6 ]
7
8
9 def sort_item(x):
10     return x[1]
11
12 items.sort(key=sort_item)
13
```

# Unpacking 1

## # Example 1

```
● ● ●  
1 coordinates = (3, 4)  
2 x, y = coordinates
```

## # Example 2

```
● ● ●  
1 s = "hello"  
2 a, b, c, d, e = s
```

## # Example 3

```
● ● ●  
1 person = {'name': 'John', 'age': 30, 'city': 'New York'}  
2 name, age, city = person.values()
```

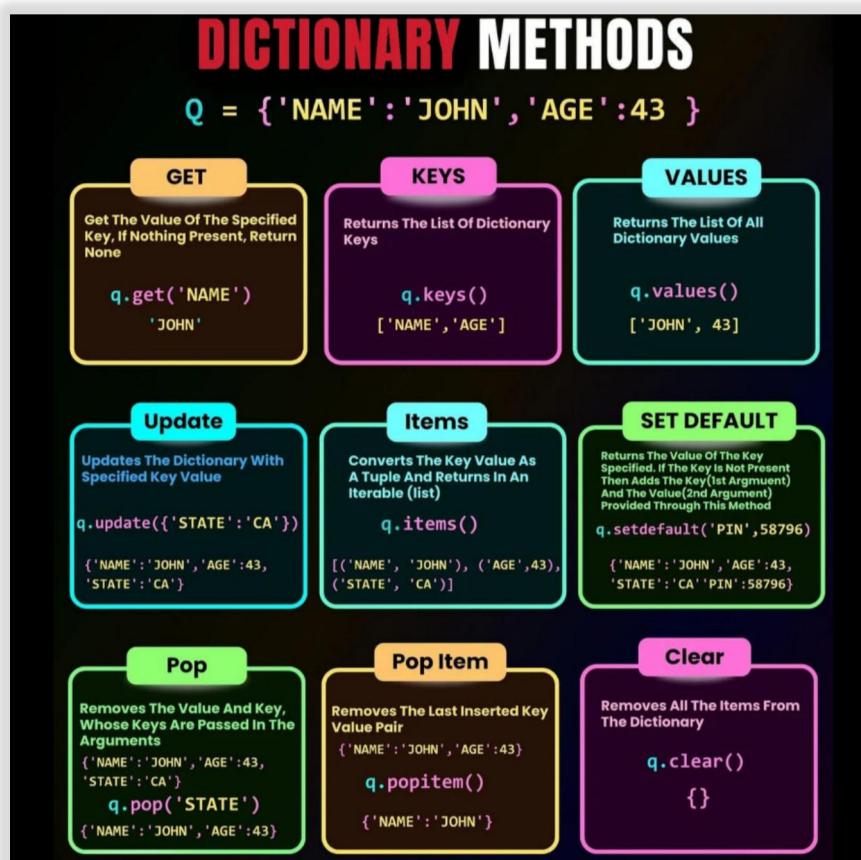
# Dictionary

- ❖ Dictionaries are used to store data values in key:value pairs.
- ❖ A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.
- ❖ Dictionary items are presented in **key:value pairs**, and can be referred to by using the key name.
- ❖ The values in dictionary items can be of any data type:

## # Example 1

```
● ● ●
1 thisdict = {
2     "brand": "Ford",
3     "electric": False,
4     "year": 1964,
5     "colors": ["red", "white", "blue"]
6 }
7 print(thisdict["brand"])
```

- ◆ We can access dic using `[ ]` but if we write the wrong name it would give an error  
`print(customer["Name"])`
- ◆ So we better use `.get()`, in case of error it return boolean value 'None'  
`print(customer.get("name"))`
- ◆ we use default value of the name not found  
`print(customer.get("name","1"))`



# Python Functions

- ❖ A function is a block of code which only runs when it is called.
- ❖ Function by default return None

```
def my_function():
    print("Hello from a function")

my_function()
```

## Arguments vs parameters :

- argument --> are the actual values that are passed to the function when it is invoked or called
- parameters --> are the variables that are defined or used inside the parentheses while defining a function.

## Keyword Arguments

- You can also send arguments with the key = value syntax.
- This way the order of the arguments does not matter.
- Used to improve readability

```
=====
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
=====
```

- Must use keyword argument after positional argument

```
=====
my_function("Tobias", child1 = "Emil")
=====
```

# Exceptions

- ❖ there are several built-in Python exceptions that can be raised when an error occurs during the execution of a program.
- ❖ Exceptions: Exceptions are raised when the program is syntactically correct, but the code results in an error

**Here are some of the most common types of exceptions in Python:**

**SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.

**TypeError:** This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.

**NameError:** This exception is raised when a variable or function name is not found in the current scope.

**IndexError:** This exception is raised when an index is out of range for a list, tuple, or other sequence types.

**KeyError:** This exception is raised when a key is not found in a dictionary.

**ValueError:** This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.

**AttributeError:** This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.

**IOError:** This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.

**ZeroDivisionError:** This exception is raised when an attempt is made to divide a number by zero.

**ImportError:** This exception is raised when an import statement fails to find or load a module.

# Class

- A Class is like an object constructor, or a "blueprint" for creating objects.

## Naming:

Capitalize every letter of every word

Class PointOption:

## Object :

Object in python is dynamic , you can add new attributes to it without the need to modify the class definition

## Constructor :

Its magic method and is executed when making new object

self : its reference to the current object

```
mypoint = Point(3, 4)
```

The \_\_init\_\_ method is automatically called with self representing the mypoint instance

```
class Point: def __init__(self, x, y):
```

```
•••
1 class Point:
2     def __init__(self,x,y): # creating Constructor
3         self.x=x
4         self.y=y
5
6     def draw(self):
7         print(f"x={self.x} , y={self.y}")
8
9 mypoint=Point(1,2) # creating object
10 mypoint.draw()
11
12
```

When we call function `draw()` , python by default gives it current object for self  
Like : `mypoint.draw(mypoint)`

So we don't have to do that

## Instance attributes vs class attributes

Instance attributes : they belong to the created object

Class attributes : they belong to any object we create ( like global variables )

## Instance methods vs class methods

Instance methods : any method created inside a class

Class methods (factory method): they used to create initialization method that return object

## Magic Methods

- ❖ They called automatically by python interpret when we create an object
- 1- **\_\_init\_\_** method in Python is used to initialize objects of a class. It is also called a constructor  
2- **\_\_str\_\_** used to obtain string representations of objects

## Comparing Objects:

```
● ● ●
1 point1=Point.zero() # creating object
2 point2=Point.zero() # creating object
3
4 print(point1 == point2) # --> output : False
```

Comparing gives false because by default (==) compare the addresses for these objects , so we have to use Magic methods for comparison:

This Magic methods defines the behaviour when comparing objects

**\_\_eq\_\_(self, other)**

Defines behavior for the equality operator, ==.

**\_\_ne\_\_(self, other)**

Defines behavior for the inequality operator, !=.

**\_\_lt\_\_(self, other)**

Defines behavior for the less-than operator, <.

**\_\_gt\_\_(self, other)**

Defines behavior for the greater-than operator, >.

**\_\_le\_\_(self, other)**

Defines behavior for the less-than-or-equal-to operator, <=.

**\_\_ge\_\_(self, other)**

Defines behavior for the greater-than-or-equal-to operator, >=.

Note :

If you define **\_\_gt\_\_** you don't have to define **\_\_lt\_\_** , python will figure out what to do

## Performing Arithmetic Operations on objects

We also use magic methods.

`__add__(self, other)`

Implements addition.

`__sub__(self, other)`

Implements subtraction.

`__mul__(self, other)`

Implements multiplication.

`__floordiv__(self, other)`

Implements integer division using the // operator.

`__div__(self, other)`

Implements division using the / operator.

`__truediv__(self, other)`

Implements true division. Note that this only works when `from __future__ import division` is in effect.

`__mod__(self, other)`

Implements modulo using the % operator.

`__divmod__(self, other)`

Implements behavior for long division using the `divmod()` built in function.

`__pow__`

Implements behavior for exponents using the \*\* operator.

`__lshift__(self, other)`

Implements left bitwise shift using the << operator.

`__rshift__(self, other)`

Implements right bitwise shift using the >> operator.

`__and__(self, other)`

Implements bitwise and using the & operator.

`__or__(self, other)`

Implements bitwise or using the | operator.

`__xor__(self, other)`

Implements bitwise xor using the ^ operator.

# Custom Containers

- To create a custom container in Python, you can define a new class that inherits from an existing container class, such as `list` or `dict`.
- You can then add your own methods and attributes to the new class to create a custom container that meets your needs

## The magic behind containers

### `__getitem__(self, key)`

Defines behavior for when an item is accessed, using the notation `self[key]`. This is also part of both the mutable and immutable container protocols. It should also raise appropriate exceptions: `TypeError` if the type of the key is wrong and `KeyError` if there is no corresponding value for the key.

### `__setitem__(self, key, value)`

Defines behavior for when an item is assigned to, using the notation `self[key] = value`. This is part of the mutable container protocol. Again, you should raise `KeyError` and `TypeError` where appropriate.

### `__iter__(self)`

Should return an iterator for the container. Iterators are returned in a number of contexts, most notably by the `iter()` built in function and when a container is looped over using the form `for x in container:`. Iterators are their own objects, and they also must define an `__iter__` method that returns `self`.

### `__len__(self)`

Returns the length of the container. Part of the protocol for both immutable and mutable containers.

```
 1 class TagCloud:
 2     def __init__(self): # creating Constructor
 3         self.tags={} # Creating dictionary
 4
 5     def add(self,tag):
 6         self.tags[tag.lower()]= self.tags.get(tag.lower(),0) + 1 # add to dict new value (tag+1)
 7
 8     def __getitem__(self,tag): # Defines behavior for when an item is accessed, using the notation self[key]
 9         return self.tags.get(tag.lower(),0)
10
11     def __setitem__(self,tag,value): # Defines behavior for when an item is assigned to, using the notation self[key] = value
12         self.tags[tag.lower()] = value
13
14     def __len__(self):
15         return len(self.tags) # return the length of dictionary
16
17     def __iter__(self):
18         return iter(self.tags) # return iterator so we can use it in loops
19
20     def __str__(self):
21         return f"{self.tags}"
22
23
24
25 cloud = TagCloud() #create Object
26 cloud.add("Python") # add new key to dict
27 cloud.add("C++") # add new key to dict
28
29 print(cloud["python"]) # find a key and print it
30 cloud["python"] = 5 # update key pair by using __setitem__ , because we can only access dict not class using []
31 print(cloud["python"]) # print the key in dict
32
33 print(cloud) # print your class by using __str__
```

To make our dictionary private we just need to add `(__)` before its name

```
● ● ●
1 class TagCloud:
2     def __init__(self): # creating Constructor
3         self.__tags={} # Creating dictionary
```

This is more like warning (don't touch its private ) But we still can access it  
To know more :

Each class has `(__dict__)` this contains all the attributes in any class  
If we print it we see `(__TagCloud__tags)` which we can use to see the dict content

# Properties

## What is property() Function in Python

- Python `property()` function is a built-in function that allows us to create a special type of attribute called a property for a class.
- Properties are used to encapsulate the access to an object attribute and to add some logic to the process such as computation, access control, or validation.

## Below are the ways by which we can create property for a class in Python:

- Using `property()` method -- we still can access methods (getter and setter)
- Using `@property` decorator -- (cleaner code and hide getter and setter methods)

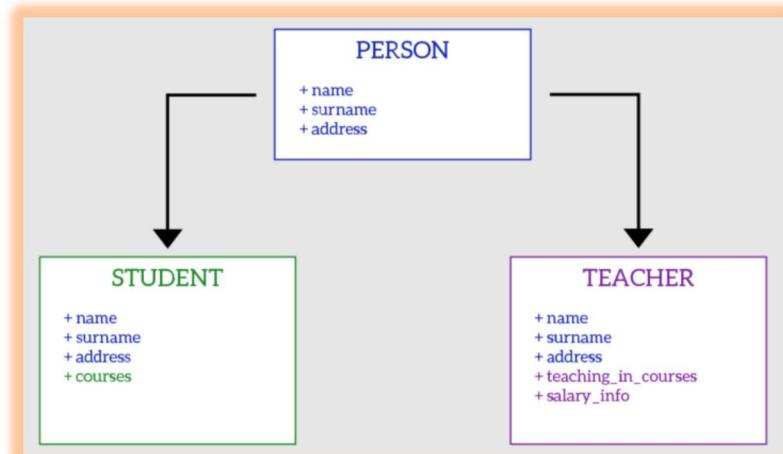
```
1 class Price:
2     def __init__(self,price):
3         self.set_price(price)
4
5     def get_price(self):
6         return self.__price
7
8     def set_price(self,value):
9         if value<0:
10             raise ValueError("price cant be negative")
11         self.__price=value
12
13 # making a setter and getter for our attribute (price)
14 price=property(get_price,set_price)
15
16 product = Price(5)
17 product.price=25      # this will go to setter method
18 print(product.price) # this will go to getter method and return our needed value
19
```

```
1 class Price:
2     def __init__(self,price):
3         self.__price=price      #make our attribute private by typeing __ before it
4
5     # getting the values
6     @property
7     def price(self):
8         return self.__price
9
10    @price.setter
11    def price(self,value):
12        if value<0:
13            raise ValueError("price cant be negative")
14        self.__price=value
15
16
17 product = Price(5) #create object
18
19 product.price=22      # set property
20 print(product.price) # get property
21
```

# Inheritance

## What is Inheritance used for

- we use it when we have repeated code in few classes and we want them to inherit this from one class only (Parent Class), so that if a problem happened in the code we don't have to fix it in every class
- We can inherit methods and attributes as well



- We use super() method to access our parent class from child class

```
1 class Person:
2     def __init__(self, fname, lname):
3         self.firstname = fname
4         self.lastname = lname
5
6     def printname(self):
7         print(self.firstname, self.lastname)
8
9 class Student(Person):
10    def __init__(self):
11        super().__init__("mohamed", "ahmed") # Access Parent class (Person)
12
13
14 x = Student() # creating student object will inherit from Person class
15 x.printname()
```

- Method Overriding
  - init method of student overwrite the init of person class  
That's why we call super.\_\_init\_\_ to call the parent call init

# Object Class

## What is this ?

- Every class in python derived from class called object
- We can use important method called `issubclass(child, parent)` to see if any class is derived from another class

# Multiple Inheritance

## What is Inheritance used for

When we inherit the same method from 2 classes , it runs the first one inherited

```
● ● ●
1 class Employee:
2     def greet(self):
3         print("hi Employee")
4
5 class Person:
6     def greet(self):
7         print("hi Person")
8
9 class Manager(Employee,Person):
10    pass
11
12
13 mr=Manager()
14 mr.greet() # will output hi Employee
15
```

# Python Abstraction Base Classes

## What is it ?

- Abstraction Base Classes (ABCs) in Python are a way to define abstract classes and abstract methods.
- An abstract class is a class that cannot be instantiated and is meant to be subclassed by non-abstract classes.
- **Abstract methods** within an abstract class are declared but do not provide an implementation.
- **Subclasses** must provide their own implementation of these abstract methods.
- We can't make instance (object) from abstracted class

## When we use it ?

- If we have a method in Parent class and want to inherit it in every class Then we should use abstraction Base class and abs decorator
- To tell the interpreter that this method is not implemented in this new class so we don't forget implementing it

```
1 from abc import ABC , abstractmethod
2
3
4 class InvalidOperationError(Exception): # create a custom exception
5     pass
6
7 class Stream(ABC): # Parent Class
8     def __init__(self):
9         self.opened=False
10
11    def open(self):
12        if self.opened:
13            raise InvalidOperationError("stream is already opened")
14        self.opened=True
15
16    def close(self):
17        if not self.opened:
18            raise InvalidOperationError("stream is already closed")
19        self.opened=False
20
21    @abstractmethod
22    def read(self): ←
23        pass
24
25
26 class FileStream(Stream):
27     def read(self):
28         print("reading data from file")
29
30
31 class NetworkStream(Stream):
32     pass
33
34
35
36
37 network1 = NetworkStream() # error --> Cannot instantiate abstract class "NetworkStream", "Stream.read" is abstract
```

- ❖ This enforce the Subclasses to implement a method

# Python Polymorphism

## What is it ?

- The word "Polymorphism" means "[many forms](#)", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

## Function Polymorphism

For strings `len()` returns the number of characters:

For tuples `len()` returns the number of items in the tuple:

For dictionaries `len()` returns the number of key/value pairs in the dictionary:

## Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

Like `draw()` method here

```
● ● ●
1 from abc import ABC , abstractmethod
2
3 class UIControl(ABC):
4     @abstractmethod
5     def draw(self):
6         pass
7
8 class TextBox(UIControl):
9     def draw(self):
10        print("TextBox")
11
12 class DropDownList(UIControl):
13     def draw(self):
14        print("DropDownList")
15
16
17 # Function to draw multiple controls
18 def draw(controls):
19     for control in controls:
20         control.draw()
21
22 ddl=DropDownList()
23 textbox=TextBox()
24
25 draw([ddl,textbox]) # pass objects to this function draw as dictionary
```

## To achieve Polymorphism behaviour

1-define a base class

2-In this class define a common behaviour (common method)

# Duck Typing in Python

## What is it ?

- Duck Typing is a type system used in dynamic languages.
- where the type or the class of an object is less important than the method it defines.
- Using Duck Typing, we do not check types at all. Instead, we check for the presence of a given method or attribute.

```
● ● ●  
1 # Function to draw multiple controls  
2 def draw(controls):  
3     for control in controls:  
4         control.draw()
```

Like this function draw()

It only looks for the existence of draw() method not the type of control

## Example 2:

```
● ● ●  
1 class Duck:  
2     def quack(self):  
3         return "Quack, quack!"  
4  
5 class Dog:  
6     def quack(self):  
7         return "Woof, woof!"  
8  
9 class Car:  
10    def drive(self):  
11        return "Vroom, vroom!"  
12  
13 # Function that takes any object with a 'quack' method  
14 def make_sound(entity):  
15     return entity.quack()  
16  
17 # Creating instances  
18 duck_instance = Duck()  
19 dog_instance = Dog()  
20 car_instance = Car()  
21  
22 # Using the function with different types of objects  
23 print(make_sound(duck_instance)) # Output: Quack, quack!  
24 print(make_sound(dog_instance)) # Output: Woof, woof!  
25 print(make_sound(car_instance)) # Output: Vroom, Vroom!
```

## Extending Built-in Types

### Example:

```
● ● ●  
1 class Text(str):  
2     def duplicate(self):  
3         return self+self # will act as string and concatenate them  
4  
5 class TrackableList(list):  
6     def append(self,object): # overdrive append  
7         print("append called") # add new feature to append  
8         super().append(object) # call list to execute append  
9  
10 list = TrackableList([1,2,3])  
11  
12 list.append(88)  
13 print(list) # output : [1,2,3,88] and print "append called"
```

# Python Modules

## What is a Module?

Consider a module to be the same as a code library.  
A file containing a set of functions you want to include in your application.

## Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

## Use a Module

Now we can use the module we just created, by using the `import` statement:

```
● ● ●
1 import mymodule
2
3 mymodule.greeting("Jonathan")
```

## Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

```
● ● ●
1 import mymodule as mx
2
3 a = mx.person1["age"]
4 print(a)
```

## Using the `dir()` Function

There is a built-in function to list all the function names (or variable names) in a module

```
● ● ●
1 import platform
2
3 x = dir(platform)
4 print(x)
```

## Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

```
● ● ●
1 from mymodule import person1
2
3 print (person1["age"])
```

# Python Packages

## What is a Python Package?

Package in Python is a folder that contains various modules as files.

### Creating Package

Let's create a package in Python named mypckg that will contain two modules mod1 and mod2. To create this module follow the below steps:

- Create a folder named mypckg.
- Inside this folder create an empty Python file i.e. `__init__.py`
- Then create two modules mod1 and mod2 in this folder.

### Understanding `__init__.py`

- ◆ `__init__.py` helps the Python interpreter recognize the folder as a package.
- ◆ It also specifies the resources to be imported from the modules.
- ◆ If the `__init__.py` is empty this means that all the functions of the modules will be imported.

### Import Modules from a Package

We can import these Python modules using the `from...import` statement and the `dot(.)` operator.

```
● ● ●  
1 import package_name.module_name  
2
```

# Working With Paths

```
● ● ●
1 # Working With Paths
2 from pathlib import Path
3
4 # create absolute path
5 myRepo_abs = Path(
6     r"D:\1 Embedded\My Repo's\Studying Python") # r = raw string --> ignore escape sequence like \
7 # create relative path
8 myRepo_relative = Path(
9     r"mypackages\text.txt")
10
11 print(myRepo_abs.exists())
12 print(myRepo_abs.is_file())
13 print(myRepo_abs.is_dir())
14
15 print(myRepo_relative.exists()) # does it exist
16 print(myRepo_relative.is_file()) # is it file or not
17 print(myRepo_relative.is_dir()) # is it directory or not
18 print(myRepo_relative.name) # file name
19 print(myRepo_relative.suffix) # file extension type
20 print(myRepo_relative.stem) # file name without extension
21 print(myRepo_relative.parent) # parent folder where file exist
22
23 # file not exist yet , we only represent its path
24 path = myRepo_abs.with_name("add_file.txt")
25 print(path) # D:\1 Embedded\My Repo's\Studying Python\add_file.txt
26
27 # change extension of the path or file
28 path = myRepo_abs.with_suffix(".exe")
29 print(path) # D:\1 Embedded\My Repo's\Studying Python.exe
30
```

# List Comprehension

```
● ● ●
1 # Traditional for loop to create a list of squares
2 squares = []
3 for i in range(5):
4     squares.append(i ** 2)
5
6 # Using list comprehension for the same result
7 squares_comp = [i ** 2 for i in range(5)]
```

# Working With Directories

```
● ● ●
1 # Working With Paths
2 from pathlib import Path
3
4 # create absolute path
5 myRepo_abs = Path(
6     r"D:\1 Embedded\My Repo's\Studying Python") # r = raw string --> ignore escape sequence like \
7
8 # create relative path
9 myRepo_relative = Path(r".") # current directory
10
11 # print(myRepo_relative.exists()) # does it exist
12 # print(myRepo_relative.mkdir()) # create this directory
13 # print(myRepo_relative.rmdir()) # remove this directory
14 # print(myRepo_relative.name) # directory name
15 # print(myRepo_relative.rename) # rename this directory
16
17 # it return the list of files and directories (as generator) but not not recursively
18 print(myRepo_relative.iterdir())
19
20 # iterate through repo folder and print each (folder or file) inside but not recursively
21 for p in myRepo_relative.iterdir():
22     print(p)
23
24
25 paths = [p for p in myRepo_relative.iterdir() if p.is_dir()]
26 # output: [WindowsPath('.git'), WindowsPath('mypackages')]
27 print(paths)
28 ######
29 # Traditional for loop to create a list of squares
30 squares = []
31 for i in range(5):
32     squares.append(i)
33 # Using list comprehension for the same result
34 squares_comp = [i for i in range(5)]
35 #####
36
37 # we use glob method to fix iterdir() problems like recursive and search specific files
38 py_files = [pf for pf in myRepo_relative.glob("*.py")]
39 print(py_files)
40
41 # search recursively
42 py_files = [pf for pf in myRepo_relative.glob("**/*.py")]
43 print(py_files)
44
45 # search recursively using rglob
46 py_files = [pf for pf in myRepo_relative.rglob("*.py")]
47 print(py_files)
48
```

# Working With Files

```
● ● ●
1 # Working With Paths
2 from pathlib import Path
3 from time import ctime
4 import shutil
5
6 # create absolute path
7 myRepo_abs = Path(
8     r"D:\1 Embedded\My Repo's\Studying Python") # r = raw string --> ignore escape sequence like \
9
10 # create relative path
11 myRepo_relative = Path(r"mypackages/text.txt")
12
13 print(myRepo_relative.exists())           # does it exist
14 print(myRepo_relative.rename("_init_.py")) # rename this file
15 print(myRepo_relative.unlink())           # delete this file
16
17 # .st_ctime: This accesses the creation time attribute of the file or directory from the stat object
18 # ctime() : This function is used to convert a time in seconds to human readable time
19 # output: Sat Jan  6 12:22:30 2024
20 print(ctime(myRepo_relative.stat().st_ctime))
21
22 # these functions handle opening and closing the file
23 print(myRepo_relative.read_bytes()) # read binary data
24 print(myRepo_relative.read_text())
25
26 myRepo_relative.write_text("hello file") # overwrite the text in file
27
28
29 ######
30 # Copying File (not the best option)
31 #####
32 source = Path(r"mypackages/text.txt")
33 target = Path(r"mypackages") / "test_file.txt"
34
35 target.write_text(source.read_text())
36
37
38 #####
39 # Copying File using shutil
40 #####
41 source = Path(r"mypackages/text.txt")
42 target = Path(r"mypackages") / "test_file.txt"
43
44 shutil.copy(source, target)
45
```

# Python Package Index

## Pip

Its used to install other packages, its commands are:

- pip install
- pip list
- pip uninstall

## Virtual Environment

We use it to install our packages with specific versions sepearted from packages installed in default path , its commands are:

- python -m venv env      **create env**
- env\bin\activate.bat      **activate enb**
- deactivate

## Pipenv

- ❖ Pipenv automatically creates and manages a virtual environment for your projects in another location away from your project, as well as adds and removes dependencies from your Pipfile as you install.
- ❖ When we install package using pipenv , 2 files automatically generated to tracks of our files and dependencies

its commands are:

- pipenv shell : Spawns a shell within the virtualenv.
- exit
- pipenv install : will look at pipfile and install all our dependencies
- Pipenv --venv : show the location of env
- pipenv --rm : remove the env

## Pipfile

It keep tracks of our files and dependencies , so we can run our script on different machine without worrying about missing dependencies

- Pipenv install --ignore--pipfile

This commands ignore pipfile and install dependencies based on pipfile.lock

Because that's what we have on our development machine

# Documentation

## Pip

You can document your code (class , method) so when someone call them and hover on them a description appears , we call that intellisense

```
def convert(path):
    """
    paramters:
    path(str) : the path to pdf

    return:
    str: pdf content
    """
    print("ih")
No value for argument 'path' in fu
Argument missing for parameter "pa
(function) def convert(path: Unkn
paramters:
path(str) : the path to pdf
return:
str: pdf content
convert()
```

## Pydoc

You can use this command to print HTML documentation file about package or module

Command :

```
python -m pydoc -w math
```

The -w flag is used with pydoc to generate HTML documentation

The -m flag is used to run a module as a script instead of specifying a file path.

# Ternary operator

## Syntax

```
expression_if_true if condition else expression_if_false
```

## Example

```
● ● ●
1 # normal comparsion
2 age = 22
3 if age > 18:
4     message = "ok"
5 else:
6     message = "not ok"
7 print(message)
8
9 # Ternary operator
10 message = "ok" if age > 18 else "not ok"
11 print(message)
12
```

# Logical operator

- **and**
- **or**
- **not**

```
● ● ●
1 high_income = True
2 Good_credits = False
3 student = False
4
5 if (high_income or Good_credits) and not student:
6     print("ok")
7
```

# chaining comparison operators

```
● ● ●
1 # Chaining comparison operators with an if condition
2 x = 5
3 y = 10
4 z = 15
5
6 if x < y and y < z:
7     print("All conditions are satisfied.")
8 if x < y < z:
9     print("All conditions are satisfied.")
10
11 # they are the same ( with using and operator or without using it)
12
```

# Comparing strings

Python will compare the first character of each string, and if they are different, it will return the result based on their order in the alphabet. If they are the same, it will compare the next character, and so on.

```
● ● ●
1 print("apple" > "bag")    # False, because 'a' < 'b'
2 print("apple" > "ape")    # True, because 'p' > 'e'
3 print("apple" > "apple")  # False, because they are equal
4
```

# Iterable

## common iterable objects in Python:

### Built-in sequences:

- Lists: [1, 2, 3, 4]
- Tuples: (5, 6, 7)
- Strings: "hello"
- Sets: {8, 9, 10}
- Dictionaries (keys, values, or items): {'a': 1, 'b': 2}
- File objects: Opened files can be iterated through line by line.
- Custom objects: You can create custom iterable objects by defining the `__iter__()` method.

## iterate through dictionary

- ❖ you can iterate through a dictionary using various methods.
- ❖ Here are three common ways to iterate through the key-value pairs, keys, or values of a dictionary:
  - Iterating Through Key-Value Pairs ([items\(\)](#)).
  - Iterating Through Keys ([keys\(\)](#)).
  - Iterating Through Values ([values\(\)](#)).

```
● ● ●
1 my_dict = {'a': 1, 'b': 2, 'c': 3}
2
3 for key, value in my_dict.items():
4     print(f'Key: {key}, Value: {value}')
5     """ output """
6 """
7 Key: a, Value: 1
8 Key: b, Value: 2
9 Key: c, Value: 3
10 """
11
12
13 for key in my_dict.keys():
14     print(f'Key: {key}')
15     """ output """
16 """
17 Key: a
18 Key: b
19 Key: c
20 """
21
22
23 for value in my_dict.values():
24     print(f'Value: {value}')
25     """ output """
26 """
27 Value: 1
28 Value: 2
29 Value: 3
30 """
31
```

# Unpacking

- ❖ **(\*args)** and **(\*\*kwargs)** are special parameters that can be used to pass a variable number of arguments to a function.

For example, def add(\*args): return sum(args) defines a function that can take any number of numbers and return their sum. add(1, 2, 3) would return 6.

```
 1 colors = ["red", "green", "blue"]
 2 r, g, b = colors # unpacking the list
 3 print(r) # prints "red"
 4 print(g) # prints "green"
 5 print(b) # prints "blue"
 6
 7
 8 #####
 9 def average(a, b, c):
10     return (a + b + c) / 3
11
12 numbers = [10, 20, 30]
13 avg = average(*numbers) # unpacking the list with *
14 print(avg) # prints 20.0
15
16
17 #####
18 def greet(name, age):
19     print(f"Hello, {name}. You are {age} years old.")
20
21 person = {"name": "Alice", "age": 25}
22 greet(**person) # unpacking the dictionary with **
```

# enumerate

```
 1 numbers = ["a", "2", "3"]
 2
 3
 4 # enumerate is useful for obtaining an indexed list
 5 for index, letter in enumerate(numbers):
 6     print(index, letter)
 7
```

# os module

```
● ● ●
1 import os
2
3
4 # Get the current working directory
5 my_directory = os.getcwd()
6
7 # List files in the current working directory
8 files = os.listdir(my_directory)
9 print("Files in the current directory:", files)
10
11 # Create a new directory named "test file"
12 os.makedirs("test file")
13
14 # Combine the current directory and a filename
15 file1 = "file.txt"
16 combined = os.path.join(my_directory, file1)
17
18 # Check if the combined path exists
19 ret = os.path.exists(combined)
20 print("Path exists:", ret)
21
22 # Check if the combined path points to a file
23 ret = os.path.isfile(combined)
24 print("Is a file:", ret)
25
26 # Check if the current directory is a directory
27 ret = os.path.isdir(my_directory)
28 print("Is a directory:", ret)
29
30 # Get the size of a specific file (main.py) in the current directory
31 stats = os.stat(combined).st_size
32 print("Size of main.py:", stats, "bytes")
33
34 # Get the value of the PATH environment variable
35 v1 = os.environ.get("PATH")
36 print("PATH environment variable:", v1)
37
```

# Map function

```
●●●
1 items = [
2     ("product2", 25),
3     ("product1", 15),
4     ("product0", 24),
5     ("product3", 22)
6 ]
7
8 # Extracting values from tuples using a loop
9 # Create an empty list to store the second elements of each tuple (numbers)
10 prices = []
11
12 # Loop through each tuple in the 'items' list and append the second element to the 'prices' list
13 for item in items:
14     prices.append(item[1])
15
16
17 # Using the map function to achieve the same result in a more concise way
18 # Define a lambda function to extract the second element of each tuple
19 # Use the map function to apply the lambda function to each item in the 'items' list
20 # Convert the result to a list
21 prices2 = list(map(lambda item: item[1], items))
22
23
24
```

Inputs:

We give it a function object using lambda  
And list of tuples

behaviour

It use this function and give it items as inputs  
It return a map class contains the results  
Then use list() to convert it to a list

We can see that this lambds is  
the same as writing this function

```
●●●
1 def function(item):
2     return item[1]
```

# Positional arguments vs keyword arguments

## Positional Arguments:

- Positional arguments are passed to a function based on their position or order in the function definition

```
● ● ●  
1 def add_numbers(a, b):  
2     return a + b  
3  
4 result = add_numbers(3, 5)  
5 print(result) # Output: 8
```

## Keyword Arguments:

- Keyword arguments are passed to a function using the parameter names, explicitly specifying which value is assigned to which parameter.
- This allows you to pass arguments out of order.

```
● ● ●  
1 def greet(name, greeting):  
2     return f"{greeting}, {name}!"  
3  
4 message = greet(name="Alice", greeting="Hello")  
5 print(message) # Output: Hello, Alice!
```

## Mix of Positional and Keyword Arguments:

- You can use both positional and keyword arguments in a function call. However, positional arguments must come before keyword arguments.

```
● ● ●  
1 def describe_person(name, age, occupation):  
2     return f"{name} is {age} years old and works as a {occupation}."  
3  
4 info = describe_person("Bob", 30, occupation="engineer")  
5 print(info) # Output: Bob is 30 years old and works as an engineer.  
6
```

# class method vs static method

- ❖ both class methods and static methods are used to define methods that belong to a class rather than an instance of the class.
- ❖ The difference between them is that a class method takes the class itself as the first argument, while a static method does not take any implicit arguments.

## Class Method:

- They are used when you want to define a method that operates on the class itself, rather than on instances of the class.
- One useful example where class methods are important and needed is when you want to create alternative constructors for your class.
  - Alternative constructors are methods that return a new instance of the class with different parameters or configurations.

## Static Method:

- They are used when you want to define a method that does not operate on the class or its instances.
- They are often used to create utility functions that take some parameters and work upon those parameters.

```
 1 # python class method vs static method
 2 class Person:
 3     person_count = 0 # Class Variable
 4
 5     def __init__(self, name, age):
 6         self.name = name
 7         self.age = age
 8         Person.person_count += 1 # update Class Variable
 9
10     @classmethod
11     def from_birth_year(cls, name, year):
12         age = 2024 - year
13         return cls(name, age) # Return a new person object
14
15     @staticmethod
16     def is_adult(age):
17         return age >= 18
18
19 # Create a person object using the constructor
20 p1 = Person("Alice", 25)
21
22 # Create another person object using the class method
23 p2 = Person.from_birth_year("Bob", 1998)
24
25 # you can use static method without creating object
26 print(Person.is_adult(16)) # False
27 print(Person.is_adult(21)) # True
28
```

## Another example of Class Method:

- In this example you can use this class method to create a date object from a string, without having to manually typing the day, month, and year as arguments

```
● ● ●
1 class Date:
2     def __init__(self, day, month, year):
3         # Validate the input and assign the attributes
4         self.day = day
5         self.month = month
6         self.year = year
7
8     @classmethod
9     def from_string(cls, date_string):
10        # Split the string by "/"
11        day, month, year = map(int, date_string.split("/"))
12        # Return a new date object
13        return cls(day, month, year)
14
15
16 date = Date.from_string("15/04/2023")
17 print(date.day)    # 15
18 print(date.month) # 4
19 print(date.year)  # 2023
20
```

# List Comprehensions

- ❖ list comprehensions provide a concise way to create lists.
- ❖ They allow you to construct a list using a single line of code, making your code more readable and expressive.

Syntax ---> [expression for item in iterable if condition]

# Example 1 (map an items)

```
● ● ●
1 items = {
2     ("key1",10),
3     ("key2",19),
4     ("key3",15)
5 }
6
7 # theses 2 lines are doing the same thing
8 list0 = list(map(lambda item : item[1] , items)) # map() + list()
9 list1 = [item[1] for item in items]           # List Comprehensions
```

Its better to use list Comprehensions and less complications

# Example 2 (filter an items)

```
● ● ●
1 # theses 2 lines are doing the same thing
2 list2 = list(filter(lambda item : item[1] >= 10 , items))    # map() + list()
3 list3 = [item for item in items if item[1] >= 10 ]           # List Comprehensions
```

# ZIP Function

*What it does:*

- ❖ Takes two or more iterables (like lists, tuples, strings) as input.
- ❖ Pairs up corresponding elements from each iterable .
- ❖ Creates an iterator that yields tuples, where each tuple contains elements from the same position in the input iterables.

**Example :**

```
● ● ●  
1 fruits = ["apple", "banana", "cherry"]  
2 colors = ["red", "yellow", "pink"]  
3 prices = [1.25, 0.75, 1.50]  
4  
5 items = list(zip(fruits, colors, prices))  
6 print(items) # Output: [('apple', 'red', 1.25), ('banana', 'yellow', 0.75), ('cherry', 'pink', 1.5)]
```

*Key points:*

- The length of the zipped iterator is determined by the shortest iterable.
- You can convert the zip object to a list or other iterable using `list(zip(...))`.

*It's often used for:*

- Iterating over multiple iterables in parallel.
- Creating dictionaries from paired data.
- Transposing data structures (like rows to columns).

# Use Cases Examples:

## Example 1: (slice string)

- 1 - Using generator to loop through string and use list Comprehensions to return list.
- 2 - Using generator to loop through string and get pairs of string.

```
● ● ●
1 var = "2701"
2 var= [var[index:index+2] for index in range(0, len(var), 2)]
3 print(var) # Return List ... output : ['27', '01']
4
5
6 var = "2701"
7 var= (var[index:index+2] for index in range(0, len(var), 2))
8 print(var) # Return Generator ... output : <generator object <genexpr>
```

## Example 2: (add separator between strings )

- 1 - loop through string and get pairs of string then use "`".join()`

To concatenate all the pairs of characters we got from the generator expression into a single string, with a space (" ") inserted between each pair.

```
● ● ●
1 var = "270102"
2 var = " ".join(var[index:index+2] for index in range(0,len(var),2))
3 print(var) # output : 27 01 02
```

.....

