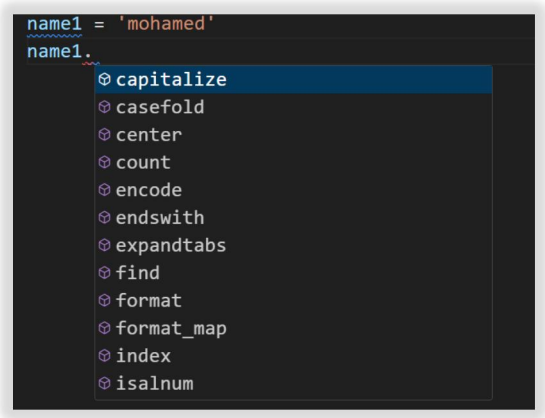
A python interpreter :

- ❖ 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).



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'))
```

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'))
```

```
yohayed
```

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 return 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.

```
print("int(123) is:", int('123'))

# converting a string (in binary) to integer
print("For 0b101, int is:", int("0b101", 2))    # int is: 5

# converting a string (in octal) to integer
print("For 0o16, int is:", int("0o16", 8))      # output : int is: 14

# converting a string (in Hex) to integer
print("For 0xA, int is:", int("0xA", 16))       # output : int is: 10
```

```
# Given hexadecimal string
number = '270188'

# Step 1: Separate every 2 bytes in a list
number_list = [number[i:i+2] for i in range(0, len(number), 2)]
print(number_list)  # Output: ['27', '01', '88']

# Step 2: Convert each hexadecimal string to its integer value
integer_values = [int(hex_str, 16) for hex_str in number_list]
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.

```python
name ="Alice"
age = 30

# Using f-string to format a string
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

```python
name = r"C:\Users\user\Desktop\Python OS.docx"

directory= name.split("\\")[0]
file_name= name.split("\\")[-1]

print(directory) # output --> C:
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

```python
""" example 1 """
words = ["Hello", "world"]
sentence = " ".join(words) # use a space as the separator
print(sentence)            # prints "Hello world"


""" example 2 """
number = '270188' # the string to be modified
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
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
`()`	Parentheses
`**`	Exponent
`+x`, `-x`, `~x`	Unary plus, Unary minus, Bitwise NOT
`*`, `/`, `//`, `%`	Multiplication, Division, Floor division, Modulus
`+`, `-`	Addition, Subtraction
`<<`, `>>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	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

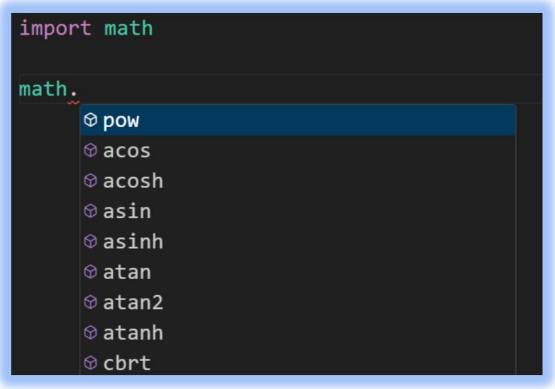
- the function will return the nearest integer.

```
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




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

# Trigonometric Functions
sin_value = math.sin(math.radians(30))  # Output: 0.49999999999999994

# 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.

```python
# 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
```

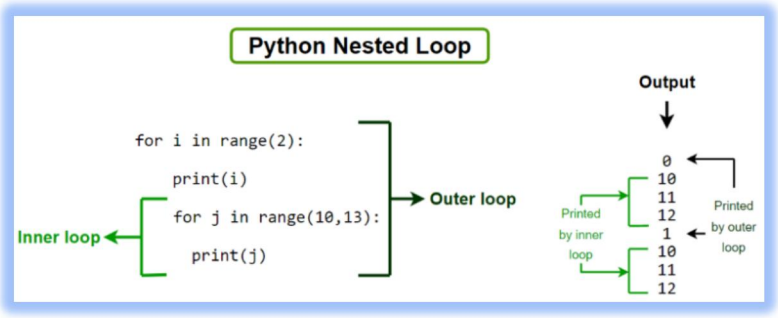
range() :

- 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 start 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 the 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

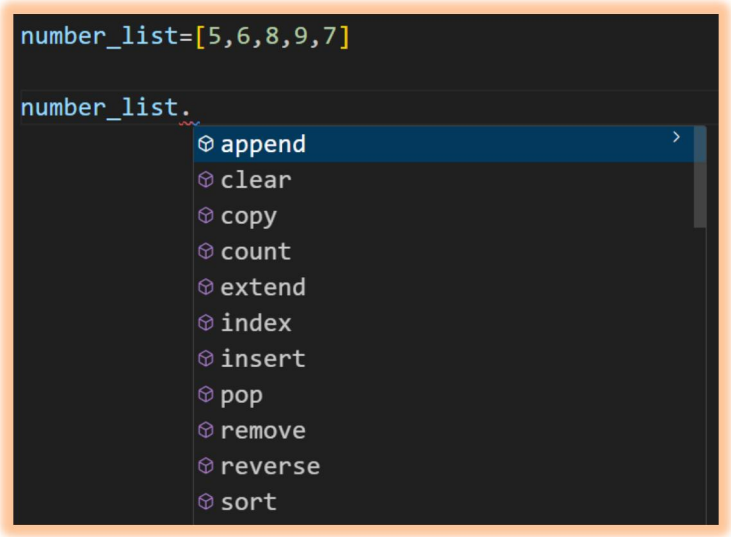


List

- 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

Lambda function

```
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 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

```
items = [
    ("product2", 25),
    ("product1", 15),
    ("product0", 24),
    ("product3", 22)
]

items.sort(key=lambda x: x[1])

print(items)
```

x = is the argument that represents the tuple

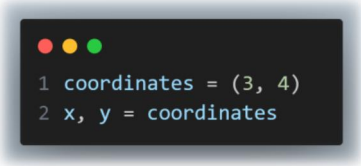
x[1] = is the index that returns the second value of the tuple

Lambda is way better than doing this

```
items = [
    ("product2", 25),
    ("product1", 15),
    ("product0", 24),
    ("product3", 22)
]


def sort_item(x):
    return x[1]

items.sort(key=sort_item)
```

Unpacking 1

 # Example 1



```
coordinates = (3, 4)
x, y = coordinates
```

 # Example 2

```
s = "hello"
a, b, c, d, e = s
```

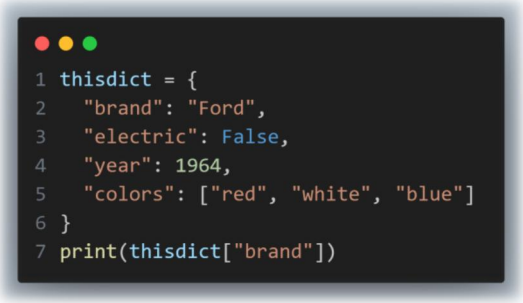
 # Example 3

```
person = {'name': 'John', 'age': 30, 'city': 'New York'}
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



```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
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 , 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

slef : 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
11 mypoint.draw()
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

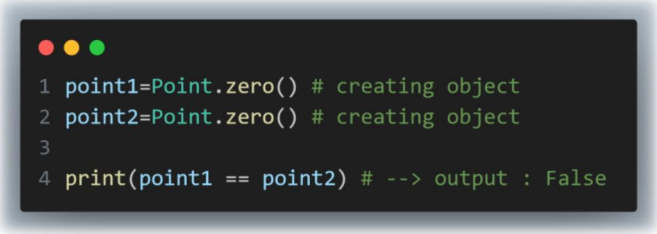
Instance methods : any method created inside a class
Class methods (factory method): they used to create initialization method that return object

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:



```
point1=Point.zero() # creating object
point2=Point.zero() # creating object

print(point1 == point2) # --> output : False
```

Comparing gives false because by default (==) compare the addresses for theses 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[nkey] = 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.

```python
class TagCloud:
    def __init__(self): # creating Constructor
        self.tags={}     # Creating dictionary

    def add(self,tag):
        self.tags[tag.lower()]= self.tags.get(tag.lower(),0) + 1 # add to dict new value (tag+1)

    def __getitem__(self,tag): # Defines behavior for when an item is accessed, using the notation self[key]
        return self.tags.get(tag.lower(),0)

    def __setitem__(self,tag,value): # Defines behavior for when an item is assigned to, using the notation self[nkey] = value
        self.tags[tag.lower()] = value

    def __len__(self):
        return len(self.tags) # return the length of dictionary

    def __iter__(self):
        return iter(self.tags) # return iterator so we can use it in loops

    def __str__(self):
        return f"{self.tags}"



cloud = TagCloud()      #create Object
cloud.add("Python")     # add new key to dict
cloud.add("C++")        # add new key to dict

print(cloud["python"])  # find a key and print it
cloud["python"] = 5     # update key pair by using __setitem__ , because we can only access dict not class using []
print(cloud["python"])  # print the key in dict

print(cloud)            # print your class by using __str__
```

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

```
class TagCloud:
    def __init__(self): # creating Constructor
        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

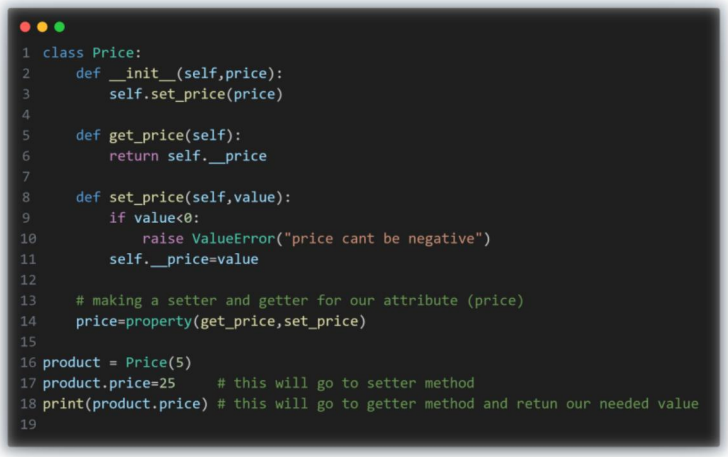
Properties

What is property() Function in Python

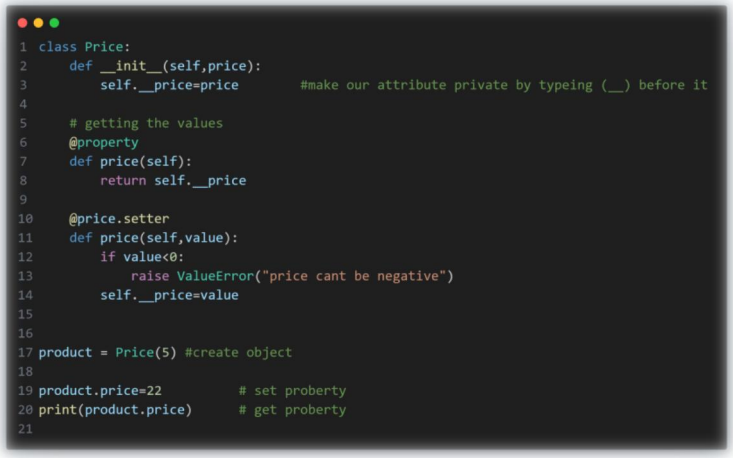
- Python property() function is a built-in function that allows us to create a special type of attribute called a property for a class.
- Properties are used to **encapsulate** the access to an object attribute and to add some logic to the process such as computation, access control, or validation.

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)



```python
class Price:
    def __init__(self,price):
        self.set_price(price)

    def get_price(self):
        return self.__price

    def set_price(self,value):
        if value<0:
            raise ValueError("price cant be negative")
        self.__price=value

    # making a setter and getter for our attribute (price)
    price=property(get_price,set_price)

product = Price(5)
product.price=25     # this will go to setter method
print(product.price) # this will go to getter method and retun our needed value

```

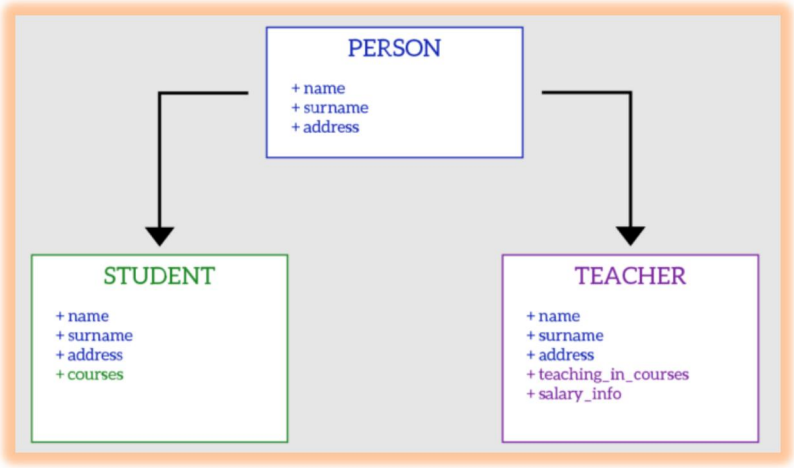


```python
class Price:
    def __init__(self,price):
        self.__price=price        #make our attribute private by typeing (__) before it

    # getting the values
    @property
    def price(self):
        return self.__price

    @price.setter
    def price(self,value):
        if value<0:
            raise ValueError("price cant be negative")
        self.__price=value


product = Price(5) #create object

product.price=22          # set proberty
print(product.price)      # get proberty

```

Inheritance

What is Inheritance used for

- we use it when we have repeated code an 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

```python
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname  = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
    def __init__(self):
       super().__init__("mohamed","ahmed") # Access Parent class (Person)


x = Student() # creating student object will inherit from Person class
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 of 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

```python
class Employee:
    def greet(self):
        print("hi Employee")

class Person:
    def greet(self):
        print("hi Person")

class Manager(Employee,Person):
    pass


mr=Manager()
mr.greet() # will ouput hi Employee

```

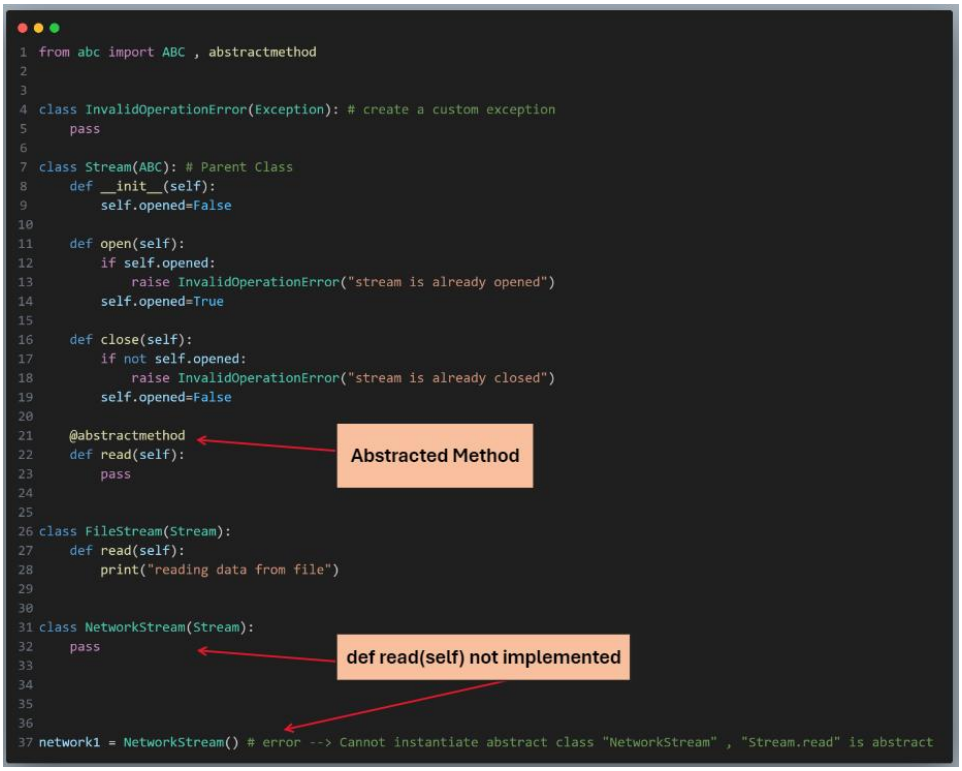
Python Abstraction Base Classes

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



❖ **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

```python
from abc import ABC , abstractmethod

class UIControl(ABC):
    @abstractmethod
    def draw(self):
        pass

class TextBox(UIControl):
    def draw(self):
        print("TextBox")

class DropDownList(UIControl):
    def draw(self):
        print("DropDownList")


# Function to draw multiple controls
def draw(controls):
    for control in controls:
        control.draw()

ddl=DropDownList()
textbox=TextBox()

draw([ddl,textbox]) # pass objects to this function draw as dictionary
```

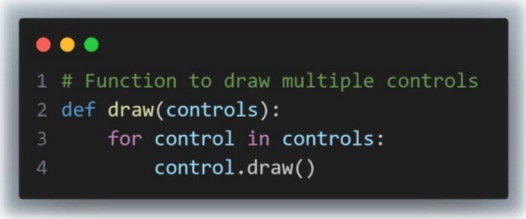
To achieve Polymorphism behaviour

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

Duck Typing in Python

What is it ?

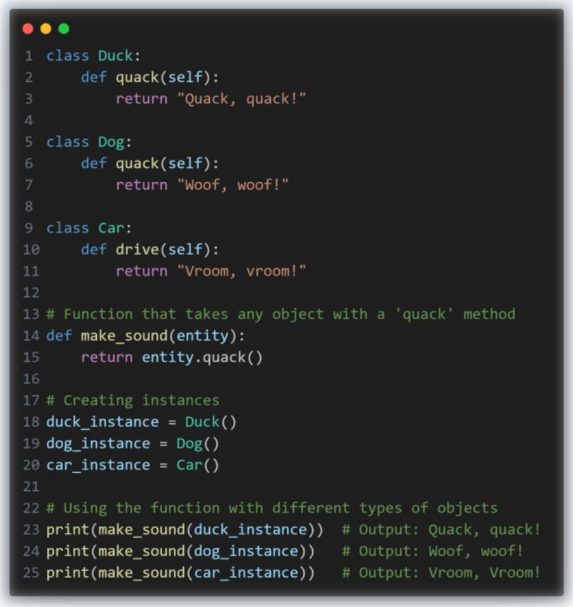
- Duck Typing is a type system used in dynamic languages.
- where the type or the class of an object is less important than the method it defines.
- Using Duck Typing, we do not check types at all. Instead, we check for the presence of a given method or attribute.



```python
# Function to draw multiple controls
def draw(controls):
    for control in controls:
        control.draw()
```

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

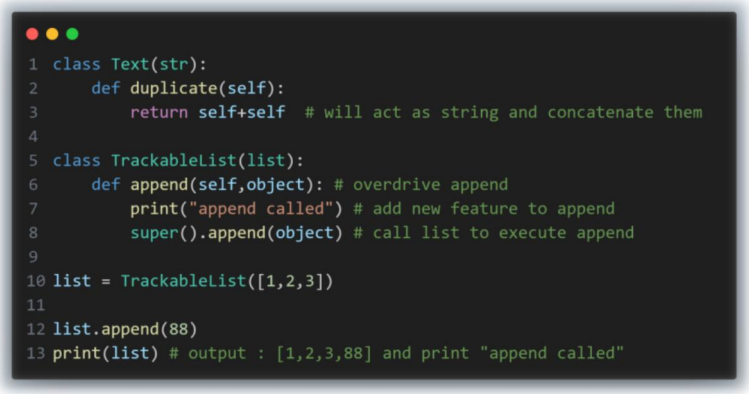
Example 2:



```python
class Duck:
    def quack(self):
        return "Quack, quack!"

class Dog:
    def quack(self):
        return "Woof, woof!"

class Car:
    def drive(self):
        return "Vroom, vroom!"

# Function that takes any object with a 'quack' method
def make_sound(entity):
    return entity.quack()

# Creating instances
duck_instance = Duck()
dog_instance = Dog()
car_instance = Car()

# Using the function with different types of objects
print(make_sound(duck_instance))  # Output: Quack, quack!
print(make_sound(dog_instance))   # Output: Woof, woof!
print(make_sound(car_instance))   # Output: Vroom, Vroom!
```

Extending Built-in Types

Example:



```python
class Text(str):
    def duplicate(self):
        return self+self  # will act as string and concatenate them

class TrackableList(list):
    def append(self,object): # overdrive append
        print("append called") # add new feature to append
        super().append(object) # call list to execute append

list = TrackableList([1,2,3])

list.append(88)
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

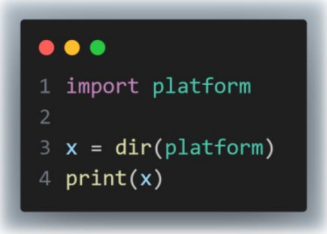
Now we can use the module we just created, by using the `import` statement:

Re-naming a Module

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

Using the dir() Function

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



Import From Module

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

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:

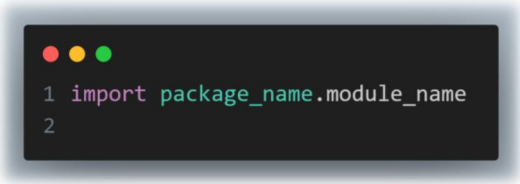
- Create a folder named mypckg.
- Inside this folder create an empty Python file i.e. __init__.py
- Then create two modules mod1 and mod2 in this folder.

Understanding __init__.py

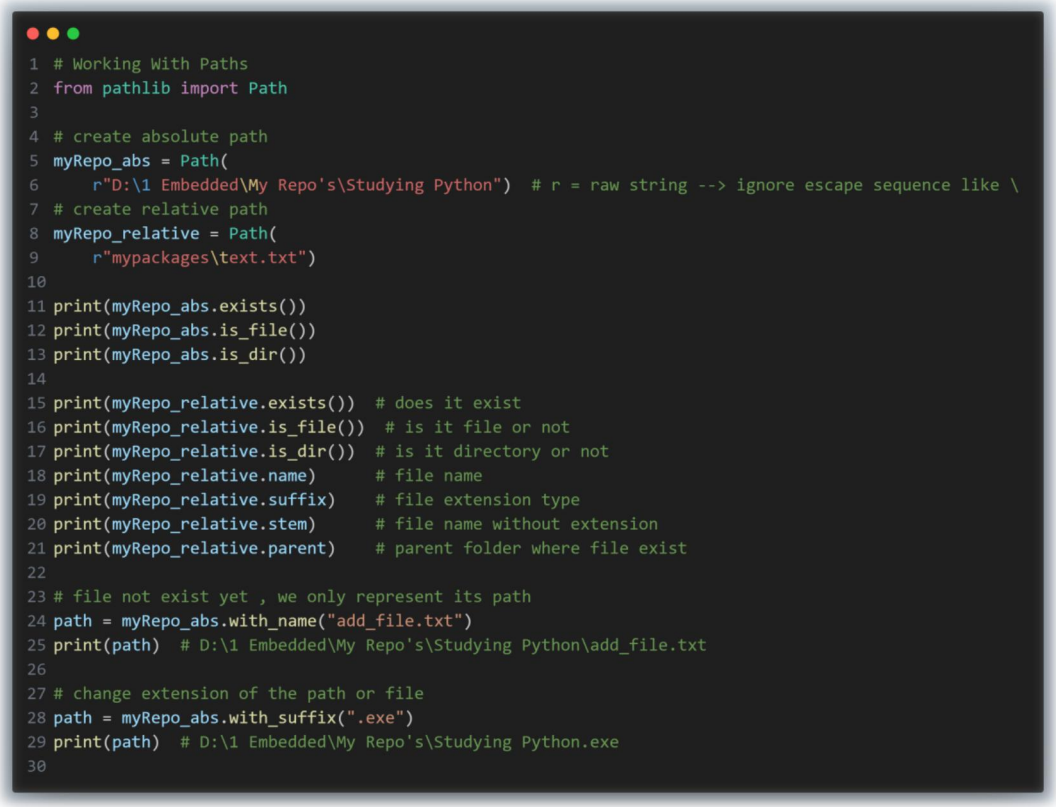
- __init__.py helps the Python interpreter recognize the folder as a package.
- It also specifies the resources to be imported from the modules.
- If the __init__.py is empty this means that all the functions of the modules will be imported.

Import Modules from a Package

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

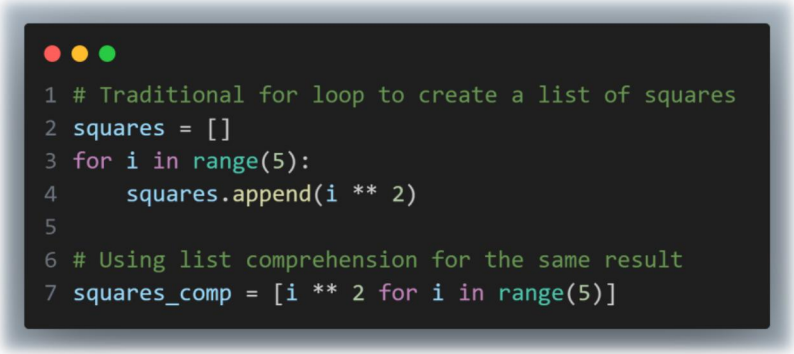


Working With Paths



```python
# Working With Paths
from pathlib import Path

# create absolute path
myRepo_abs = Path(
    r"D:\1 Embedded\My Repo's\Studying Python")  # r = raw string --> ignore escape sequence like \
# create relative path
myRepo_relative = Path(
    r"mypackages\text.txt")

print(myRepo_abs.exists())
print(myRepo_abs.is_file())
print(myRepo_abs.is_dir())

print(myRepo_relative.exists())  # does it exist
print(myRepo_relative.is_file())  # is it file or not
print(myRepo_relative.is_dir())  # is it directory or not
print(myRepo_relative.name)      # file name
print(myRepo_relative.suffix)    # file extension type
print(myRepo_relative.stem)      # file name without extension
print(myRepo_relative.parent)    # parent folder where file exist

# file not exist yet , we only represent its path
path = myRepo_abs.with_name("add_file.txt")
print(path)  # D:\1 Embedded\My Repo's\Studying Python\add_file.txt

# change extension of the path or file
path = myRepo_abs.with_suffix(".exe")
print(path)  # D:\1 Embedded\My Repo's\Studying Python.exe

```

List Comprehension



```python
# Traditional for loop to create a list of squares
squares = []
for i in range(5):
    squares.append(i ** 2)

# Using list comprehension for the same result
squares_comp = [i ** 2 for i in range(5)]
```

Working With Directories

```python
# Working With Paths
from pathlib import Path

# create absolute path
myRepo_abs = Path(
    r"D:\1 Embedded\My Repo's\Studying Python")  # r = raw string --> ignore escape sequence like \

# create relative path
myRepo_relative = Path(r".")  # current diroctory

# print(myRepo_relative.exists())  # does it exist
# print(myRepo_relative.mkdir())   # create this directory
# print(myRepo_relative.rmdir())   # remove this directory
# print(myRepo_relative.name)      # directory name
# print(myRepo_relative.rename)    # rename this directory

# it return the list of files and directories (as generator) but not not recursively
print(myRepo_relative.iterdir())

# iterate through repo folder and print each (folder or file) inside but not recursively
for p in myRepo_relative.iterdir():
    print(p)


paths = [p for p in myRepo_relative.iterdir() if p.is_dir()]
# output: [WindowsPath('.git'), WindowsPath('mypackages')]
print(paths)
############################################################
# Traditional for loop to create a list of squares
squares = []
for i in range(5):
    squares.append(i)
# Using list comprehension for the same result
squares_comp = [i for i in range(5)]
############################################################

# we use glob method to fix iterdir() problems like recursive and search specific files
py_files = [pf for pf in myRepo_relative.glob("*.py")]
print(py_files)

# search recursively
py_files = [pf for pf in myRepo_relative.glob("**/*.py")]
print(py_files)

# search recursively using rglob
py_files = [pf for pf in myRepo_relative.rglob("*.py")]
print(py_files)

```

Working With Files

```python
# Working With Paths
from pathlib import Path
from time import ctime
import shutil

# create absolute path
myRepo_abs = Path(
    r"D:\1 Embedded\My Repo's\Studying Python")  # r = raw string --> ignore escape sequence like \

# create relative path
myRepo_relative = Path(r"mypackages/text.txt")

print(myRepo_relative.exists())                  # does it exist
print(myRepo_relative.rename("__init__.py"))    # rename this file
print(myRepo_relative.unlink())                  # delete this file

# .st_ctime: This accesses the creation time attribute of the file or directory from the stat object
# ctime()  : This function is used to convert a time in seconds to human readable time
# output: Sat Jan  6 12:22:30 2024
print(ctime(myRepo_relative.stat().st_ctime))

# these functions handle opening and closing the file
print(myRepo_relative.read_bytes())  # read binary data
print(myRepo_relative.read_text())

myRepo_relative.write_text("hello file")  # overwrite the text in file


############################################################
# Copying File (not the best option)
############################################################
source = Path(r"mypackages/text.txt")
target = Path(r"mypackages") / "test_file.txt"

target.write_text(source.read_text())


############################################################
# Copying File using shutil
############################################################
source = Path(r"mypackages/text.txt")
target = Path(r"mypackages") / "test_file.txt"

shutil.copy(source, target)

```

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

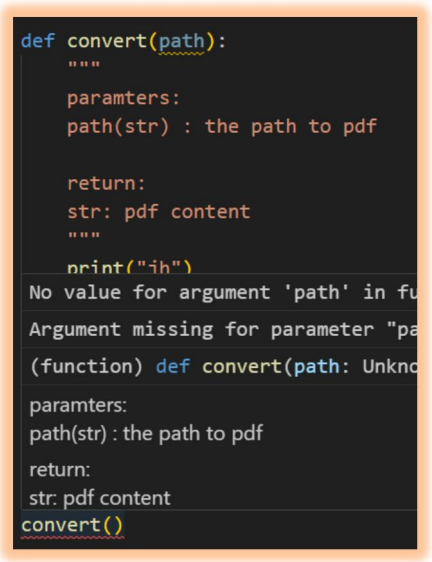
It keep tracks of our files and dependencies , so we can run our script on different machine without worrying about missing dependencies

- 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



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

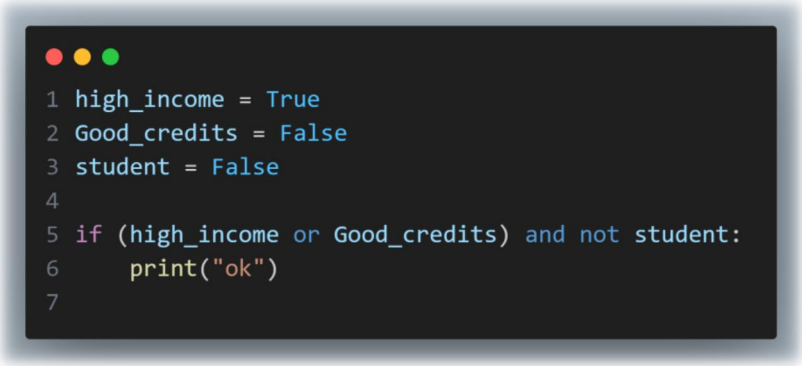
expression_if_true if condition else expression_if_false

Example

```python
# normal comparsion
age = 22
if age > 18:
    message = "ok"
else:
    message = "not ok"
print(message)

# Ternary operator
message = "ok" if age > 18 else "not ok"
print(message)

```

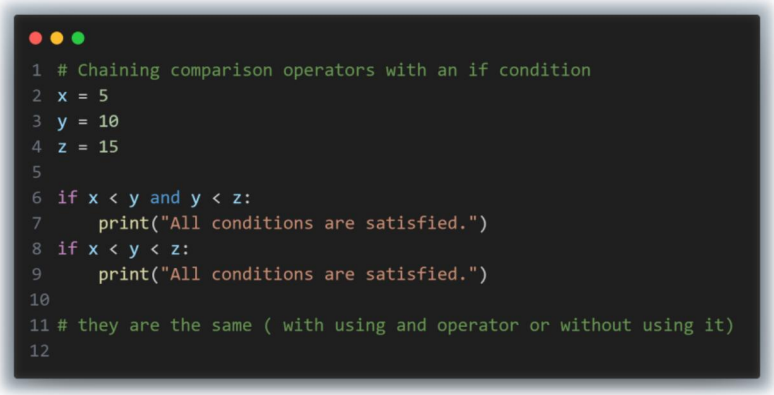
Logical operator

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



```python
high_income = True
Good_credits = False
student = False

if (high_income or Good_credits) and not student:
    print("ok")

```

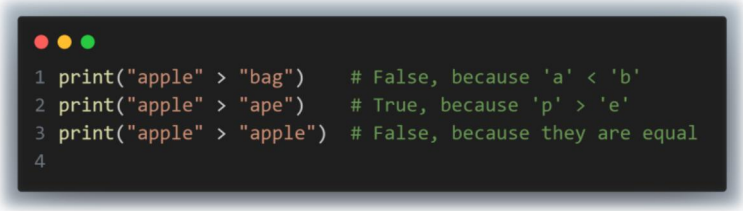
chaining comparison operators



```python
# Chaining comparison operators with an if condition
x = 5
y = 10
z = 15

if x < y and y < z:
    print("All conditions are satisfied.")
if x < y < z:
    print("All conditions are satisfied.")

# they are the same ( with using and operator or without using it)

```

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.



```python
print("apple" > "bag")    # False, because 'a' < 'b'
print("apple" > "ape")    # True, because 'p' > 'e'
print("apple" > "apple")  # False, because they are equal

```

Iterable

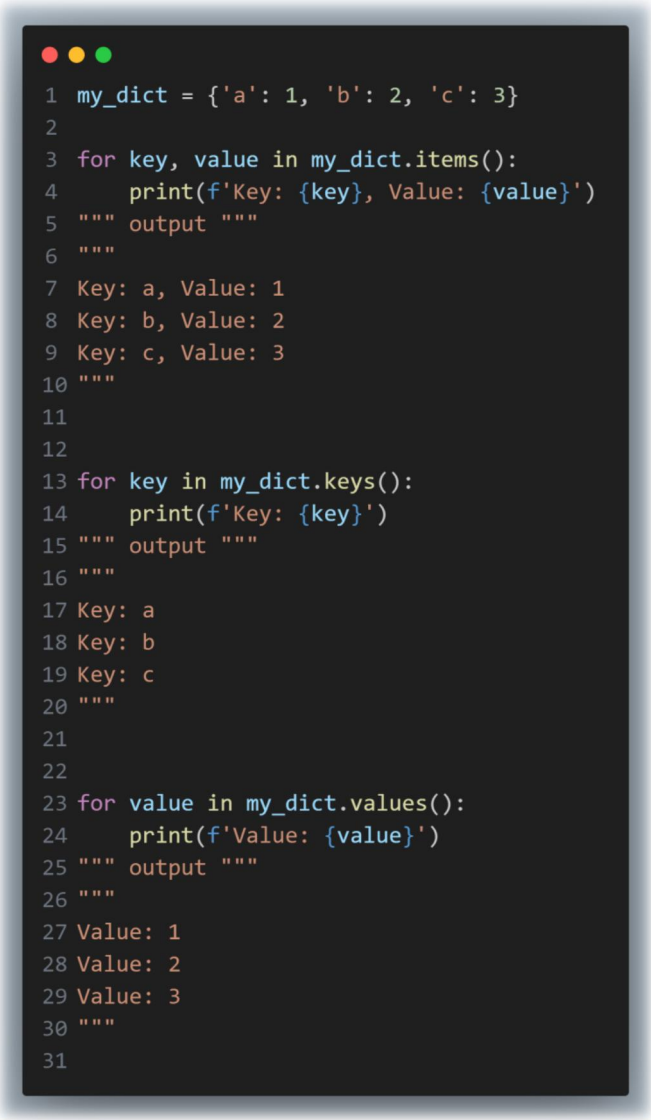
common iterable objects in Python:

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()).

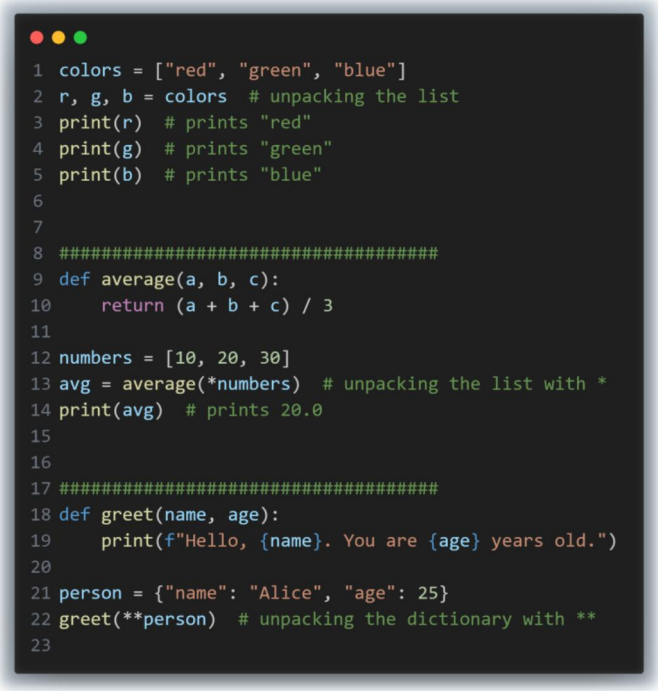


```python
my_dict = {'a': 1, 'b': 2, 'c': 3}

for key, value in my_dict.items():
    print(f'Key: {key}, Value: {value}')
""" output """
"""
Key: a, Value: 1
Key: b, Value: 2
Key: c, Value: 3
"""


for key in my_dict.keys():
    print(f'Key: {key}')
""" output """
"""
Key: a
Key: b
Key: c
"""


for value in my_dict.values():
    print(f'Value: {value}')
""" output """
"""
Value: 1
Value: 2
Value: 3
"""

```

Unpacking

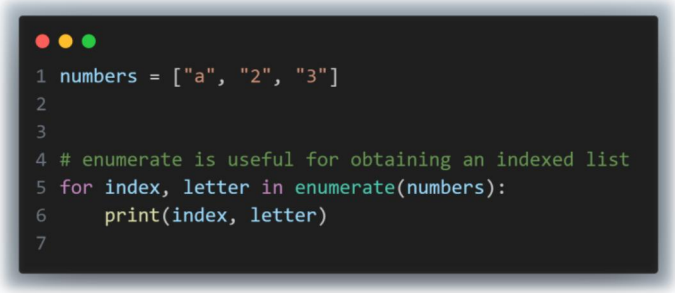
- **(*args)** and **(**kwargs)** are special parameters that can be used to pass a variable number of arguments to a function.

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.



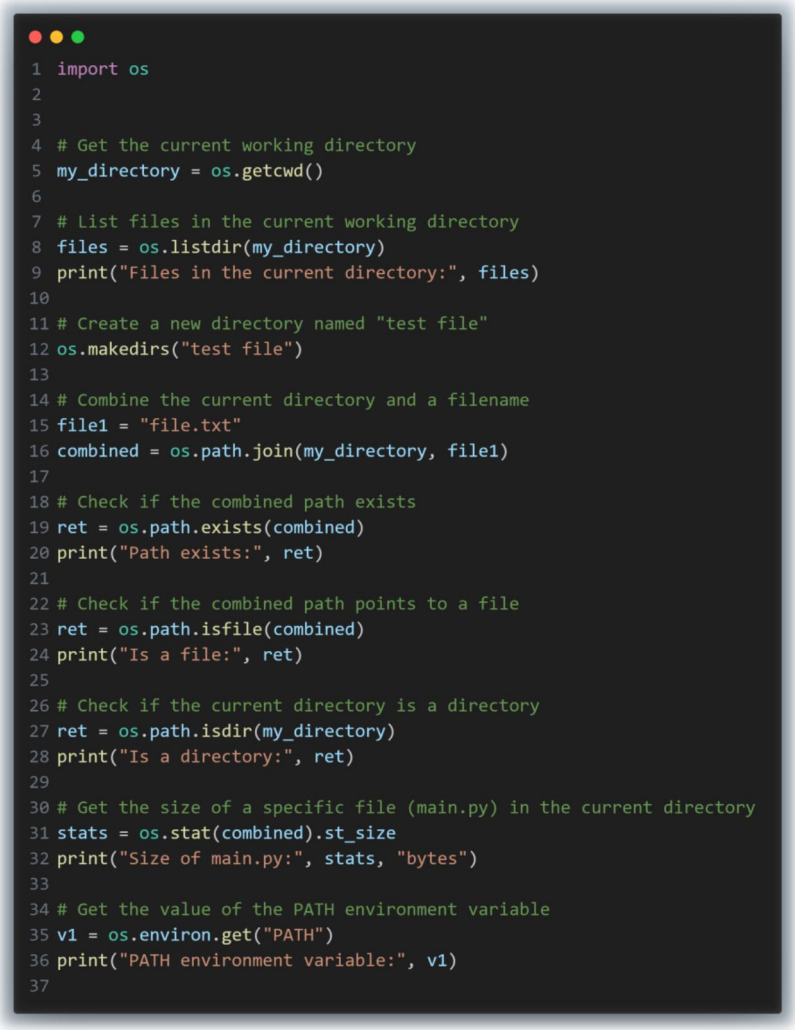
```python
colors = ["red", "green", "blue"]
r, g, b = colors  # unpacking the list
print(r)  # prints "red"
print(g)  # prints "green"
print(b)  # prints "blue"


###################################
def average(a, b, c):
    return (a + b + c) / 3

numbers = [10, 20, 30]
avg = average(*numbers)  # unpacking the list with *
print(avg)  # prints 20.0


###################################
def greet(name, age):
    print(f"Hello, {name}. You are {age} years old.")

person = {"name": "Alice", "age": 25}
greet(**person)  # unpacking the dictionary with **

```

enumerate



```python
numbers = ["a", "2", "3"]


# enumerate is useful for obtaining an indexed list
for index, letter in enumerate(numbers):
    print(index, letter)

```

os module



```python
import os


# Get the current working directory
my_directory = os.getcwd()

# List files in the current working directory
files = os.listdir(my_directory)
print("Files in the current directory:", files)

# Create a new directory named "test file"
os.makedirs("test file")

# Combine the current directory and a filename
file1 = "file.txt"
combined = os.path.join(my_directory, file1)

# Check if the combined path exists
ret = os.path.exists(combined)
print("Path exists:", ret)

# Check if the combined path points to a file
ret = os.path.isfile(combined)
print("Is a file:", ret)

# Check if the current directory is a directory
ret = os.path.isdir(my_directory)
print("Is a directory:", ret)

# Get the size of a specific file (main.py) in the current directory
stats = os.stat(combined).st_size
print("Size of main.py:", stats, "bytes")

# Get the value of the PATH environment variable
v1 = os.environ.get("PATH")
print("PATH environment variable:", v1)

```

Map function

```python
items = [
    ("product2", 25),
    ("product1", 15),
    ("product0", 24),
    ("product3", 22)
]

# Extracting values from tuples using a loop
# Create an empty list to store the second elements of each tuple (numbers)
prices = []

# Loop through each tuple in the 'items' list and append the second element to the 'prices' list
for item in items:
    prices.append(item[1])


# Using the map function to achieve the same result in a more concise way
# Define a lambda function to extract the second element of each tuple
# Use the map function to apply the lambda function to each item in the 'items' list
# Convert the result to a list
prices2 = list(map(lambda item: item[1], items))
```

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

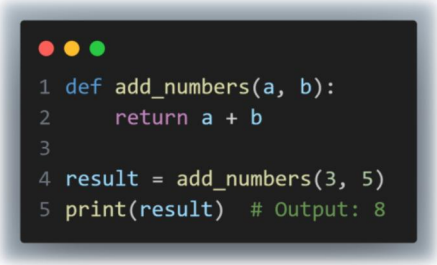
We can see that this lambds is the same as writing this function

```python
def function(item):
    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



```python
def add_numbers(a, b):
    return a + b

result = add_numbers(3, 5)
print(result)  # Output: 8
```

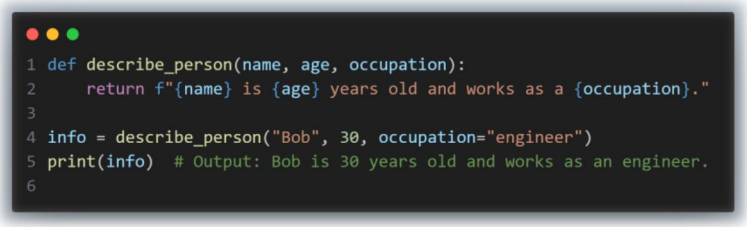
Keyword Arguments:

- 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.

```python
def greet(name, greeting):
    return f"{greeting}, {name}!"

message = greet(name="Alice", greeting="Hello")
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.



```python
def describe_person(name, age, occupation):
    return f"{name} is {age} years old and works as a {occupation}."

info = describe_person("Bob", 30, occupation="engineer")
print(info)  # Output: Bob is 30 years old and works as an engineer.

```

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.

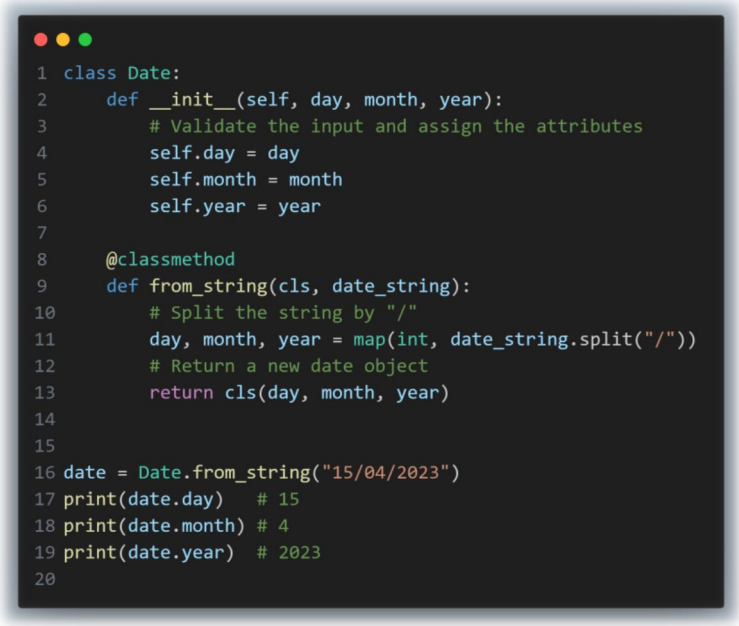
Static Method:

- 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.

```python
# python class method vs static method
class Person:
    person_count = 0 # Class Variable

    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.person_count += 1 # update Class Variable

    @classmethod
    def from_birth_year(cls, name, year):
        age = 2024 - year
        return cls(name, age) # Return a new person object

    @staticmethod
    def is_adult(age):
        return age >= 18

# Create a person object using the constructor
p1 = Person("Alice", 25)

# Create another person object using the class method
p2 = Person.from_birth_year("Bob", 1998)

# you can use static method without creating object
print(Person.is_adult(16)) # False
print(Person.is_adult(21)) # True

```

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 an arguments



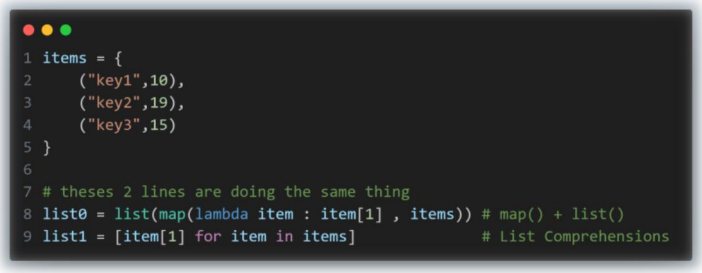
```python
class Date:
    def __init__(self, day, month, year):
        # Validate the input and assign the attributes
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_string):
        # Split the string by "/"
        day, month, year = map(int, date_string.split("/"))
        # Return a new date object
        return cls(day, month, year)


date = Date.from_string("15/04/2023")
print(date.day)   # 15
print(date.month) # 4
print(date.year)  # 2023

```

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]

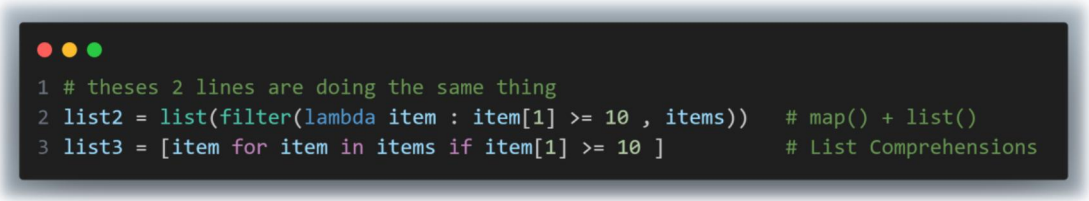
 # Example 1 (map an items)



```
items = {
    ("key1",10),
    ("key2",19),
    ("key3",15)
}

# theses 2 lines are doing the same thing
list0 = list(map(lambda item : item[1] , items)) # map() + list()
list1 = [item[1] for item in items]              # List Comprehensions
```

Its better to use list Comprehensions and less complications

 # Example 2 (filter an items)



```
# theses 2 lines are doing the same thing
list2 = list(filter(lambda item : item[1] >= 10 , items))   # map() + list()
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 :

```
fruits = ["apple", "banana", "cherry"]
colors = ["red", "yellow", "pink"]
prices = [1.25, 0.75, 1.50]

items = list(zip(fruits, colors, prices))
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(...))*.

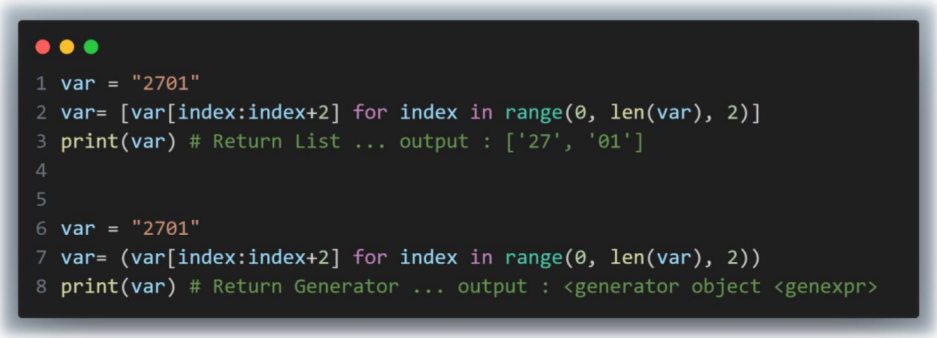
It's often used for:

- 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.



```
var = "2701"
var= [var[index:index+2] for index in range(0, len(var), 2)]
print(var) # Return List ... output : ['27', '01']


var = "2701"
var= (var[index:index+2] for index in range(0, len(var), 2))
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.

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

……