



Python 3



Agenda

- **Python Scopes**
- **Modules in python**
- **File handling in python**
- **Mutability and Immutability**
- **Asynchronous programming**
- **Lab2**



Python Scopes


In Python, a scope refers to a region of a program where a variable or name can be accessed. There are four main scopes in Python:

Local scope:

Local scope refers to variables defined within a function. Variables in the local scope are only accessible within that function.

Local scope

python

 Copy code


```
def my_function():  
    x = 10 # x is a variable in the local scope of the function  
    print(x) # prints 10  
  
my_function()  
print(x) # Raises a NameError, x is not defined in this scope
```

Enclosing scope:

Enclosing scope refers to variables defined in an outer function, which is enclosing the current function. Variables in the enclosing scope are accessible within the inner function.

Enclosing scope

python

 Copy code

```
def outer_function():  
    x = 10 # x is a variable in the local scope of outer_function  
  
    def inner_function():  
        print(x) # x is accessible within inner_function because it is in the enclosing scope  
  
    inner_function()  
  
outer_function()
```

Global scope:

Global scope refers to variables defined outside of any function or class, in the main body of the code. Variables in the global scope are accessible anywhere in the code.

Global scope

python

 Copy code

```
x = 10 # x is a variable in the global scope

def my_function():
    print(x) # x is accessible within the function because it is in the global scope


my_function()
print(x) # prints 10
```

Built-in scope:

Built-in scope refers to the built-in functions and modules that are available in Python by default. Examples of built-in functions include `print()`, `len()`, and `range()`.

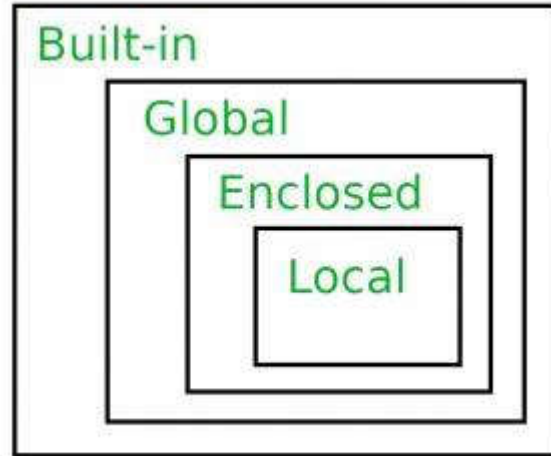
Built-in scope

python

 Copy code

```
print(len([1, 2, 3])) # len is a built-in function that is accessible in any scope
```

Python follows the LEGB (Local, Enclosing, Global, Built-in) rule to determine the scope of a variable or name. This means that Python first looks for the variable in the local scope, then in the enclosing scope, then in the global scope, and finally in the built-in scope.



Understanding the scope of variables in your code is important to avoid naming conflicts and to write clean, organized, and maintainable code.

Modules in python

In Python, a module is a file containing Python definitions and statements. The file name is the module name with the suffix .py. Modules can contain classes, functions, and variables that can be used in other Python programs.

Here are some examples of modules in Python:

math module:

The math module provides a set of mathematical functions that are not built-in in Python. To use this module, you need to import it at the beginning of your code:

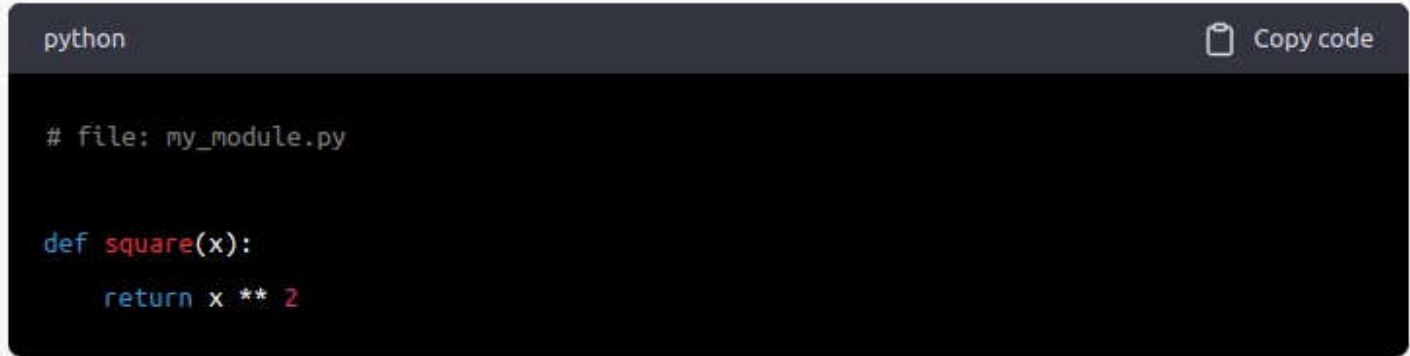
```
python Copy code

import math

# Example usage:
x = math.sqrt(16)
print(x) # Output: 4.0
```

Custom modules in Python work in the same way as built-in modules. They are simply Python files containing definitions and statements that can be imported and used in other Python programs.


To create a custom module, you need to create a new Python file with a .py extension and define the functions or classes that you want to include in the module. For example, let's create a custom module called my_module with a function that returns the square of a number:



```
python Copy code  
  
# file: my_module.py  
  
def square(x):  
    return x ** 2
```

To use this module in another Python program, you need to import it using the import statement. For example:

python

 Copy code


```
import my_module
```

```
result = my_module.square(5)
```

```
print(result) # Output: 25
```

You can also import specific functions or variables from a module using the from keyword. For example:

python

 Copy code

```
from my_module import square
```

```
result = square(5)
```

```
print(result) # Output: 25
```

You can also give a module an alias by using the `as` keyword.
For example:



```
python Copy code  
  
import my_module as mm  
  
result = mm.square(5)  
print(result) # Output: 25
```

That's a basic overview of how custom modules work in Python. By organizing your code into reusable modules, you can make your code more modular, easier to maintain, and easier to share with others.


File handling in python

File handling in Python is a way to interact with files on the file system. It enables us to read, write, append, and manipulate files in different ways. Here are some examples of file handling in Python:

Reading a file

To read a file in Python, you can use the `open()` function to open the file in read mode ('r') and then use the `read()` method to read the contents of the file.

python

 Copy code

```
# opening a file in read mode
file = open("example.txt", "r")

# reading the contents of the file
contents = file.read()


# printing the contents of the file
print(contents)

# closing the file
file.close()
```


Writing to a file

To write to a file in Python, you can use the `open()` function to open the file in write mode ('w') and then use the `write()` method to write the contents to the file.

python

 Copy code

```
# opening a file in write mode
file = open("example.txt", "w")


# writing to the file
file.write("Hello, world!")

# closing the file
file.close()
```

Appending to a file

To append to a file in Python, you can use the `open()` function to open the file in append mode ('a') and then use the `write()` method to append the contents to the file.

python

 Copy code

```
# opening a file in append mode
file = open("example.txt", "a")


# appending to the file
file.write("Hello again, world!")

# closing the file
file.close()
```

Using a context manager

A context manager is a convenient way to handle file I/O in Python, as it automatically handles opening and closing the file. You can use the with statement to create a context manager for a file.

python

 Copy code

```
# using a context manager to open and read a file
with open("example.txt", "r") as file:
    contents = file.read()
    print(contents)

# using a context manager to open and write to a file
with open("example.txt", "w") as file:
    file.write("Hello, world!")

# using a context manager to open and append to a file
with open("example.txt", "a") as file:
    file.write("Hello again, world!")
```

In the examples above, "example.txt" is the name of the file we are reading from, writing to, or appending to. You can replace this with the name of any file you want to work with.

How context manager works?

In Python, a context manager is an object that defines the methods `__enter__()` and `__exit__()`. It is typically used with the `with` statement to automatically set up and clean up resources, such as files or network connections.

When you use the **with** statement with a context manager, Python calls the `__enter__()` method of the context manager object, which sets up the resource and returns it. The resource is then assigned to a variable that you specify in the `with` statement.

After the code inside the with block has executed, Python calls the `_exit_()` method of the context manager object. This method is responsible for cleaning up the resource and handling any exceptions that occurred during the execution of the code.

Here's an example of a context manager that opens a file and automatically closes it when the with block is exited:

```
class File:

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None


    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
```

In the example above, File is a class that defines the `_enter()` and `__exit()` methods. The `__enter()` method opens the file and returns it, while the `__exit__()` method closes the file.

You can use this context manager to open and read a file like this:

python

 Copy code

```
with File('example.txt', 'r') as f:  
    contents = f.read()  
    print(contents)
```

When the with block is entered, Python calls the `_enter_()` method of the File object, which opens the file and returns it. The file object is assigned to the variable `f`.

When the with block is exited, Python calls the `_exit_()` method of the File object, which closes the file.

Using a context manager like this makes your code cleaner and more readable, as it eliminates the need to manually open and close resources. It also ensures that resources are cleaned up properly, even if an exception occurs.

Mutability and Immutability

In Python, mutability and immutability refer to whether an object can be changed or not. Mutable objects can be modified after they are created, while immutable objects cannot.

Here are some examples of mutable and immutable objects in Python:

Mutable objects:

- Lists:

Lists are mutable in Python, which means that you can change their elements after they are created. For example, you can add or remove elements from a list, or modify the values of existing elements.

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list) # Output: [1, 2, 3, 4]

my_list[0] = 5
print(my_list) # Output: [5, 2, 3, 4]
```

- Dictionaries:

Dictionaries are also mutable in Python. You can add, remove, or modify key-value pairs in a dictionary.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
my_dict['d'] = 4
print(my_dict) # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}

my_dict['b'] = 5
print(my_dict) # Output: {'a': 1, 'b': 5, 'c': 3, 'd': 4}
```

Immutable objects:

- Strings:

Strings are immutable in Python, which means that you cannot change their contents after they are created. For example, you cannot modify a single character of a string. Instead, you have to create a new string.

```
my_string = "hello"  
# my_string[0] = "H" # This will cause an error  
new_string = "H" + my_string[1:]  
print(new_string) # Output: "Hello"
```

- Tuples:

Tuples are also immutable in Python. Once you create a tuple, you cannot change its elements.

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 4 # This will cause an error
```

In summary, mutability and immutability refer to whether an object can be changed or not in Python. Mutable objects, such as lists and dictionaries, can be modified after they are created, while immutable objects, such as strings and tuples, cannot be changed.

While mutability and immutability seem straightforward, there are some tricky cases that can cause confusion. Here are some examples:

1 - Mutable objects as elements of immutable objects:

You can have a mutable object, such as a list or dictionary, as an element of an immutable object, such as a tuple. In this case, the mutable object can still be modified even though it is part of an immutable object.

```
my_tuple = ([1, 2], 3, 4)
my_tuple[0].append(5)
print(my_tuple) # Output: ([1, 2, 5], 3, 4)
```

In this example, we create a tuple with a list as its first element. We then append an element to the list, even though the tuple itself is immutable.

2 - Immutable objects as elements of mutable objects:

You can also have an immutable object, such as a string or tuple, as an element of a mutable object, such as a list or dictionary. In this case, the immutable object cannot be modified directly, but it can still be replaced with a new object.

```
my_list = [(1, 2), 3, 4]
my_list[0] = (5, 6)
print(my_list) # Output: [(5, 6), 3, 4]
```

In this example, we create a list with a tuple as its first element. We then replace the tuple with a new tuple, even though the elements of the tuple itself are immutable.

3 - Aliasing:

Aliasing refers to the situation where two variables refer to the same object. In this case, modifying one variable will also modify the other variable, since they both refer to the same object.

```
list1 = [1, 2, 3]
list2 = list1
list2.append(4)
print(list1) # Output: [1, 2, 3, 4]
```

In this example, we create a list and assign it to list1. We then assign list1 to list2, so that both variables refer to the same list object. We then append an element to list2, which also modifies list1 since they both refer to the same object.

To avoid these tricky cases, it is important to be aware of the mutability and immutability of objects in Python, and to be careful when working with mutable objects. In general, it is a good idea to use immutable objects whenever possible, since they are less prone to unexpected modifications.

Functions

In Python, a function is a reusable block of code that performs a specific task. Functions help to break down a program into smaller, more manageable parts and make the code easier to understand and maintain.

A function is defined using the `def` keyword, followed by the function name, and any parameters that the function takes as inputs. The code block inside the function is indented and executed when the function is called.

Here's an example of a simple function that takes two numbers as inputs and returns their sum:

```
python Copy code  
  
def add_numbers(num1, num2):  
    """This function takes two numbers as input and returns their sum"""  
    return num1 + num2
```

In the above code, `add_numbers` is the name of the function, and `num1` and `num2` are the input parameters. The `return` statement specifies the output of the function.

To call this function and get the sum of two numbers, you can use the following code:

```
python Copy code  
  
result = add_numbers(5, 10)  
print(result) # Output: 15
```

In the above code, `add_numbers(5, 10)` calls the `add_numbers` function with 5 and 10 as inputs, and assigns the result to the `result` variable. The `print` statement then outputs the value of `result`, which is 15.

Functions can also have default parameter values, which are used if the function is called without that parameter being explicitly passed. Here's an example:

```
python Copy code  
  
def greet(name, greeting="Hello"):  
    """This function takes a name and a greeting as input and returns a message"""  
    message = f"{greeting}, {name}!"  
    return message
```

In the above code, `greeting="Hello"` sets the default value of the `greeting` parameter to "Hello". If you call the `greet` function without passing a `greeting`, it will use the default value of "Hello".

Here's an example:

```
python Copy code  
  
print(greet("John")) # Output: Hello, John!  
print(greet("Jane", "Hi")) # Output: Hi, Jane!
```

In the first call to `greet`, the `greeting` parameter is not passed, so it uses the default value of `"Hello"`. In the second call, the `greeting` parameter is explicitly passed as `"Hi"`, so it overrides the default value.

Functions can also have variable-length arguments, which allow you to pass any number of arguments to the function. Here's an example:

```
python Copy code  
  
def calculate_average(*args):  
    """This function takes any number of arguments and returns their average"""  
    if len(args) == 0:  
        return 0  
    else:  
        return sum(args) / len(args)
```

In the above code, `*args` is used to specify that the function can take any number of arguments. The if statement checks if the number of arguments is 0, and returns 0 if it is.

The else statement calculates the sum of all the arguments and returns their average.


Here's an example of calling the `calculate_average` function with different numbers of arguments:

```
python Copy code  
  
print(calculate_average()) # Output: 0  
print(calculate_average(5, 10)) # Output: 7.5  
print(calculate_average(1, 2, 3, 4, 5)) # Output: 3.0
```

In the first call to `calculate_average`, no arguments are passed, so it returns 0. In the second call, two arguments are passed (5 and 10),

example of using **kwargs in Python

python

 Copy code

```
def print_details(name, age, **kwargs):  
    """This function takes a name, age, and any number of additional keyword arguments and  
    print(f"Name: {name}")  
    print(f"Age: {age}")  
    for key, value in kwargs.items():  
        print(f"{key.capitalize()}: {value}")  
  
# Call the function with three arguments and one keyword argument  
print_details("John", 25, occupation="Engineer")
```


In the above code, the `print_details` function takes two required arguments (name and age) and any number of additional keyword arguments using `**kwargs`. The function then prints the name and age, followed by any additional keyword arguments.

In the function call `print_details("John", 25, occupation="Engineer")`, name is set to "John", age is set to 25, and the keyword argument `occupation` is set to "Engineer". The function then prints the following output:

```
Name: John
```

```
Age: 25
```

```
Occupation: Engineer
```

You can also pass any number of additional keyword arguments to the function:

```
python Copy code  
  
# Call the function with three arguments and two keyword arguments  
print_details("Jane", 30, occupation="Doctor", location="New York")
```

In the above code, name is set to "Jane", age is set to 30, and two keyword arguments (occupation and location) are passed. The function then prints the following output:

```
Name: Jane  
Age: 30  
Occupation: Doctor  
Location: New York
```

Using `**kwargs` allows you to pass any number of additional keyword arguments to a function, making the function more flexible and versatile.

Asynchronous programming

Asynchronous programming is a technique in Python that allows for non-blocking execution of code, meaning that a program can continue running while waiting for a long-running task to complete. This approach is useful for I/O-bound applications such as network communication, database access, and web scraping, where the program can make progress while waiting for data to be received or processed.

In Python, asynchronous programming can be implemented using the `asyncio` library. The library provides a way to write asynchronous code using coroutines, which are functions that can be paused and resumed during execution. Here is an example of a simple asynchronous program that prints "Hello, world!" after a delay of one second:

```
import asyncio

async def hello():
    print("Hello")
    await asyncio.sleep(1)
    print("world")

loop = asyncio.get_event_loop()
loop.run_until_complete(hello())
```

In this code, the hello function is defined as an asynchronous coroutine using the `async` keyword. The `await asyncio.sleep(1)` line pauses the execution of the coroutine for one second, allowing other tasks to run in the meantime.

Finally, the `loop.run_until_complete(hello())` call runs the coroutine to completion.

Another example of asynchronous programming in Python involves making multiple HTTP requests in parallel. Here is an example program that downloads the HTML of several web pages asynchronously:

```
import asyncio
import aiohttp

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        urls = [
            'https://www.python.org',
            'https://docs.python.org',
            'https://pypi.org',
            'https://github.com'
        ]
        tasks = [asyncio.create_task(fetch(session, url)) for url in urls]
        html_pages = await asyncio.gather(*tasks)
        for html in html_pages:
            print(html[:100])

asyncio.run(main())
```

In this code, the fetch function is defined to download the HTML of a single web page asynchronously using the aiohttp library. The main function creates a ClientSession object and uses it to create tasks for each URL in the urls list. These tasks are then run in parallel using the asyncio.gather function, which returns a list of the HTML pages downloaded. Finally, the program prints the first 100 characters of each HTML page.

In conclusion, asynchronous programming is a powerful technique in Python that allows for non-blocking execution of code. The asyncio library provides a way to write asynchronous code using coroutines, which are functions that can be paused and resumed during execution. Asynchronous programming can be used to speed up I/O-bound applications, such as network communication and web scraping, by allowing multiple tasks to run in parallel.

How asyncio works?

Asyncio is an event-driven, single-threaded concurrency framework that enables non-blocking I/O and cooperative multitasking in Python. It is built on top of Python's event loop, which is a mechanism that handles and schedules the execution of tasks in a single thread.

When an asyncio program starts, it creates an event loop, which is responsible for managing the execution of tasks. Tasks are represented as coroutines, which are functions that can be paused and resumed during execution. When a coroutine encounters an await statement, it suspends its execution and allows the event loop to execute other tasks until the awaited operation is completed. Once the awaited operation is completed, the coroutine resumes its execution from the point where it was paused.

Here's a simplified overview of how asyncio works in the background:

1- An event loop is created: When an asyncio program starts, it creates an event loop using the `asyncio.get_event_loop()` function.

2- Coroutines are defined: The program defines one or more coroutines that represent tasks to be executed. Each coroutine is defined with the `async def` keyword and can contain `await` statements that indicate points at which the coroutine should yield control to the event loop.

3 - Tasks are created: The program creates one or more tasks by wrapping each coroutine in an `asyncio.create_task()` function call. Tasks are added to the event loop, which schedules their execution.

4 - The event loop runs: The event loop continuously iterates over the set of tasks that are waiting to be executed. For each task, the event loop checks whether it is ready to be executed, based on the completion status of any awaited operations. If the task is ready, the event loop resumes its execution.

5 - Awaited operations are executed: When a task encounters an await statement, it suspends its execution and returns control to the event loop. The event loop then looks for another task to execute. If the awaited operation completes while the task is suspended, the event loop resumes the task's execution from the point where it was paused.

6 - Tasks are completed: When a task completes its execution, it is removed from the event loop's set of tasks.

Overall, asyncio enables high-performance, scalable, and non-blocking I/O in Python by allowing multiple tasks to be executed in a single thread. The event loop manages the scheduling and execution of these tasks, allowing the program to handle many I/O-bound operations concurrently with low overhead.

What is the event loop means ?

The event loop is a fundamental concept in asynchronous programming and is a mechanism used to manage and coordinate the execution of asynchronous tasks. In Python's asyncio library, the event loop is the central object that manages the execution of tasks and schedules the execution of coroutines.

In the context of asyncio, the event loop can be thought of as a continuously running loop that waits for events, such as I/O operations, timers, or signals, and dispatches them to the appropriate task or coroutine for handling. When a coroutine encounters an await statement, it suspends its execution and allows the event loop to execute other tasks until the awaited operation is completed. Once the awaited operation is completed, the coroutine resumes its execution from the point where it was paused.

The event loop runs in a single thread and cooperates with the operating system to efficiently schedule and execute tasks.

It is responsible for the following tasks:

- Registering and unregistering I/O event sources, such as sockets or file descriptors

- Waiting for I/O events to occur
- Dispatching events to the appropriate callback or coroutine
- Managing timeouts and delayed calls
- Handling operating system signals

When a program using asyncio starts, it creates an event loop object using the `asyncio.get_event_loop()` function. The program then adds tasks to the event loop using the `asyncio.create_task()` function. The event loop schedules the execution of these tasks and manages their execution using a cooperative multitasking approach.

Overall, the event loop is the core of the asyncio library and is responsible for the efficient and coordinated execution of asynchronous tasks. It provides a single-threaded, non-blocking, and highly scalable concurrency model that is suitable for I/O-bound applications.

Lab2:

- 1 - Write a Python program that uses the math module to calculate the area of a circle with a radius of 5.
- 2 - Create a custom Python module called my_module with a function that takes a string as input and returns the reverse of the string. Then write a Python program that imports the my_module module and uses the reverse_string function to reverse the string "Hello, world!".
- 3- Write a Python program that imports only the randint function from the random module and uses it to generate a random integer between 1 and 10.

4 - Write a Python program that imports the datetime module with the alias dt and uses the now function to get the current date and time.

5- Write a Python program that reads a file named example.txt and counts the number of lines in the file.

6 - Write a Python program that reads a file named example.txt and prints the contents of the file in reverse order.

7 - Write a Python program that reads a file named example.txt and removes all newline characters from the file.

8 - Write a Python program that reads a file named example.txt and writes its contents to a new file named copy. txt.

- 9 - Write a Python program that reads a file named example.txt and prints the longest word in the file.
- 10 - Write a function that takes a list of integers as input and returns the sum of all even numbers in the list.
- 11 - Write a function that takes a list of strings as input and returns a new list containing only the strings that start with the letter "a".
- 12 - Write a function that takes a dictionary as input and returns a new dictionary with the keys and values swapped (i.e., the keys become the values and the values become the keys).
- 13 - Write a function that takes a list of numbers as input and returns the largest and smallest numbers in the list.
- 14 - Write a function that takes a list of numbers as input and returns a new list containing the squares of each number in the input list.

15 - Write a function that takes an arbitrary number of lists as input using `*args` and returns a new list that contains all the elements from all the input lists.

16 - Write a function that takes a string as input and an arbitrary number of keyword arguments using `**kwargs`. The function should replace all instances of the keyword argument keys in the input string with their corresponding values.

Async is advanced topic that needs good understanding [optional quiz you are not required to solve it]

17 - Write an async function that sleeps for a specified number of seconds and then prints a message to the console.

18 - Implement a Python program that downloads a single webpage using `aiohttp` and prints the HTTP status code to the console.

19 - Write an async function that computes the sum of two numbers and returns the result after a simulated delay using `asyncio.sleep()`.

20 - Implement a program that reads a list of URLs from a file, and downloads the contents of each URL in parallel using `asyncio`. Print the number of downloaded pages to the console.

21 - Write an async function that reads a file in chunks and returns the contents of the file as a string. Use `asyncio.sleep()` to simulate I/O delays between each chunk.

22 - Implement a program that spawns a new subprocess using `asyncio.create_subprocess_exec()` and waits for it to complete. Print the subprocess exit code to the console.

23 - Write an async function that generates a sequence of integers from 1 to 10 using `asyncio.sleep()` to introduce a delay between each number. Print each number to the console as it is generated.

24 - Implement a program that listens for incoming connections on a specified port using asyncio. When a connection is received, send a static message to the client and close the connection.

25 - Write an async function that reads data from a websocket using the websockets library and prints it to the console. Use `asyncio.sleep()` to introduce a delay between each received message.

26 - Implement a program that downloads multiple files in parallel using asyncio. Use the `asyncio.Queue` class to manage a pool of worker tasks and the `aiohttp` library to download files. Print the name of each downloaded file to the console.

Thank you

Hesham Sayed

E-mail: hesham.sayed636@gmail.com

