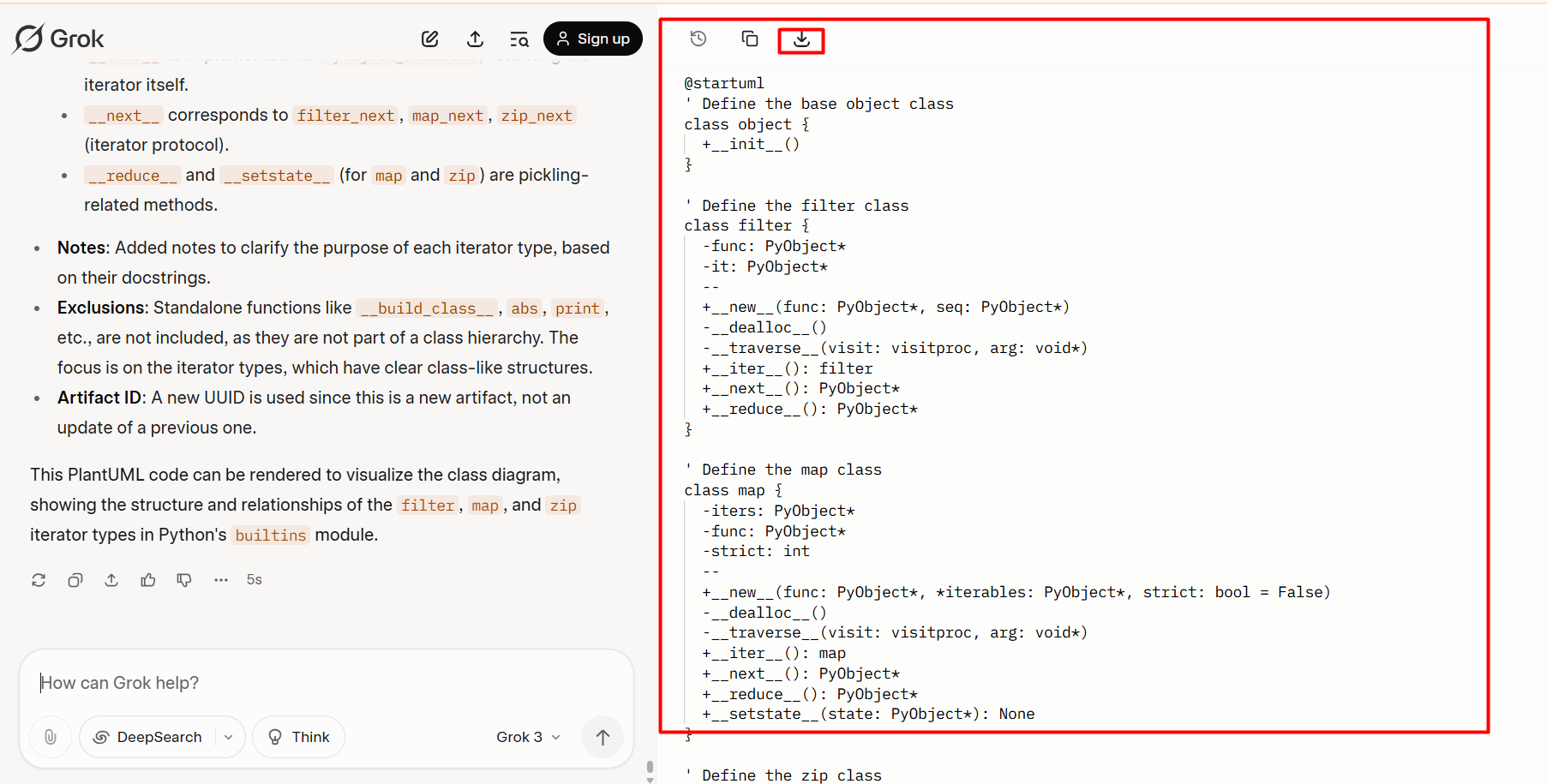
**LECTURE-07 (Saturday 24-May-2025)**

* Polymorphism: It is one of the four core principles of Object-Oriented Programming (OOP), along with “Encapsulation”, “Inheritance” and “Abstraction”. The word "polymorphism" comes from Greek, meaning "many forms." There are two types of polymorphism:

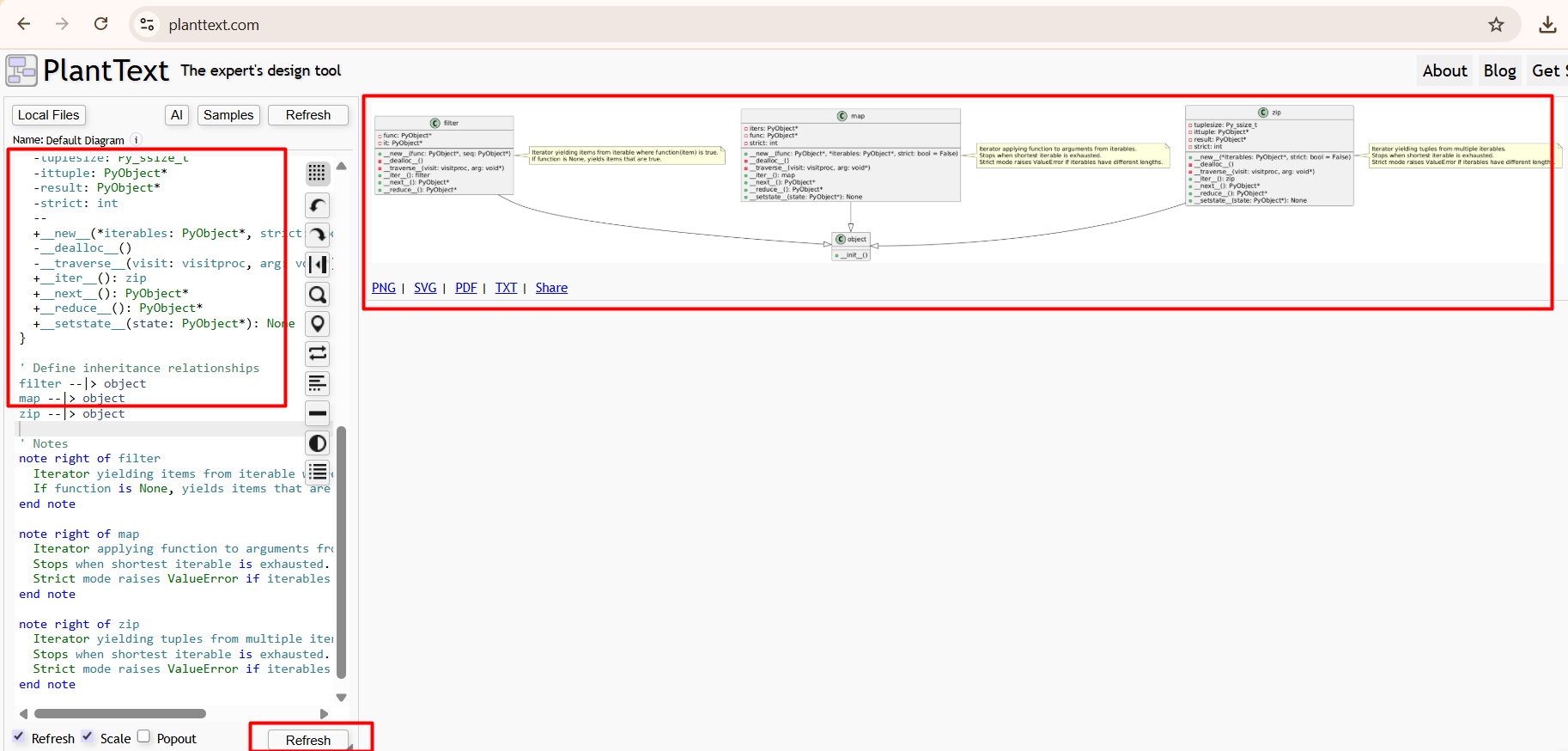
1. Compile-Time Polymorphism (Static Polymorphism): “Method Overloading” achieves this type of polymorphism as defining multiple functions/methods with the same name but different parameters in the same class.
2. Run-Time Polymorphism (Dynamic Polymorphism): “Method Overriding” achieves this type of polymorphism as the base/parent class defines a function/method, and the subclass/child/inherited class provides a specific implementation. It is usually implemented through virtual methods or interfaces.

* We will study Polymorphism in later classes.
* “builtins.py” is a generic class of python which contains default/built-in/generic methods of python.
* Source code of “builtins.py” 🡪 <https://github.com/python/cpython/blob/main/Python/bltinmodule.c>
* To get UML diagram of any code:

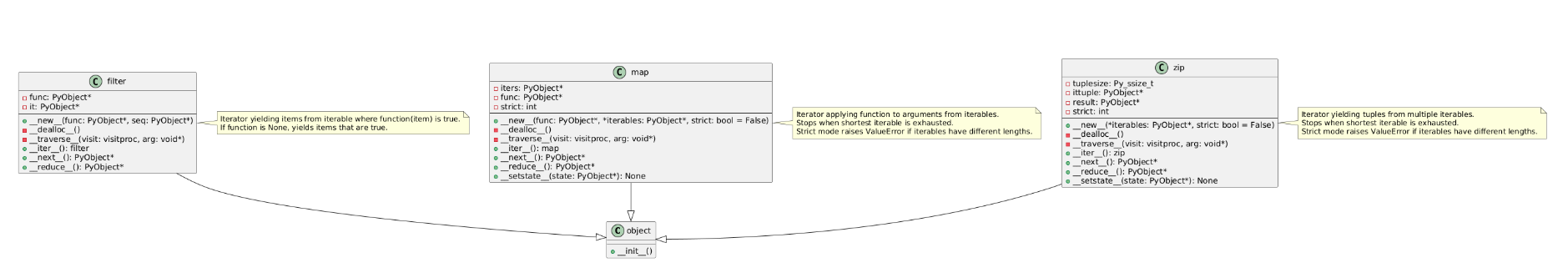
1. Copy complete code from <https://github.com/python/cpython/blob/main/Python/bltinmodule.c>
2. Open <https://grok.com/> because in <https://chatgpt.com/> , complete code cannot be pasted which is copied in step 1 as code is too big for ChatGPT message.
3. In grok, write prompt “generate UML diagram from below code:” and in newline paste code which is copied in step 1
4. We get below output in file, download this file.



1. Open downloaded file “UML Diagram for Python Builtins Iterator Types” and copy text present inside file
2. Open <https://planttext.com/> and paste UML diagram file’s copied data and press “Refresh” button and you will get UML diagram in output



UML Diagram:

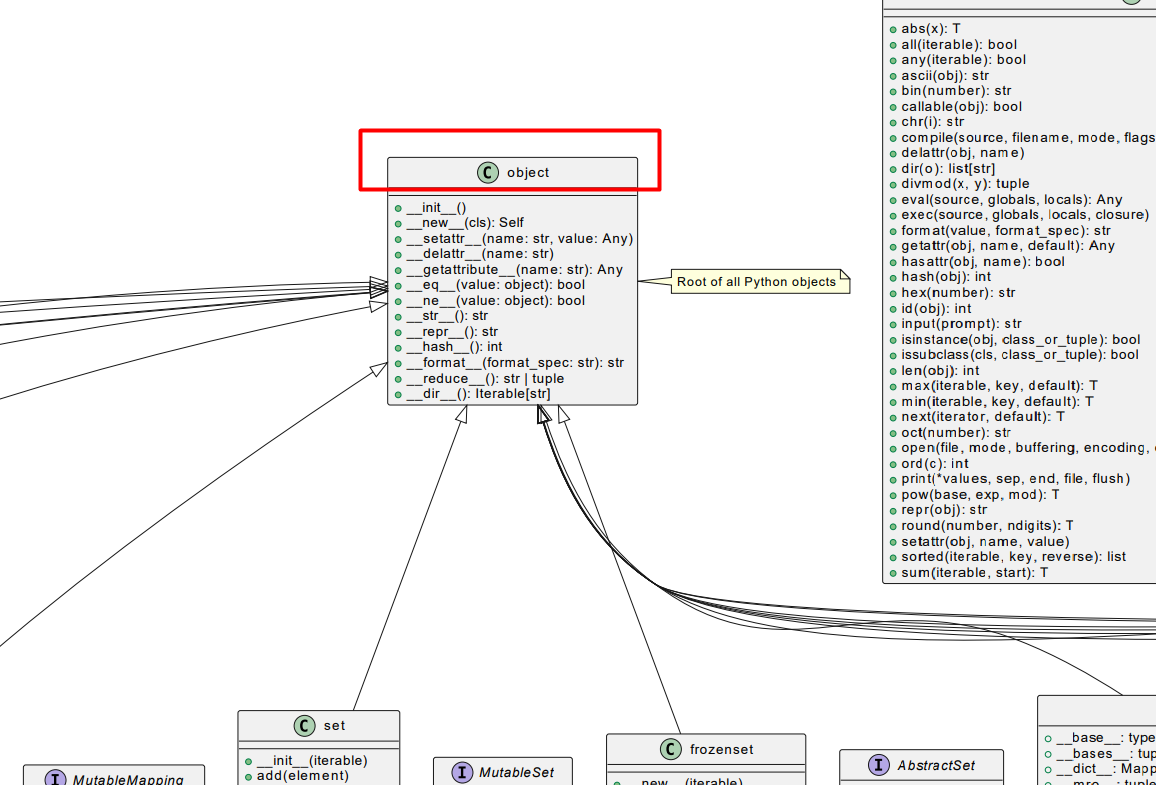


1. You can save UML diagram in all available formats like PDF, PNG SVG, TXT as shown in above image or you can share link of generated UML diagram link:

<https://planttext.com?text=pLLDRzH03BtdL_X6j-po0rO2zG18N0eI4jTeDd4sHfEPC7RQRWl_7NkI_KefI0W1boIsupVllNWcLomkoT3vuXcyneO2WhG86yS8SVCPAu7AE-OYN_UfXmBWULbI82dBYqNnhJXlRyWBfhFEAMMTgsO8rHe-xDvdj4jBaSmIgt69WBUwmgnX2OnVJkA5DPHbZSxxM6L6EI79tM1YrCmDiIsHRtsArH9SsceSgSxjfaRvMUzwuhedS2TZzfIU5XBMGuNpqjo9plLdDbZyC0fMPZFDF_259L6bn2d8hrsvD52tySZdPaqOcnWzl8ItpZF-QSjKrs_vbKkCecCeLiptqq-xXkiOy96lztJkgyNPLnbwrK_t6QHaUofbD3oNPlucvC7FP-zdflz3ZrNZN_EOGek9n8KA8Q5tGZ5mIpqNqqPUhRw-cew3meRwDBQFSHehx7KKv2Be5H9jMu7O7BVPExDKOe8TeQyfR455TtfcfDZ1tc-uLKPwjEZG6vaBUsK1n21fm1U6qXoAbZP9on6H9pnfdQZ1EBLWgC4O5JDUuomVIBc-zpiZTK2NQ9zfw338Z2GRaOyIUpQw0RYDIONBKOKomxlM3InOZo_Ru40NQsNWY97Xa_C3lab9rwRc20sjZWtKr3JgGn3m6BRIyjCwndbvREwuHKRYdKu_zSRh_r1mfO_wD_eE>

* UML diagram is a road map. To learn new things, you can use UML diagram. If any new code comes then for understanding, generate road map by UML diagram
* Python UML diagram : <https://github.com/panaversity/learn-modern-ai-python/blob/main/python-all-types-and-methods.pdf>

In UML diagram, where arrow goes, it means parent class:



Here, everything belongs to “object” class means “object” class is the parent class of all generic/built-in classes.

* Return function: It return values and returned value can be stored in variable.

Example:

abc = len("def")

print(abc)

Output:

3

* Non-Return function: It do not return any value.

Example:

abc = print("def")

print(abc)

Output:

def

None

* “list” datatype is an immutable means any element which is present in list, can be changed at any time.
* Default function: It do not requires any parameter for execution.

Example:

def default\_function():

  print("Default function executed.")

default\_function()

Output:

Default function executed.

* Required Parameters function: It do not requires any parameter for execution.

Example:

def required\_parameter\_function(a : int, b : int):

  print("Required Parameter function executed.")

required\_parameter\_function(10,20)

Output:

Required Parameter function executed.

* Revision of Lecture-06 functions
* With Block function: In Python, a "with block" refers to using “with” statement to wrap the execution of a block of code with methods defined by a context manager.
* A context manager in Python is an object that properly manages resources when entering and exiting a block of code, usually with the “with” statement. It ensures that setup and cleanup operations are finished automatically and safely even if exceptions are raised. It is useful for:

1. Opening/closing files
2. Acquiring/releasing locks
3. Managing database connections
4. Starting/stopping timers or logging

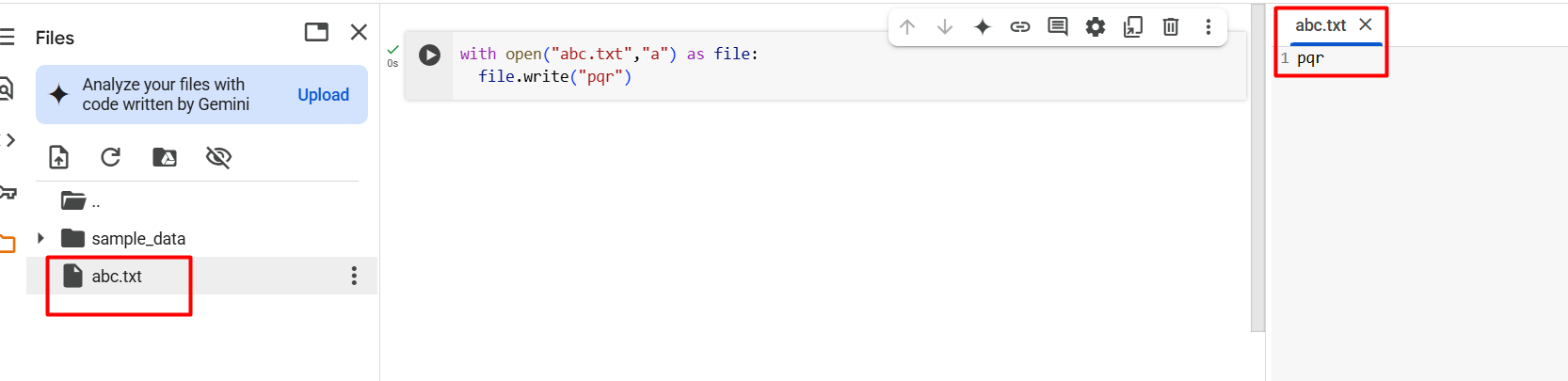
* Example of “With Block”:

Example:

with open("abc.txt","a") as file:

  file.write("hi")

Output:



Here, when code executed then file “abc.txt” generated which contains text “pqr”.

Breakdown:

1. **“with open("abc.txt", "a") as file:”:** This uses a context manager (with statement), which ensures the file is properly closed after the block is executed even if an error occurs.
2. **"abc.txt"** is the name of the file.
3. **"a"** stands for append mode. It means:

a) If the file does not exist then new file will created.

b) If file already exists then new content will added at the end of the file, without deleting the existing content.

1. **“file”** is a variable that represents the opened file.
2. **“file.write("hi")”:** This writes the string "hi" to the file. This uses a context manager (with statement), which ensures the file closed after the block executed even if an error occurs.

* Prompt : create simple code example for custom with block

Output Code:

# prompt: create simple code example for custom with block

class MyCustomWithBlock:

  def \_\_enter\_\_(self):

    print("Entering the 'with' block")

    return self  # Return the object itself or another object if needed

  def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

    print("Exiting the 'with' block")

    # exc\_type, exc\_value, and traceback are non-None if an exception occurred

    if exc\_type:

      print(f"An exception of type {exc\_type} occurred: {exc\_value}")

    return False  # Return True if the exception was handled, False otherwise

# Example usage

with MyCustomWithBlock() as my\_object:

  print("Inside the 'with' block")

  # You can access my\_object here if \_\_enter\_\_ returned something other than None

print("Outside the 'with' block")

# Example with an exception

try:

  with MyCustomWithBlock():

    print("Inside the 'with' block (with exception)")

    raise ValueError("Something went wrong!")

except ValueError:

  print("Caught the exception outside the 'with' block")

Output:

Entering the 'with' block

Inside the 'with' block

Exiting the 'with' block

Outside the 'with' block

Entering the 'with' block

Inside the 'with' block (with exception)

Exiting the 'with' block

An exception of type <class 'ValueError'> occurred: Something went wrong!

Caught the exception outside the 'with' block

Here,

“MyCustomWithBlock”: custom context manager class

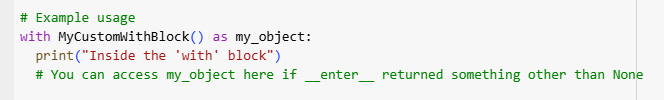
“\_\_enter\_\_(self)”: This method is called when the “with” statement is entered.

“returns self”: “self” is the instance/object of the class “MyCustomWithBlock”. This returned value can be assigned to the variable after “as” in the “with” statement (e.g., “my\_object”).

“\_\_exit\_\_(self, exc\_type, exc\_value, traceback)”: This method is called when the `with` block is exited, whether normally or due to an exception. “exc\_type”, “exc\_value”, and “traceback” are arguments that provide information about an exception if occurred within the “with” block. If no exception occurred then value of “exc\_type”, “exc\_value”, and “traceback” is “None”. If “exc\_type” contains some value, it means an exception occurred. The method returns “False“ but If this method returns “True” then it indicates that the exception was handled within the “\_\_exit\_\_” method, and the exception is suppressed (not re-raised) outside the “with” block. Returning “False” (in above code) means the exception is not handled by the “\_\_exit\_\_” method and will be re-raised outside the “with” block.

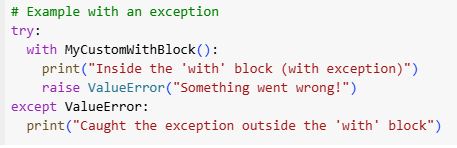
“MyCustomWithBlock” class is used twice in above code:

1) First,



Here, “with” block executes without error. “\_\_enter\_\_” method called then the code inside the “with” block runs and finally “\_\_exit\_\_” method called.

2) Second,



Here, “with” block is wrapped in a “try-except” (try-catch in other programming languages) block.

Inside the “with” block, s “ValueError” exception is raised.

The “\_\_exit\_\_” method called when the exception occurred. Since “\_\_exit\_\_” returns “False” which means the exception is not handled and is re-raised outside the “with” block, where exception is caught by the “except ValueError:” clause.

* Prompt: create simple code example for custom with block using function

Output Code:

# prompt: create simple code example for custom with block using function

import sys

from contextlib import contextmanager

@contextmanager

def my\_context():

  print("Entering the context...")

  try:

    yield "This is the yielded value"

  except Exception as e:

    print(f"An error occurred: {e}")

  finally:

    print("Exiting the context...")

# Example usage of the custom with block

with my\_context() as value:

  print(f"Inside the context. Value: {value}")

  # You can raise an exception here to see the exception handling

  # raise ValueError("Something went wrong!")

print("Outside the context.")

Output:

Entering the context...

Inside the context. Value: This is the yielded value

Exiting the context...

Outside the context.

Here,

“@contextmanager”: It declares a custom context manager using the contextlib.contextmanager decorator, which allows you to create a context manager using a simple generator function.

“my\_context” function is a generator function.

When the “with” statement is entered, the code before “yield” is executed "Entering the context...".

The “yield” statement pauses the generator and returns a value "This is the yielded value" to be assigned to the variable “value” in the “with” statement “with my\_context() as value:”

The code inside the “with” block is then executed.

If an exception occurs within the “with” block then the generator function’s execution resumes at the “except” block, handling the error.

Regardless of whether an exception occurred or not, the code in the “finally” block is executed when the “with” block is exited.

* Context Manager provides a clean way to manage resources (like file operations, network connections, etc.) ensuring that cleanup actions (like closing files) are performed even if errors occur.
* Synchronous function: focused on communicating with one person at a time. It is blocking that one task must finish entirely before the next one starts. By default, we create sync function.
* Asynchronous function: focused on communicating with multiple persons at a time. It is non-blocking (when using “await”) that allows the program to do other things while waiting for an operation to complete. Async function runs code parallel.
* Prompt : create simple example of sync and async function

Output Code:

# prompt: create simple example of sync and async function

import time

import asyncio

def sync\_function():

    print("Starting synchronous function")

    time.sleep(10)  # Simulate a blocking operation

    print("Synchronous function finished")

async def async\_function():

    print("Starting asynchronous function")

    await asyncio.sleep(10)  # Simulate an awaitable operation

    print("Asynchronous function finished")

# Running the synchronous function

print("Calling synchronous function:")

sync\_function()

print("Synchronous call finished")

# Running the asynchronous function (requires an event loop)

print("\nCalling asynchronous function:")

asyncio.run(async\_function())

print("Asynchronous call finished")

# Example showing how async can potentially run concurrently (brief demonstration)

async def main():

    print("\nCalling async functions concurrently (brief demonstration):")

    task1 = asyncio.create\_task(async\_function())

    task2 = asyncio.create\_task(async\_function())

    await task1

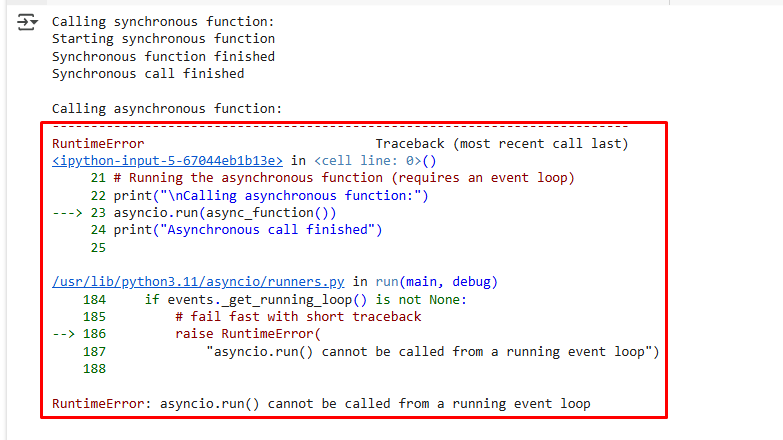
    await task2

asyncio.run(main())

print("Concurrent async calls finished")

Output:

When run above code at Google Colab then at line “asyncio.run(async\_function())” when calling async function.



While same code executed success fully at VS Code.

**Output at VSCode:**

Calling synchronous function:

Starting synchronous function

Synchronous function finished

Synchronous call finished

Calling asynchronous function:

Starting asynchronous function

Asynchronous function finished

Asynchronous call finished

Calling async functions concurrently (brief demonstration):

Starting asynchronous function

Starting asynchronous function

Asynchronous function finished

Asynchronous function finished

Concurrent async calls finished

---------------------------------------------------

Here,

The main difference between the “sync\_function” and “async\_function” in above provided code lies in how they handle operations that take time (like waiting for I/O, simulating with “time.sleep” or “asyncio.sleep”).

“sync\_function”: It executes operations sequentially, one after another. When “time.sleep(10)” is called, the entire program pauses and waits for those 10 seconds to pass before proceeding to the next line. While the synchronous function is "sleeping" or performing a blocking operation, no other code can run. The program effectively frozen during that time.

In above code, “sync\_function()” is called and the line “print("Synchronous call finished")” is only executed after the “sync\_function” has completely finished.

“async\_function”: It uses “async” keyword before function name in function declaration and uses “await” keyword before task in function body. It doesn't pause the entire program when it encounters an “await” operation like “asyncio.sleep(10)”. Instead, when “await asyncio.sleep(2)” is called, the “async\_function” yields control back to the event loop. While the “async\_function” is waiting, the event loop can switch to and execute other tasks. This allows other asynchronous functions or parts of the program to run concurrently.

“asyncio.run()“ is needed to start the asynchronous event loop, which manages the execution of asynchronous functions.

In the above code, “asyncio.run(main())” is the “main” function creates two tasks for “async\_function” because “asyncio.sleep” is awaitable and the event loop can switch between the two tasks while they are "sleeping", potentially allowing them to overlap their waiting periods and finish sooner than if they were run purely sequentially in a synchronous manner. This demonstrated by the "Concurrent async calls finished" output appearing before "Starting asynchronous function" messages.

* Library “asyncio” is required for async operations.
* Database calls is async
* Time difference between sync and async methods code execution

**Async Code:**

import time

import asyncio

async def async\_function(sec : int):

    print(f"Starting asynchronous function for time interval : {sec} ")

    await asyncio.sleep(sec)  # Simulate an awaitable operation

    print(f"Asynchronous function finished for time interval : {sec} ")

async\_start\_time = time.time()

asyncio.run(async\_function(5))

asyncio.run(async\_function(7))

asyncio.run(async\_function(4))

asyncio.run(async\_function(8))

asyncio.run(async\_function(9))

async\_end\_time = time.time()

print(f"Async time difference: {async\_end\_time - async\_start\_time} seconds")

**Output:**

Starting asynchronous function for time interval : 5

Asynchronous function finished for time interval : 5

Starting asynchronous function for time interval : 7

Asynchronous function finished for time interval : 7

Starting asynchronous function for time interval : 4

Asynchronous function finished for time interval : 4

Starting asynchronous function for time interval : 8

Asynchronous function finished for time interval : 8

Starting asynchronous function for time interval : 9

Asynchronous function finished for time interval : 9

Async time difference: 33.04879808425903 seconds

**Sync Code:**

import time

import asyncio

def sync\_function(sec : int):

    print(f"Starting synchronous function for time interval : {sec} ")

    time.sleep(sec)  # Simulate a blocking operation

    print(f"Synchronous function finished for time interval : {sec} ")

sync\_start\_time = time.time()

sync\_function(5)

sync\_function(7)

sync\_function(4)

sync\_function(8)

sync\_function(9)

sync\_end\_time = time.time()

print(f"Sync time difference: {sync\_end\_time - sync\_start\_time} seconds")

**Output:**

Starting synchronous function for time interval : 5

Synchronous function finished for time interval : 5

Starting synchronous function for time interval : 7

Synchronous function finished for time interval : 7

Starting synchronous function for time interval : 4

Synchronous function finished for time interval : 4

Starting synchronous function for time interval : 8

Synchronous function finished for time interval : 8

Starting synchronous function for time interval : 9

Synchronous function finished for time interval : 9

Sync time difference: 33.051204204559326 seconds

* Identity operators: In Python, identity operators are used to compare the memory locations of two objects to determine if they are the same object, not just equal in value. Identity is not the same as equality (==). Use equality (==) to check for equality of values while “is” to check for identity (same object). There are two identity operators:

1. “is”: It returns “True” if both variables point to the same object (i.e., same memory location).

Example:

a = [1, 2, 3]

b = a

print(a is b)

Ouput:

True

Here, “True” because both “a” and “b” refer to the same object (same memory location)

1. “is not”: It returns “True” if both variables do not point to the same object (i.e., same memory location).

Example:

a = [1, 2, 3]

b = [1, 2, 3]

print(a is not b)

Output:

True

Here, “True” because “a” and “b” are two separate objects with the same contents.