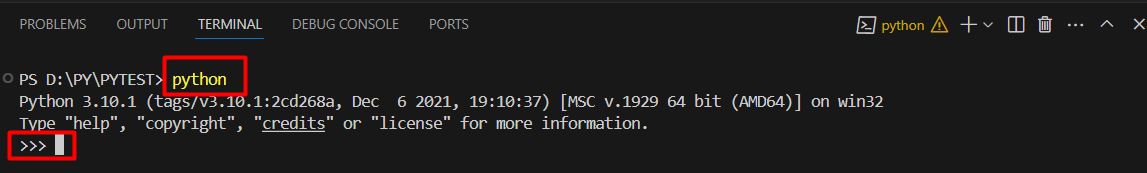
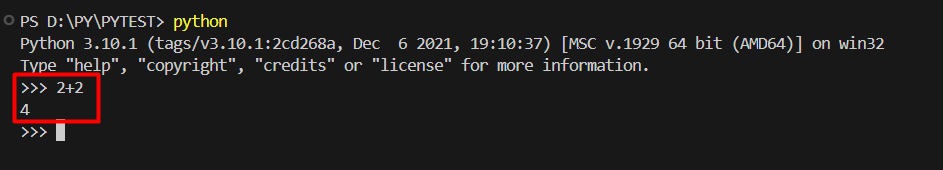
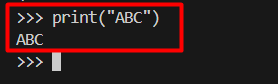
**LECTURE-08 (Saturday 31-May-2025)**

* Documentation of code implementation is very important
* The symbol “>>>” is commonly used in Python documentation, especially in doctests, to indicate an interactive Python shell prompt.
* In Python documentation, especially in tutorials and examples, “>>>” used to indicate lines that entered at the Python interactive prompt (also called the REPL: Read-Eval-Print Loop).
* The "interactive Python shell prompt" refers to the interface you get when you run Python in interactive mode, allowing you to type and execute Python code line by line, and see immediate results.
* Open Python interactive shell prompt in VSCode:

1. Open terminal in “Git Bash” or “Powershell”
2. Write “python” and press “Enter”







Here, lines starting with “>>>” represents user input typed at the Python shell while lines that follow (without the “>>>”) are the output returned by the interpreter.

“>>>” helps differentiate between what you type and what Python returns.

* “>>>” is a common format used by the doctest module to test code embedded in documentation.
* In VSCode, create python file “test.py” and add below code in file

def square(x):

    """

    Returns the square of a number.

    >>> square(-3)

    9

    """

    return x \* x

if \_\_name\_\_ == "\_\_main\_\_":

    import doctest

    doctest.testmod()

Here,

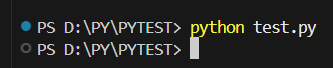
>>> square(-3)

    9

It is doctest means calling function “square” with required parameter value “-3” whose expected result is “9”.

There are two ways of running doctest:

1. run directly: means open terminal type “python test.py” and execute, if nothing failed then nothing will come in terminal

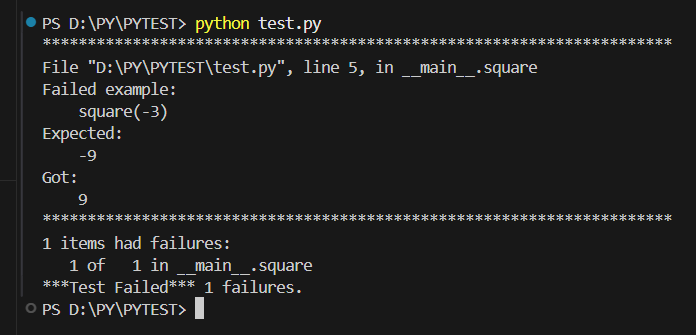


If test failed then means, output is not as per expected value then error raise. If we change above as

>>> square(-3)

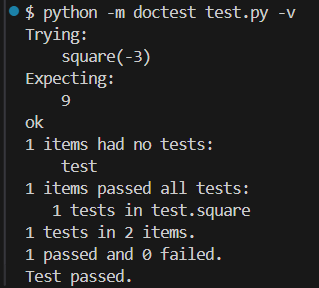
    -9

Then in output, error raised:

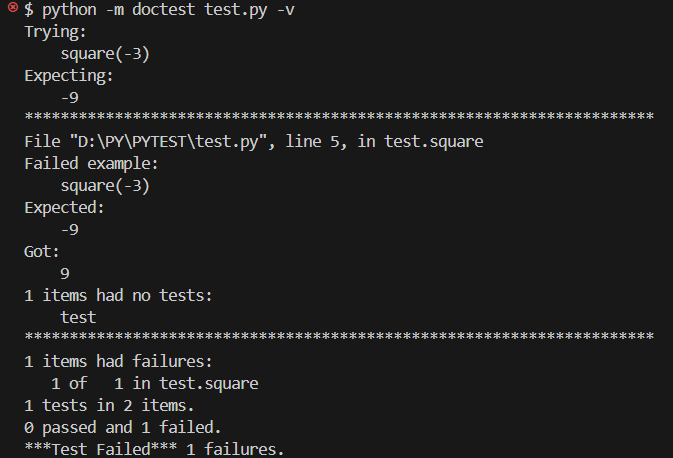


1. run in Git Bash terminal: means open “Git Bash” terminal and execute command “python -m doctest test.py –v”

Output:



Error Output:



TIPS:

1. Always write the exact output as it appears in the Python shell.
2. Use >>> for the first line and... for continued lines (e.g., multi-line statements).
3. Keep examples simple and correct, as they also serve as tests.

* Revision of functions
* Decorator function: It call another functions inside its body and returns wrapped function. It is required to define “wrapper” function in decorator function.
* Example of using with block in our custom function

from contextlib import contextmanager

@contextmanager

def my\_open():

  print("Database connection establised.")

  yield

  print("Database connection closed.")

with my\_open():

  print("Line1")

  print("Pakistan Zindabad")

Output:

Database connection establised.

Line1

Pakistan Zindabad

Database connection closed.

* Sync function:

%%time

import time

def function1(sec : int):

  time.sleep(sec)

  print(f"Mesage : {sec}")

function1(2)

function1(3)

function1(4)

Output:

Mesage : 2

Mesage : 3

Mesage : 4

CPU times: user 42.5 ms, sys: 7.38 ms, total: 49.9 ms

Wall time: 9 s

Here,

%%time: It is a magic command that used in Jupyter notebooks (or other IPython environments). It measures the total time takes to execute all the code in the current cell. It prints the wall time (actual time from start to end) once the cell finishes running.

* Asycnrohous function:

%%time

import time

import asyncio

async def function1(sec : int):

  time.sleep(sec)

  return print(f"Mesage : {sec}")

asyncio.gather(function1(2))

* We should know about functions of asyncio library

%%time

import time

import asyncio

async def function1(sec : int):

  time.sleep(sec)

  return print(f"Mesage : {sec}")

result = asyncio.gather(function1(2), function1(3), function1(4))

Output:

CPU times: user 200 µs, sys: 0 ns, total: 200 µs

Wall time: 208 µs

If we run below code in Google Colab:

import time

import asyncio

async def async\_function():

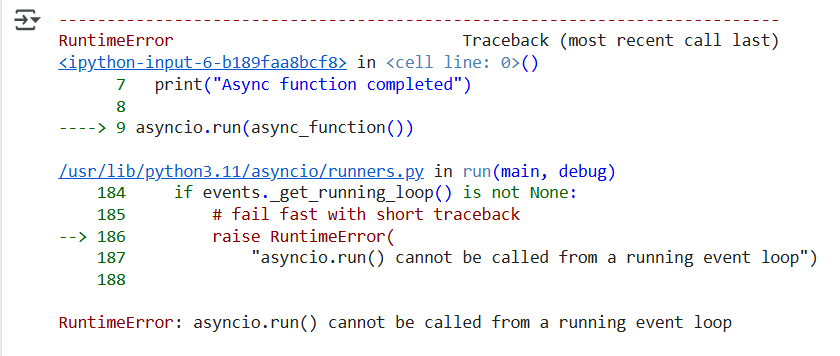
  print("Starting async function")

  await asyncio.sleep(2)

  print("Async function completed")

asyncio.run(async\_function())

We got error:



To fix this error, change code to below to install new package “nest\_asyncio” and use that package:

import time

import asyncio

!pip install nest\_asyncio

import nest\_asyncio

nest\_asyncio.apply()

async def async\_function():

  print("Starting async function")

  await asyncio.sleep(2)

  print("Async function completed")

asyncio.run(async\_function())

Output:

Requirement already satisfied: nest\_asyncio in /usr/local/lib/python3.11/dist-packages (1.6.0)

Starting async function

Async function completed

* <https://github.com/panaversity/learn-modern-ai-python/blob/main/00_python_colab/09_exception_handling/Agentic_AI_Python_Lesson_09_Exception_Handling.ipynb>
* Exception Handling: Below are the type of errors:

1. Development time
2. Testing time
3. Logical error
4. Production error
5. Development error

* Example of Error:

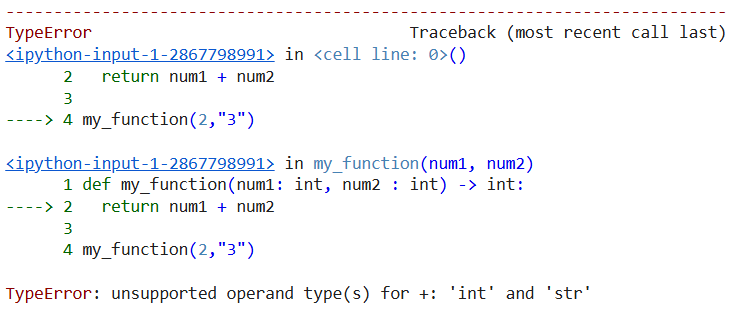
Code:

def my\_function(num1: int, num2 : int) -> int:

  return num1 + num2

my\_function(2,"3")

Error in Output:



Here, error occurred because in method argument, we have passed string value “3” while data type of method parameter is “int”

* Cursor: It is an AI-powered code editor (based on VS Code) that integrates large language models (like GPT) to help developers write, refactor, and understand code.
* For any function or file, if we want to create unit test then file name of test will be like

File: main.py

Test file: test\_main.py

* Example of test:

Code:

def function1(a : int, b : int) -> int:

  return a + b

To create unit test related to above code, use prompt: generate unit test for above code cell

Output:

# prompt: generate unit test for above code cell

import unittest

class TestFunction1(unittest.TestCase):

  def test\_function1\_positive\_integers(self):

    self.assertEqual(function1(2, 3), 5)

  def test\_function1\_negative\_integers(self):

    self.assertEqual(function1(-2, -3), -5)

  def test\_function1\_mixed\_integers(self):

    self.assertEqual(function1(2, -3), -1)

    self.assertEqual(function1(-2, 3), 1)

  def test\_function1\_zero(self):

    self.assertEqual(function1(0, 5), 5)

    self.assertEqual(function1(5, 0), 5)

    self.assertEqual(function1(0, 0), 0)

# To run the tests in a Colab notebook, you can do the following:

# unittest.main(argv=['first-arg-is-ignored'], exit=False)



Used above code in VS Code to run unit test.

File: main.py

Code:

def function1(a : int, b : int) -> int:

  return a + b

File: test\_main.py

Code:

import unittest

from main import function1

class TestFunction1(unittest.TestCase):

  def test\_function1\_positive\_integers(self):

    self.assertEqual(function1(2, 3), 5)

  def test\_function1\_negative\_integers(self):

    self.assertEqual(function1(-2, -3), -5)

  def test\_function1\_mixed\_integers(self):

    self.assertEqual(function1(2, -3), -1)

    self.assertEqual(function1(-2, 3), 1)

  def test\_function1\_zero(self):

    self.assertEqual(function1(0, 5), 5)

    self.assertEqual(function1(5, 0), 5)

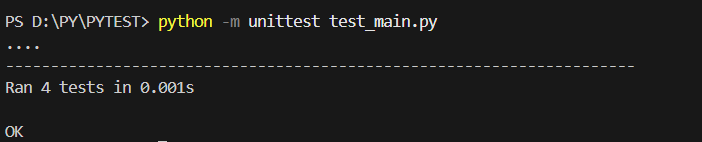
    self.assertEqual(function1(0, 0), 0)

Here, at top, added reference of “function1” of file “main.py”

File contains four unit tests for “function1”

1. test\_function1\_positive\_integers
2. test\_function1\_negative\_integers
3. test\_function1\_mixed\_integers
4. test\_function1\_zero

Run below command in terminal to execute unit test present in file “test\_main.py”:



Here, four unit tests executed successfully.

In addition, a folder created “\_\_pycache\_\_” which contains files of compiled python files of “main.py” and “test\_main.py”.

* Example of Logical Error:

Code:

def calculate\_average(numbers):

  total = 0

  for num in numbers:

    total += num

  return total  # Logical error here because the logic for computing the average (total / len(numbers)) is missing.

nums = [10, 20, 30]

average = calculate\_average(nums)

print("Average:", average)

* Example of Production Error:

Example 1:

file = open("abc.txt")

Output Error:

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

FileNotFoundError Traceback (most recent call last)

[<ipython-input-1-3389400699>](https://localhost:8080/) in <cell line: 0>()

----> 1 file = open("abc.txt")

FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'

Example 2:

print(7/0)

Output Error:

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

ZeroDivisionError Traceback (most recent call last)

[<ipython-input-2-3418871042>](https://localhost:8080/) in <cell line: 0>()

----> 1 print(7/0)

ZeroDivisionError: division by zero

Example 3:

names : list[str] = ["A", "B", "C"]

print(names[10])

Ouput Error:

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

IndexError Traceback (most recent call last)

[<ipython-input-3-3905196323>](https://localhost:8080/) in <cell line: 0>()

**1** names : list[str] = ["A", "B", "C"]

**2**

----> 3 print(names[10])

IndexError: list index out of range

* Try-except: For any code, which is expect to raise error, we need to add try-except block.

Example of try-except block:

print("hi")

try:

  7/0

except ZeroDivisionError:

  print("You are dividing by zero which is not allowed")

print("bye")

Output:

hi

You are dividing by zero which is not allowed

bye

Example of more than one error handling in one except block:

names = ["A", "B", "C"]

try:

  print(names[10])

except (ZeroDivisionError, IndexError):

  print("wrong value")

Output:

wrong value

OR

names = ["A", "B", "C"]

try:

  print(10/0)

except (ZeroDivisionError, IndexError):

  print("wrong value")

Output:

wrong value

Example of more than one except blocks:

names = ["A", "B", "C"]

try:

  print(names[10])

except (ZeroDivisionError):

  print("zero division")

except (IndexError):

  print("index error")

Output:

index error

OR

names = ["A", "B", "C"]

try:

  print(9/0)

except (ZeroDivisionError):

  print("zero division")

except (IndexError):

  print("index error")

Output:

zero division

* If at any line, error raised then code present after that line, will not be executed.

Example:

names = ["A", "B", "C"]

try:

  print(names[10])

  print(7/10)

except (ZeroDivisionError):

  print("zero division")

except (IndexError):

  print("index error")

Output:

index error

Here, line “print(7/10)” will not be executed and to overcome this issue, we can use multiple try-except blocks.

Example:

names = ["A", "B", "C"]

try:

  print(names[10])

except (IndexError):

  print("index error")

try:

  print(7/0)

except (ZeroDivisionError):

  print("zero division")

Output:

index error

zero division

* To handle any error, we can use “Exception” class in except block. Exception will handle any error.

Example:

names = ["A", "B", "C"]

try:

  print(names[10])

  open("abc.txt")

  names[10]

except Exception as e:

  print("error occurred : ", e)

Output:

error occurred : list index out of range

OR

names = ["A", "B", "C"]

try:

  open("abc.txt")

  names[10]

except Exception as e:

  print("error occurred : ", e)

Output:

error occurred : [Errno 2] No such file or directory: 'abc.txt'

OR

names = ["A", "B", "C"]

try:

  names[10]

except Exception as e:

  print("error occurred : ", e)

Output:

error occurred : list index out of range

OR

names = ["A", "B", "C"]

try:

  print(names[10])

  open("abc.txt")

  names[10]

except (ZeroDivisionError):

  print("zero division")

except (IndexError):

  print("index error")

except Exception as e:

  print("index error", e)

Above code will handle specific and other errors as well.

* Raise function: This function is used to raise custom (user-defined) exception.

Example:

class CustomError(Exception):

  pass

try:

  raise CustomError("This is a custom error")

except CustomError as e:

  print(e)

Output:

This is a custom error

OR

class CustomError(Exception):

  pass

names = ["A", "B", "C"]

try:

  try:

    print(7/0)

  except (ZeroDivisionError):

    raise CustomError("This is my error")

  except (IndexError):

    print("index error")

  except Exception as e:

    print("exception occurred", e)

except CustomError as e:

  print(e)

Output:

This is my error