PES UNIVERSITY

Department of Computer Science and Engineering

PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

What is software testing?

Software testing is a process

- to evaluate the functionality of a software application with an intent to find whether the developed software meets the specified requirements or not.
- to identify the defects to ensure that the product is defect-free in order to produce a quality product.

Why software testing is important?

- Software defects can damage a brand's reputation — leading to frustrated and lost customers.
- In extreme cases, a bug or defect can cause serious malfunctions/failures.

Examples:

- Consider Nissan having to recall over 1 million cars due to a software defect in the airbag sensor detectors.
- A software bug that caused the failure of a USD 1.2 billion military satellite launch.

Benefits of software testing.

1. Product Quality
2. Customer Satisfaction
3. Security
4. Cost-effectiveness

Python assert Keyword

What is Assertion?

- Assertions in any programming language are the debugging tools that help in the smooth flow of code.
- Assertions are mainly assumptions that a programmer knows always wants to be true and hence puts them in code so that failure of them doesn't allow the code to execute further.

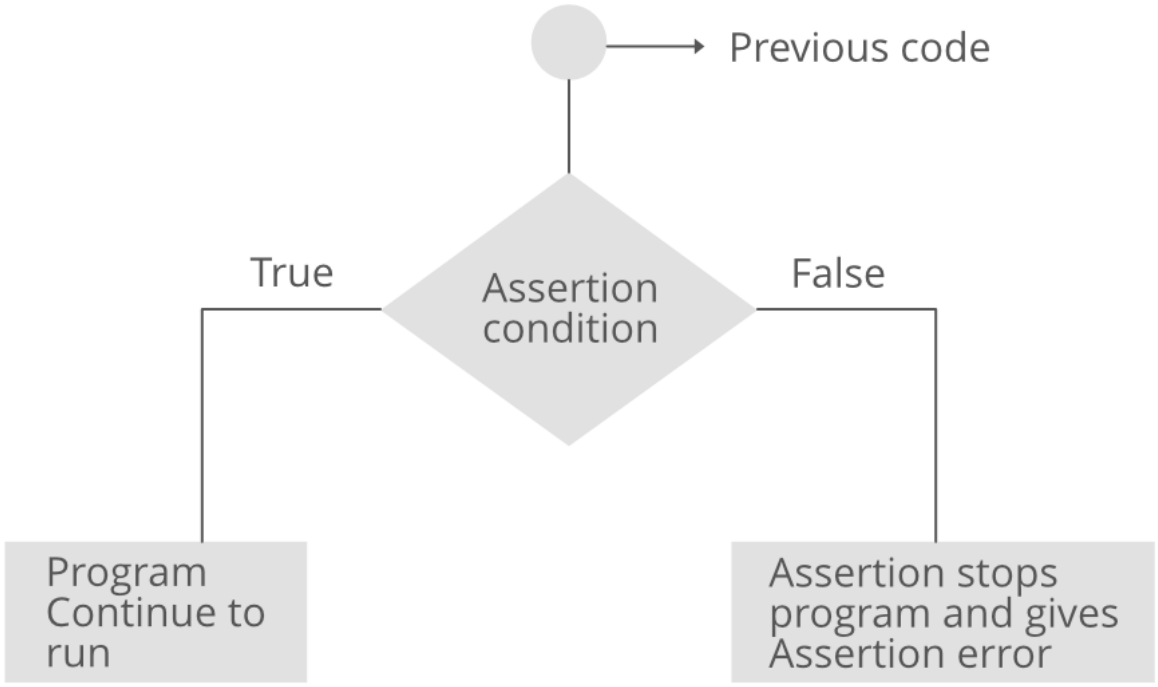
In simpler terms, Assertion is the assert keyword with boolean expression, assert checks whether the boolean expression is True or False.

If the boolean expression is evaluated to True then it does nothing and continues the execution of the program.

If the boolean expression is evaluated to False then it stops the execution of the program and throws an AssertionError.

ELet us look at the flowchart of the assertion.

Flowchart of Assertion



Assert Keyword

In python assert keyword helps in achieving this task. This statement simply takes input a boolean condition, which when returns true doesn't return anything, but if it is computed to be false, then it raises an AssertionError along with the optional message provided.

Syntax:

assert condition [, Error_Message]

Parameters:

- condition: The boolean condition returning true or false.
- Error_Message: The optional argument to be printed in console in case of AssertionError
- Returns: Returns AssertionError, in case the condition evaluates to false along with the error message which when provided.

Example 1:

Test if a condition returns True:

In [2]:
```
x = "hello"
```

In [3]:
```
#if condition returns True, then nothing happens:
assert x == "hello"
print('How are you?')
```

```
How are you?
```

In [3]:
```
#if condition returns False, AssertionError is raised:
assert x == "goodbye", 'Strings are not matching'
print('How are you?')
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [3], in <cell line: 2>()
      1 #if condition returns False, AssertionError is raised:
----> 2 assert x == "goodbye", 'Strings are not matching'
      3 print('How are you?')

AssertionError: Strings are not matching
```

Example 2: Python assert keyword without error message

In [9]:
```
# initializing number
a = 4
b = 1

# using assert to check whether denominator is 0
print("The value of a / b is : ")
assert b != 0
print(a / b)
```

```
The value of a / b is :
4.0
```

Python assert keyword with error message

In [4]:
```
# initializing number
a = 4
b = 0

# using assert to check for 0
print("The value of a / b is : ")
assert b != 0, 'The value of b is 0'
print(a / b)
```

```
The value of a / b is :
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [4], in <cell line: 7>()
      5 # using assert to check for 0
      6 print("The value of a / b is : ")
----> 7 assert b != 0, 'The value of b is 0'
      8 print(a / b)

AssertionError: The value of b is 0
```

Example 3: Python assert keyword with error message

In [5]:
```
# initializing number
a = 4
b = 0

# using assert to check for 0
print("The value of a / b is : ")
assert b != 0, "Division by zero"    #Write a message if the condition is False
print(a / b)
```

```
The value of a / b is :
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [5], in <cell line: 7>()
      5 # using assert to check for 0
      6 print("The value of a / b is : ")
----> 7 assert b != 0, "Division by zero"   #Write a message if the condition is False
      8 print(a / b)

AssertionError: Division by zero
```

Practical Application

- This has a much greater utility in testing and Quality assurance role in any development domain.
- Different types of assertions are used depending upon the application.

Example 4: A program that only allows the batch with all hot food to be dispatched, else rejects the whole batch.

In [9]:
```
# initializing list of foods temperatures
batch1 = [ 40, 26, 39, 30, 25, 21]

# initializing cut temperature
cut = 26   # less than 26 are not allowed

# using assert to check for temperature greater than cut
for i in batch1:
    assert i >= 26, "Batch is Rejected"   #Write a message if the condition is False
    print (str(i) + " is O.K" )
print('Batch is Accepted')
```

```
40 is O.K
26 is O.K
39 is O.K
30 is O.K
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [9], in <cell line: 8>()
      7 # using assert to check for temperature greater than cut
      8 for i in batch1:
----> 9     assert i >= 26, "Batch is Rejected"   #Write a message if the condition is F
alse
     10     print (str(i) + " is O.K" )
     11 print('Batch is Accepted')

AssertionError: Batch is Rejected
```

In [8]:
```
# initializing list of foods temperatures
batch2 = [ 39, 26, 37, 32, 40, 38]

# initializing cut temperature
cut = 26   # less than 26 are not allowed

# using assert to check for temperature greater than cut
for i in batch2:
    assert i >= 26, "Batch is Rejected"   #Write a message if the condition is False
    print (str(i) + " is O.K" )
print('Batch is Accepted')
```

```
39 is O.K
26 is O.K
37 is O.K
32 is O.K
```

```
40 is O.K
38 is O.K
Batch is Accepted
```

Pytest: helps you write better programs

- Pytest is a python based testing framework. Pytest is free and open source.
- PyTest is mainly used for writing tests for APIs. It helps to write tests from simple unit tests to complex functional tests.

Why use PyTest?

Some of the advantages of pytest are

- Very easy to start with because of its simple and easy syntax.
- Can run tests in parallel.
- Can run a specific test or a subset of tests

To install the latest version of pytest, execute the following command:

$ pip install pytest

Confirm the installation using the following command to display the help section of pytest.

$ pytest -h

An example of a simple test:

In [12]:
```python
# test_sample.py
def inc_by_1(x):
    return x + 1

def test_inc_by_1():
    assert inc_by_1(3) == 4
    print('Code is fine')
```

```
(base) C:\Users\DELL\Python Pgms - PCPS>pytest test_sample.py
================================== test session starts ==================================
platform win32 -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0
rootdir: C:\Users\DELL\Python Pgms - PCPS
plugins: anyio-3.5.0
collected 1 item

test_sample.py .                                                                  [100%]

=================================== 1 passed in 0.05s ===================================
```

In [ ]:
```python
# test_sample.py
def inc_by_1(x):
    return x + 2  # wrong code

def test_inc_by_1():
    assert inc_by_1(3) == 4
```

To execute it:

$ pytest test_sample.py

```
(base) C:\Users\DELL\Python Pgms - PCPS>pytest test_sample.py
================================== test session starts ==================================
platform win32 -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0
rootdir: C:\Users\DELL\Python Pgms - PCPS
plugins: anyio-3.5.0
collected 1 item

test_sample.py F                                                                  [100%]

======================================= FAILURES ========================================
_____________________________________ test_inc_by_1 _____________________________________

    def test_inc_by_1():
>       assert inc_by_1(3) == 4
E       assert 5 == 4
E        +  where 5 = inc_by_1(3)

test_sample.py:7: AssertionError
================================== short test summary info ==================================
FAILED test_sample.py::test_inc_by_1 - assert 5 == 4
=================================== 1 failed in 0.23s ===================================
```

Note:

- F says failure
- Dot(.) says success.

In [ ]:
```python
# test_sample1.py

def test_builtin_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"
```

In [ ]:
```python
# test_sample2.py
import math

def test_sqrt():
    num = 25
    assert math.sqrt(num) == 5

def test_square():
    num = 7
    assert 7*7 == 40

def test_equality():
    assert 10 == 11
```

```
(base) C:\Users\DELL\Python Pgms - PCPS>pytest test_sample2.py -v
================================== test session starts ==================================
platform win32 -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0 -- C:\Users\DELL\anaconda3\python
.exe
cachedir: .pytest_cache
rootdir: C:\Users\DELL\Python Pgms - PCPS
plugins: anyio-3.5.0
collected 3 items

test_sample2.py::test_sqrt PASSED                                                 [ 33%]
test_sample2.py::test_square FAILED                                               [ 66%]
test_sample2.py::test_equality FAILED                                             [100%]
```

```
========================================= FAILURES =========================================
_____________________________________ test_square _____________________________________

    def test_square():
        num = 7
>       assert 7*7 == 40
E       assert (7 * 7) == 40

test_sample2.py:10: AssertionError
_____________________________________ test_equality _____________________________________

    def test_equality():
>       assert 10 == 11
E       assert 10 == 11

test_sample2.py:13: AssertionError
================================== short test summary info ==================================
FAILED test_sample2.py::test_square - assert (7 * 7) == 40
FAILED test_sample2.py::test_equality - assert 10 == 11
================================ 2 failed, 1 passed in 0.19s ================================
```

Note:

$ pytest

The above command(without filename) will execute all the files of format test_ *or* _test in the current directory and subdirectories.

In [1]:
```
# test_compare.py
def test_greaterthan():
    num = 100
    assert num > 100

def test_greater_equal():
    num = 100
    assert num >= 100

def test_lessthan():
    num = 100
    assert num < 200
```

To execute the tests from a specific file, use the following syntax

In [ ]:
```
$ pytest filename -v              # -v increases the verbosity.
```

How command pytest Identifies the Test Files and Test Methods

- By default command pytest only identifies the file names starting with test_ or ending with _test as the test files.
- We can explicitly mention other filenames through ($ pytest filename.py).
- Pytest requires the test method names to start with “test.” All other method names will be ignored even if we explicitly ask to run those methods.

See some examples of valid and invalid pytest file names for batch execution

- test_login.py -> valid
- login_test.py -> valid
- testlogin.py -> invalid
- logintest.py -> invalid

Note: Yes we can explicitly ask pytest to pick testlogin.py and logintest.py

$ pytest testlogin.py

$ pytest logintest.py

See some examples of valid and invalid pytest test methods

- def test_file1_method1(): -> valid
- def testfile1_method1(): -> valid
- def file1_method1(): -> invalid

Note: Even if we explicitly mention function call file1_method1() pytest will not run this method.

Debugging With pdb

- In all programming exercises, it is difficult to go far and deep without a handy debugger.
- pdb is part of Python's standard library, so it's always there and available for use.

pdb (Python DeBugger) — An interactive source code debugger

- The module pdb defines an interactive source code debugger for Python programs.
- pdb supports
    1. setting (conditional) breakpoints and single stepping at the source line level,
    2. inspection of stack frames,
    3. source code listing, and
    4. evaluation of arbitrary Python code in the context of any stack frame.
- Pdb also supports post-mortem debugging and can be called under program control.

The debugger is extensible – it is actually defined as the class Pdb. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules bdb and cmd.

The debugger's prompt is (Pdb).

The simplest operation under a debugger is to step through the code.

- That is to run one line of code at a time and wait for your acknowledgment before proceeding into next.
- The reason we want to run the program in a stop-and-go fashion is to allow us to check the logic and value or verify the algorithm.

For a larger program, we may not want to step through the code from the beginning as it may take a long time before we reached the line that we are interested in.

- Therefore, debuggers also provide a breakpoint feature that will kick in when a specific line of code is reached.
- From that point onward, we can step through it line by line.

Walk-through of using a debugger

Let's see how we can make use of a debugger with an example.

In [ ]:
```
# Given a list of strings, find their corresponding lengths
# filename: debugging_demo1.py
def what1(x):
```

```
    res = []
    for w in x:
        res.append(len(w))
    return res

a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
b = what1(a)
print(b)
```

Assume this program is saved as debugging_demo1.py, to run this program in command line is simply enter:

$ python debugging_demo1.py

and the solution will be printed on the screen.

But if we want to run it with the Python debugger, we enter the following in command line:

$ python -m pdb debugging_demo1.py

- The -m pdb part is to load the pdb module and let the module to execute the file debugging_demo1.py for you.
- When you run this command, you will be welcomed with the pdb prompt as follows:

```
base) C:\Users\DELL\Python Pgms - PCPS>python -m pdb debugging_demo1.py
 c:\users\dell\python pgms - pcps\debugging_demo1.py(3)<module>()
> def what1(x):
Pdb)
```

At the beginning of a debugger session, we start with the first line of the program.

We can use

1. n to move to the next line, or
2. s to step into a function and
3. h for help:

```
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb)
```

Printing a Variable's Value

In this example, we'll look at using pdb in its simplest form: checking the value of a variable.

Insert the following code at the location where you want to break into the debugger:

In [ ]:
```
import pdb; pdb.set_trace()
```

- When the line above is executed, Python stops and waits for you to tell it what to do next.
- You'll see a (Pdb) prompt. This means that you're now paused in the interactive debugger and can enter a command.

Starting in Python 3.7, there's another way to enter the debugger.

PEP 553 describes the built-in function breakpoint(), which makes entering the debugger easy and consistent:

In [ ]:
```
breakpoint()
```

- By default, breakpoint() will import pdb and call pdb.set_trace(), as shown above.
- breakpoint() method is more flexible and allows you to control debugging behavior via its API.
- If you're using Python 3.7 or later, I encourage you to use breakpoint() instead of pdb.set_trace().

You can also break into the debugger, without modifying the source and using pdb.set_trace() or breakpoint(), by running Python directly from the command-line and passing the option -m pdb.

$ python -m pdb debugging_demo1.py

Let's look at the example. Here's the example1.py source:

In [ ]:
```
#example1.py
filename = __file__
import pdb; pdb.set_trace()
print(f'path = {filename}')
```

If you run this, you should get the following output:

```
(base) C:\Users\DELL\Python Pgms - PCPS\testing and debugging>python example1.py
> c:\users\dell\python pgms - pcps\testing and debugging\example1.py(3)<module>()
-> print(f'path = {filename}')
(Pdb) _
```

Now enter p filename. You should see:

```
(Pdb) p filename
'C:\\Users\\DELL\\Python Pgms - PCPS\\testing and debugging\\example1.py'
(Pdb) _
```

(Pdb) is pdb's prompt. It's waiting for a command. Use the command q to quit debugging and exit.

In [ ]:
```
#example2.py
import os

def get_path(filename):
```

```
    """Return file's path or empty string if no path."""
    head, tail = os.path.split(filename)
    import pdb; pdb.set_trace()
    return head

filename = __file__
print(f'path = {get_path(filename)}')
```

```
(base) C:\Users\DELL\Python Pgms - PCPS\testing and debugging>python example2.py
> c:\users\dell\python pgms - pcps\testing and debugging\example2.py(8)get_path()
-> return head
(Pdb) p head
'C:\\Users\\DELL\\Python Pgms - PCPS\\testing and debugging'
(Pdb) p tail
'example2.py'
(Pdb)
```

In [ ]:
```
# Example3.py
def what1(x):
    res = []
    for w in x:
        res.append(len(w))
    return res

a1 = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
breakpoint()    #import pdb; pdb.set_trace()
b = what1(a1)
print(b)
```

breakpoint()

When execution reaches this point, the program stops and you're dropped into the pdb debugger.

The pdb module defines the following functions; each enters the debugger in a slightly different way:

1)pdb.run(statement, globals=None, locals=None)

- Execute the statement (given as a string or a code object) under debugger control.
- The debugger prompt appears before any code is executed; you can set breakpoints and type continue, or you can step through the statement using step or next (all these commands are explained below).
- The optional globals and locals arguments specify the environment in which the code is executed; by default the dictionary of the module **main** is used. (See the explanation of the built-in exec() or eval() functions.)

In [ ]:
```
# example4.py

def divide(numerator, denominator):
    res = numerator/denominator
    return res

import pdb
pdb.run('divide(10,5)')
```

```
(base) C:\Users\DELL\Python Pgms - PCPS\testing and debugging>python example4.py
> <string>(1)<module>()
(Pdb) s
--Call--
> c:\users\dell\python pgms - pcps\testing and debugging\example4.py(3)divide()
-> def divide(numerator, denominator):
(Pdb) n
> c:\users\dell\python pgms - pcps\testing and debugging\example4.py(4)divide()
-> res = numerator/denominator
(Pdb) n
> c:\users\dell\python pgms - pcps\testing and debugging\example4.py(5)divide()
-> return res
(Pdb) p numerator,denominator,res
(10, 5, 2.0)
(Pdb)
```

Here string(1) module() means that we are at the start of string passed to run() and no code has executed yet. In the above example we stepped into the divide function using s.

runeval() does the same thing as run() except that it also returns the value of executed code.

2)pdb.runeval(expression, globals=None, locals=None)

- Evaluate the expression (given as a string or a code object) under debugger control.
- When runeval() returns, it returns the value of the expression. Otherwise this function is similar to run().

In [ ]:
```
# example5.py

def divide(numerator, denominator):
    result = numerator/denominator
    return result

import pdb
pdb.runeval('divide(10,5)')
```

```
(base) C:\Users\DELL\Python Pgms - PCPS\testing and debugging>python example5.py
> <string>(1)<module>()
(Pdb) n
--Return--
> <string>(1)<module>()->2.0
(Pdb)
```

3)pdb.runcall(function, *args, **kwds)

- runcall() allows us to pass a Python callable itself instead of a string.
- Call the function (a function or method object, not a string) with the given arguments.
- When runcall() returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

In [ ]:
```
# example5a.py

def divide(numerator, denominator):
    result = numerator/denominator
    return result

import pdb
pdb.runcall(divide,10,5)
```

```
(base) C:\Users\DELL\Python Pgms - PCPS\testing and debugging>python example5a.py
> c:\users\dell\python pgms - pcps\testing and debugging\example5a.py(4)divide()
-> result = numerator/denominator
(Pdb) n
> c:\users\dell\python pgms - pcps\testing and debugging\example5a.py(5)divide()
-> return result
(Pdb) p result
2.0
(Pdb)
```

4)pdb.set_trace(*, header=None)

- This is the most common way to debug programs, it basically involves adding the line pdb.set_trace() in the source code wherever we want our program to stop.
- This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).
- If given, header is printed to the console just before debugging begins. Changed in version 3.7: The keyword-only argument header.

5)pdb.pm()

Enter post-mortem debugging of the traceback found in sys.last_traceback.

In [ ]:
```
# example6.py                           # line-01
                                        # line-02
def divide(numerator, denominator):     # line-03
    result = numerator/denominator      # line-04
    return result

divide(10,0)
```

```
>>> import example6.py
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\DELL\Python Pgms - PCPS\testing and debugging\example6.py", line 7, in <mod
le>
    divide(10,0)
  File "C:\Users\DELL\Python Pgms - PCPS\testing and debugging\example6.py", line 4, in divi
e
    result = numerator/denominator
ZeroDivisionError: division by zero
>>> import pdb; pdb.pm()
> c:\users\dell\python pgms - pcps\testing and debugging\example6.py(4)divide()
-> result = numerator/denominator
(Pdb)
```

6)pdb.post_mortem(traceback=None)

- Enter post-mortem debugging of the given traceback object.
- If no traceback is given, it uses the one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

We can handle the current exception being handled using pdb.post_mortem() without any argument:

In [ ]:
```
# example7.py

def divide(numerator, denominator):
```

```
    try:
        return numerator/denominator
    except Exception:
        import pdb; pdb.post_mortem()

divide(10,0)
```

```
(base) C:\Users\DELL\Python Pgms - PCPS\testing and debugging>python example7.py
> c:\users\dell\python pgms - pcps\testing and debugging\example7.py(5)divide()
-> return numerator/denominator
(Pdb) _
```

In [ ]:
```
# example8.py

import pdb
import sys
pdb.post_mortem(sys.last_traceback)
```

```
(base) C:\Users\DELL\Python Pgms - PCPS\testing and debugging>python example7.py
> c:\users\dell\python pgms - pcps\testing and debugging\example7.py(5)divide()
-> return numerator/denominator
(Pdb) _
```

Debugger Commands

- p-To print the varaible value.
- n-Continue execution untill next line is reached.
- l-To list the source code.
- b-List all the breakpoints.
- h-To get the list of available commnads.
- q-To quit the debugger.
- u (up): Allows you to go up one level in the backtrace.
- d (down): Allows you to go down one level in the backtrace.

Essential pdb Commands

Just enter h or help to get a list of all commands or help for a specific command or topic.

For quick reference, here's a list of essential commands:

Command	Description
`p`	Print the value of an expression.
`pp`	Pretty-print the value of an expression.
`n`	Continue execution until the next line in the current function is reached or it returns.
`s`	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).
`c`	Continue execution and only stop when a breakpoint is encountered.
`unt`	Continue execution until the line with a number greater than the current one is reached. With a line number argument, continue execution until a line with a number greater or equal to that is reached.
`l`	List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing.
`ll`	List the whole source code for the current function or frame.
`b`	With no arguments, list all breaks. With a line number argument, set a breakpoint at this line in the current file.
`w`	Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.
`u`	Move the current frame count (default one) levels up in the stack trace (to an older frame).
`d`	Move the current frame count (default one) levels down in the stack trace (to a newer frame).
`h`	See a list of available commands.
`h <topic>`	Show help for a command or topic.
`h pdb`	Show the full pdb documentation.
`q`	Quit the debugger and exit.

References:

1. https://docs.python.org/3/library/doctest.html
2. https://docs.pytest.org/en/6.2.x/

In [ ]:
```
$ python -m pdb myscript.py
```

- When invoked as a script, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally.
- After post-mortem debugging (or after normal exit of the program), pdb will restart the program.
- Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.