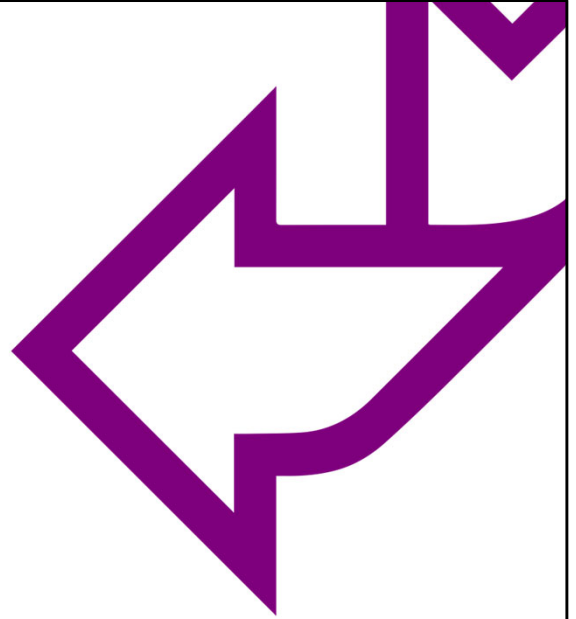




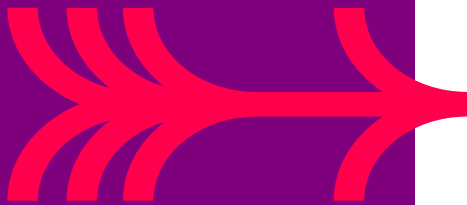
# Python 3 Programming

Testing





# TESTING



## Contents

- What is software testing
- Types of testing
- Manual vs automated
- Create a simple calculator app
- Docstrings
- Docstrings testing
- Docstrings automated testing
- Unit testing
- The assert statement
- Test runners
- Test scripts
- The Pytest module
- Pytest output
- Summary

## QA What is software testing?

### Checks whether the Application meets the expected requirements:

- Check for **Functional** requirements (When I click this button, an email is sent).
- Check for **Non-Functional** requirements (email is sent within 1 seconds and in a secure way).

### Program code and logic is errorfree:

- Logical errors
- Arithmetic calculations...

### Application errors:

- Different modules work well together.
- System as a whole works.
- Works on the client system.

3

Software testing is the process of evaluating and verifying that a software application does what it is supposed to do and matches requirements. It is simply to confirm that it is defect free and there is no missing functionality.

## QA Why software testing is so important

### Testing helps to solve problems early because:

- bugs are expensive to fix in production.
- lives could be affected (airline, transport, etc.).
- reputation could be affected.
- performance and security are critical.

### Examples of disasters caused by not testing:

- <https://dzone.com/articles/the-biggest-software-failures-in-recent-years>

4

There are many benefits of software testing, but the prime goal is to provide the best features and experience with no bugs or side effects. This in turn builds your reputation and enhances customer satisfaction. And leads to repeat business.

It can also lead to cost savings with early defect detection, economical fixes, fewer design changes and lower maintenance costs. Comprehensive testing can also lead to better quality code with lower failure.

In this digital web enabled world, safety and security is so important. Think about buying goods online, sharing personal details, or critical infrastructure and services. Testing can improve security against hackers, malicious attacks and thefts. It can even save lives by ensuring the software in the aviation world works correctly.

## QA What are the different types of test?



**Tests can be categorised as these general types:**

Manual or automated  
Unit testing  
Integration testing  
System testing  
Acceptance testing



**Investigate unit testing using several Python modules:**

Document testing using **doctest**  
Test framework using **unittest**  
Test framework using **pytest**

## QA Manual or automated tests

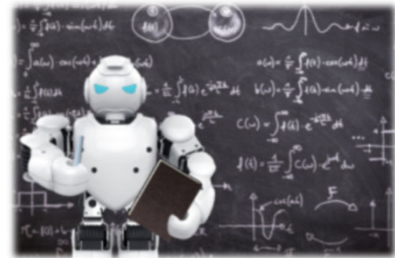
### Manual testing

- Cannot be designed and performed by anyone.
- Requires deep analysis of the requirements and specialised knowledge.
- Exploratory testing.



### Automated testing

- Manually checking code may not be possible. For example, when testing: Performance, Parallel execution, Load tests, Penetration test...
- Unit testing is often used to test code.
- It is the duty of the coder to write these.



6

You may not have realised, but you probably have already performed testing on your code! When you run your application, most coders will check to see if key features work, and if they do, they might consider a follow-on test. That is known as exploratory testing and is a form of manual testing, usually without a plan.

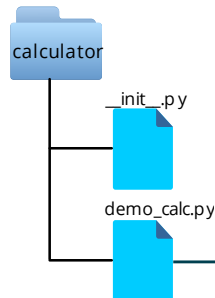
As the application scales, the list of features, and expected inputs and outputs increase. And every time the code changes you will have to manually re-apply all the tests. And possibly add more. Clearly not workable.

The solution is to automate your test plan using scripts. Luckily, Python already comes with a set of tools and libraries to automate your tests for your applications. We will discover how to use the **doctest** module to do DocString testing, and the **unittest** and **pytest** modules to perform automated unit testing.

## QA A simple calculator app

### Demonstration

- Create a new Package called calculator.
- Notice the `__init__.py` file.
- Create a new script called `demo_calc.py`.
- Define add, multiply, and divide functions.
- Add and multiply functions should allow multiple parameters.
- Define `main()` function to call functions.



demo\_calc.py

```
def add(*args):
    total = 0
    for num in args:
        total += num
    return total

def mul(*args):
    total = 1
    for num in args:
        total *= num
    return total

def div(x, z):
    return float(f"{x/z:.3f}")

def main():
    print(f"4 + 3 = {add(4, 3)}")
    print(f"4 - 3 = {sub(4, 3)}")
    print(f"4 * 3 = {mul(4, 3)}")
    print(f"4 / 3 = {div(4, 3)}")
    return None

if __name__ == "__main__":
    main()
```

To practice our testing capabilities, we will create a simple calculator application **demo\_calc.py** with add, multiply, and divide operations.

Test the program by creating a `main()` function that calls all the functions with two simple parameters 4 and 3. Execute the program and confirm that all is well.

You may have done some of these steps in an earlier chapter.

## QA A simple calculator app

Manual testing on the REPL or command prompt.

```
IDLE Shell 3.10.6
>>> import sys
>>> sys.path.append(r"C:\labs\projects\ProjectA\calculator")
>>> import demo_calc
>>> demo_calc.add(4, 3)
7
>>> demo_calc.mul(4, 3)
12
>>> demo_calc.div(4, 3)
1.333
>>>
>>>
```

```
Select Command Prompt
C:\labs\projects\ProjectA\calculator> python demo_calc.py
4 + 3 = 7
4 * 3 = 12
4 / 3 = 1.333
C:\labs\projects\ProjectA\calculator>
```

8

Test the app manually by executing the script in your command prompt or importing the script at the REPL prompt and calling the functions manually. You may have to import the **sys** module and append the location of the script to the `sys.path` list.

You might want to manually test the add and multiply functions with multiple numeric parameters.

You did remember to put comments in your script or is there a better way to document your script?





## DOCUMENT STRINGS AND TESTING



### Document and test your code – Docstrings:

- Documents the script.
- Documents the functions.
- PEP 008 compliance.
- Provides help().
- It can be tested!



9

If you are interested in writing your documentation at the same time as the code and want to incorporate test cases at the same time, then Python's **doctest** module should be considered.

It provides a simple testing framework that allows you to document your script with test examples and automate the testing of these examples – and helps keep your code and documentation synchronised.

Some coders would say that documentation is just as important as code, and some may even argue it is more important. But all would agree that anything that helps testing is a good thing!

## QA Docstrings – format and location

**Docstrings are triple quoted strings that document your code as you code:**

- Module docstrings placed before any code.
- Function/class docstrings at start of block before any code.
- Can be simple single line or multi-line.

```
"""
    Calculator program with add, multiply,
    and divide functions.
"""

def add(*args):
    """ Returns the sum of all parameters """
    total = 0
    for num in args:
        total += num
    return total
```

demo\_calc2.py

```
def mul(*args):
    """ Returns the product of all
    parameters. """
    total = 1
    for num in args:
        total *= num
    return total

def div(x, z):
    """ Returns x divided by z, as a float
    to 3 decimal places. """
    return float(f"{x/z:.3f}")
```

10

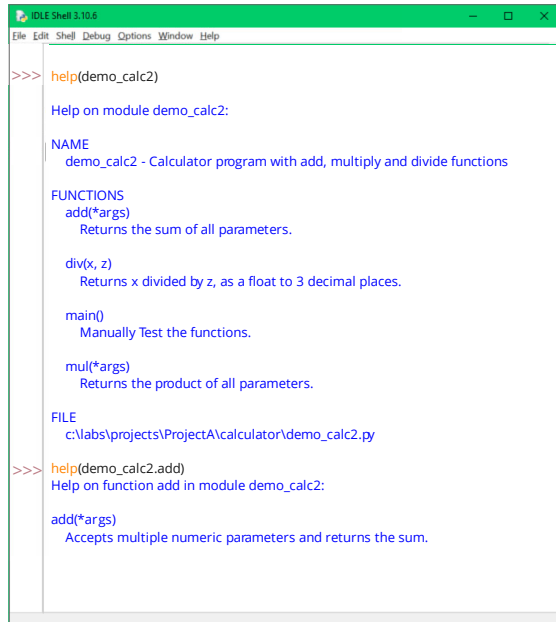
Comments are useful for describing code and in Python we use a # symbol for single line comments. Unfortunately, these comments are ignored by the interpreter, unavailable at runtime and can often become out of date!

Step forward, **docstrings** – these triple quoted (single or double) strings allow multi-line comments that remain within your code at runtime. They are displayed with the built-in help() function and modules such as MkDocs and Sphinx can be used to generate project documentation.

The special attribute `__doc__` for packages, modules, classes and functions.

Docstrings can be added to packages, modules, classes, methods and functions and are described in the PEP 257 document.

## QA Docstrings – for documentation



```
>>> help(demo_calc2)

Help on module demo_calc2:

NAME
demo_calc2 - Calculator program with add, multiply and divide functions

FUNCTIONS
add(*args)
    Returns the sum of all parameters.

div(x, z)
    Returns x divided by z, as a float to 3 decimal places.

main()
    Manually Test the functions.

mul(*args)
    Returns the product of all parameters.

FILE
c:\labs\projects\ProjectA\calculator\demo_calc2.py

>>> help(demo_calc2.add)
Help on function add in module demo_calc2:

add(*args)
    Accepts multiple numeric parameters and returns the sum.
```

Use `help()` to see the docstrings:

```
>>> help(demo_calc2)
>>> help(demo_calc2.add)
```

Or the special attribute `__doc__`:

```
>>> print(demo_calc2.__doc__)
>>> print(demo_calc2.add.__doc__)
```

Use `pydoc` module to display in text or HTML:

```
c:> %PYTHONPATH%\pydoc demo_calc2.py
c:> %PYTHONPATH%\pydoc -w demo_calc2.py
```

Use the built-in `help()` function to display the embedded docstrings. You can also display the docstring for a specific function/method by using the dot notation – `help(demo_calc.add)`.

Alternatively, within a script, you can print out docstrings using the special `__doc__` special attribute which can be combined with package, module, class, function or method names.

The **pydoc** module, from the Python standard library, can also be used on the command line to print out the docstrings in text or html using the `-w` option.

## QA Docstrings – for usage examples

Docstrings can also have embedded USAGE examples:

- Same format as testing at REPL prompt.
- Can have multiple examples in each docstring.

demo\_calc3.py

```
def add(*args):  
    """ Returns the sum of all arguments  
    >>> add(4, 3)  
    7  
    >>> add(10, 20, 30)  
    60  
    """  
    total = 0  
    for num in args:  
        total += num  
    return total
```

```
def mul(*args):  
    """ Returns the product of all parameters.  
    >>> mul(4, 3)  
    12  
    """  
    total = 1  
    for num in args:  
        total *= num  
    return total  
  
def div(x, z):  
    """ Returns the result of x divided by z,  
    as a float to 3 decimal places.  
    >>> div(4, 3)  
    1.333  
    """  
    return float(f"{x/z:.3f}")
```

12

You can also embed USAGE examples of each function in the docstrings. These examples are in the same format as the calls you would make at the REPL (>>>) prompt and include input parameters and expected output. The function must be deterministic which means it should always return the same output for the same inputs.

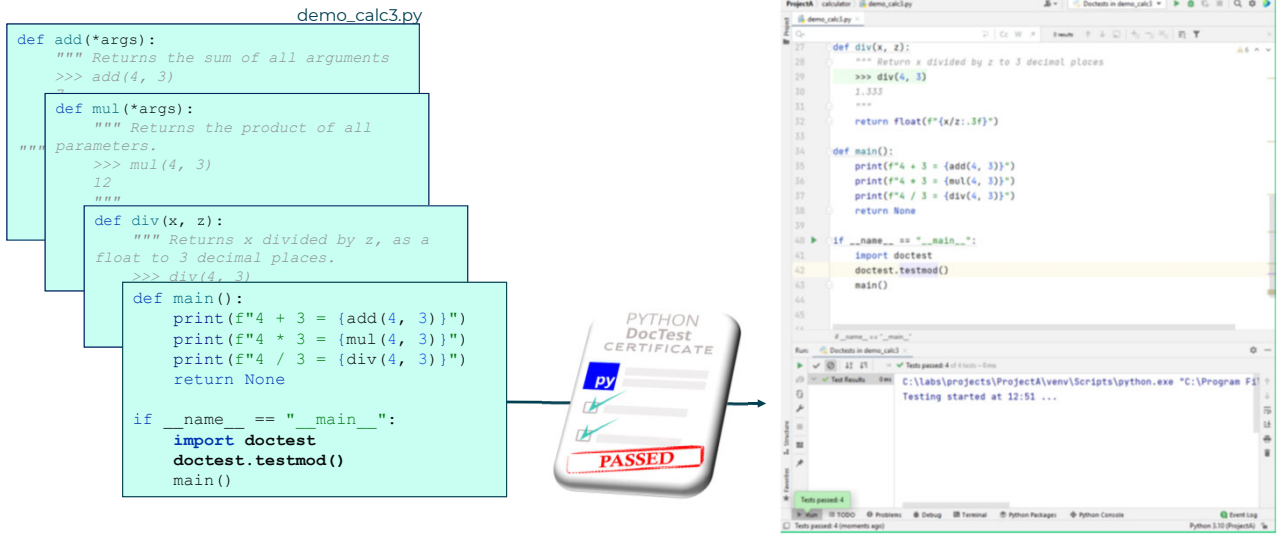
This helps the user understand how to call the function and what is expected out for given parameters, and how to manually test the function is working and matches the documentation.

But there is another benefit of USAGE examples in docstrings..

## QA Docstrings – automated testing

Docstring USAGE examples testing can be automated:

- Import the **doctest** standard library module.



The **doctest** module provides a framework for simple and quick automation of acceptance tests for integration and system testing. These are important as integration testing are used to confirm that the different components of your application all work together. And system testing is used to confirm that you have ticked off the specifications for your project.

Docstrings and doctests can be embedded at the package level down to the class, function and method level. At the higher package level, they can be used for integration testing and at the lower function level they are suitable for unit testing – check the code and documentation as you code!

And the document tests can even be written before the code! How about that for a good idea!

# QA Docstrings – automated testing

Docstring USAGE examples testing can be automated:

- Apply an error in your docstring and test again.

The image illustrates a workflow for automated testing of docstrings. On the left, a code editor shows the `demo_calc3_errors.py` file with three functions: `add`, `mul`, and `div`. The `add` function's docstring is edited to return 8 instead of the sum. The `div` function's docstring is edited to return 1.334 instead of the division result. A 'PYTHON DocTest CERTIFICATE FAILED' badge is shown in the center. On the right, a screenshot of a Python IDE shows the test results for `demo_calc3_errors.py`. The output highlights the failures for `add(4, 3)` and `div(4, 3)`, showing the expected values (8 and 1.334) and the actual values (8 and 1.333).

```
demo_calc3_errors.py

def add(*args):
    """ Returns the sum of all arguments
    >>> add(4, 3)
    8
    >>> add(10, 20, 30)
    60
    """

def mul(*args):
    """ Returns the product of all arguments.
    >>> mul(4, 3)
    12
    """

def div(x, z):
    """ Returns x divided by z, as a float
    to 3 decimal places.
    >>> div(4, 3)
    1.334
    """
    return x / z

def main():
    print(f"4 + 3 = {add(4, 3)}")
    print(f"4 * 3 = {mul(4, 3)}")
    print(f"4 / 3 = {div(4, 3)}")
    return None

if __name__ == "__main__":
    import doctest
    doctest.testmod()
    main()
```

PYTHON DocTest CERTIFICATE FAILED

Run demo\_calc3\_errors.py

```
def add(*args):
    """ Return sum of x and z
    >>> add(4, 3)
    8
    >>> add(10, 20, 30)
    60
    """
    total = 0
    for num in args:
        total += num
    return total

def mul(*args):
    """ Returns the product of all arguments.
    >>> mul(4, 3)
    12
    """
    product = 1
    for num in args:
        product *= num
    return product

def div(x, z):
    """ Returns x divided by z, as a float
    to 3 decimal places.
    >>> div(4, 3)
    1.334
    """
    return x / z

def main():
    print(f"4 + 3 = {add(4, 3)}")
    print(f"4 * 3 = {mul(4, 3)}")
    print(f"4 / 3 = {div(4, 3)}")
    return None

if __name__ == "__main__":
    import doctest
    doctest.testmod()
    main()
```

Run demo\_calc3\_errors.py

```
File "C:\labs\projects\ProjectA\calculator\demo_calc3_errors.py", line 7, in ...
Failed example:
add(4, 3)
Expected:
8
Got:
8

File "C:\labs\projects\ProjectA\calculator\demo_calc3_errors.py", line 29, in ...
Failed example:
div(4, 3)
Expected:
1.334
Got:
1.333
```

In this example, we apply two errors to the DocStrings and re-run the automated testing.

You will notice in the output that Expected and Actual values are highlighted and a summary of how many failures occurred. This can happen when code is changed and the DocStrings are not updated accordingly.



## TESTING USING UNITTEST

### Unit testing:

- Unit vs integration testing.
- Assertions.
- Using the **unittest** module.



15

Testing your documentation versus the code is one form of testing. But how do test that our application is working completely and do the individual features all work correctly. In this part, we will examine integration testing versus unit testing, and will discover how simple it is to perform unit testing using the Python library module **unittest**.

## QA Integration vs unit testing

How to diagnose a complex application and project?

### Integration testing

- 2<sup>nd</sup> level of testing.
- Testing the components work as a complete entity.
- Sometimes called black box testing and often performed by a test team.



### Unit testing

- 1<sup>st</sup> level of testing.
- Testing individual units of code (classes, functions, and methods).
- Sometimes called white box testing and performed by the coder.



**Test step – call a function with parameters.**

**Test Assertion – check that it performs correctly.**

16

When we purchase a new car, we all hope that the manufacturer has tested our car in readiness for it to be driven out the showroom. The manufacturer will have tested all components individually to check that they all work – this is called unit testing. They will also have performed additional tests to check that all the components when assembled work together – this is called integration testing.

Integration testing can be more problematic. For example, if the engine does not start when you turn the key (or press a start button), then many things could have failed. It could be the battery in the key fob, the car battery, the starter motor, the alternator, fuel, or electrical wiring problems.

The developer typically tests their units of code (classes, functions, methods) during the coding phase of your application. Whilst the integration testing is often managed by a testing team using scripts and tools.

In the testing world, if you turn to start the engine, this is known as



a Test Step, and if you hear the motor turning and the lights in the console switching on this is called a Test Assertion.

Modern cars will normally inform you if something is not working correctly – this is an example of a unit test built-in to the car's software. For example, you may be informed that you have low fuel or battery level.

## QA Unit testing – using assert

Python has a built-in function called `assert()` for performing unit testing.

### **assert**

- Convenient method of inserting debugging assertions.
- Raises an exception based on a Boolean expression.
- An `AssertionError` is raised if boolean is `False`.
- Can have an optional expression.

```
assert expression1 [, expression2]
```

```
>>> assert demo_calc.add(4, 3) == 7, "Should be 7"
```

Can be ignored when optimisation is requested at runtime by:

- `PYTHONOPTIMIZE = 0`
- Command line option, `python -O`

```
If __debug__:  
    assert add(4, 3) == 7, "Should be 7"
```

17

Python supports tools and modules for performing Integration and Unit testing, but in this session we will be focusing on performing unit testing.

The Python built-in `assert()` function has been inspired by the equivalent function in C/C++ although it has a different behaviour. It is designed to provide sanity testing within code rather than a full-blown test function. It is designed to be used during development so that the program would fail and generate an `AssertionError` if an expression fails. Probably not what the end-user would like to see in production code. So either, remove from production code or test the `__debug__` special attribute before using the `assert` statement.

The `__debug__` attribute can be altered by enabling the compiler for optimization by setting the `PYTHONOPTIMIZE=0` environment variable or with a command line switch (`-O`).

## QA Unit testing

test\_calc.py

### Test cases:

- Write your test cases in a separate file.
- Keeps code and testing separate.
- Could write tests before the code.
- Downside is only the first failure is displayed.

### Test runners:

- Specialised tools for running tests.
- Checking output.
- Debugging and diagnostic tools.
- **unittest** and **pytest** are popular.

```
"""    Test Cases for Calculator app.: """
from calculator import demo_calc

def test_add():
    assert demo_calc.add(4, 3) == 7, "Error should return 7"
    assert demo_calc.add(10, 20, 30) == 60, "Error should
return 60"
    return None

def test_mul():
    assert demo_calc.mul(4, 3) == 12, "Error should return 12"
    return None

def test_div():
    assert demo_calc.div(4, 3), "Error should return '1.333'"
    return None

def main():
    """    Execute Test functions """
    test_add()
    test_mul()
    test_div()
    print("Everything passed")
    return None

if __name__ == "__main__":
    main()
```

18

Rather than keeping your assert statements in your production code, it may be wiser to have them in a separate Python file. These are called a **test case**, and you could put all the test cases for your program in this file. This method of structuring your tests is great for simple checks, but what if multiple tests fail – only the first one would be displayed.

The solution is to use a **test runner**, this is a special application for running tests, checking their output and provides additional debugging and analysis tools.

There are several test runner modules to choose from including unittest, pytest and nose2. In this session we will investigate using unittest and pytest.

## QA Test runners – unittest

The **unittest** module:

- Part of Python standard library.
- Provides a testing framework and test runner.
- Popular in commercial and open-source projects.

### unittest

- Tests to be defined as methods in a class.
- Requires some knowledge of OOP.
- Uses special assertion methods.
- Inherit from `unittest.TestCase`.

tests\test\_demo\_calc.py

```
import unittest
from calculator import demo_calc

class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(demo_calc.add(4, 3), 7, "Should be 7")
        self.assertEqual(demo_calc.add(4.2, 3.5), 7.7, "Should be 8.7")
        return None

    def test_mul(self):
        self.assertEqual(demo_calc.mul(4, 3), 12, "Should be 12")
        self.assertEqual(demo_calc.mul(102, 3, -2), -612, "Should be -612")
        return None

    def test_div(self):
        self.assertEqual(demo_calc.div(4, 3), 1.333, "Should be 1.333")
        return None

if __name__ == "__main__":
    unittest.main()
```

The unittest module has been a member of the Python standard library since 2.7 and is popular for providing a testing framework and test runner for commercial and open-source projects.

It does require a little knowledge of OOP as the tests are defined as methods in a class. The class must inherit from the **unittest.TestCase** class imported from the module **unittest**. This new class inherits special assertion methods for writing test cases.

The previous test cases can be converted using the following simple steps:

1. Import unittest
2. Create a new class TestCalc, that inherits from unittest.TestCase class
3. Change built-in assert statements to unittest assertions
4. Prefix assertion statements with self (it's an OOP thing)
5. Execute unittest.main() rather than main()

## QA Test runners – unittest and assertions

### unittest assertions:

- Numerous methods to assert test on:
  - values.
  - types.
  - existence of variables.
- Name file starting with 'test'.

| Method                  | Equivalent to    |
|-------------------------|------------------|
| .assertEqual(a, b)      | a == b           |
| .assertTrue(x)          | bool(x) is True  |
| .assertFalse(x)         | bool(x) is False |
| .assertIs(a, b)         | a is b           |
| .assertIsNone(x)        | x is None        |
| .assertIn(a, b)         | a in b           |
| .assertIsInstance(a, b) | isinstance(a, b) |
| .assertRaises(a, *args) | a == Exception   |

### Writing assertions:

- Tests should be repeatable.
- Deterministic/predictable.
- Relate to your input data.
- In this example, we are testing for the function raising an exception.

```
def div(x, z):  
    """ Returns x divided by z, as a float. """  
    if z == 0:  
        raise ZeroDivisionError("Divisor must be zero")  
    return float(f"{x/z:.3f}")
```

test\_calc4.py

```
def test_div(self):  
    self.assertRaises(ZeroDivisionError, demo_calc.div,  
4, 0)  
    return None
```

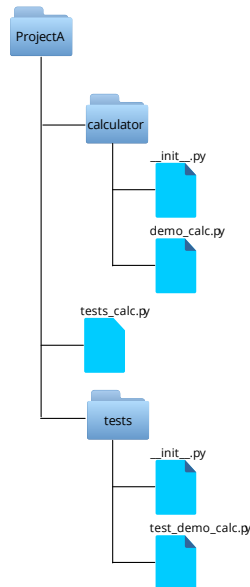
test\_demo\_calc2.py

The unittest module provides many more assertions and features over the built-in `assert()` function. There are assertions for testing that values are Equal or NotEqual, variables exist or of a particular type, are Boolean or raise Exceptions.

Import unittest at the REPL prompt and use the `help(unittest)` function to see the complete list of assertions and how to use them. The slide shows an example of testing whether a `ZeroDivisionError` is raised in your code or not.

When writing test assertions, it is good practice to design them so they are repeatable, deterministic (always the same result for the same input test) and relate to your input data.

## QA Test scripts – location and executing tests



### Standard test cases:

- Located in folder above application folder.
- Needs to be able to import application.
- Name file starting with 'test'.

### Complex test cases:

- Create a sub package called tests.
- Split tests into files starting with 'test\*.py'.

### Running a single test module:

```
C:\...\ProjectA> python -m unittest tests.test_demo_calc
```

### Running a single test case:

```
C:\...\ProjectA> python -m unittest tests.test_demo_calc.test_add
```

### Running all tests:

```
C:\...\ProjectA> python -m unittest discover
```

21

When writing simple test scripts, locate them in the folder above the application folder as they will need to import the application. Give the script a name starting with 'test'.

Once the script becomes too large than it is advisable to create sub package called tests and split the test script into separate files; the convention is to give the file names starting with 'test\_'. The tests package folder should be in your main project folder.

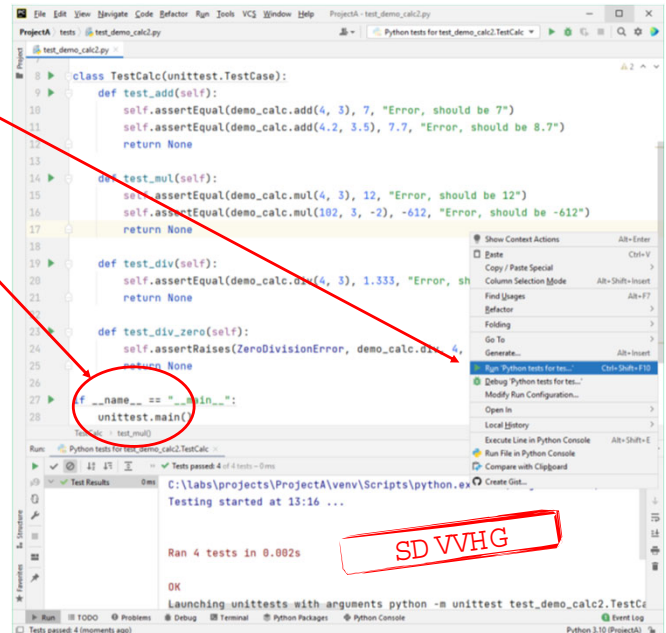
Remember the `__init__.py` file indicates that the calculator and tests folders can be imported as a module from the parent directory.

The test scripts can be executed from the command line in several ways by importing the **unittest** module and then the test module or test function as an argument. Alternatively, if you have named your folder and scripts with the prefix 'test', then you can tell the unittest module to '**discover**' and execute all test folders and scripts.

## QA Test scripts – executing tests in Pycharm

### Executing your test script in Pycharm:

- Right-click > Run Python Tests > script.
- Or Menu > Run > Run Python Tests > script.
- Test runner executes your test code.
- Test result in console.



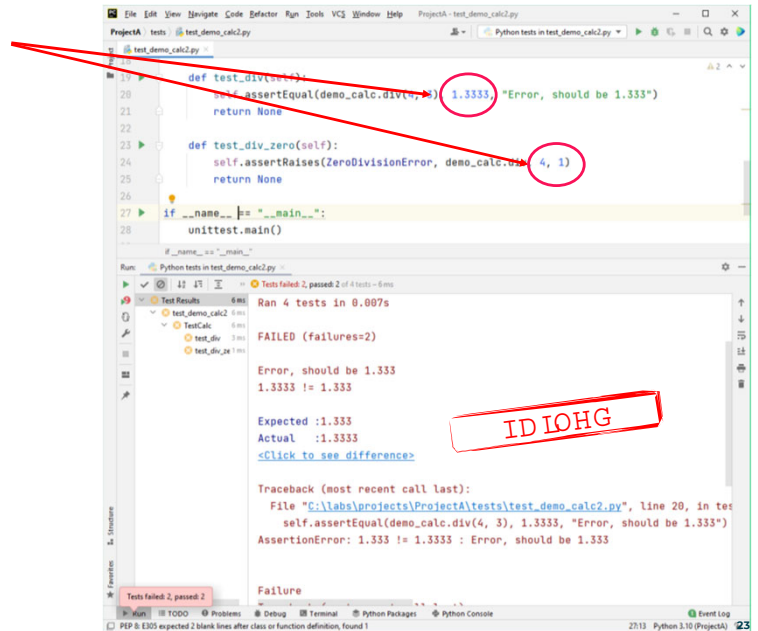
To execute your test script, either right-click and select Run Python Tests, or select Run menu option and click Run Python Tests. This will execute `unittest.main()` which is the Test Runner Application which will interpret and execute your tests and display the results of all tests to the console.

## QA Test scripts – executing tests in Pycharm

Edit your test script to report failures.  
Execute the test script.

In the output:

- Execution results.
- Number of failures.
- Detailed about each failed entry.
- Traceback to failed line of code.
- The assertion and expected and actual result.



Edit your Test script to force it to fail. On the slide we have altered the expected result from the division to be 4 decimal places, rather than 3. And have changed the parameters so that a `ZeroDivisionError` is not raised.

Execute the test runner again and notice the execution results in the console, the number of failures and detail about each failed entry. A traceback to the failed line of code and the expected and actual result of the test will also be displayed. In the navigation test window, you can choose the entire Test Results or individual Test Cases.



## QA Testing using pytest

### Unit Testing:

- Boilerplate code.
- Using the **pytest** module.
- Common data for testing using fixtures.
- Varying data for testing using parametrization.



24

As good as the **unittest** module is, it still has some shortcomings including its interface. There are several alternative testing frameworks and **pytest** is one the most popular. It is feature rich, has additional plug-ins and is more pleasing to use. It can even run your existing tests out of the box including those written using **unittest**.

In this session, we will examine, how to create a test runner using **Pytest**.

## Boilerplate vs elegance

Tests should be readable and consist of minimal information to understand the test case.

### unittest

- Part of the Python standard library.
- Boilerplate code.
- Repeated verbose code.
- Import module, create class, inherit from TestCase.
- Write special methods for each test case.
- Use one of the many self.assert\* methods.

### Pytest

- Must be installed first.
- Simple test functions (no classes or methods required).
- Uses built-in assert() function or any expression that evaluates to True.
- Elegant and simple, and nicer output.

25

In computer programming, boilerplate code is often used to describe code that is verbose in nature, that is repeated often with minimal code. Did you notice anything whilst using unittest? Before we did any assertions, we had to import the **unittest** module, create a new class, inherit from TestCase – and then write a method for each test case using one of the many self.assert\* methods.

And we must do that for each test script! Some would say that is unreadable, inelegant or bordering on boilerplate code!

Alternatively, we could use the **pytest** module that does away with all that and allows us to write simple test functions and use the built-in assert() function or Boolean statements. If you are familiar with assert() then the learning curve is shorter than having to learn and use unittest.

The only convention you must follow is to prefix your test modules with the name 'test'. It also has nicer output!

## QA Installing pytest

**Pytest is not part of the Python standard library so needs to be installed.**

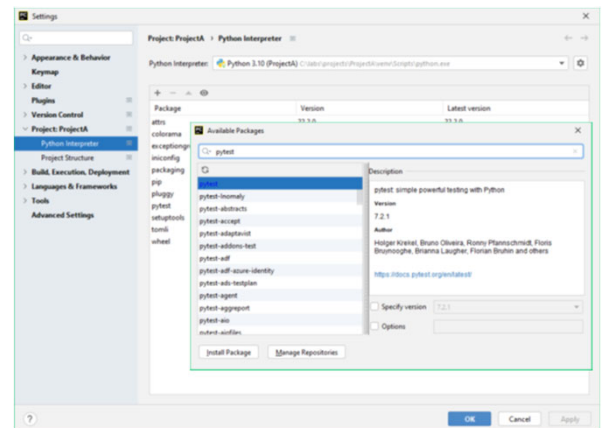
### Command line

- Can be installed in a virtual environment (venv) using pip.
- Change into your project folder.

```
c:\labs\projects\ProjectA> python -m venv venv
c:\labs\projects\ProjectA> .\venv\Scripts\activate
(venv) c:\labs\ProjectA> python -m pip install pytest
```

### Pycharm

- File>Settings>Project>Project Interpreter.
- Select +
- Search pytest.
- Select Install pytest.



Pytest needs to be installed first as it is not in the Python standard library. It can be installed from the command lines using pip or from within Pycharm. It can also be installed in a virtual environment.

Pytest: <https://docs.pytest.org/en/7.2.x/>

As an aside, if you are not familiar with Virtual Environments then ask your instructor or use the links below. A virtual environment is a way to isolate a Python interpreter with its own independent set of library modules and versions; so that different projects can manage their own dependencies.

Several Virtual environments are available including venv (python only), conda (python and other languages) and anaconda (commercial environment and package manager) to choose from.

Venv: <https://docs.python.org/3/library/venv.html>

Conda: <https://docs.conda.io/projects/conda/en/stable/>

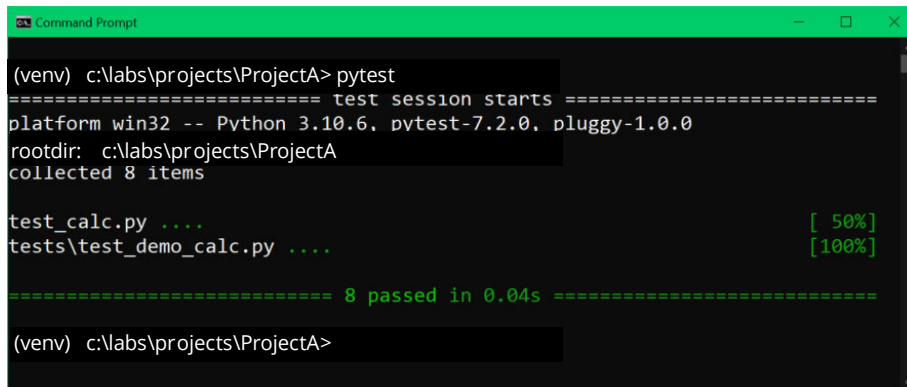
Anaconda: <https://www.anaconda.com/products/distribution>

## QA pytest output

### Command line – reporting success

- Execute pytest in your project folder.
- Discover test scripts.
- Nicer output – dot (test passed), F (Failed), E (Exception).

```
(venv) C:\project> pytest
```



```
Command Prompt

(venv) c:\labs\projects\ProjectA> pytest
===== test session starts =====
platform win32 -- Python 3.10.6, pytest-7.2.0, pluggy-1.0.0
rootdir: c:\labs\projects\ProjectA
collected 8 items

test_calc.py .... [ 50%]
tests\test_demo_calc.py .... [100%]

===== 8 passed in 0.04s =====

(venv) c:\labs\projects\ProjectA>
```

27

Execute pytest at the command line inside your project folder and it will discover all the existing test scripts (with a 'test' prefix).

The report displays the system state, version of Python and pytest and optional plugins. The folder to search for tests and the number of tests discovered.

The output indicates a dot (test passed), an F (test failed) or an E (Exception raised). The overall progress of the test cases are displayed as a percentage.

## QA pytest output

### Reporting failures

```
Command Prompt
platform win32 -- Python 3.10.6, pytest-7.2.0, pluggy-1.0.0
rootdir: c:\labs\projects\ProjectA
collected 8 items

test_calc.py F... [ 50%]
tests\test_demo_calc.py .F.. [100%]

===== FAILURES =====
test_add

def test_add():
    assert demo_calc.add(4, 3) == 7, "Error should return 7"
> assert demo_calc.add(10, 20, 30) == 50, "Error should return 60"
E   AssertionError: Error should return 60
E   assert 60 == 50
E   + where 60 = <function add at 0x0000012E9FEDE7A0>(10, 20, 30)
E   + where <function add at 0x0000012E9FEDE7A0> = demo_calc.add

test_calc.py:13: AssertionError
TestCalc.test_div

self = <tests.test_demo_calc.TestCalc testMethod=test_div>

def test_div(self):
> self.assertEqual(demo_calc.div(4, 3), 1.3333, "Error, should be 1.333")
E   AssertionError: 1.333 != 1.3333 : Error, should be 1.333

tests\test_demo_calc.py:25: AssertionError
===== short test summary info =====
FAILED test_calc.py::test_add - AssertionError: Error should return 60
FAILED tests\test_demo_calc.py::TestCalc::test_div - AssertionError: 1.333 != 1.3333 : Error, should be 1.333
===== 2 failed, 6 passed in 0.06s =====
```

28

Edit your test scripts and add an error and rerun the pytest command. In this instance it displays that there were two failures and gives detailed breakdown of the failure, followed by an overall status report.

## QA pytest output

### Pycharm – reporting SUCCESS:

- Right-click > Run Python Tests > script.
- Or Menu > Run > Run Python Tests > script.
- pytest executes your test code.

The screenshot shows the PyCharm IDE interface. The top pane displays a Python script named `test_calc2.py` with the following content:

```
def main():  
    """ Execute Test functions """  
    test_add()  
    test_mul()  
    test_div()  
    # test_div_zero()  
    print("Everything passed")  
    return None  
  
if __name__ == "__main__":  
    main()
```

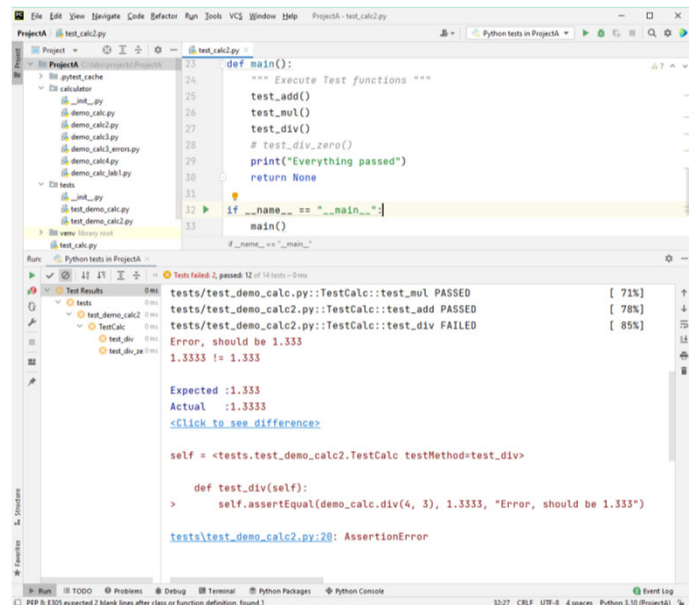
The bottom pane shows the output of running the tests. The output is as follows:

```
Python tests in test_calc2.py  
Tests passed: 4 of 4 tests - 0ms  
Testing started at 14:09 ...  
C:\Program Files\JetBrains\PyCharm Community Edition 2021.2\plugins\python-ce\helpers\distutils import version  
Launching pytest with arguments C:/Labs/projects/ProjectA/test_calc2.py --no-header  
  
===== test session starts =====  
collecting ... collected 4 items  
  
test_calc2.py::test_add PASSED [ 25%]  
test_calc2.py::test_mul PASSED [ 50%]  
test_calc2.py::test_div PASSED [ 75%]  
test_calc2.py::test_div_zero PASSED [100%]  
  
===== 4 passed in 0.02s =====  
  
Process finished with exit code 0
```

## pytest output

### Pycharm – reporting FAILURE:

- Right-click > Run Python Tests > script.
- Or Menu > Run > Run Python Tests > script.
- pytest executes your test code.
- Detailed test report in console.



```
def main():
    """ Execute Test functions """
    test_add()
    test_mul()
    test_div()
    # test_div_zero()
    print("Everything passed")
    return None

if __name__ == "__main__":
    main()
    if __name__ == "__main__":
```

Test Results

| Test   | Result | Percentage |
|--|--------|------------|
| tests/test_demo_calc.py::TestCalc::test_mul  | PASSED | [ 71%]     |
| tests/test_demo_calc2.py::TestCalc::test_add | PASSED | [ 78%]     |
| tests/test_demo_calc2.py::TestCalc::test_div | FAILED | [ 85%]     |

Error, should be 1.333

Expected :1.333

Actual :1.3333

<Click to see difference>

self = <tests.test\_demo\_calc2.TestCalc testMethod=test\_div>

```
def test_div(self):
    self.assertEqual(demo_calc.div(4, 3), 1.3333, "Error, should be 1.333")
```

tests/test\_demo\_calc2.py:20: AssertionError

When tests fail, Pytest will generate a detailed report in the console. In this slide, we have added two obvious errors to the test\_add() and test\_div() and both are reported followed by a summary that 2 failed and 6 tests passed.

Pytest can run tests in parallel and can also filter tests by directory, pattern matching the filenames or by category of tests.



## SUMMARY

### **Software testing is important!**

- Bugs are expensive to fix in production.
- Could save reputation and lives.
- Could improve performance and security.

### **Tests can be manual or automated**

- Automated is better.

### **Document your code**

- Use DocStrings and Doctest module.

### **Useful Testing Frameworks**

- Write test cases.
- Create test runners.
- Unittest and pytest are popular.

### **Software testing is important!**



## QA Managing test data - fixtures

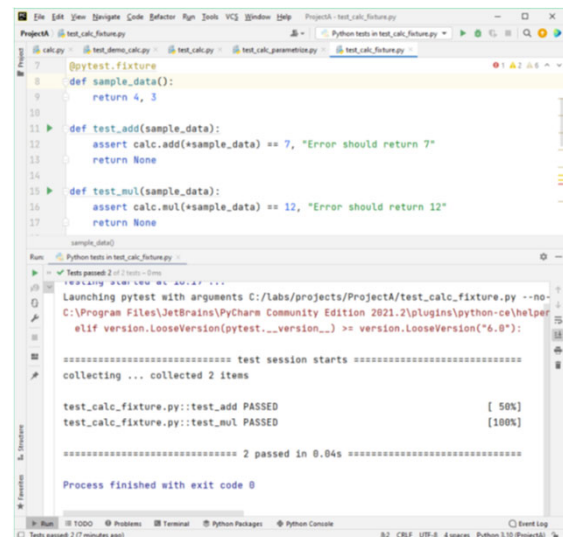
### Pytest fixture

- Special function decorated with **@pytest.fixture**.
- Provides data or test double to the test function.
- Test function accepts fixture as a parameter.
- Can simplify multiple tests that use same data.
- Can be centralised in a module called **conftest.py**.

```
@pytest.fixture
def sample_data():
    return 4, 3

def test_add(sample_data):
    assert calc.add(*sample_data) == 7, "Error should return 7"
    return None

def test_mul(sample_data):
    assert calc.mul(*sample_data) == 12, "Error should return 12"
    return None
```



32

If you have many tests that share the same test data, then you can create a special function in **pytest** called a fixture. Fixtures are functions that can return a large range of values for your test functions. Like a stunt double, they provide test doubles to your test function, and the test function accepts them explicitly as parameters. In some cases, they can simplify multiple test functions and reduce the amount of boilerplate code.

If you want to make a fixture available to your entire project, then you can place them in a special module called **conftest.py**. Pytest looks for this module in each directory, but if placed in the project parent directory then it will be available to the parent and all sub directories without importing it.

## QA Managing multiple test data - parametrized

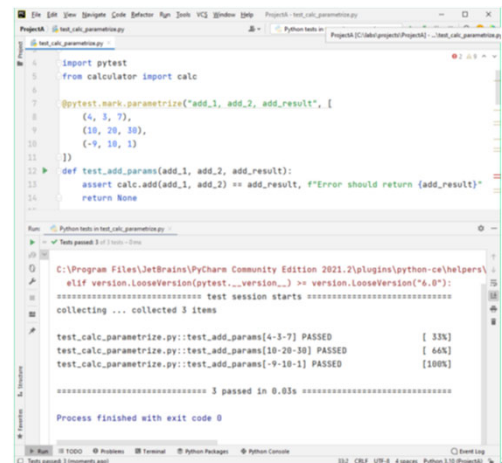
### Pytest – combining tests

- Solution to different inputs and outputs.
- Generalises the code.
- Scales better.
- Uses `@pytest.mark.parametrize()`.

`@pytest.mark.parametrize("param1, param2, result", [ values, values ])`

```
import pytest
from calculator import calc

@pytest.mark.parametrize("add_1, add_2, add_result", [
    (4, 3, 7),
    (10, 20, 30),
    (-9, 10, 1)
])
def test_add_params(add_1, add_2, add_result):
    assert calc.add(add_1, add_2) == add_result, f"Error should return {add_result}"
    return None
```



Fixtures can be useful to reduce duplication in code but are limited to common parameters and outputs. If you have test data with slightly different inputs and outputs, then **pytest** allows you to combine and generalise the test data using `parametrize()`.

The first parameter to the `parametrize()` function is a comma-delimited string of parameter names which incorporates the input data and expected output. The second parameter is a list or tuple of the values to be tested.

A note of caution. Do not over complicate your parametrization so that it tends to boiler code. Test should be clear and simple.