# Getting started

Python includes a really useful module in its standard library that is called **doctest**. The doctest module will look at your code and determine if you have any text that looks like an interactive Python session. Then doctest will execute those strings to verify that they work the way they are written.

According to the Python documentation, doctest has three primary uses:

> - Checking that a module's docstrings are up-to-date by verifying the examples in the docstrings execute correctly
> - Do regression testing using the interactive examples in the text or
> - Write tutorial documentation for your package or module. You can do this by adding lots of examples in your docstrings that show inputs and outputs.

Let's create a simple addition function that we can use to demonstrate what you might do with doctest. Save the following code into a file named **mymath.py**:

```
def add(a, b):
    """
    Return the addition of the arguments: a + b

    add(1, 2)
    #3
    add(-1, 10)
    #9
    add('a', 'b')
    #'ab'
    add(1, '2')
    #Traceback (most recent call last):
    #  File "test.py", line 17, in <module>
    #    add(1, '2')
    #  File "test.py", line 14, in add
    #    return a + b
    #TypeError: unsupported operand type(s) for +: 'int' and 'str'
    """
```

```
    return a + b

if __name__ == '__main__':

    import doctest
    doctest.testmod()
```

You will notice that we have a very large docstring here with what appears to be a session from the Python interpreter. This is exactly what you want to do when you write your docstrings for doctest. You can run your function in the Python interpreter or IDLE and test it out. Then you can take some of the examples you tried along with its output and add it to your function's docstring. At the very end of this example, we import doctest and call its **testmod** function. The testmod() call will then search all the docstrings in the module and try to run the examples that look like interpreter sessions.

Let's try running this code to see what kind of output we get.

```
mike@mac: python test.py
```

You will notice that all we need to do is call Python and pass it our test script. When you run this code, you shouldn't see any output whatsoever. That means the test passed. If you want to actually see some output, then you will need to enable verbose mode by passing in **-v**:

```
mike@mac-028: python3 test.py -v
Trying:
    add(1, 2)
Expecting:
    3
ok
Trying:
    add(-1, 10)
Expecting:
    9
ok
Trying:
    add('a', 'b')
Expecting:
    'ab'
ok
Trying:
    add(1, '2')
Expecting:
    Traceback (most recent call last):
      File "docked.py", line 17, in <module>
        add(1, '2')
      File "docked.py", line 14, in add
        return a + b
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   4 tests in __main__.add
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

As the last line indicates, our test ran 4 tests and they all passed. There's another way to call doctest on a module that is only on the command line. You will need to modify the example from earlier to remove the if statement at the end so that your script now only contains the function. Then you can run the following:

```
python -m doctest -v test.py
```

This command tells Python to run the doctest module in verbose mode against the script we created.