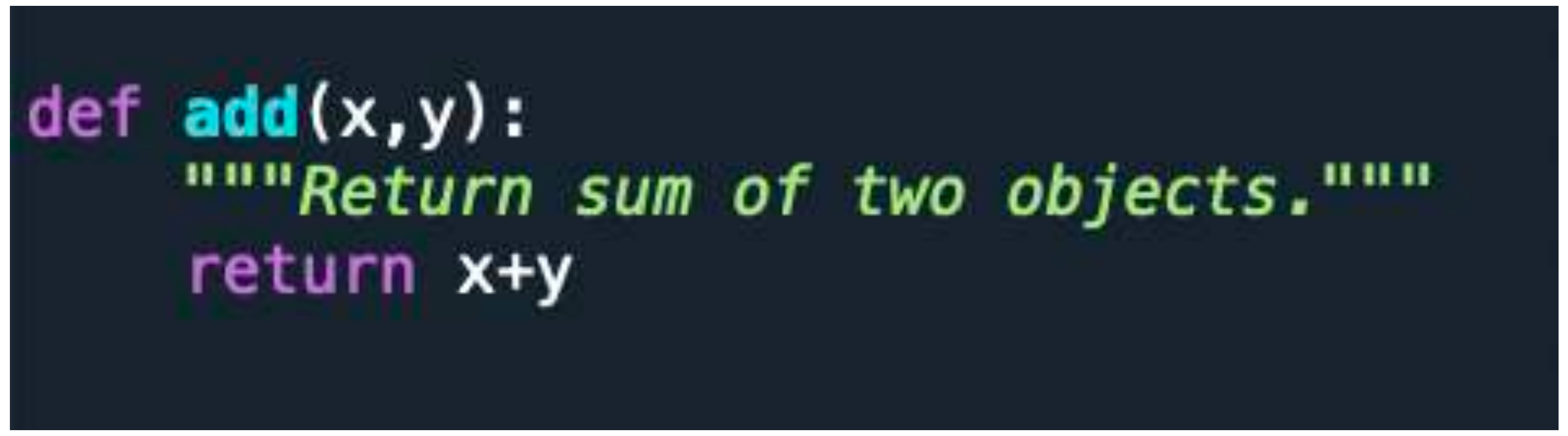
Unit Testing

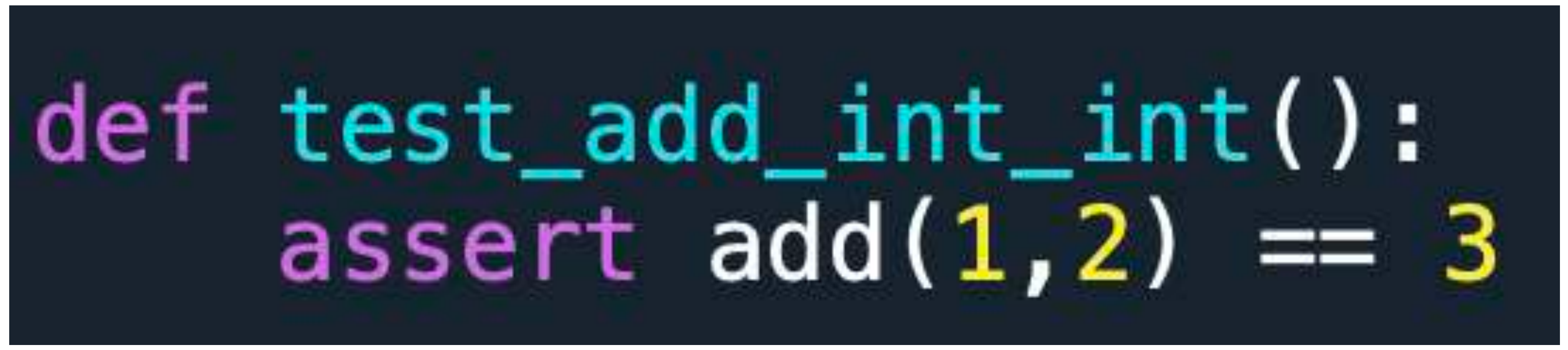
Unit tests

Check if a single unit of code works as expected/desired

They should be small, precise, and independent



```
def add(x,y):
    """Return sum of two objects."""
    return x+y
```



```
def test_add_int_int():
    assert add(1,2) == 3
```

Why Unit Tests?

- **Fix bugs** + Make sure they are not reproduced
- Helps with refactoring
- Like a documentation (but it is compiled/interpreted)

UT frameworks

- **Python**: pytest, nose, doctest, unittest
- **C++**: Catch, Google Test, Boost.Test, CppUnit, ...
- Get tools to make things easier (automation, reports, fixtures, ...)

Moc Objects

- Commonly used in testing OO code
- Create objects that are difficult include
  - Non-controlled or non-deterministic behaviour (current time, current temperature, ...)
  - State difficult to reproduce (network error, large database, ...)

Test fixtures

- Set up (preconditions)
- Assert
- Tear down (postconditions)

Hands on - Unit Test 1

- Standalone
- Capitalise

Test Driven Development

- Write unit tests that fails
- Write the minimum (sensible) code to pass them
- Refactor

full test coverage and less useless code

Hands on - Unit Test 1

- Factorial
- Accumulator

Best practices

- If you find a bug, turn it into a test case
- When debugging, write tests
- Always leave the code in a better state than you found it in

Hands on - add fixture to the palindrome tests

Hands on - Wallet test

Hands on - Code review tests