Testing with pytest!

Big Idea: You can write a function to *test* the correctness of another function!

- This is generally called *unit testing* in industry
  - Helps you confirm correctness during development
  - Helps you avoid accidentally breaking things that were previously working

- The strategy:
  1. Implement the "skeleton" of the function you are working on
     - Name, parameters, return type, and some dummy (wrong/naive!) return value
  2. Think of examples use cases of the function and what you expect it to return in each case
  3. Write a test function that makes the call(s) and compares expected return value with actual
  4. Once you have a failing test case running, go correctly implement the function's body
  5. Repeat steps #3 and #4 until your function meets specifications

- This gives you a framework for knowing your code is behaving as you expect

Example: Writing and Testing a **total** Function (1/2)

Let's write a function to add up all elements of a float list!

Step 0) Implement the function skeleton:

```
def total(xs: List[float]) -> float:
    return -1.0 # return a dummy value (wrong but correct type)
```

Step 1) Think of some example uses...

total([1, 2, 3]) *should* return **6.0**

total([110]) *should* return **110.0**

total([]) *should* return **0.0**

Setting up a **pytest** Test Module

- To test the definitions of a module, first create a sibling module with the same name, but ending in **_test**
  - Example name of definitions module: **lessons.ls24_module**
  - Example name of tests module: **lessons.ls24_module_test**
  - This convention is common to pytest

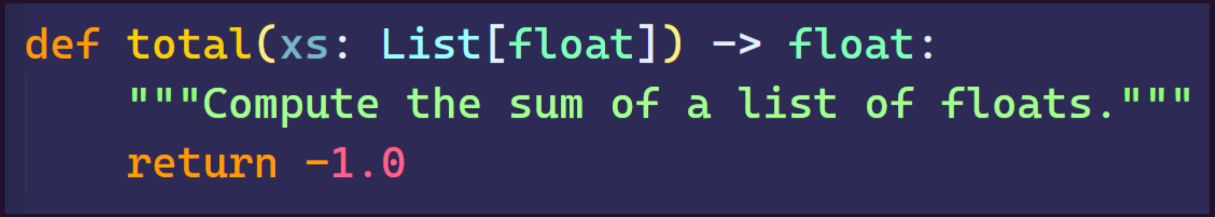
- Then, In the test module, import the definitions you'd like to test

- Next, add tests which are procedures whose names *begin* with **test_**
  - Example test name: test_total_empty

- To run the test(s), two options:
  1. In a new terminal: **python -m pytest [package_folder/python_module_test.py]**
  2. Use the Python Extension in VSCode's Tests Pane

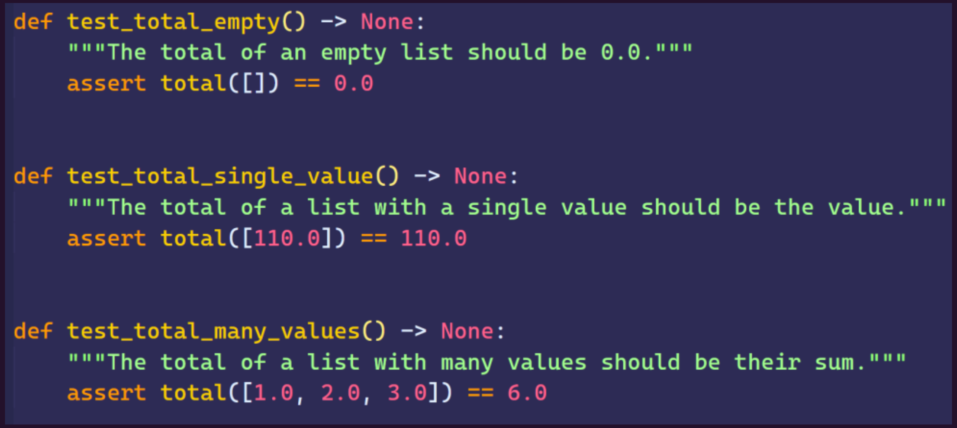
Follow-Along: Testing **`total`**

- Let's implement a function to sum the elements of an array
- Function Skeleton:



```python
def total(xs: List[float]) -> float:
    """Compute the sum of a list of floats."""
    return -1.0
```

- What are our test cases?



```python
def test_total_empty() -> None:
    """The total of an empty list should be 0.0."""
    assert total([]) == 0.0


def test_total_single_value() -> None:
    """The total of a list with a single value should be the value."""
    assert total([110.0]) == 110.0


def test_total_many_values() -> None:
    """The total of a list with many values should be their sum."""
    assert total([1.0, 2.0, 3.0]) == 6.0
```

Test-driven Function Writing

- **Before you implement a function**, focus on concrete examples of *how the function should behave as if it were already implemented.*

- Key questions to ask:

1. **What are some *usual* arguments?**
   - These are called *use cases.*

2. **What are some valid but *unusual* arguments?**
   - These are your *edge cases.*

3. Given those arguments,
   what is your <u>expected</u> return value for each set of inputs?

Test-Driven Programming: Case Study **join**

- Suppose you want to write a function named **`join`**

- Its purpose is to make a string out of an int list **`xs's`** values where each element is separated by some delimiter.

  Example: joining xs with values [1, 2, 3] and delimiter "-" returns "1-2-3"

- Its signature is this: **`def join(xs: List[int], delimiter: str) -> str`**

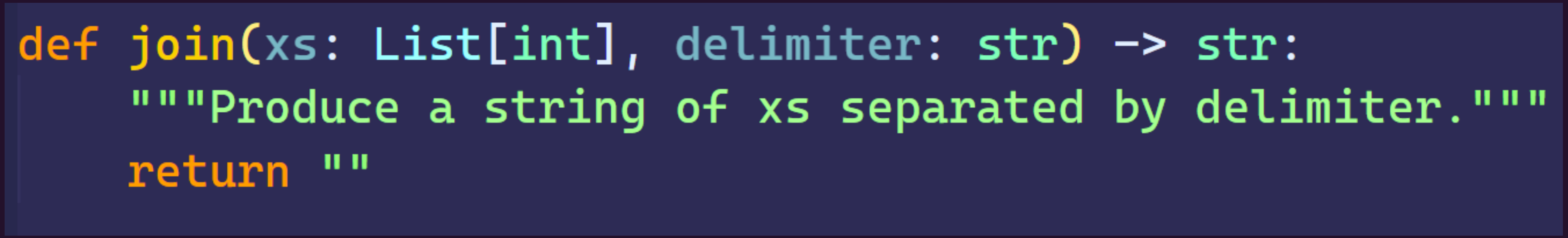
1. **What are some *usual* input parameters?**
   - These are called *use cases.*

2. **What are some valid but *unusual* input parameters?**
   - These are your *edge cases.*

3. Given those input parameters,
   what is your <u>expected</u> return value for each set of inputs?

Testing Use/Edge Cases Programmatically

- After you have some use and edge cases, implement the skeleton of the function that is *syntactically valid* but *intentionally incomplete*
  - Typically this means define the function and do nothing inside of the body except return a valid literal value. For example:



```python
def join(xs: List[int], delimiter: str) -> str:
    """Produce a string of xs separated by delimiter."""
    return ""
```

- Then, turn your use and edge cases into programmatic tests.

Testing is no substitute for critical thinking…

- Passing your own tests doesn't ensure your function is correct!
  - Your tests must cover a useful range of cases

- Rules of Thumb:
  - Test 2+ use cases and 1+ edge cases.
  - When a function has if-else statements, try to write a test that reaches each branch.