UNIT TESTS

An Introduction to Computer Science

Let's learn about Unit Tests.

Unit Tests

Make it easier to...

1. Find problems earlier
2. Change things later
3. Glue together code

Instead, programmers use collections of **Unit Tests** to determine if their code works as expected.
Unit testing is an important professional practice to help you find problems early, facilitate changes down the road, and make it easier to glue together pieces of code.

Example Unit Test

```
from cisc108 import assert_equal

def feet_to_inches(distance:int)->int:
    return distance * 12

assert_equal(feet_to_inches(2), 24)
```

Test module

Function

Unit Test

A unit test is pretty simple: call a function with a known input and compare the result to a known output.
For example, consider this function that converts feet to inches.
Above the function call is a special statement that imports, or makes available, the assert_equal function from the test module.
We use this function on the line below to write our unit test.
We call feet_to_inches and pass in 2, and assert that we expect a result of 24.

Representative Cases

	Input	→	Output
Positive Integer	1	→	12
Positive Float	.5	→	6.0
Zero	0	→	0
Negative Integer	-4	→	-48
Large Positive Integer	10000	→	120000

A tricky part of making unit tests is thinking of appropriate test cases. The goal is to cover many possible scenarios, while writing the minimum number of cases. In our conversion function before, we tried it with a small positive number, a decimal number, a negative number, a large number, and zero. These are all representative cases. Depending on what our function is doing, you might need more or less test cases.

Code Coverage

```
from cisc108 import assert_equal

def feet_to_inches(distance:int)->int:
    return distance * 12

assert_equal(feet_to_inches(2), 24)
assert_equal(feet_to_inches(.5), 6.0)
assert_equal(feet_to_inches(0), 0)
assert_equal(feet_to_inches(-4), -48)
assert_equal(feet_to_inches(10000), 120000)
```

When you write many unit tests, you increase the coverage of your code.
Coverage is usually measured by the number of lines of code that are executed, divided by the number of lines of code you've written.
Right now, as long as you write any unit tests, you'll be able to get good code coverage.
Soon, we will see that you will need to write more unit tests to handle loops and conditionals.

Failing Unit Tests

Function: `inches_to_feet()`

Given input: `1`

Expected output: `12`

Actual output: `-5`

`12 != 5`, you have an error in your calculation!

When a unit test fails, it should report the actual and expected values, so that the programmer can then debug their program.
If you only try a function with one output, you might think that the function is correct. But until you run unit tests, you shouldn't believe that.
Of course, unit tests are not conclusive either, they are merely evidence to support the hypothesis that your code is correct.

Passing Unit Tests

Feedback: Incorrect Answer

Trace Variables

Instructor Feedback

Your `make_polite` function did not produce the correct output for the values `''`.

Expected: `', please'`

Actual: `'No'`

The larger your program, the more important it is to unit test individual pieces.
However, unit tests are also helpful as you learn the basics.
Every program you write in this course will be unit tested, which means that we will try running your program against many inputs and outputs.
Sometimes you will be asked to write your own unit tests, and sometimes we will run our own unit tests on your code.
Just because your program appears to give the right output, you must pass all the unit tests!

Judging Unit Tests

```python
from cisc108 import assert_equal

def add(left: int, right: int) -> int:
    return left + 4

# Valid tests shows the correct behavior
assert_equal(add(1, 4), 5)
assert_equal(add(3, 4), 7)

# Thorough tests expose the incorrect version's failure
assert_equal(add(4, 3), 7)
assert_equal(add(5, 5), 10)
```

Your collection of unit tests need to be **valid** and **thorough**.
A collection is valid if the assertions pass for all possible correct implementations of the program.
A collection is thorough if the assertions fail for any possible incorrect implementations.
Although it is impossible to consider every possible way to write a program, you should spend some time thinking about alternatives to your solution.
It can be very difficult to think of test cases that break your program, but doing so is critical to writing good tests!