Programming with Python

Defensive Programming

Learning Objectives

- Explain what an assertion is.
- Add assertions that check the program's state is correct.
- Correctly add precondition and postcondition assertions to functions.
- Explain what test-driven development is, and use it when creating new functions.
- Explain why variables should be initialized using actual data values rather than arbitrary constants.

Our previous lessons have introduced the basic tools of programming: variables and lists, file I/O, loops, conditionals, and functions. What they *haven't* done is show us how to tell whether a program is getting the right answer, and how to tell if it's *still* getting the right answer as we make changes to it.

To achieve that, we need to:

- Write programs that check their own operation.
- Write and run tests for widely-used functions.
- Make sure we know what "correct" actually means.

The good news is, doing these things will speed up our programming, not slow it down. As in real carpentry — the kind done with lumber — the time saved by measuring carefully before cutting a piece of wood is much greater than the time that measuring takes.

Assertions

The first step toward getting the right answers from our programs is to assume that mistakes *will* happen and to guard against them. This is called defensive programming, and the most common way to do it is to add assertions to our code so that it checks itself as it runs. An assertion is simply a statement that something must be true at a certain point in a program. When Python sees one, it evaluates the assertion's condition. If it's true, Python does nothing, but if it's false, Python halts the program immediately and prints the error message if one is provided. For example, this piece of code halts as soon as the loop encounters a value that isn't positive:

```
numbers = [1.5, 2.3, 0.7, -0.001, 4.4]
total = 0.0
for n in numbers:
    assert n > 0.0, 'Data should only contain positive values'
    total += n
print('total is:', total)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-19-33d87ea29ae4> in <module>()
      2 total = 0.0
      3 for n in numbers:
----> 4     assert n > 0.0, 'Data should only contain positive values'
      5     total += n
      6 print('total is:', total)

AssertionError: Data should only contain positive values
```

Programs like the Firefox browser are full of assertions: 10-20% of the code they contain are there to check that the other 80-90% are working correctly. Broadly speaking, assertions fall into three categories:

- A precondition is something that must be true at the start of a function in order for it to work correctly.
- A postcondition is something that the function guarantees is true when it finishes.
- An invariant is something that is always true at a particular point inside a piece of code.

For example, suppose we are representing rectangles using a tuple of four coordinates `(x0, y0, x1, y1)`, representing the lower left and upper right corners of the rectangle. In order to do some calculations, we need to normalize the rectangle so that the lower left corner is at the origin and the longest side is 1.0 units long. This function does that, but checks that its input is correctly formatted and that its result makes sense:

```
def normalize_rectangle(rect):
    '''Normalizes a rectangle so that it is at the origin and 1.0 units long on its
longest axis.'''
    assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
    x0, y0, x1, y1 = rect
    assert x0 < x1, 'Invalid X coordinates'
    assert y0 < y1, 'Invalid Y coordinates'

    dx = x1 - x0
    dy = y1 - y0
    if dx > dy:
        scaled = float(dx) / dy
        upper_x, upper_y = 1.0, scaled
    else:
        scaled = float(dx) / dy
        upper_x, upper_y = scaled, 1.0

    assert 0 < upper_x <= 1.0, 'Calculated upper X coordinate invalid'
    assert 0 < upper_y <= 1.0, 'Calculated upper Y coordinate invalid'

    return (0, 0, upper_x, upper_y)
```

The preconditions on lines 2, 4, and 5 catch invalid inputs:

```
print(normalize_rectangle( (0.0, 1.0, 2.0) )) # missing the fourth coordinate
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-21-3a97b1dcab70> in <module>()
----> 1 print(normalize_rectangle( (0.0, 1.0, 2.0) )) # missing the fourth coordina
te

<ipython-input-20-408dc39f3915> in normalize_rectangle(rect)
      1 def normalize_rectangle(rect):
      2     '''Normalizes a rectangle so that it is at the origin and 1.0 units lon
g on its longest axis.'''
----> 3     assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
      4     x0, y0, x1, y1 = rect
      5     assert x0 < x1, 'Invalid X coordinates'

AssertionError: Rectangles must contain 4 coordinates
```

```
print(normalize_rectangle( (4.0, 2.0, 1.0, 5.0) )) # X axis inverted
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-22-f05ae7878a45> in <module>()
----> 1 print(normalize_rectangle( (4.0, 2.0, 1.0, 5.0) )) # X axis inverted

<ipython-input-20-408dc39f3915> in normalize_rectangle(rect)
      3     assert len(rect) == 4, 'Rectangles must contain 4 coordinates'
      4     x0, y0, x1, y1 = rect
----> 5     assert x0 < x1, 'Invalid X coordinates'
      6     assert y0 < y1, 'Invalid Y coordinates'
      7

AssertionError: Invalid X coordinates
```

The post-conditions help us catch bugs by telling us when our calculations cannot have been correct. For example, if we normalize a rectangle that is taller than it is wide everything seems OK:

```
print(normalize_rectangle( (0.0, 0.0, 1.0, 5.0) ))
```

```
(0, 0, 0.2, 1.0)
```

but if we normalize one that's wider than it is tall, the assertion is triggered:

```
print(normalize_rectangle( (0.0, 0.0, 5.0, 1.0) ))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-24-5f0ef7954aeb> in <module>()
----> 1 print(normalize_rectangle( (0.0, 0.0, 5.0, 1.0) ))

<ipython-input-20-408dc39f3915> in normalize_rectangle(rect)
     16
     17     assert 0 < upper_x <= 1.0, 'Calculated upper X coordinate invalid'
---> 18     assert 0 < upper_y <= 1.0, 'Calculated upper Y coordinate invalid'
     19
     20     return (0, 0, upper_x, upper_y)

AssertionError: Calculated upper Y coordinate invalid
```

Re-reading our function, we realize that line 10 should divide `dy` by `dx` rather than `dx` by `dy`. (You can display line numbers by typing Ctrl-M, then L.) If we had left out the assertion at the end of the function, we would have created and returned something that had the right shape as a valid answer, but wasn't. Detecting and debugging that would almost certainly have taken more time in the long run than writing the assertion.

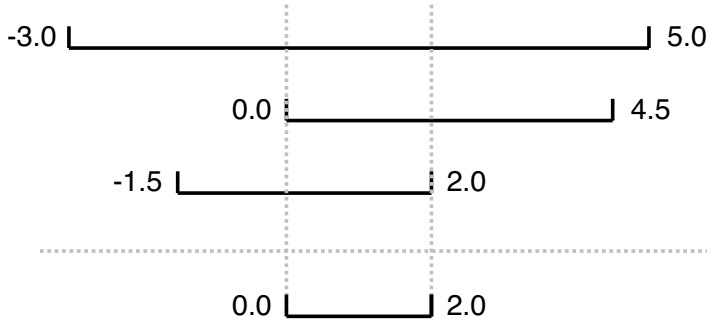
But assertions aren't just about catching errors: they also help people understand programs. Each assertion gives the person reading the program a chance to check (consciously or otherwise) that their understanding matches what the code is doing.

Most good programmers follow two rules when adding assertions to their code. The first is, *fail early, fail often*. The greater the distance between when and where an error occurs and when it's noticed, the harder the error will be to debug, so good code catches mistakes as early as possible.

The second rule is, *turn bugs into assertions or tests*. Whenever you fix a bug, write an assertion that catches the mistake should you make it again. If you made a mistake in a piece of code, the odds are good that you have made other mistakes nearby, or will make the same mistake (or a related one) the next time you change it. Writing assertions to check that you haven't regressed (i.e., haven't re-introduced an old problem) can save a lot of time in the long run, and helps to warn people who are reading the code (including your future self) that this bit is tricky.

Test-Driven Development

An assertion checks that something is true at a particular point in the program. The next step is to check the overall behavior of a piece of code, i.e., to make sure that it produces the right output when it's given a particular input. For example, suppose we need to find where two or more time series overlap. The range of each time series is represented as a pair of numbers, which are the time the interval started and ended. The output is the largest range that they all include:



Most novice programmers would solve this problem like this:

1. Write a function `range_overlap`.
2. Call it interactively on two or three different inputs.
3. If it produces the wrong answer, fix the function and re-run that test.

This clearly works — after all, thousands of scientists are doing it right now — but there's a better way:

1. Write a short function for each test.
2. Write a `range_overlap` function that should pass those tests.
3. If `range_overlap` produces any wrong answers, fix it and re-run the test functions.

Writing the tests *before* writing the function they exercise is called test-driven development (TDD). Its advocates believe it produces better code faster because:

1. If people write tests after writing the thing to be tested, they are subject to confirmation bias, i.e., they subconsciously write tests to show that their code is correct, rather than to find errors.
2. Writing tests helps programmers figure out what the function is actually supposed to do.

Here are three test functions for `range_overlap`:

```
assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
assert range_overlap([ (2.0, 3.0), (2.0, 4.0) ]) == (2.0, 3.0)
assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-25-d8be150fbef6> in <module>()
----> 1 assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
      2 assert range_overlap([ (2.0, 3.0), (2.0, 4.0) ]) == (2.0, 3.0)
      3 assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)

AssertionError:
```

The error is actually reassuring: we haven't written `range_overlap` yet, so if the tests passed, it would be a sign that someone else had and that we were accidentally using their function.

And as a bonus of writing these tests, we've implicitly defined what our input and output look like: we expect a list of pairs as input, and produce a single pair as output.

Something important is missing, though. We don't have any tests for the case where the ranges don't overlap at all:

```
assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == ???
```

What should `range_overlap` do in this case: fail with an error message, produce a special value like `(0.0, 0.0)` to signal that there's no overlap, or something else? Any actual implementation of the function will do one of these things; writing the tests first helps us figure out which is best *before* we're emotionally invested in whatever we happened to write before we realized there was an issue.

And what about this case?

```
assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == ???
```

Do two segments that touch at their endpoints overlap or not? Mathematicians usually say "yes", but engineers usually say "no". The best answer is "whatever is most useful in the rest of our program", but again, any actual implementation of `range_overlap` is going to do *something*, and whatever it is ought to be consistent with what it does when there's no overlap at all.

Since we're planning to use the range this function returns as the X axis in a time series chart, we decide that:

1. every overlap has to have non-zero width, and
2. we will return the special value `None` when there's no overlap.

`None` is built into Python, and means "nothing here". (Other languages often call the equivalent value `null` or `nil`). With that decision made, we can finish writing our last two tests:

```
assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-26-d877ef460ba2> in <module>()
----> 1 assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
      2 assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None

AssertionError:
```

Again, we get an error because we haven't written our function, but we're now ready to do so:

```
def range_overlap(ranges):
    '''Return common overlap among a set of [low, high] ranges.'''
    lowest = 0.0
    highest = 1.0
    for (low, high) in ranges:
        lowest = max(lowest, low)
        highest = min(highest, high)
    return (lowest, highest)
```

(Take a moment to think about why we use `max` to raise `lowest` and `min` to lower `highest`). We'd now like to re-run our tests, but they're scattered across three different cells. To make running them easier, let's put them all in a function:

```
def test_range_overlap():
    assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
    assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
    assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
    assert range_overlap([ (2.0, 3.0), (2.0, 4.0) ]) == (2.0, 3.0)
    assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)
```

We can now test `range_overlap` with a single function call:

```
test_range_overlap()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-29-cf9215c96457> in <module>()
----> 1 test_range_overlap()

<ipython-input-28-5d4cd6fd41d9> in test_range_overlap()
      1 def test_range_overlap():
----> 2     assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
      3     assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
      4     assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
      5     assert range_overlap([ (2.0, 3.0), (2.0, 4.0) ]) == (2.0, 3.0)

AssertionError:
```

The first test that was supposed to produce `None` fails, so we know something is wrong with our function. We *don't* know whether the other tests passed or failed because Python halted the program as soon as it spotted the first error. Still, some information is better than none, and if we trace the behavior of the function with that input, we realize that we're initializing `lowest` and `highest` to 0.0 and 1.0 respectively, regardless of the input values. This violates another important rule of programming: *always initialize from data*.

Pre- and post-conditions

Suppose you are writing a function called `average` that calculates the average of the numbers in a list. What pre-conditions and post-conditions would you write for it? Compare your answer to your neighbor's: can you think of a function that will pass your tests but not hers or vice versa?

Testing assertions

Given a sequence of values, the function `running` returns a list containing the running totals at each index.

```
running([1, 2, 3, 4])
```

```
[1, 3, 6, 10]
```

```
running('abc')
```

```
['a', 'ab', 'abc']
```

Explain in words what the assertions in this function check, and for each one, give an example of input that will make that assertion fail.

```
def running(values):
    assert len(values) > 0
    result = [values[0]]
    for v in values[1:]:
        assert result[-1] >= 0
        result.append(result[-1] + v)
        assert result[-1] >= result[0]
    return result
```

Fixing and testing

Fix `range_overlap`. Re-run `test_range_overlap` after each change you make.