Subtests

Subtests

Python 3 only. And really valuable.

Imagine a function `numwords()`, counting unique words:

```
>>> numwords("Good, good morning. Beautiful morning!")
3
```

Testing numwords()

```
class TestWords(unittest.TestCase):
    def test_whitespace(self):
        self.assertEqual(2, numwords("foo bar"))
        self.assertEqual(2, numwords("    foo bar"))
        self.assertEqual(2, numwords("foo\tbar"))
        self.assertEqual(2, numwords("foo   bar"))
        self.assertEqual(2, numwords("foo bar    \t   \t"))
        # And so on, for three words, repeated words, etc.
```

This has two problems.

Less repetition...

```
    def test_whitespace_forloop(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            self.assertEqual(2, numwords(text))
```

More maintainble. But...

... but problematic

That approach creates more problems than it solves.

```
$ python3 -m unittest test_words_forloop.py
F
======================================================
FAIL: test_whitespace_forloop (test_words_forloop.TestWords)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_forloop.py", line 17, in test_whitespace_forloop
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Pop quiz: what exactly went wrong?

A Better Way

We need something that is (a) maintainable, and (b) clear in the error reporting.

Python 3 solves this with **subtests**.

```
    def test_whitespace_subtest(self):
        texts = [
        "foo bar",
        "    foo bar",
        "foo\tbar",
        "foo   bar",
        "foo bar    \t   \t",
        ]
    for text in texts:
        with self.subTest(text=text):
            self.assertEqual(2, numwords(text))
```

Subtest Reporting

```
$ python3 -m unittest test_words_subtest.py
======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo\tbar')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo bar    \t   \t')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 4

------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures=2)
```

self.subTest()

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

`self.subTest()` creates a context for assertions.

Even if that assertion fails, the test continues through the for loop.

ALL failures are collected and reported at the end, with clear information identifying the exact problem.

Args to subTest()

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

The key-value pairs to `subTest()` are used in reporting the output. They can be anything you like.

Pay attention: the symbol `text` has *two different meanings* on these lines.

- The argument to `numwords()`
- A field in the failure report

Reporting Fields

Suppose you wrote:

```
for text in texts:
    with self.subTest(input_text=text):
        self.assertEqual(2, numwords(text))
```

Then the failure output looks like:

```
FAIL: test_whitespace_subtest (test_words_subtest.TestWords)
(input_text='foo\tbar')
```

Extra Fields

You can pass in other args, for example:

```
for index, text in enumerate(texts):
    with self.subTest(text=text, index=index):
        self.assertEqual(2, numwords(text))
```

Result:

```
FAIL: test_whitespace_subtest (test_numwords.TestWords) (text='foo\tbar',
index=2)
```

Multiple Arguments

Our original code for `split_amount()`:

```
from unittest import TestCase
from splitting import split_amount

class TestSplitting(TestCase):
    def test_split_amount(self):
        self.assertEqual([1], split_amount(1, 1))
        self.assertEqual([2, 2], split_amount(4, 2))
        self.assertEqual([2, 2, 1], split_amount(5, 3))
        self.assertEqual([3, 3, 2, 2, 2], split_amount(12, 5))
```

How to we rewrite this using subtests?

Using tuples

```
class TestSplitting(TestCase):
    def test_split_amount(self):
        testparams = [
            # expected, amount, n
            ([1], 1, 1),
            ([2, 2], 4, 2),
            ([2, 2, 1], 5, 3),
            ([3, 3, 2, 2, 2], 12, 5),
        ]
        for expected, amount, n in testparams:
            with self.subTest(amount=amount, n=n):
                self.assertEqual(expected, split_amount(amount, n))
```

Arg dicts

```
class TestSplitting(TestCase):
    def test_split_amount(self):
        testparams = [
            # (expected, args_dict)
            ([1],
             {'amount': 1, 'n': 1}),
            ([2, 2],
             {'amount': 4, 'n': 2}),
            ([2, 2, 1],
             {'amount': 5, 'n': 3}),
            ([3, 3, 2, 2, 2],
             {'amount': 12, 'n': 5}),
            ]
        for expected, args in testparams:
            with self.subTest(amount=args['amount'], n=args['n']):
                self.assertEqual(expected, split_amount(args['amount'],
args['n']))
```

Star Star

You can pass in dict data to a function call using **.

```
>>> def normal_function(a, b, c):
...     print("a: {} b: {} c: {}".format(a,b,c))
...
>>> numbers = {"a": 7, "b": 5, "c": 3}
>>> normal_function(**numbers)
a: 7 b: 5 c: 3
>>> # Exactly equivalent to:
... normal_function(a=numbers["a"], b=numbers["b"], c=numbers["c"])
a: 7 b: 5 c: 3
```

The set of dict keys MUST exactly match the set of arguments in the function when you do this. No extras, none missing.

subTest and **

```
class TestSplitting(TestCase):
    def test_split_amount(self):
        testparams = [
            # (expected, kwargs)
            ([1],
             {'amount': 1, 'n': 1}),
            ([2, 2],
             {'amount': 4, 'n': 2}),
            ([2, 2, 1],
             {'amount': 5, 'n': 3}),
            ([3, 3, 2, 2, 2],
             {'amount': 12, 'n': 5}),
        ]
    for expected, kwargs in testparams:
        with self.subTest(**kwargs):
            self.assertEqual(expected, split_amount(**kwargs))
```

Lab: Subtests

Instructions: `lab-subtests.txt`

(Note this builds on the previous lab - you must complete it first.)

- In `labs` folder
- First follow the instructions to modify `textlib.py` and `test_textlib.py`
- When you are done, study the solution - compare to what you wrote.
- ... and then optionally follow the extra credit instructions