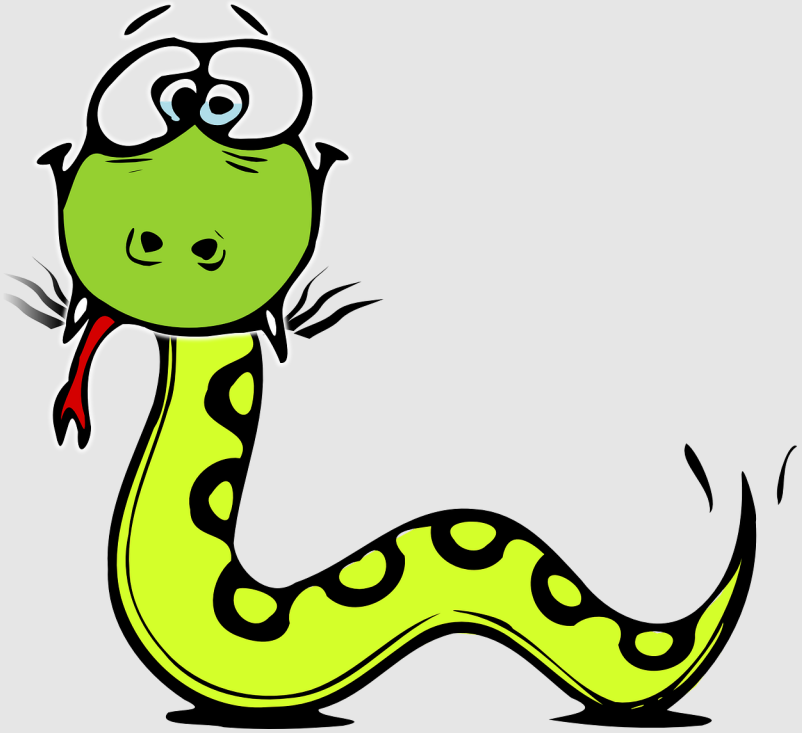


Python Workshop

What I learned by adding tests to `search-analytics`

- Use the right tools
- Virtualise the environment
- Structure the project
- Test your code
- Use a code coverage tool
- Lint your code - including the tests

Use the right tools

- `pyenv` - for Python language and dependency versioning
- `venv` - the Python virtual environment
- `pip` - the Python package manager
- `unittest` - the test module that's included in the Python standard library
- `coverage` - a code coverage tool
- `pylint` - your code should conform to PEP-8 - the Python Style Guide

Getting setup

```
$ cd ~
$ git clone https://github.com/pyenv/pyenv.git ~/.pyenv
```

For `zsh`

```
$ echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.zshrc
$ echo 'command -v pyenv >/dev/null || export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.zshrc
$ echo 'eval "$(pyenv init -)"' >> ~/.zshrc

$ cat .zshrc

$ source ~/.zshrc
```

Getting setup

For `bash`

```
$ echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
$ echo 'command -v pyenv >/dev/null || export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
$ echo 'eval "$(pyenv init -)"' >> ~/.bashrc

$ cat .bashrc

$ echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.profile
$ echo 'command -v pyenv >/dev/null || export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.profile
$ echo 'eval "$(pyenv init -)"' >> ~/.profile

$ cat .profile

$ source ~/.bashrc
```

Install Python

```
$ pyenv install 3.11.1
$ pyenv global 3.11.1
$ pyenv versions

$ python --version
```

Create a project structure

```
$ cd <path_to_your_source_code>

$ mkdir python-workshop
$ cd python-workshop

$ pyenv local 3.11.1 # creates a `.python-version` file
$ python rehash

$ touch requirements.txt
$ mkdir src
$ mkdir test
```

Create a Virtual Environment

```
$ python -m venv venv
$ source venv/bin/activate
```

To `deactivate` the environment

```
(venv) $ deactivate
```

Add your project dependencies

```
(venv) $ echo 'coverage' >> requirements.txt
(venv) $ echo 'pylint' >> requirements.txt
(venv) $ pip install -r requirements.txt
```

Time for some code

test/test_string_utils.py

```
import unittest

from src.string_utils import count

class TestStringUtils(unittest.TestCase):
  def test_count(self):
    self.assertEqual(count("Monty Python"), 12)
```

Run the test

```
(venv) $ python -m unittest discover
```

You sould get some errors telling you the thing you're trying to test doesn't exist yet.

src/string_utils.py

```
def count(string):
  return len(string)
```

Run the tests again

```
(venv) $ python -m unittest discover
```

AND... Nothing!

```
----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Can you `discover` the clue?

We ran...

```
(venv) $ python -m unittest discover
```

The `discover` option in that command means we need to tell Python that we are creating a module.

To do that we need to create an empty file named `__init__.py` (that's double underscore either side of init).

```
(venv) $ touch test/__init__.py
```

Let's try again

```
(venv) $ python -m unittest discover
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

SUCCESS!

Ok? More code...

test/test_string_utils.py

```
  def test_contains(self):
    self.assertEqual(contains("Monty", "Monty Python and the Holy Grail"), True)
    self.assertEqual(contains("Monty", "Star Wars"), False)
```

src/string_utils.py

```
def contains(text, string):
  return text in string
```

And test...

```
(venv) $ python -m unittest discover
E.
======================================================================
ERROR: test_contains (test.test_string_utils.TestStringUtils.test_contains)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/gcl/code/python-workshop/test/test_string_utils.py", line 10, in test_contains
    self.assertEqual(contains("Monty", "Monty Python and the Holy Grail"), True)
                     ^^^^^^^^
NameError: name 'contains' is not defined

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (errors=1)
```

It's trying to tell us that it can't find the `contains` method. We forgot to tell it!

Let's do that

Update `test/test_string_utils.py` and add `contains`

```
from src.string_utils import count, contains
```

You can use a splat (*) operator instead of listing the methods, but I'm going to suggest you don't. Turns out listing the methods is useful when refactoring things.

Now that's fixed we can try again...

```
(venv) $ python -m unittest discover
```

We should get two passing tests now.

Be nice to see how much code our tests cover

```
(venv) $ coverage run -m unittest discover
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Looks like it did something, but where's the report?

Turns out coverage needs two commands

One to produce the data, and one to produce the report.

So, we really want this...

```
(venv) $ coverage run -m unittest discover && coverage report -m
```

What about linting our code?

```
(venv) $ pylint --recursive=y ./src ./test
```

You should see a bunch of warnings and errors with references telling you what to fix and where.

But, I know what you're thinking - can we auto fix these issues?

We need another tool for that

Remember PEP-8?

Here's a tool that makes the bits of your code that it can - comply with PEP-8.

```
(venv) $ echo 'autopep8' >> requirements.txt
(venv) $ pip install -r requirements.txt
(venv) $ pyenv rehash
(venv) $ autopep8 --in-place --aggressive --aggressive src/*.py test/*.py
```

So, what did it fix?

```
(venv) $ pylint --recursive=y ./src ./test
```

About now, my fingers are getting tired of typing those long commands...

But, we can fix that. We could write a CLI, but for now let's just add some `aliases`?

```
alias py-up='source venv/bin/activate'
alias py-down='deactivate'
alias py-bundle='pip install -r requirements.txt'
alias py-test='python -m unittest discover'
alias py-cov='coverage run -m unittest discover && coverage report -m'
alias py-cop='pylint --recursive=y ./src ./test'
```

Add these to your `.bashrc` or `.zshrc` file, reload your shell and relax.

Debugging Python with `pdb`

But, before we debug anything, how can we use the Python REPL to run our code...?

```
(env) $ python
>>> from src.string_utils import count, contains
>>> count("Hello World")
11
>>> contains('Spam', "What a wonderful thing Python is")
False
>>> contains('Spam', "What a wonderful thing Spam is")
True
>>> exit()
```

Now we can debug

```
(env) $ python
```

```
>>> import pdb
>>> from src.string_utils import count, contains
>>> pdb.run('count("Hello World")')
> <string>(1)<module>()
(Pdb) h # help
(Pdb) s # step into function
(Pdb) l # list the source code
(Pdb) a # show argument value(s)
(Pdb) p arg # print the value of the a parameter
(Pdb) r # run the code at this point
(Pdb) c # continue to next breakpoint
(Pdb) n # continue to next line
(Pdb) q # quit the debugger
>>> exit()
```

More ways to envoke `pdb`

- Invoke as a script - to debug scripts `python -m pdb some_script.py`
- Add `import pdb; pdb.set_trace()` to your source code at a specific point to interactivaly drop you into `pdb`

Envoking `pdb` in your source code

Update your source code...

```
def count(string):
    import pdb
    pdb.set_trace()
    return len(string)
```

Start the REPL...

```
(env) $ python
```

Run the code...

```
>>> import pdb
>>> from src.string_utils import count, contains
>>> count("Hello World")
```

Need more `pdb`?

Complete reference at https://docs.python.org/3/library/pdb.html

Like most debuggers... `pdb` is not the easiest of tools to use, but is extremely powerful and useful when you need it.

Exercise

Write a function that:

- takes a string
- splits that string - by space - into an array
- removes duplicate values
- orders the array alphabetically

Test your code. Do it TDD for bonus points!

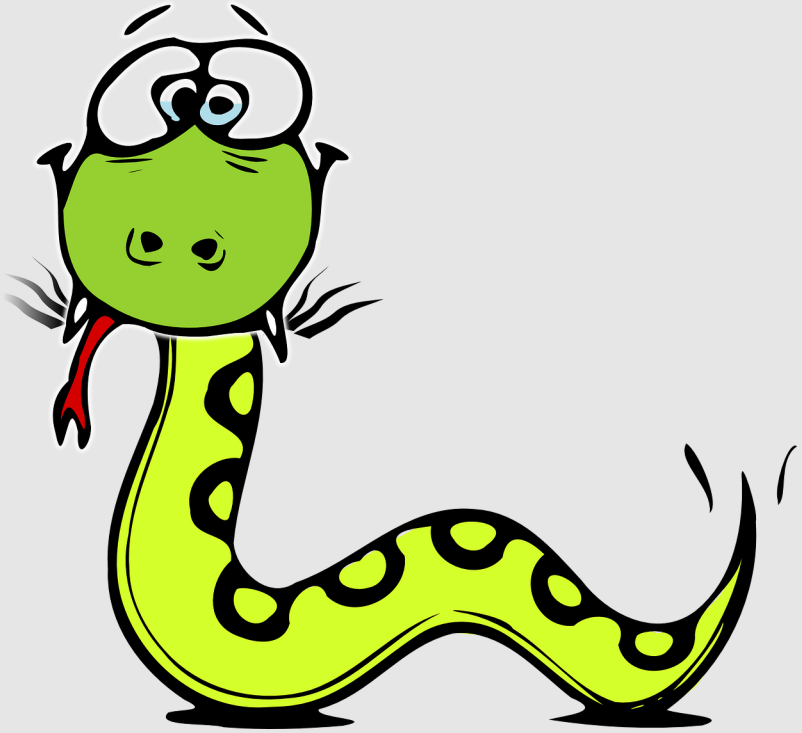
So, given `my_function("Lovely Spam Wonderful Spam Lovely Spam Wonderful Spam Spam Spam Spam Spam Lovely Spam Lovely Spam Lovely Spam Spam Spam Spam Spam")`

Your function should return `['Lovely', 'Spam', 'Wonderful']`

Are you getting `line too long` linter errors?

Get `pylint` to ignore the error by adding this directly above the line in question...

```
# pylint: disable=line-too-long
```



That's all folks!