labibi Documentation

Release 0.1

C. Titus Brown

April 09, 2015

Contents

3. Titus Brown <ctb@msu.edu> and Alexandra Pawlik <a.pawlik@software.ac.uk>

We will use this Etherpad site to distribute commands.

Explain: minute cards; stick notes on monitors.

The khmer project: https://github.com/ged-lab/khmer/

The final 'sqer' project: https://github.com/ngs-docs/sqer-demo

CHAPTER 1

Session I: Testing

For the rest of the sessions, you'll need an account at http://github.com/ as well as an account at https://readthedocs.org/.

1.1 The 'sqer' Python package

We're going to create a Python utility called 'sqer' to give read statistics.

Let's start by creating a 'sqer' library.

Make a directory 'sqer'.

Inside this make another directory 'sqer'.

In 'sqer/sqer' open a file '__init__.py'

From within the top level directory 'sqer' you should be able to do

```
python -c "import sqer; print sqer"
```

Go into the 'sqer' directory and initialize a git repo:

```
git init
```

Add the sqer/ package directory:

```
git add *
```

```
git status
```

Note the .pyc file – this is not a source file, but rather a generated file. Let's remove it from the commit:

```
git rm --cached sqer/__init__.pyc
```

and also ignore it from here on out:

```
echo '*.pyc' > .gitignore
```

Now:

```
git status
```

will not show it as a file, and 'git add' will not add it unless it's forced.

Next,

```
git add .gitignore
```

and commit:

```
git commit -am "initial commit"
```

Now 'git status' should show you only untracked files, no differences.

Note: You can use 'git log' to get a history of commits.

1.2 Writing some code (and tests)

Let's start by writing a function that computes the sum of legit DNA bases in a sequence record. The main thing you need to know here is that each sequence record will come from the screed utility, which will give us record objects with a 'name', 'sequence', optional 'accuracy' (for FASTQ), and optional 'description' (from the FASTA/FASTQ sequence name).

So, put:

```
def sum_bp(record):
    return len(record.sequence)
```

in 'sqer/__init__.py'.

Now, let's add a test. Create a directory 'tests' and put a file 'test_basic.py' in it; this file should contain:

```
import sqer

class FakeRecord(object):
    def __init__(self, sequence, name=''):
        self.sequence = sequence
        self.name = name

def test_sum_bp():
    r = FakeRecord('ATGC')
    assert sqer.sum_bp(r) == 4
```

Here, 'FakeRecord' is a stub object that lets you test your code by faking an object type solely for testing.

Now, run:

```
nosetests
```

You should see:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

You can also run 'nosetests -v' to get more verbose output.

Tests pass? Great, add and commit it with git!

```
git add tests
git commit -am "initial tests"
```

Now, let's add a new function, 'sum_bp_records', to `sqer/__init__.py`.

```
def sum_bp_records(records):
    total = 0
    for record in records:
        total += sum_bp(record)

    return total
```

How shall we test this? Well, all it expects is an iterable of records: add this to tests/test_basic.py:

```
def test_sum_bp_records():
    rl = [ FakeRecord("A"), FakeRecord("G") ]
    assert sqer.sum_bp_records(rl) == 2
```

Now run 'nosetests' again – works? No complaints?

Great, commit it with git:

```
git status
git commit -am "added sum_bp_records and test"
```

1.2.1 Exercises

1. Write a test to handle (and ignore) non-ACGT. (Fix the code.)
2. Write a test to verify that lower-case is handled. (Fix the code.)
3. Write a function to calculate the average length of records in a file; test it.

1.3 Writing a script

Let's write something to let us use this from the command line. Put the following code in `count-read-bp.py`:

```
#! /usr/bin/env python
import argparse
import screed
import sqer

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('filenames', nargs='+')

    args = parser.parse_args()

    total = 0
    for filename in args.filenames:
        records = screed.open(filename)
        total += sqer.sum_bp_records(records)

    print '%d bp total' % total

if __name__ == '__main__':
    main()
```

Next, 'chmod +x count-read-bp.py'. This makes UNIX aware that it's an executable file.

Try running it:

```
./count-read-bp.py
```

Note the friendly error message! Note that you can use '-h', too.

How do we test this??

Put:

```
>a
ATCG
>b
GCTA
```

in a file 'reads.fa'. Then:

```
./count-read-bp.py reads.fa
```

You should see '8 bp total'. Great!

Commit:

```
git add count-read-bp.py reads.fa
git commit -am "command-line script count-read-bp, plus test data"
```

Check with 'git status'. Do you have editor remainder files (like ~ files from using emacs)? Add them to .gitignore and commit the changes.

1.4 Testing command line scripts

Put this in a file 'tests/test_scripts.py':

```
import subprocess
import os
thisdir = os.path.dirname(__file__)
thisdir = os.path.normpath(thisdir)

sqerdir = os.path.join(thisdir, '../')
sqerdir = os.path.normpath(sqerdir)


def test_count_reads():
    scriptpath = os.path.join(sqerdir, 'count-read-bp.py')
    datapath = os.path.join(sqerdir, 'reads.fa')

    p = subprocess.Popen([scriptpath, datapath],
                         stdout=subprocess.PIPE,
                         stderr=subprocess.PIPE)
    (out, err) = p.communicate()

    assert p.returncode == 0
    assert "8 bp total" in out, out
```

Now run 'nosetests' – what does it say?

Add and commit:

```
git add tests/test_scripts.py
git commit -am "added test for the count-read-bp.py script"
```

1.5 Regression tests with command line scripts

Grab some data from somewhere (e.g. 25k.fq.gz from training files) and put it in `test-reads.fq`. You can subset the 25k.fq.gz file if you want:

```
gunzip -c 25k.fq.gz | head -400 > test-reads.fq
```

Add another test to `sqer/test_scripts.py`:

```
def test_count_reads_2():
    scriptpath = os.path.join(sqerdir, 'count-read-bp.py')
    datapath = os.path.join(sqerdir, 'test-reads.fq')
    print thisdir, sqerdir, scriptpath, datapath

    p = subprocess.Popen([scriptpath, datapath],
                         stdout=subprocess.PIPE,
                         stderr=subprocess.PIPE)
    (out, err) = p.communicate()

    assert p.returncode == 0
    assert "8 bp total" in out, out
```

And now run 'nosetests'.

It should break, right? :)

Fix the last 'assert' code, then rerun; when it all passes, do:

```
git add test-reads.fq
git status
```

Make sure that only what you think should be there is there; then do:

```
git commit -am "added regression test"
```

1.6 Reorganize

Let's put the data files under data/:

```
mkdir data
mv test-reads.fq data
mv reads.fa data/test-reads.fa
```

...now, fix the tests!

1.7 Exercises

1. Add friendly output to the script, e.g. files opened, # records processed.
2. Add a flag for 'silence':

```
parser.add_argument("-s", dest="silent", type=bool)
```

and

```
if args.silent: ...
```

1.8 Testing summary

Points to cover:

- any functions named 'test*' in files named 'test*' are executed.
- unit tests are for small bits of code;
- script tests (the first one) are for testing the script API;
- regression tests are for making sure behavior stays the same. (We didn't actually count the number of bases in that file, right? We just assumed it was counting them right.)
- these three types of tests are for *different purposes* and test different things! Which one is most useful?
- 'print' statements and the like inside the tests are captured, and only output upon error.
- assert statements are the way to check things.

Slightly more advanced topics if people are interested:

- what do you do about output files? (temp directories)
- how do you measure if your tests are "good enough"? (code coverage)

1.9 Advanced exercises

4. Write a reservoir sampling algorithm.

- An Introduction to the Nose Testing Framework

  http://ivory.idyll.org/articles/nose-intro.html

CHAPTER 2

Session 2: git / version control

2.1 Basic git

Tutorial link:

https://github.com/apawlik/TGAC-6-Nov-2013/tree/master/version-control

Cover:

- branching and merging;
- conflicts;
- creating a project on github;
- pushing to the project on github;

2.2 Git pull requests

Cover:

- setting up a pull request;
- viewing differences;
- commenting on individual lines;
- multiple pushes;
- updating from another branch;
- github markdown, including checklists!
- online editing of github files.

2.3 External Resources

- Github Flow description

  http://scottchacon.com/2011/08/31/github-flow.html

CHAPTER 3

Session 3: Python packages and installs

3.1 Using virtualenv

To create a virtual environment:

```
python -m virtualenv ~/env
```

To activate it as your default Python environment:

```
. ~/env/bin/activate
```

Now, even without root, you can do `pip install` of whatever packages you like.

To deactivate it,

```
deactivate
```

3.2 Building a 'setup.py'

In the sqer directory,

1. grab the latest ez_setup.py from https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py:

   ```
   curl -O https://bitbucket.org/pypa/setuptools/raw/19873119647deae8a68e9ed683317b9ee170a8d8/ez_se
   ```

2. Put the following in setup.py:

   ```
   import ez_setup
   ez_setup.use_setuptools()

   from setuptools import setup

   setup(name="sqer",
         version="0.1",
         packages=['sqer'],
         install_requires=["screed >= 0.7"],
         setup_requires=["nose >= 1.0",],
         scripts=["count-reads.py"],
         test_suite = 'nose.collector',
   )
   ```

3. Put the following in setup.cfg:

```
    [nosetests]
    verbosity = 2
```

Now you can do:

```
python setup.py test
```

to run the tests, and:

```
python setup.py install
```

This will install 'sqer' so that (a) it's importable from anywhere,

```
python -c "import sqer"
```

and (b) the script(s) are in your path so that:

```
count-read-bp.py data/test-reads.fa
```

should work from anywhere.

Remember to add and commit to git:

```
git add setup.cfg setup.py
git commit -am "added install configuration"
```

Note that if you create a .tar.gz,

```
cd ..
tar czf /tmp/sqer.tar.gz sqer
cd sqer
```

you can now do:

```
pip install /tmp/sqer.tar.gz
```

and this will also work with URLs to the .tar.gz as well as github files & release links...

One final comments: 'git status' will show you that the directory is getting messy. Add:

```
*.egg
*.egg-info
build
```

to .gitignore, and then commit:

```
git commit -am "updated gitignore with setup.py detritus"
```

It's probably time to do a 'git push origin master' too!

3.3 Building a default/basic 'Makefile'

Put the following in 'Makefile' in the seqr/ directory:

```
all:
    python setup.py build

install:
    python setup.py install
```

```
clean:
     python setup.py clean
     rm -fr build

test:
     python setup.py test
```

Note: 'make' is picky about tabs vs spaces – the lines after the ':' need to be indented with tabs to work properly.

This will now let us do 'make' (which will execute the first target, 'all'); 'make install'; 'make clean'; and 'make test'. These will do the obvious things.

The important thing here is that all of these are *standard* make commands. If I see a Makefile in a repository, then I assume that it's got the commands above. Convention, convention, convention!

Remember to:

```
git add Makefile
git commit -am "added Makefile"
```

3.4 Documentation

We're going to build some docs using Sphinx and reStructuredText.

Do:

```
mkdir doc
cd doc
sphinx-quickstart
```

Use default values for everything; specify project name, author, and version.

Now, in the 'doc' directory, do:

```
make html
```

and look at _build/html/index.html

Let's flesh this out a bit – edit 'index.rst' and add an indented 'details' under Contents, e.g.:

```
Contents:

.. toctree::
   :maxdepth: 2

   details
```

Now create 'details.rst' to contain:

```
===============
Project Details
===============

sqer is awesome.

Important details
=================

This where all my documentation goes.
```

...and run 'make html' again. Look at _build/html/index.html.

Be sure to do:

```
rm -fr _build
git add *
git commit -am "added docs"
```

And also add a rule to the top-level Makefile:

```
doc:
     cd doc && make html
```

(and git add/commit the Makefile changes.)

Now, push this all to github:

```
git push origin master
```

and let's go configure it at http://readthedocs.org/.

Reminder: under your github project, settings, service hooks, enable the 'readthedocs' service hook.

CHAPTER 4

Session 4: Analysis pipelines and IPython Notebook

4.1 A slightly more useful sqer script

Put the following in `sqer/calc-lengths.py`:

```
#! /usr/bin/env python
import argparse
import screed
import sqer

def main():
   parser = argparse.ArgumentParser()
   parser.add_argument('filenames', nargs='+')

   args = parser.parse_args()

   total = 0
   for filename in args.filenames:
       records = screed.open(filename)
       for record in records:
          print len(record.sequence)

if __name__ == '__main__':
   main()
```

then

```
chmod +x calc-lengths.py
git add calc-lengths.py
git commit -am "added calc-lengths.py"
```

4.1.1 Exercises

1. Write a test for calc-lengths.py!

4.2 Write a little analysis pipeline

Create a directory `pipeline` under sqer:

```
mkdir pipeline
```

and copy in the 'trinity-nematostella.fa.gz' file from the training files into this directory (any FASTA/FASTQ file will do here), gunzip it, and then rename it to `assembly.fa`.

Now, create `pipeline/Makefile` containing:

```
all: lengths.txt

lengths.txt: assembly.fa
     ../calc-lengths.py assembly.fa > lengths.txt
```

Now, when you type 'make', it will run your analysis pipeline. (...pretend that 'calc-lengths.py' takes a long time or something :)

4.3 Start up IPython Notebook

From within the pipeline directory, run:

```
ipython notebook --pylab=inline
```

Click on 'New Notebook'. In this new notebook, enter:

```
data = numpy.loadtxt('lengths.txt')
hist(data, bins=100)
xlabel('Sequence lengths')
ylabel('N sequences with that length')
title('Sequence length spectrum')
savefig('hist.pdf')
```

and hit 'Shift-ENTER' to execute.

Voila!

Save the notebook (File... save...)

Now, do (from within the pipeline directory):

```
ls -1 assembly.fa lengths.txt > .gitignore
git add Makefile .gitignore *.ipynb
git commit -am "analysis makefile and notebook"
```

and then:

```
git push origin master
```

Now go find the raw URL to your notebook on github, copy it, and then paste it in at:

```
http://nbviewer.ipython.org
```

Voila!

Additional IPython resources:

- The ipynb site: http://ipython.org/notebook.html
- A gallery of interesting notebooks: https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks
- The matplotlib gallery: http://matplotlib.org/gallery.html

Note that you can use '%loadpy' in IPython Notebook to grab code from online and import it into your notebook automagically.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*