Software Testing, TDD, and Documentation

Programming (for biologists)

BIOL 7800

Software Testing

Writing code to test logical "units" of your program

What are units **here**?

```python
def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
```

Software Testing

Writing code to test logical "units" of your program

example14.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
```

unit

unit

unit

Here, functions are the logical unit to test…

Software Testing

Writing code to test logical "units" of your program

example105-amie-s.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py | example105-amie-s.py

```python
class WordCount():
    '''Counts the words of a file'''
    def __init__(self, file_name):
        '''Requisite attributes'''
        self.file_name = file_name
        self.file_path = os.path.abspath(file_name)
        self.master_word_tuple = self.construct_master()
        self.unique_words = set(self.master_word_tuple)

    def __str__(self):
        return '{}:\n words'.format(
                self.file_path)

    def construct_master(self):
        '''Construct master list of all words in file'''
        master_list = []
        # check that input file opens in read mode w/out error
        with open(self.file_name, 'r') as f:
            for line in f:
                word_list = self.process_line(line)
                master_list += (word for word in word_list)
        return tuple(master_list)

    def process_line(self, strng):
        '''
        Parse a string by spaces into a list of words
        '''
        strng = strng.lower().replace('\r\n', ' ')
        strng = strng.replace('  ', ' ').replace('   ', ' ').replace('-', ' ')
        new_strng = ''
        for character in strng:
            if character.isalpha() or character == ' ':
                new_strng += character
        list_of_words = new_strng.split(' ')
        return list_of_words
```

What are logical "units"?

Software Testing

Writing code to test logical "units" of your program

example105-amie-s.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py | example105-amie-s.py

```python
class WordCount():
    '''Counts the words of a file'''
    def __init__(self, file_name):
        '''Requisite attributes'''
        self.file_name = file_name
        self.file_path = os.path.abspath(file_name)
        self.master_word_tuple = self.construct_master()
        self.unique_words = set(self.master_word_tuple)

    def __str__(self):
        return '{}:\n words'.format(
                self.file_path)

    def construct_master(self):
        '''Construct master list of all words in file'''
        master_list = []
        # check that input file opens in read mode w/out error
        with open(self.file_name, 'r') as f:
            for line in f:
                word_list = self.process_line(line)
                master_list += (word for word in word_list)
        return tuple(master_list)

    def process_line(self, strng):
        '''
        Parse a string by spaces into a list of words
        '''
        strng = strng.lower().replace('\r\n', ' ')
        strng = strng.replace('  ', ' ').replace('   ', ' ').replace('-', ' ')
        new_strng = ''
        for character in strng:
            if character.isalpha() or character == ' ':
                new_strng += character
        list_of_words = new_strng.split(' ')
        return list_of_words
```

unit

unit

unit

Here, methods are the logical unit to test…

Software Testing

Writing code to test logical "units" of your program

example105-amie-s.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py | example105-amie-s.py

```python
class WordCount():
    '''Counts the words of a file'''
    def __init__(self, file_name):
        '''Requisite attributes'''
        self.file_name = file_name
        self.file_path = os.path.abspath(file_name)
        self.master_word_tuple = self.construct_master()
        self.unique_words = set(self.master_word_tuple)

    def __str__(self):
        return '{}:\n words'.format(
                self.file_path)

    def construct_master(self):
        '''Construct master list of all words in file'''
        master_list = []
        # check that input file opens in read mode w/out error
        with open(self.file_name, 'r') as f:
            for line in f:
                word_list = self.process_line(line)
                master_list += (word for word in word_list)
        return tuple(master_list)

    def process_line(self, strng):
        '''
        Parse a string by spaces into a list of words
        '''
        strng = strng.lower().replace('\r\n', ' ')
        strng = strng.replace('  ', ' ').replace('   ', ' ').replace('-', ' ')
        new_strng = ''
        for character in strng:
            if character.isalpha() or character == ' ':
                new_strng += character
        list_of_words = new_strng.split(' ')
        return list_of_words
```

unit

unit

unit

When we test these units, we are doing what is known as "unit testing"

Software Testing

Helps you write better (more discrete) code

Helps make code more extensible

Helps find (and fix) corner cases

Helps you comfortably *build on* and *refactor* older code

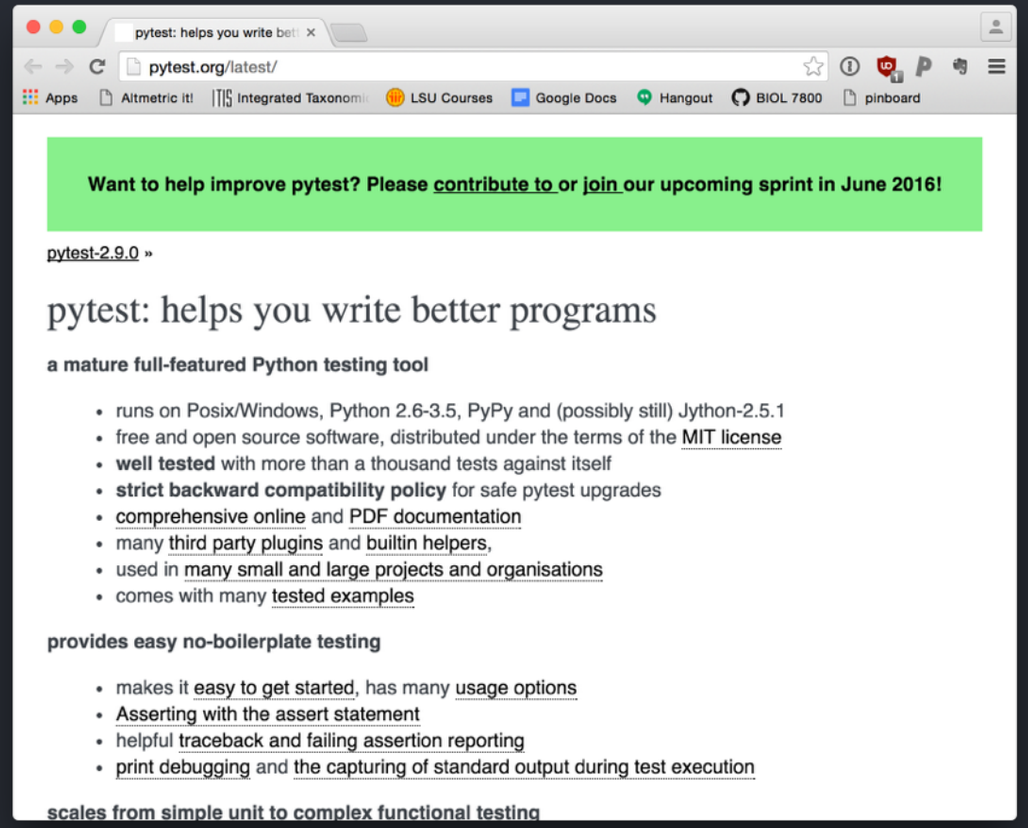
Image: wikimedia commons

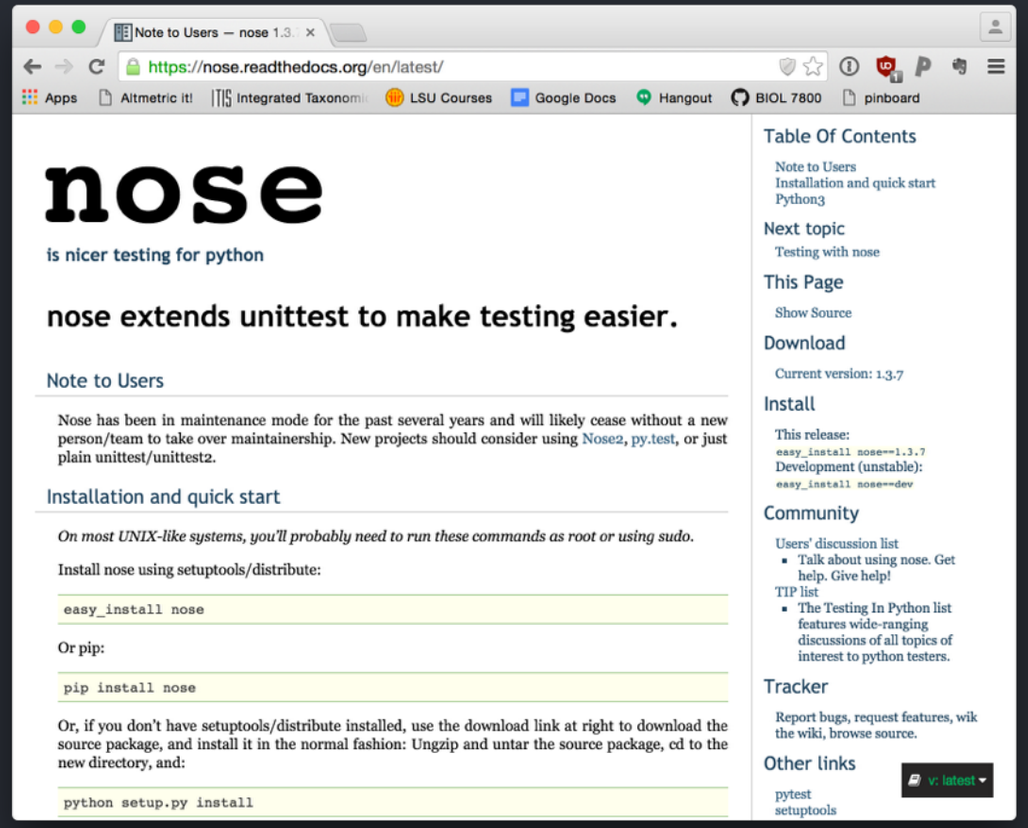
Software Testing: unittest

unittest

The **native** python module/library for unit testing

But there are other 3rd party unit testing libraries





These often, like nose, extend unittest to make testing more efficient

Software Testing: unittest

example14.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


```

Write a test for this unit

Write a test for this unit

Write a test for this unit

Software Testing: unittest

lec_15_test.py

```
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):
    pass
```

If we are testing lec_15.py, we usually name the file lec_15_test.py

This is important

Software Testing: unittest

lec_15_test.py

```
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):
    pass
```

If we are testing
usually name the file
lec_15_test.py

This is
important

We import unittest
and our code to test,
lec_15

Software Testing: unittest

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):
    pass
```

If we are testing
usually name the file
lec_15_test.py

This is
important

We import
and our code to test,
lec_15

We create a new class
that subclasses the
unittest.TestCase
class

This "brings in" all of
the methods from
unittest

Software Testing: unittest

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):
    pass
```

If we are testing
usually name the file
lec_15_test.py

This is
important

We import
and our code to test,
lec_15

We create a new class
that
unittest.TestCase
class

test class names
should always start
with Test

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        pass

    def test_my_sum(self):
        pass

    def test_power(self):
        pass
```

Normally, we would lay out our skeleton test units

Each is a new method of our TestLec15 class

These methods are also prefixed with test_

And, method names are usually equal to the functions we're testing

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        pass

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

Normally, we would include an ifmain statement.

But, what happens if we include ifmain and run lec_15_test.py?

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

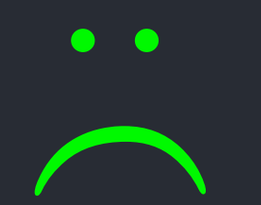
lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        pass

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

$ python lec_15_test.py

Traceback (most recent call last):
  File "lec_15_test.py", line 18, in <module>
    main()
NameError: name 'main' is not defined



Why?

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        pass

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

With unittests, we run the unittest.main() method

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        pass

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

Output showing three tests running successfully

```
$ python lec_15_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

unittest.main() gives you all this pretty goodness

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        pass

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

Output showing three tests running successfully

Why do tests **pass**?
They're not doing anything…

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        pass

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

Output showing three tests running successfully

Why do tests **pass**?

Basically, tests have no explicit content yet, so they pass *by default*

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'god'
        self.assertEqual(observed, expected)

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

Run our function

Define what we expect

Test obs *v.* expected

assertEqual method of unittest.testcase class

```
$ python lec_15_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'wrong'
        self.assertEqual(observed, expected)

```

What happens when a test fails?

Will be wrong!!

```
..F
======================================================================
FAIL: test_reverse (__main__.TestLec15)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "lec_15_test.py", line 11, in test_reverse
    self.assertEqual(observed, expected)
AssertionError: 'god' != 'wrong'
- god
+ wrong
----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'wrong'
        self.assertEqual(observed, expected)

    def test_my_sum(self):
        pass

    def test_power(self):
        pass

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

What happens when a test fails?

Will be wrong!!

```
..F
======================================================================
FAIL: test_reverse (__main__.TestLec15)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "lec_15_test.py", line 11, in test_reverse
    self.assertEqual(observed, expected)
```

Why did this test run 3rd?

Software Testing: unittest

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)


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

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'god'
        self.assertEqual(observed, expected)

    def test_my_sum(self):
        observed = lec_15.my_sum([1, 2, 3])
        self.assertEqual(observed, 6)

    def test_power(self):
        observed = lec_15.power(2, 2)
        self.assertEqual(observed, 4)

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

Fill in rest of tests…

```
$ python lec_15_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Success!!

Software Testing

Ok, great. But <u>why</u>?

Helps you write better (more discrete) code

smaller units are easier to rest (and re-use)

Helps make code more extensible

tests ensure units operate as expected in different contexts

Helps find (and fix) corner cases

tests often identify potential problems before they become problems

Helps you comfortably *<u>build on</u>* and *<u>refactor</u>* older code

tests often identify potential problems before they become problems

Image: wikimedia commons

Software Testing: unittest

Helps you comfortably *build on* and *refactor* older code

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    if isinstance(num, str):
        num = int(num)
    if isinstance(power, str):
        power = int(power)
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
```

Let's say we want to add code to deal with num and pow values that are strings and not integers

Software Testing: unittest

Helps you comfortably *build on* and *refactor* older code

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    if isinstance(num, str):
        num = int(num)
    if isinstance(power, str):
        power = int(power)
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)
```

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'god'
        self.assertEqual(observed, expected)

    def test_my_sum(self):
        observed = lec_15.my_sum([1, 2, 3])
        self.assertEqual(observed, 6)

    def test_power(self):
        observed = lec_15.power(2, 2)
        self.assertEqual(observed, 4)

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

unit test allows us to know those additions did not change our original intent…

```
$ python lec_15_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Software Testing: unittest

Helps you comfortably *build on* and *refactor* older code

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    if isinstance(num, str):
        num = int(num)
    if isinstance(power, str):
        power = int(power)
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)
```

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'god'
        self.assertEqual(observed, expected)

    def test_my_sum(self):
        observed = lec_15.my_sum([1, 2, 3])
        self.assertEqual(observed, 6)

    def test_power(self):
        observed = lec_15.power(2, 2)
        self.assertEqual(observed, 4)
        observed = lec_15.power('2', '2')
        self.assertEqual(observed, 4)

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

unit test also allows us to test new additions to code

```
$ python lec_15_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Software Testing: unittest

Helps you comfortably *build on* and *refactor* older code

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    if isinstance(num, str):
        num = int(num)
    if isinstance(power, str):
        power = int(power)
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)
```

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'god'
        self.assertEqual(observed, expected)

    def test_my_sum(self):
        observed = lec_15.my_sum([1, 2, 3])
        self.assertEqual(observed, 6)

    def test_power_int(self):
        observed = lec_15.power(2, 2)
        self.assertEqual(observed, 4)

    def test_power_string(self):
        observed = lec_15.power('2', '2')
        self.assertEqual(observed, 4)

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

Often, we want to keep our unit tests as smaller discrete units
(easier to tell what is causing a problem)

```
$ python lec_15_test.py
....
----------------------------------------------------------------------
Ran 3 tests in 0.000s
OK
```

Software Testing: unittest

Helps you comfortably *build on* and *refactor* older code

lec_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

lec_15.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    if isinstance(num, str):
        num = int(num)
    if isinstance(power, str):
        power = int(power)
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
    print(my_power)

```

lec_15_test.py

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15Reverse(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'god'
        self.assertEqual(observed, expected)


class TestLec15Sum(unittest.TestCase):

    def test_my_sum(self):
        observed = lec_15.my_sum([1, 2, 3])
        self.assertEqual(observed, 6)


class TestLec15Power(unittest.TestCase):
    def test_power_int(self):
        observed = lec_15.power(2, 2)
        self.assertEqual(observed, 4)

    def test_power_string(self):
        observed = lec_15.power('2', '2')
        self.assertEqual(observed, 4)

if __name__ == '__main__':
    unittest.main()

```

We can also organize unit tests as a class per function

Software Testing: unittest

Helps you comfortably *build on* and *refactor* older code

lec_15_test.py

```python
#!/usr/bin/env python

import unittest
import lec_15

class TestLec15Reverse(unittest.TestCase):

    def test_reverse(self):
        observed = lec_15.reverse('dog')
        expected = 'god'
        self.assertEqual(observed, expected)


class TestLec15Sum(unittest.TestCase):

    def test_my_sum(self):
        observed = lec_15.my_sum([1, 2, 3])
        self.assertEqual(observed, 6)


class TestLec15Power(unittest.TestCase):
    def test_power_int(self):
        observed = lec_15.power(2, 2)
        self.assertEqual(observed, 4)

    def test_power_string(self):
        observed = lec_15.power('2', '2')
        self.assertEqual(observed, 4)

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

We can also organize unit tests as a class per function

Running tests in verbose (-v/--verbose) mode

```
$ python lec_15_test.py -v

test_power_int (__main__.TestLec15Power) ... ok
test_power_string (__main__.TestLec15Power) ... ok
test_reverse (__main__.TestLec15Reverse) ... ok
test_my_sum (__main__.TestLec15Sum) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

TDD

Test Driven Development

Test Driven Development

Is a method or a mode of software development

example14.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py

```python
#!/usr/bin/env python


def reverse(my_string):
    return my_string[::-1]


def my_sum(my_list):
    if not isinstance(my_list, list):
        raise IOError("Must input a list")
    else:
        return sum(my_list)


def power(num, power):
    return num**power


def main():
    my_string = 'able was i ere i saw elba'
    rev_my_string = reverse(my_string)
    print(rev_my_string)
    my_list = [2, 4, 6, 8]
    sum_my_list = my_sum(my_list)
    print(sum_my_list)
    my_power = power(20, 20)
```

Based on this idea of "testing" your code

And you build your new code base from tests of "units" of your code

Test Driven Development

Is a

1. Each new method or function starts with a test case
2. Run test(s). Ensure new test fails.
3. Write the function with minimal code to pass test(s)
4. Run test(s). Ensure new test passes.
5. Refactor (and re-run tests)

Test Driven Development

Is a

1. Each new method or function starts with a test case

This helps code author **focus** on a function's requirements

and define what code *needs* to do

Test Driven Development

Is a

2. Run test(s). Ensure new test fails.

This ensures test is functioning,
and that failure is only in the new code

Test Driven Development

Is a

3. Write the function with minimal code to pass test(s)

4. Run test(s). Ensure new test passes.

5. Refactor (and re-run tests)

We need to implement the new function until we pass the test(s), refactoring code as we go

(this is usually a rinse-and-repeat sort of operation)

Test Driven Development

Is a

TDD	Makes code author think about code (setup, organization, etc.) **before** writing it

And the difference from strict unit tests

unittests	Is that unit tests are written after writing functions to ensure code functions as expected

Software Testing

unittests *versus* TDD

Regardless of your choice, unit testing and TDD greatly increase the

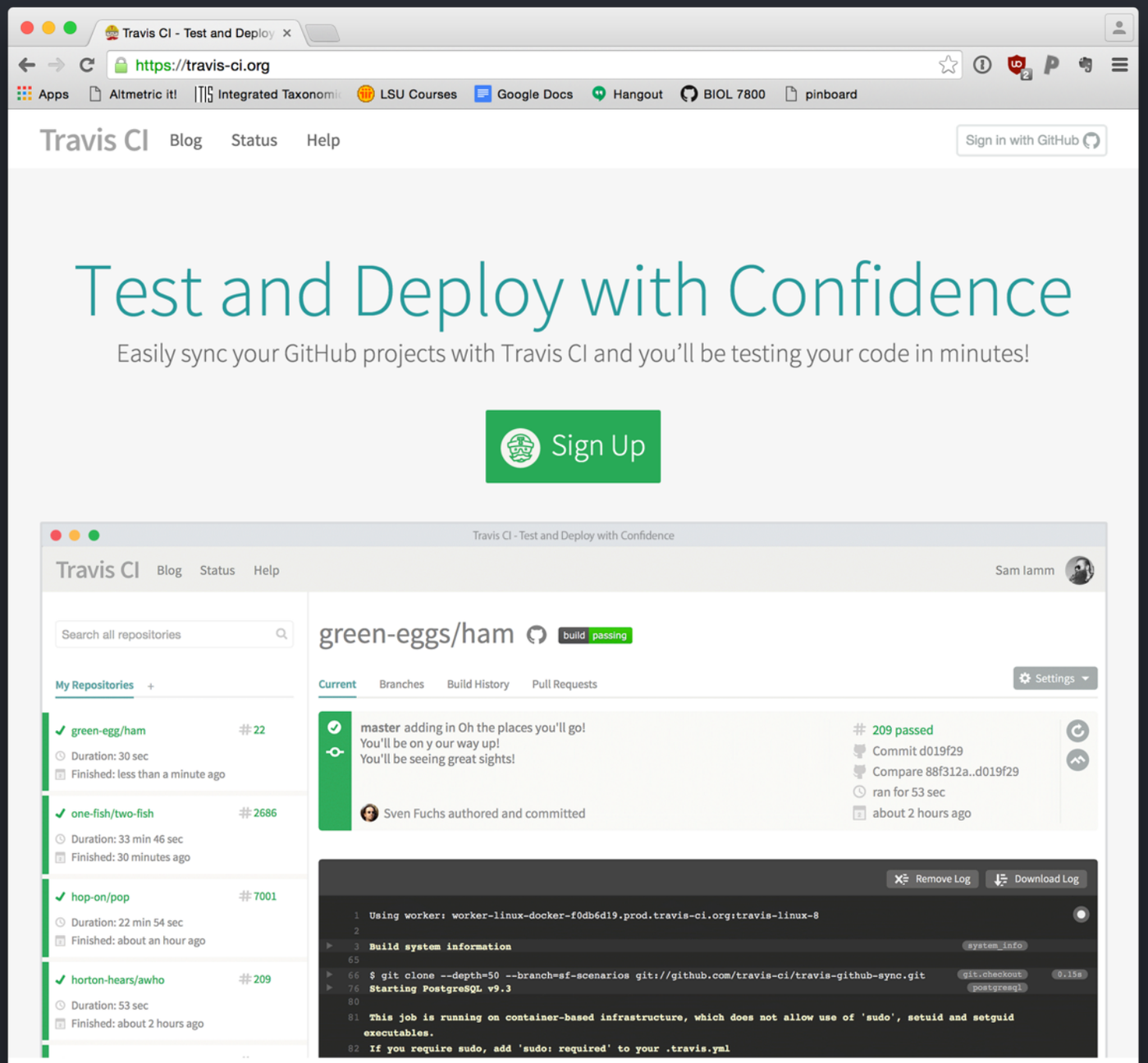
reliability repeatability usability extensibility

Of **your** code

And, **when starting a software project**, it's always easier to start with tests than it is to go back later….

Image: _____

Software Testing



The home of open source testing

Over 200k open source projects and 126k users are testing on Travis CI.

github

Allows you to run
automated software tests
after any commit

using **travis-ci**

Software Testing

faircloth-lab/illumiprocessor

GitHub, Inc. [US] https://github.com/faircloth-lab/illumiprocessor

Apps | Altmetric it! | Integrated Taxonomi | LSU Courses | Google Docs | Hangout | BIOL 7800 | pinboard

This repository Search | Pull requests | Issues | Gist

faircloth-lab / illumiprocessor | Watch 6 | Star 8 | Fork 5

Code | Issues 3 | Pull requests 1 | Wiki | Pulse | Graphs | Settings

pre-process illumina reads http://illumiprocessor.readthedocs.org/ — Edit

116 commits | 2 branches | 10 releases | 1 contributor

Branch: master | New pull request | New file | Upload files | Find file | SSH | git@github.com:faircloth- | Download ZIP

brantfaircloth Update to fix TruSeq LT adapter sequence which had too many GTGTs — Latest commit e58930a on Aug 4, 2015

bin	rearrange program structure for easier tests	2 years ago
docs	Update to fix TruSeq LT adapter sequence which had too many GTGTs	8 months ago
illumiprocessor	Alter check against digest of trimmomatic output	2 years ago
test	add additional tests of sequence object	2 years ago
.gitignore	ignore py.log files	2 years ago
.travis.yml	adding travis.yml	2 years ago
LICENSE.txt	add license	2 years ago
MANIFEST	add MANIFEST	2 years ago
README.md	update trimmomatic citation	2 years ago
pre-process-example.conf	update example file for current version	2 years ago
setup.py	bump version	2 years ago

README.md

illumiprocessor build passing

illumiprocessor is a tool to batch process illumina sequencing reads using the excellent trimmomatic package. The program takes a configuration file that is formatted in Microsoft Windows INI file format (key:value pairs, see the example file).

illumiprocessor will trim adapter contamination from SE and PE illumina reads and is capable of dealing with double-indexed reads and read trimming (example to come shortly). The current version of illumiprocessor uses trimmomatic instead of scythe and sickle (used in v1.x) because we have found the performance of trimmomatic to be better, particularly when dealing with double-indexed illumina reads. However, you may find that running scythe after trimming with illumiprocessor or trimmomatic ensures that every bit of potential adapter contamination is removed.

illumiprocessor is suited to parallel processing in which each set of illumina reads are processed on a separate (physical) compute core. **illumiprocessor assumes that all fastq files input to the program represent individuals samples (i.e.,**

Travis CI - Test and Deploy

https://travis-ci.org

Apps | Altmetric it! | Integrated Taxonomi | LSU Courses | Google Docs | Hangout | BIOL 7800 | pinboard

Travis CI | Blog | Status | Help | Sign in with GitHub

Test and Deploy with Confidence

Easily sync your GitHub projects with Travis CI and you'll be testing your code in minutes!

Sign Up

Travis CI - Test and Deploy with Confidence

Travis CI | Blog | Status | Help | Sam Iamm

Search all repositories | green-eggs/ham build passing

My Repositories | Current | Branches | Build History | Pull Requests | Settings

using travis-ci

Automatically tests code when it changes (on commit)

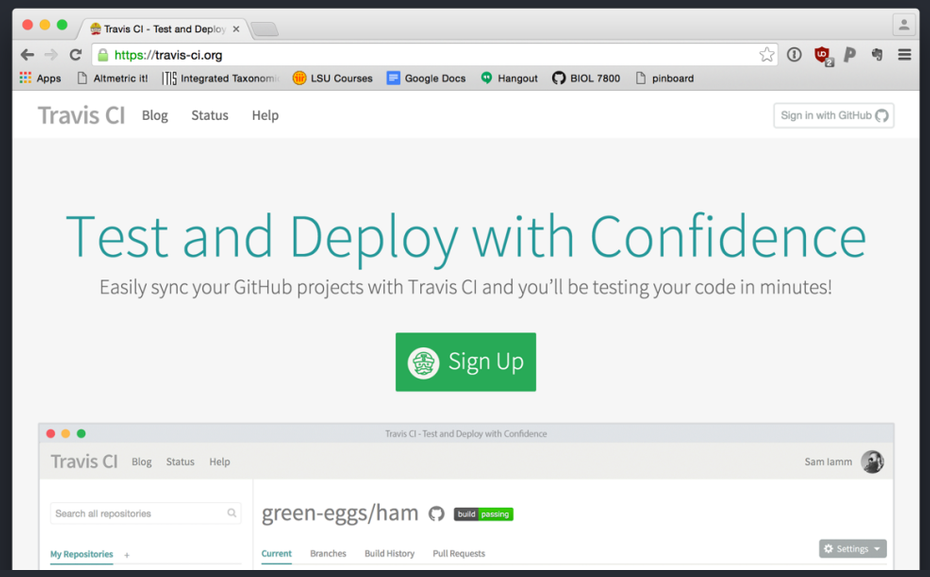
Software Testing

illumiprocessor/test_illumip

GitHub, Inc. [US] https://github.com/faircloth-lab/illumiprocessor/blob/master/test/test_illumiprocessor.py

Apps | Altmetric it! | Integrated Taxonomi | LSU Courses | Google Docs | Hangout | BIOL 7800 | pinboard

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
(c) 2014 Brant Faircloth || http://faircloth-lab.org/
All rights reserved.

This code is distributed under a 3-clause BSD license. Please see
LICENSE.txt for more information.

Created on 31 January 2014 12:36 PST (-0800)
"""


import os
import glob
import hashlib

import pdb


class TestGetTruHtReads:
    def test_enough_reads(self, fake_truht_reads):
        assert len(fake_truht_reads) == 2

    def test_correct_file_names(self, fake_truht_reads):
        expected_r1 = set([
            'fake-truht_S1_L001_R1_001.fastq.gz',
            'fake-truht_S2_L001_R1_001.fastq.gz'
        ])
        expected_r2 = set([
            'fake-truht_S1_L001_R2_001.fastq.gz',
            'fake-truht_S2_L001_R2_001.fastq.gz'
        ])
        observed_r1 = []
        observed_r2 = []
        for read in fake_truht_reads:
            observed_r1.extend([os.path.basename(r) for r in read.r1])
            observed_r2.extend([os.path.basename(r) for r in read.r2])
        assert set(observed_r1) == expected_r1
        assert set(observed_r2) == expected_r2


class TestS1SequenceData:
    def test_home_dir(self, s1):
        assert s1.homedir == os.path.join(
            os.path.dirname(__file__),
            "truht/clean/fake-truht1"
            )

    def test_s1_i5(self, s1):
        assert s1.i5 == 'i5-06_F'

    def test_s1_i5a(self, s1):
        assert s1.i5a == 'AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGTAGTTGGCTGTGTAGATCTCGGTGGTCGCCGTATCATT'

    def test_s1_i5s(self, s1):
        assert s1.i5s == 'AGTTGGCT'

    def test_s1_i5s_revcomp(self, s1):
        assert s1.i5s_revcomp is True

    def test_s1_i7(self, s1):
        assert s1.i7 == 'i7-09_11'
```



using travis-ci

Automatically tests code when it changes (on commit)

Free for open-source projects

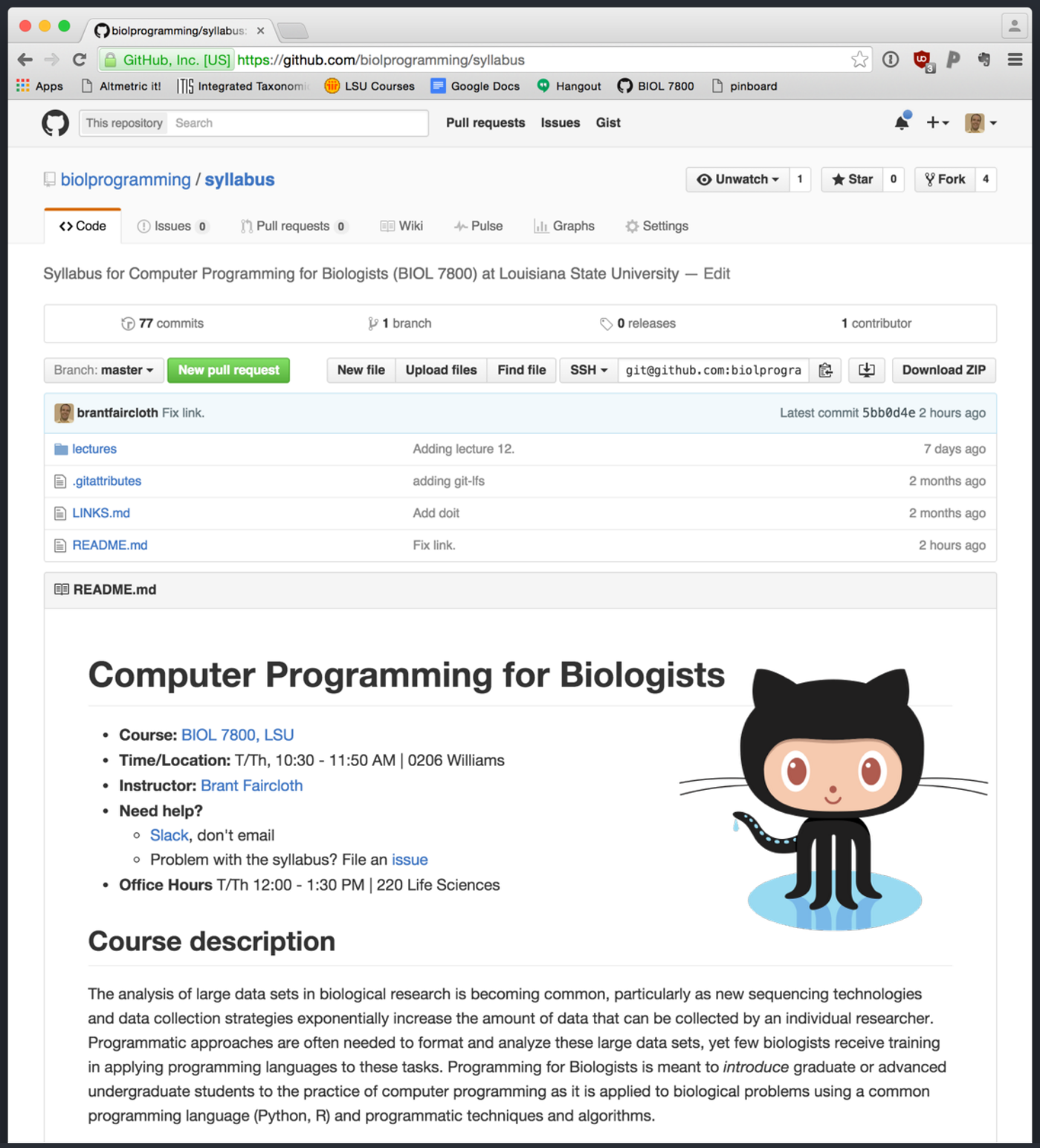
Documentation

One of the most under-appreciated and under-valued aspects of a good software package

Good documentation can **make** or **break** your application

(and determine whether you ever have any users)

Documentation



github

README.md

At top-level of a repository offers a nice, easy way to quickly document a smaller program or package.

These `README.md` documents are markdown formatted

Documentation

These **README.md** documents
are markdown formatted

README.md — /Users/bcf/github-app/biol7800/syllabus

README.md

```
# Computer Programming for Biologists

* **Course:** [BIOL 7800, LSU](http://catalog.lsu.edu/preview_course_nopop.php?catoid=1&coid=1001)
* **Time/Location:** T/Th, 10:30 - 11:50 AM | 0206 Williams
* **Instructor:** [Brant Faircloth](https://github.com/brantfaircloth/)
* **Need help?**
    * [Slack](https://biolprogramming.slack.com), don't email
    * Problem with the syllabus? File an [issue](https://github.com/biolprogramming/syllabus/issues)
* **Office Hours** T/Th 12:00 - 1:30 PM | 220 Life Sciences

## Course description

The analysis of large data sets in biological research is becoming common, particularly as new sequencing technologies and data collection strategies exponentially increase the amount of data that can be collected by an individual researcher.  Programmatic approaches are often needed to format and analyze these large data sets, yet few biologists receive training in applying programming languages to these tasks.  Programming for Biologists is meant to _introduce_ graduate or advanced undergraduate students to the practice of computer programming as it is applied to biological problems using a common programming language (Python, R) and programmatic techniques and algorithms.

## Course credo

This course **is** going to challenge _and_ frustrate you.  A lot.  I promise.  You are learning a new language really quickly - that's a hard thing to do.  Along with the hard parts of learning a new language, in this case, comes having to learn a number of new tools
```

https://raw.githubusercontent.com/biolprogramming/syllabus/master/README.md

Apps | Altmetric it! | Integrated Taxonomi | LSU Courses | Google Docs | Hangout | BIOL 7800 | pinboard

```
# Computer Programming for Biologists

* **Course:** [BIOL 7800, LSU]
(http://catalog.lsu.edu/preview_course_nopop.php?catoid=1&coid=1001)
* **Time/Location:** T/Th, 10:30 - 11:50 AM | 0206 Williams
* **Instructor:** [Brant Faircloth](https://github.com/brantfaircloth/)
* **Need help?**
    * [Slack](https://biolprogramming.slack.com), don't email
    * Problem with the syllabus? File an [issue]
(https://github.com/biolprogramming/syllabus/issues)
* **Office Hours** T/Th 12:00 - 1:30 PM | 220 Life Sciences

## Course description

The analysis of large data sets in biological research is becoming common,
particularly as new sequencing technologies and data collection strategies
exponentially increase the amount of data that can be collected by an
individual researcher.  Programmatic approaches are often needed to format
and analyze these large data sets, yet few biologists receive training in
applying programming languages to these tasks.  Programming for Biologists
is meant to _introduce_ graduate or advanced undergraduate students to the
practice of computer programming as it is applied to biological problems
using a common programming language (Python, R) and programmatic techniques
and algorithms.

## Course credo

This course **is** going to challenge _and_ frustrate you.  A lot.  I
promise.  You are learning a new language really quickly - that's a hard
thing to do.  Along with the hard parts of learning a new language, in this
case, comes having to learn a number of new tools that you have not
(likely) been exposed to.  That's also really hard.  You're also going to
have to actually **think** on top of all that. But, if you think, and work,
and collaborate with your classmates to understand what's going on, you
**will** end up learning much, much more in a shorter period of time than
you expected.

## Teaching philosophy / Communication

I'm here to _help_ you learn to program a computer.  It's up to you to
learn how to make that work _for you_.  I view my role as providing
guidance and direction and your role as using that guidance and direction
```

Documentation

e.g. seqtk

lh3/seqtk: Toolkit for proce

GitHub, Inc. [US] https://github.com/lh3/seqtk

Apps | Altmetric it! | Integrated Taxonomi | LSU Courses | Google Docs | Hangout | BIOL 7800 | pinboard

README.md

Introduction

Seqtk is a fast and lightweight tool for processing sequences in the FASTA or FASTQ format. It seamlessly parses both FASTA and FASTQ files which can also be optionally compressed by gzip.

Seqtk Examples

- Convert FASTQ to FASTA:

```
seqtk seq -a in.fq.gz > out.fa
```

- Convert ILLUMINA 1.3+ FASTQ to FASTA and mask bases with quality lower than 20 to lowercases (the 1st command line) or to `N` (the 2nd):

```
seqtk seq -aQ64 -q20 in.fq > out.fa
seqtk seq -aQ64 -q20 -n N in.fq > out.fa
```

- Fold long FASTA/Q lines and remove FASTA/Q comments:

```
seqtk seq -Cl60 in.fa > out.fa
```

- Convert multi-line FASTQ to 4-line FASTQ:

```
seqtk seq -l0 in.fq > out.fq
```

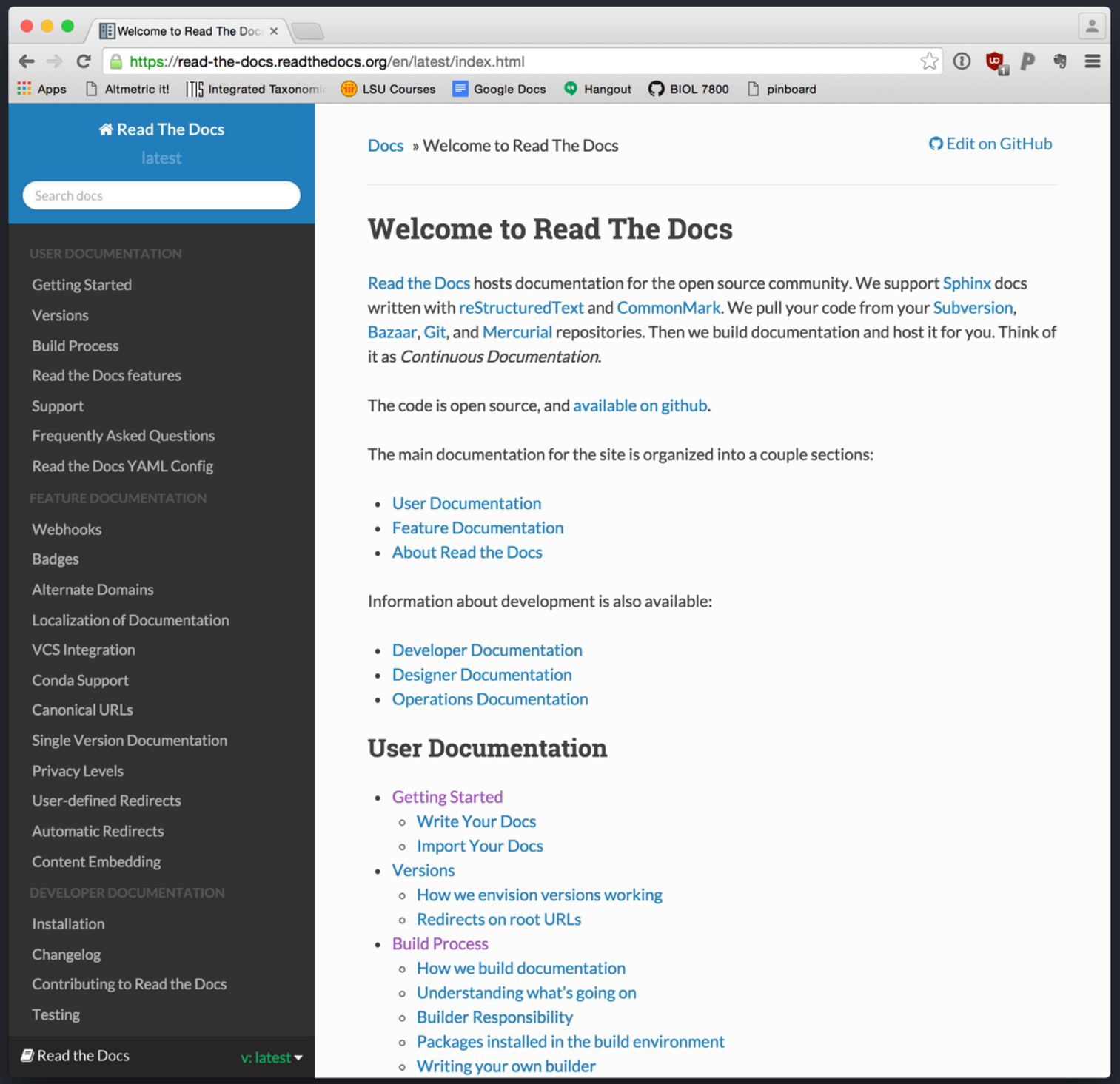
- Reverse complement FASTA/Q:

```
seqtk seq -r in.fq > out.fq
```

- Extract sequences with names in file `name.lst`, one sequence name per line:

Documentation

Read The Docs



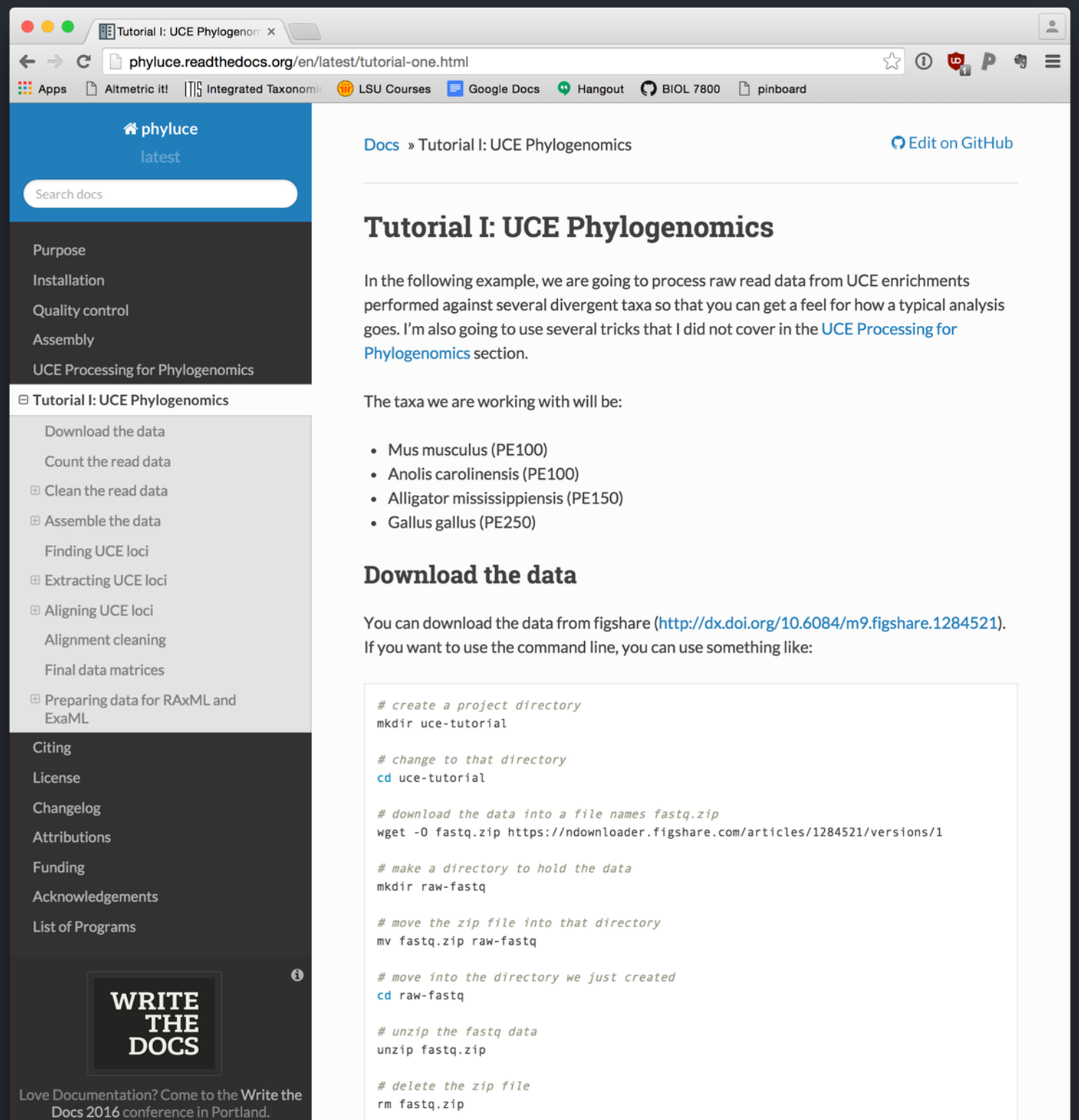
For **larger** programs and packages

Uses a **docs** directory in your repository

Automatically rebuilds documentation when it changes

Documentation

Read The Docs



Gives you a really nice HTML & pdf version of your docs, available online

Free for open-source projects