

# 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?

```
3
4 def reverse(my_string):
5     return my_string[::-1]
6
7
8 def my_sum(my_list):
9     if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
```



# Software Testing

Writing code to test **logical "units"** of your program

example14.py

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

```
2  
3  
4 def reverse(my_string):  
5     return my_string[::-1]
```

← unit

```
6  
7  
8 def my_sum(my_list):  
9     if not isinstance(my_list, list):  
10         raise IOError("Must input a list")  
11     else:  
12         return sum(my_list)
```

← unit

```
13  
14  
15 def power(num, power):  
16     return num**power
```

← unit

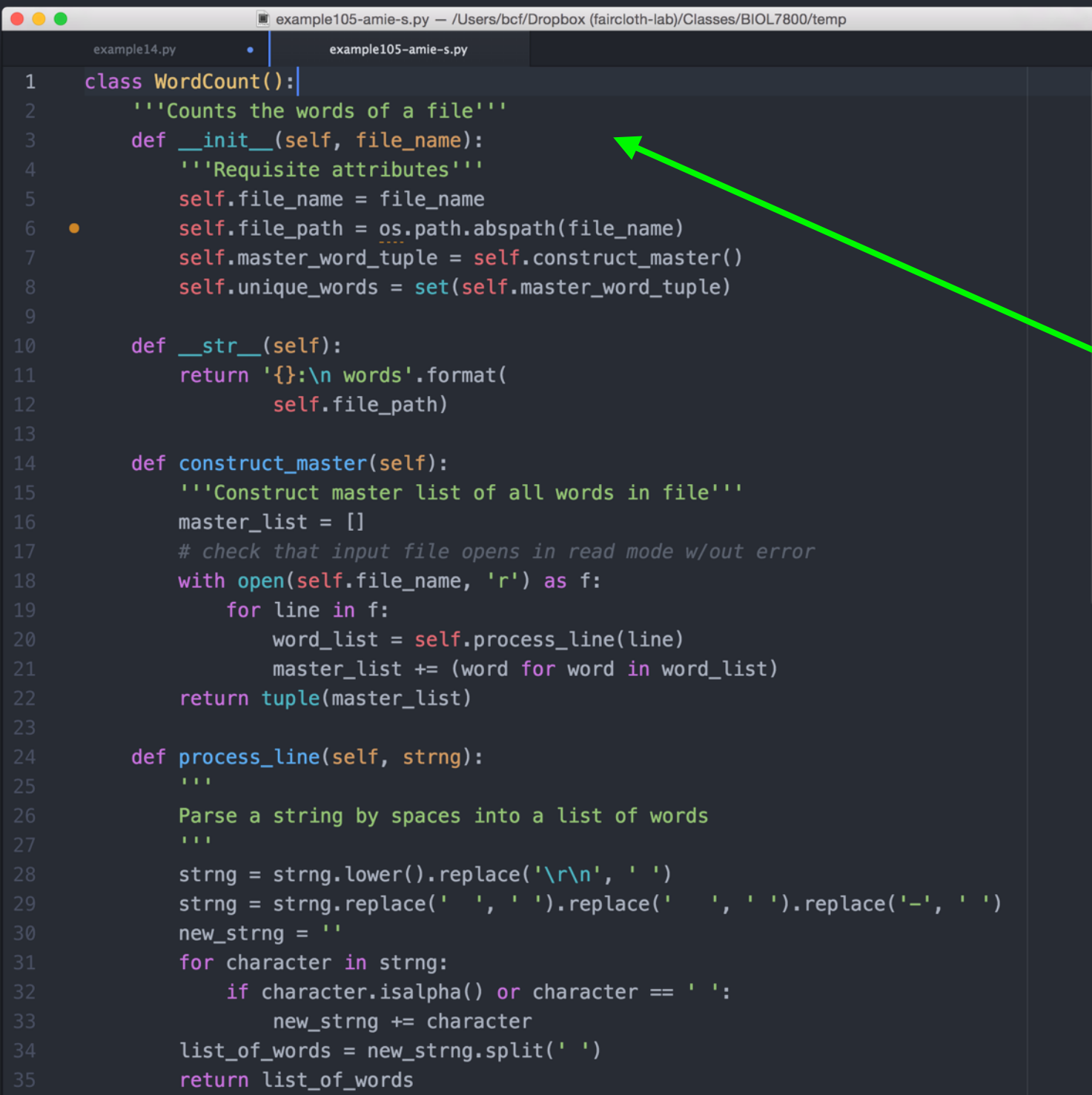
```
17  
18  
19 def main():  
20     my_string = 'able was i ere i saw elba'  
21     rev_my_string = reverse(my_string)  
22     print(rev_my_string)  
23     my_list = [2, 4, 6, 8]  
24     sum_my_list = my_sum(my_list)  
25     print(sum_my_list)  
26     my_power = power(20, 20)
```

Here, **functions** are the logical unit to test...



# Software Testing

Writing code to test logical "units" of your program



```
1 class WordCount():
2     '''Counts the words of a file'''
3     def __init__(self, file_name):
4         '''Requisite attributes'''
5         self.file_name = file_name
6         self.file_path = os.path.abspath(file_name)
7         self.master_word_tuple = self.construct_master()
8         self.unique_words = set(self.master_word_tuple)
9
10    def __str__(self):
11        return '{}:\n words'.format(
12            self.file_path)
13
14    def construct_master(self):
15        '''Construct master list of all words in file'''
16        master_list = []
17        # check that input file opens in read mode w/out error
18        with open(self.file_name, 'r') as f:
19            for line in f:
20                word_list = self.process_line(line)
21                master_list += (word for word in word_list)
22        return tuple(master_list)
23
24    def process_line(self, strng):
25        '''
26        Parse a string by spaces into a list of words
27        '''
28        strng = strng.lower().replace('\r\n', ' ')
29        strng = strng.replace(' ', ' ').replace('-', ' ').replace('_', ' ')
30        new_strng = ''
31        for character in strng:
32            if character.isalpha() or character == ' ':
33                new_strng += character
34        list_of_words = new_strng.split(' ')
35        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
1 class WordCount():
2     '''Counts the words of a file'''
3     def __init__(self, file_name):
4         '''Requisite attributes'''
5         self.file_name = file_name
6         self.file_path = os.path.abspath(file_name)
7         self.master_word_tuple = self.construct_master()
8         self.unique_words = set(self.master_word_tuple)
9
10    def __str__(self):
11        return '{}:\n words'.format(
12            self.file_path)
13
14    def construct_master(self):
15        '''Construct master list of all words in file'''
16        master_list = []
17        # check that input file opens in read mode w/out error
18        with open(self.file_name, 'r') as f:
19            for line in f:
20                word_list = self.process_line(line)
21                master_list += (word for word in word_list)
22        return tuple(master_list)
23
24    def process_line(self, strng):
25        '''
26        Parse a string by spaces into a list of words
27        '''
28        strng = strng.lower().replace('\r\n', ' ')
29        strng = strng.replace(' ', ' ').replace('-', ' ').replace('-', ' ')
30        new_strng = ''
31        for character in strng:
32            if character.isalpha() or character == ' ':
33                new_strng += character
34        list_of_words = new_strng.split(' ')
35        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
1 class WordCount():
2     '''Counts the words of a file'''
3     def __init__(self, file_name):
4         '''Requisite attributes'''
5         self.file_name = file_name
6         self.file_path = os.path.abspath(file_name)
7         self.master_word_tuple = self.construct_master()
8         self.unique_words = set(self.master_word_tuple)
9
10    def __str__(self):
11        return '{}:\n words'.format(
12            self.file_path)
13
14    def construct_master(self):
15        '''Construct master list of all words in file'''
16        master_list = []
17        # check that input file opens in read mode w/out error
18        with open(self.file_name, 'r') as f:
19            for line in f:
20                word_list = self.process_line(line)
21                master_list += (word for word in word_list)
22        return tuple(master_list)
23
24    def process_line(self, strng):
25        '''
26        Parse a string by spaces into a list of words
27        '''
28        strng = strng.lower().replace('\r\n', ' ')
29        strng = strng.replace(' ', ' ').replace('-', ' ')
30        new_strng = ''
31        for character in strng:
32            if character.isalpha() or character == ' ':
33                new_strng += character
34        list_of_words = new_strng.split(' ')
35        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

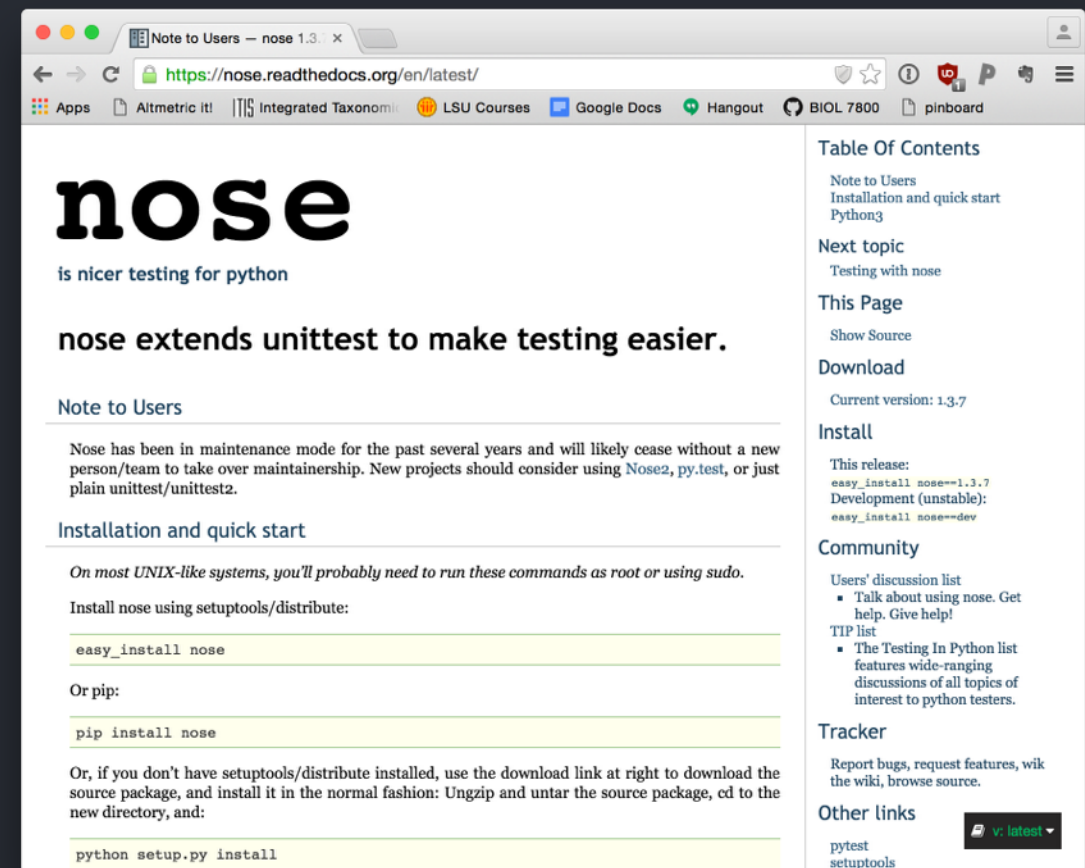
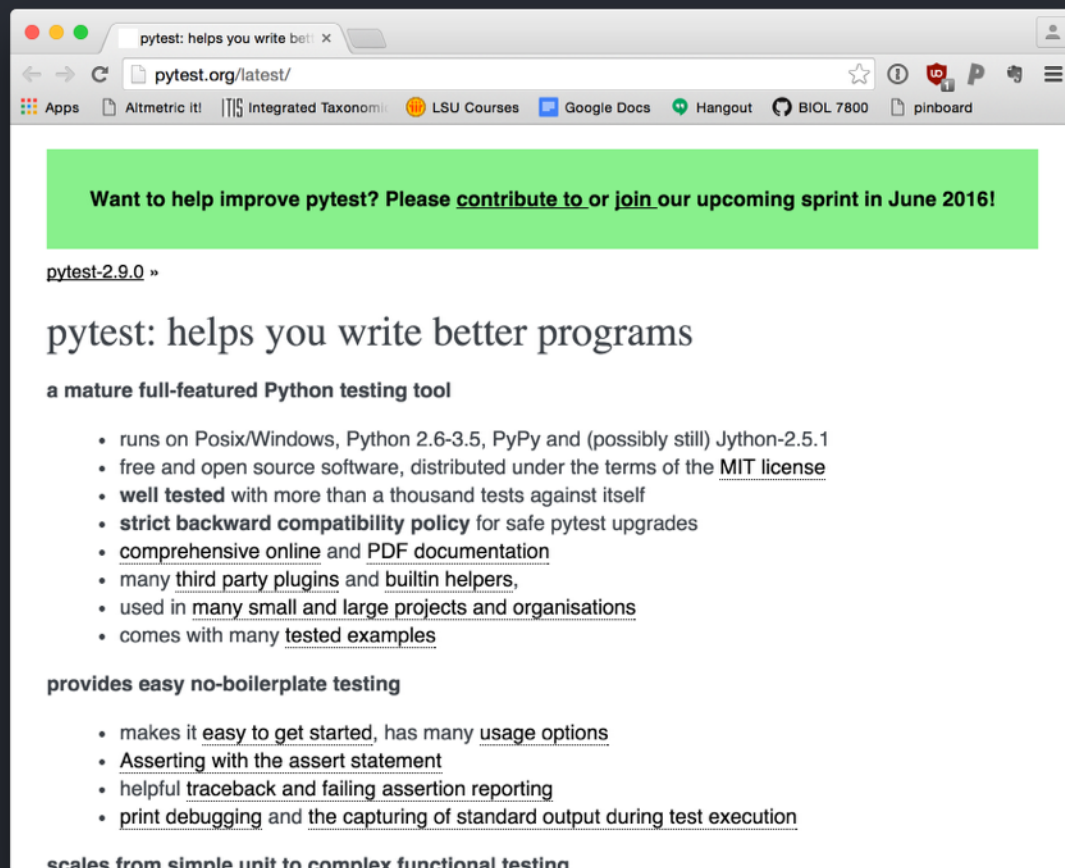


# Software Testing: unittest

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

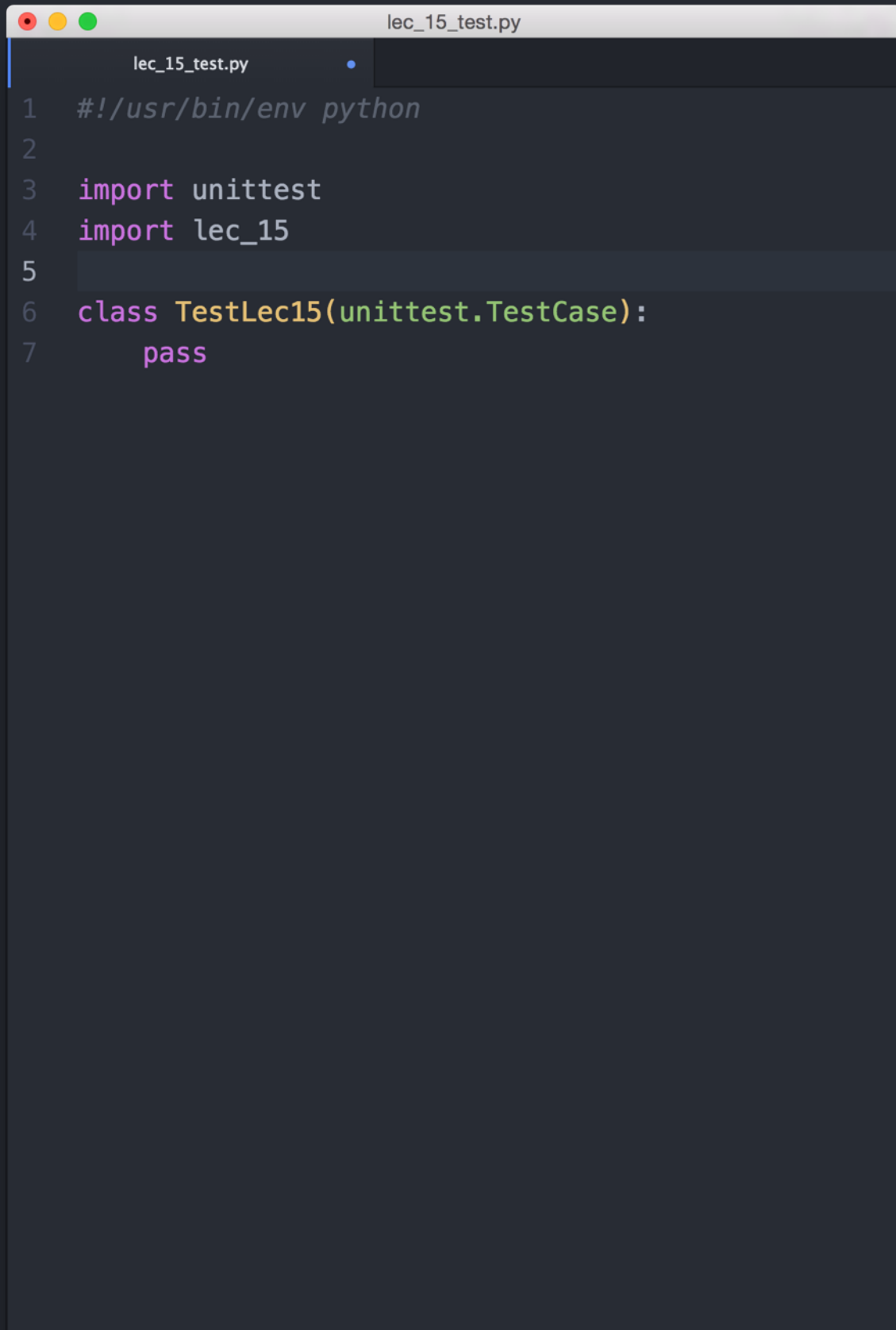
```
example14.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp
example14.py
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
```

← 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
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7      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
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7      pass
```

If we are testing  
usually name the file  
lec\_15\_test.py

We import **unittest**  
and our code to test,  
**lec\_15**

This is  
**important**

# Software Testing: **unittest**

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7      pass
```

If we are testing  
usually name the file  
**lec\_15\_test.py**

We import  
and our code to test,  
**lec\_15**

This is  
**important**

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
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7      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

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          pass
10
11     def test_my_sum(self):
12         pass
13
14     def test_power(self):
15         pass
16
```

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
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          pass
10
11     def test_my_sum(self):
12         pass
13
14     def test_power(self):
15         pass
16
17 if __name__ == '__main__':
18     main()
19
```

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

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          pass
10
11     def test_my_sum(self):
12         pass
13
14     def test_power(self):
15         pass
16
17 if __name__ == '__main__':
18     main()
19
```

\$ 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
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          pass
10
11     def test_my_sum(self):
12         pass
13
14     def test_power(self):
15         pass
16
17 if __name__ == '__main__':
18     unittest.main()
```

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

# Software Testing: unittest

lec\_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          pass
10
11     def test_my_sum(self):
12         pass
13
14     def test_power(self):
15         pass
16
17 if __name__ == '__main__':
18     unittest.main()
19
```

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

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          pass
10
11     def test_my_sum(self):
12         pass
13
14     def test_power(self):
15         pass
16
17 if __name__ == '__main__':
18     unittest.main()
19
```

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

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          pass
10
11     def test_my_sum(self):
12         pass
13
14     def test_power(self):
15         pass
16
17 if __name__ == '__main__':
18     unittest.main()
19
```

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

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'god'
11         self.assertEqual(observed, expected)
12
13     def test_my_sum(self):
14         pass
15
16     def test_power(self):
17         pass
18
19 if __name__ == '__main__':
20     unittest.main()
21
```

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

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'wrong'
11         self.assertEqual(observed, expected)
12
```

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

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'wrong'
11         self.assertEqual(observed, expected)
12
13     def test_my_sum(self):
14         pass
15
16     def test_power(self):
17         pass
18
19 if __name__ == '__main__':
20     unittest.main()
21
```

What happens when  
a test fails?

Will be wrong!!

..F ← Why did this test run 3rd?

=====

FAIL: test\_reverse (\_\_main\_\_.TestLec15)

-----

Traceback (most recent call last):  
File "lec\_15\_test.py", line 11, in test\_reverse  
self.assertEqual(observed, expected)



# Software Testing: unittest

lec\_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

```
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     return num**power
17
18
19 def main():
20     my_string = 'able was i ere i saw elba'
21     rev_my_string = reverse(my_string)
22     print(rev_my_string)
23     my_list = [2, 4, 6, 8]
24     sum_my_list = my_sum(my_list)
25     print(sum_my_list)
26     my_power = power(20, 20)
27     print(my_power)
28
29
30 if __name__ == '__main__':
31     main()
```

lec\_15\_test.py

```
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'god'
11         self.assertEqual(observed, expected)
12
13     def test_my_sum(self):
14         observed = lec_15.my_sum([1, 2, 3])
15         self.assertEqual(observed, 6)
16
17     def test_power(self):
18         observed = lec_15.power(2, 2)
19         self.assertEqual(observed, 4)
20
21 if __name__ == '__main__':
22     unittest.main()
```

```
23 $ python lec_15_test.py
```

...

Ran 3 tests in 0.000s

OK

Fill in rest of  
tests...

Success!!



# Software Testing

Ok, great. But why?

Helps you write better (more discrete) code  
smaller units are easier to test (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 build on and refactor older code  
tests often identify potential problems before they become problems



# 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
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15  def power(num, power):
16      if isinstance(num, str):
17         num = int(num)
18      if isinstance(power, str):
19         power = int(power)
20      return num**power
21
22
23  def main():
24      my_string = 'able was i ere i saw elba'
25      rev_my_string = reverse(my_string)
26      print(rev_my_string)
27      my_list = [2, 4, 6, 8]
28      sum_my_list = my_sum(my_list)
29      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
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     if isinstance(num, str):
17         num = int(num)
18     if isinstance(power, str):
19         power = int(power)
20     return num**power
21
22
23 def main():
24     my_string = 'able was i ere i saw elba'
25     rev_my_string = reverse(my_string)
26     print(rev_my_string)
27     my_list = [2, 4, 6, 8]
28     sum_my_list = my_sum(my_list)
29     print(sum_my_list)
30     my_power = power(20, 20)
31     print(my_power)
32
```

lec\_15\_test.py

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'god'
11         self.assertEqual(observed, expected)
12
13     def test_my_sum(self):
14         observed = lec_15.my_sum([1, 2, 3])
15         self.assertEqual(observed, 6)
16
17     def test_power(self):
18         observed = lec_15.power(2, 2)
19         self.assertEqual(observed, 4)
20
21 if __name__ == '__main__':
22     unittest.main()
23
```

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
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     if isinstance(num, str):
17         num = int(num)
18     if isinstance(power, str):
19         power = int(power)
20     return num**power
21
22
23 def main():
24     my_string = 'able was i ere i saw elba'
25     rev_my_string = reverse(my_string)
26     print(rev_my_string)
27     my_list = [2, 4, 6, 8]
28     sum_my_list = my_sum(my_list)
29     print(sum_my_list)
30     my_power = power(20, 20)
31     print(my_power)
32
```

lec\_15\_test.py

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'god'
11         self.assertEqual(observed, expected)
12
13     def test_my_sum(self):
14         observed = lec_15.my_sum([1, 2, 3])
15         self.assertEqual(observed, 6)
16
17     def test_power(self):
18         observed = lec_15.power(2, 2)
19         self.assertEqual(observed, 4)
20         observed = lec_15.power('2', '2')
21         self.assertEqual(observed, 4)
22
23 if __name__ == '__main__':
24     unittest.main()
25
```

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

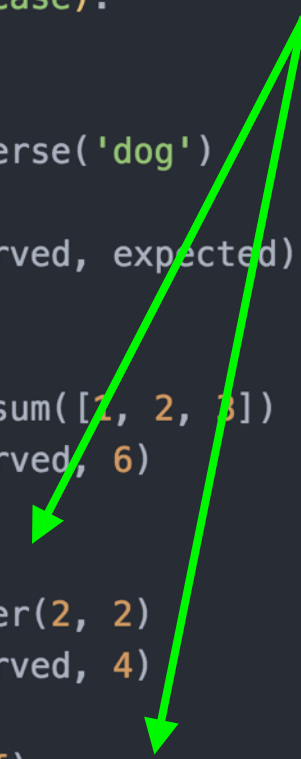
lec\_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

```
lec_15.py
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     if isinstance(num, str):
17         num = int(num)
18     if isinstance(power, str):
19         power = int(power)
20     return num**power
21
22
23 def main():
24     my_string = 'able was i ere i saw elba'
25     rev_my_string = reverse(my_string)
26     print(rev_my_string)
27     my_list = [2, 4, 6, 8]
28     sum_my_list = my_sum(my_list)
29     print(sum_my_list)
30     my_power = power(20, 20)
31     print(my_power)
32
```

lec\_15\_test.py

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'god'
11         self.assertEqual(observed, expected)
12
13     def test_my_sum(self):
14         observed = lec_15.my_sum([1, 2, 3])
15         self.assertEqual(observed, 6)
16
17     def test_power_int(self):
18         observed = lec_15.power(2, 2)
19         self.assertEqual(observed, 4)
20
21     def test_power_string(self):
22         observed = lec_15.power('2', '2')
23         self.assertEqual(observed, 4)
24
25 if __name__ == '__main__':
26     unittest.main()
27
```

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

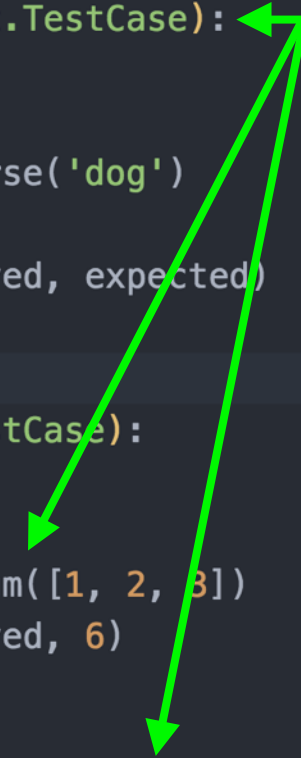
lec\_15.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

```
lec_15.py
1  #!/usr/bin/env python
2
3
4  def reverse(my_string):
5      return my_string[::-1]
6
7
8  def my_sum(my_list):
9      if not isinstance(my_list, list):
10         raise IOError("Must input a list")
11     else:
12         return sum(my_list)
13
14
15 def power(num, power):
16     if isinstance(num, str):
17         num = int(num)
18     if isinstance(power, str):
19         power = int(power)
20     return num**power
21
22
23 def main():
24     my_string = 'able was i ere i saw elba'
25     rev_my_string = reverse(my_string)
26     print(rev_my_string)
27     my_list = [2, 4, 6, 8]
28     sum_my_list = my_sum(my_list)
29     print(sum_my_list)
30     my_power = power(20, 20)
31     print(my_power)
32
```

lec\_15\_test.py

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15Reverse(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'god'
11         self.assertEqual(observed, expected)
12
13
14  class TestLec15Sum(unittest.TestCase):
15
16      def test_my_sum(self):
17          observed = lec_15.my_sum([1, 2, 3])
18          self.assertEqual(observed, 6)
19
20
21  class TestLec15Power(unittest.TestCase):
22
23      def test_power_int(self):
24          observed = lec_15.power(2, 2)
25          self.assertEqual(observed, 4)
26
27      def test_power_string(self):
28          observed = lec_15.power('2', '2')
29          self.assertEqual(observed, 4)
30
31  if __name__ == '__main__':
32      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

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
```

```
lec_15_test.py
1  #!/usr/bin/env python
2
3  import unittest
4  import lec_15
5
6  class TestLec15Reverse(unittest.TestCase):
7
8      def test_reverse(self):
9          observed = lec_15.reverse('dog')
10         expected = 'god'
11         self.assertEqual(observed, expected)
12
13
14  class TestLec15Sum(unittest.TestCase):
15
16      def test_my_sum(self):
17          observed = lec_15.my_sum([1, 2, 3])
18          self.assertEqual(observed, 6)
19
20
21  class TestLec15Power(unittest.TestCase):
22      def test_power_int(self):
23          observed = lec_15.power(2, 2)
24          self.assertEqual(observed, 4)
25
26      def test_power_string(self):
27          observed = lec_15.power('2', '2')
28          self.assertEqual(observed, 4)
29
30  if __name__ == '__main__':
31      unittest.main()
32
```



A blue pencil with a sharpened lead tip lies diagonally across the frame. The background is a light-colored surface with a grid of numbers. Some numbers are in green, while others are in blue. The numbers are arranged in a pattern that suggests a multiplication table or a similar mathematical grid. The pencil is positioned over the grid, with its tip pointing towards the bottom right.

# TDD

# Test Driven Development

# Test Driven Development

Is a **method** or a **mode** of software development

example14.py

```
#!/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....



# Software Testing

github

Allows you to run  
**automated software tests**  
after any commit

← using **travis-ci**

The screenshot shows the Travis CI website interface. The top section features the Travis CI logo and navigation links (Blog, Status, Help). A large heading reads "Test and Deploy with Confidence" with a subtext "Easily sync your GitHub projects with Travis CI and you'll be testing your code in minutes!". A green "Sign Up" button is prominent. Below this, a section titled "My Repositories" lists several projects: green-egg/ham, one-fish/two-fish, hop-on/pop, and horton-hears/awho, each with a status icon and build count. The main content area displays the details for the "green-eggs/ham" repository, showing a "build passing" status. It includes a commit message, author information (Sven Fuchs), and a list of build statistics (209 passed, Commit d019f29, Compare 88f312a..d019f29, ran for 53 sec, about 2 hours ago). A terminal log snippet is visible at the bottom, showing system information and build steps like "git clone" and "Starting PostgreSQL v9.3".

Travis CI - Test and Deploy with Confidence

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 Blog Status Help

Search all repositories

My Repositories

- ✓ green-egg/ham #22  
Duration: 30 sec  
Finished: less than a minute ago
- ✓ one-fish/two-fish #2686  
Duration: 33 min 46 sec  
Finished: 30 minutes ago
- ✓ hop-on/pop #7001  
Duration: 22 min 54 sec  
Finished: about an hour ago
- ✓ horton-hears/awho #209  
Duration: 53 sec  
Finished: about 2 hours ago

green-eggs/ham build passing

Current Branches Build History Pull Requests Settings

master adding in Oh the places you'll go!  
You'll be on your way up!  
You'll be seeing great sights!

Sven Fuchs authored and committed

# 209 passed  
Commit d019f29  
Compare 88f312a..d019f29  
ran for 53 sec  
about 2 hours ago

Remove Log Download Log

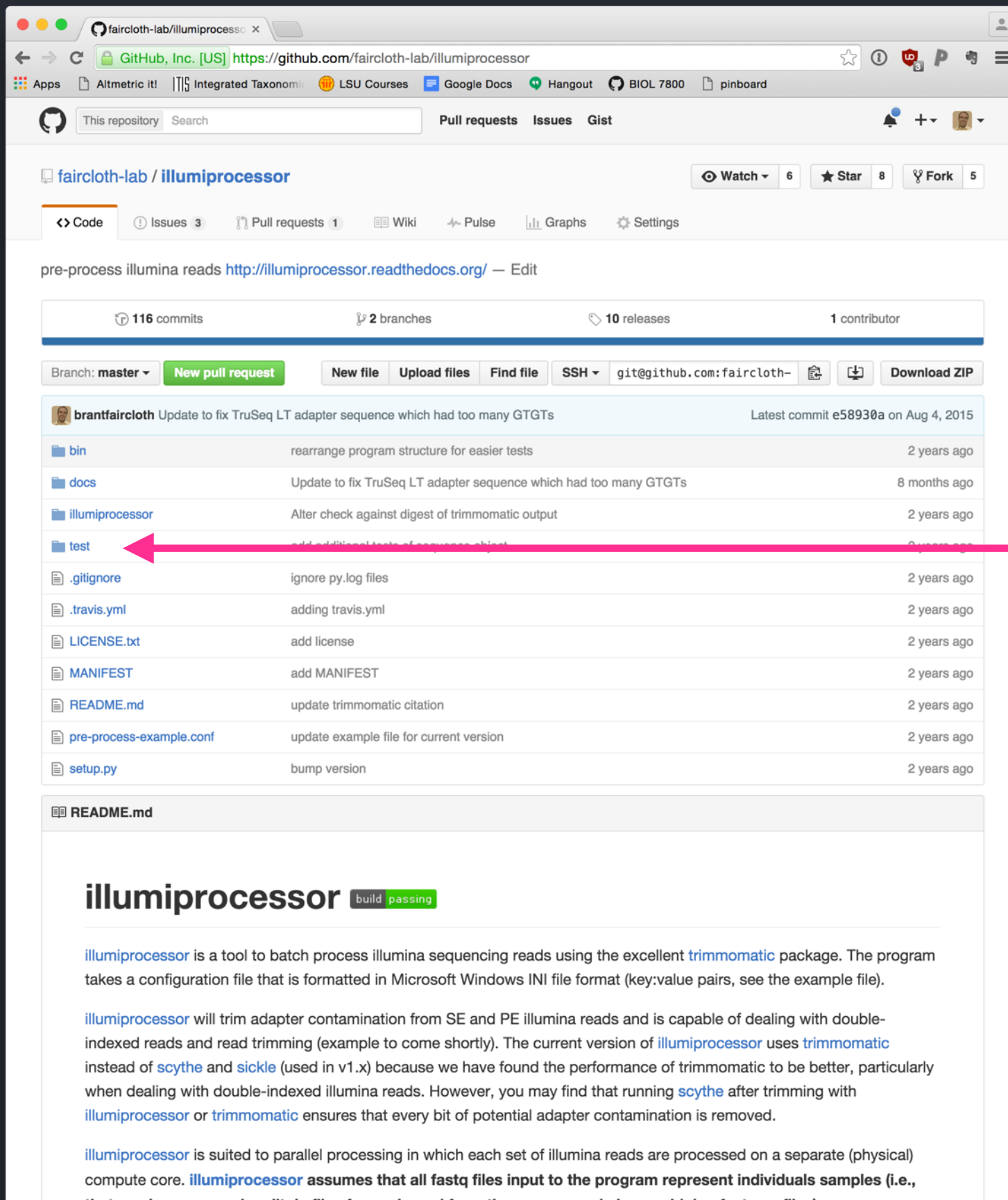
```
1 Using worker: worker-linux-docker-f0db6d19.prod.travis-ci.org:travis-linux-8
2
3 Build system information system_info
65
66 $ git clone --depth=50 --branch=sf-scenarios git://github.com/travis-ci/travis-github-sync.git git_checkout 0.15s
76 Starting PostgreSQL v9.3 postgresql
80
81 This job is running on container-based infrastructure, which does not allow use of 'sudo', setuid and setgid
  executables.
82 If you require sudo, add 'sudo: required' to your .travis.yml
```

The home of open source testing

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



# Software Testing



faircloth-lab / **illumiprocessor**

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	added 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

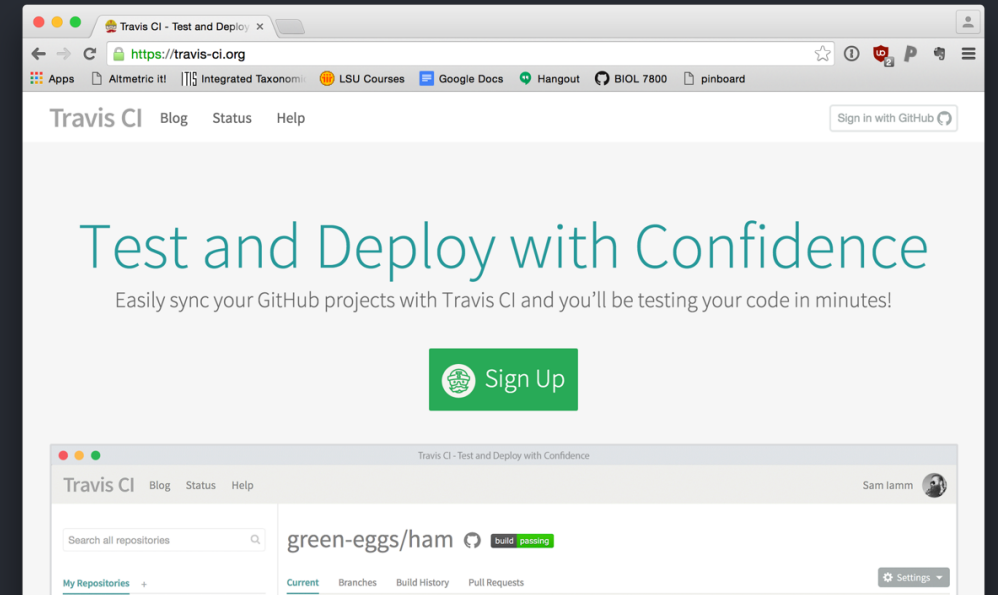
README.md

## 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 individual samples (i.e.,



Travis CI - Test and Deploy

## 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 Blog Status Help

Search all repositories

green-eggs/ham build passing

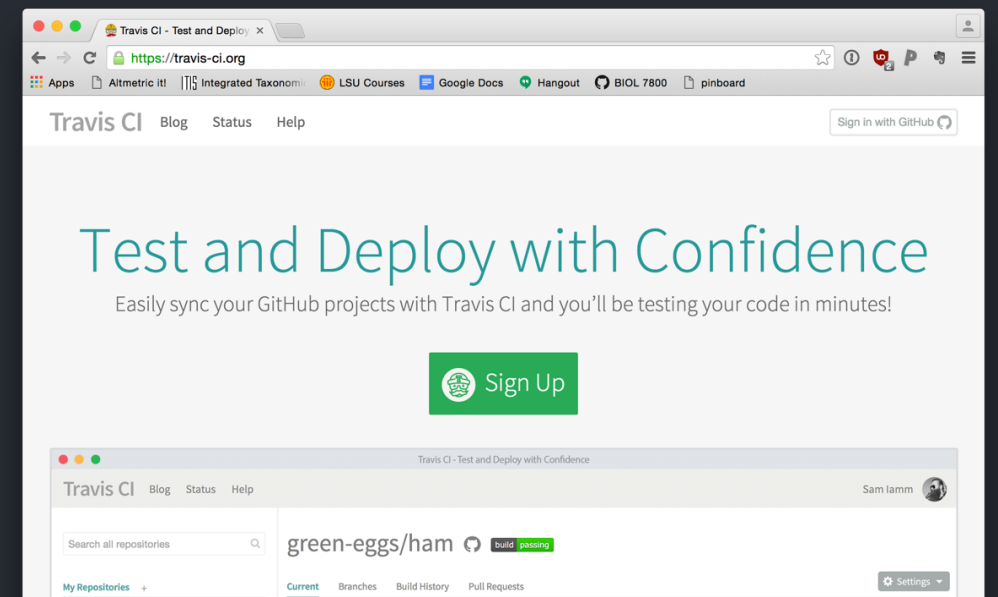
My Repositories Current Branches Build History Pull Requests Settings

using **travis-ci**

**Automatically tests** code  
when it changes  
(on commit)

# Software Testing

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 """
5 (c) 2014 Brant Faircloth || http://faircloth-lab.org/
6 All rights reserved.
7
8 This code is distributed under a 3-clause BSD license. Please see
9 LICENSE.txt for more information.
10
11 Created on 31 January 2014 12:36 PST (-0800)
12 """
13
14 import os
15 import glob
16 import hashlib
17
18 import pdb
19
20
21 class TestGetTruHtReads:
22     def test_enough_reads(self, fake_truht_reads):
23         assert len(fake_truht_reads) == 2
24
25     def test_correct_file_names(self, fake_truht_reads):
26         expected_r1 = set([
27             'fake-truht_S1_L001_R1_001.fastq.gz',
28             'fake-truht_S2_L001_R1_001.fastq.gz'
29         ])
30         expected_r2 = set([
31             'fake-truht_S1_L001_R2_001.fastq.gz',
32             'fake-truht_S2_L001_R2_001.fastq.gz'
33         ])
34         observed_r1 = []
35         observed_r2 = []
36         for read in fake_truht_reads:
37             observed_r1.extend([os.path.basename(r) for r in read.r1])
38             observed_r2.extend([os.path.basename(r) for r in read.r2])
39         assert set(observed_r1) == expected_r1
40         assert set(observed_r2) == expected_r2
41
42
43 class TestS1SequenceData:
44     def test_home_dir(self, s1):
45         assert s1.homedir == os.path.join(
46             os.path.dirname(__file__),
47             "truht/clean/fake-truht1"
48         )
49
50     def test_s1_i5(self, s1):
51         assert s1.i5 == 'i5-06_F'
52
53     def test_s1_i5a(self, s1):
54         assert s1.i5a == 'AGATCGGAAGAGCGCTGTAGGGAAAGAGTGTAGTTGGCTGTGTAGATCTCGGTGGTCGCCGTATCATT'
55
56     def test_s1_i5s(self, s1):
57         assert s1.i5s == 'AGTTGGCT'
58
59     def test_s1_i5s_revcomp(self, s1):
60         assert s1.i5s_revcomp is True
61
62     def test_s1_i7(self, s1):
63         assert s1.i7 == 'i7-00_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)



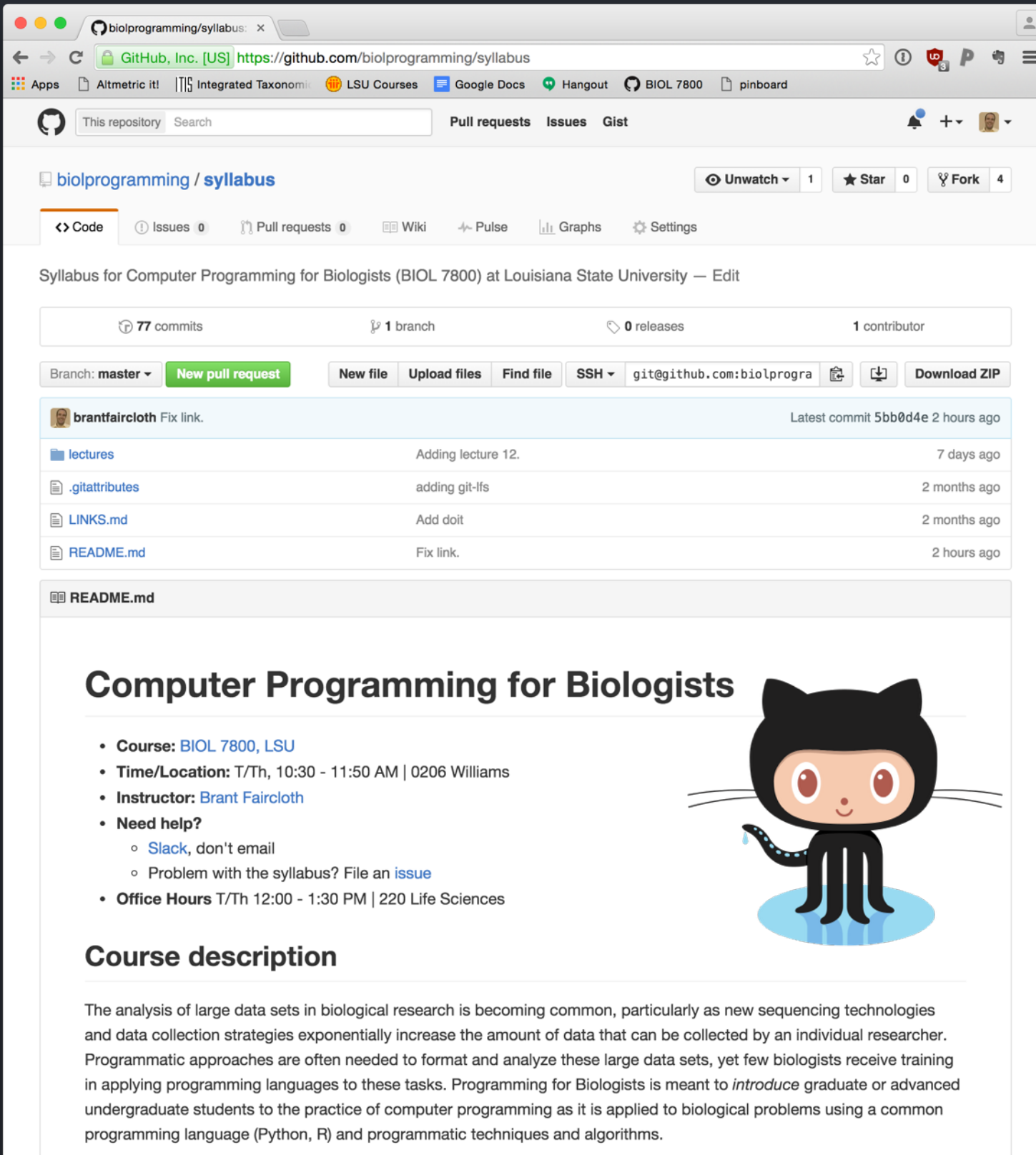
# Documentation



## 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

A screenshot of a web browser showing a GitHub repository page for 'biolprogramming/syllabus'. The browser's address bar shows the URL 'https://github.com/biolprogramming/syllabus'. The repository page includes a search bar, navigation tabs for 'Code', 'Issues', 'Pull requests', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. Below these, it shows repository statistics: 77 commits, 1 branch, 0 releases, and 1 contributor. A table of recent commits is visible, with the latest commit by 'brantfaircloth' titled 'Fix link.' at 2 hours ago. The 'README.md' file is selected, showing its content. The README is titled 'Computer Programming for Biologists' and includes a bulleted list of course details: Course (BIOL 7800, LSU), Time/Location (T/Th, 10:30 - 11:50 AM | 0206 Williams), Instructor (Brant Faircloth), Need help? (Slack, don't email; Problem with the syllabus? File an issue), and Office Hours (T/Th 12:00 - 1:30 PM | 220 Life Sciences). A cartoon illustration of the GitHub Octocat is positioned to the right of the course details. Below the list is a section titled 'Course description' with a paragraph of text about the course's focus on data analysis and programming in biology.

biolprogramming / syllabus

Unwatch 1 Star 0 Fork 4

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Syllabus for Computer Programming for Biologists (BIOL 7800) at Louisiana State University — Edit

77 commits 1 branch 0 releases 1 contributor

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


brantfaircloth Fix link. Latest commit 5bb0d4e 2 hours ago

lectures	Adding lecture 12.	7 days ago
.gitattributes	adding git-lfs	2 months ago
LINKS.md	Add doit	2 months ago
README.md	Fix link.	2 hours ago

README.md

## Computer Programming for Biologists

- **Course:** [BIOL 7800, LSU](#)
- **Time/Location:** T/Th, 10:30 - 11:50 AM | 0206 Williams
- **Instructor:** [Brant Faircloth](#)
- **Need help?**
  - [Slack](#), don't email
  - Problem with the syllabus? File an [issue](#)
- **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.

# Documentation

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

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

README.md
1 # Computer Programming for Biologists
2
3 * **Course:** [BIOL 7800,
4   • LSU](http://catalog.lsu.edu/preview_course_nopop.php?catoid=1&coid=10
5   • 01)
6 * **Time/Location:** T/Th, 10:30 – 11:50 AM | 0206 Williams
7 * **Instructor:** [Brant
8   • Faircloth](https://github.com/brantfaircloth/)
9 * **Need help?**
10   • [Slack](https://biolprogramming.slack.com), don't email
11   • Problem with the syllabus? File an
12   • [issue](https://github.com/biolprogramming/syllabus/issues)
13 * **Office Hours** T/Th 12:00 – 1:30 PM | 220 Life Sciences
14
15 ## Course description
16
17 The analysis of large data sets in biological research is becoming
18 • common, particularly as new sequencing technologies and data
19 • collection strategies exponentially increase the amount of data that
20 • can be collected by an individual researcher. Programmatic
21 • approaches are often needed to format and analyze these large data
22 • sets, yet few biologists receive training in applying programming
23 • languages to these tasks. Programming for Biologists is meant to
24 • introduce graduate or advanced undergraduate students to the
25 • practice of computer programming as it is applied to biological
26 • problems using a common programming language (Python, R) and
27 • programmatic techniques and algorithms.
28
29 ## Course credo
30
31 This course is going to challenge and frustrate you. A lot. I
32 • promise. You are learning a new language really quickly – that's a
33 • hard thing to do. Along with the hard parts of learning a new
34 • language, in this case, comes having to learn a number of new tools
35 •
```

```
https://raw.githubusercontent.com/biolprogramming/syllabus/master/README.md
https://raw.githubusercontent.com/biolprogramming/syllabus/master/README.md
← → ↻ 🔍 ☆ ⓘ ⚠ P ⚙
Apps Altmetric IT Integrated Taxonomy 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

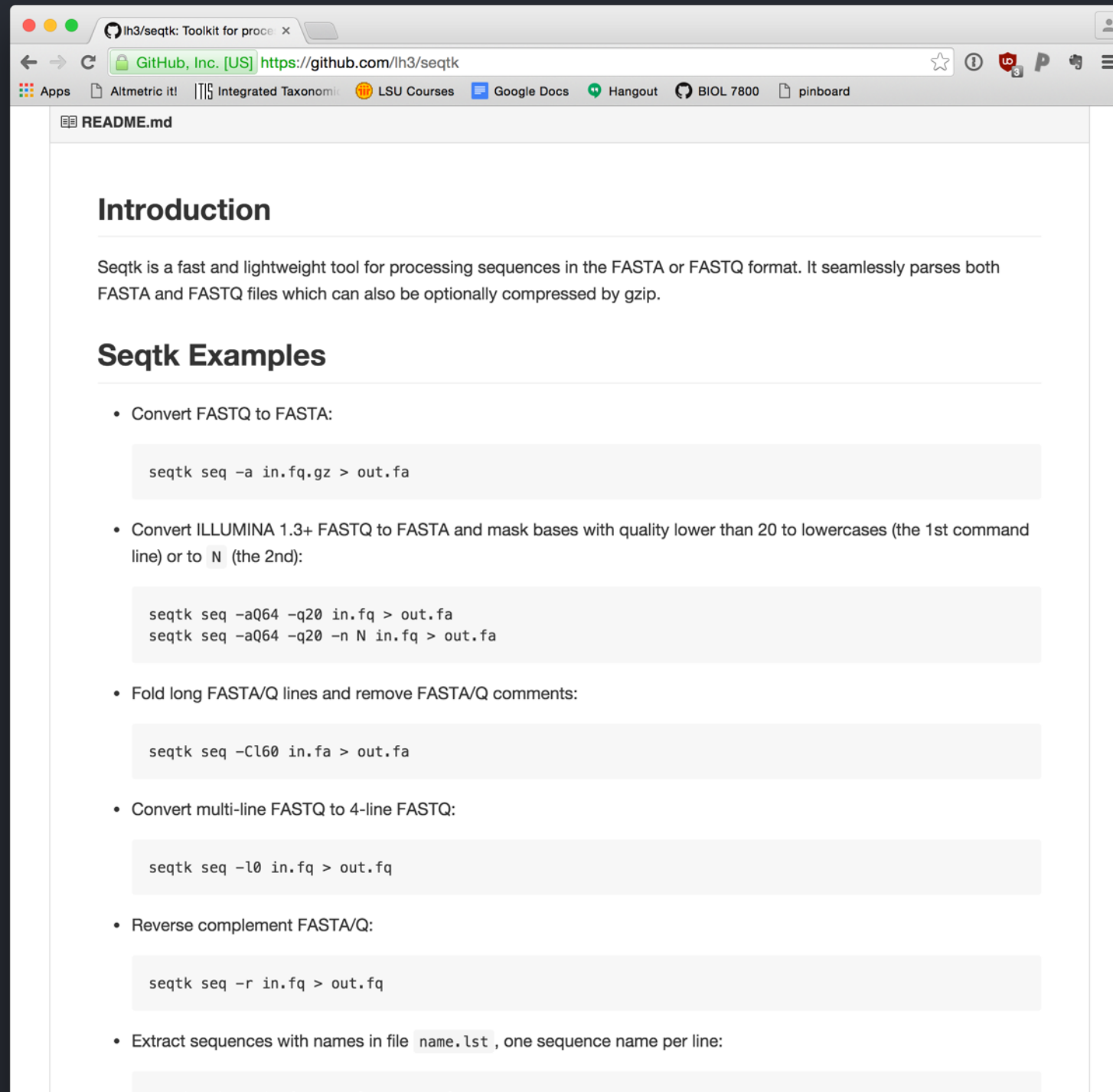
This course is going to challenge and frustrate you. A lot. I
promise. You are learning a new language really quickly – that's a hard
thing to do. Along with the hard parts of learning a new language, in this
case, comes having to learn a number of new tools that you have not
(likely) been exposed to. 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



The screenshot shows a web browser window displaying the GitHub repository for seqtk. The browser's address bar shows the URL <https://github.com/lh3/seqtk>. The page title is "lh3/seqtk: Toolkit for processing sequences". The browser's tab bar shows several open tabs, including "Apps", "Altmetric it!", "Integrated Taxonomi", "LSU Courses", "Google Docs", "Hangout", "BIOL 7800", and "pinboard". The main content of the page is the README.md file, which is titled "Introduction". The introduction text states: "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." Below the introduction is a section titled "Seqtk Examples". This section contains a list of examples, each with a corresponding command-line instruction in a code block. The examples are: 1. Convert FASTQ to FASTA: `seqtk seq -a in.fq.gz > out.fa`. 2. 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` and `seqtk seq -aQ64 -q20 -n N in.fq > out.fa`. 3. Fold long FASTA/Q lines and remove FASTA/Q comments: `seqtk seq -Cl60 in.fa > out.fa`. 4. Convert multi-line FASTQ to 4-line FASTQ: `seqtk seq -l0 in.fq > out.fq`. 5. Reverse complement FASTA/Q: `seqtk seq -r in.fq > out.fq`. 6. Extract sequences with names in file `name.lst`, one sequence name per line: (The command is not shown in the screenshot).

lh3/seqtk: Toolkit for processing sequences

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:

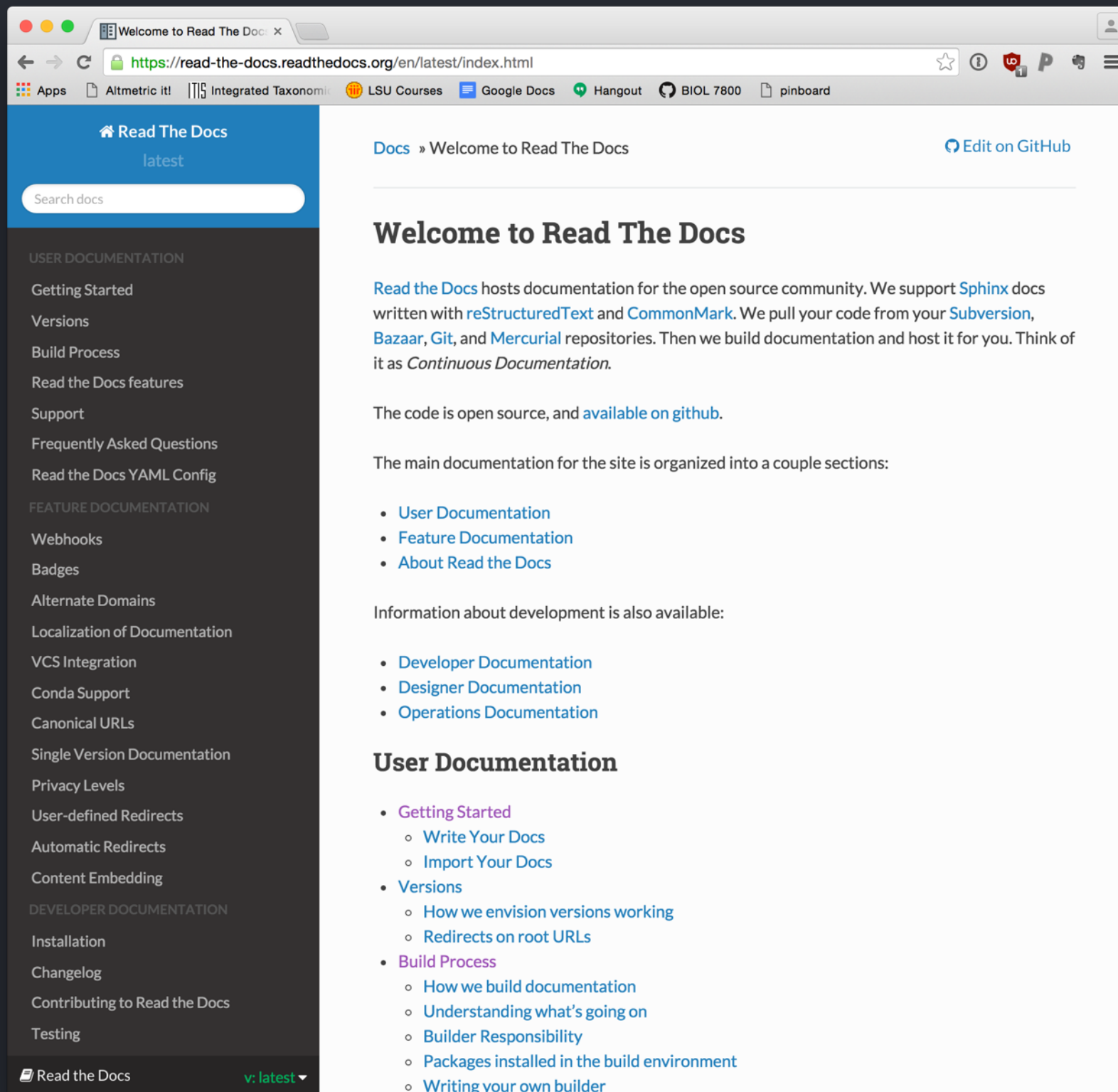
```
seqtk seq -l0 in.fq > out.fq
```
- Reverse complement FASTA/Q:

```
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

The screenshot shows a web browser window with the URL `phyluce.readthedocs.org/en/latest/tutorial-one.html`. The page title is "Tutorial I: UCE Phylogenomics". The left sidebar contains a navigation menu with links to "phyluce latest", "Search docs", "Purpose", "Installation", "Quality control", "Assembly", "UCE Processing for Phylogenomics", "Tutorial I: UCE Phylogenomics", "Download the data", "Count the read data", "Clean the read data", "Assemble the data", "Finding UCE loci", "Extracting UCE loci", "Aligning UCE loci", "Alignment cleaning", "Final data matrices", "Preparing data for RAxML and ExaML", "Citing", "License", "Changelog", "Attributions", "Funding", "Acknowledgements", and "List of Programs". The main content area has a header "Docs » Tutorial I: UCE Phylogenomics" with an "Edit on GitHub" link. The title "Tutorial I: UCE Phylogenomics" is followed by a paragraph: "In the following example, we are going to process raw read data from UCE enrichments performed against several divergent taxa so that you can get a feel for how a typical analysis goes. I'm also going to use several tricks that I did not cover in the [UCE Processing for Phylogenomics](#) section." Below this, it says "The taxa we are working with will be:" followed by a list: "Mus musculus (PE100)", "Anolis carolinensis (PE100)", "Alligator mississippiensis (PE150)", and "Gallus gallus (PE250)". The section "Download the data" follows, with text: "You can download the data from figshare (<http://dx.doi.org/10.6084/m9.figshare.1284521>). If you want to use the command line, you can use something like:" and a code block containing shell commands: 

```
# create a project directory
mkdir uce-tutorial

# change to that directory
cd uce-tutorial

# download the data into a file names fastq.zip
wget -O fastq.zip https://ndownloader.figshare.com/articles/1284521/versions/1

# make a directory to hold the data
mkdir raw-fastq

# move the zip file into that directory
mv fastq.zip raw-fastq

# move into the directory we just created
cd raw-fastq

# unzip the fastq data
unzip fastq.zip

# delete the zip file
rm fastq.zip
```

WRITE THE DOCS

Love Documentation? Come to the Write the Docs 2016 conference in Portland.