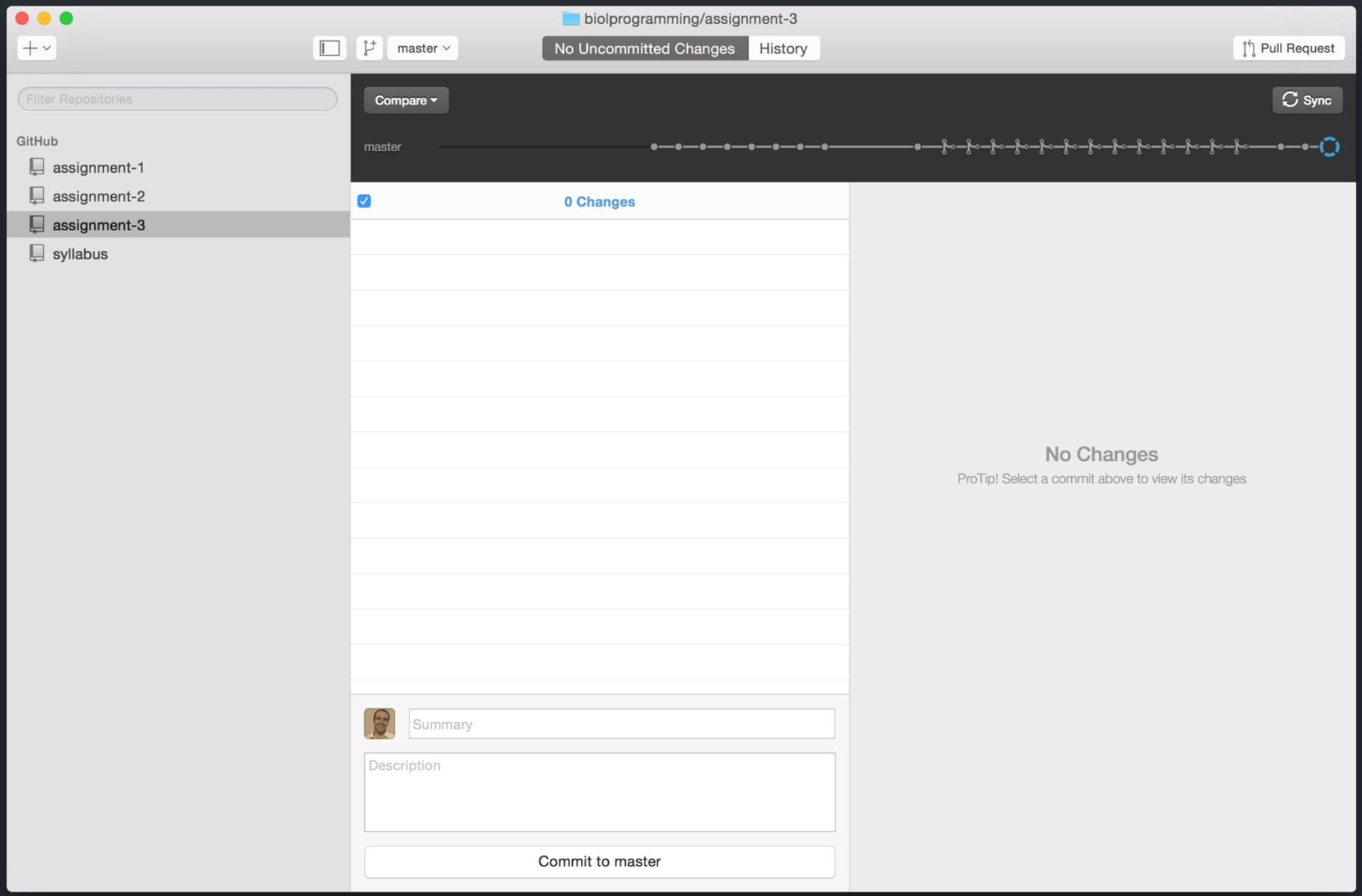
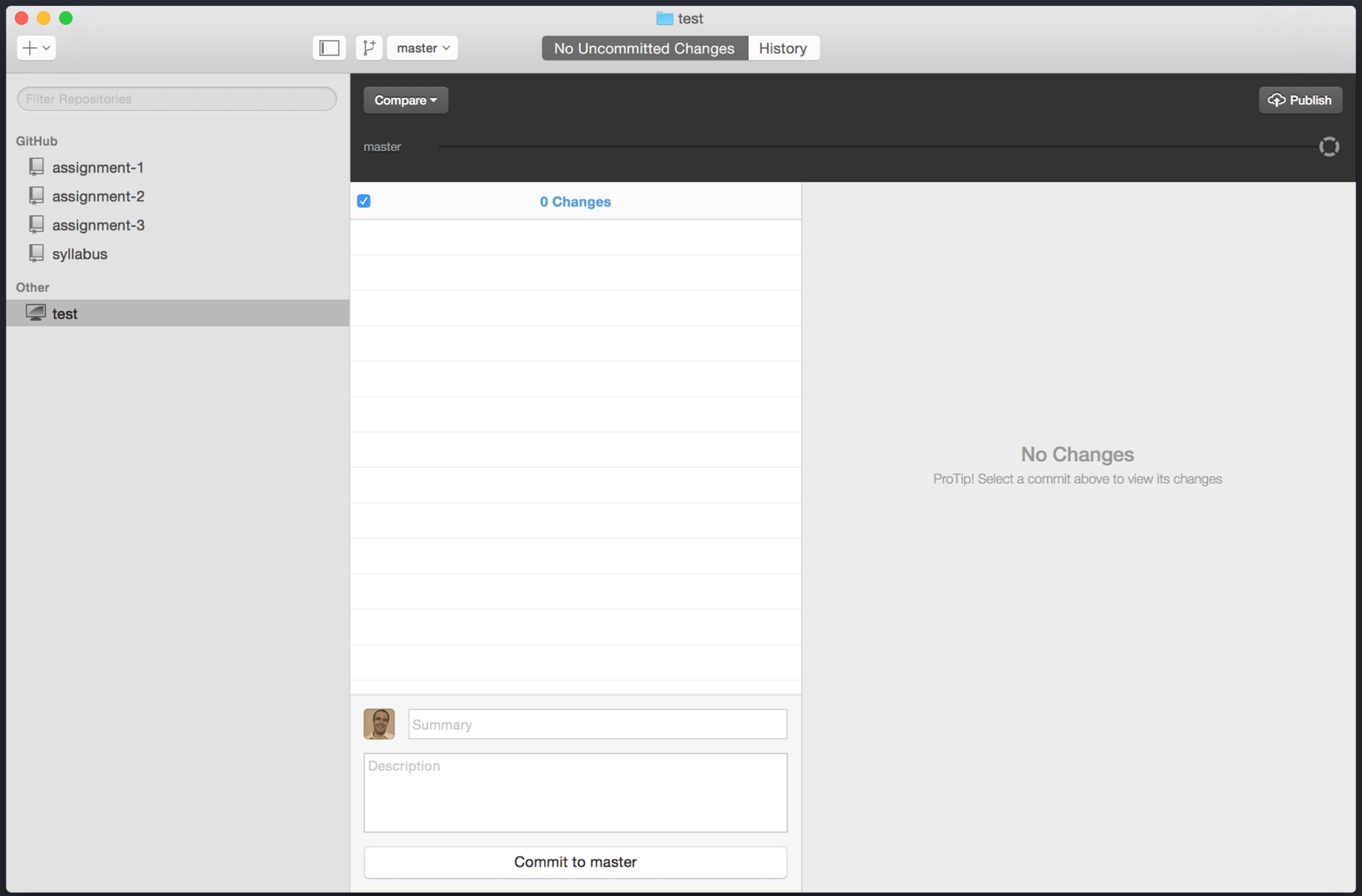
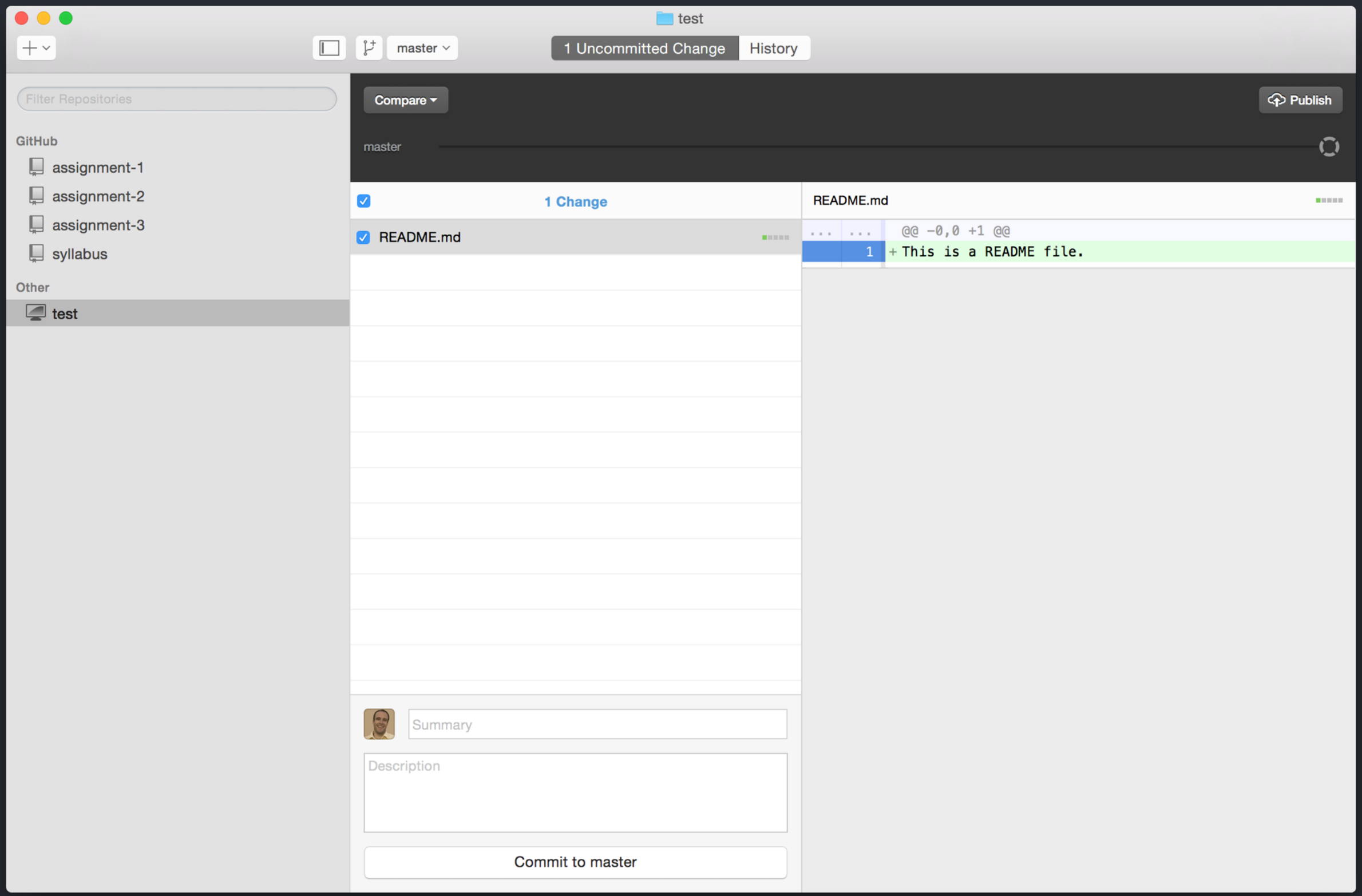
First… a review



New repository…



Add a file… the commit



This is what it looks like…

```
bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md
```

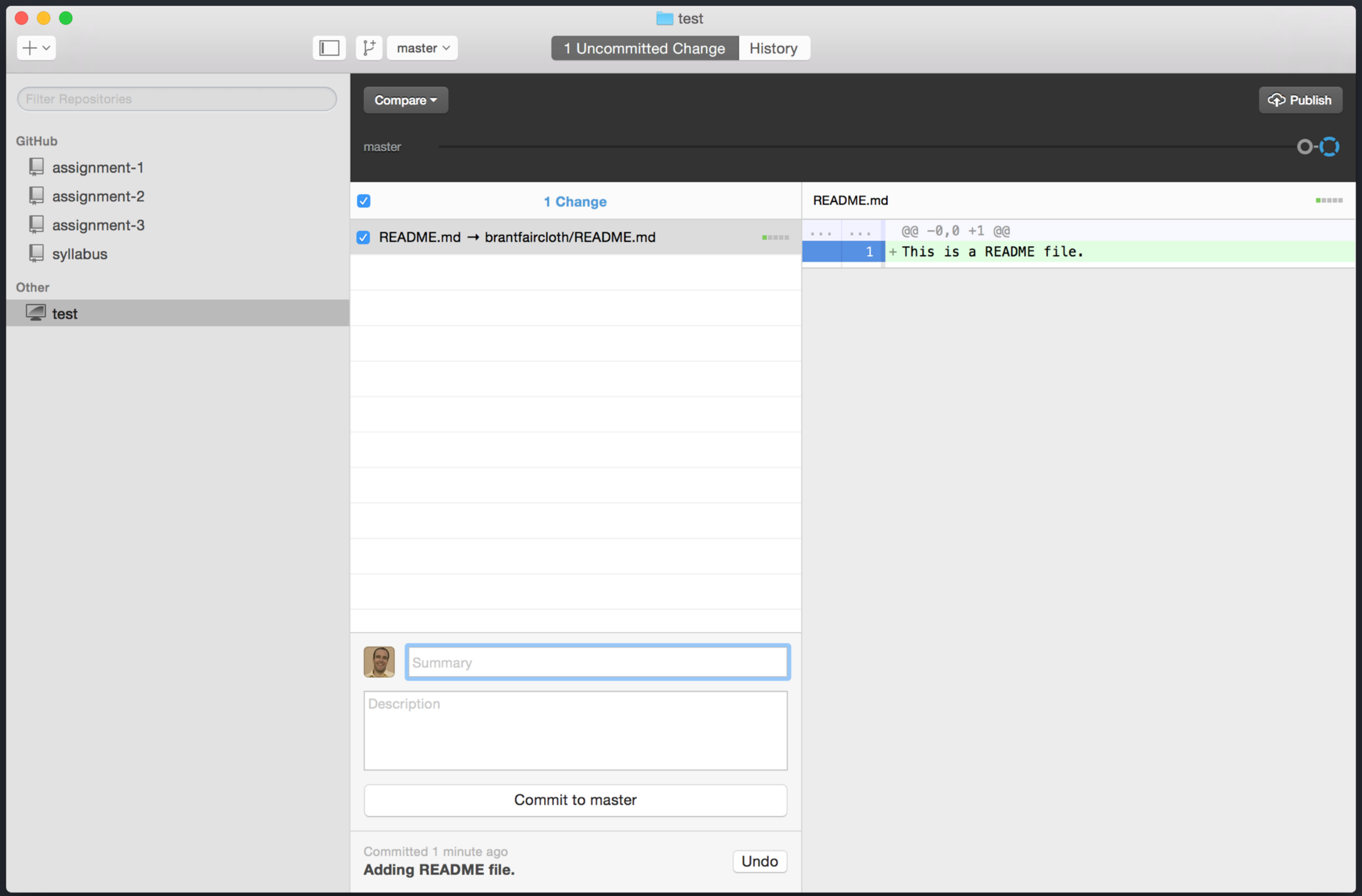
Make a new directory…

```
bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ mkdir brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md       brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master)
$
```

Now, if I move the file…

```
bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ mkdir brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md       brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md       brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ mv README.md brantfaircloth
```

This is what I see...



And this is git in terminal...

```
bcf at brant-4 in ~/github-app/biol7800/test (master)
$ mv README.md brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master●●)
$ git st
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        brantfaircloth/

no changes added to commit (use "git add" and/or "git commit -a")
```

And this is what the repo looks like…

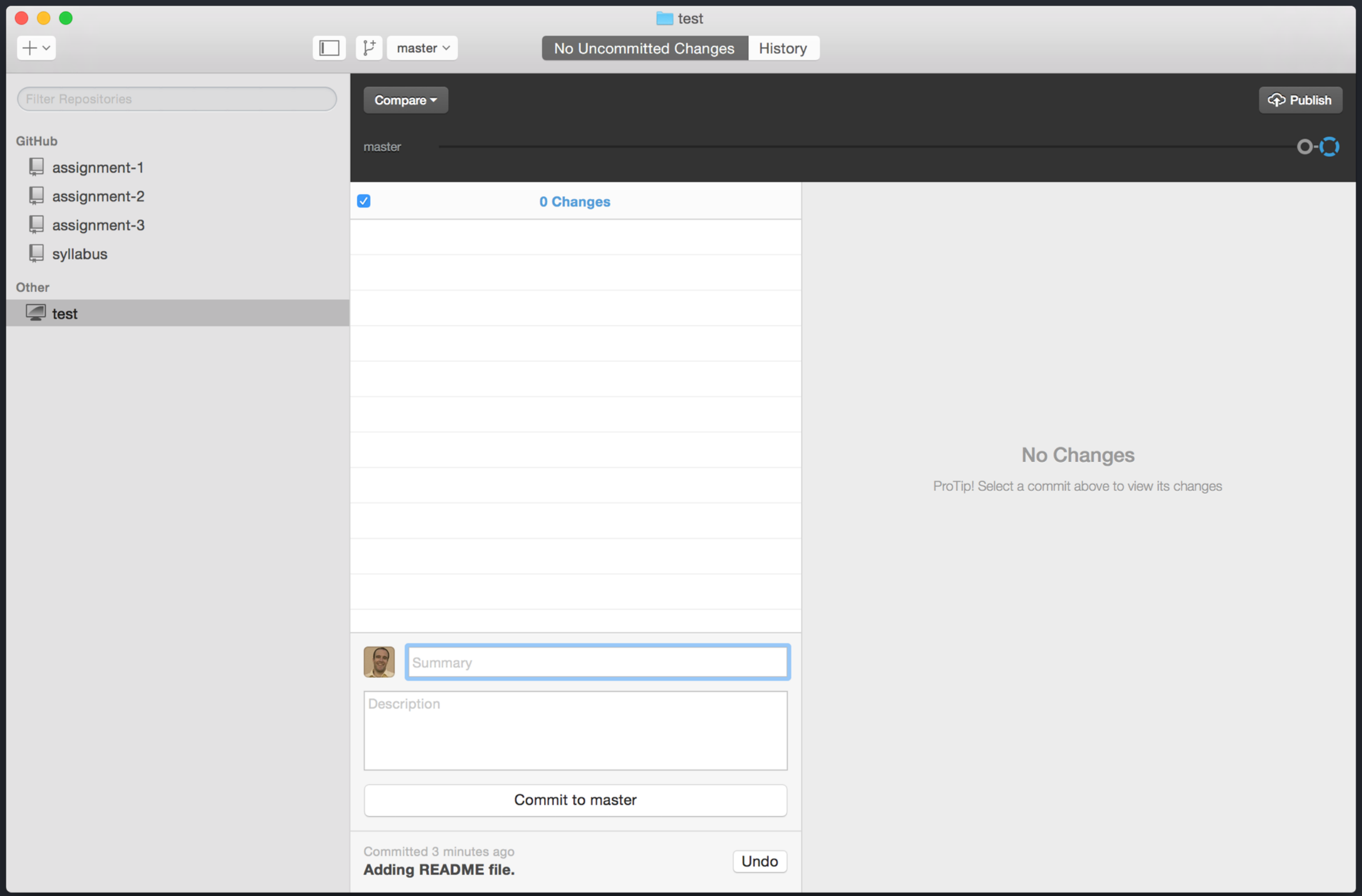
```
bcf at brant-4 in ~/github-app/biol7800/test (master●●)
$ ls
brantfaircloth
```

But that's not what I want...

So, I move the file back

```
bcf at brant-4 in ~/github-app/biol7800/test (master●●)
$ mv brantfaircloth/README.md ./

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md       brantfaircloth
```

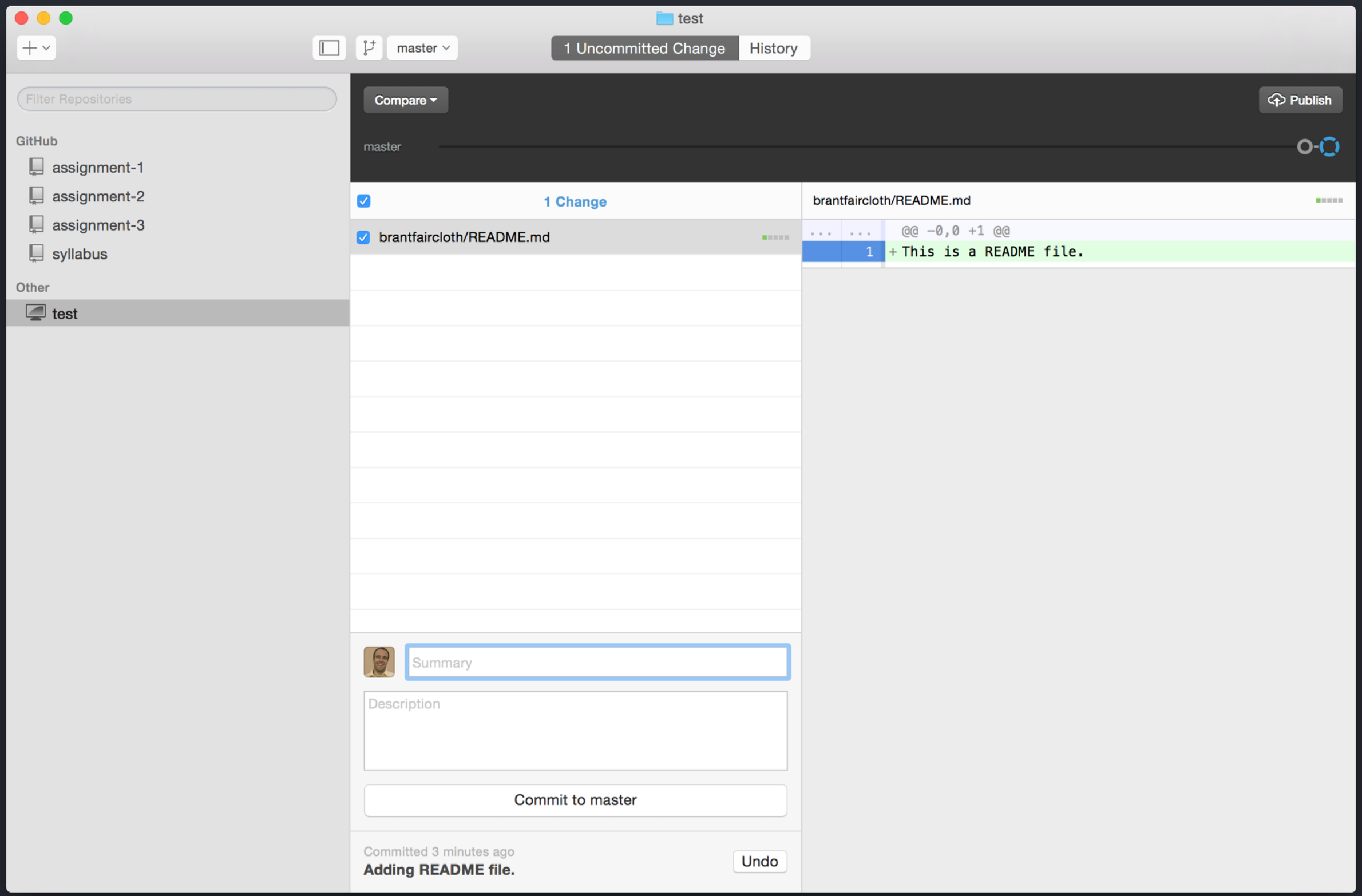
This is what I see…



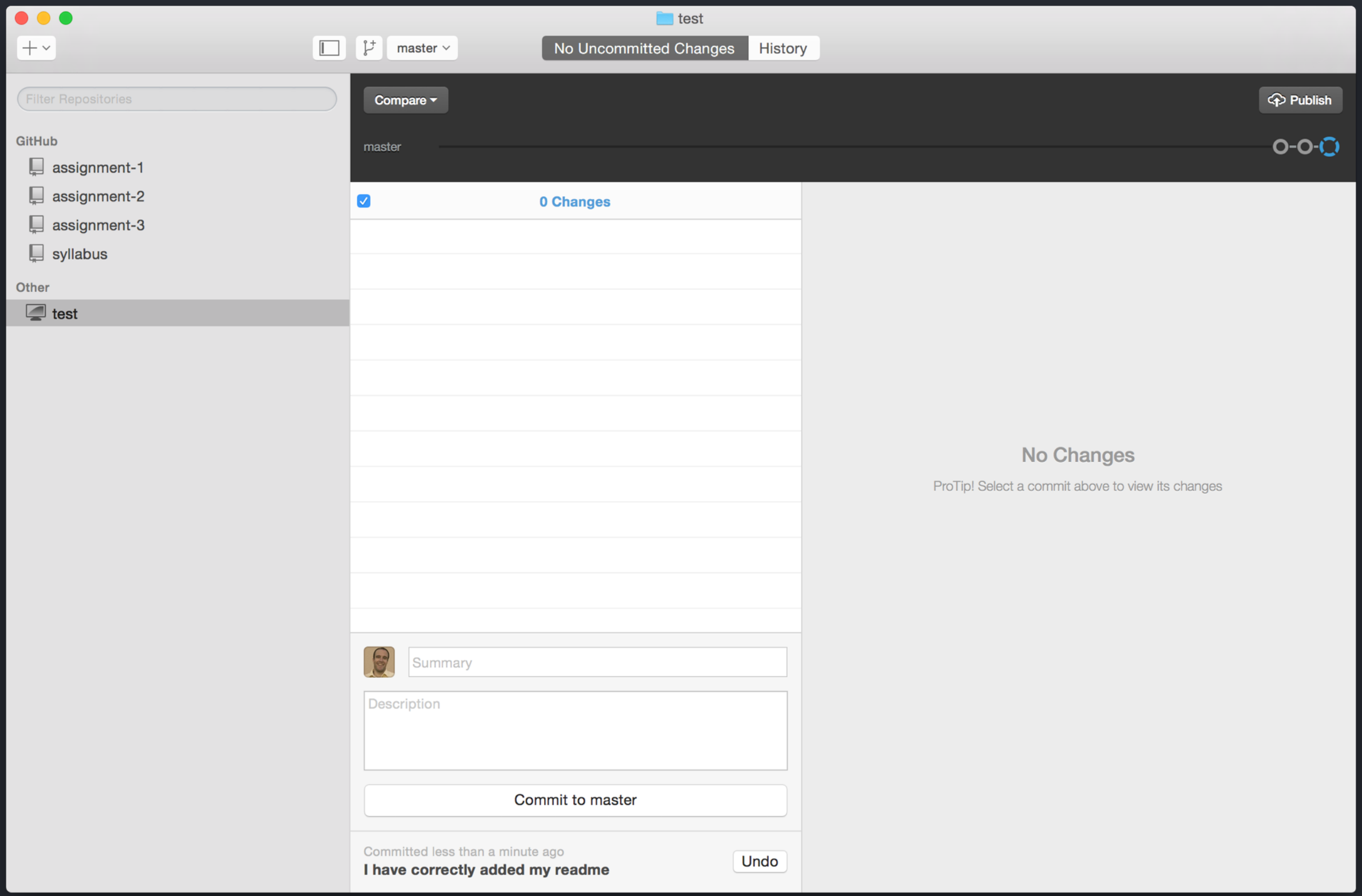
Now, if I copy the file…

```
bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls
README.md       brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ cp README.md brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master●)
$ ls
README.md       brantfaircloth

bcf at brant-4 in ~/github-app/biol7800/test (master)
$ ls brantfaircloth
README.md
```

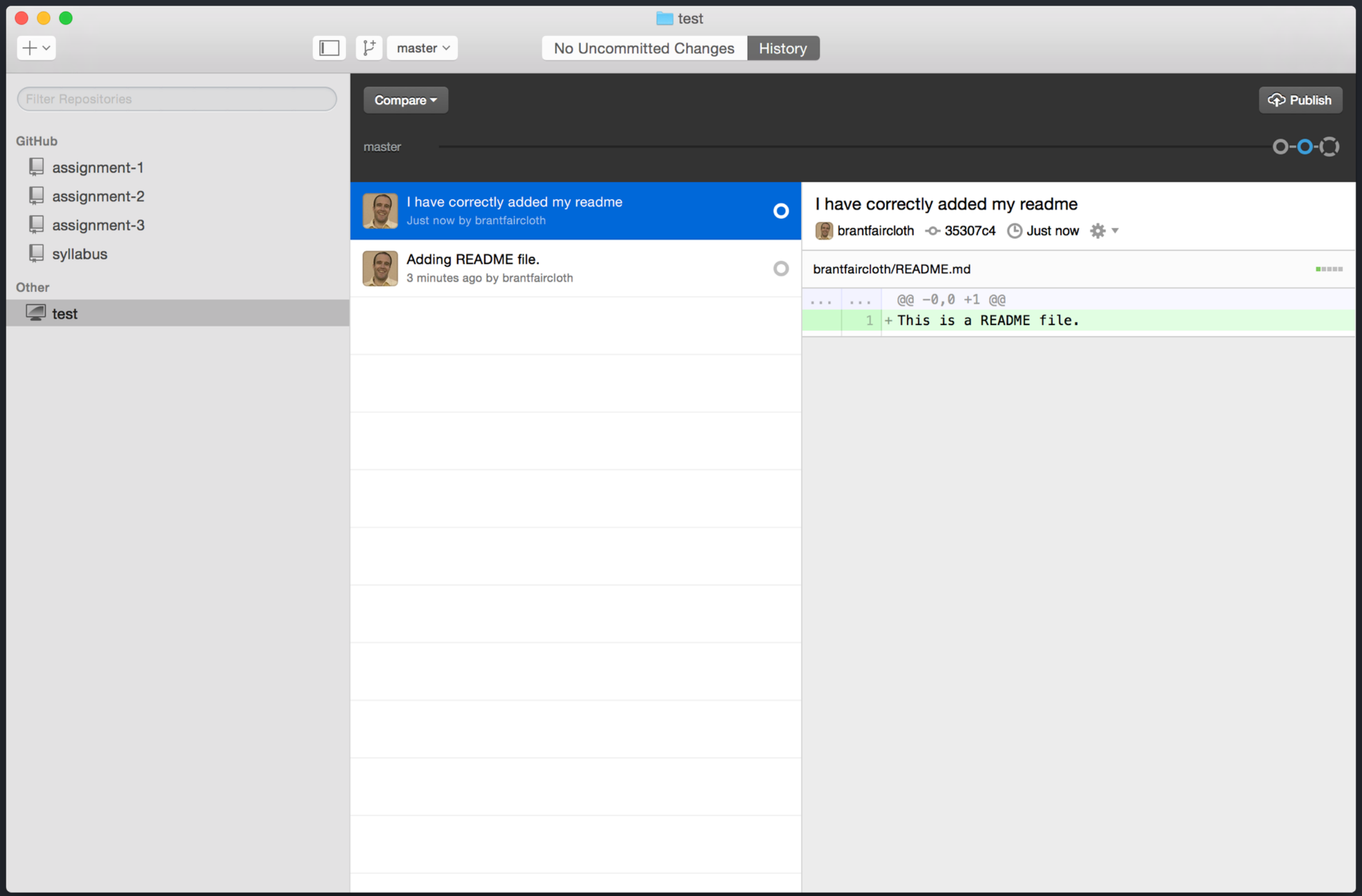
This is what I see…



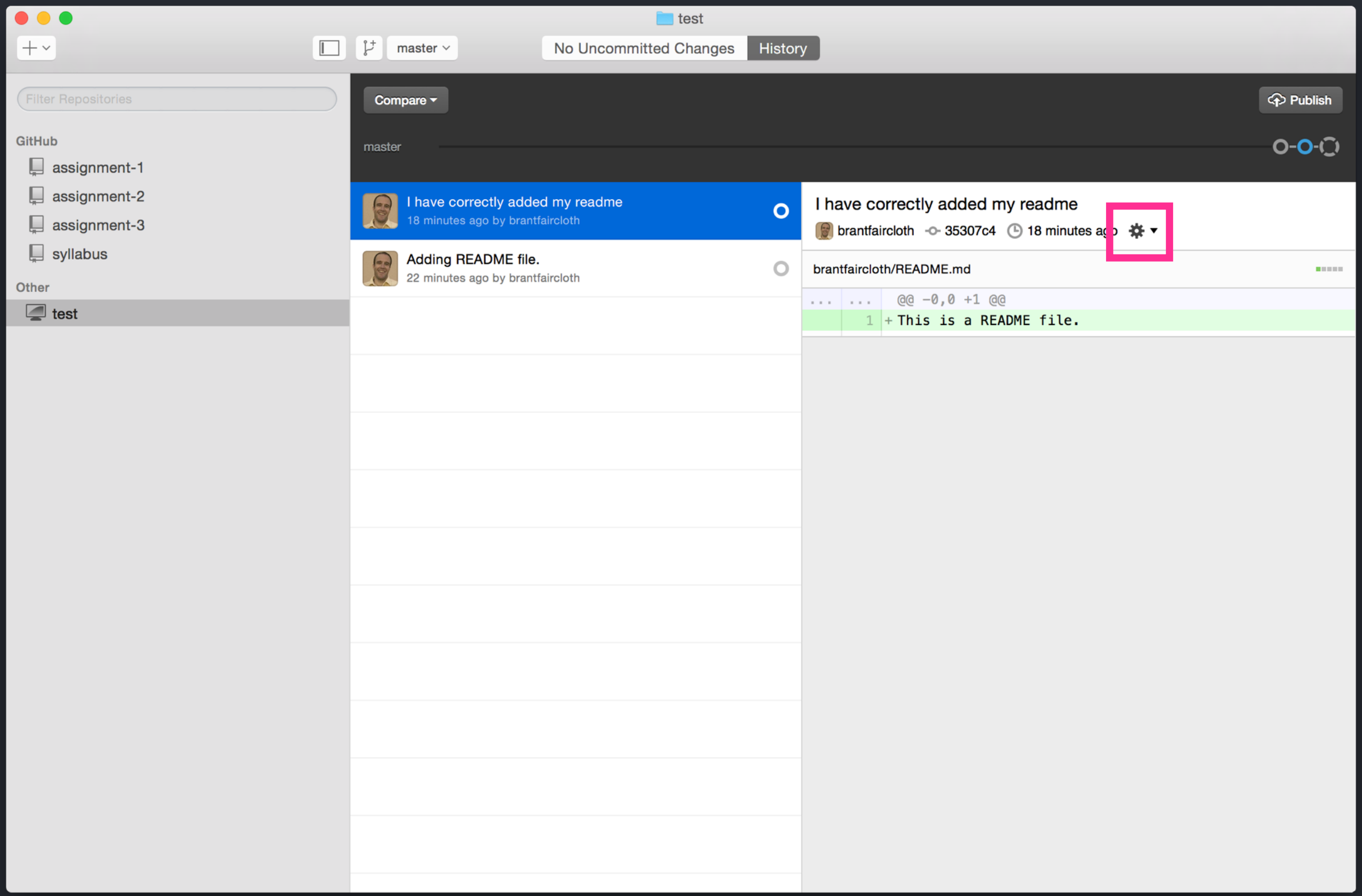
After the commit…

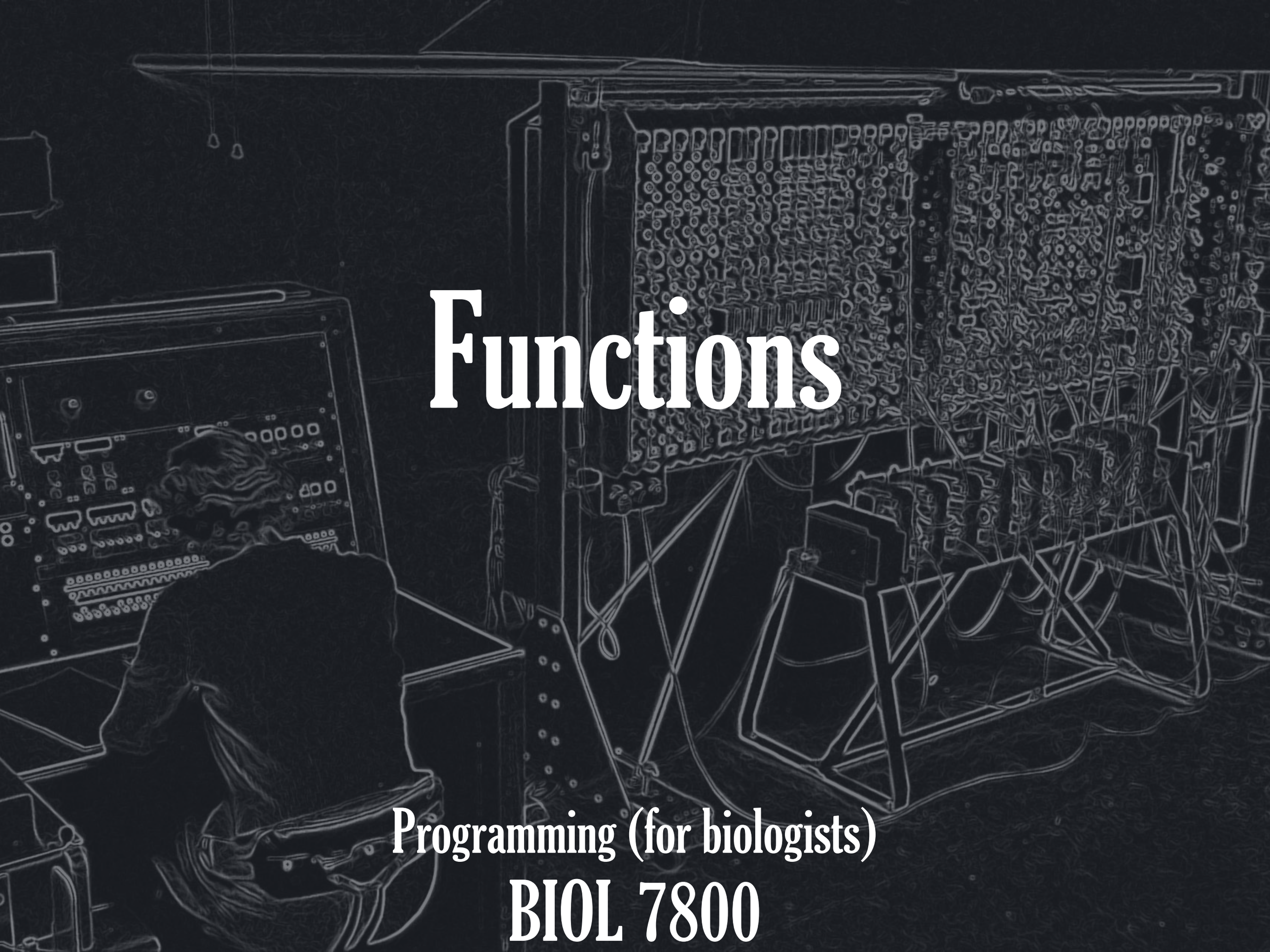


I can check the history…



If I want to revert the commit…





Functions

Programming (for biologists)

BIOL 7800

Functions

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

example.py

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

"""
example.py

Created by Brant Faircloth on 27 Jan 2016.
Copyright 2016 Brant C. Faircloth. All rights reserved.
"""


def function1():
    print("BIOL 7800")


def main():
    function1()

if __name__ == '__main__':
    main()

```

File 0 | Project 0 | No Issues | example.py* | 7:42 | LF | UTF-8 | Python

Functions

What are they?

A **function** is an organized group of statements with a name

You specify the statements that run when you **“call”** the **function** name.

Functions

What are they?

`print()` is a function

`type()` is a function

`int()` is a function

`float()` is a function

etc., etc., etc.

Functions

What are they?

For example, this is the **source code** for

random.randrange()

lib/python3.5/random.py

example.py • random.py

```
170     def randrange(self, start, stop=None, step=1, _int=int):
171         """Choose a random item from range(start, stop[, step]).
172
173         This fixes the problem with randint() which includes the
174         endpoint; in Python this is usually not what you want.
175
176         """
177
178         # This code is a bit messy to make it fast for the
179         # common case while still doing adequate error checking.
180         istart = _int(start)
181         if istart != start:
182             raise ValueError("non-integer arg 1 for randrange()")
183         if stop is None:
184             if istart > 0:
185                 return self._randbelow(istart)
186             raise ValueError("empty range for randrange()")
187
188         # stop argument supplied.
189         istop = _int(stop)
190         if istop != stop:
191             raise ValueError("non-integer stop for randrange()")
192         width = istop - istart
193         if step == 1 and width > 0:
194             return istart + self._randbelow(width)
195         if step == 1:
196             raise ValueError("empty range for randrange() (%d,%d, %d)" %
197
198         # Non-unit step argument supplied.
199         istep = _int(step)
200         if istep != step:
201             raise ValueError("non-integer step for randrange()")
202         if istep > 0:
203             n = (width + istep - 1) // istep
204         elif istep < 0:
205             n = (width + istep + 1) // istep
206         else:
207             raise ValueError("zero step for randrange()")
```

Functions

Why?

1. Allow you to **organize and group** statements

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function_to_reverse_complement(sequence_data):
6       # do some stuff
7       pass
8
9
10  def function_to_complement(sequence_data):
11      # do some stuff
12      pass
13
14
15  def function_to_reverse(sequence_data):
16      # do some stuff
17      pass
18
19
20  def main():
21      data = 'ACGTGTCGTCACAC'
22      function_to_reverse_complement(data)
23      function_to_complement(data)
24      function_to_reverse(data)
25
26  if __name__ == '__main__':
27      main()
28
```

File 0 | Project 0 | ✔ No Issues | example.py* | 25:1 | LF | UTF-8 | Python

Functions

Why?

2. Allow you to **reuse** statements

(and only write them once)

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function_to_reverse_complement(sequence_data):
6       # do some stuff
7       pass
8
9
10  def function_to_complement(sequence_data):
11      # do some stuff
12      pass
13
14
15  def function_to_reverse(sequence_data):
16      # do some stuff
17      pass
18
19
20  def main():
21      data = 'ACGTGTCGTCACAC'
22      function_to_complement(data)
23      function_to_reverse(data)
24      function_to_complement(data)
25
26  if __name__ == '__main__':
27      main()
28
```

File 0 | Project 0 | No Issues | example.py* | 19:1 | LF | UTF-8 | Python

Functions

Why?

3. Allow you to **divide up** programs (smaller pieces easier to follow AND debug)

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function_to_reverse_complement(sequence_data):
6       # do some stuff
7       pass
8
9
10  def function_to_complement(sequence_data):
11      # do some stuff
12      pass
13
14
15  def function_to_reverse(sequence_data):
16      # do some stuff
17      pass
18
19
20  def main():
21      data = 'ACGTGTCGTCACAC'
22      function_to_complement(data)
23      function_to_reverse(data)
24      function_to_complement(data)
25
26  if __name__ == '__main__':
27      main()
28
```

File 0 | Project 0 | No Issues | example.py* | 19:1 | LF | UTF-8 | Python

Functions

Why?

4. Functions are **re-usable**

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

example.py

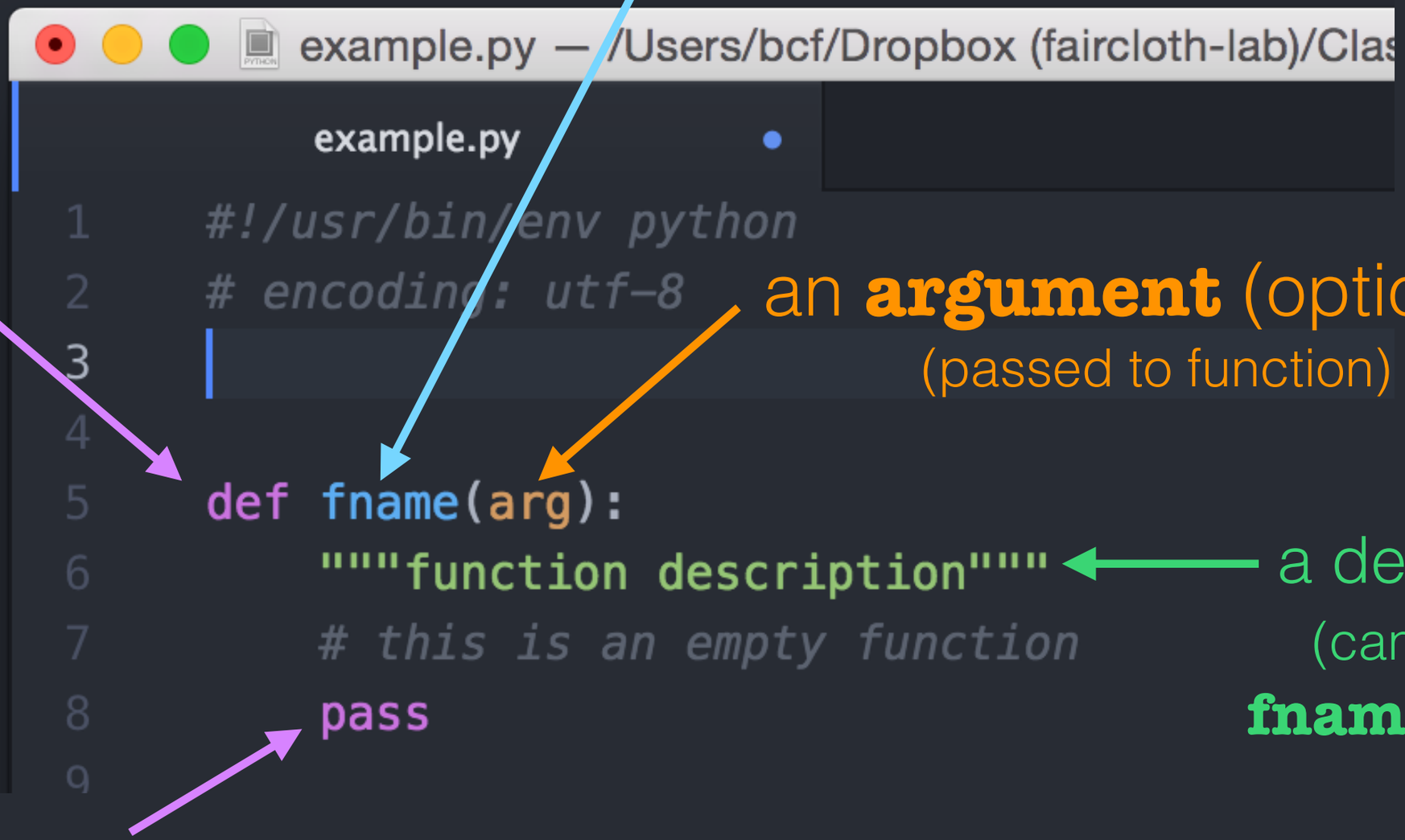
```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function_to_reverse_complement(sequence_data):
6       # do some stuff
7       pass
8
9
10  def function_to_complement(sequence_data):
11      # do some stuff
12      pass
13
14
15  def function_to_reverse(sequence_data):
16      # do some stuff
17      pass
```

We can **import example.function_to_reverse**
and use it in another programs

One reason to keep them **small** and **atomic**.

Anatomy of a Function

the function **name**



def

an **argument** (optional)
(passed to function)

a description
(can access)
fname.__doc__

basically means skip
fxn for now.
(this is where your code will go)

Anatomy of Some Functions

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg):
6       """function description"""
7       # this is an empty function
8       pass
9
10
11  def example1():
12      """reverse a string"""
13      s = 'this is my string'
14      print("One way:")
15      print(''.join(reversed(s)))
16
17
18  def example2():
19      """reverse a string another way"""
20      s = 'this is my string'
21      print("Another way:")
22      print(s[::-1])
23
24  example1()
25  example2()
26
```

What didn't I do?

File 0 Project 0 No Issues example.py* 3:1 LF UTF-8 Python

Anatomy of a Function

Here, we are **passing four** values into the function using **arguments**

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1, arg2, arg3, arg4):
6       """function description"""
7       # this is an empty function
8       print(arg1)
9       print(arg2)
10      print(arg3)
11
12
13  fname("cats", "better", "than", "dogs")
14
```

arguments (optional)
(passed to function)

File 0 | Project 0 | ✔ No Issues | example.py* | 7:29 | LF | UTF-8 | Python

Anatomy of a Function

Arguments are optional

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

example.py

arguments (empty)

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname():
6       """function description"""
7       # this is an empty function
8       print("rabbits")
9       print("are")
10      print("better")
11
12
13  fname()
14
```

File 0 | Project 0 | ✔ No Issues | example.py* | 13:8 | LF | UTF-8 | Python

Anatomy of a Function

Arguments can also have **default** values

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1="rabbits"):
6       """function description"""
7       # this is an empty function
8       print(arg1)
9       print("are")
10      print("better")
11
12
13  fname()
14
```

File 0 | Project 0 | ✔ No Issues | example.py* | 13:8 | LF | UTF-8 | Python

Anatomy of a Function

Arguments can also have **default** values

that can be overridden

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1="rabbits"):
6       """function description"""
7       # this is an empty function
8       print(arg1)
9       print("are")
10      print("better")
11
12
13  fname("scorpions")
14
```

We're overriding the default

File 0 | Project 0 | ✔ No Issues | example.py* | 1:1 | LF | UTF-8 | Python

Anatomy of a Function

Order **matters**

Arguments with defaults **come last**.

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1="rabbits", arg2):
6       """function description"""
7       # this is an empty function
8       print(arg1)
9       print("are")
10      print("better")
11
12
13    fname("scorpions")
```

File 1 | Project 1 | 1 Issue example.py 5:31 | LF | UTF-8 | Python

Anatomy of a Function

Order **matters**

Arguments with defaults **come last**.

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1="rabbits", arg2):
6       """function description"""
7       # this is an empty function
8       print(arg1)
9       print("are")
10      print("better")
11
12
13  fname("scorpions")
```

File 1 | Project 1 | ✖ 1 Issue example.py

```
bcf at brant-4 in ~/Dropbox/Classes/BIOL7800/temp
$ python example.py
  File "example.py", line 5
    def fname(arg1="rabbits", arg2):
                             ^
SyntaxError: non-default argument follows default argument
```

Anatomy of a Function

Function **variables** are **local**

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

example.py

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

def fname(arg1="rabbits"):
    """function description"""
    # this is an empty function
    print(arg1)
    print("are")
    print("better")

fname("scorpions")
print arg1
```

Won't work

File 1 | Project 1 | 1 Issue example.py* 11:1 LF UTF-8 Python

Anatomy of a Function

Function **variables** are **local**

(to the **function**)

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1="rabbits"):
6       """function description"""
7       # this is an empty function
8       print(arg1)
9       print("are")
10      print("better")
11
12
13  fname("scorpions")
14  print arg1
15
```

File 1 Project 1 ✖ 1 Issue example.py* 11:1

```
bcf at brant-4 in ~/Dropbox/Classes/BIOL7800/temp
$ python example.py
scorpions
are
better
Traceback (most recent call last):
  File "example.py", line 14, in <module>
    print(arg1)
NameError: name 'arg1' is not defined
```

Anatomy of a Function

Functions can **return** values (but don't **have** to)

example.py — /Users/bcf/Dropbox (fair

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1):
6       """function description"""
7       # this is an empty function
8       print(arg1 + arg1)
9
10
11  fname(2)
12
```

File 0 Project 0 ✔ No Issues example.py* 8:23

"void" functions

(do not return a value)

example.py — /Users/bcf/Dropbox (fair

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1):
6       """function description"""
7       # this is an empty function
8       result = arg1 + arg1
9       return result
10
11
12  from_fname_function = fname(2)
13
```

File 0 Project 0 ✔ No Issues example.py* 6:31

"fruitful" functions

(return a value)

Anatomy of a Function

So, **why** does this work?

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4   arg1 = "hot dogs"
5
6   def fname():
7       """function description"""
8       # this is an empty function
9       print(arg1)
10      print("are")
11      print("better")
12
13
14  fname()
15  print(arg1)
16
```

File 1 | Project 1 | 1 Issue example.py 8:32

1. zsh

```
(py35)
bcf at brant-4 in ~/Dropbox/Classes/BIOL7800/temp
$ python example.py
hot dogs
are
better
hot dogs
(py35)
bcf at brant-4 in ~/Dropbox/Classes/BIOL7800/temp
$
```

Anatomy of a Function

So, **why** does this work?

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4   arg1 = "hot dogs"
5
6   def fname():
7       """function description"""
8       # this is an empty function
9       print(arg1)
10      print("are")
11      print("better")
12
13
14  fname()
15  print(arg1)
16
```

in “main” program

File 1 | Project 1 | 1 Issue example.py 8:32 | LF | UTF-8 | Python

Anatomy of a Function

So, **why** does this work?

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4   arg1 = "hot dogs"
5
6   def fname():
7       """function description"""
8       # this is an empty function
9       print(arg1)
10      print("are")
11      print("better")
12
13
14  fname()
15  print(arg1)
16
```

AKA "**global**" variable

File 1 | Project 1 | 1 Issue example.py 8:32 | LF | UTF-8 | Python

"global" variables

You generally should **AVOID** them

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

example.py

```
2   # encoding: utf-8
3
4   ARG1 = "hot dogs"
5
6   def fname():
7       """function description"""
8       # this is an empty function
9       print(ARG1)
10      print("are")
11      print("better")
12
13
14  fname()
15  print(ARG1)
16
```

But, if you use them, the convention is to **CAP** their names

That way, you ***know*** they are global

File 1 | Project 1 | 1 Issue example.py* | 16:1 | LF | UTF-8 | Python

"global" and "local" refer to Variable Scope

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

example.py

```
2   # encoding: utf-8
3
4   ARG1 = "hot dogs"
5
6   def fname():
7       """function description"""
8       # this is an empty function
9       print(ARG1)
10      print("are")
11      print("better")
12
13
14  fname()
15  print(ARG1)
16
```

But, if you use them, the convention is to **CAP** their names

That way, you ***know*** they are global

File 1 | Project 1 | 1 Issue example.py* | 16:1 | LF | UTF-8 | Python

Variable Scope

main() function helps ensure variables are **“encapsulated”**

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1):
6       """function description"""
7       # this is an empty function
8       print(arg1)
9       print("are")
10      print("better")
11
12
13  def main():
14      var1 = "hot dogs"
15      fname(var1)
16
17  if __name__ == '__main__':
18      main()
19
```

All variables here are encapsulated, thus “local”

They can’t easily **“leak”** to other, unintended functions

File 0 | Project 0 | ✔ No Issues | example.py* | 18:8 | (1, 6) | LF | UTF-8 | Python

Flow of Execution

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg1):
6       """function description"""
7       # this is an empty function
8       result = arg1 + arg1
9       return result
10
11
12  def main():
13      var1 = 2
14      from_fname_function = fname(var1)
15      print(from_fname_function)
16
17  if __name__ == '__main__':
18      main()
```

What is the flow of execution here?

File 0 | Project 0 | ✔ No Issues | example.py* | 1:1 | LF | UTF-8 | Python

Flow of Execution

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

example.py

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Tue Jan 26 12:41:32 2016
4   @author: Marco
5   """
6
7
8   def main():
9       # I nested one function inside the other
10      def get_numbers():
11          # creating a list with the numbers in the desired rang
12          print([elem for elem in range(0, 51, 5)])
13      return get_numbers()
14
15
16  if __name__ == '__main__':
17      main()
18
```

What is the flow of execution here?

File 0 | Project 0 | ✔ No Issues | example.py* | 18:1 | LF | UTF-8 | Python

Flow of Execution

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

example.py

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Tue Jan 26 12:41:32 2016
4   """
5
6
7   def main():
8       # I nested one function inside the other
9       def get_numbers():
10          # creating a list with the numbers in the desired rang
11          print([elem for elem in range(0, 51, 5)])
12      return get_numbers()
13
14
15  if __name__ == '__main__':
16      main()
17
```

File 0 | Project 0 | ✔ No Issues | example.py* | 3:36 | LF | UTF-8 | Python

What else might be slightly wrong?

Flow of Execution

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

example.py

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Tue Jan 26 12:41:32 2016
4   """
5
6
7   def main():
8       get_numbers()
9       # I nested one function inside the other
10      def get_numbers():
11          # creating a list with the numbers in the desired rang
12          print([elem for elem in range(0, 51, 5)])
13      return get_numbers()
14
15
16  if __name__ == '__main__':
17      main()
18
```

File 2 | Project 2 | 2 Issues | example.py* | 1:1 | LF | UTF-8 | Python

What else might be slightly wrong?