Modules and Testing

Presented by

Colin Pierce and Skylar Wyant

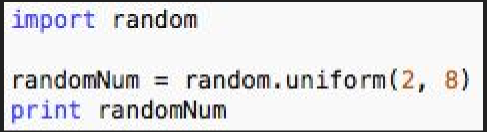
Modules Goals

- Review modules
- Learn how to use modules in a meaningful way to reuse code
- Be able to write and use our own modules
- Learn about namespaces and how they are useful
- Discuss docstrings and their applications within modules

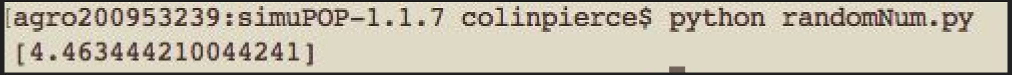
Modules Review & Introduction

- What is a module? random module
  - os, random, argparse
- How do we import a module?
  - import random
- How can we use modules in a way that allows us to re-use code?

Module example

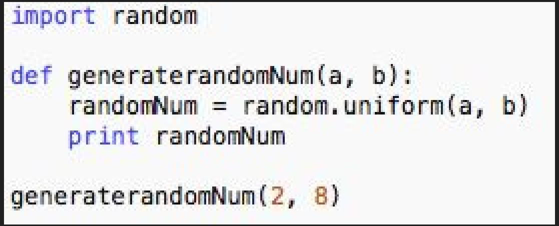


```
import random

randomNum = random.uniform(2, 8)
print randomNum
```

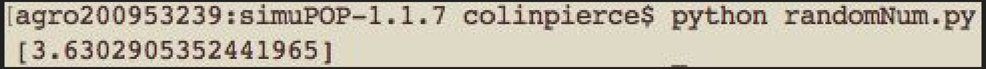


```
[agro200953239:simuPOP-1.1.7 colinpierce$ python randomNum.py
[4.463444210044241]
```

How could we re-use this piece of code, for example in another script?



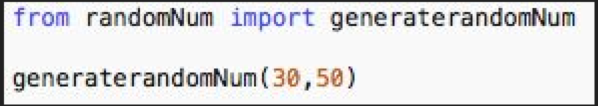
```
import random

def generaterandomNum(a, b):
    randomNum = random.uniform(a, b)
    print randomNum

generaterandomNum(2, 8)
```



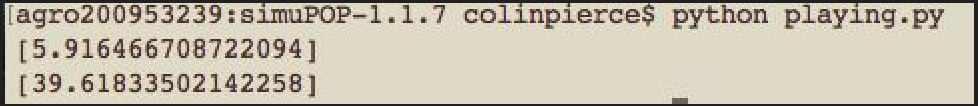
```
[agro200953239:simuPOP-1.1.7 colinpierce$ python randomNum.py
[3.6302905352441965]
```

randomNum.py

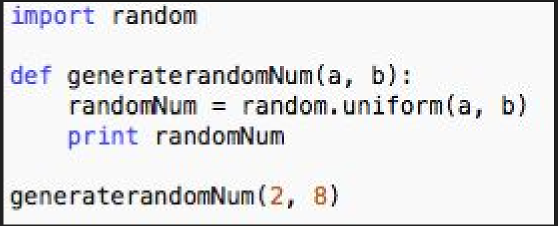
Module example contd...



```
from randomNum import generaterandomNum

generaterandomNum(30,50)
```



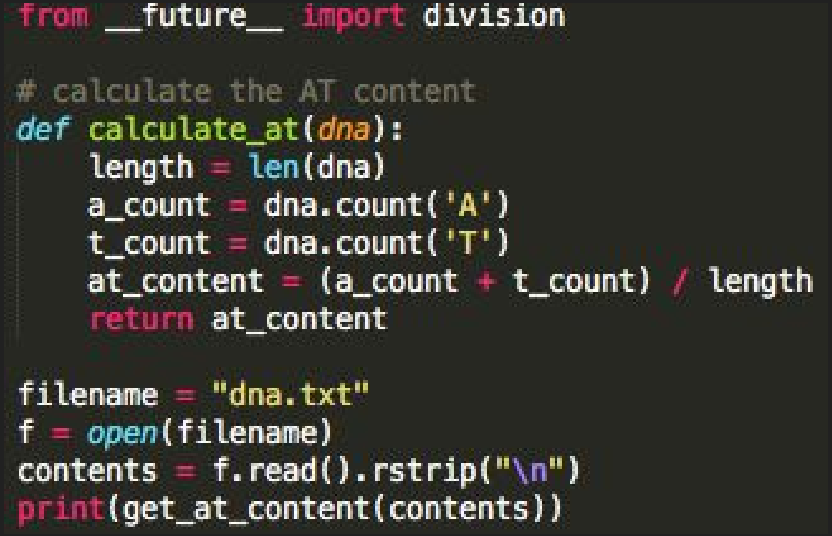
```
[agro200953239:simuPOP-1.1.7 colinpierce$ python playing.py
[5.916466708722094]
[39.61833502142258]
```



```
import random

def generaterandomNum(a, b):
    randomNum = random.uniform(a, b)
    print randomNum

generaterandomNum(2, 8)
```

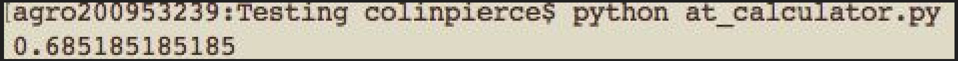
randomNum.py

Example: reading a DNA sequence, printing AT content



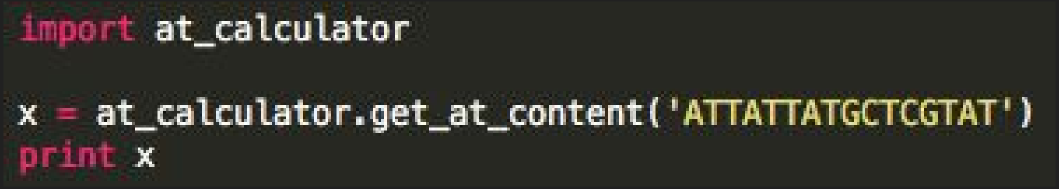
```python
from __future__ import division

# calculate the AT content
def calculate_at(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content

filename = "dna.txt"
f = open(filename)
contents = f.read().rstrip("\n")
print(get_at_content(contents))
```

at_calculator.py

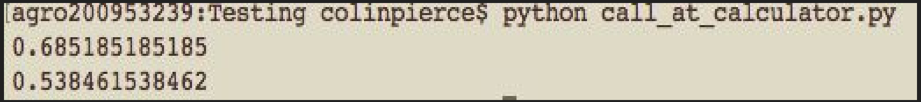


```
agro200953239:Testing colinpierce$ python at_calculator.py
0.685185185185
```

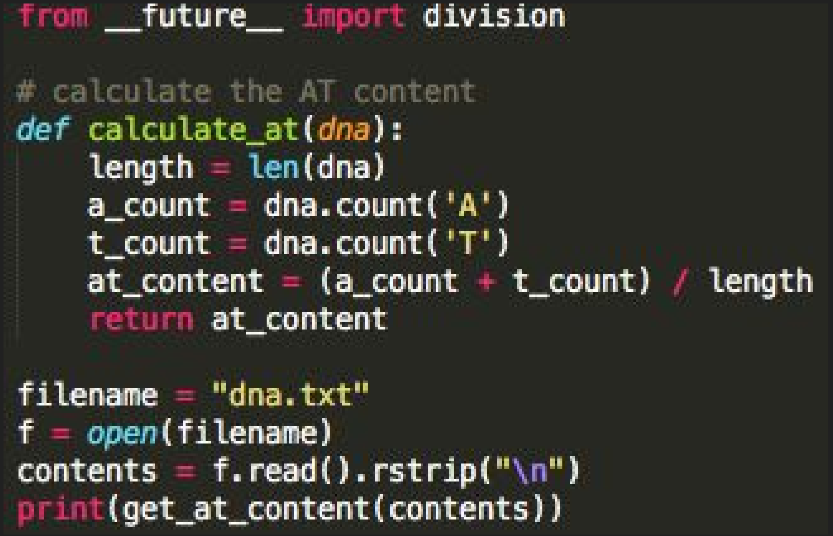
Call on the at_calculator script as a module



```python
import at_calculator

x = at_calculator.get_at_content('ATTATTATGCTCGTAT')
print x
```

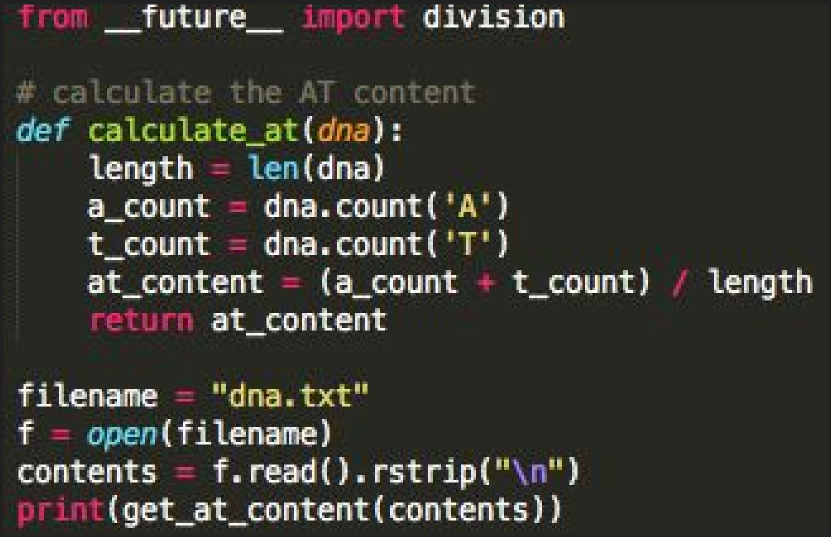


```
[agro200953239:Testing colinpierce$ python call_at_calculator.py
0.685185185185
0.538461538462
```



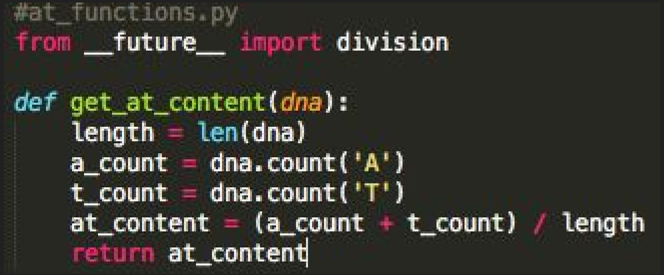
```python
from __future__ import division

# calculate the AT content
def calculate_at(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content

filename = "dna.txt"
f = open(filename)
contents = f.read().rstrip("\n")
print(get_at_content(contents))
```

Option 1: Split at_calculator.py

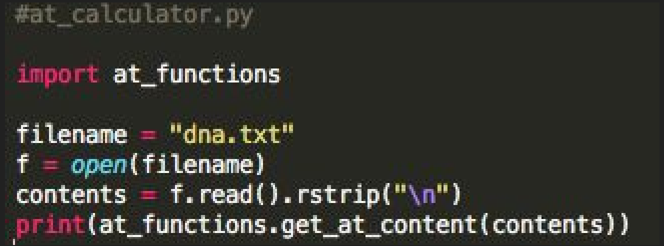


```python
from __future__ import division

# calculate the AT content
def calculate_at(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content

filename = "dna.txt"
f = open(filename)
contents = f.read().rstrip("\n")
print(get_at_content(contents))
```

Library!

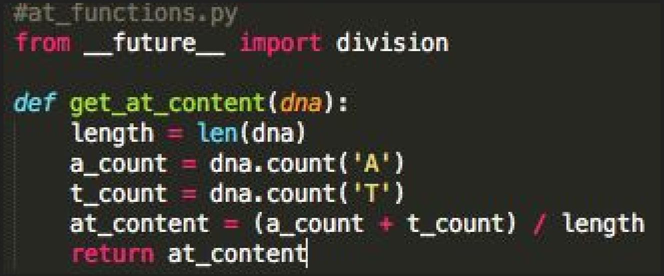


```python
#at_functions.py
from __future__ import division

def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```

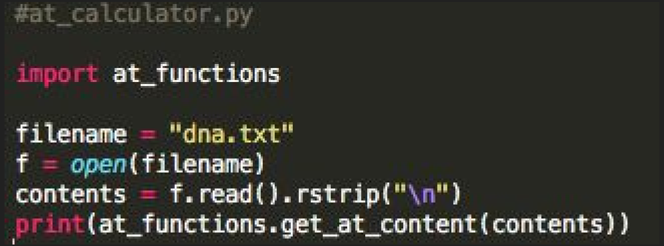


```python
#at_calculator.py

import at_functions

filename = "dna.txt"
f = open(filename)
contents = f.read().rstrip("\n")
print(at_functions.get_at_content(contents))
```

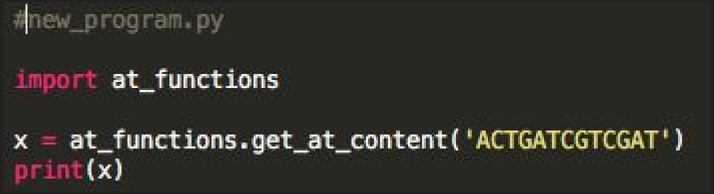
Option 1: Split at_calculator.py



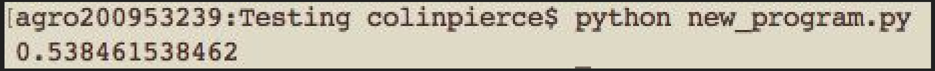
```python
#at_functions.py
from __future__ import division

def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```



```python
#at_calculator.py

import at_functions

filename = "dna.txt"
f = open(filename)
contents = f.read().rstrip("\n")
print(at_functions.get_at_content(contents))
```

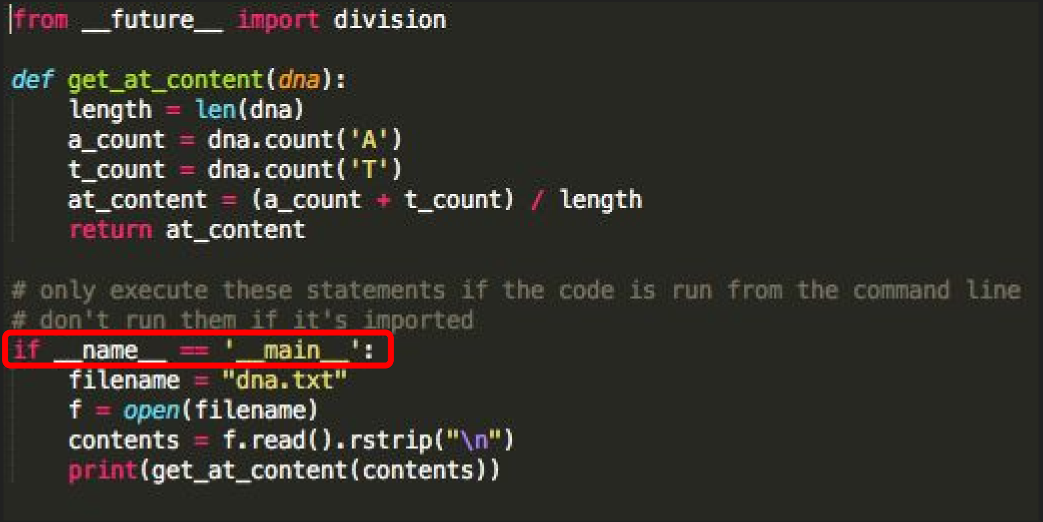


```python
#new_program.py

import at_functions

x = at_functions.get_at_content('ACTGATCGTCGAT')
print(x)
```



```
agro200953239:Testing colinpierce$ python new_program.py
0.538461538462
```

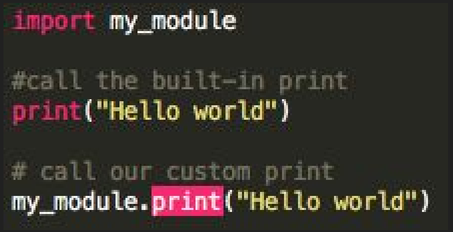
Option 2: Make top-level code conditional



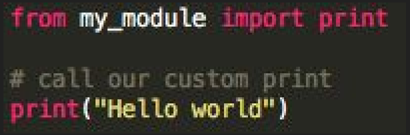
```python
from __future__ import division

def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content

# only execute these statements if the code is run from the command line
# don't run them if it's imported
if __name__ == '__main__':
    filename = "dna.txt"
    f = open(filename)
    contents = f.read().rstrip("\n")
    print(get_at_content(contents))
```

- When is this method useful?
  - Want code that can act as both a library AND a program
  - Want to include set of unit tests along with a library

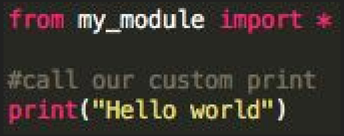
Names and namespaces



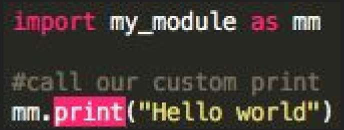
```
import my_module

#call the built-in print
print("Hello world")

# call our custom print
my_module.print("Hello world")
```



```
from my_module import print

# call our custom print
print("Hello world")
```



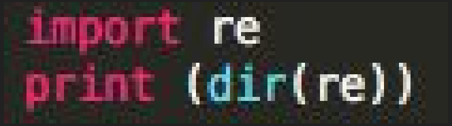
```
from my_module import *

#call our custom print
print("Hello world")
```



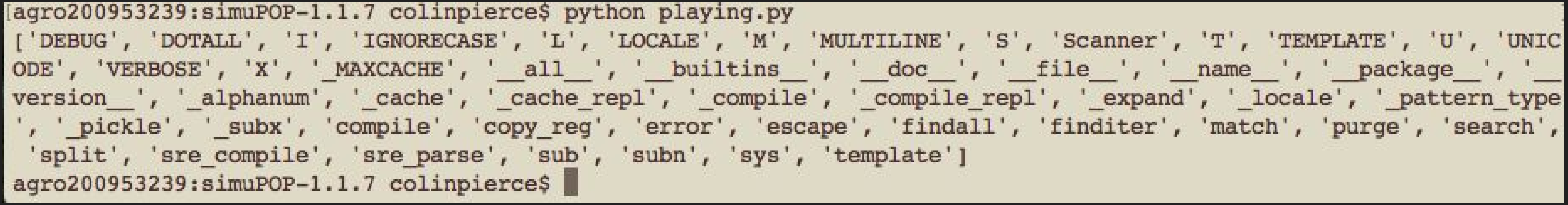
```
import my_module as mm

#call our custom print
mm.print("Hello world")
```

- my_module has its own “namespace”
- The print function defined in my-module belongs to the my_module namespace
- DANGER!
    - ‘print’ will now refer to function defined in my_module, NOT the built-in function
    - No way of knowing which functions might be overwritten
    - Poor readability (mm)

What’s in a module?

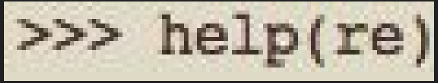


```python
import re
print (dir(re))
```

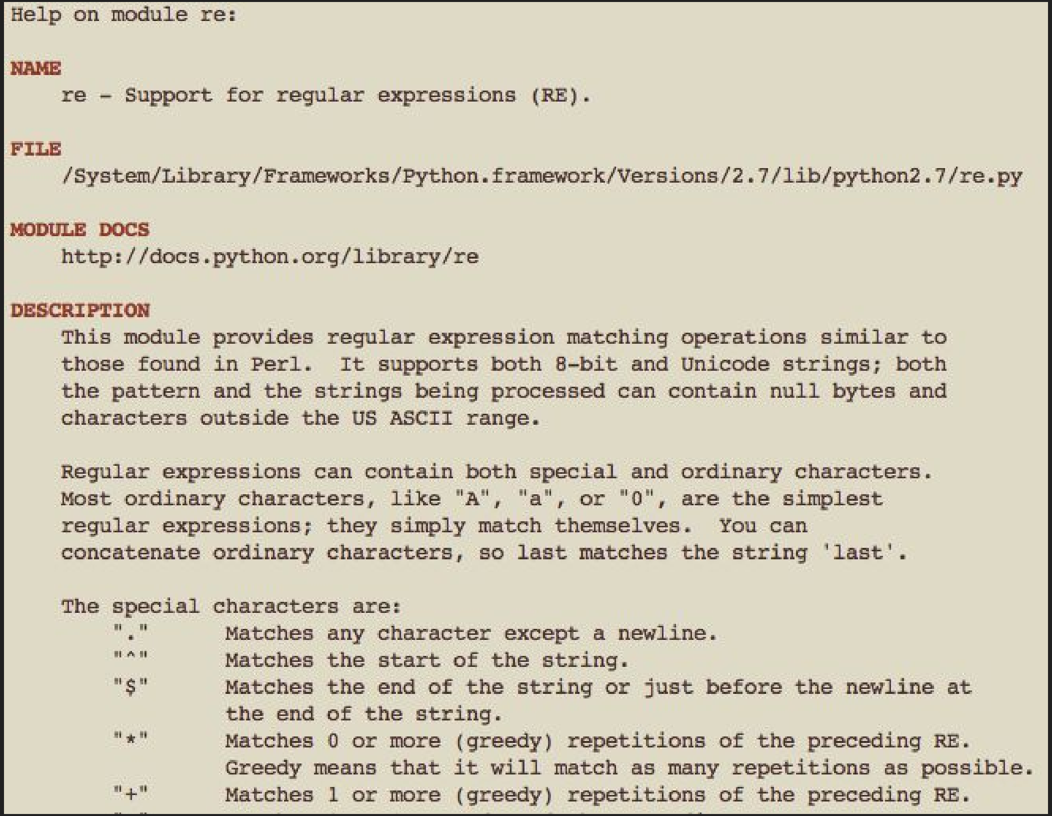


```
agro200953239:simuPOP-1.1.7 colinpierce$ python playing.py
['DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'S', 'Scanner', 'T', 'TEMPLATE', 'U', 'UNIC
ODE', 'VERBOSE', 'X', '_MAXCACHE', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '__
version__', '_alphanum', '_cache', '_cache_repl', '_compile', '_compile_repl', '_expand', '_locale', '_pattern_type
', '_pickle', '_subx', 'compile', 'copy_reg', 'error', 'escape', 'findall', 'finditer', 'match', 'purge', 'search',
 'split', 'sre_compile', 'sre_parse', 'sub', 'subn', 'sys', 'template']
agro200953239:simuPOP-1.1.7 colinpierce$ 
```

Accessing embedded documentation on a module

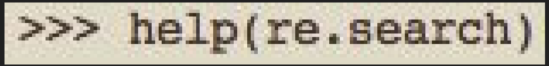


```
>>> help(re)
```

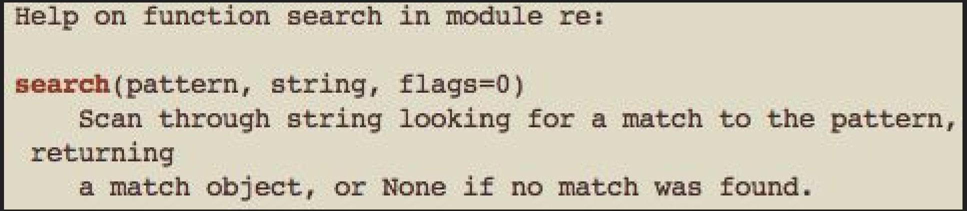


```
Help on module re:

NAME
    re - Support for regular expressions (RE).

FILE
    /System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/re.py

MODULE DOCS
    http://docs.python.org/library/re

DESCRIPTION
    This module provides regular expression matching operations similar to
    those found in Perl.  It supports both 8-bit and Unicode strings; both
    the pattern and the strings being processed can contain null bytes and
    characters outside the US ASCII range.

    Regular expressions can contain both special and ordinary characters.
    Most ordinary characters, like "A", "a", or "0", are the simplest
    regular expressions; they simply match themselves.  You can
    concatenate ordinary characters, so last matches the string 'last'.

    The special characters are:
        "."      Matches any character except a newline.
        "^"      Matches the start of the string.
        "$"      Matches the end of the string or just before the newline at
                 the end of the string.
        "*"      Matches 0 or more (greedy) repetitions of the preceding RE.
                 Greedy means that it will match as many repetitions as possible.
        "+"      Matches 1 or more (greedy) repetitions of the preceding RE.
```

Accessing embedded documentation on a single function

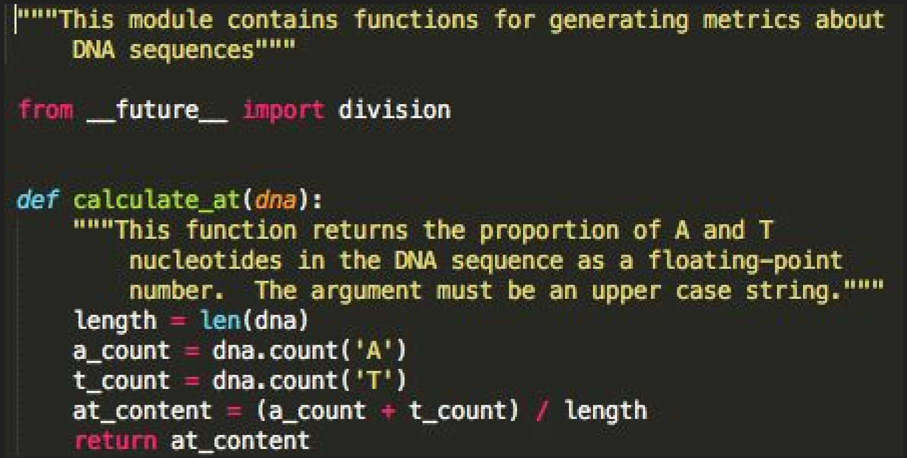


```
>>> help(re.search)
```

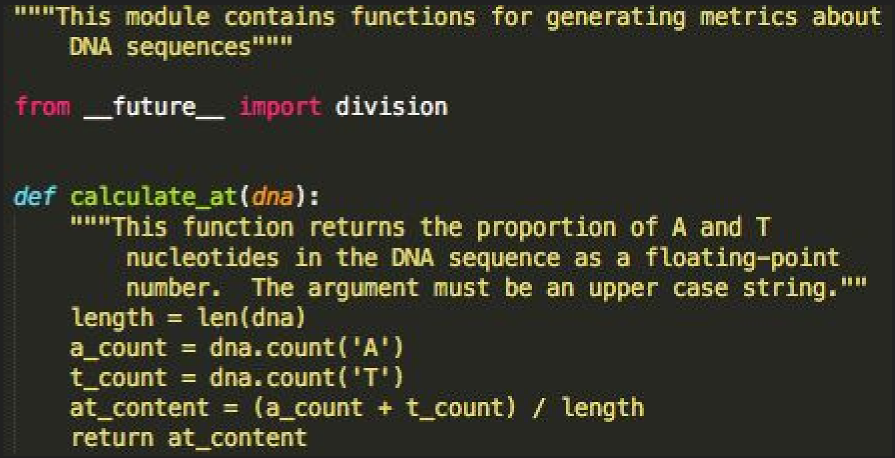


```
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern,
 returning
    a match object, or None if no match was found.
```

Docstrings: Adding our own documentation



```python
"""This module contains functions for generating metrics about
    DNA sequences"""

from __future__ import division


def calculate_at(dna):
    """This function returns the proportion of A and T
        nucleotides in the DNA sequence as a floating-point
        number.  The argument must be an upper case string."""
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```



```python
"""This module contains functions for generating metrics about
    DNA sequences"""

from __future__ import division


def calculate_at(dna):
    """This function returns the proportion of A and T
        nucleotides in the DNA sequence as a floating-point
        number.  The argument must be an upper case string."""
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```

Testing Goals

Think about test cases: What should your code do when provided with a specific input? What should happen if it doesn’t produce the expected result?

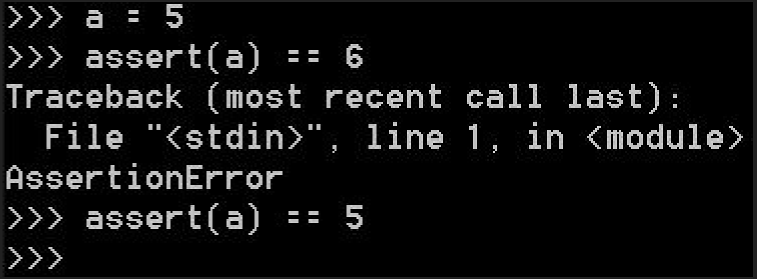
Ideally, a failed test should tell you exactly what went wrong and where.

Decide whether you want your test program to stop after the first failed test, or continue performing additional tests.

Test each small bit of code. Make it modular.

Assert statements

This is the simplest way to test that the code produces the output you expect it to.

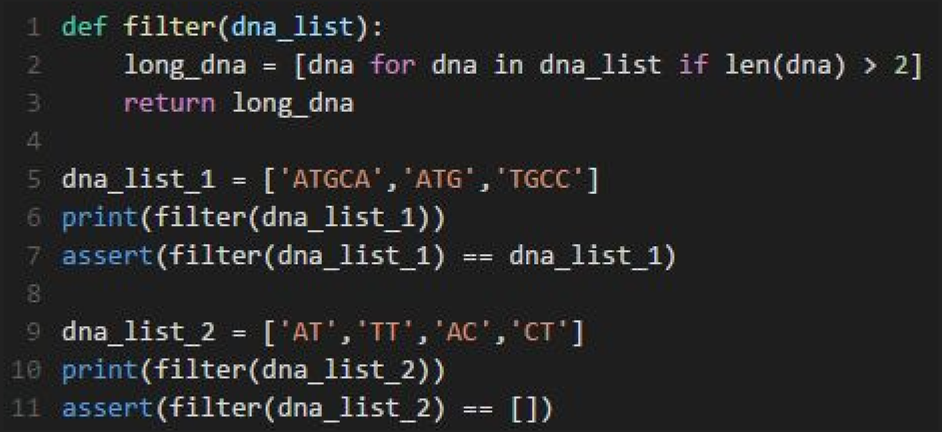


```
>>> a = 5
>>> assert(a) == 6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert(a) == 5
>>>
```

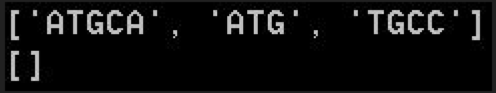
If the specified condition isn't true, the program will raise an AssertionError and fail out.

Multiple assertion statements get unwieldy quickly with complex bits of code. Additionally, the exception raised doesn't tell us much about what went wrong. The unittest module provides a better testing structure for these reasons.

Example



```python
def filter(dna_list):
    long_dna = [dna for dna in dna_list if len(dna) > 2]
    return long_dna

dna_list_1 = ['ATGCA','ATG','TGCC']
print(filter(dna_list_1))
assert(filter(dna_list_1) == dna_list_1)

dna_list_2 = ['AT','TT','AC','CT']
print(filter(dna_list_2))
assert(filter(dna_list_2) == [])
```



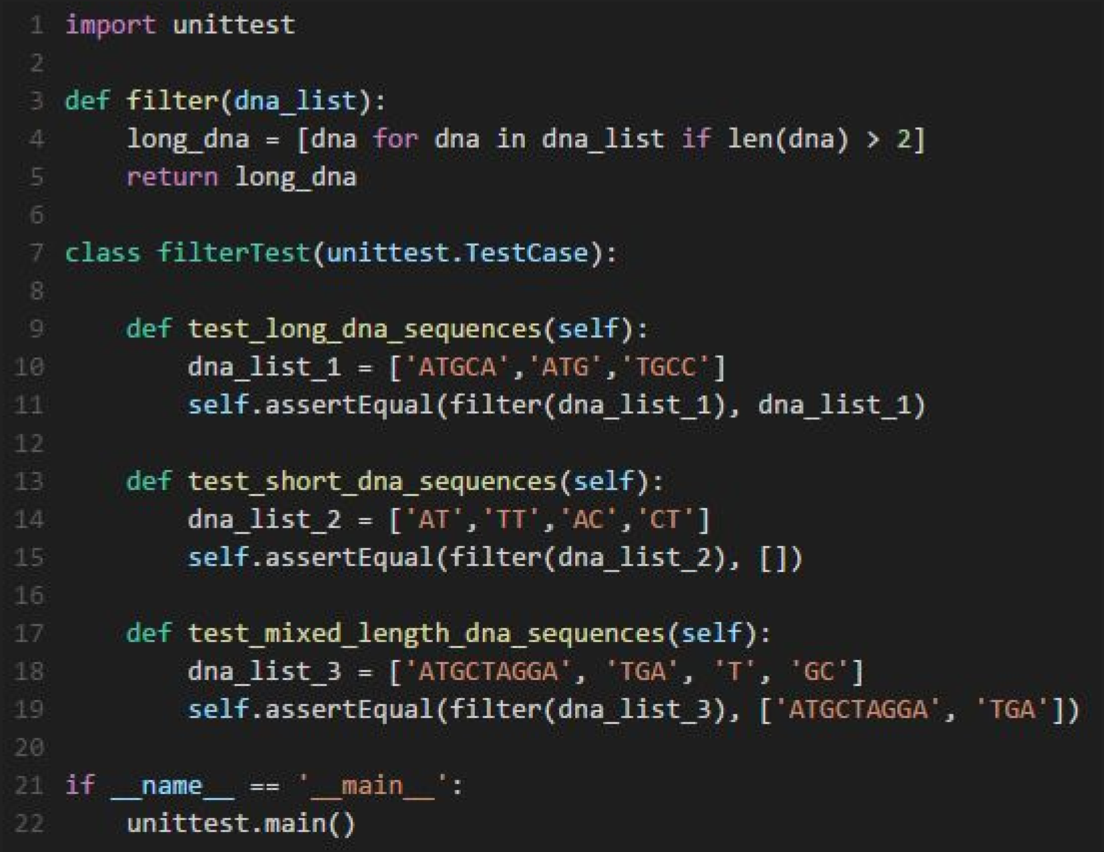
```
['ATGCA', 'ATG', 'TGCC']
[]
```

No AssertionErrors were raised, so our code passed the tests successfully.

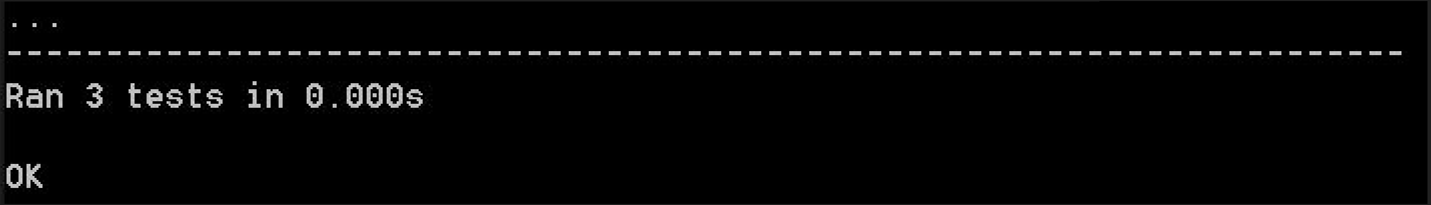
Positive control: does the code produce the results we want when we feed it good data?

Negative control: does the code correctly fail to give us results when we feed it bad data?

Writing a Test Class

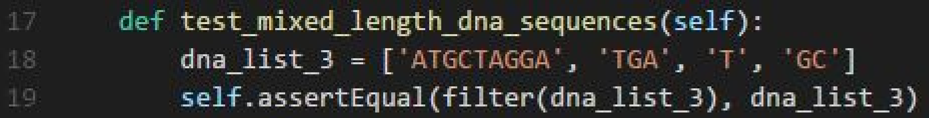


```python
import unittest

def filter(dna_list):
    long_dna = [dna for dna in dna_list if len(dna) > 2]
    return long_dna

class filterTest(unittest.TestCase):

    def test_long_dna_sequences(self):
        dna_list_1 = ['ATGCA','ATG','TGCC']
        self.assertEqual(filter(dna_list_1), dna_list_1)

    def test_short_dna_sequences(self):
        dna_list_2 = ['AT','TT','AC','CT']
        self.assertEqual(filter(dna_list_2), [])

    def test_mixed_length_dna_sequences(self):
        dna_list_3 = ['ATGCTAGGA', 'TGA', 'T', 'GC']
        self.assertEqual(filter(dna_list_3), ['ATGCTAGGA', 'TGA'])

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

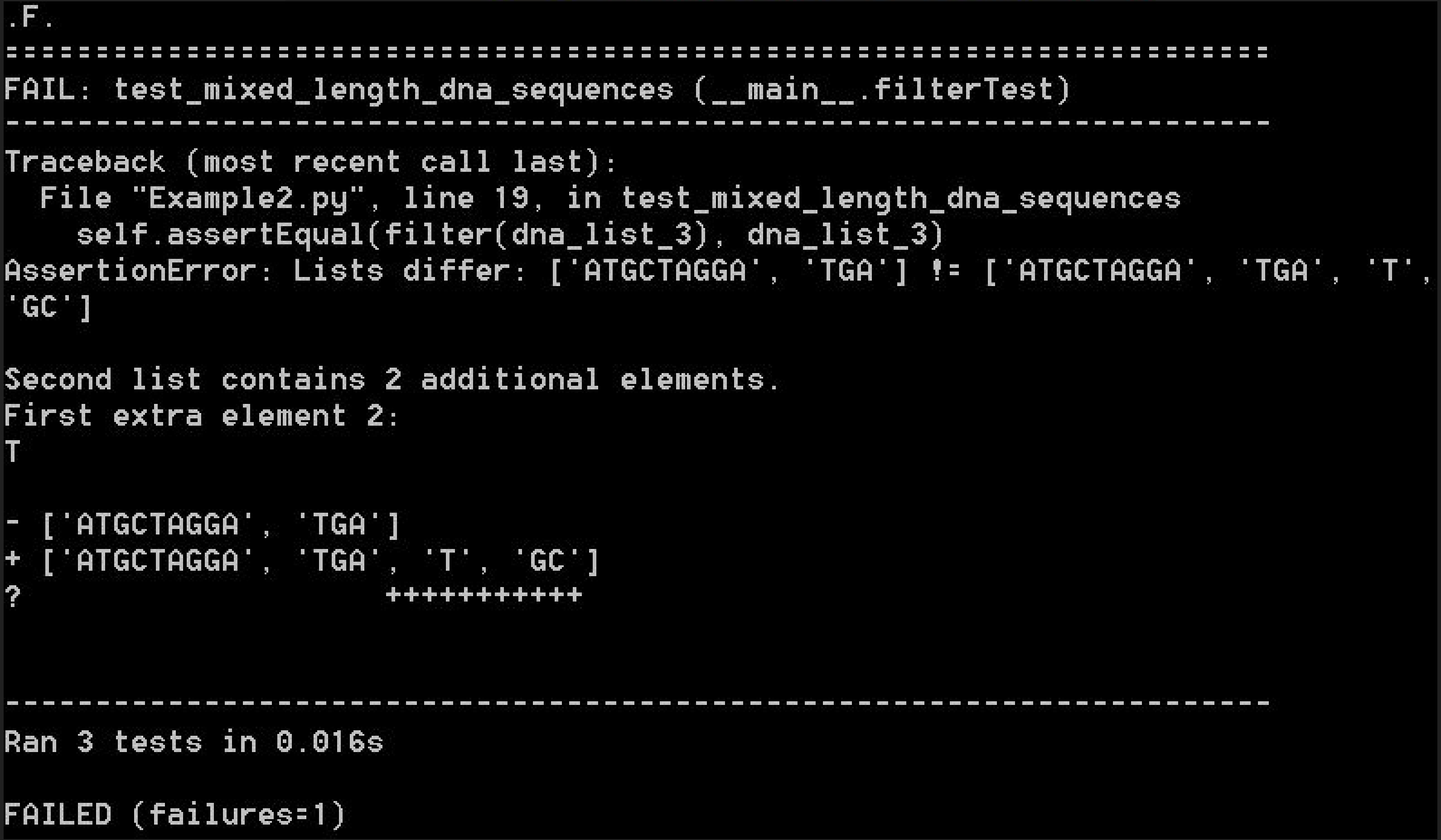


```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

- imports unittest module
- unittest.TestCase is a class inside the unittest module
- assertEqual is a method of the unittest.TestCase class
- the last if statement assures that the tests are only run if the script is run from the command line
- each test MUST start with test
- tests are run alphabetically



```
17      def test_mixed_length_dna_sequences(self):
18          dna_list_3 = ['ATGCTAGGA', 'TGA', 'T', 'GC']
19          self.assertEqual(filter(dna_list_3), dna_list_3)
```



```
.F.
======================================================================
FAIL: test_mixed_length_dna_sequences (__main__.filterTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "Example2.py", line 19, in test_mixed_length_dna_sequences
    self.assertEqual(filter(dna_list_3), dna_list_3)
AssertionError: Lists differ: ['ATGCTAGGA', 'TGA'] != ['ATGCTAGGA', 'TGA', 'T',
'GC']

Second list contains 2 additional elements.
First extra element 2:
T

- ['ATGCTAGGA', 'TGA']
+ ['ATGCTAGGA', 'TGA', 'T', 'GC']
?                     +++++++++++



----------------------------------------------------------------------
Ran 3 tests in 0.016s

FAILED (failures=1)
```

Problems with Multiple Test Interactions

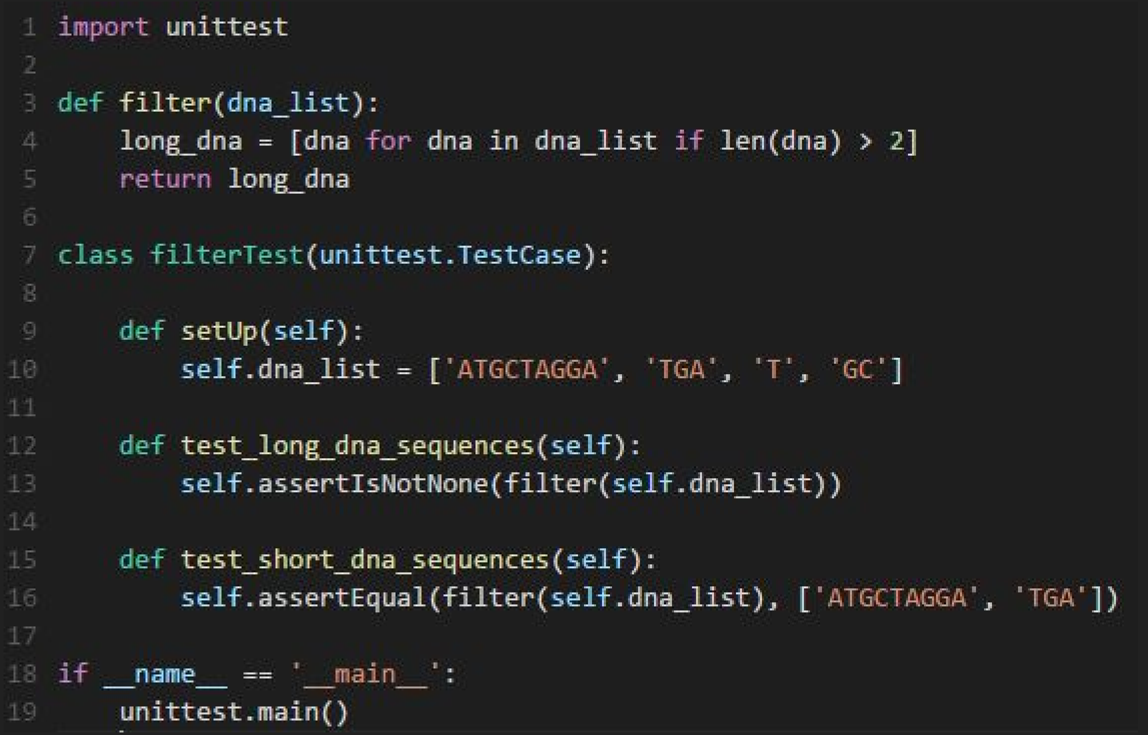
What if I was using the same dna_list for all the tests? The first test would remove any dna sequences less than three base pairs, and then any following tests would have nothing more to remove. Essentially, I would gain no information at all from running any tests after the first one.

Fortunately, there’s a way to get around this.

setUp and tearDown

The setUp method will regenerate the dna_list variable before each test.

There's also a tearDown method to do a specified action after each test in case you want to delete temporary files or close network connections.



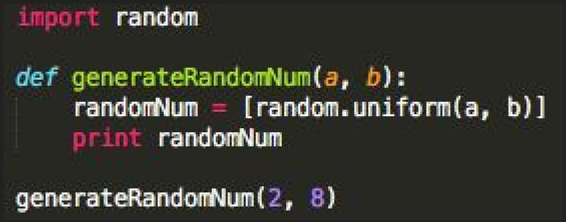
```python
import unittest

def filter(dna_list):
    long_dna = [dna for dna in dna_list if len(dna) > 2]
    return long_dna

class filterTest(unittest.TestCase):

    def setUp(self):
        self.dna_list = ['ATGCTAGGA', 'TGA', 'T', 'GC']

    def test_long_dna_sequences(self):
        self.assertIsNotNone(filter(self.dna_list))

    def test_short_dna_sequences(self):
        self.assertEqual(filter(self.dna_list), ['ATGCTAGGA', 'TGA'])

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

Coding Challenge!

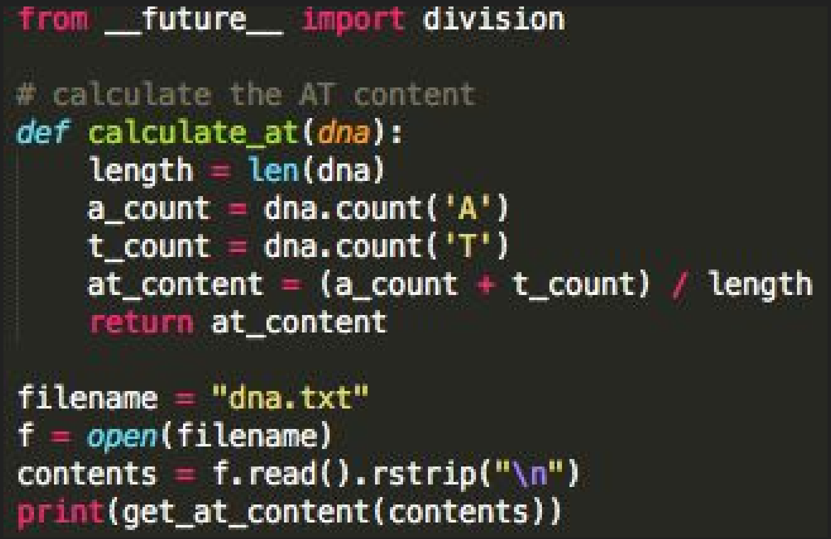
Coding Challenge - Modules

Write your own module that does something - anything!

Make sure it includes at least one function that can be called



```python
import random

def generateRandomNum(a, b):
    randomNum = [random.uniform(a, b)]
    print randomNum

generateRandomNum(2, 8)
```



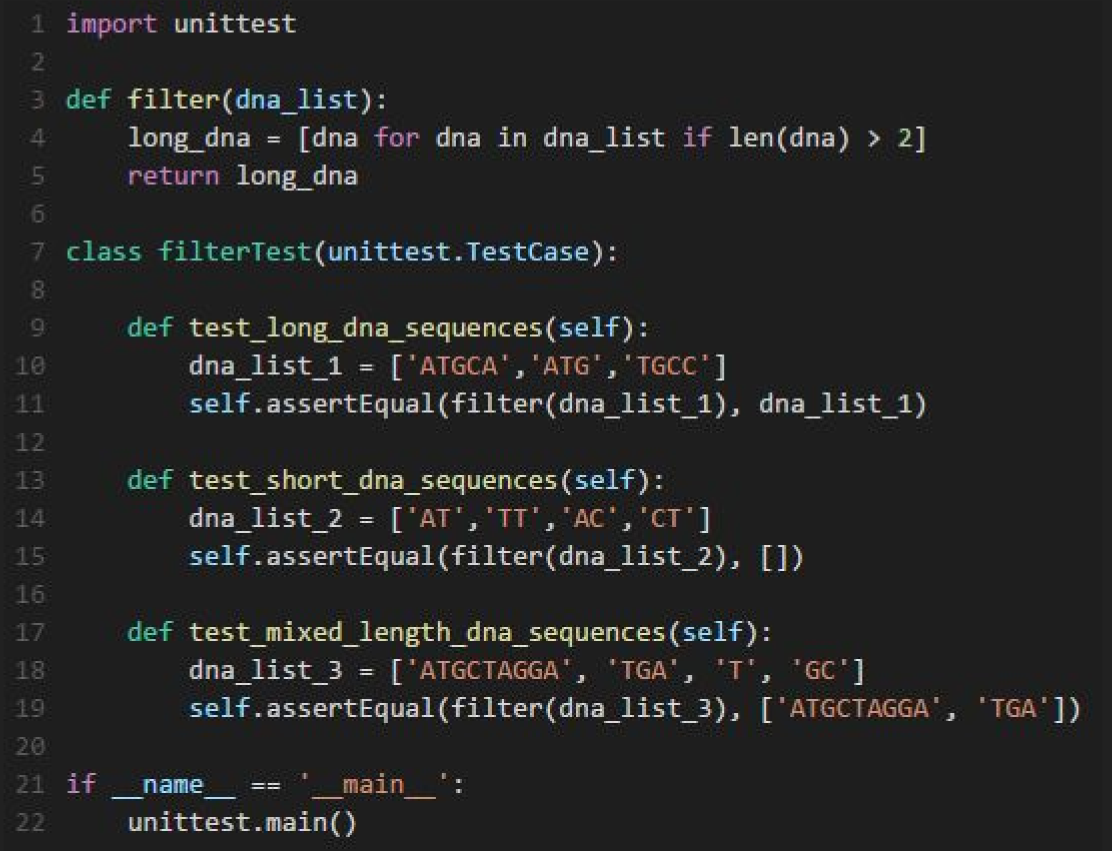
```python
from __future__ import division

# calculate the AT content
def calculate_at(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content

filename = "dna.txt"
f = open(filename)
contents = f.read().rstrip("\n")
print(get_at_content(contents))
```

Coding Challenge - Testing

Write a basic test for your new module.

Instead of defining your function at the top, import it as a module.

Example code is shown here.



```python
import unittest

def filter(dna_list):
    long_dna = [dna for dna in dna_list if len(dna) > 2]
    return long_dna

class filterTest(unittest.TestCase):

    def test_long_dna_sequences(self):
        dna_list_1 = ['ATGCA','ATG','TGCC']
        self.assertEqual(filter(dna_list_1), dna_list_1)

    def test_short_dna_sequences(self):
        dna_list_2 = ['AT','TT','AC','CT']
        self.assertEqual(filter(dna_list_2), [])

    def test_mixed_length_dna_sequences(self):
        dna_list_3 = ['ATGCTAGGA', 'TGA', 'T', 'GC']
        self.assertEqual(filter(dna_list_3), ['ATGCTAGGA', 'TGA'])

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