

STATS 507

Data Analysis in Python

Lecture 22: Algorithms, Profiling and Testing

Some material adapted from Appendix B of A. Downey's *Think Python*

<http://greenteapress.com/wp/think-python-2e/>

What makes a good algorithm?

We have seen examples of good and bad data structures for a task

Ex: list vs set/dictionary for testing set membership

Ex: certain operations on pandas tables are fast

How do we make such judgments?

What makes a good algorithm?

We have seen examples of good and bad data structures for a task

Ex: list vs set/dictionary for testing set membership

Ex: certain operations on pandas tables are fast

How do we make such judgments?

Answer 1: run timing experiments (i.e., profile our code)

But then our answer to “what algorithm/structure is better?” is highly machine- and implementation-dependent.

What makes a good algorithm?

We have seen examples of good and bad data structures for a task

Ex: list vs set/dictionary for testing set membership

Ex: certain operations on pandas tables are fast

How do we make such judgments?

Answer 2: algorithmic analysis

Provides a theoretical framework for comparing algorithms in terms of **worst-case** runtime and space requirements (i.e., how long they run and how much memory they need).

Measuring time and space usage

We measure an algorithm's runtime and space usage in terms of input size n
e.g., number of objects in a set, length of a list to be sorted, etc.

Example: Suppose algorithm A takes $100n+1$ steps of computation to solve a problem of size n while algorithm B takes n^2+n+1

Input size	Runtime of A	Runtime of B
10	1001	111
100	10001	10101
1 000	100001	1001001
10 000	1000001	$>10^8$

B looks better than A for smaller inputs, but for n large, A is **much** faster than B. This is the motivation for **asymptotic analysis**, in which we compare algorithms based on their leading-order runtime terms.

Big-O notation

We form equivalence classes of runtimes according to these leading-order terms
e.g., $10n+1$, $2n-1$, $n+1000$, are all $O(n)$ because leading-order terms are n

Test your understanding: what order are each of the following?

$$10n^3-n+1$$

$$n-100$$

$$n^2+n+1$$

$$1000$$

Big-O notation

We form equivalence classes of runtimes according to these leading-order terms
e.g., $10n+1$, $2n-1$, $n+1000$, are all $O(n)$ because leading-order terms are n

Test your understanding:

$10n^3-n+1$	$O(n^3)$
$n-100$	$O(n)$
n^2+n+1	$O(n^2)$
1000	$O(1)$

Big-O notation

We form equivalence classes of runtimes according to these leading-order terms
e.g., $10n+1$, $2n-1$, $n+1000$, are all $O(n)$ because leading-order terms are n

Test your understanding:

$10n^3-n+1$	$O(n^3)$
$n-100$	$O(n)$
n^2+n+1	$O(n^2)$
1000	$O(1)$

c is any constant
(doesn't depend on n).

Order	Common Name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Runtimes of basic Python operations

Arithmetic: addition, subtraction, multiplication, division, all constant time*

Indexing: run in constant time, regardless of the size of the sequence

Note: this is **not** the same as the time to check every entry of a sequence

For-loop and reduce-like operations: linear time in the length of the sequence

Provided that each operation in the for loop is constant-time.

* technically, this is only approximately true

Runtimes of basic Python operations

Arithmetic: addition, subtraction, multiplication, division, all constant time*

Indexing: run in constant time, regardless of the size of the sequence

Note: this is **not** the same as the time to check every entry of a sequence

For-loop and reduce-like operations: linear time in the length of the sequence

Provided that each operation in the for loop is constant-time.

```
1 s = 0
2 for x in t:
3     s += x
```

Each addition requires 1 unit of computation (i.e., constant-order computation time).

We perform constant-order computational work for each element of list t , so the total runtime to sum the elements is proportional to the length of list t .

* technically, this is only approximately true

Constant-order work on each element of a list

Experiment: create lists of different lengths, time how long it takes to sum the elements of a list of that length. We expect to see **linear dependence**.

`seqLens` stores the different sequence lengths we're going to use.

```
1 seqLens = np.arange(1e5, 1e6, 1e4)
2 runtimes = np.zeros(len(seqLens))
3 for n in range(len(seqLens)):
4     slen = int(seqLens[n])
5     seq = np.random.random(size=slen)
6     tstart = time.time()
7     sum(seq)
8     tend = time.time()
9     runtimes[n] = tend - tstart
```

Constant-order work on each element of a list

Experiment: create lists of different lengths, time how long it takes to sum the elements of a list of that length. We expect to see **linear dependence**.

```
1 seqLens = np.arange(1e5, 1e6, 1e4)
2 runtimes = np.zeros(len(seqLens))
3 for n in range(len(seqLens)):
4     slen = int(seqLens[n])
5     seq = np.random.random(size=slen)
6     tstart = time.time()
7     sum(seq)
8     tend = time.time()
9     runtimes[n] = tend - tstart
```

`seqLens` stores the different sequence lengths we're going to use.

For each length, generate a random list of numbers...

Constant-order work on each element of a list

Experiment: create lists of different lengths, time how long it takes to sum the elements of a list of that length. We expect to see **linear dependence**.

```
1 seqLens = np.arange(1e5, 1e6, 1e4)
2 runtimes = np.zeros(len(seqLens))
3 for n in range(len(seqLens)):
4     slen = int(seqLens[n])
5     seq = np.random.random(size=slen)
6     tstart = time.time()
7     sum(seq)
8     tend = time.time()
9     runtimes[n] = tend - tstart
```

`seqLens` stores the different sequence lengths we're going to use.

For each length, generate a random list of numbers...

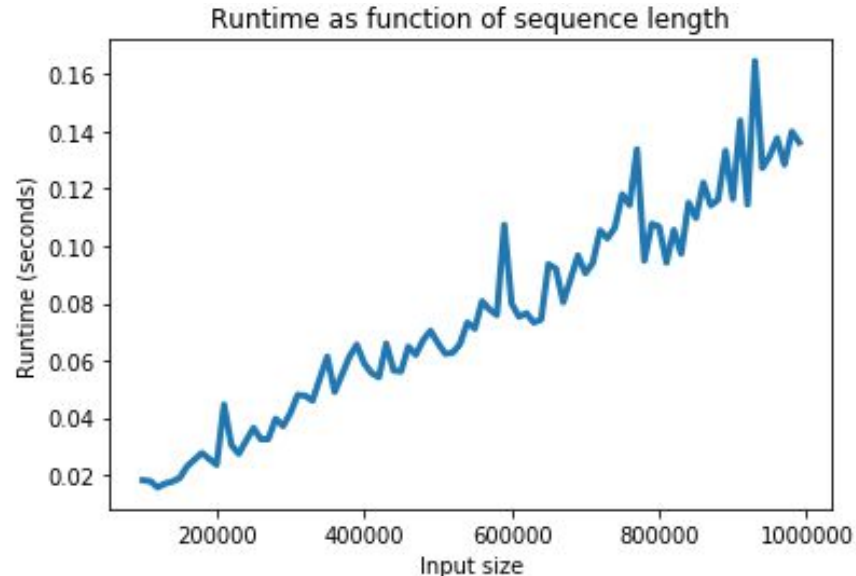
...and time how long it takes to sum them up.

Constant-order work on each element of a list

Experiment: create lists of different lengths, time how long it takes to sum the elements of a list of that length. We expect to see **linear dependence**.

```
1 seqLens = np.arange(1e5, 1e6, 1e4)
2 runtimes = np.zeros(len(seqLens))
3 for n in range(len(seqLens)):
4     slen = int(seqLens[n])
5     seq = np.random.random(size=slen)
6     tstart = time.time()
7     sum(seq)
8     tend = time.time()
9     runtimes[n] = tend - tstart
```

```
1 plt.plot(seqLens, runtimes, linewidth=3)
2 plt.title('Runtime as function of sequence length')
3 plt.xlabel('Input size')
4 plt.ylabel('Runtime (seconds)')
```

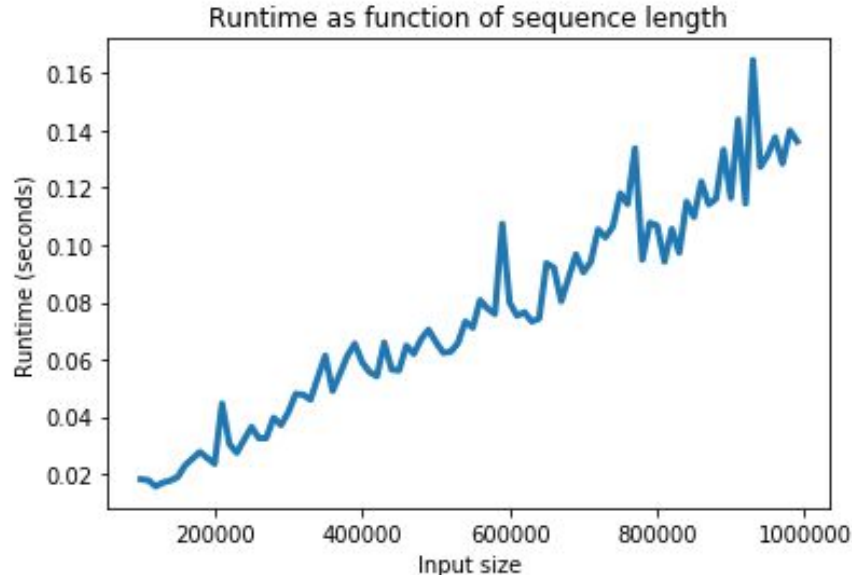


Constant-order work on each element of a list

Experiment: create lists of different lengths, time how long it takes to sum the elements of a list of that length. We expect to see **linear dependence**.

Note: there is some variability here because other processes were running on my computer at the same time as the experiment.

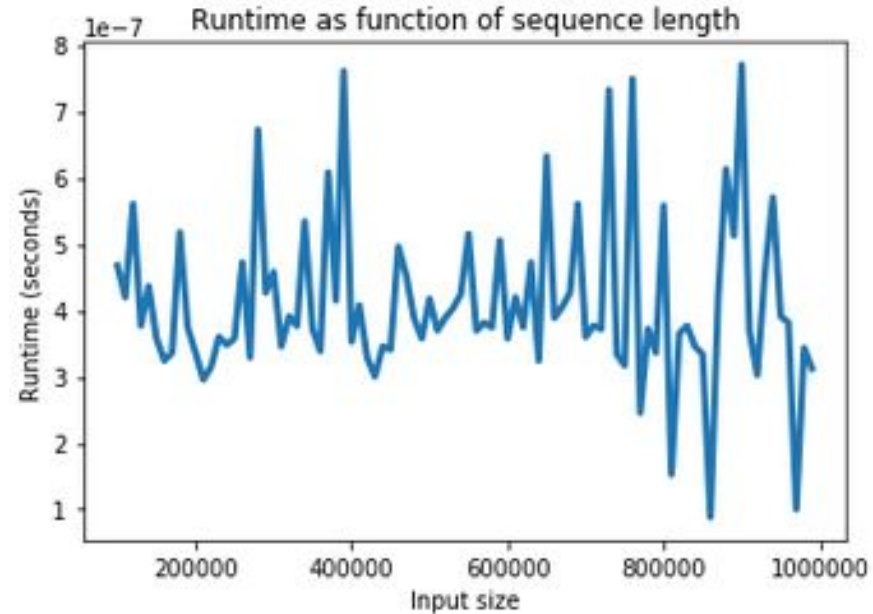
```
1 seqLens = np.arange(1e5, 1e6, 1e4)
2 runtimes = np.zeros(len(seqLens))
3 for n in range(len(seqLens)):
4     slen = int(seqLens[n])
5     seq = np.random.random(size=slen)
6     tstart = time.time()
7     sum(seq)
8     tend = time.time()
9     runtimes[n] = tend - tstart
```



Interesting side-note: `len(t)` is constant time

Experiment: create lists of different lengths, time how long it takes to get the length of the list.

```
1 seqLens = np.arange(1e5, 1e6, 1e4)
2 nTrials=100
3 runtimes = np.zeros((len(seqLens), nTrials))
4 for n in range(len(seqLens)):
5     slen = int(seqLens[n])
6     seq = list(np.random.random(size=slen))
7     for m in range(nTrials):
8         tstart = time.time()
9         len(seq)
10        tend = time.time()
11        runtimes[n,m] = tend-tstart
```



`len(seq)` takes constant time because in Python, the length is an attribute of a list, which gets updated whenever the list is changed.

Sorting

Problem: given a list, sort the list in ascending order

The best sorting algorithms sort a length- n list time $O(n \log n)$

But let's first look at some suboptimal sorting algorithms

```
1 def argmax(t):
2     if len(t)==0: # Handle a weird edge case.
3         return (None, float('-inf'))
4     (i,m)=(0,t[0])
5     for j in range(1,len(t)):
6         if t[j] > m:
7             (i,m) = (j,t[j])
8     return (i,m)
9 def naive_sort(t):
10    n=len(t)
11    for k in range(1,len(t)):
12        # Find the largest element and its index
13        (i,m) = argmax(t[:n-k+1])
14        # Swap the maximum with the last element
15        (t[i],t[n-k])=(t[n-k],m)
16    return t
```

This is called **selection sort**. We look for the biggest element, move it to the end of the list, and then repeat on the rest of the list.

`argmax` finds the largest element and its index.

Sorting

Problem: given a list, sort it in ascending order

The best sorting algorithms sort a length- n list time $O(n \log n)$

But let's first look at some suboptimal sorting algorithms

```
1 def argmax(t):
2     if len(t)==0: # Handle a weird edge case.
3         return (None, float('-inf'))
4     (i,m)=(0,t[0])
5     for j in range(1,len(t)):
6         if t[j] > m:
7             (i,m) = (j,t[j])
8     return (i,m)
9 def naive_sort(t):
10    n=len(t)
11    for k in range(1,len(t)):
12        # Find the largest element and its index
13        (i,m) = argmax(t[:n-k+1])
14        # Swap the maximum with the last element
15        (t[i],t[n-k])=(t[n-k],m)
16    return t
```

This is called **selection sort**. We look for the biggest element, move it to the end of the list, and then repeat on the rest of the list.

In the k -th iteration of the for-loop, we look at $n-k$ elements, so the total work is $1+2+\dots+n = O(n^2)$.

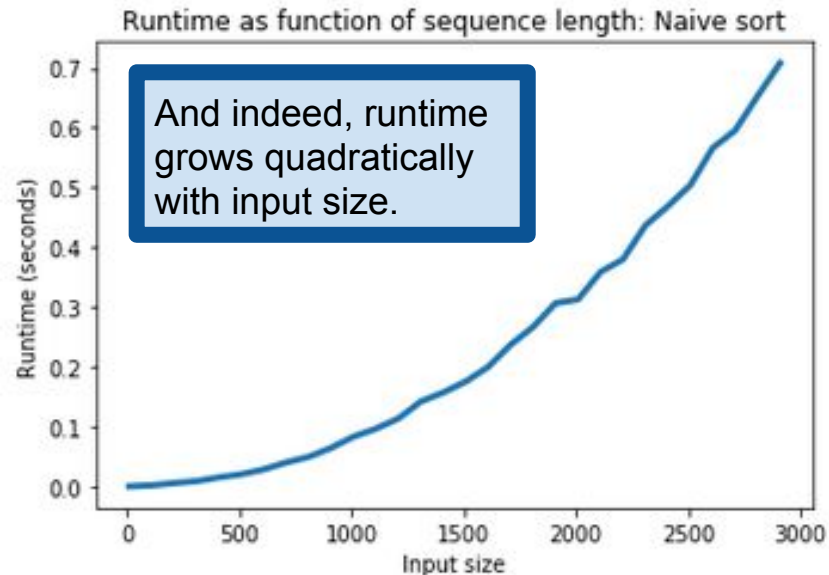
Sorting

Problem: given a list, sort it in ascending order

The best sorting algorithms sort a length- n list time $O(n \log n)$

But let's first look at some suboptimal sorting algorithms

```
1 def argmax(t):
2     if len(t)==0: # Handle a weird edge case.
3         return (None,float('-inf'))
4     (i,m)=(0,t[0])
5     for j in range(1,len(t)):
6         if t[j] > m:
7             (i,m) = (j,t[j])
8     return (i,m)
9 def naive_sort(t):
10    n=len(t)
11    for k in range(1,len(t)):
12        # Find the largest element and its index
13        (i,m) = argmax(t[:n-k+1])
14        # Swap the maximum with the last element
15        (t[i],t[n-k])=(t[n-k],m)
16    return t
```



https://en.wikipedia.org/wiki/Selection_sort

Sorting

Problem: given a list, sort it in ascending order

The best sorting algorithms sort a length- n list time $O(n \log n)$

```
1 def quicksort(t):
2     if len(t) <= 1:
3         return t
4     (less,mid,more) = (list(),list(),list())
5     pivot = t[0]
6     mid.append(t[0])
7     for i in range(1,len(t)):
8         if t[i] == pivot:
9             mid.append(t[i])
10        elif t[i] < pivot:
11            less.append(t[i])
12        else: # t[i] > pivot
13            more.append(t[i])
14    return quicksort(less) + mid + quicksort(more)
```

This is called **quicksort**. We pick a “pivot” element from the list, split the list into elements less than, equal to, and greater than the pivot, and recurse on the less-than and greater-than lists. This pattern should look familiar from your binary search problem in HW2.

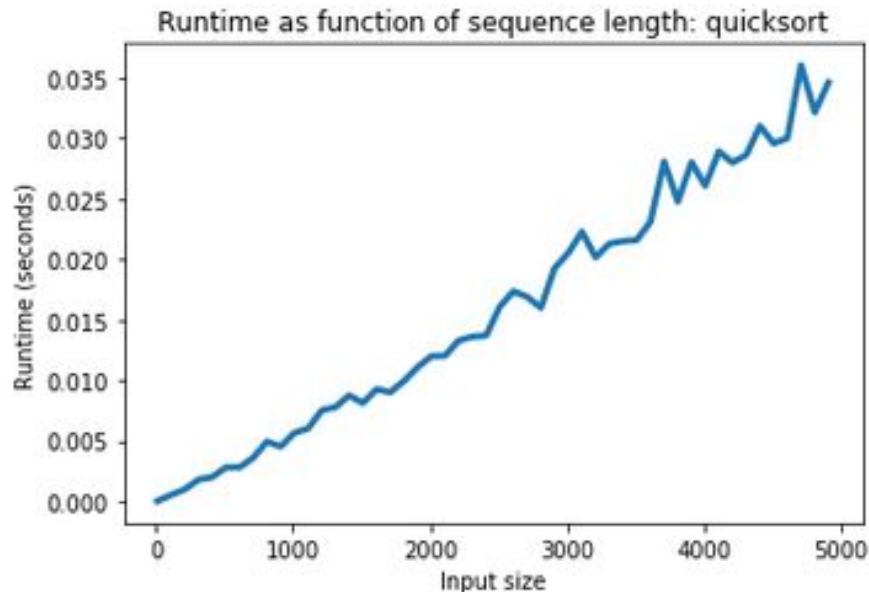
This recursion is the important part. `less` and `more` contain the elements less than and greater than the pivot, but they may not yet be sorted.

Sorting

Problem: given a list, sort it in ascending order

The best sorting algorithms sort a length- n list time $O(n \log n)$

```
1 def quicksort(t):
2     if len(t) <= 1:
3         return t
4     (less,mid,more) = (list(),list(),list())
5     pivot = t[0]
6     mid.append(t[0])
7     for i in range(1,len(t)):
8         if t[i] == pivot:
9             mid.append(t[i])
10        elif t[i] < pivot:
11            less.append(t[i])
12        else: # t[i] > pivot
13            more.append(t[i])
14    return quicksort(less) + mid + quicksort(more)
```



Sorting

Problem: given a list, sort it in ascending order

The best sorting algorithms sort a length- n list time $O(n \log n)$

```
1 def quicksort(t):
2     if len(t) <= 1:
3         return t
4     (less,mid,more) = (list(),list(),list())
5     pivot = t[0]
6     mid.append(t[0])
7     for i in range(1,len(t)):
8         if t[i] == pivot:
9             mid.append(t[i])
10        elif t[i] < pivot:
11            less.append(t[i])
12        else: # t[i] > pivot
13            more.append(t[i])
14    return quicksort(less) + mid + quicksort(more)
```

Proving that quicksort takes $O(n \log n)$ runtime is beyond the scope of this course, but it should be intuitively clear: the runtime $T(n)$ as a function of n should obey $T(n) = 2 \cdot T(n/2) + C$ for some constant C , and $T(n) = n \log n$ is such a function.

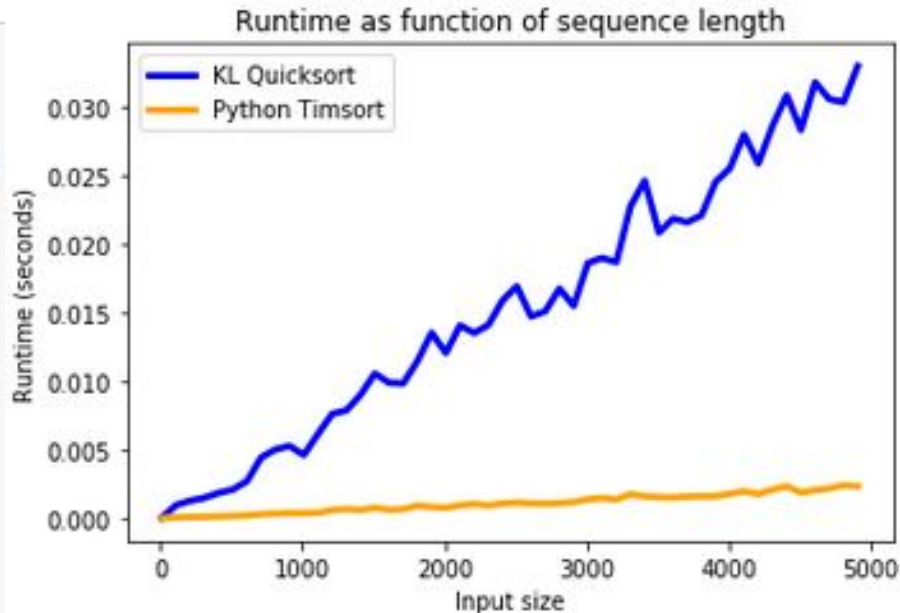
Aside: the house always wins, Python edition

If there is a Python implementation of the thing you are trying to do, use it.

(and the same goes all the more so for numpy/scipy!)

You should not expect to out-wit the Python developers!

```
1 seqLens = np.arange(1e1,5*1e3,1e2)
2 ntrials=10
3 myruntimes = np.zeros((len(seqLens),ntrials))
4 pythonruntimes = np.zeros((len(seqLens),ntrials))
5 for n in range(len(seqLens)):
6     slen = int(seqLens[n])
7     seq = list(np.random.random(size=slen))
8     for m in range(ntrials):
9         tstart = time.time()
10        quicksort(seq)
11        tend = time.time()
12        myruntimes[n,m] = tend-tstart
13        tstart = time.time()
14        sorted(seq)
15        tend = time.time()
16        pythonruntimes[n,m] = tend-tstart
```



<https://en.wikipedia.org/wiki/Timsort>

Profiling Code

Say you've written some code, but it's fairly slow

How should you spend your time in optimizing it?

Most software engineers would agree that you should find the slowest part of your program and concentrate on making that part faster.

A **profiler** is a program that runs other programs and summarizes how long each part took to run.

time: the simplest approach

Sometimes, all we want to do is compare the runtimes of two different solutions to a problem. For this, the `time` module is often enough.

But note that timing in this way doesn't tell us **where** in the process of checking set membership we are taking all our time.

Other profiling tools will give us more granular summaries of runtime information.

```
1 import time
2 from random import randint
3 listlen = 1000000
4 list_of_numbers = listlen*[0]
5 dict_of_numbers = dict()
6 for i in range(listlen):
7     n = randint(1000000,9999999)
8     list_of_numbers[i] = n
9     dict_of_numbers[n] = 1
```

```
1 start_time = time.time()
2 8675309 in list_of_numbers
3 time.time() - start_time
```

0.027842044830322266

```
1 start_time = time.time()
2 8675309 in dict_of_numbers
3 time.time() - start_time
```

0.0001232624053955078

profile and cProfile

The two packages are so similar that they share a documentation page:
<https://docs.python.org/3/library/profile.html>

Two related modules that both support profiling of code.

`cProfile` is implemented in C, and thus avoids some of the overhead of Python

`profile` is basically the same as `cProfile`, but more is implemented in Python
More features, at the cost of (slightly) less accurate timing

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

```
3 function calls in 0.026 seconds
```

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	<string>:1(<module>)
1	0.000	0.000	0.026	0.026	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Unless you're doing some serious software engineering, `cProfile` is probably right for you.

profile and cProfile

Profiling your code is simple: pass the command that you want to profile, **as a string**, to the profiler's `run` method.

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

3 function calls in 0.026 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	<string>:1(<module>)
1	0.000	0.000	0.026	0.026	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

cProfile uses the `exec` function to run a string as Python code.
<https://docs.python.org/3.5/library/functions.html#exec>

profile and cProfile

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

3 function calls in 0.026 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	<string>:1(<module>)
1	0.000	0.000	0.026	0.026	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Number of times each function was called

profile and cProfile

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

3 function calls in 0.026 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	<string>:1(<module>)
1	0.000	0.000	0.026	0.026	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Total time spent inside this function
(but not in subcalls of the function).

profile and cProfile

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

3 function calls in 0.026 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	<string>:1(<module>)
1	0.000	0.000	0.026	0.026	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Total time per call (averaged over all calls to the function).

profile and cProfile

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

3 function calls in 0.026 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	<string>:1(<module>)
1	0.000	0.000	0.026	0.026	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Total time spent in the function,
including function subcalls.

profile and cProfile

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

3 function calls in 0.026 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	string:1(<module>)
1	0.000	0.000	0.026	0.026	built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	method 'disable' of '_lsprof.Profiler' objects}

Cumulative time spent in the function,
including function subcalls.

profile and cProfile

```
1 import cProfile
2 cProfile.run('8675309 in list_of_numbers')
```

3 function calls in 0.026 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.026	0.026	0.026	0.026	<string>:1(<module>)
1	0.000	0.000	0.026	0.026	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Names of the functions, with their files and line numbers.

profile and cProfile

Recall that this is slow....

...while this is fast.

But why is one faster than the other, and where does the slow one spend all its time?...

```
1 def naive_fibo(n):
2     if n < 0:
3         raise ValueError('Negative Fibonacci number?')
4     if n==0:
5         return 0
6     elif n==1:
7         return 1
8     else:
9         return naive_fibo(n-1) + naive_fibo(n-2)
10
11 known = {0:0, 1:1}
12 def fibo(n):
13     if n in known:
14         return known[n]
15     else:
16         f = fibo(n-1) + fibo(n-2)
17         known[n] = f
18     return(f)
```

fibonacci.py

```
1 import fibonacci
2 cProfile.run('fibonacci.naive_fibo(30)')
```

2692540 function calls (4 primitive calls) in 2.583 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	2.583	2.583	<string>:1(<module>)
2692537/1	2.583	0.000	2.583	2.583	fibonacci.py:1(naive_fibo)
1	0.000	0.000	2.583	2.583	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

```
1 cProfile.run('fibonacci.fibo(30)')
```

62 function calls (4 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)
59/1	0.000	0.000	0.000	0.000	fibonacci.py:12(fibo)
1	0.000	0.000	0.000	0.000	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

```
1 import fibonacci
2 cProfile.run('fibonacci.naive_fibo(30)')
```

2692540 function calls (4 primitive calls) in 2.583 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	2.583	2.583	<string>:1(<module>)
2692537/1	2.583	0.000	2.583	2.583	fibonacci.py:1(naive_fibo)
1	0.000	0.000	2.583	2.583	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

naive_fibo(30) results in >2.5M (recursive) calls!

```
1 cProfile.run('fibonacci.fibo(30)')
```

62 function calls (4 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)
59/1	0.000	0.000	0.000	0.000	fibonacci.py:12(fibo)
1	0.000	0.000	0.000	0.000	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

```
1 import fibonacci
2 cProfile.run('fibonacci.naive_fibo(30)')
```

2692540 function calls (4 primitive calls) in 2.583 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	2.583	2.583	<string>:1(<module>)
2692537/1	2.583	0.000	2.583	2.583	fibonacci.py:1(naive_fibo)
1	0.000	0.000	0.000	0.000	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Note: the total time per call is negligible, but the cumulative time is not.

```
1 cProfile.run('fibonacci.fibo(30)')
```

62 function calls (4 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)
59/1	0.000	0.000	0.000	0.000	fibonacci.py:12(fibo)
1	0.000	0.000	0.000	0.000	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

A more realistic example: fitting a model

This example code uses `numpy` and `sklearn`, the latter of which you don't know about, yet. For now, it's enough to know that: `generate_data` generates data from a simple linear model and saves it to a pair of files; `load_data` loads data from those files; and `olsmodel.fit(x,y)` fits the model `olsmodel` to the data `x, y`.

This function is the important part. It generates data, writes it to a file, reads it back in and fits a model. Let's see where Python spends most of its time in this function.

```
1  import numpy as np                                ols_expt.py
2  from sklearn import linear_model
3  def generate_data(n, beta, Xfile, Yfile):
4      p = beta.size # beta is a numpy vector.
5      # Each data point is drawn indep'ly with
6      # independent Laplace-distributed entries
7      x = np.random.laplace(0, 1, size=(n,p))
8      # Observed data is beta^T x + normal noise.
9      noise = np.random.normal(0, 100, size=n)
10     y = np.matmul(beta,x.T) + noise
11     np.savetxt(Xfile, x)
12     np.savetxt(Yfile, y)
13 def load_data(Xfile,Yfile):
14     x = np.loadtxt(Xfile)
15     y = np.loadtxt(Yfile)
16     return (x,y)
17 def run_experiment(n, beta, Xfile, Yfile):
18     generate_data(n,beta,Xfile,Yfile)
19     (x,y) = load_data(Xfile,Yfile)
20     olsmodel = linear_model.LinearRegression()
21     olsmodel.fit(x,y)
```

Reminder of what our experiment does

```
def run_experiment(n, beta, Xfile, Yfile):  
    generate_data(n, beta, Xfile, Yfile)  
    (x, y) = load_data(Xfile, Yfile)  
    olsmodel = linear_model.LinearRegression()  
    olsmodel.fit(x, y)
```

```
1 import cProfile  
2 from ols_expt import *  
3 cProfile.run('run_experiment(100000, np.array([1,2,-3,4,-5]), "x.dat", "y.dat")')
```

3804974 function calls (3704972 primitive calls) in 4.600 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
24	0.000	0.000	0.000	0.000	<frozen importlib._bootstrap>:997(_handle_fromlist)
1	0.000	0.000	0.000	0.000	
...	
1	0.000	0.000	0.000	0.000	numerictypes.py:962(find_common_type)
1	0.007	0.007	3.195	3.195	ols_expt.py:13(load_data)
1	0.001	0.001	4.599	4.599	ols_expt.py:17(run_experiment)
1	0.000	0.000	1.378	1.378	ols_expt.py:3(generate_data)
32	0.000	0.000	0.000	0.000	parse.py:109(_coerce_args)
16	0.000	0.000	0.000	0.000	parse.py:361(urlparse)

I cropped a bunch of output from the cProfile report.

Reminder of what our experiment does

```
def run_experiment(n, beta, Xfile, Yfile):  
    generate_data(n,beta,Xfile,Yfile)  
    (x,y) = load_data(Xfile,Yfile)  
    olsmodel = linear_model.LinearRegression()  
    olsmodel.fit(x,y)
```

```
1 import cProfile  
2 from ols_expt import *  
3 cProfile.run('run_experiment(100000, np.array([1,2,-3,4,-5]), "x.dat", "y.dat")')
```

3804974 function calls (3704972 primitive calls) in 4.600 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
24	0.000	0.000	0.000	0.000	bootstrap>:997(_handle_fromlist)
1	0.000	0.000	0.000	0.000	
1	0.000	0.000	0.000	0.000	
1	0.000	0.000	0.000	0.000	
1	0.007	0.007	3.195	3.195	ols_expt.py:13(load_data)
1	0.001	0.001	4.599	4.599	ols_expt.py:17(run_experiment)
1	0.000	0.000	1.378	1.378	ols_expt.py:3(generate_data)
16	0.000	0.000	0.000	0.000	parse.py:361(urlparse)

Important point: vast majority of the execution time is spent on I/O, vanishingly little on actual computation.

How do I know if my code works?

Once we've written a program, how do we verify that it works as intended?

Problems often have edge cases that we may not think of ahead of time

Easy to make mistakes in code

Until now, you probably have done something like:

1. Write a function to do something
2. Try running the function on a bunch of different inputs
3. Search for problems with print statements

How do I know if my code works?

Once we've written a program, how do we verify that it works as intended?

Problems often have edge cases that we may not think of

Easy to make mistakes in code

Until now, you probably have done something like:

1. Write a function to do something
2. Try running the function on a bunch of different inputs
3. Search for problems with print statements

This works well enough for small projects, but it doesn't scale well. Better is to write a **test suite** for your program.

How do I know if my code works?

How can we (more) systematically find errors like this one?

```
1 def is_prime(x):  
2     if n <= 1:  
3         return False  
4     elif n==2:  
5         return True  
6     else:  
7         ulim = math.ceil(math.sqrt(x))  
8         for k in range(2,ulim):  
9             if n%k==0:  
10                return False  
11                return True
```

```
1 is_prime(2)
```

True

```
1 is_prime(3)
```

True

```
1 is_prime(4)
```

True

Python unittest module

Supports nicely organized test suites for your program

Note: there are plenty of other testing suites out there

```
1 def is_prime(x):
2     if n <= 1:
3         return False
4     elif n==2:
5         return True
6     else:
7         ulim = math.ceil(math.sqrt(x))
8         for k in range(2,ulim):
9             if n%k==0:
10                return False
11            return True
```

```
1 class PrimeTest(unittest.TestCase):
2     def test_base(self):
3         self.assertFalse(is_prime(-1))
4         self.assertFalse(is_prime(0))
5         self.assertFalse(is_prime(1))
6         self.assertTrue(is_prime(2))
7         self.assertTrue(is_prime(3))
8     def test_seive(self):
9         # Composite numbers are not prime
10        for q in range(2,100):
11            for b in range(2,100):
12                self.assertFalse(is_prime(q*b))
13
```

unittest module:

<https://docs.python.org/3/library/unittest.html>

Python unittest module

Supports nicely organized test suites for your program

Note: there are plenty of other testing suites out there

```
1 def is_prime(x):
2     if n <= 1:
3         return False
4     elif n==2:
5         return True
6     else:
7         ulim = math.ceil(math.sqrt(x))
8         for k in range(2,ulim):
9             if n%k==0:
10                return False
11                return True
```

```
1 class PrimeTest(unittest.TestCase):
2     def test_base(self):
3         self.assertFalse(is_prime(-1))
4         self.assertFalse(is_prime(0))
5         self.assertFalse(is_prime(1))
6         self.assertTrue(is_prime(2))
7         self.assertTrue(is_prime(3))
8     def test_seive(self):
9         # Composite numbers are not prime
10        for q in range(2,100):
11            for b in range(2,100):
12                self.assertFalse(is_prime(q*b))
13
```

Note: `unittest` is most naturally used from the command line. Some examples will seem a bit clumsy because we are running them in Python instead.

`unittest` module:

<https://docs.python.org/3/library/unittest.html>

Python unittest module

Supports nicely organized test suites for your program

Note: there are plenty of other testing suites out there

```
1 def is_prime(x):
```

Tests are encapsulated in a class that extends `unittest.TestCase`.

```
2     return True
```

Methods prefaced by `test_` will run automatically once we run the test suite.

```
3     return False
```

```
4     return True
```

```
1 class PrimeTest(unittest.TestCase):
```

```
2     def test_base(self):
```

```
3         self.assertFalse(is_prime(-1))
```

```
4         self.assertFalse(is_prime(0))
```

```
5         self.assertFalse(is_prime(1))
```

```
6         self.assertTrue(is_prime(2))
```

```
7         self.assertTrue(is_prime(3))
```

```
8     def test_seive(self):
```

```
9         # Composite numbers are not prime
```

```
10        for q in range(2,100):
```

```
11            for b in range(2,100):
```

```
12                self.assertFalse(is_prime(q*b))
```

Note: a collection of tests is typically called a **test suite**. `unittest` uses this term to refer to a collection of `TestCase` objects (or a collection of objects that inherit from `TestCase`).

Python unittest module

```
1 class PrimeTest(unittest.TestCase):
2     def test_base(self):
3         self.assertFalse(is_prime(-1))
4         self.assertFalse(is_prime(0))
5         self.assertFalse(is_prime(1))
6         self.assertTrue(is_prime(2))
7         self.assertTrue(is_prime(3))
8     def test_seive(self):
9         # Composite numbers are not prime
10        for q in range(2,100):
11            for b in range(2,100):
12                self.assertFalse(is_prime(q*b))
13
14 prime_suite = unittest.defaultTestLoader.loadTestsFromTestCase(PrimeTest)
15 unittest.TextTestRunner().run(prime_suite)
```

Initializes an instance of `PrimeTest` and sets some of its attributes for us.

The `unittest.TextTestRunner` runs all the tests in our `PrimeTest` object.

Reminder: only methods prefaced by `test_` will be run as part of the test!

Python unittest module

```
14 prime_suite = unittest.defaultTestLoader.loadTestsFromTestCase(PrimeTest)
15 unittest.TextTestRunner().run(prime_suite)
```

```
.F
```

```
=====
FAIL: test_seive (__main__.PrimeTest)
-----
```

```
Traceback (most recent call last):
```

```
  File "<ipython-input-5-2e2a707dd63a>", line 12, in test_seive
```

```
    self.assertFalse(is_prime(q*b))
```

```
AssertionError: True is not false
```

```
-----
Ran 2 tests in 0.004s
```

```
FAILED (failures=1)
```

```
<unittest.runner.TextTestResult run=2 errors=0 failures=1>
```

If one or more tests fail, `unittest` will raise an error, and tell you which test(s) failed.

The results would also be stored in a `TextTestResult` object, if we had chosen to assign the output.

Python unittest module

Let's correct the error.


```
1 def is_prime(x):
2     if n <= 1:
3         return False
4     elif n==2:
5         return True
6     else:
7         ulim = math.ceil(math.sqrt(x))
8         for k in range(2,ulim):
9             if n%k==0:
10                return False
11            return True
```

```
1 def is_prime(n):
2     if n <= 1:
3         return False
4     elif n==2:
5         return True
6     else:
7         ulim = math.ceil(math.sqrt(n))
8         for k in range(2,ulim+1):
9             if n%k==0:
10                return False
11            return True
```

Python unittest module

```
1 def is_prime(n):
2     if n <= 1:
3         return False
4     elif n==2:
5         return True
6     else:
7         ulim = math.ceil(math.sqrt(n))
8         for k in range(2,ulim+1):
9             if n%k==0:
10                return False
11            return True
12
13 prime_suite = unittest.defaultTestLoader.loadTestsFromTestCase(PrimeTest)
14 unittest.TextTestRunner().run(prime_suite)
```

Using the same set of tests as before,
all defined in the `PrimeTest` object.



```
..
```

```
-----  
Ran 2 tests in 0.029s
```

```
OK
```

```
<unittest.runner.TextTestResult run=2 errors=0 failures=0>
```

Python unittest module

This function operates on files (and creates new files). So to test it, we need our test suite to create files for testing and check that the new files are as expected.

```
1 def file2upper(infile, outfile):
2     '''Takes a file infile, and copies it to
3     file outfile, but with all words in upper-case.'''
4     if type(infile) != str:
5         raise TypeError('Input file name must be a string.')
6     if type(outfile) != str:
7         raise TypeError('Output file name must be a string.')
8     with open(infile, 'r') as infh:
9         with open(outfile, 'w') as outfh:
10            for line in infh:
11                outfh.write(line.upper())
```

Often, it is useful to set up some files or objects before running our tests. This can be done using the `setUp` and `tearDown` methods.

The `setUp` method is called **before** each test. Here, our setup involves creating a directory and moving into it. This provides a “sandbox” for us to operate in where we won’t touch important files elsewhere.

The `tearDown` method is called **after** each test. Here, our tear down just requires that we delete the files that we created in the test directory and then delete the test directory.

```
1 class UpperTest(unittest.TestCase):
2     '''Test that file2upper works properly.'''
3     testdir='testdir' # Name of the test directory
4     testtext='The Quick Brown Fox Jumps Over the Lazy Dog.'
5     infile='in.txt' # We'll always process this file...
6     outfile='out.txt' # and write results to this file.
7
8
9
10
11
12
13
14
15     def setUp(self):
16         '''Create a test directory and create a few
17         files that we will work with in the test cases.'''
18         try:
19             os.mkdir(self.testdir) # Create a test dir...
20         except FileExistsError:
21             pass # foo already exists as a directory.
22         os.chdir(self.testdir)
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```
24 def test_empty(self):
25     with open( self.infile, 'w') as f:
26         pass # Results in an empty file.
27     file2upper(self.infile, self.outfile)
28     with open(self.outfile, 'r') as f:
29         for line in f:
30             self.assertTrue(line.isupper())
31 def test_lower(self):
32     with open(self.infile, 'w') as f:
33         f.write(self.testtext.lower())
34     file2upper(self.infile, self.outfile)
35     with open(self.outfile, 'r') as f:
36         for line in f:
37             self.assertTrue(line.isupper())
38 def test_mixed(self):
39     with open(self.infile, 'w') as f:
40         f.write(self.testtext)
41     file2upper(self.infile, self.outfile)
42     with open(self.outfile, 'r') as f:
43         for line in f:
44             self.assertTrue(line.isupper())
45 def test_upper(self):
46     with open(self.infile, 'w') as f:
47         f.write(self.testtext.upper())
48     file2upper(self.infile, self.outfile)
49     with open(self.outfile, 'r') as f:
50         for line in f:
51             self.assertTrue(line.isupper())
```

Reminder: the pattern is `setUp`, run a test, then `tearDown`.

The `setUp/tearDown` pattern ensures that each of these tests takes place in an otherwise empty, clean directory.

`file2upper` is a fairly simple function, so this `setUp/tearDown` framework isn't particularly necessary, but it should be clear that for functions or objects that do more complicated things, it can be a very useful. For example, if we were writing tests for our `Time` object, the `setUp/tearDown` methods would enable us to create a new `Time` object for each test without having to repeat the same few lines of code everywhere.

Python unittest module

```
1 upper_suite = unittest.defaultTestLoader.loadTestsFromTestCase(UpperTest)
2 unittest.TextTestRunner().run(upper_suite)
```

```
....
```

```
-----
Ran 4 tests in 0.020s
```

```
OK
```

```
<unittest.runner.TextTestResult run=4 errors=0 failures=0>
```

Parting note: the `unittest` module supports a whole lot of additional functionality and control over tests, but most of them are going to be beyond your needs unless you expect to be a software engineer. The module is useful to us as data scientists primarily in that it provides a (comparatively) clean way to encapsulate your testing code.