# STATS 507
# Data Analysis in Python

## Lecture 21: 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 $\mathbf{n^2+n+1}$

| 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)** |

| 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 |

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

# 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.

```
s = 0
for x in t:
    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.

```
seqlens = np.arange(1e5,1e6,1e4)
runtimes = np.zeros(len(seqlens))
for n in range(len(seqlens)):
    slen = int(seqlens[n])
    seq = np.random.random(size=slen)
    tstart = time.time()
    sum(seq)
    tend = time.time()
    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.**

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

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

```
seqlens = np.arange(1e5,1e6,1e4)
runtimes = np.zeros(len(seqlens))
for n in range(len(seqlens)):
    slen = int(seqlens[n])
    seq = np.random.random(size=slen)
    tstart = time.time()
    sum(seq)
    tend = time.time()
    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.**

```
seqlens = np.arange(1e5,1e6,1e4)
runtimes = np.zeros(len(seqlens))
for n in range(len(seqlens)):
    slen = int(seqlens[n])
    seq = np.random.random(size=slen)
    tstart = time.time()
    sum(seq)
    tend = time.time()
    runtimes[n] = tend-tstart
```

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

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

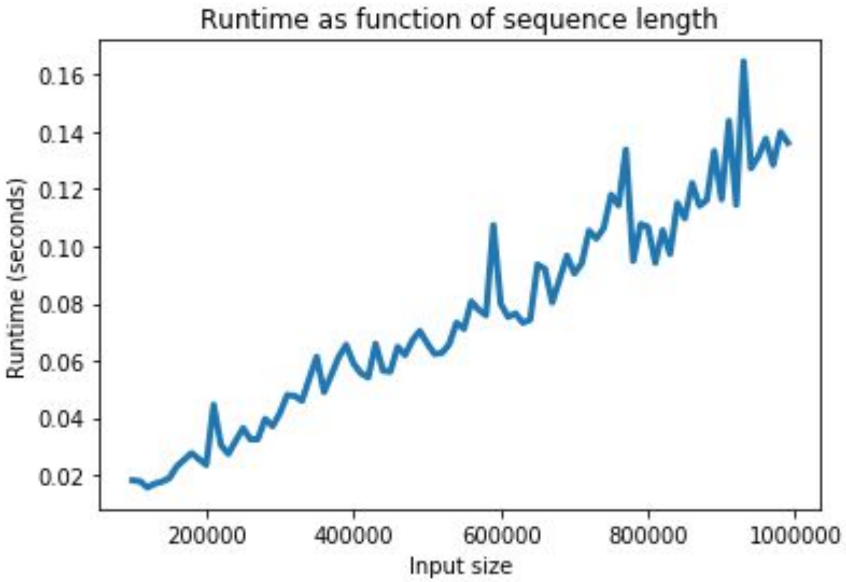
...and time how long it takes to sum them up.

# 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.**

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

```
seqlens = np.arange(1e5,1e6,1e4)
runtimes = np.zeros(len(seqlens))
for n in range(len(seqlens)):
    slen = int(seqlens[n])
    seq = np.random.random(size=slen)
    tstart = time.time()
    sum(seq)
    tend = time.time()
    runtimes[n] = tend-tstart
```

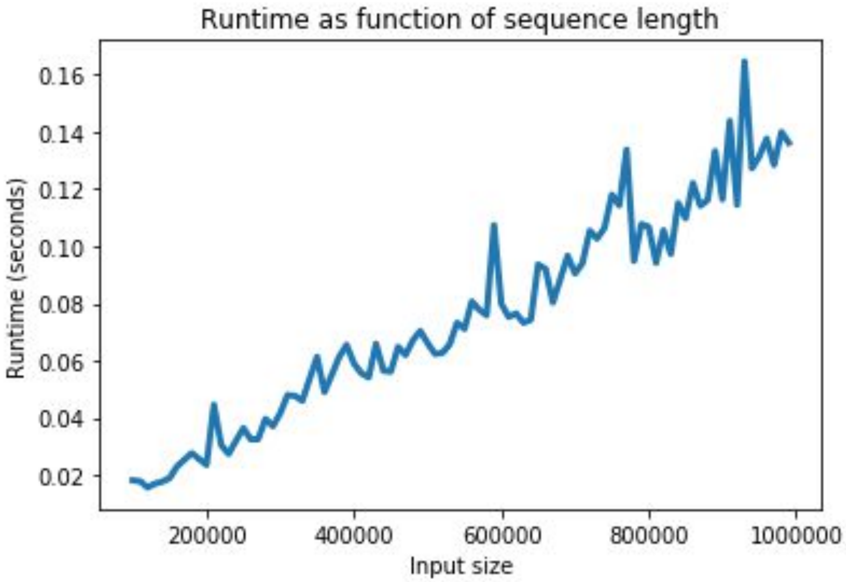


# Constant-order work on each element of a list

**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.**

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

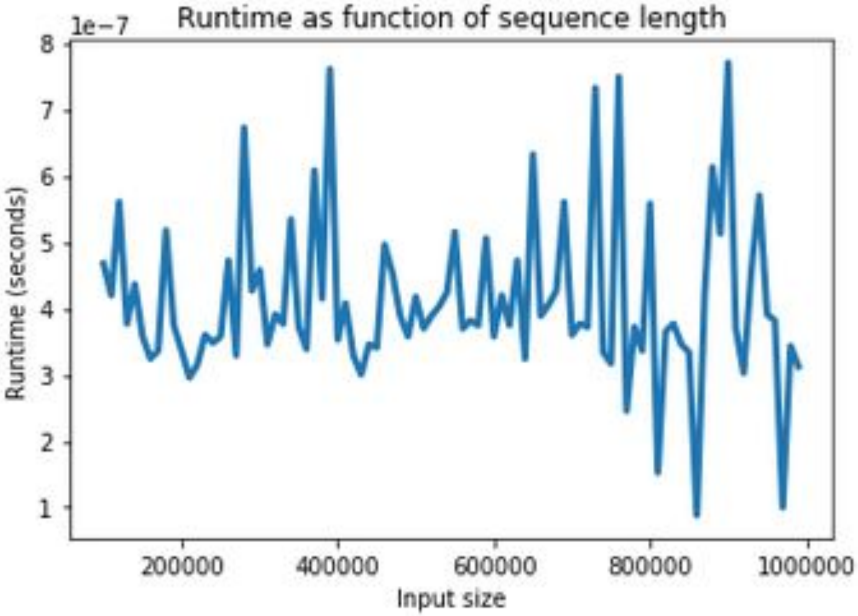
```python
seqlens = np.arange(1e5,1e6,1e4)
runtimes = np.zeros(len(seqlens))
for n in range(len(seqlens)):
    slen = int(seqlens[n])
    seq = np.random.random(size=slen)
    tstart = time.time()
    sum(seq)
    tend = time.time()
    runtimes[n] = tend-tstart
```



# 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.

```python
seqlens = np.arange(1e5,1e6,1e4)
ntrials=100
runtimes = np.zeros((len(seqlens),ntrials))
for n in range(len(seqlens)):
    slen = int(seqlens[n])
    seq = list(np.random.random(size=slen))
    for m in range(ntrials):
        tstart = time.time()
        len(seq)
        tend = time.time()
        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

```python
def argmax(t):
    if len(t)==0: # Handle a weird edge case.
        return (None,float('-inf'))
    (i,m)=(0,t[0])
    for j in range(1,len(t)):
        if t[j] > m:
            (i,m) = (j,t[j])
    return (i,m)
def naive_sort(t):
    n=len(t)
    for k in range(1,len(t)):
        # Find the largest element and its index
        (i,m) = argmax(t[:(n-k+1)])
        # Swap the maximum with the last element
        (t[i],t[n-k])=(t[n-k],m)
    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.

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)

But let’s first look at some suboptimal sorting algorithms

```
def argmax(t):
    if len(t)==0: # Handle a weird edge case.
        return (None,float('-inf'))
    (i,m)=(0,t[0])
    for j in range(1,len(t)):
        if t[j] > m:
            (i,m) = (j,t[j])
    return (i,m)
def naive_sort(t):
    n=len(t)
    for k in range(1,len(t)):
        # Find the largest element and its index
        (i,m) = argmax(t[:(n-k+1)])
        # Swap the maximum with the last element
        (t[i],t[n-k])=(t[n-k],m)
    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+...+n = $O(n^2)$.

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

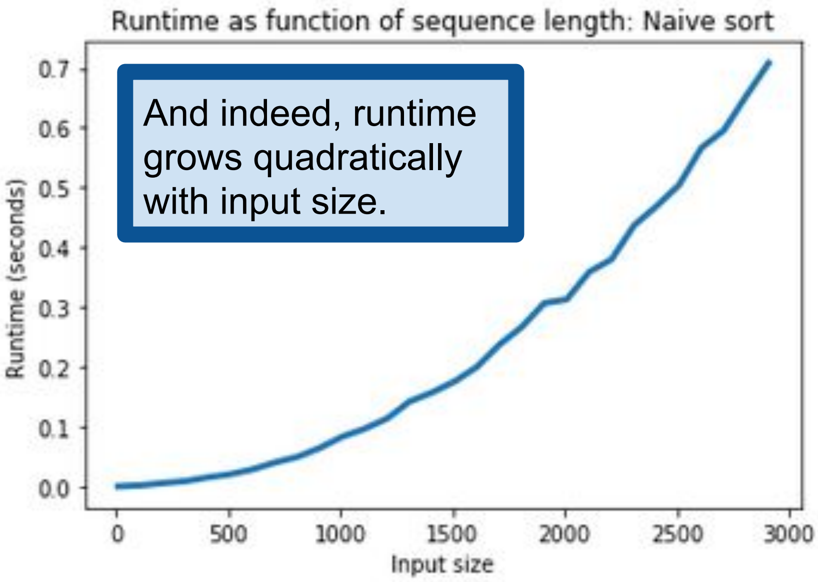
# Sorting

**Problem:** given a list, sort it in ascending order

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

But let’s first look at some suboptimal sorting algorithms

```python
def argmax(t):
    if len(t)==0: # Handle a weird edge case.
        return (None,float('-inf'))
    (i,m)=(0,t[0])
    for j in range(1,len(t)):
        if t[j] > m:
            (i,m) = (j,t[j])
    return (i,m)
def naive_sort(t):
    n=len(t)
    for k in range(1,len(t)):
        # Find the largest element and its index
        (i,m) = argmax(t[:(n-k+1)])
        # Swap the maximum with the last element
        (t[i],t[n-k])=(t[n-k],m)
    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)

```python
def quicksort(t):
    if len(t) <= 1:
        return t
    (less,mid,more) = (list(),list(),list())
    pivot = t[0]
    mid.append(t[0])
    for i in range(1,len(t)):
        if t[i] == pivot:
            mid.append(t[i])
        elif t[i] < pivot:
            less.append(t[i])
        else: # t[i] > pivot
            more.append(t[i])
    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, an recurse on the less-than and greater-than lists. This pattern should look familiar from your binary search problem in HW2.

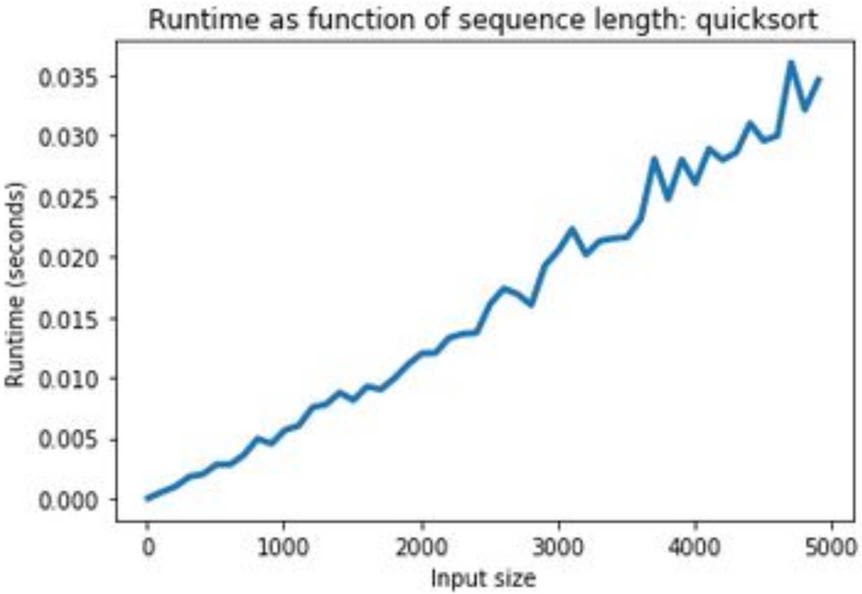
This recursion is the important part. `less` and `more` contain the elements less than and greater than the pivot, but they may not yet be sorted.

# Sorting

**Problem:** given a list, sort it in ascending order

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

```python
def quicksort(t):
    if len(t) <= 1:
        return t
    (less,mid,more) = (list(),list(),list())
    pivot = t[0]
    mid.append(t[0])
    for i in range(1,len(t)):
        if t[i] == pivot:
            mid.append(t[i])
        elif t[i] < pivot:
            less.append(t[i])
        else: # t[i] > pivot
            more.append(t[i])
    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)

```python
def quicksort(t):
    if len(t) <= 1:
        return t
    (less,mid,more) = (list(),list(),list())
    pivot = t[0]
    mid.append(t[0])
    for i in range(1,len(t)):
        if t[i] == pivot:
            mid.append(t[i])
        elif t[i] < pivot:
            less.append(t[i])
        else: # t[i] > pivot
            more.append(t[i])
    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*T(n/2) + C for some constant C, and T(n) = n log n is such a function.

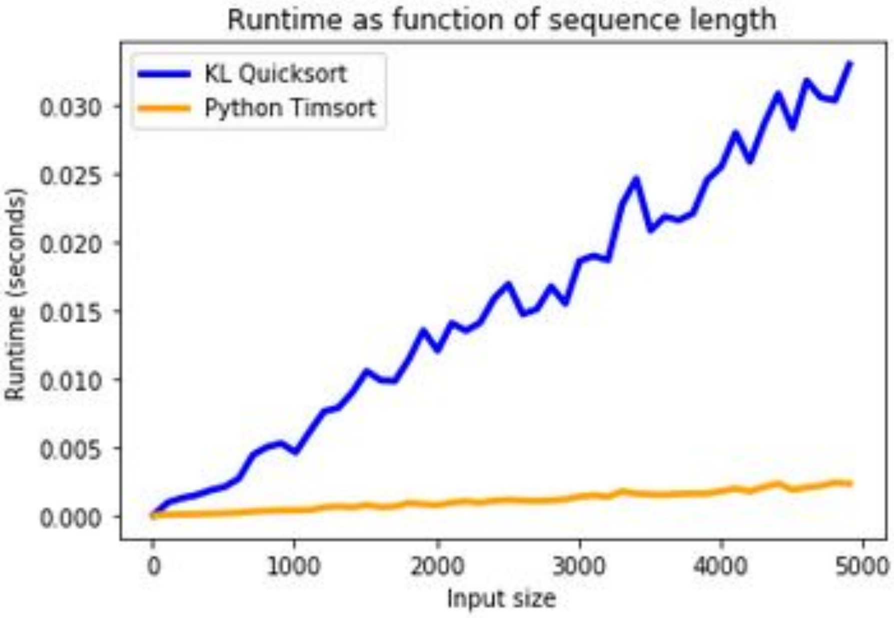
# Aside: the house always wins, Python edition

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!

```
seqlens = np.arange(1e1,5*1e3,1e2)
ntrials=10
myruntimes = np.zeros((len(seqlens),ntrials))
pythonruntimes = np.zeros((len(seqlens),ntrials))
for n in range(len(seqlens)):
    slen = int(seqlens[n])
    seq = list(np.random.random(size=slen))
    for m in range(ntrials):
        tstart = time.time()
        quicksort(seq)
        tend = time.time()
        myruntimes[n,m] = tend-tstart
        tstart = time.time()
        sorted(seq)
        tend = time.time()
        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.

```python
import time
from random import randint
listlen = 1000000
list_of_numbers = listlen*[0]
dict_of_numbers = dict()
for i in range(listlen):
    n = randint(1000000,9999999)
    list_of_numbers[i] = n
    dict_of_numbers[n] = 1
```

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.

```python
start_time = time.time()
8675309 in list_of_numbers
time.time() - start_time
```

```
0.027842044830322266
```

```python
start_time = time.time()
8675309 in dict_of_numbers
time.time() - start_time
```

```
0.00012326240539550078
```

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

# `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

```
import cProfile
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.

```
import cProfile
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`

```python
import cProfile
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`

```python
import cProfile
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`

```
import cProfile
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`

```python
import cProfile
cProfile.run('8675309 in list_of_numbers')
```

```
         3 function calls in 0.026 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall  ilename: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

```
import cProfile
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?...

**fibonacci.py**

```python
def naive_fibo(n):
    if n < 0:
        raise ValueError('Negative Fibonacci number?')
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return naive_fibo(n-1) + naive_fibo(n-2)

known = {0:0, 1:1}
def fibo(n):
    if n in known:
        return known[n]
    else:
        f = fibo(n-1) + fibo(n-2)
        known[n] = f
        return(f)
```

```python
import fibonacci
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}
```

```python
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}
```

```python
import fibonacci
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!

```python
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}
```

```python
import fibonacci
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}
```

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

```python
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.

**ols_expt.py**

```
import numpy as np
from sklearn import linear_model
def generate_data(n, beta, Xfile, Yfile):
    p = beta.size # beta is a numpy vector.
    # Each data point is drawn indep'ly with
    # independent Laplace-distributed entries
    x = np.random.laplace(0, 1, size=(n,p))
    # Observed data is beta^T x + normal noise.
    noise = np.random.normal(0, 100, size=n)
    y = np.matmul(beta,x.T) + noise
    np.savetxt(Xfile, x)
    np.savetxt(Yfile, y)
def load_data(Xfile,Yfile):
    x = np.loadtxt(Xfile)
    y = np.loadtxt(Yfile)
    return (x,y)
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)
```

Reminder of what our experiment does

```
    return (x,y)
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)
```

```
import cProfile
from ols_expt import *
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  tottime  percall  cumtime  percall filename:lineno(function)
       24    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:997(_handle_fromlist)
        1    0.000    0.0
        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)
```

```
import cProfile
from ols_expt import *
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  tottime  percall  cumtime  percall filename:lineno(function)
    24    0.(                                        otstrap>:997(_handle_fromlist)
     1    0.(
                                                     ind_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)
    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?

```python
def is_prime(x):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(x))
        for k in range(2,ulim):
            if n%k==0:
                return False
        return True
```

```python
is_prime(2)
```

True

```python
is_prime(3)
```

True

```python
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

```python
def is_prime(x):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(x))
        for k in range(2,ulim):
            if n%k==0:
                return False
        return True
```

```python
class PrimeTest(unittest.TestCase):
    def test_base(self):
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
    def test_seive(self):
        # Composite numbers are not prime
        for q in range(2,100):
            for b in range(2,100):
                self.assertFalse(is_prime(q*b))
```

`unittest` module:
[https://docs.python.org/3/library/unittest.html](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

```
def is_prime(x):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(x))
        for k in range(2,ulim):
            if n%k==0:
                return False
        return True
```

```
class PrimeTest(unittest.TestCase):
    def test_base(self):
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
    def test_seive(self):
        # Composite numbers are not prime
        for q in range(2,100):
            for b in range(2,100):
                self.assertFalse(is_prime(q*b))
```

**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](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

```python
def is_prime(x):
        return True

                return False
        return True
```

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

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

```python
class PrimeTest(unittest.TestCase):
    def test_base(self):
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
    def test_seive(self):
        # Composite numbers are not prime
        for q in range(2,100):
            for b in range(2,100):
                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

```python
class PrimeTest(unittest.TestCase):
    def test_base(self):
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
    def test_seive(self):
        # Composite numbers are not prime
        for q in range(2,100):
            for b in range(2,100):
                self.assertFalse(is_prime(q*b))

prime_suite = unittest.defaultTestLoader.loadTestsFromTestCase(PrimeTest)
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

```
prime_suite = unittest.defaultTestLoader.loadTestsFromTestCase(PrimeTest)
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.

```python
def is_prime(x):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(x))
        for k in range(2,ulim):
            if n%k==0:
                return False
        return True
```

```python
def is_prime(n):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(n))
        for k in range(2,ulim+1):
            if n%k==0:
                return False
        return True
```

# Python `unittest` module

```python
def is_prime(n):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(n))
        for k in range(2,ulim+1):
            if n%k==0:
                return False
        return True

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

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

OK

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

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

# 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.

```python
def file2upper(infile, outfile):
    '''Takes a file infile, and copies it to
    file outfile, but with all words in upper-case.'''
    if type(infile) != str:
        raise TypeError('Input file name must be a string.')
    if type(outfile) != str:
        raise TypeError('Output file name must be a string.')
    with open(infile, 'r') as infh:
        with open(outfile, 'w') as outfh:
            for line in infh:
                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.

```python
class UpperTest(unittest.TestCase):
    '''Test that file2upper works properly.'''
    testdir='testdir' # Name of the test directory
    testtext='The Quick Brown Fox Jumps Over the Lazy Dog.'
    infile='in.txt' # We'll always process this file...
    outfile='out.txt' # and write results to this file.

    def setUp(self):
        '''Create a test directory and create a few
        files that we will work with in the test cases.'''
        try:
            os.mkdir(self.testdir) # Create a test dir...
        except FileExistsError:
            pass # foo already exists as a directory.
        os.chdir(self.testdir)

    def tearDown(self):
        '''Delete the test directory.'''
        os.remove(self.infile)
        os.remove(self.outfile)
        os.chdir('..') # Up a level out of tesdir.
        os.rmdir(self.testdir)
```

```python
    def test_empty(self):
        with open( self.infile, 'w') as f:
            pass # Results in an empty file.
        file2upper(self.infile, self.outfile)
        with open(self.outfile, 'r') as f:
            for line in f:
                self.assertTrue(line.isupper())
    def test_lower(self):
        with open(self.infile, 'w') as f:
            f.write(self.testtext.lower())
        file2upper(self.infile, self.outfile)
        with open(self.outfile, 'r') as f:
            for line in f:
                self.assertTrue(line.isupper())
    def test_mixed(self):
        with open(self.infile, 'w') as f:
            f.write(self.testtext)
        file2upper(self.infile, self.outfile)
        with open(self.outfile, 'r') as f:
            for line in f:
                self.assertTrue(line.isupper())
    def test_upper(self):
        with open(self.infile, 'w') as f:
            f.write(self.testtext.upper())
        file2upper(self.infile, self.outfile)
        with open(self.outfile, 'r') as f:
            for line in f:
                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

```
upper_suite = unittest.defaultTestLoader.loadTestsFromTestCase(UpperTest)
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.