Last time... Recursion

Recursion

```python
def f(n):
    if n == 0:
        return 0
    else:
        return 1 + f(n - 1)
```

```python
def fib(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

Recursion tree

Mutual recursion

```python
def even(n):
    if n == 0:
        return True
    else:
        return odd(n - 1)
```

```python
def odd(n):
    if n == 0:
        return False
    else:
        return even(n - 1)
```

Mutual recursion
Lecture Overview

• Sorting
• List comprehension
• Data visualization

Disclaimer: Much of the material and slides for this lecture were borrowed from
— R. Anderson, M. Ernst and B. Howe in University of Washington CSE 140
— C. van Loan in Cornell University CS 1110 Introduction to Computing
Lecture Overview

• Sorting
• List comprehension
• Data visualization
Sorting

hamlet = "to be or not to be that is the question whether tis nobler in the mind to suffer".split()

print("hamlet:", hamlet)

print("sorted(hamlet):", sorted(hamlet))
print("hamlet:", hamlet)

print("hamlet.sort():", hamlet.sort())
print("hamlet:", hamlet)

- Lists are **mutable** – they can be changed
  - including by functions
Sorting

```python
hamlet: ['to', 'be', 'or', 'not', 'to', 'be', 'that', 'is', 'the', 'question', 'whether', 'tis', 'nobler', 'in', 'the', 'mind', 'to', 'suffer']

sorted(hamlet): ['be', 'be', 'in', 'is', 'mind', 'nobler', 'not', 'or', 'question', 'suffer', 'that', 'the', 'the', 'tis', 'to', 'to', 'to', 'whether']

hamlet: ['to', 'be', 'or', 'not', 'to', 'be', 'that', 'is', 'the', 'question', 'whether', 'tis', 'nobler', 'in', 'the', 'mind', 'to', 'suffer']

hamlet.sort(): None

hamlet: ['be', 'be', 'in', 'is', 'mind', 'nobler', 'not', 'or', 'question', 'suffer', 'that', 'the', 'tis', 'to', 'to', 'to', 'whether']
```
Customizing the sort order

**Goal:** sort a list of names *by last name*

```python
names = ['Isaac Newton', 'Albert Einstein', 'Niels Bohr', 'Marie Curie', 'Charles Darwin', 'Louis Pasteur', 'Galileo Galilei', 'Margaret Mead']

print("names:", names)
```

This does NOT work:

```python
print("sorted(names):", sorted(names))
```

When sorting, how should we compare these names?

"Niels Bohr"
"Charles Darwin"

```python
sorted(names): ['Albert Einstein', 'Charles Darwin', 'Galileo Galilei', 'Isaac Newton', 'Louis Pasteur', 'Margaret Mead', 'Marie Curie', 'Niels Bohr']
```
Sort key

A sort key is a different value that you use to sort a list, instead of the actual values in the list

def last_name(str):
    return str.split(" ")[1]

print('last_name("Isaac Newton"):',
last_name("Isaac Newton"))

Two ways to use a sort key:
  1. Create a new list containing the sort key, and then sort it
  2. Pass a key function to the sorted function
1. Use a sort key to create a new list

Create a different list that contains the sort key, sort it, then extract the relevant part:

```python
names = ['Isaac Newton', 'Fred Newton', 'Niels Bohr']
keyed_names = []
for name in names:
    last_name, full_name = name.split(' ', 1)
    keyed_names.append([last_name, name])

keyed_names: [['Newton', 'Isaac Newton'], ['Newton', 'Fred Newton'], ['Bohr', 'Niels Bohr']]
sorted(keyed_names): [['Bohr', 'Niels Bohr'], ['Newton', 'Fred Newton'], ['Newton', 'Isaac Newton']]
sorted(keyed_names, reverse = True): [['Newton', 'Isaac Newton'], ['Newton', 'Fred Newton'], ['Bohr', 'Niels Bohr']]
```

Take a look at the list you created, it can now be sorted:

```python
print("keyed_names:", keyed_names)
print("sorted(keyed_names):", sorted(keyed_names))
print("sorted(keyed_names, reverse = True):")
print(sorted(keyed_names, reverse = True))
```

(This works because Python compares two elements that are lists elementwise.)

```python
sorted_keyed_names = sorted(keyed_names, reverse = True)
```

sorted_names: ['Isaac Newton', 'Fred Newton', 'Niels Bohr']

2) Sort the list new list.

```python
sorted_names = []
for keyed_name in sorted_keyed_names:
    sorted_names.append(keyed_name[1])
print("sorted_names:", sorted_names)
```

3) Extract the relevant part.
2. Use a sort key as the **key** argument

Supply the **key** argument to the **sorted** function or the **sort** function

```python
def last_name(str):
    return str.split(" ")[1]

names = ["Isaac Newton", "Fred Newton", "Niels Bohr"]

print("sorted(names, key = last_name)":
print(sorted(names, key = last_name))

print("sorted(names, key = last_name, reverse = True)":
print(sorted(names, key = last_name, reverse = True))

print(sorted(names, key = len))

def last_name_len(name):
    return len(last_name(name))

print(sorted(names, key = last_name_len))
```

```
sorted(names, key = last_name): ['Niels Bohr', 'Isaac Newton', 'Fred Newton']
sorted(names, key = last_name, reverse = True): ['Isaac Newton', 'Fred Newton', 'Niels Bohr']
[
Niels Bohr', 'Fred Newton', 'Isaac Newton']
['Niels Bohr', 'Isaac Newton', 'Fred Newton']
```
itemgetter is a function that returns a function.

import operator

print(operator.itemgetter(2, 7, 9, 10)("dumbstricken"))
operator.itemgetter(2, 5, 7, 9)("homesickness")
operator.itemgetter(2, 7, 9, 10)("pumpernickel")
operator.itemgetter(2, 3, 6, 7)("seminaked")
operator.itemgetter(1, 2, 4, 5)("smirker")

operator.itemgetter(9, 7, 6, 1)("beatnikism")
operator.itemgetter(14, 13, 5, 1)("Gedankenexperiment")
operator.itemgetter(12, 10, 9, 5)("mountebankism")

All: ('m', 'i', 'k', 'e')
Using `itemgetter`

```python
from operator import itemgetter

student_score = ('Robert', 8)
itemgetter(0)(student_score) ⇒ “Robert”
itemgetter(1)(student_score) ⇒ 8

student_scores = [('Robert', 8), ('Alice', 9), ('Tina', 7)]

• Sort the list by name:
  `sorted(student_scores, key=itemgetter(0))`

• Sort the list by score
  `sorted(student_scores, key=itemgetter(1))`
```
Two Ways to Import `itemgetter`

```python
from operator import itemgetter
student_score = ('Robert', 8)
itemgetter(0)(student_score)  # ⇒ “Robert”
itemgetter(1)(student_score)  # ⇒ 8

or

import operator
student_score = ('Robert', 8)
operator.itemgetter(0)(student_score)  # ⇒ “Robert”
operator.itemgetter(1)(student_score)  # ⇒ 8
```
Sorting based on two criteria

Two approaches:
- Approach #1: Use an `itemgetter` with two arguments
- Approach #2: Sort twice (most important sort last)

```
student_scores = [('Robert', 8), ('Alice', 9), ('Tina', 10), ('James', 8)]
```

**Goal**: sort based on score; if there is a tie within score, sort by name

Approach #1:
```
sorted(student_scores, key=itemgetter(1,0))
```

Approach #2:
```
sorted_by_name = sorted(student_scores, key=itemgetter(0))
sorted_by_score = sorted(sorted_by_name, key=itemgetter(1))
```
Sort on most important criteria LAST

• Sorted by score (ascending), when there is a tie on score, sort using name

```python
from operator import itemgetter

student_scores = [('Robert', 8), ('Alice', 9), ('Tina', 10), ('James', 8)]

sorted_by_name = sorted(student_scores, key=itemgetter(0))
>>> sorted_by_name
[('Alice', 9), ('James', 8), ('Robert', 8), ('Tina', 10)]

sorted_by_score = sorted(sorted_by_name, key=itemgetter(1))
>>> sorted_by_score
[('James', 8), ('Robert', 8), ('Alice', 9), ('Tina', 10)]
```
More sorting based on two criteria

If you want to sort different criteria in different directions, you must use multiple calls to `sort` or `sorted`

```python
student_scores = [('Robert', 8), ('Alice', 9), ('Tina', 10), ('James', 8)]
```

**Goal:** sort score from highest to lowest; if there is a tie within score, sort by name alphabetically (= lowest to highest)

```python
sorted_by_name = sorted(student_scores, key=itemgetter(0) )
sorted_by_hi_score = sorted(sorted_by_name, key=itemgetter(1), reverse=True)
```
Sorting: strings vs. numbers

• Sorting the powers of 5:

```python
>>> sorted([125, 5, 3125, 625, 25])
[5, 25, 125, 625, 3125]
```

```python
>>> sorted(['125', '5', '3125', '625', '25'])
['125', '25', '3125', '5', '625']
```
Sorting

from BBC Documentary: The Secret Rules of Modern Living Algorithms
Different sorting algorithms

3.1 Simple sorts
   3.1.1 Insertion sort
   3.1.2 Selection sort

3.2 Efficient sorts
   3.2.1 Merge sort
   3.2.2 Heapsort
   3.2.3 Quicksort

3.3 Bubble sort and variants
   3.3.1 Bubble sort
   3.3.2 Shell sort
   3.3.3 Comb sort

3.4 Distribution sort
   3.4.1 Counting sort
   3.4.2 Bucket sort
   3.4.3 Radix sort

A sorting algorithm is an algorithm that puts elements of a list in a certain order which require input data to be in sorted lists; it is also often useful for other problems. The following are required:

1. The output is in nondecreasing order (each element is no smaller than its predecessor).
2. The output is a permutation (reordering) of the input.

Further, the data is often taken to be in an array, which allows random access and fast access to the middle elements. Since the dawn of computing, the sorting problem has attracted a great deal of research, resulting in a wide variety of algorithms. A comparison sorting algorithm has a worst-case running time of \(O(n \log n)\).
Bubble sort

• It repeatedly steps through the list to be sorted,
• compares each pair of adjacent items and swaps them if they are in the wrong order.
• The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
• The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.
Bubble sort

def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
Insertion sort

• Idea:

• maintain a sorted sublist in the lower positions of the list.
• Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger.

Done!
def insertionSort(alist):
    for index in range(1, len(alist)):
        currentvalue = alist[index]
        position = index

        while position > 0 and alist[position - 1] > currentvalue:
            alist[position] = alist[position - 1]
            position = position - 1

        alist[position] = currentvalue

alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertionSort(alist)
print(alist)
Insertion sort

https://www.youtube.com/watch?v=ROalU379I3U
Mergesort

• Merge sort is a prototypical divide-and-conquer algorithm.

• It was invented in 1945, by John von Neumann.

• Like many divide-and-conquer algorithms it is most easily described recursively.
  1. If the list is of length 0 or 1, it is already sorted.
  2. If the list has more than one element, split the list into two lists, and use mergesort to sort each of them.
  3. Merge the results.
def merge(left, right):
    result = []
    (i, j) = (0, 0)
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i = i + 1
        else:
            result.append(right[j])
            j = j + 1
    while i < len(left):
        result.append(left[i])
        i = i + 1
    while j < len(right):
        result.append(right[j])
        j = j + 1
    return result
def mergeSort(L):
    if len(L)<2:
        return L[:]
    else:
        middle = len(L)//2
        left = mergeSort(L[:middle])
        right = mergeSort(L[middle:]
        return merge(left, right)

a = mergeSort([2,1,3,4,5,-1,8,6,7])
### Sorting Algorithm Animations

**Problem Size:** 20 · 30 · 40 · 50  
**Magnification:** 1x · 2x · 3x

**Algorithm:** Insertion · Selection · Bubble · Shell · Merge · Heap · Quick · Quick3

**Initial Condition:** Random · Nearly Sorted · Reversed · Few Unique

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Random</th>
<th>Nearly Sorted</th>
<th>Reversed</th>
<th>Few Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td><img src="image" alt="Insertion Random" /></td>
<td><img src="image" alt="Insertion Nearly Sorted" /></td>
<td><img src="image" alt="Insertion Reversed" /></td>
<td><img src="image" alt="Insertion Few Unique" /></td>
</tr>
<tr>
<td>Selection</td>
<td><img src="image" alt="Selection Random" /></td>
<td><img src="image" alt="Selection Nearly Sorted" /></td>
<td><img src="image" alt="Selection Reversed" /></td>
<td><img src="image" alt="Selection Few Unique" /></td>
</tr>
<tr>
<td>Bubble</td>
<td><img src="image" alt="Bubble Random" /></td>
<td><img src="image" alt="Bubble Nearly Sorted" /></td>
<td><img src="image" alt="Bubble Reversed" /></td>
<td><img src="image" alt="Bubble Few Unique" /></td>
</tr>
<tr>
<td>Shell</td>
<td><img src="image" alt="Shell Random" /></td>
<td><img src="image" alt="Shell Nearly Sorted" /></td>
<td><img src="image" alt="Shell Reversed" /></td>
<td><img src="image" alt="Shell Few Unique" /></td>
</tr>
<tr>
<td>Merge</td>
<td><img src="image" alt="Merge Random" /></td>
<td><img src="image" alt="Merge Nearly Sorted" /></td>
<td><img src="image" alt="Merge Reversed" /></td>
<td><img src="image" alt="Merge Few Unique" /></td>
</tr>
<tr>
<td>Heap</td>
<td><img src="image" alt="Heap Random" /></td>
<td><img src="image" alt="Heap Nearly Sorted" /></td>
<td><img src="image" alt="Heap Reversed" /></td>
<td><img src="image" alt="Heap Few Unique" /></td>
</tr>
<tr>
<td>Quick</td>
<td><img src="image" alt="Quick Random" /></td>
<td><img src="image" alt="Quick Nearly Sorted" /></td>
<td><img src="image" alt="Quick Reversed" /></td>
<td><img src="image" alt="Quick Few Unique" /></td>
</tr>
<tr>
<td>Quick3</td>
<td><img src="image" alt="Quick3 Random" /></td>
<td><img src="image" alt="Quick3 Nearly Sorted" /></td>
<td><img src="image" alt="Quick3 Reversed" /></td>
<td><img src="image" alt="Quick3 Few Unique" /></td>
</tr>
</tbody>
</table>
Lecture Overview

• Sorting
• List comprehension
• Data visualization
Three Ways to Define a List

• Explicitly write out the whole thing:
  `squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`

• Write a loop to create it:
  `squares = []`
  `for i in range(11):`
    `squares.append(i*i)`

• Write a **list comprehension**:
  `squares = [i*i for i in range(11)]`

• A list comprehension is a concise description of a list
• A list comprehension is shorthand for a loop
Two ways to convert Centigrade to Fahrenheit

ctemps = [17.1, 22.3, 18.4, 19.1]

**With a loop:**

```python
ftemps = []
for c in ctemps:
    f = celsius_to_farenheit(c)
    ftemps.append(f)
```

**With a list comprehension:**

```python
ftemps = [celsius_to_farenheit(c) for c in ctemps]
```

The comprehension is usually shorter, more readable, and more efficient.
Syntax of a comprehension

\[
[(x,y) \text{ for } x \text{ in seq1} \text{ for } y \text{ in seq2} \text{ if } \text{sim}(x,y) > \text{threshold}]
\]

- **expression**
  - **for clause (required)** assigns value to the variable `x`
  - zero or more additional **for clauses**
  - something that can be iterated
- zero or more **if clauses**
Semantics of a comprehension

\[
[(x,y) \text{ for } x \text{ in } \text{seq1 for } y \text{ in } \text{seq2 if sim}(x,y) > \text{threshold}]
\]

result = []
for x in seq1:
    for y in seq2:
        if sim(x,y) > threshold:
            result.append( (x,y) )

... use result ...
Types of comprehensions

List

\[
[ i\times2 \text{ for } i \text{ in range}(3) ]
\]

Set

\{
 i\times2 \text{ for } i \text{ in range}(3)
\}

Dictionary

\{
 key: value \text{ for } item \text{ in } sequence \ldots
\}
\{
 i: i\times2 \text{ for } i \text{ in range}(3)
\}
Cubes of the first 10 natural numbers

Goal:
  Produce: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

With a loop:

cubes = []
for x in range(10):
    cubes.append(x**3)

With a list comprehension:

cubes = [x**3 for x in range(10)]
Powers of 2, $2^0$ through $2^{10}$

**Goal:** $[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]$

$[2**i for i in range(11)]$
Even elements of a list

**Goal:** Given an input list `nums`, produce a list of the even numbers in `nums`

```python
nums = [3, 1, 4, 1, 5, 9, 2, 6, 5]
⇒ [4, 2, 6]

[num for num in nums if num % 2 == 0]
```
Goal: A list of all possible dice rolls.

With a loop:
rolls = []
for r1 in range(1,7):
    for r2 in range(1,7):
        rolls.append((r1,r2))

With a list comprehension:
rolls = [ (r1,r2) for r1 in range(1,7) for r2 in range(1,7)]
All above-average 2-die rolls

**Goal:** Result list should be a list of 2-tuples:

\[
[(2, 6), (3, 5), (3, 6), (4, 4), (4, 5), (4, 6), (5, 3), (5, 4), (5, 5), (5, 6),
(6, 2), (6, 3), (6, 4), (6, 5), (6, 6)]
\]

\[
[(r1, r2) \text{ for } r1 \text{ in } [1,2,3,4,5,6] \\
  \text{ for } r2 \text{ in } [1,2,3,4,5,6] \\
  \text{ if } r1 + r2 > 7]
\]

OR

\[
[(r1, r2) \text{ for } r1 \text{ in } \text{range}(1, 7) \\
  \text{ for } r2 \text{ in } \text{range}(8-r1, 7)]
\]
Goal: A matrix were each element is the sum of it's row and column numbers.

With a loop:

```python
def make_matrix_with_loop():
    matrix = []
    for i in range(5):
        row = []
        for j in range(5):
            row.append(i+j)
        matrix.append(row)
    return matrix
```

With a list comprehension:

```python
def make_matrix_with_list_comprehension():
    matrix = [[i+j for j in range(5)] for i in range(5)]
    return matrix
```
Function $4x^2 - 4$

With a loop:

```python
num_list = []
for i in range(-10,11):
    num_list.append(4*i**2 - 4)
```

With a list comprehension:

```python
num_list = [4*i**2 - 4 for i in range(-10,11)]
```
Normalize a list

With a loop:

```python
num_list = [6, 4, 2, 8, 9, 10, 3, 2, 1, 3]
total = float(sum(num_list))
for i in range(len(num_list)):
    num_list[i] = num_list[i]/float(total)
```

With a list comprehension:

```python
num_list = [i/total for i in num_list]
```
Dictionary mapping integers to multiples under 20

**With a loop:**

```python
for n in range(1, 11):
    multiples_list = []
    for i in range(1, 21):
        if i%n == 0:
            multiples_list.append(i)
    multiples[n] = multiples_list
```

**With a dictionary comprehension:**

```python
multiples = {n: [i for i in range(1, 21) if i%n == 0] for n in range(1, 11)}
```

```python
{1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], 2: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20], 3: [3, 6, 9, 12, 15, 18], 4: [4, 8, 12, 16, 20], 5: [5, 10, 15, 20], 6: [6, 12, 18], 7: [7, 14], 8: [8, 16], 9: [9, 18], 10: [10, 20]}
```
A word of caution

List comprehensions are great, but they can get confusing. Error on the side of readability.

```python
nums = [n for n in range(100) if sum([int(j) for j in str(n)]) % 7 == 0]
```

```python
nums = []
for n in range(100):
    digit_sum = sum([int(j) for j in str(n)])
    if digit_sum % 7 == 0:
        nums.append(n)
```
A word of caution

List comprehensions are great, but they can get confusing. Error on the side of readability.

```python
def sum_digits(n):
    digit_list = [int(i) for i in str(n)]
    return sum(digit_list)
```

```python
nums = [n for n in range(100) if sum(digits([int(j) for j in str(n)])) % 7 == 0]
```
Ternary Assignment

A common pattern in python

```python
if x > threshold:
    flag = True
else:
    flag = False
Or
flag = False
if x > threshold:
    flag = True
```

46
Ternary Assignment

A common pattern in python

```python
if x > threshold:
    flag = True
else:
    flag = False
```

flag = True if x > threshold else False

Ternary Expression
Three elements
Ternary Assignment

flag = True if x > threshold else False

- Only works for single expressions as results.
- Only works for if and else (no elif)
Ternary Assignment

**Goal:** A list of 'odd' or 'even' if that index is odd or even.

```python
the_list = []
for i in range(16):
    if i%2 == 0:
        the_list.append('even')
    else:
        the_list.append('odd')

or

the_list = []
for i in range(16):
    the_list.append('even' if i%2 == 0 else 'odd')
```
**Goal:** A list of 'odd' or 'even' if that index is odd or even.

```python
the_list = []
for i in range(16):
    if i%2 == 0:
        the_list.append('even')
    else:
        the_list.append('odd')
```

or

```python
the_list = ['even' if i%2 == 0 else 'odd' for i in range(16)]
```
Lecture Overview

• Sorting
• List comprehension
• Data visualization
A Motivating Problem

• For various cities around the world, we would like to examine the “Sun Up” time throughout the year.

• How does it vary from day to day?

• What are the monthly averages?

Sun Up Time = Sunset Time – Sunrise Time
How Does Sun-Up Time Vary Day-to-Day?
How Does Sun-Up Time Vary Month-to-Month?
The Task Before Us...

1. Find a website where the data can be found.
2. Get that data into a file on our computer.
3. Understand how the data is laid out in the file.
4. Write python code that gets that data (or some aspect of it) into your Python environment.
Where Do We Get the Data?

• Lots of choices. Google “Sunset Sunrise times”

• We will use the U.S. Naval Observatory data service:

• Visit:

http://www.usno.navy.mil/
From the Website...

Astronomical Applications

Data Services

Sun and Moon rise and set times, Moon phases, eclipses, seasons, positions of solar system objects, and other data

Complete Sun and Moon Data for One Day
Sun or Moon Rise/Set Table for One Year
Phases of the Moon
more...
We Downloaded Rise/Set Data For a Number of Cities

Anaheim  Anchorage  Arlington  Athens  Atlanta
Baltimore  Bangkok  Beijing  Berlin  Bogota
Boston  BuenosAires  Cairo  Chicago  Cincinnati
Cleveland  Denver  Detroit  Honolulu  Houston
Ithaca  Johannesburg  KansasCity  Lagos  London
LosAngeles  MexicoCity  Miami  Milwaukee  Minneapolis
Moscow  NewDelhi  NewYork  Oakland  Minneapolis
Philadelphia  Phoenix  Pittsburgh  RiodeJaneiro  Paris
SanFrancisco  Seattle  Seoul  Sydney  Rome
Teheran  Tokyo  Toronto  Washington  Tampa

Wellington
One .dat File Per City

We put all these files in a directory called RiseSetData.

.dat and .txt files are common ways to house simple data. Don’t worry about the difference.
.txt and .dat Files have Lines

MyFile.dat

abcd
123 abc d fdd
xyz
3.14159   2.12345

There is an easy way to read the data in such a file line-by-line
Read and Print the Data in Ithaca.dat

FileIO.py

FileName = 'RiseSetData/Ithaca.dat'
f = file(FileName, 'r')
for s in f:
    print s
f.close()

RiseSetData and FileIO.py must be in the same folder.
The provider of the file typically tells you how the data is structured
From the Naval Observatory Website

• The first line names the city and the second line encodes its latitude and longitude, e.g.,

Ithaca
W07629N4226

and ...
From the Naval Observatory Website

• The rise and set times are then specified day-by-day with the data for each month housed in a pair of columns.

• In particular, columns 2k and 2k+1 have the rise and set times for month k (Jan=1, Feb = 2, Mar = 3, etc.)

• Column 1 specifies day-of-the-month, 1 through 31. Blanks are used for nonexistent dates (e.g., April 31).
The Data for a Particular City is Housed in a 33-line .dat file

<table>
<thead>
<tr>
<th>Line</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>2</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>3</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>28</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>29</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>31</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
</tbody>
</table>

Line 1 has the name of the city
The Data for a Particular City is Housed in a 33-line .dat file

Ithaca

W07629N4226


Line 2 encodes its longitude and latitude
**Helper Function: LongLat**

- A latlong string has length 11, e.g. **W08140N4129**

```python
def LongLat(s):
    """ Returns a tuple (Long,Lat) of floats that are the equivalent (in degrees) of the longitude and latitude encoded by s.

PredC: s an 11-character string of the form 'cdddmCDDMM' where cdddm specifies longitude in degrees and minutes with c = 'W' or 'E' and CDDMM species latitude in degrees and minutes with C = 'N' or 'S'
"""
    Long = float(s[1:4])+float(s[4:6])/60
    if s[0]=='E':
        Long = -Long
    Lat = float(s[7:9])+float(s[9:11])/60
    if s[6]=='S':
        Lat = -Lat
    return (Lat,Long)
```
The Data for a Particular City is Housed in a 33-line .dat file

Ithaca
W07629N4226

|   | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |
| 1 | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |
| 2 | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |
| 3 | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |
| 28| R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |
| 29| R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |
| 30| R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |
| 31| R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S | R | S |

The remaining lines house the rise-set data.
Each R and S is a length-4 string: ‘0736’
Helper Function: ConvertTime

def ConvertTime(s):
    """ Returns a float that is the equivalent (in hours) of the time encoded by s.

'2145' means 9:45 pm.

PredC: s a 4-character string of the form hhmm that specifies time.
"""

    x = float(s[:2])+float(s[2:])/60
    return x

• In comes a length-4 string and back comes a float that encodes the time in hours
• ‘0736’ ----> 7 + 36/60 hours ----> 7.6
The Data for a Particular City is Housed in a 33-line .dat file

Ithaca
W07629N4226

Day - Number followed by 12 rise-set pairs, one pair for each month
The Data for a Particular City is Housed in a 33-line .dat file

Day -Number followed by 11 rise-set pairs, one pair for each month except February
The Data for a Particular City is Housed in a 33-line .dat file


Day-Number followed by 7 rise-set pairs, one pair for each 31-day month
Recall the Motivating Problem

• For various cities around the world, we would like to examine the “Sun Up” time throughout the year.

• How does it vary from day to day?

• What are the monthly averages?
def SunUp(CityName):
    FileName = 'RiseSetData/' + CityName + '.dat'
    f = file(FileName, 'r');
    lineNum = 0
    for s in f:
        parts = s.split()
        lineNum += 1
        if lineNum == 1:
            City = parts[0]
        elif lineNum == 2:
            Lat, Long = LatLong(parts[0])
        else:
            Code that builds the RiseTime and SetTime arrays
            f.close()
    return (City, Lat, Long, SetTime - RiseTime)
Building **RiseTime** and **SetTime** arrays

... # Remaining lines have rise/set pairs
day = int(parts[0])
# Get all the rise and set times
RiseTimeList = ConvertTime(parts[1:len(parts):2])
SetTimeList = ConvertTime(parts[2:len(parts):2])
p = len(RiseTimeList)
for k in range(p):
    if day<=28:
        # All months have at least 28 days
        starts = [0,31,59,90,120,151,181,212,243,273,304,334]
        dayIndex = day + starts[k] - 1
    elif day==29 or day==30:
        # All months except February have a day 29 and a day 30
        starts = [0,59,90,120,151,181,212,243,273,304,334]
        dayIndex = day + starts[k] - 1
    else:
        # Only January, March, May, July, August, October, and December have
        # a day 31.
        starts = [0,59,120,181,212,273,334]
        dayIndex = day + starts[k] - 1
    RiseTime[dayIndex] = RiseTimeList[k]
    SetTime[dayIndex] = SetTimeList[k]
from pylab import *

# Plot a 1-dim numpy array
City, Lat, Long, D = SunUp('Ithaca')
plot(D)
show()

This is how you display the values in a numpy array like D.
How about a title and a labeling of the y-axis?
A Simple Plot

# Plot a 1-dim numpy array
City, Lat, Long, D = SunUp('Ithaca')
plot(D)

# The title
titlestr = '%s Lat = %6.2f Long = %6.2f' % (City,Lat,Long)
title(titlestr,fontsize=16)
# Label the y-axis
ylabel('Hours of Sunlight',fontsize=16)

show()
Modify the x range and the y range
A Simple Plot

# Plot a 1-dim numpy array
city, lat, long, d = sunup('Ithaca')
plot(d)

# The title
titlestr = '%s Lat = %6.2f Long = %6.2f' % (city, lat, long)
title(titlestr, fontsize=16)
# Label the y-axis
ylabel('Hours of Sunlight', fontsize=16)

# set the range of x and the range of y
xlim(0, 364)
ylim(5, 20)

show()
Label the x-axis with month names
A Simple Plot

# Plot a 1-dim numpy array
City, Lat, Long, D = SunUp('Ithaca')
plot(D)

# The title
titlestr = '%s Lat = %6.2f Long = %6.2f' % (City,Lat,Long)
title(titlestr,fontsize=16)

# Label the y-axis
ylabel('Hours of Sunlight',fontsize=16)

# set the range of x and the range of y
xlim(0,364)
ylim(5,20)

# Position ticks along the x-axis and label them
c = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']
t = [15,45,75,105,135,165,195,225,255,285,315,345]
xticks( t,c)

show()
Ithaca  Lat = 42.43  Long = 76.48

Add a Grid
A Simple Plot

# Plot a 1-dim numpy array
City, Lat, Long, D = SunUp('Ithaca')
plot(D)

# The title
titlestr = '%s Lat = %6.2f Long = %6.2f' % (City,Lat,Long)
title(titlestr,fontsize=16)

# Label the y-axis
ylabel('Hours of Sunlight',fontsize=16)

# set the range of x and the range of y
xlim(0,364)
ylim(5,20)

# Position ticks along the x-axis and label them
c = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']
t = [15,45,75,105,135,165,195,225,255,285,315,345]
xticks( t,c)

# Draw a grid
for k in range(6,20):
    # Draw horizontal line from (0,k) to (65,k)
    plot(array([0,365]),array([k,k]),color='red',linestyle=':')
for k in [0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334]:
    # Draw vertical line from (k,5) to (k,20)
    plot(array([k,k]),array([5,20]),color='red',linestyle=':')

show()
def MonthAverages(CityName):
    x = zeros((12,1))
    City, Lat, Long, D = SunUp(CityName)
    start = [0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334]
    finish = [30, 58, 89, 119, 150, 180, 211, 242, 272, 303, 333, 364]
    for k in range(12):
        z = D[start[k]:finish[k]]
        x[k] = sum(z)/len(z)
    return x
A Bar Plot

\[ M = \text{MonthAverages('Ithaca')} \]

\[
\text{bar(range(12), M, facecolor='magenta')}
\]

\[
\text{xlim(-.2, 12)}
\]

\[
\text{ylabel('Average Hours of Sunlight')}\]

\[
\text{title(A.City, fontsize=16)}
\]

\[
\text{show()}
\]
The diagram shows the average hours of sunlight in Ithaca over the months of the year. The months with the highest average hours of sunlight are June and July, while the lowest are January and December. Notable peaks are seen in May and August as well.