# average4.py

# A program to average a set of numbers
# Illustrates sentinel loop using empty string as sentinel

```python
def main():
    sum = 0.0
    count = 0
    xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    print "The average of the numbers is", sum / count
main()
```

---

# Example Problem: Simple Statistics

- Many programs deal with large collections of similar information.
  - Words in a document
  - Students in a course
  - Data from an experiment
  - Customers of a business
  - Graphics objects drawn on the screen
  - Cards in a deck

---

# Sample Problem: Simple Statistics

Let’s review some code we wrote in chapter 8:

```python
# averaged.py
# A program to average a set of numbers
# Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    print "The average of the numbers in", sum / count
main()
```
Sample Problem: Simple Statistics

- The median is the data value that splits the data into equal-sized parts.
- For the data 2, 4, 6, 9, 13, the median is 6, since there are two values greater than 6 and two values that are smaller.
- One way to determine the median is to store all the numbers, sort them, and identify the middle value.

Sample Problem: Simple Statistics

- The standard deviation is a measure of how spread out the data is relative to the mean.
- If the data is tightly clustered around the mean, then the standard deviation is small. If the data is more spread out, the standard deviation is larger.
- The standard deviation is a yardstick to measure/express how exceptional the data is.

Sample Problem: Simple Statistics

- The standard deviation is \[ \sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{n-1}} \]
- Here \( \mu \) is the mean, \( x_i \) represents the \( i \)th data value and \( n \) is the number of data values.
- The expression \( (x_i - \mu)^2 \) is the square of the “deviation” of an individual item from the mean.

Sample Problem: Simple Statistics

- The numerator is the sum of these squared “deviations” across all the data.
- Suppose our data was 2, 4, 6, 9, and 13.
- The mean is 6.8
- The numerator of the standard deviation is \( (6.8-2)^2 + (6.8-4)^2 + (6.8-6)^2 + (6.8-9)^2 + (6.8-13)^2 = 149.6 \)
- \[ \sigma = \sqrt{\frac{149.6}{4}} = \sqrt{37.4} = 6.11 \]

Sample Problem: Simple Statistics

- As you can see, calculating the standard deviation not only requires the mean (which can’t be calculated until all the data is entered), but also each individual data element!
- We need some way to remember these values as they are entered.

Applying Lists

- We need a way to store and manipulate an entire collection of numbers.
- We can’t just use a bunch of variables, because we don’t know many variables there will be.
- What do we need? Some way of combining an entire collection of values into one object.
Lists and Arrays

- Python lists are ordered sequences of items. For instance, a sequence of \( n \) numbers might be called \( S \):
  \[ S = s_0, s_1, s_2, \ldots, s_{n-1} \]
- Specific values in the sequence can be referenced using subscripts.
- By using numbers as subscripts, mathematicians can succinctly summarize computations over items in a sequence using subscript variables.
  \[ \sum_{i=0}^{n-1} s_i \]

Lists and Arrays

- Suppose the sequence is stored in a variable \( s \). We could write a loop to calculate the sum of the items in the sequence like this:
  \[ \text{sum} = 0 \]
  \[ \text{for } i \text{ in range}(n): \text{sum} = \text{sum} + s[i] \]
- Almost all computer languages have a sequence structure like this, sometimes called an array.

Lists and Arrays

- A list or array is a sequence of items where the entire sequence is referred to by a single name (i.e. \( s \)) and individual items can be selected by indexing (i.e. \( s[i] \)).
- In other programming languages, arrays are generally a fixed size, meaning that when you create the array, you have to specify how many items it can hold.
- Arrays are generally also homogeneous, meaning they can hold only one data type.

List Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{seq}&gt; + \text{&lt;seq&gt;})</td>
<td>Concatenation</td>
</tr>
<tr>
<td>(\text{&lt;seq&gt;} \ast \text{&lt;int-expr&gt;})</td>
<td>Repetition</td>
</tr>
<tr>
<td>(\text{&lt;seq&gt;}[])</td>
<td>Indexing</td>
</tr>
<tr>
<td>(\text{len(&lt;seq&gt;)})</td>
<td>Length</td>
</tr>
<tr>
<td>(\text{&lt;seq&gt;}[:])</td>
<td>Slicing</td>
</tr>
<tr>
<td>(\text{for &lt;var&gt; in &lt;seq&gt;:})</td>
<td>Iteration</td>
</tr>
<tr>
<td>(\text{&lt;expr&gt;} \in \text{&lt;seq&gt;})</td>
<td>Membership (Boolean)</td>
</tr>
</tbody>
</table>

List Operations

- Except for the membership check, we’ve used these operations before on strings.
- The membership operation can be used to see if a certain value appears anywhere in a sequence.
  >>> \(1\text{st} = [1,2,3,4]\)
  >>> 3 in \(1\text{st}\)
  True
List Operations

- The summing example from earlier can be written like this:
  ```python
  sum = 0
  for x in s:
    sum = sum + x
  ```

- Unlike strings, lists are mutable:
  ```python
  >>> lst = [1, 2, 3, 4]
  >>> lst[3] = 4
  >>> lst[3] = "Hello"
  >>> lst
  [1, 2, 3, 'Hello']
  ```

- Here, `nums` is being used as an accumulator, starting out empty, and each time through the loop a new value is tacked on.
  ```python
  >>> lst = [3, 1, 4, 1, 5, 9]
  >>> lst.append(2)
  >>> lst
  [3, 1, 4, 1, 5, 9, 2]
  ```

```
List Operations

- Lists are often built up one piece at a time using `append`:
  ```python
  nums = []
  x = input('Enter a number: ')
  while x >= 0:
    nums.append(x)
    x = input('Enter a number: ')
  ```

```
```
List Operations

- A list of identical items can be created using the repetition operator. This command produces a list containing 50 zeroes:
  ```python
  zeroes = [0] * 50
  ```

```
```
List Operations

- Most of these methods don’t return a value – they change the contents of the list in some way.
- Lists can grow by appending new items, and shrink when items are deleted. Individual items or entire slices can be removed from a list using the `del` operator.
  ```python
  >>> lst = [3, 1, 4, 1, 5, 9]
  >>> lst.append(2)
  >>> lst
  [3, 1, 4, 1, 5, 9, 2]
  ```

- Inserting an item at a specific position in the list:
  ```python
  >>> lst.insert(4, "Hello")
  >>> lst
  [3, 1, 4, 1, 5, 9, 2, 'Hello']
  ```

- Counting occurrences of an item in the list:
  ```python
  >>> lst.count(1)
  2
  ```

- Removing an item from the list:
  ```python
  >>> lst.remove(1)
  >>> lst
  [9, 5, 4, 'Hello', 2, 1]
  ```

- Popping the last element from the list:
  ```python
  >>> lst.pop()
  1
  ```

- Slicing the list to create a new list:
  ```python
  >>> lst[1:3]
  [4, 1]
  ```

```

---

Python Programming, 1/e

19

20

21

22

23

24
List Operations

- >>> myList=[34, 26, 0, 10]
  >>> del myList[1]
  >>> myList
  [34, 0, 10]
  >>> del myList[1:3]
  >>> myList
  [34]

- del isn’t a list method, but a built-in operation that can be used on list items.

List Operations

- Lists support a number of convenient and frequently used methods.
- Lists will grow and shrink as needed.

Statistics with Lists

- Let’s write a function called getNumbers that gets numbers from the user.
- We’ll implement the sentinel loop to get the numbers.
- An initially empty list is used as an accumulator to collect the numbers.
- The list is returned once all values have been entered.

Statistics with Lists

- Basic list principles
  - A list is a sequence of items stored as a single object.
  - Items in a list can be accessed by indexing, and sublists can be accessed by slicing.
  - Lists are mutable; individual items or entire slices can be replaced through assignment statements.

Statistics with Lists

- One way we can solve our statistics problem is to store the data in lists.
- We could then write a series of functions that take a list of numbers and calculates the mean, standard deviation, and median.
- Let's rewrite our earlier program to use lists to find the mean.

Statistics with Lists

- Using this code, we can get a list of numbers from the user with a single line of code:
  - data = getNumbers()
Statistics with Lists

- Now we need a function that will calculate the mean of the numbers in a list.
  - Input: a list of numbers
  - Output: the mean of the input list
  - Definition:
    ```python
def mean(nums):
    sum = 0.0
    for num in nums:
        sum = sum + num
    return sum / len(nums)
```
def median(nums):
    nums.sort()
    size = len(nums)
    midPos = size // 2
    if size % 2 == 0:
        median = (nums[midPos] + nums[midPos-1]) / 2.0
    else:
        median = nums[midPos]
    return median

def main():
    print('This program computes mean, median and standard deviation.
    data = getNumbers()
    xbar = mean(data)
    std = stdDev(data, xbar)
    med = median(data)
    print('The mean is', xbar)
    print('The standard deviation is', std)
    print('The median is', med)

if __name__ == '__main__': main()
Lists of Objects

- Let’s begin with the file processing. The following code reads through the data file and creates a list of students.
- def readStudents(filename):
  infile = open(filename, 'r')
  students = []
  for line in infile:
    students.append(makeStudent(line))
  infile.close()
  return students

We’re using the makeStudent from the gpa program, so we’ll need to remember to import it.

Lists of Objects

- Using the functions readStudents and writeStudents, we can convert our data file into a list of students and then write them back to a file. All we need to do now is sort the records by GPA.
- In the statistics program, we used the sort method to sort a list of numbers. How does Python sort lists of objects?

Lists of Objects

- Python compares items in a list using the built-in function cmp.
- cmp takes two parameters and returns -1 if the first comes before the second, 0 if they’re equal, and 1 if the first comes after the second.

Lists of Objects

- To make sorting work with our objects, we need to tell sort how the objects should be compared.
- We do this by writing our own custom cmp-like function and then tell sort to use it when sorting.
- To sort by GPA, we need a routine that will take two students as parameters and then returns -1, 0, or 1.
Lists of Objects

- We can use the built-in `cmp` function.
  ```python
  def cmpGPA(s1, s2):
      return cmp(s1.gpa(), s2.gpa())
  ```
- We can now sort the data by calling `sort` with the appropriate comparison function (`cmpGPA`) as a parameter.
  ```python
  data.sort(cmpGPA)
  ```

Designing with Lists and Classes

- In the `dieView` class from chapter ten, each object keeps track of seven circles representing the position of pips on the face of the die.
- Previously, we used specific instance variables to keep track of each, `pip1`, `pip2`, `pip3`, ...

```
# dieView.py
# A program to sort student information into GPA order
from sys import Student
import Student

def readStudent(filename):
    stu = open(filename, "r")
    students = []
    stu.readline()  # Skip header
    students.append(makeStudent(stu))
    return students

def writeStudent(students, filename):
    outf = open(filename, "w")
    outf.write("name\t\t\t\t\t\tGPA\n" + S7920459)
    for s in students:
        outf.write(s.firstName + "\t\t\t\t\t\t" + str(s.gpa()) + "\n")
    outf.close()

def cmpGPA(s1, s2):
    return cmp(s1.gpa(), s2.gpa())

def main():
    filename = raw_input("Enter the name of the data file: ").strip()
    data = readStudents(filename)
    data.sort(cmpGPA)
    writeStudents(data, filename)
    print "The data has been written to", filename
    if _name_ == _main_
        main()
```

Designing with Lists and Classes

- What happens if we try to store the circle objects using a list?
  - In the previous program, the pips were created like this:
    ```python
    self.pip1 = self.__makePip(cx, cy)
    ```
  - `__makePip` is a local method of the `DieView` class that creates a circle centered at the position given by its parameters.
  ```python
  def __makePip(self, cx, cy):
      return circle(cx, cy, 0.5, [0, 0, 0], [0, 0, 0])
  ```

Designing with Lists and Classes

- One approach is to start with an empty list of pips and build up the list one pip at a time.
  ```python
  self.pips = []
  pips.append(self.__makePip(cx-off, cy-off))
  pips.append(self.__makePip(cx-off, cy-off))
  self.pips = pips
  ```
An even more straightforward approach is to create the list directly.

```python
self.pips = [self.__makePip(cx-offset, cy-offset),
             self.__makePip(cx, cy),
             self.__makePip(cx+offset, cy+offset)]
```

Python is smart enough to know that this object is continued over a number of lines, and waits for the `']`

Listing objects like this, one per line, makes it much easier to read.

```
for pip in self.pips:
    pip.setFill(self.background)
```

Putting our pips into a list makes many actions simpler to perform.

To blank out the die by setting all the pips to the background color:

```
for pip in self.pips:
    pip.setFill(self.background)
```

This cut our previous code from seven lines to two!

```
for pip in self.pips:
    pip.setFill(self.foreground)
```

We can turn the pips back on using the pips list. Our original code looked like this:

```
self.pip1.setFill(self.foreground)
self.pip4.setFill(self.foreground)
self.pip7.setFill(self.foreground)
```

Into this:

```
self.pips[0].setFill(self.foreground)
self.pips[3].setFill(self.foreground)
self.pips[6].setFill(self.foreground)
```

Here's an even easier way to access the same methods:

```
for i in [0,3,6]:
    self.pips[i].setFill(self.foreground)
```

We can take advantage of this approach by keeping a list of which pips to activate!

```
for i in on:
    self.pips[i].setFill(self.foreground)
```

Determine the list of pip indexes to turn on

Loop through the list of indexes - turn on those pips

```
for pip in self.pips:
    if value == 1:
        on = [3]
    elif value == 2:
        on = [0,6]
    elif value == 3:
        on = [0,3,6]
    elif value == 4:
        on = [0,2,4,6]
    elif value == 5:
        on = [0,2,3,4,6]
    else:
        on = [0,1,2,3,4,5,6]
    for i in on:
        self.pips[i].setFill(self.foreground)
```

We can do even better!

The correct set of pips is determined by value. We can make this process table-driven instead.

```
for i in [0,3,6]:
    self.pips[i].setFill(self.foreground)
```

We can use a list where each item on the list is itself a list of pip indexes.

For example, the item in position 3 should be the list [0,3,6] since these are the pips that must be turned on to show a value of 3.
Designing with Lists and Classes

- Here’s the table-driven code:

```python
onTable = [ [], [3], [2,4], [2,3,4], [0,2,4,6], [0,2,3,4,6], [0,1,2,4,5,6] ]

for pip in self.pips:
    self.pip.setFill(self.background)

on = onTable[value]
for i in on:
    self.pips[i].setFill(self.foreground)
```

- The table is padded with `[ ]` in the 0 position, since it shouldn’t ever be used.
- The `onTable` will remain unchanged through the life of a `dieView`, so it would make sense to store this table in the constructor and save it in an instance variable.

---

Case Study: Python Calculator

- The new `dieView` class shows how lists can be used effectively as instance variables of objects.
- Our pips list and `onTable` contain circles and lists, respectively, which are themselves objects.
- We can view a program itself as a collection of data structures (collections and objects) and a set of algorithms that operate on those data structures.

A Calculator as an Object

- Let’s develop a program that implements a Python calculator.
- Our calculator will have buttons for
  - The ten digits (0-9)
  - A decimal point (.)
  - Four operations (+, -, *, /)
  - A few special keys
    - ‘C’ to clear the display
    - ‘<’ to backspace in the display
    - ‘=’ to do the calculation

```python
```
A Calculator as an Object

We can take a simple approach to performing the calculations. As buttons are pressed, they show up in the display, and are evaluated and displayed when the = is pressed.

We can divide the functioning of the calculator into two parts: creating the interface and interacting with the user.

Constructing the Interface

First, we create a graphics window.
The coordinates were chosen to simplify the layout of the buttons.
In the last line, the window object is stored in an instance variable so that other methods can refer to it.
def __init__(self):
    # create the window for the calculator
    win = GraphWin("calculator")
    win.setCoords(0,0,6,7)
    win.setBackground("slategray")
    self.win = win

Constructing the Interface

Our next step is to create the buttons, reusing the button class.
A standard button is defined as

def __init__(self, cx, cy, label):
    self.win = win
    self.cx, self.cy = cx, cy
    self.button = Text(Point(cx, cy), label)
    self.button.setSize(30)
    self.button.setStyle("bold")
    self.button.setTextColor("black")
    self.button.setFill("white")
    self.button.setFace("Times New Roman")

bspecs contains a list of button specifications, including the center point of the button and its label.

Constructing the Interface

Each specification is a tuple.
A tuple looks like a list but uses '(') rather than '['].
Tuples are sequences that are immutable.

Conceptually, each iteration of the loop starts with an assignment:
(cx, cy, label)=next item from bspecs
Each item in bspecs is also a tuple.
When a tuple of variables is used on the left side of an assignment, the corresponding components of the tuple on the right side are unpacked into the variables on the left side.
The first time through it's as if we had:
cx, cy, label = 2,1,"0"
Constructing the Interface

- Each time through the loop, another tuple from bSpecs is unpacked into the variables in the loop heading.
- These values are then used to create a Button that is appended to the list of buttons.
- Creating the display is simple – it’s just a rectangle with some text centered on it. We need to save the text object as an instance variable so its contents can be accessed and changed.

Processing Buttons

- Now that the interface is drawn, we need a method to get it running.
- We’ll use an event loop that waits for a button to be clicked and then processes that button.
- def run(self):
      # Infinite 'event loop' to process button clicks.
      while True:
          key = self.getButton()
          if key != None:
              self.processButton(key)

Processing Buttons

- We continue getting mouse clicks until a button is clicked.
- To determine whether a button has been clicked, we loop through the list of buttons and check each one.
- def getButton(self):
      # Waits for a button to be clicked and
      # returns the label of
      # the button that was clicked.
      while True:
          p = self.win.getMouse()
          for b in self.buttons:
              if b.clicked(p):
                  return b.getLabel() # method exit

Processing Buttons

- Having the buttons in a list like this is a big win. A for loop is used to look at each button in turn.
- If the clicked point p turns out to be in one of the buttons, the label of the button is returned, providing an exit from the otherwise infinite loop.

Processing Buttons

- The last step is to update the display of the calculator according to which button was clicked.
- A digit or operator is appended to the display. If key contains the label of the button, and text contains the current contents of the display, the code is:
      self.display.setText(text+key)
Processing Buttons

- The clear key blanks the display:
  ```python
  self.display.setText(""")
  ```
- The backspace key strips off one character:
  ```python
  self.display.setText(text[:-1])
  ```
- The equal key causes the expression to be evaluated and the result displayed.

Exception handling is necessary here to catch run-time errors if the expression being evaluated isn’t a legal Python expression. If there’s an error, the program will display 'ERROR' rather than crash.

Non-Sequential Collections

- Python provides another built-in data type for collections, called a `dictionary`.
- Not all computer languages have dictionaries, while almost all have arrays or lists.

Dictionary Basics

- The combination of a social security with other data is known as a `key-value pair`.
- We access the value (the student information) associated with a particular key (the social security number).
- It’s easy to think of many key-value pairs: usernames and passwords, names and phone numbers, etc.
- A collection that allows us to look up data with arbitrary keys is called a `mapping`.
- Python dictionaries are `mappings`.
- Some languages call them `hashes` or `associative arrays`. 

Dictionary Basics

- Typically, when we retrieve information from a sequential collection, we look it up by its `position`, or index, in the collection.
- Say you want to retrieve data about students or employees based on social security numbers.
Dictionary Basics

- A dictionary can be created in Python by listing key-value pairs inside curly braces.
- >>> passwd = {"guido": "superprogrammer", "turing": "genius", "bill": "monopoly"}
- Keys and values are joined with ':', and commas are used to separate pairs.

Dictionary Basics

- The main use of a dictionary is to look up the value associated with a particular key, using indexing notation.
- >>> passwd["guido"]
  'superprogrammer'
- >>> passwd["bill"]
  'monopoly'
- <dictionary>[<key>] returns the object associated with the given key.

Dictionary Basics

- Dictionaries are mutable. The value associated with a key can be changed with assignment.
- >>> passwd["bill"] = "bluescreen"
  >>> passwd
t {'turing': 'genius', 'bill': 'bluescreen', 'guido': 'superprogrammer'}
- Did you notice the dictionary didn’t print out in the same order it was entered? Mappings are unordered.

Dictionary Basics

- Python stores dictionaries in a way that makes key lookup very efficient.
- If you want to keep a collection of items in a certain order, use a sequence!

Dictionary Basics

- Dictionaries are mutable collections that implement a mapping from keys to values.
- Keys can be any immutable type, and values can be any type, including programmer-defined classes.

Dictionary Operations

- Python dictionaries support several built-in operations.
- Dictionaries can be extended (data added after creation) by adding new entries.
  >>> passwd["newuser"] = "ImANewbie"
  >>> passwd
t {'turing': 'genius', 'bill': 'bluescreen', 'newuser': 'ImANewbie', 'guido': 'superprogrammer'}
Dictionary Operations

- A common way to build a dictionary is to start with an empty collection and add the key-value pairs one at a time.
- Suppose usernames and passwords were stored in a file called passwords, where each line of the file contains a username and password, separated by a space.

```python
passwd = {}
for line in open('passwords', 'r'):
    user, pass = string.split(line)
    passwd[user] = pass
```

```python
# Dictionary Operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dict&gt;.has_key(&lt;key&gt;)</td>
<td>Returns true if dictionary contains the specified key, false otherwise. Same as has_key.</td>
</tr>
<tr>
<td>&lt;dict&gt;.keys()</td>
<td>Returns a list of the keys.</td>
</tr>
<tr>
<td>&lt;dict&gt;.values()</td>
<td>Returns a list of the values.</td>
</tr>
<tr>
<td>&lt;dict&gt;.items()</td>
<td>Returns a list of tuples (key, value)</td>
</tr>
<tr>
<td>&lt;dict&gt;.get(&lt;key&gt;, &lt;default&gt;)</td>
<td>If dictionary has key, returns its value; otherwise returns default.</td>
</tr>
<tr>
<td>del &lt;dict&gt; [&lt;key&gt;]</td>
<td>Deletes the specified entry.</td>
</tr>
<tr>
<td>&lt;dict&gt;.clear()</td>
<td>Deletes all entries.</td>
</tr>
</tbody>
</table>
```

Example Program: Word Frequency

- Now that we have dictionaries, we can use them to create a program that analyzes text and counts how many times each word appears in the document.
- This type of analysis can be used as a measure of style similarity between two documents, and is used by web indexing and archiving program (like Internet search engines).

```python
# Example Program: Word Frequency

>>> passwd.keys()
['turing', 'bill', 'newuser', 'guido']
>>> passwd.values()
['genius', 'bluescreen', 'ImANewbie', 'superprogrammer']
>>> passwd.items()
[('turing', 'genius'), ('bill', 'bluescreen'), ('newuser', 'ImANewbie'), ('guido', 'superprogrammer')]
>>> passwd.has_key('bill')
True
>>> 'fred' in passwd
False
>>> passwd.get('bill', 'unknown')
'bluescreen'
>>> passwd.get('john', 'unknown')
'unknown'
>>> passwd.clear()
>>> passwd
{}
```
**Example Program: Word Frequency**

- We can use a dictionary where the keys are strings representing the words, and the values are ints that count how many times the words appear.
- Let’s call our dictionary `counts`.
- To update the count for a word `w`:
  ```python
counts[w] = counts[w] + 1
  ```

**Example Program: Word Frequency**

- There’s only one catch: The first time we encounter a word, it’s not yet in `counts`. Attempting to access a non-existent key produces a run-time `KeyError`.
  - if `w` is already in `counts`:
    - add one to the count for `w`
  - else:
    - set count for `w` to 1

**Example Program: Word Frequency**

- This decision ensures that the first time a word is encountered, it will be added to the dictionary with a count of 1.
  - if `counts.has_key(w)`:
    ```python
counts[w] = counts[w] + 1
  ```
  - else:
    ```python
counts[w] = 1
  ```

**Example Program: Word Frequency**

- A more elegant approach:
  ```python
counts[w] = counts.get(w, 0) + 1
  ```
  - If `w` is not already in the dictionary, `get` returns 0, and the result is that the entry for `w` is set to 1.

**Example Program: Word Frequency**

- The first task is to split the text document into a sequence of words.
- While doing this, all text will be converted to lower case (so words like “Spam” and “spam” match), and punctuation will be removed (so “Spam!” matches “Spam!”).

```python
fname = raw_input("File to analyze: ")
# read file as one long string
text = open(fname, 'r').read()
# convert all letters to lower case
text = string.lower(text)
# replace each punctuation character with a space
for ch in "!#$%&()*+,-./:;<=>?@\[\]^\_\|~":
text = string.replace(text, ch, ' ')
# split string at whitespace to form a list of words
words = string.split(text)
# Now we can easily loop through the words to build
# the counts dictionary
```
Example Program: Word Frequency

- We build the counts dictionary:
  ```python
counts = {}
  for w in words:
    counts[w] = counts.get(w, 0) + 1
```
- We could print a report that summarizes the contents of `counts`.
  We could put the words into alphabetical order.
- How can we do that?

Example Program: Word Frequency

- What we really want is the \( n \) most common words in the document.
- To organize the report like this, we need to first sort on the counts of the words, rather than the words themselves. Then we print the first \( n \).

Example Program: Word Frequency

- We need a custom comparison function that takes two items (like our word-count tuples) and returns either -1, 0, or 1, giving their relative ordering.
  ```python
def compareItems((w1, c1), (w2, c2)):
  if c1 > c2:
    return -1
  elif c1 == c2:
    return cmp(w1, w2)
  else:
    return 1
```
- This function takes two parameters, each is a two-valued tuple.
- If the count on the first tuple is greater than the count of the second, then the first item should precede the second, so we return -1.
def compareItems((w1, c1), (w2, c2)):
    if c1 > c2:
        return -1
    elif c1 == c2:
        return cmp(w1, w2)
    else:
        return 1

# sort our items
items.sort(compareItems)

for i in range(n):
    print %-10s%5d % items[i]

Sample Program: Word Frequency

The last step is to print out the n most common words.

for i in range(n):
    print "%10s%5d % items[i]

Note the formatting in the print:

A string is left-justified in 10 spaces
Followed by an int right-justified in 5 spaces
Normally we'd need two values to fill the "slots", but items[i] is already two values.