Python Programming: An Introduction to Computer Science

Chapter 11
Data Collections

Objectives

- To understand the use of lists (arrays) to represent a collection of related data.
- To be familiar with the functions and methods available for manipulating Python lists.
- To be able to write programs that use lists to manage a collection of information.

Objectives

- To be able to write programs that use lists and classes to structure complex data.
- To understand the use of Python dictionaries for storing nonsequential collections.

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:

```
# average4.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 "\nThe average of the numbers is", sum / count

main()
```


Sample Problem: Simple Statistics

- This program allows the user to enter a sequence of numbers, but the program itself doesn't keep track of the numbers that were entered – it only keeps a running total.
- Suppose we want to extend the program to compute not only the mean, but also the median and standard deviation.

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

$$s = \sqrt{\frac{\sum(\bar{x} - x_i)^2}{n-1}}$$

- Here $\bar{x}$ is the mean, x_i represents the i^{th} data value and n is the number of data values.
- The expression $(\bar{x} - x_i)^2$ 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

$$(6.8-2)^2 + (6.8-4)^2 + (6.8-6)^2 + (6.8-9)^2 + (6.8-13)^2 = 149.6$$

$$s = \sqrt{\frac{149.6}{5-1}} = \sqrt{37.4} = 6.11$$

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 numbers 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*:
  $S = s_0, s_1, s_2, s_3, \ldots, s_{n-1}$
- 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:

```
sum = 0
for i in range(n):
    sum = 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.

Lists and Arrays

- Python lists are dynamic. They can grow and shrink on demand.
- Python lists are also *heterogeneous*, a single list can hold arbitrary data types.
- Python lists are mutable sequences of arbitrary objects.

List Operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

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.

```
>>> lst = [1,2,3,4]
>>> 3 in lst
True
```

List Operations

- The summing example from earlier can be written like this:

```
sum = 0
for x in s:
    sum = sum + x
```

- Unlike strings, lists are mutable:

```
>>> lst = [1,2,3,4]
>>> lst[3]
4
>>> lst[3] = "Hello"
>>> lst
[1, 2, 3, 'Hello']
>>> lst[2] = 7
>>> lst
[1, 2, 7, 'Hello']
```


List Operations

- A list of identical items can be created using the repetition operator. This command produces a list containing 50 zeroes:

```
zeroes = [0] * 50
```


List Operations

- Lists are often built up one piece at a time using append.

```
nums = []
x = input('Enter a number: ')
while x >= 0:
    nums.append(x)
    x = input('Enter a number: ')
```

- Here, `nums` is being used as an accumulator, starting out empty, and each time through the loop a new value is tacked on.

List Operations

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

List Operations

```
>>> lst = [3, 1, 4, 1, 5, 9]
>>> lst.append(2)
>>> lst
[3, 1, 4, 1, 5, 9, 2]
>>> lst.sort()
>>> lst
[1, 1, 2, 3, 4, 5, 9]
>>> lst.reverse()
>>> lst
[9, 5, 4, 3, 2, 1, 1]
>>> lst.index(4)
2
>>> lst.insert(4, "Hello")
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1, 1]
>>> lst.count(1)s
2
>>> lst.remove(1)
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1]
>>> lst.pop(3)
3
>>> lst
[9, 5, 4, 'Hello', 2, 1]
```


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.

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

- 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.

List Operations

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

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

- 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

```
def getNumbers():
    nums = []      # start with an empty list

    # sentinel loop to get numbers
    xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        nums.append(x)    # add this value to the list
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    return nums
```

- 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

```
def mean(nums):
    sum = 0.0
    for num in nums:
        sum = sum + num
    return sum / len(nums)
```


Statistics with Lists

- The next function to tackle is the standard deviation.
- In order to determine the standard deviation, we need to know the mean.
  - Should we recalculate the mean inside of `stdDev`?
  - Should the mean be passed as a parameter to `stdDev`?

Statistics with Lists

- Recalculating the mean inside of `stdDev` is inefficient if the data set is large.
- Since our program is outputting both the mean and the standard deviation, let's compute the mean and pass it to `stdDev` as a parameter.

Statistics with Lists

```
def stdDev(nums, xbar):
    sumDevSq = 0.0
    for num in nums:
        dev = xbar - num
        sumDevSq = sumDevSq + dev * dev
    return sqrt(sumDevSq/(len(nums)-1))
```

- The summation from the formula is accomplished with a loop and accumulator.
- `sumDevSq` stores the running sum of the squares of the deviations.

Statistics with Lists

- We don't have a formula to calculate the median. We'll need to come up with an algorithm to pick out the middle value.
- First, we need to arrange the numbers in ascending order.
- Second, the middle value in the list is the median.
- If the list has an even length, the median is the average of the middle two values.

Statistics with Lists

- Pseudocode -

```
sort the numbers into ascending order
if the size of the data is odd:
    median = the middle value
else:
    median = the average of the two middle values
return median
```

Statistics with Lists

```
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
```


Statistics with Lists

- With these functions, the main program is pretty simple!

```
def main():
    print 'This program computes mean, median and standard deviation.'

    data = getNumbers()
    xbar = mean(data)
    std = stdDev(data, xbar)
    med = median(data)

    print '\nThe mean is', xbar
    print 'The standard deviation is', std
    print 'The median is', med
```


Statistics with Lists

- Statistical analysis routines might come in handy some time, so let's add the capability to use this code as a module by adding:

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


Lists of Objects

- All of the list examples we've looked at so far have involved simple data types like numbers and strings.
- We can also use lists to store more complex data types, like our student information from chapter ten.

Lists of Objects

- Our grade processing program read through a file of student grade information and then printed out information about the student with the highest GPA.
- A common operation on data like this is to sort it, perhaps alphabetically, perhaps by credit-hours, or even by GPA.

Lists of Objects

- Let's write a program that sorts students according to GPA using our `Sutdent` class from the last chapter.

```
Get the name of the input file from the user
Read student information into a list
Sort the list by GPA
Get the name of the output file from the user
Write the student information from the list into a file
```

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

- Let's also write a function to write the list of students back to a file.
- Each line should contain three pieces of information, separated by tabs: name, credit hours, and quality points.

```
def writeStudents(students, filename):
    # students is a list of Student objects
    outfile = open(filename, 'w')
    for s in students:
        outfile.write("%s\t%f\t%f\n" %
                  (s.getName(),s.getHours(),s.getQPoints()))
    outfile.close()
```


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

```
>>> cmp(1,2)
-1
>>> cmp(2,1)
1
>>> cmp("a","b")
-1
>>> cmp(1,1.0)
0
>>> cmp("a",5)
1
```

- For types that aren't directly comparable, the standard ordering uses rules like "numbers always comes before strings."

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.
- ```
  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.
- `data.sort(cmpGPA)`

Lists of Objects

- `data.sort(cmpGPA)`
- Notice that we didn't put `()`'s after the function call `cmpGPA`.
- This is because we don't want to *call* `cmpGPA`, but rather, we want to send `cmpGPA` to the sort method to use as a comparator.

Lists of Objects

```
# gpasort.py
#    A program to sort student information into GPA
#       order.


from gpa import Student, makeStudent

def readStudents(filename):
    infile = open(filename, 'r')
    students = []
    for line in infile:
        students.append(makeStudent(line))
    infile.close()
    return students

def writeStudents(students, filename):
    outfile = open(filename, 'w')
    for s in students:
        outfile.write("%s\t%f\t%f\n" %
                (s.getName(), s.getHours(),
                                s.getQPoints()))
    outfile.close()

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

def main():
    print "This program sorts student grade
        information by GPA"
    filename = raw_input("Enter the name of the data
        file: ")
    data = readStudents(filename)
    data.sort(cmpGPA)
    filename = raw_input("Enter a name for the
        output file: ")
    writeStudents(data, filename)
    print "The data has been written to", filename

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


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`, …

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:
  ```
  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.

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.
- ```
  pips = []
  pips.append(self.__makePip(cx-offset,cy-offset)
  pips.append(self.__makePip(cx-offset,cy)
  …
  self.pips = pips
  ```

Designing with Lists and Classes

- An even more straightforward approach is to create the list directly.

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

Designing with Lists and Classes

- 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!

Designing with Lists and Classes

- 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)
```


Designing with Lists and Classes

- 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!

```
Loop through pips and turn them all off
Determine the list of pip indexes to turn on
Loop through the list of indexes - turn on those pips
```


Designing with Lists and Classes

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


Designing with Lists and Classes

- We can do even better!
- The correct set of pips is determined by `value`. We can make this process *table-driven* instead.
- 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:

```
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)
```


Designing with Lists and Classes

```
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.

Designing with Lists and Classes

```
# dieview2.py
#    A widget for displaying the value of a die.
#    This version uses lists to simplify keeping track of
#    pips.

class DieView:
    """ DieView is a widget that displays a graphical
      representation
    of a standard six-sided die."""

    def __init__(self, win, center, size):
        """Create a view of a die, e.g.:
           di = GDie(myWin, Point(40,50), 20)
        creates a die centered at (40,50) having sides
        of length 20."""

        # first define some standard values
        self.win = win
        self.background = "white" # color of die face
        self.foreground = "black" # color of the pips
        self.psize = 0.1 * size   # radius of each pip
        hsize = size / 2.0        # half of size
        offset = 0.6 * hsize      # distance from center to
    outer pips

        # create a square for the face
        cx, cy = center.getX(), center.getY()
        p1 = Point(cx-hsize, cy-hsize)
        p2 = Point(cx+hsize, cy+hsize)
        rect = Rectangle(p1,p2)
        rect.draw(win)
        rect.setFill(self.background)

        # Create 7 circles for standard pip locations
        self.pips = [ self.__makePip(cx-offset, cy-offset),
                      self.__makePip(cx-offset, cy),
                      self.__makePip(cx-offset, cy+offset),
                      self.__makePip(cx, cy),
                      self.__makePip(cx+offset, cy-offset),
                      self.__makePip(cx+offset, cy),
                      self.__makePip(cx+offset, cy+offset) ]

        # Create a table for which pips are on for each
      value
        self.onTable = [ [], [3], [2,4], [2,3,4],
            [0,2,4,6], [0,2,3,4,6], [0,1,2,4,5,6] ]

        self.setValue(1)

    def __makePip(self, x, y):
        """Internal helper method to draw a pip at (x,y)"""
        pip = Circle(Point(x,y), self.psize)
        pip.setFill(self.background)
        pip.setOutline(self.background)
        pip.draw(self.win)
        return pip

    def setValue(self, value):
        """ Set this die to display value."""
        # Turn all the pips off
        for pip in self.pips:
            pip.setFill(self.background)

        # Turn the appropriate pips back on
        for i in self.onTable[value]:
            self.pips[i].setFill(self.foreground)
```


Designing with Lists and Classes

- Lastly, this example showcases the advantages of encapsulation.
  - We have improved the implementation of the `dieView` class, but we have not changed the set of methods it supports.
  - We can substitute this **new** version of the class without having to modify any other code!
  - Encapsulation allows us to build complex software systems as a set of "pluggable modules."

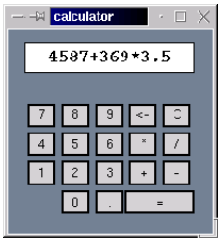
Case Study: Python Calculator

- 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

A Calculator as an Object



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.

```
# create list of buttons
        # start with all the standard sized buttons
        # bSpecs gives center coords and label of buttons
        bSpecs = [(2,1,'0'), (3,1,'.'),
                  (1,2,'1'), (2,2,'2'), (3,2,'3'), (4,2,'+'), (5,2,'-'),
                  (1,3,'4'), (2,3,'5'), (3,3,'6'), (4,3,'*'), (5,3,'/'),
                  (1,4,'7'), (2,4,'8'), (3,4,'9'), (4,4,'<-'),(5,4,'C')]
        self.buttons = []
        for cx,cy,label in bSpecs:
            self.buttons.append(Button(self.win,Point(cx,cy),.75,.75,label))
        # create the larger = button
        self.buttons.append(Button(self.win, Point(4.5,1), 1.75, .75, "="))
        # activate all buttons
        for b in self.buttons:
            b.activate()
```

- `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.

Constructing the Interface

- 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.

Constructing the Interface

- Here's the code to create the display

```
bg = Rectangle(Point(.5,5.5), Point(5.5,6.5))
bg.setFill('white')
bg.draw(self.win)
text = Text(Point(3,6), "")
text.draw(self.win)
text.setFace("courier")
text.setStyle("bold")
text.setSize(16)
self.display = text
```


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

Processing Buttons

```
try:
    result = eval(text)
except:
    result = 'ERROR'
    self.display.setText(str(result))
```

- 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

- 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

- 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.

Dictionary Basics

- 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

- 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`
    `{'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`
    `{'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, seperated by a space.

Dictionary Operations

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


Dictionary Operations

Method	Meaning

`<dict>.has_key(<key>)`
 	
`<key> in <dict>`
	Returns true if dictionary contains the specified key, false otherwise.
	Same as `has_key`

`<dict>.keys()`	Returns a list of the keys.
`<dict>.values()`	Returns a list of the values.
`<dict>.items()`	Returns a list of tuples `(key, value)` representing the key-value pairs.
`<dict>.get(<key>, <default>)`	If dictionary has `key`, returns its value; otherwise returns `default`
`del <dict> [<key>]`	Deletes the specified entry.
`<dict>.clear()`	Deletes all entries.

Dictionary Operations

```
>>> 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

- 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).

Example Program: Word Frequency

- At the highest level, this is simply a multi-accumulator problem, where we have separate accumulators for each possible word in the document.
- We can loop through each word of a document, adding one to the count of the accumulator for that word.
- Without dictionaries, we'd need hundreds, if not thousands of accumulators!

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`:

```
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):
    counts[w] = counts[w] + 1
else:
    counts[w] = 1
```


Example Program: Word Frequency

- A more elegant approach:

```
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").

Example Program: Word Frequency

```
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:

```
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

```
# get list of words that appear in document
uniqueWords = counts.keys()

# put list of words in alphabetical order
uniqueWords.sort()

# print words and associated counts
for w in uniqueWords:
    print w, counts[w]
```

- This will work, but its results may not be that interesting for large documents with many words used just a few times.

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 start by getting a list of key-value pairs using the `items` method for dictionaries:

```
items = counts.items()
```

- Items will be a list of tuples, e.g.

```
[('foo',5), ('bar',7), ('spam',376), ...]
```

- If we tried to sort items with `items.sort`, they would be sorted by the left-most element of the tuple, which is not what we want! We'd just end up with alphabetical order again!

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.

```
def compareItems((w1, c1), (w2, c2)):
    if c1 > c2:
        return -1
    elif c1 == c2:
        return cmp(w1, w2)
    else:
        return 1
```


Example Program: Word Frequency

```
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`.

Sample Program: Word Frequency

```
def compareItems((w1, c1), (w2, c2)):
      if c1 > c2:
          return -1
      elif c1 == c2:
          return cmp(w1, w2)
      else:
          return 1
```

- If the two counts are equal, we break the tie with the words' alphabetical order. This has the advantage that words with the same count will print out in alphabetical order.

Sample Program: Word Frequency

```
def compareItems((w1, c1), (w2, c2)):
      if c1 > c2:
          return -1
      elif c1 == c2:
          return cmp(w1, w2)
      else:
          return 1
```

- Lastly, if the second count is larger, we return `1`.
- To sort our items:

```
items.sort(compareItems)
```


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.

Sample Program: Word Frequency

```
# wordfreq.py
#     Program to analyze the frequency of words in a text file.
#     Illustrates Python dictionaries

import string

def compareItems((w1,c1), (w2,c2)):
    if c1 > c2:
        return -1
    elif c1 == c2:
        return cmp(w1, w2)
    else:
        return 1

def main():
    print "This program analyzes word frequency in a file"
    print "and prints a report on the n most frequent words.\n"
```


Sample Program: Word Frequency

```
# get the sequence of words from the file
    fname = raw_input("File to analyze: ")
    text = open(fname,'r').read()
    text = string.lower(text)
    for ch in '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~':
        text = string.replace(text, ch, ' ')
    words = string.split(text)

    # construct a dictionary of word counts
    counts = {}
    for w in words:
        counts[w] = counts.get(w,0) + 1

    # output analysis of n most frequent words.
    n = input("Output analysis of how many words? ")
    items = counts.items()
    items.sort(compareItems)
    for i in range(n):
        print "%-10s%5d" % items[i]
```