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:

```python
# 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 = input("Enter a number (<Enter> to quit) >> ")
    while xStr != ":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("The 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

\[
\sum (\text{data} - \bar{x})^2 = (6.8 - 2)^2 + (6.8 - 4)^2 + (6.8 - 6)^2 + (6.8 - 9)^2 + (6.8 - 13)^2 = 149.6
\]

\[
s = \sqrt{\frac{149.6}{5 - 1}} = \sqrt{37.4} = 6.11
\]
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:

  ```python
  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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;seq&gt; + &lt;seq&gt;</code></td>
<td>Concatenation</td>
</tr>
<tr>
<td><code>&lt;seq&gt; * &lt;int-expr&gt;</code></td>
<td>Repetition</td>
</tr>
<tr>
<td><code>&lt;seq&gt;[:]</code></td>
<td>Indexing</td>
</tr>
<tr>
<td><code>len(&lt;seq&gt;)</code></td>
<td>Length</td>
</tr>
<tr>
<td><code>&lt;seq&gt;[:</code></td>
<td>Slicing</td>
</tr>
<tr>
<td><code>for &lt;var&gt; in &lt;seq&gt;:</code></td>
<td>Iteration</td>
</tr>
<tr>
<td><code>&lt;expr&gt; in &lt;seq&gt;</code></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.

```python
>>> lst = [1, 2, 3, 4]
>>> 3 in lst
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']
>>> 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
```
Lists are often built up one piece at a time using `append`.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;list&gt;.append(x)</code></td>
<td>Add element x to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.sort()</code></td>
<td>Sort (order) the list. A comparison function may be passed as a parameter.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.reverse()</code></td>
<td>Reverse the list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.index(x)</code></td>
<td>Returns index of first occurrence of x.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.insert(i, x)</code></td>
<td>Insert x into list at index i.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.count(x)</code></td>
<td>Returns the number of occurrences of x in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.remove(x)</code></td>
<td>Deletes the first occurrence of x in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.pop(i)</code></td>
<td>Deletes the ith element of the list and returns its value.</td>
</tr>
</tbody>
</table>
List Operations

```python
>>> 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)
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

- \texttt{>>> myList} = [34, 26, 0, 10]
  \texttt{>>> del myList[1]}
  \texttt{>>> myList}
  [34, 0, 10]
  \texttt{>>> del myList[1:3]}
  \texttt{>>> myList}
  [34]

- \texttt{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.
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.
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

```python
def getNumbers():
    nums = []  # start with an empty list
    
    # sentinel loop to get numbers
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != ":
        x = eval(xStr)
        nums.append(x)  # add this value to the list
        xStr = 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:

```python
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

```python
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`?
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.
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
def median(nums):
    nums.sort()
    size = len(nums)
    midPos = size // 2
    if size % 2 == 0:
        median = (nums[midPos] + nums[midPos-1]) / 2
    else:
        median = nums[midPos]
    return median
With these functions, the main program is pretty simple!

```python
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:

```python
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 Student 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.

```python
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:
            print((s.getName(), s.getHours(), s.getQPoints(),
                  sep="\t", file=outfile)
        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

- To make sorting work with our objects, we need to tell `sort` how the objects should be compared.
- Can supply a function to produce the key for an object using `<list>.sort(key=<somefunc>)`
- To sort by GPA, we need a function that takes a Student as parameter and returns the student's GPA.
Lists of Objects

- def use_gpa(aStudent):
  return aStudent.gpa()

- We can now sort the data by calling sort with the key function as a keyword parameter.

- data.sort(key=use_gpa)
Lists of Objects

- `data.sort(key=use_gpa)`
- Notice that we didn’t put ()’s after the function name.
- This is because we don’t want to call `use_gpa`, but rather, we want to send `use_gpa` to the sort method.
Actually, defining \texttt{use\_gpa} was unnecessary.

The \texttt{gpa} method in the \texttt{Student} class is a function that takes a student as a parameter (formally, \texttt{self}) and returns GPA.

Can use it:

\begin{verbatim}
data.sort(key=Student.gpa)
\end{verbatim}
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:
        print(s.getName(), s.getHours(), s.getQPoints(), sep="\t", file=outfile)
    outfile.close()

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

if __name__ == '__main__':
    main()
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, \( \text{pip1, pip2, pip3, ...} \).
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.
One approach is to start with an empty list of pips and build up the list one pip at a time.

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

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

This cut our previous code from seven lines to two!
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:
  ```python
  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
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.
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)
```
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

```python
# 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.:
d1 = 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.setFill(self.background)
rect.draw(self.win)

        # 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)
```

Python Programming, 2/e 62
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.

```python
def __init__(self):
    # create the window for the calculator
    win = GraphWin("calculator")
    win.setCoords(0,0,6,7)
    win.setBackground("slategray")
    self.win = win
```
Our next step is to create the buttons, reusing the button class.

```python
# create list of 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:
  \[(cx, cy, label) = \langle\text{next item from bSpecs}\rangle\]

- Each item in \textit{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 \textit{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

```python
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
```
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.

```python
def run(self):
    # Infinite 'event loop' to process button clicks.
    while True:
        key = self.getButton()
        self.processButton(key)
```
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.

```python
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
```
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:

```python
self.display.setText(text+key)
```
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.
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.