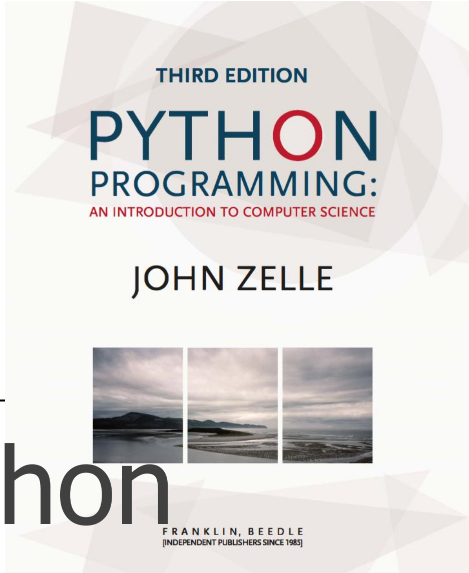

THIRD EDITION
PYTHON
PROGRAMMING:
AN INTRODUCTION TO COMPUTER SCIENCE
JOHN ZELLE
FRANKLIN, BEEDLE
[INDEPENDENT PUBLISHERS SINCE 1985]

An Introduction to Python Programming

Chapter 11: Data Collections

Objectives

- To be able to write programs that **use lists** to **manage a collection of information**.
- 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

- How to compute the **median** and **standard deviation** of the data.

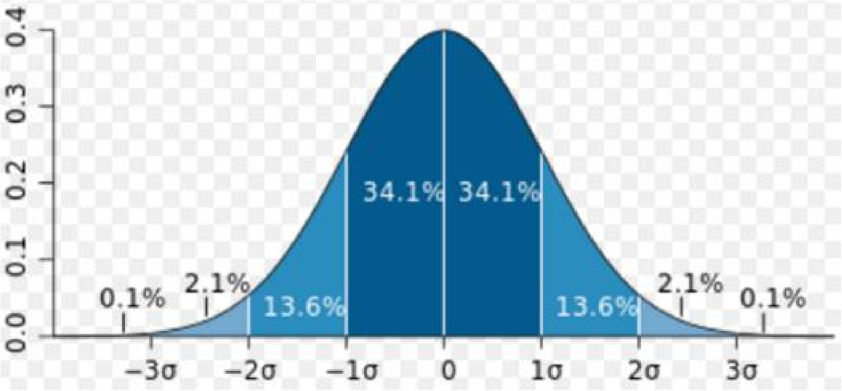
1, 3, 3, **6**, 7, 8, 9

Median = 6

1, 2, 3, **4**, **5**, 6, 8, 9

Median = (4 + 5) ÷ 2

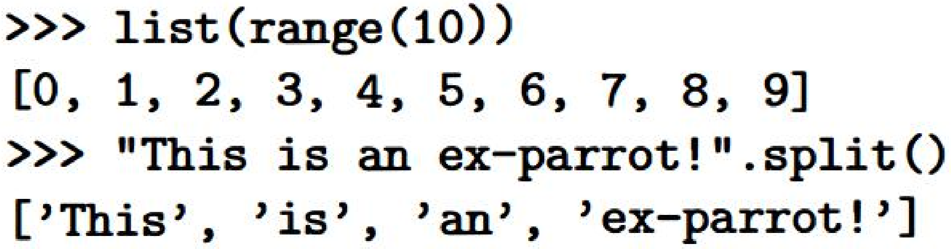
= 4.5



$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}}$$

Applying Lists

- We need a way to store and manipulate **an entire collection of numbers**.
- Use lists:



```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> "This is an ex-parrot!".split()
[’This’, ’is’, ’an’, ’ex-parrot!’]
```

Lists and Arrays

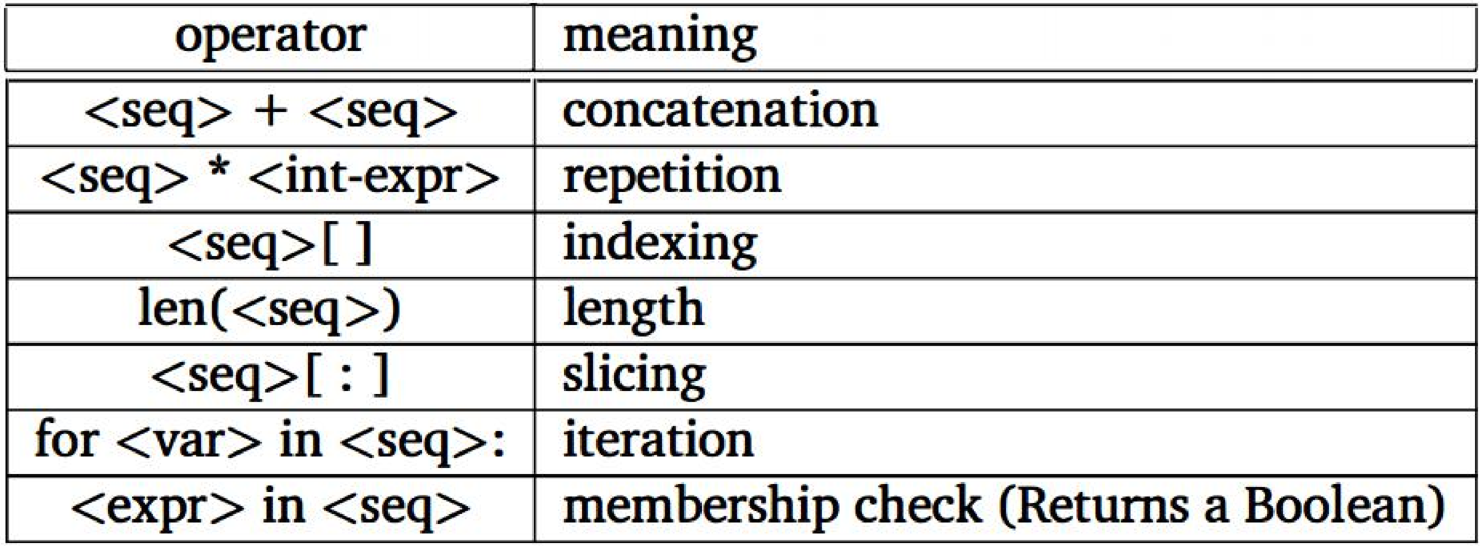
- A sequence of *n* numbers might be called *S*:

$$S = S_0, S_1, S_2, S_3, ..., S_{n-1}$$

- With a similar idea,we use **list** to represent an entire sequence and can be indexed.
  - In other languages, it is called an **array**,but arrays in other programming languages are generally **fixed size** and are also usually **homogeneous**.
  - Python lists are **mutable sequences of arbitrary objects.**

List Operations

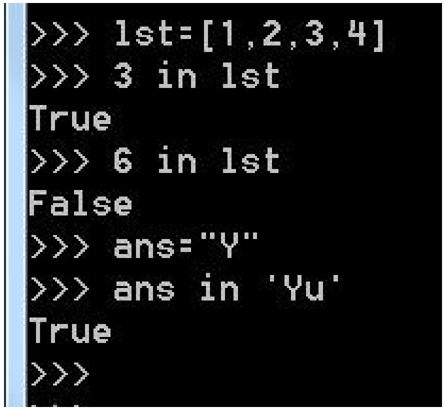
- All of the Python **built-in sequence operations** also apply to lists.



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 check (Returns a Boolean)

List Operations

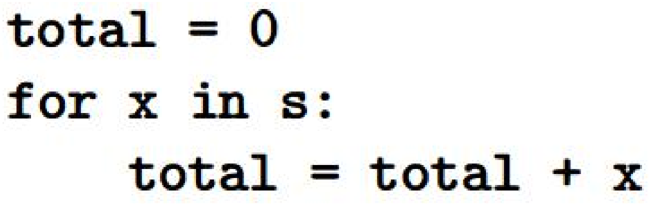
- Here are the examples checking for membership in lists and strings:



```
>>> lst=[1,2,3,4]
>>> 3 in lst
True
>>> 6 in lst
False
>>> ans="Y"
>>> ans in 'Yu'
True
>>>
```

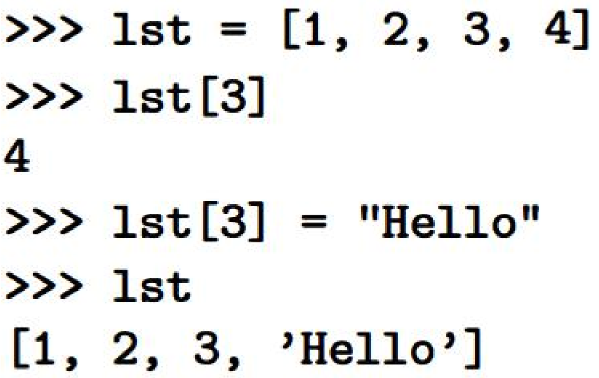
List Operations

- Suppose x is a list, the summing example will be written in this way:



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

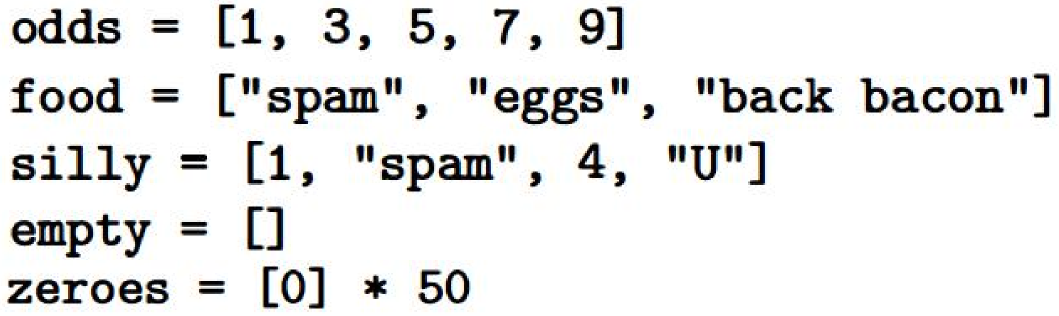
- Unlike strings, lists are **mutable**:



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

List Operations

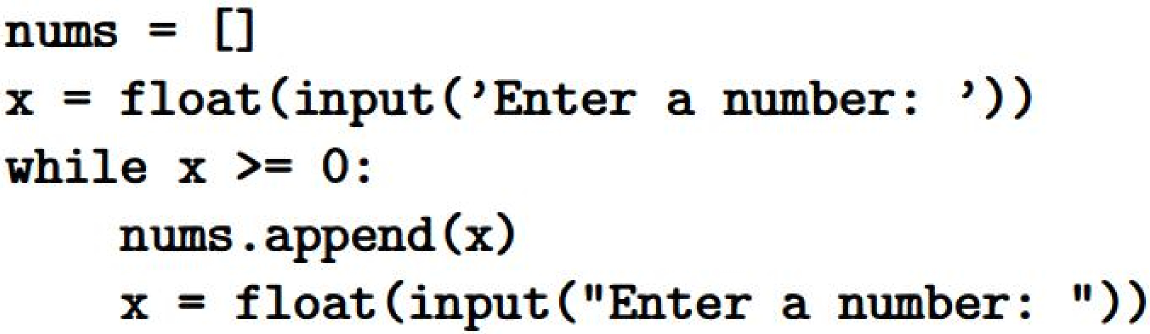
- A list can be created like this:



```
odds = [1, 3, 5, 7, 9]
food = ["spam", "eggs", "back bacon"]
silly = [1, "spam", 4, "U"]
empty = []
zeroes = [0] * 50
```

List Operations

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



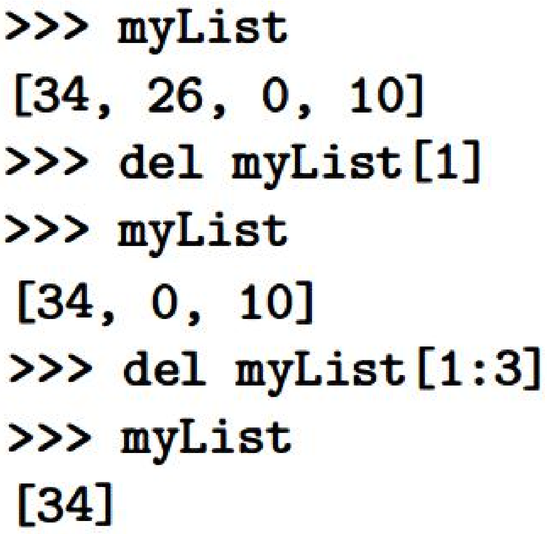
```
nums = []
x = float(input(’Enter a number: ’))
while x >= 0:
    nums.append(x)
    x = float(input("Enter a number: "))
```

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

- Individual items or entire slices can be **removed** from a list using the **del operator**:

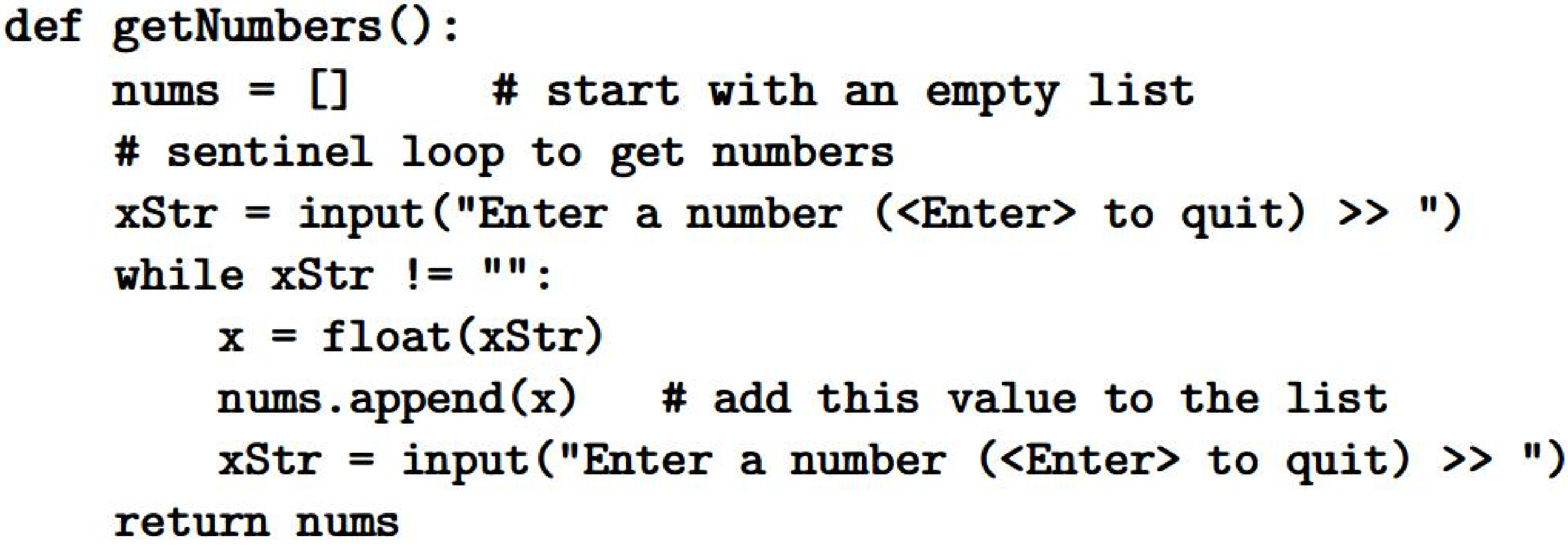


```
>>> myList
[34, 26, 0, 10]
>>> del myList[1]
>>> myList
[34, 0, 10]
>>> del myList[1:3]
>>> myList
[34]
```

del is not a list method, but a built-in operation that can be used on list items.

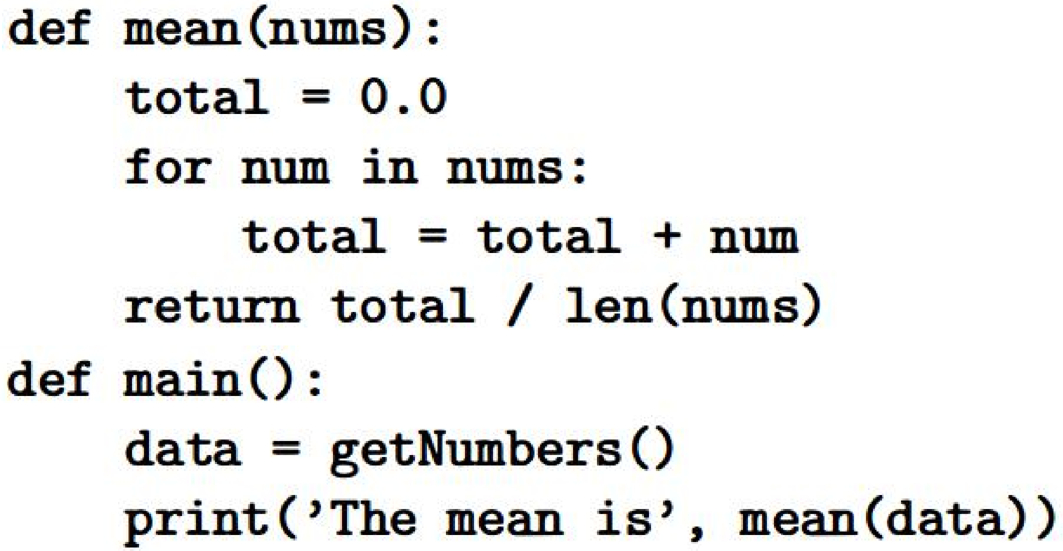
Statistics with Lists

- Let’s rewrite the program to **use lists** to find the **mean**.



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

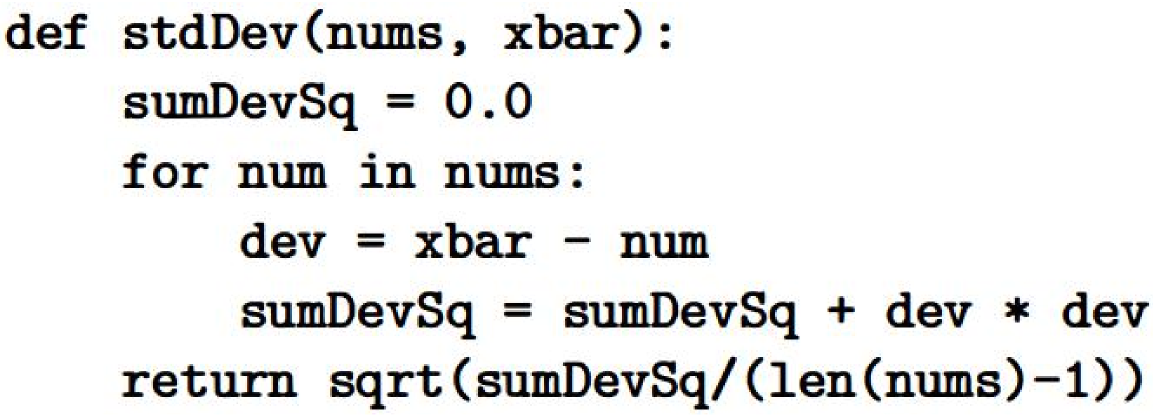
Statistics with Lists



```
def mean(nums):
    total = 0.0
    for num in nums:
        total = total + num
    return total / len(nums)
def main():
    data = getNumbers()
    print('The mean is', mean(data))
```

Statistics with Lists

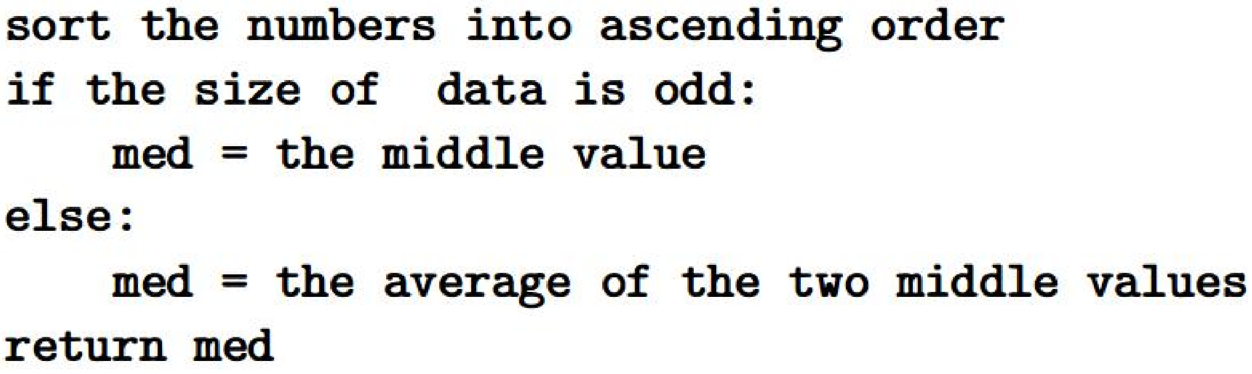
- The next function to tackle is **the standard deviation**.
  - We need **the mean** first.
  - The **mean** will be passed as a **parameter** to **`stdDev`**?



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

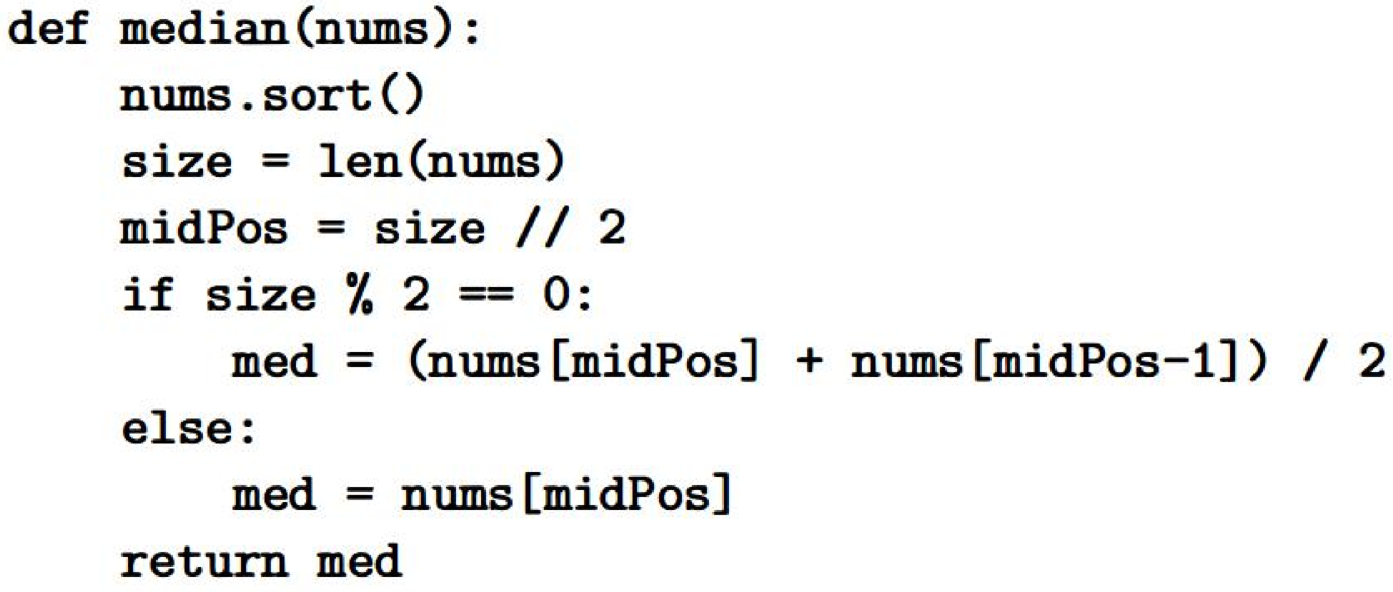
Statistics with Lists

- Finally we come to the **media function**.
- We need an algorithm to pick out the middle value.
  - First, **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**.



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

Statistics with Lists



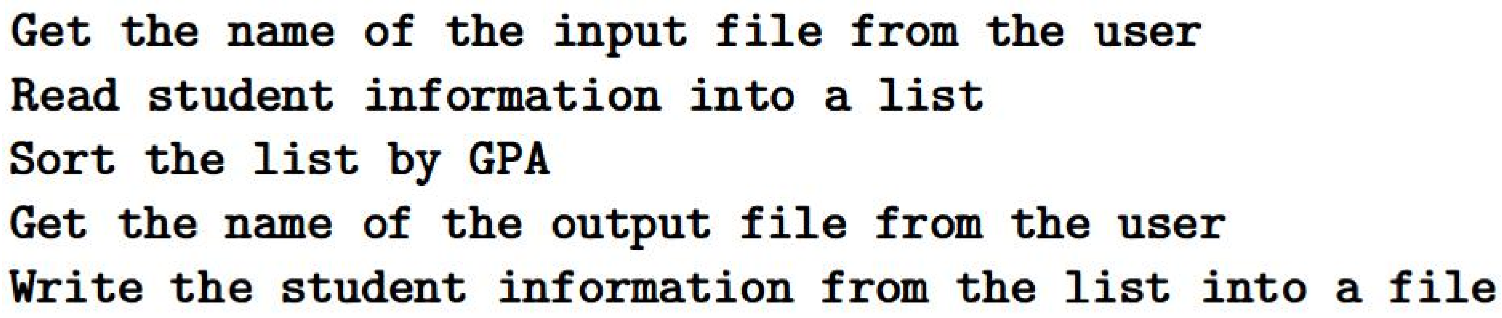
```
def median(nums):
    nums.sort()
    size = len(nums)
    midPos = size // 2
    if size % 2 == 0:
        med = (nums[midPos] + nums[midPos-1]) / 2
    else:
        med = nums[midPos]
    return med
```

Lists of Records

- Our **grade processing program** will printed out information about the student with the **highest GPA**.
- We can use **sort**(perhaps alphabetically, perhaps by credit-hours, or even by GPA).

Lists of Records

- The basic algorithm for our **program**:

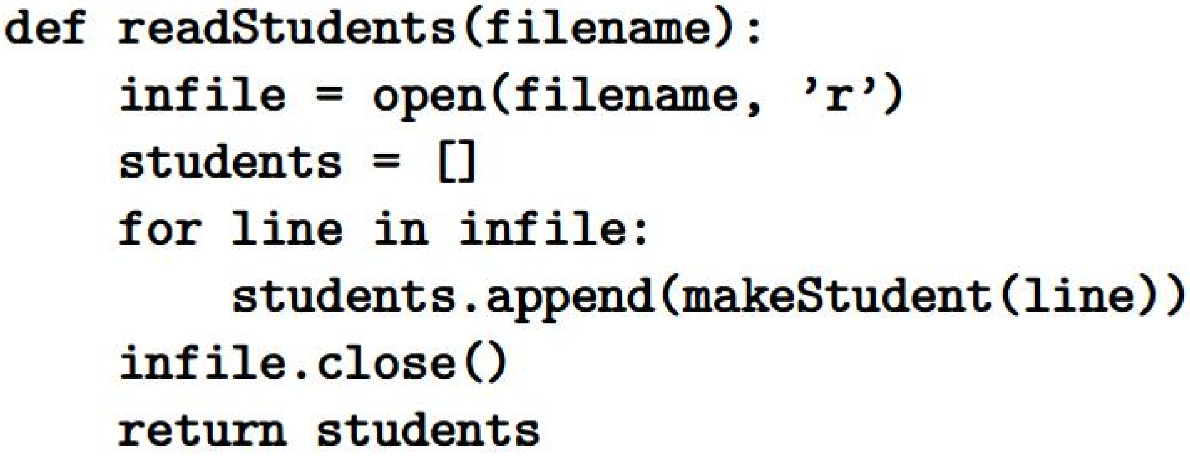


```
Get the name of the input file from the user
Read student information into a list
Sort the list by GPA
Get the name of the output file from the user
Write the student information from the list into a file
```

Student Class + List Processing

Lists of Records

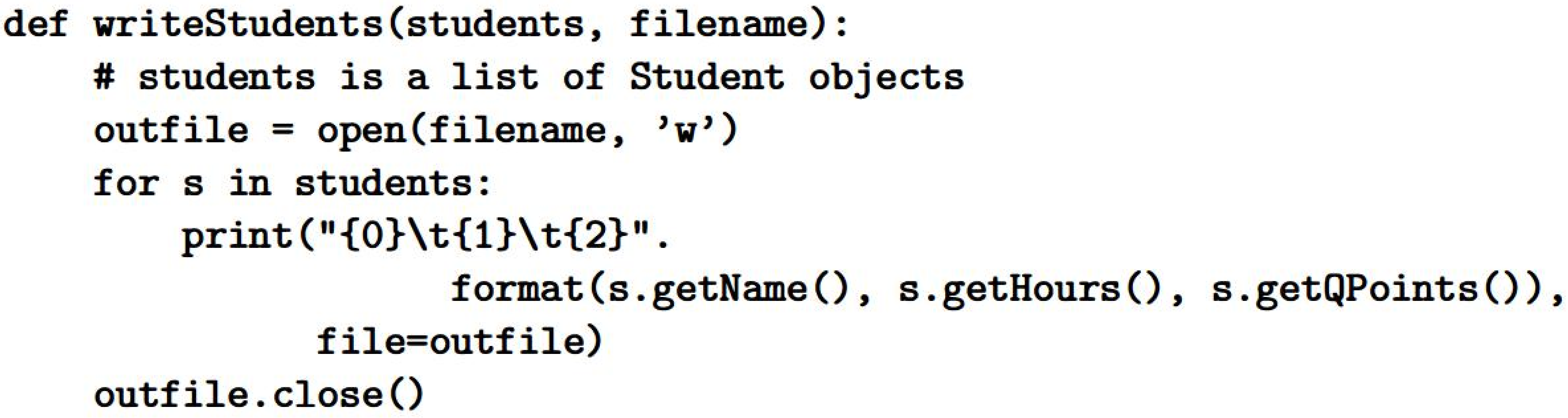
- Returns a **list of Student** objects from the file:



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

Lists of Records

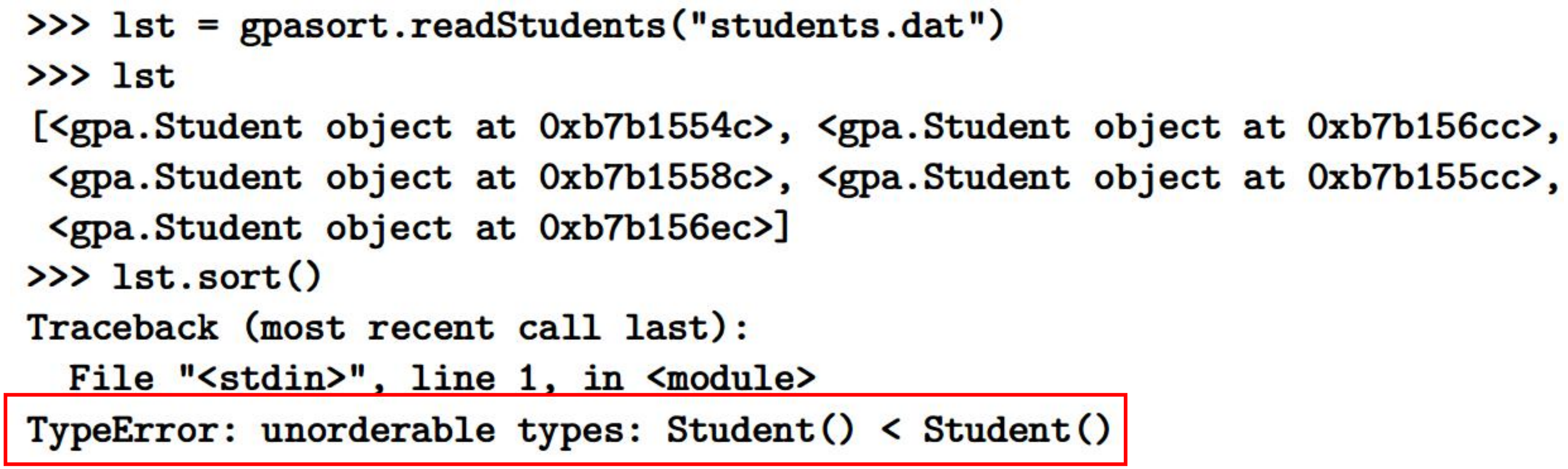
- We can also write a function that can **write the list of students back** out to a **file**.



```
def writeStudents(students, filename):
    # students is a list of Student objects
    outfile = open(filename, ’w’)
    for s in students:
        print("{0}\t{1}\t{2}".
                    format(s.getName(), s.getHours(), s.getQPoints()),
              file=outfile)
    outfile.close()
```

Lists of Records

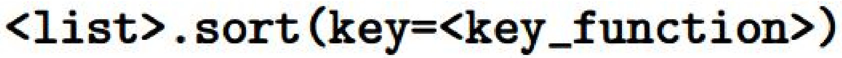
- How to **sort a list** that contains something **other than numbers**?



```
>>> lst = gpasort.readStudents("students.dat")
>>> lst
[<gpa.Student object at 0xb7b1554c>, <gpa.Student object at 0xb7b156cc>,
 <gpa.Student object at 0xb7b1558c>, <gpa.Student object at 0xb7b155cc>,
 <gpa.Student object at 0xb7b156ec>]
>>> lst.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Student() < Student()
```

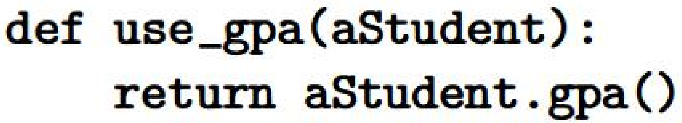
Lists of Records

- We want to order students,and the GPA is the **key**.
  - Specify the key that is used when sorting a list.



```
<list>.sort(key=<key_function>)
```

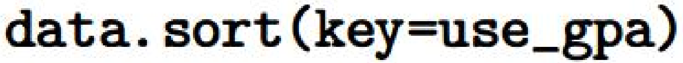
  - In our program,the **key_fuction** must take an item from the list as a parameter and returns the key value for that item.



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

Lists of Records

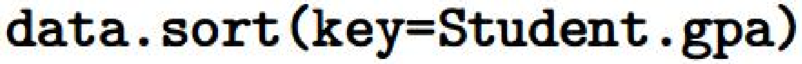
- Now we can use it to **sort a list** of Students with call to sort:



```
data.sort(key=use_gpa)
```

  - notice: **Not *use_gpa() ,***I do not want to call the function, just sending **use_gpa** to the sort method,and it will call this function anytime it needs.

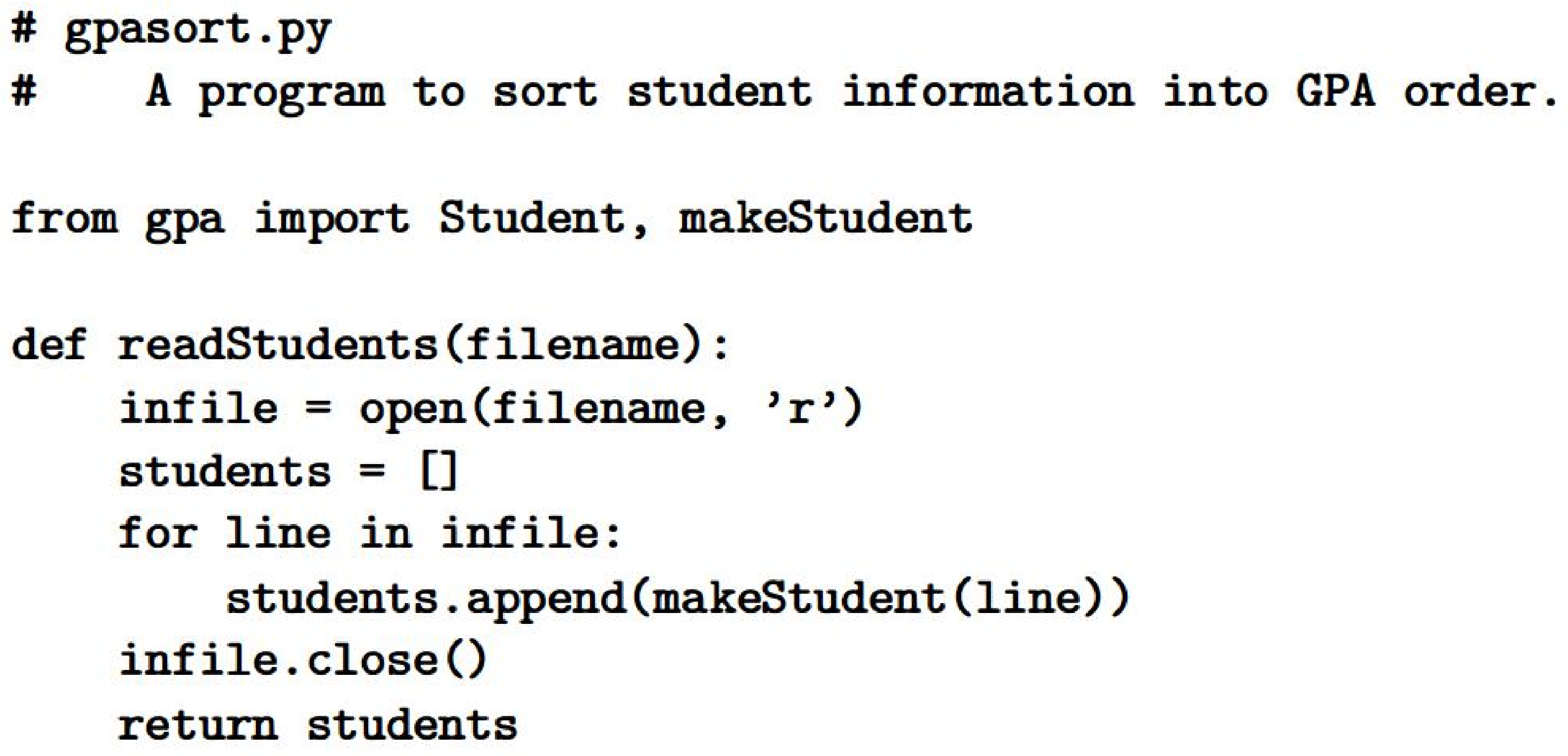
- In the **Student class**, **the GPA function** has been defined.



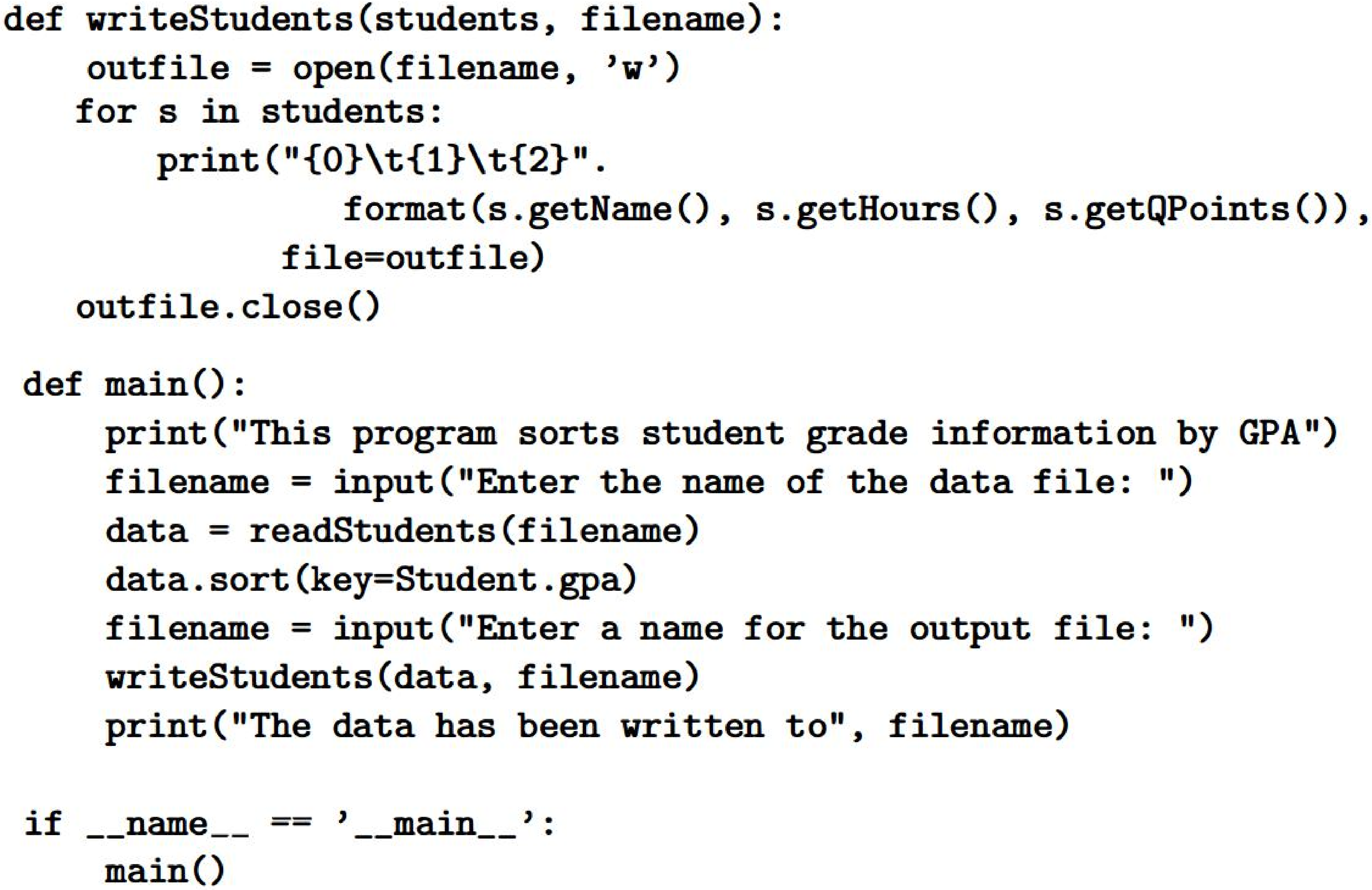
```
data.sort(key=Student.gpa)
```

Lists of Records

- Here's the completed code:



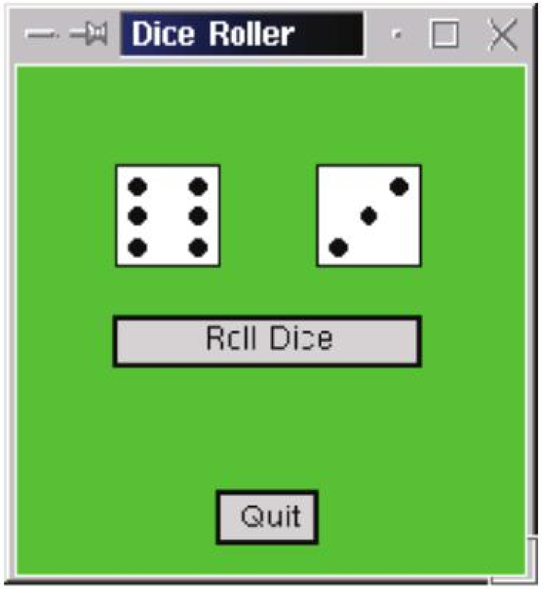
```
# gpasort.py
#    A program to sort student information into GPA order.

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("{0}\t{1}\t{2}".
         format(s.getName(), s.getHours(), s.getQPoints()),
       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(key=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()
```

Designing with Lists and Classes

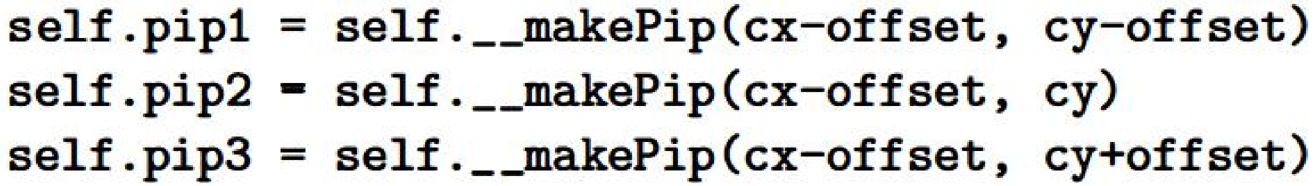
- For example:



- We used specific instance variables to keep track of each circle, **`pip1`**, **`pip2`**, **`pip3`**, …

Designing with
Lists and Classes

- Now,let's use a **collection of circle objects** stored as a **list**:put seven instance variables with a single **list** called **pips**.
- The pips were **created** with this sequence of statements inside **__init __**.

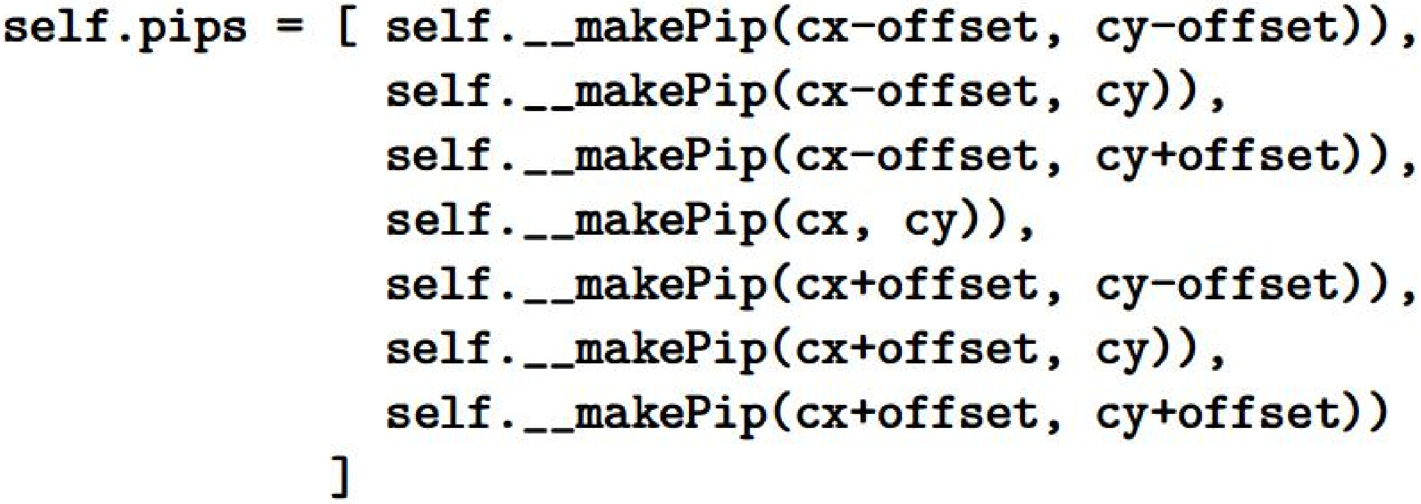


```
self.pip1 = self.__makePip(cx-offset, cy-offset)
self.pip2 = self.__makePip(cx-offset, cy)
self.pip3 = self.__makePip(cx-offset, cy+offset)
```

- **__makePip** is a local method of the **Die View class** to create a circle.

Designing with Lists and Classes

- Create a **list of pips**.

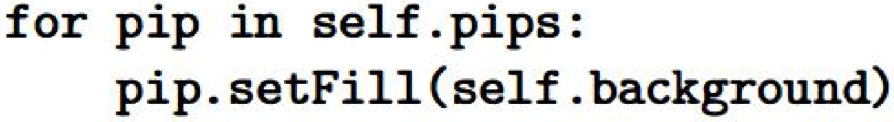


```
self.pips = [ self.__makePip(cx-offset, cy-offset)),
              self.__makePip(cx-offset, cy)),
              self.__makePip(cx-offset, cy+offset)),
              self.__makePip(cx, cy)),
              self.__makePip(cx+offset, cy-offset)),
              self.__makePip(cx+offset, cy)),
      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

Put one list element on each line.

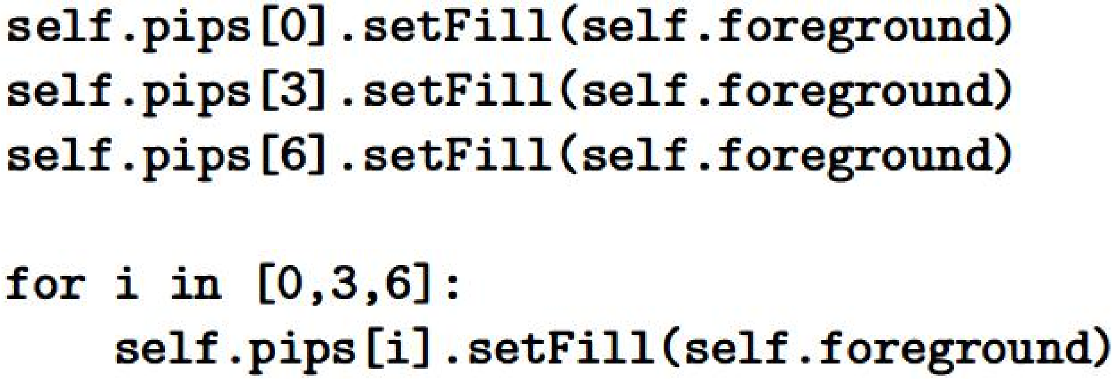
Designing with Lists and Classes

- Using pip list is much easier to **perform actions** on the **entire set**.



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

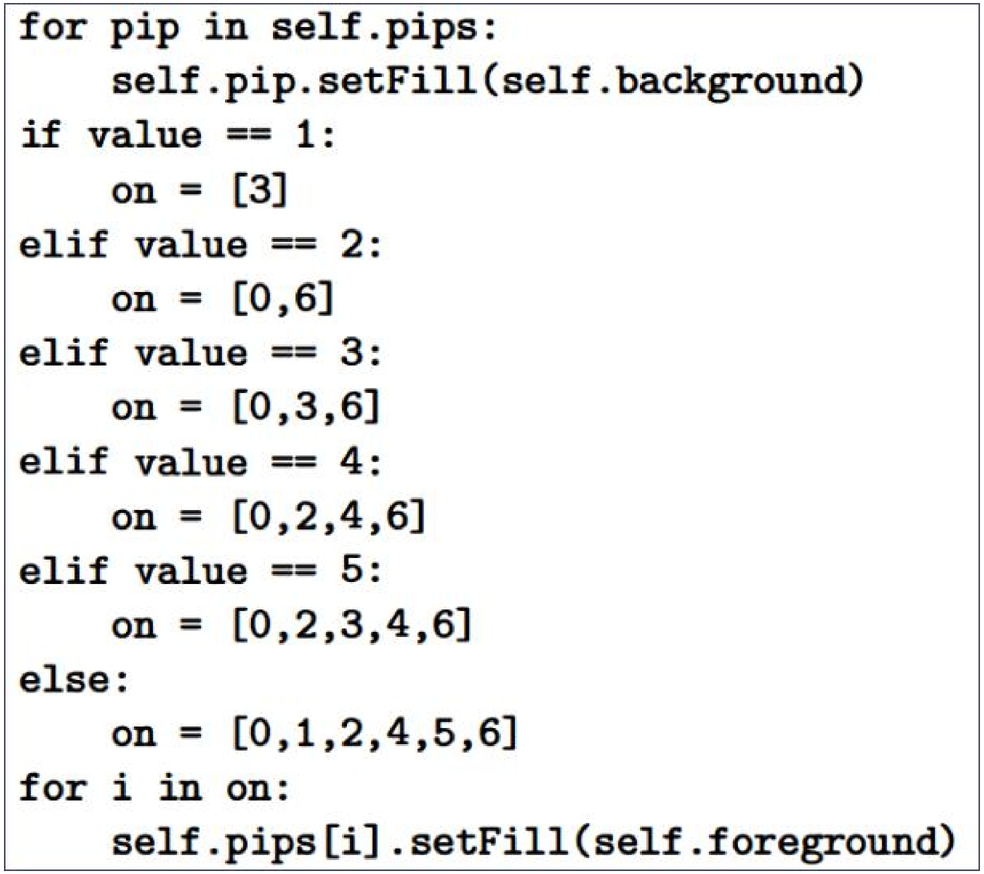
- We can turn a set of pips back on in two ways.



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

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

Designing with Lists and Classes

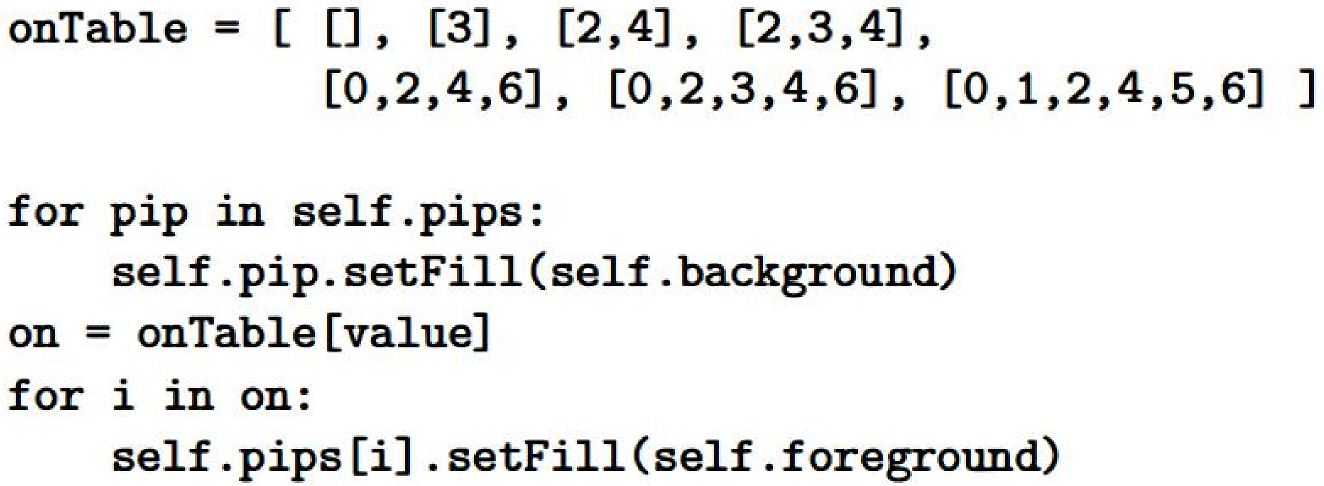
- The setValue method of the Die View class can be updated.



```
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 = [0,1,2,4,5,6]
for i in on:
    self.pips[i].setFill(self.foreground)
```

Designing with Lists and Classes

- Also, we could make this decision **table-driven** instead.



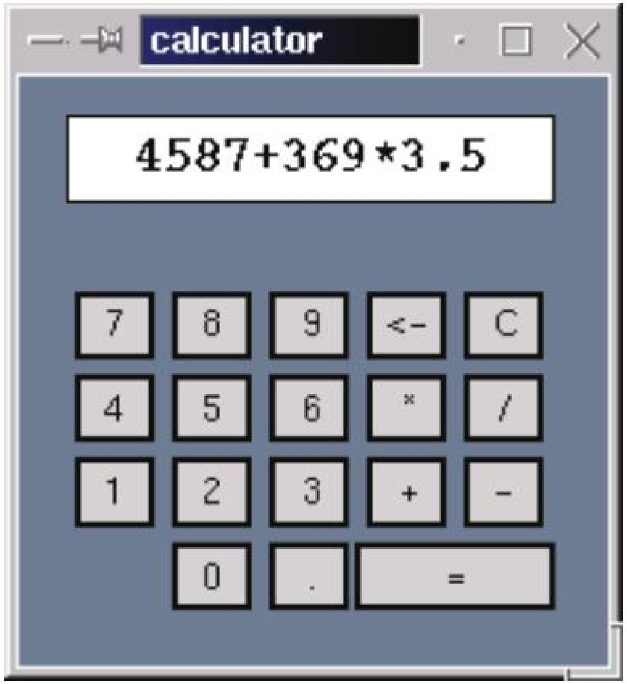
```
onTable = [ [], [3], [2,4], [2,3,4],
            [0,2,4,6], [0,2,3,4,6], [0,1,2,4,5,6] ]

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

- The **onTable** will remain **unchanged** throughout the life of any particular DieView.So the **defnition of onTable** can be put into **__init __** yelds.

Case Study: Python Calculator

- A program = a collection of data stuctures + algorithms operating on those data structures.
- This is like an object.
- Treat a Calculator as an Object!

Case Study: Python Calculator



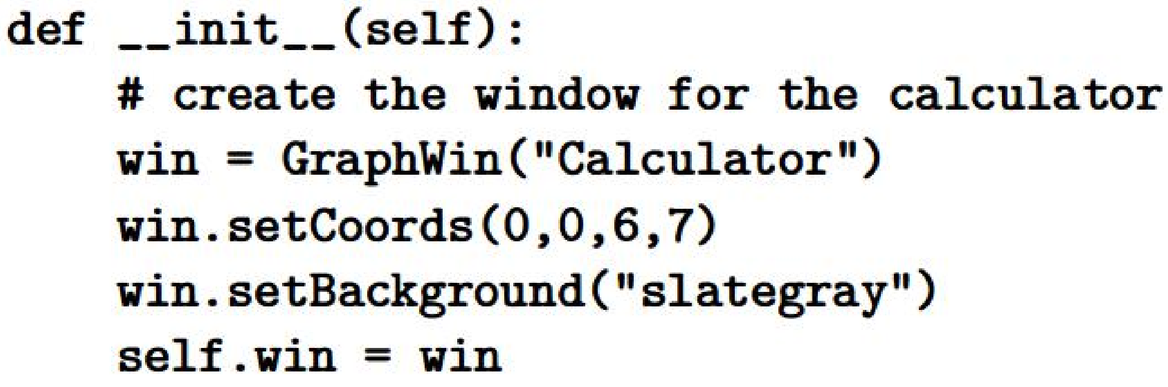
calculator

4587+369*3.5

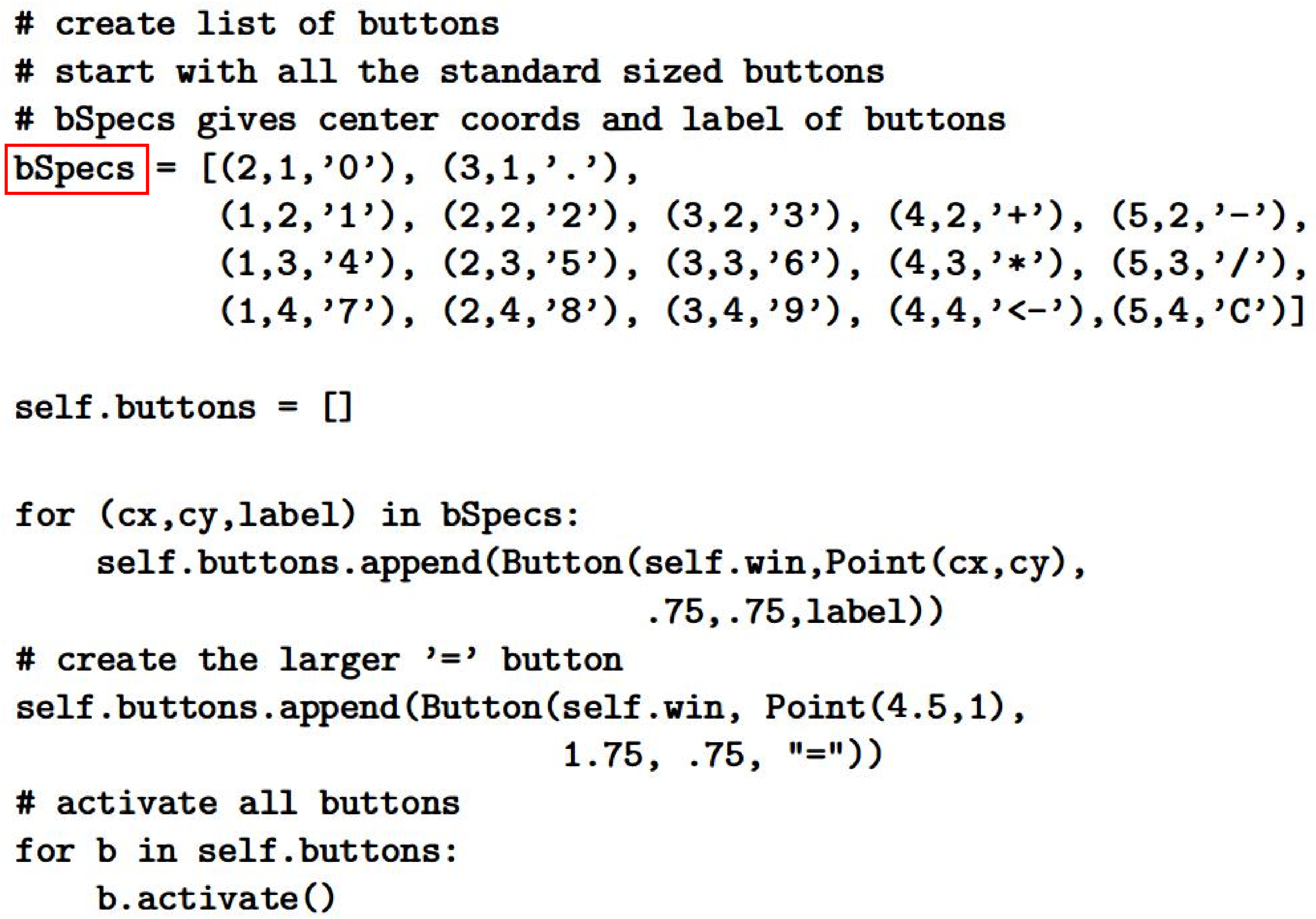
creating the interface & interacting with the user

Constructing the Interface

- The constructor for the Calculator class:
  - Create graphics window
  - Draw buttons.



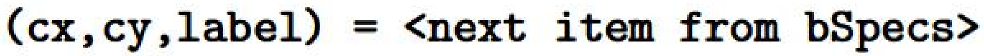
```
def __init__(self):
    # create the window for the calculator
    win = GraphWin("Calculator")
    win.setCoords(0,0,6,7)
    win.setBackground("slategray")
    self.win = win
```



```
# 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()
```

Constructing the Interface

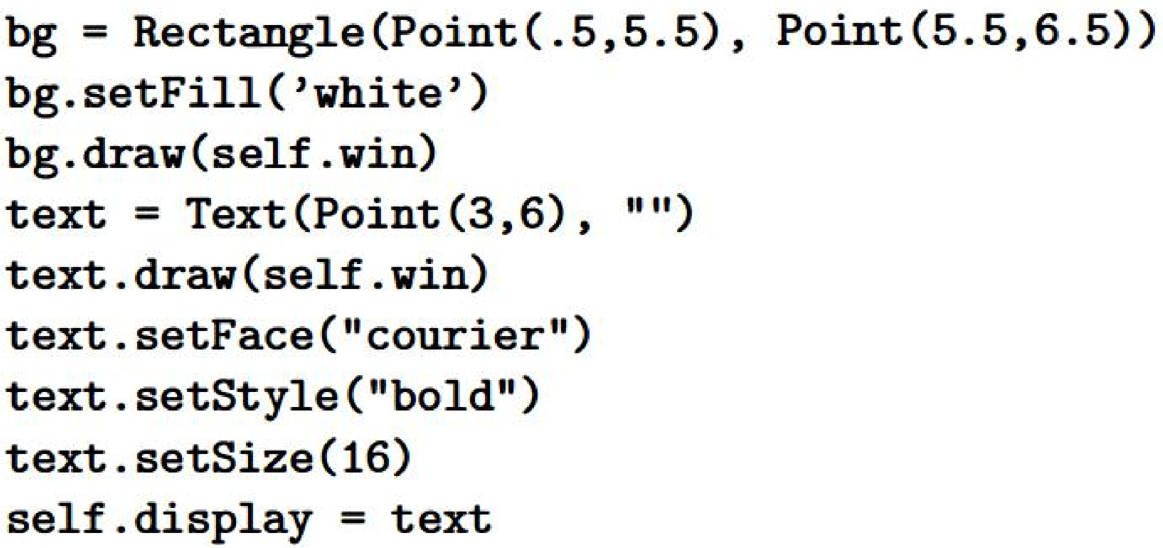
- ***bSpecs***,this list of **specifcations** is then used to create the buttons.
- Each specification is a ***tuple***.
  - A *tuple* looks like a list but uses **'()'** rather than '[]'.
  - Tuples are sequences that are **immutable**.
  - When a tuple of variables is used on the **left side** of an assignment, the components of the tuple on the **right side** are **unpacked into the variables** on the left side.



```
(cx,cy,label) = <next item from bSpecs>
```

Constructing the Interface

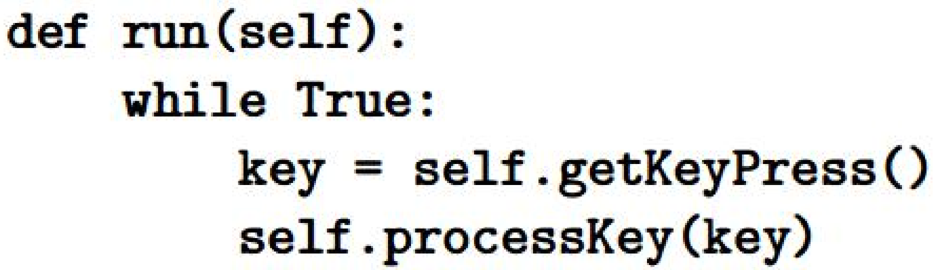
- Creating the **calculator display** is quite simple.



```
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
```

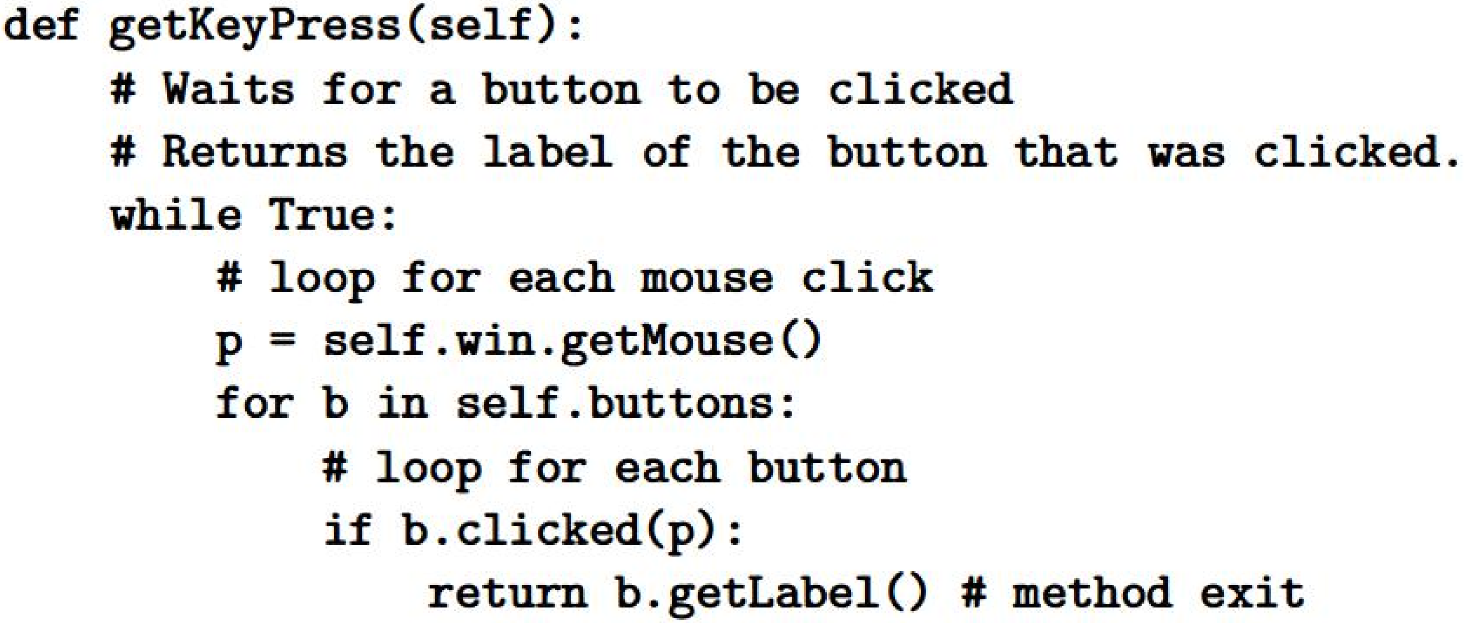
Processing Buttons

- Now that the interface is drawn, we need a **method** to get it running.



```
def run(self):
    while True:
        key = self.getKeyPress()
        self.processKey(key)
```

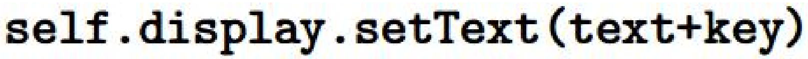
Processing Buttons



```
def getKeyPress(self):
    # Waits for a button to be clicked
    # Returns the label of the button that was clicked.
    while True:
        # loop for each mouse click
        p = self.win.getMouse()
        for b in self.buttons:
            # loop for each button
            if b.clicked(p):
                return b.getLabel() # method exit
```

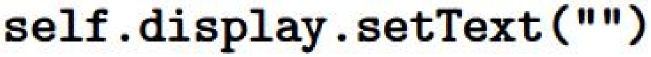
Processing Buttons

- in **processKey**:
  - □ A **digit** or **operator** is simply **appended to the display.**
  - □If key contains the **label** of the button and **text** contains the current contents of the display,the appropriate line of code will be:



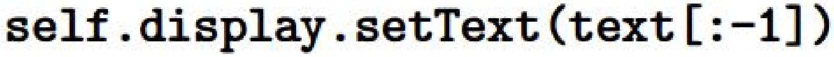
```
self.display.setText(text+key)
```

  - □The **clear** key blanks the display:



```
self.display.setText("")
```

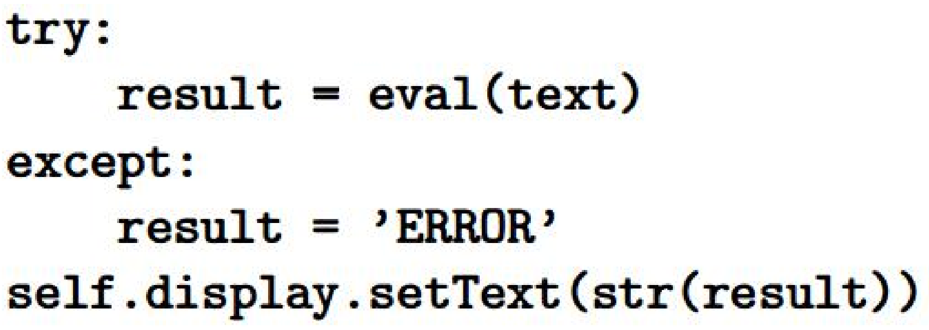
  - □The **backspace** strips off one character:



```
self.display.setText(text[:-1])
```

Processing Buttons

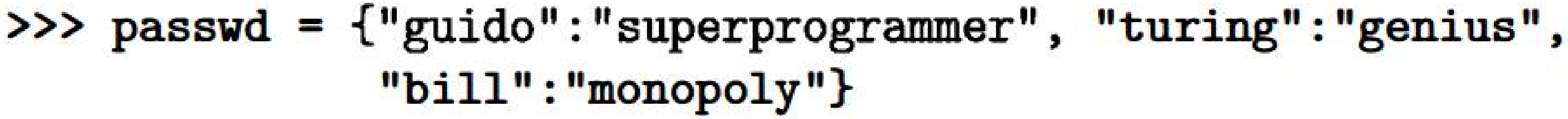
- The **equal key** causes the expression to be evaluated and the result displayed:



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

Non-sequential Collections: Dictionary Basics

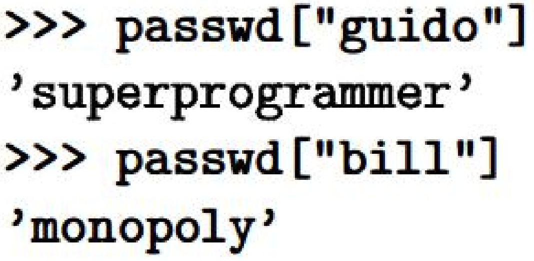
- After lists, a collection type called a **dictionary** is probably the most widely used.
- **Python dictionaries** are **mappings**.
- Dictionaries are **mutable**.
- A **dictionary** can be **created** in Python by listing **key-value pairs** inside of curly braces.



```
>>> passwd = {"guido":"superprogrammer", "turing":"genius",
              "bill":"monopoly"}
```

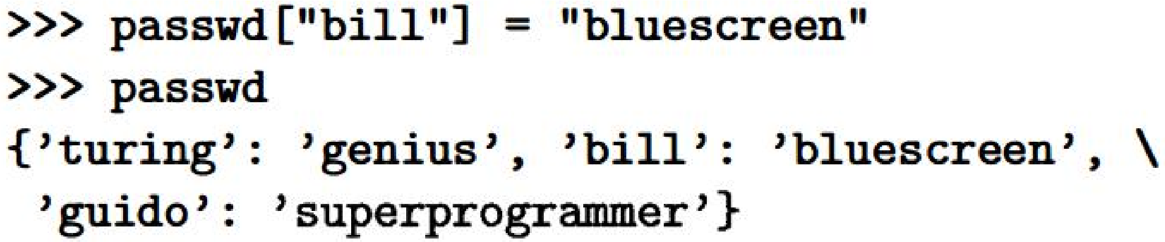
Non-sequential Collections: Dictionary Basics

- The **main use** for a dictionary is to **look up the value** associated with a particular key.



```
>>> passwd["guido"]
’superprogrammer’
>>> passwd["bill"]
’monopoly’
```

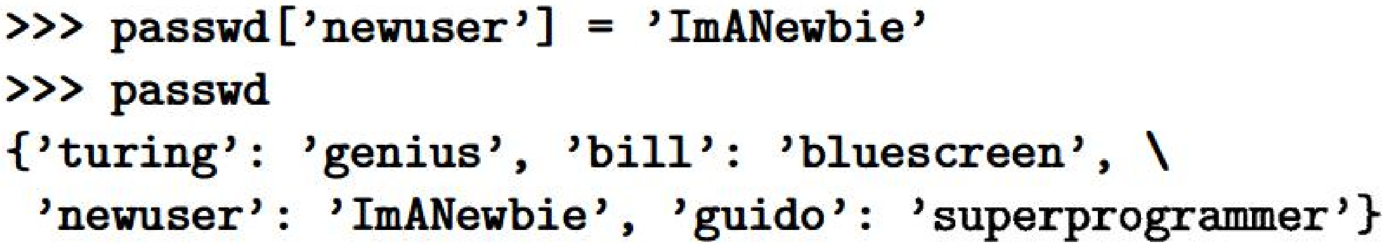
- Dictionaries can be changed.



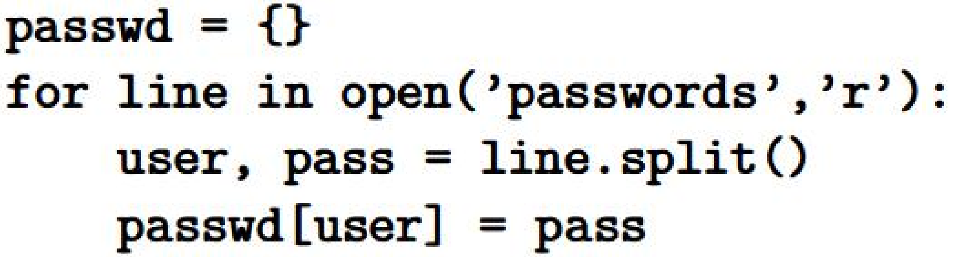
```
>>> passwd["bill"] = "bluescreen"
>>> passwd
{’turing’: ’genius’, ’bill’: ’bluescreen’, \
 ’guido’: ’superprogrammer’}
```

Non-sequential Collections: Dictionary Operations

- We can **expand** the dictionary by assigning a password for the new userame:

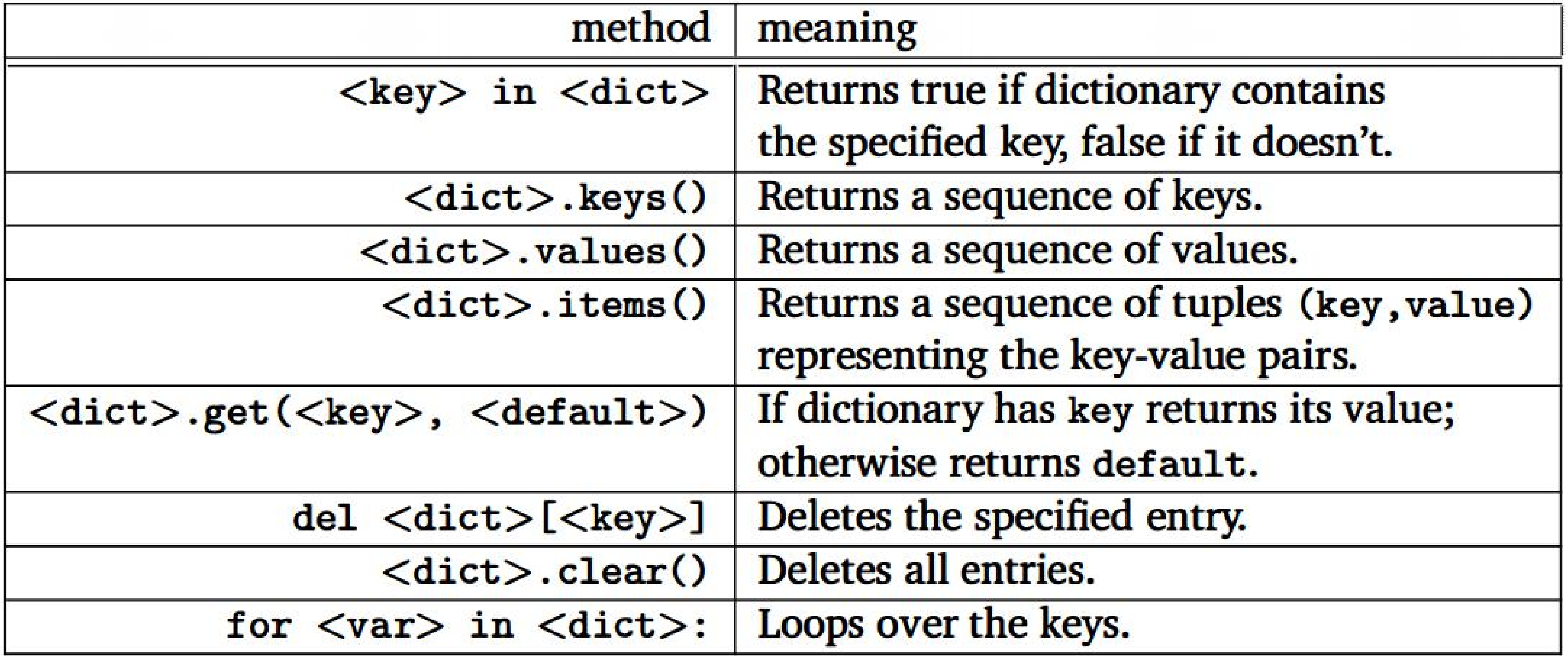


```
>>> passwd[’newuser’] = ’ImANewbie’
>>> passwd
{’turing’: ’genius’, ’bill’: ’bluescreen’, \
 ’newuser’: ’ImANewbie’, ’guido’: ’superprogrammer’}
```



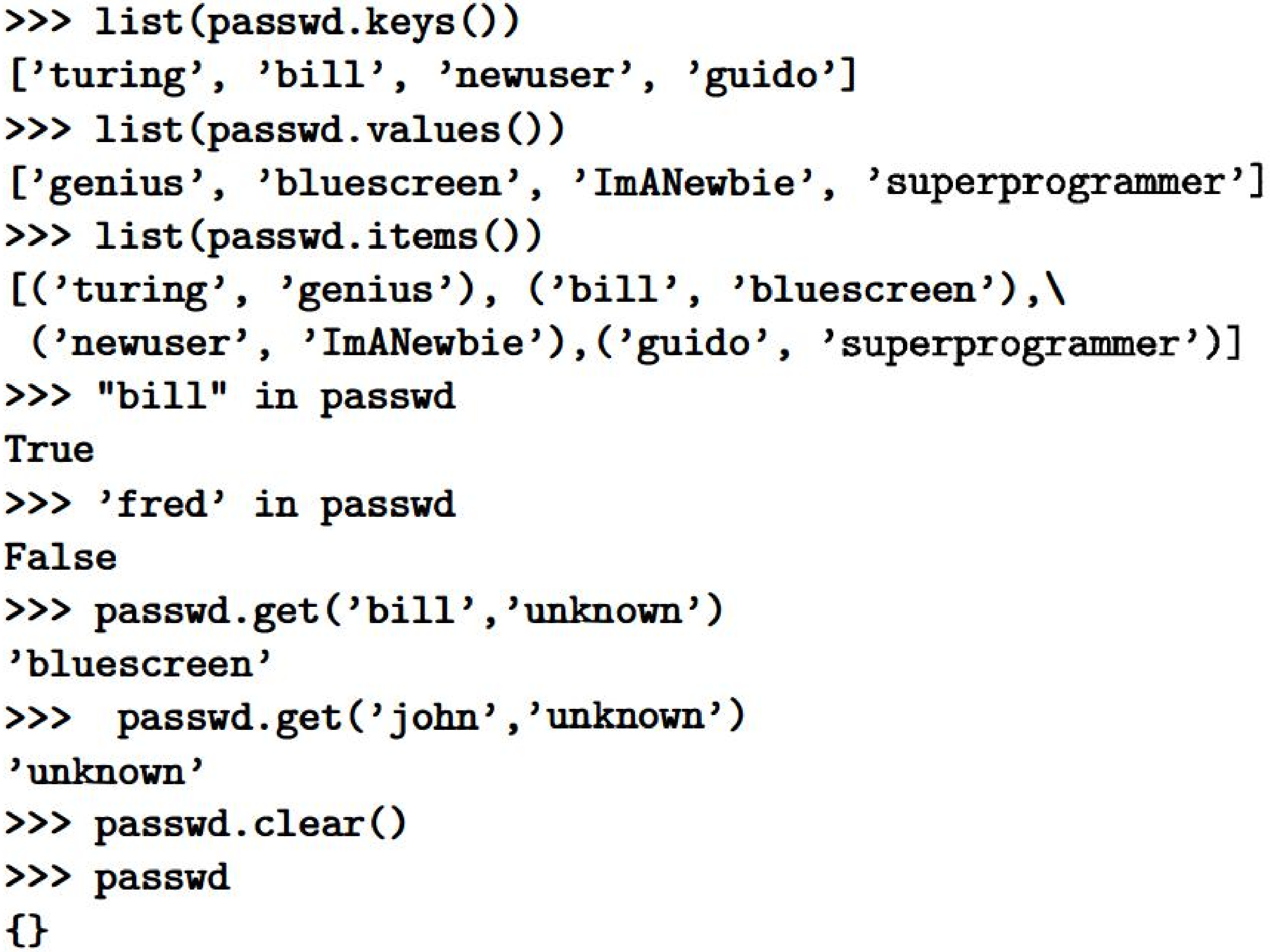
```
passwd = {}
for line in open(’passwords’,’r’):
    user, pass = line.split()
    passwd[user] = pass
```

Non-sequential Collections: Dictionary Operations



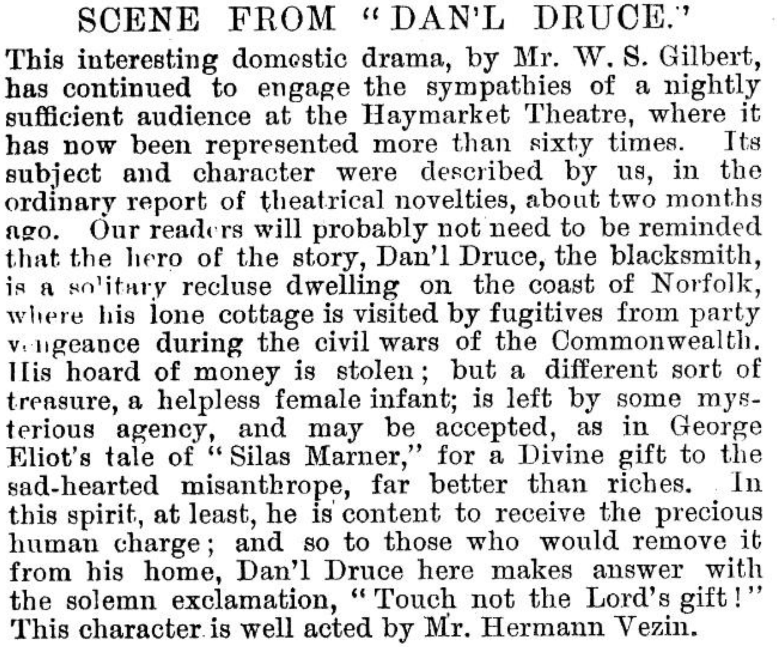
method	meaning
`<key> in <dict>`	Returns true if dictionary contains the specified key, false if it doesn't.
`<dict>.keys()`	Returns a sequence of keys.
`<dict>.values()`	Returns a sequence of values.
`<dict>.items()`	Returns a sequence of tuples `(key,value)` representing the key-value pairs.
`<dict>.get(<key>, <default>)`	If dictionary has `key` returns its value; otherwise returns `default`.
`del <dict>[<key>]`	Deletes the specified entry.
`<dict>.clear()`	Deletes all entries.
`for <var> in <dict>:`	Loops over the keys.

Non-sequential Collections: Dictionary Operations



```
>>> list(passwd.keys())
[’turing’, ’bill’, ’newuser’, ’guido’]
>>> list(passwd.values())
[’genius’, ’bluescreen’, ’ImANewbie’, ’superprogrammer’]
>>> list(passwd.items())
[(’turing’, ’genius’), (’bill’, ’bluescreen’),\
 (’newuser’, ’ImANewbie’),(’guido’, ’superprogrammer’)]
>>> "bill" in passwd
True
>>> ’fred’ in passwd
False
>>> passwd.get(’bill’,’unknown’)
’bluescreen’
>>>  passwd.get(’john’,’unknown’)
’unknown’
>>> passwd.clear()
>>> passwd
{}
```

Non-sequential Collections: Word Frequency

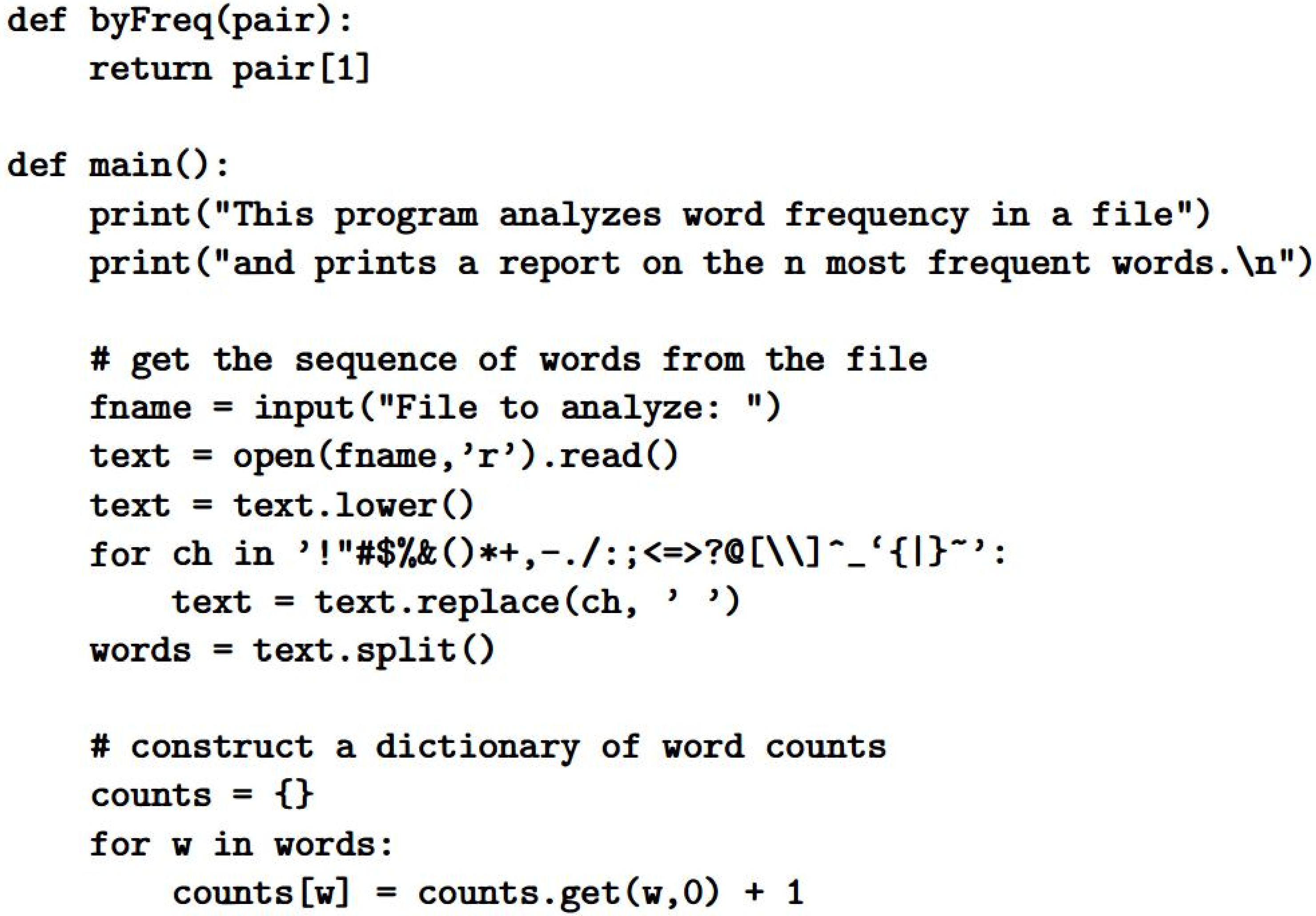


SCENE FROM "DAN'L DRUCE."

This interesting domestic drama, by Mr. W. S. Gilbert, has continued to engage the sympathies of a nightly sufficient audience at the Haymarket Theatre, where it has now been represented more than sixty times. Its subject and character were described by us, in the ordinary report of theatrical novelties, about two months ago. Our readers will probably not need to be reminded that the hero of the story, Dan'l Druce, the blacksmith, is a solitary recluse dwelling on the coast of Norfolk, where his lone cottage is visited by fugitives from party vengeance during the civil wars of the Commonwealth. His hoard of money is stolen; but a different sort of treasure, a helpless female infant; is left by some mysterious agency, and may be accepted, as in George Eliot's tale of "Silas Marner," for a Divine gift to the sad-hearted misanthrope, far better than riches. In this spirit, at least, he is content to receive the precious human charge; and so to those who would remove it from his home, Dan'l Druce here makes answer with the solemn exclamation, "Touch not the Lord's gift!" This character is well acted by Mr. Hermann Vezin.

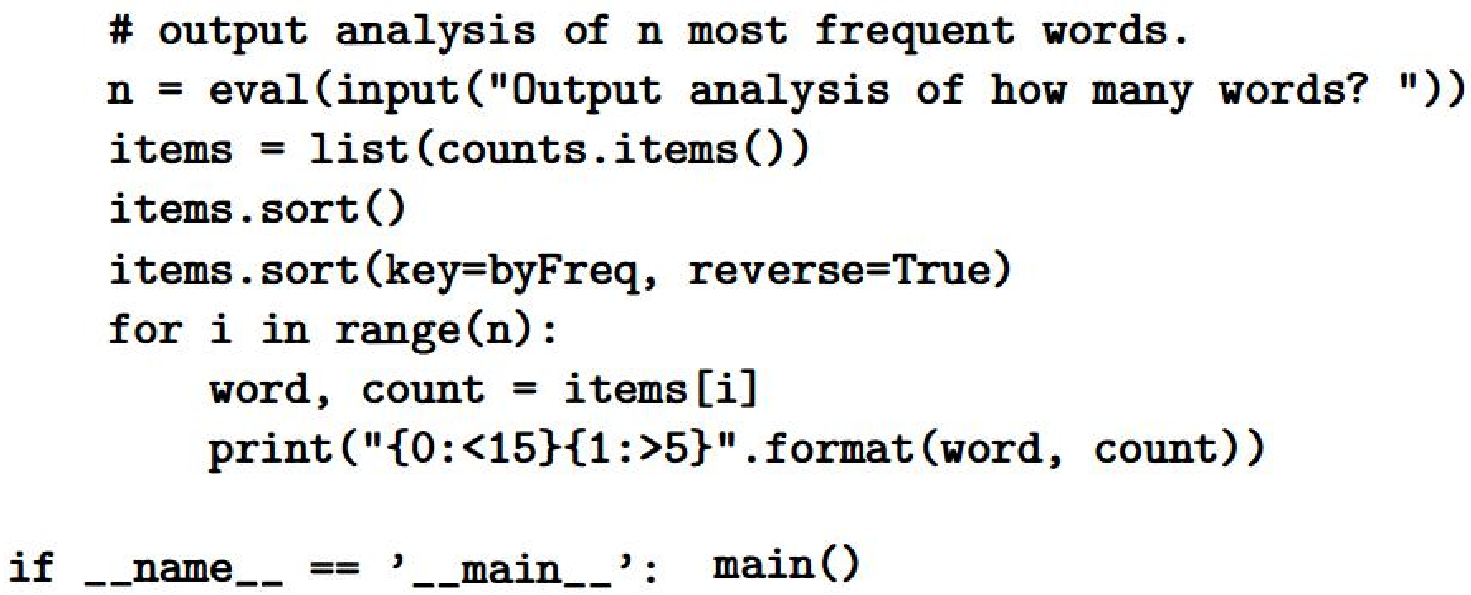
How many times each word appears?

Non-sequential Collections:
Word Frequency



```
def byFreq(pair):
    return pair[1]

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

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

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

Non-sequential Collections: Word Frequency



```
    # output analysis of n most frequent words.
    n = eval(input("Output analysis of how many words? "))
    items = list(counts.items())
    items.sort()
    items.sort(key=byFreq, reverse=True)
    for i in range(n):
        word, count = items[i]
        print("{0:<15}{1:>5}".format(word, count))

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