CS 161: Introduction to Programming and Problem-solving

Warren Harrison

Lists & Tuples

Part A

PORTLAND STATE UNIVERSITY

Scalar Variables

- Up to now, we've mainly focused on individual items through the use of *scalar variables*:
  - `testScore = 87`
- A scalar variable can hold only one value at a time
- When a new value is added, the old one is replaced

Calculating the Average Test Score

with Scalars

```
totalScore = 0
students = int(input("Student Count "))
for count in (range(students)):
    score = int(input("Enter Score "))
    totalScore = totalScore + score
avgScore = totalScore / students
print("The average score is ",avgScore)
```

Collections

- We often want to represent collections of items:
  - `cs161Scores = [87,93,66,82,77,100,89]`
- When dealing with scalar variables, we get a value, process it, then get another value, and so on.
- When dealing with collections, we get *all* the values first, and then process them

Lists

- **Lists** are used in Python to hold collections of values
- a List is represented with square brackets: []
- Initialize a named list:

```
studentList = ["Bob","Tom","Ann","Sally"]

studentList = []
```

Display the Contents of a List

```
studentList = ["Bob","Tom","Ann","Sally"]
print(studentList)
```

```
>>>
['Bob', 'Tom', 'Ann', 'Sally']
>>>
```

PORTLAND STATE UNIVERSITY

Display the Contents of a List Using a For Loop

```
studentList = ["Bob","Tom","Ann","Sally"]
for studentName in studentList:
    print(studentName)
```

Compare to:

```
studentList = ["Bob","Tom","Ann","Sally"]
print(studentList)
```

Reviewing the For Loop

```
for loop variable in sequence:
    <loop body>
```

- The for loop iterates over a sequence of items, assigning each subsequent item in the sequence to the loop variable
- We can use `range(n)` to force to loop to repeat a certain number of times

Accessing Specific List Items

```
studentList = ["Bob","Tom","Ann","Sally"]
print(studentList[0])
```

```
>>>
Bob
>>>
```

This is called indexing

Concatenating Lists

Join two (or more) lists to create one using "+"

```
studentList1 = ["Ann","Sally","Lisa"]
studentList2 = ["Bob","Tom","Mark"]
studentList3 = studentList1 + studentList2
print("Student List1: ",studentList1)
print("Student List2: ",studentList2)
print("Student List3: ",studentList3)
```

Filling a List From the Keyboard

- Create a list using **`input`**:

```
[input("Student ")]
```

- And concatenate it to the receiving list:

```
stuList = []
stuList = stuList + [input("Student ")]
```

Calculating the Average Test Score

with a List

```
totScore = 0
scrList = []
students = int(input("Student Count "))
for count in (range(students)):
  scrList = scrList + [int(input("Score "))]
for count in (range(students)):
  totScore = totScore + scrList[count]
avgScore = totScore / students
print("The average score is ",avgScore)
```

Why?

- Why do we want to deal with collections of items (lists) rather than individual items (scalars)?
- Sometimes it is nice to separate the input from the processing

Let's Revisit MPG

which report do you like best?

```
>>>
city? Portland
Odometer Reading? 10120
How many gallons? 10
Portland  MPG:  12.0
city? Oregon City
Odometer Reading? 10220
How many gallons? 10
Oregon City  MPG:  10.0
city? Gladstone
Odometer Reading? 10390
How many gallons? 13
Gladstone  MPG:  13.0769230769
city? ALL DONE
Total MPG  30.0
>>>
```

```
>>>
city? Portland
Odometer Reading? 10120
How many gallons? 10
city? Oregon City
Odometer Reading? 10220
How many gallons? 10
city? Gladstone
Odometer Reading? 10390
How many gallons? 13
city? ALL DONE
Portland MPG 12.0
Oregon City MPG 10.0
Gladstone MPG 13.0769230769
Total MPG  30.0
>>>
```

MPG w/in-line processing

```
odometer = startOdometer = 10000
totalGallons = 0
city = input("city? ")
while(city != "ALL DONE"):
    newOdometer = int(input("Odometer Reading? "))
    gallons = float(input("How many gallons? "))
    mpg = (newOdometer - odometer)/gallons
    print(city," MPG: ",mpg)
    odometer = newOdometer
    totalGallons = gallons
    city = input("city? ")
mpg = (odometer - startOdometer)/totalGallons
print("Total MPG ",mpg)
```

MPG using a list

```
odometer = startOdometer = 10000
totalGallons = cityCount = 0
cityList=odoList=galList=[]
city = input("city? ")
while(city != "ALL DONE"):
    cityList= cityList + [city]
    odoList = odoList + [int(input("Odometer Reading? "))]
    galList = galList + [float(input("How many gallons? "))]
    city = input("city? ")
for currentCity in cityList:
    newOdometer = odoList[cityCount]
    gallons = galList[cityCount]
    mpg = (newOdometer - odometer)/gallons
    print(currentCity,mpg)
    odometer = newOdometer
    totalGallons = gallons
    cityCount = cityCount + 1
mpg = (odometer - startOdometer)/totalGallons
print("Total MPG ",mpg)
```

Why?

- Why do we want to deal with collections of items (lists) rather than individual items (scalars)?
- Sometimes it is nice to separate the input from the processing
- Sometimes you need all the input items in one place to do the processing you need to do

PORTLAND STATE UNIVERSITY

Central Tendency

- a measure of a "central" or "representative" value of a collection of data
  - **Arithmetic mean** (or simply, mean) – the sum of all measurements divided by the number of observations in the data set – we usually call it the average
  - **Median** – the middle value that separates the higher half from the lower half of the data set
  - **Mode** – the most frequent value in the data set

Computing the Median

- List all the values in order from smallest to largest – this called ***sorting***
- Use the sort method – *sorts the list in place*
  - `scrList.sort()`

Sorting a List

```
scrList=[67,34,88,86,92,76,84,79,71,90]
print(scrList)
scrList.sort()
print(scrList)
```

```
>>>
[67, 34, 88, 86, 92, 76, 84, 79, 71, 90]
[34, 67, 71, 76, 79, 84, 86, 88, 90, 92]
>>>
```

PORTLAND STATE UNIVERSITY

Computing the Median

- List all the values in order from smallest to largest – this called ***sorting***
- Use the sort method – *sorts the list in place*
  - `scrList.sort()`
- Find the middle element so that an equal number of items in the list are greater than and less than the midpoint
  - The size of the list could be odd or even – two different cases …

Two Lists

- 43
- 52
- 66
- 69
- 78
- 82

- 43
- 52
- 66
- 69
- 78
- 82
- 97

Odd & Even Lists and the Median

- For odd lists, the median is the middle element
- For even lists, the median is the average of the two middle elements
- How do you tell if the list length is odd or even?

Is the List Odd or Even?

- First, find out how long the list is
- The len() function

```
listSize = len(theList)
```

- If you can divide a number in two, with no remainder, it's even – use the modulo operator, % (computes the remainder of a division)

```
remainder = listSize % 2
if remainder == 0:
```

PORTLAND STATE UNIVERSITY

What is the index of the Midpoint?

Midpoint = listSize // 2 – *note integer division*

… remember – indexes start at 0!

- 43
- 52
- 66
- 69
- 78
- 82

- 43
- 52
- 66
- 69
- 78
- 82
- 97

Computing the Median

```
scrList=[67,34,88,86,92,76,84,79,71,90,91]
scrList.sort()
print(scrList)
listSize = len(scrList)
remainder = listSize % 2
if remainder == 0:
    midpoint = listSize // 2
    median = (scrList[midpoint]+scrList[midpoint-1])/2
else:
    midpoint = listSize // 2
    median = scrList[midpoint]
print(median)
```

What are the Top Five Scores?

```
scrList=[67,34,88,86,92,76,84,79,71,90,91]
scrList.sort()
print(scrList)
print(scrList[-5:])
```

```
>>>
[34, 67, 71, 76, 79, 84, 86, 88, 90, 91, 92]
[86, 88, 90, 91, 92]
>>>
```

Reviewing Slices

- A "slice" allows us to partition off a sequential subset of the list items
- list[start:end]
- Returns the elements between the two indexes
- 0 denotes the first element
- The number of items in the list denotes the last element
- Can use negative indexes to count backwards