Week 11, Examples 1

# Basics on Lists #1.py

Programs work with “collections of data” E.g. Program needs to work with student records. Student record: name, gpa, credit hours, address, club memberships. Most languages use ARRAYS (e.g., C, Java, C++). An array is basically a list. It has a certain size (.e.g., “int names[10]” is an array/list of 10 integers in C). So a Python list can do what a C array can do, AND MORE! C/Java: each element of the array has to be of the SAME type (i.e., homogeneous) C/Java: arrays have to be of a fixed size (you define size beforehand, or you ask for more space as needed while the program runs). Python: lists are dynamic (i.e., no need to state the size beforehand) and lists are heterogeneous (i.e., the elements of the list need not be of the same type; so one item can be an integer, and the next item can be a float).

What we need to learn: 1. How to use lists with “stuff” in them 2. Which functions we can use to “do stuff” with lists 3. To be comfortable programming with lists; they are VERY useful 4. Lists are an example of “Python Sequences”. But sometimes, data is stored in a way that is non-sequential. Python DICTIONARIES are useful for this.

# Quick example of why we need to use a list

Remember earlier (Lecture #?) we wrote code to work with a sequence of numbers and compute — mean, and — variance In that lecture, you learned a clever trick: how to compute mean and variance “on the fly” (that means without storing the numbers anywhere; you used a number as soon as you read it in or as soon as you computed it). Now the mean is easy to compute on-the-fly But to compute the variance, the usual formula needs the mean, because each number must be subtracted from the mean and the result squared. We then add up all the squared terms and divide by (n-1) to get “unbiased variance”. But in that lecture, we used a trick to get the variance “recursively”) (i.e., the new variance, based on the new data item, used the old variance based on the previous data items and then updated it) If you did not use that trick (and most text-books do not even talk about it), then you’d have to compute the mean on the first pass, and store all the numbers on the first pass. Then in the second pass you would find the difference of each number from the mean and then square it etc etc. ' # So why so we need a list?

Well, to store all the numbers so you can use them in the second pass. So this is code, we will be using the standard two-pass method to get variance. '# computing properties of numbers based on lists #2.py

The text-book example gets the numbers via user-input. In our example, we'll just fill a list with random numbers and pretend that a user input these numbers somehow. '#stats.py

```python
if you save as stats.py and you can use this as your stat module # on a list of numbers. import sys from math import sqrt from random import randint def getlist(): #returns a list of numbers """ getlist generates a list of random integers """ n = int(input("How many numbers in list? ")) x = [] for j in range(n): x.append(randint(1,500)) # some random integer between 1 and 500 return(x) def xbar(nlist): # nlist is a list of numbers """ xbar computes the average of a list of numbers """ sum = 0.0 size = len(nlist) for j in range(size): sum = sum + nlist[j] if (size != 0): return(sum/size) else: print("Error: size of nlist is 0\n") sys.exit(0) def s_squared(xbar,nlist): """ s_squared computes the
```
variance of a list of numbers, using xbar 

\[
\text{sum} = 0.0 \quad \text{size} = \text{len(nlist)} \quad \text{for} \quad j \quad \text{in} \quad \text{range(size)}: \quad \text{sum} = \text{sum} + (\text{nlist}[j] - \text{xbar})(\text{nlist}[j] - \text{xbar})
\]
# if (size != 0): return(sum/size-1) else: print("Error: size of nlist is 0\n") sys.exit(0)

def even(n):
# return True if even; else return False
if (n%2 == 0):
    return(True)
else:
    return(False)
def median(nlist):
# median returns the median of a list
\[
\text{size} = \text{len(nlist)} \quad \text{for} \quad j \quad \text{in} \quad \text{range(len(nlist))}: \quad \text{print}(\text{print}(\text{nlist}[j]) \quad \text{if} \quad (\text{size} == 0):
\]
# if (size != 0): return((\text{nlist}[\text{size}2] + \text{nlist}[(\text{size}2) -1])/2)
# even: average of two middle elements of sorted list else: return(nlist[\text{size}1-1]))
# odd: middle element of sorted list

\[
\text{def range(nlist)}: \quad \text{we'll use the math lib functions (min and max)} \quad \text{and report range as (max - min)} \quad \text{size} = \text{len(nlist)} \quad \text{if} \quad (\text{size} == 0): \quad \text{return}(\text{Error: size of nlist is 0\n}) \quad \text{sys.exit(0)}
\]
#report range as (max - min)

\[
\text{def write(nlist)}: \quad \text{write out one list for} \quad j \quad \text{in} \quad \text{range(len(nlist))}: \quad \text{print}(\text{println}[j]) \quad \text{write2(alist,blist)}: \quad \text{write out both lists for} \quad j \quad \text{in} \quad \text{range(len(alist))}: \quad \text{println}[j],\text{blist}[j])
\]
# HW: The "mode" is the number that repeats most often in the list. If there are no repeat # numbers then the list has no mode. Write a simple function to find the mode.
def median(): mylist = getlist() write(mylist)
m = xbar(mylist) # get mean v = s_squared(m, mylist) # get variance: notice is the 2nd pass and you have to feed it m sd = sqrt(v) # get standard deviation

\[
\text{print}(\text{"Statistics\n}) \quad \text{print}(\text{\"Mean: {0:0.2f}}\).format(m)) \quad \text{print}(\text{\"Variance: {0:0.2f}}\).format(v)) \quad \text{print}(\text{\" Std. dev: {0:0.2f}}\).format(sd)) \quad \text{print}(\text{\" Min: },\text{println}[0],\text{\" Max: },\text{println}[\text{len(mylist)}-1]) \quad \text{print}(\text{\" Range: {0:02f}}\).format(xrange(mylist)))
\]

\[
\text{main(): \text{filename} = \text{\"volunteers.txt\n}} \quad \text{\_ save this file as volunteers.txt}\)
\]
# make sure last line has a \"\n\" at end of line followed by no data on next line, or the read # data portion of the program will find unexpected data
# The file contains """" name height (in pounds) weight (in pounds) """" #of volunteer players who want to join a basketball team #3.py 
# save only the data below_Duck,Daffy 3 5 Fudd,Elmer 3.1 10 Bunny,Bugs 3.5 10 Baggins,Bilbo 4.1 12 Baggins,Frodo 3.9 11
# Stooge,Larry 5.5 130 Stooge,Moe 5.5 128 Stooge,Curly 5.5 190 Coyote,Wiley 3.2 15 Runner,Road 2.6 2 Yosemite,Sam 2.8 20
# Duck,Huey 2.4 1 Duck,Dewey 2.1 12 Duck,Louie 1.9 3.9 Cat,Sylvester 2.83 10 Woodpecker,Wdy 1.5 2 Devil,Tasmanian 1.1
# Mouse,Mickey 2.12 3 Mouse,Minnie 1.9 1.95 Pooh,Winnie 4.39 4.39 # #4.py 
# read in a list #of basketball volunteers and select players; use Python class, objects, sort
# order the players: descending height list, descending weight list, #descending ht*wt list # the top 10 players in each list could make up the
# top 5 actives and next 5 reserves \class Bplayer: def init(self,name,height,weight):
self.name = name self.ht = float(height) #comes in as string, so make it a number self.wt =
float(weight) def getName(self): return self.name def getHeight(self): return self.ht def
getWeight(self): return self.wt def getScore(self): #score is a product of height and weight return self
getMakePlayer(VolunteerString): # used by main to make player objects name, ht, wt =
VolunteerString.split() # data on line is separated by whitespace (i.e., blanks) return Bplayer(name, # ht, wt) # IMPORTANT: It makes and returns a player object ! # some helper functions that we can give
to the \"sort\" function to use as the \"key\" field (i.e., the \# field on which to do the sorting). # you'll
see that we don't have to use these functions if we don't want to. We can use the methods # we
already have: getHeight, getWeight, etc .... because the functions call these methods anyway. def
use ht(aPlayer): return(aPlayer.getHeight()) def use wt(aPlayer): return(aPlayer.getWeight()) def
use score(aPlayer): return(aPlayer.getScore()) def main(): filename = \"volunteers.txt\n" open(filename, \"r\") playerlist = [] print(\"Reading in the volunteer list....................... \n\") for line in in file:
p = makeBplayer(line) # p is now a basketball player object print(p.name,p.ht,p.wt,p.getScore())
playerlist.append(p) in line.close() # now playerlist has all the volunteer objects print(\"\n\") A
basketball team has 5 actives and 5 reserves. How to choose our team? We can # choose based on
the 10 tallest, 10 heaviest or 10 with highest score htlist = playerlist.copy() wtlist = playerlist.copy()
scorelist = playerlist.copy() htlist.sort(key = use ht, reverse = True) # important: we are sorting
based on height, but we # are passing the “use_ht” function via its name to # the built-in sort function. Notice that we did not say # key = use_ht() because that would have called the # function. We do not want to do that. We only want to # give the function name to the sort function. # notice how we sort to get descending order print (“Name	Height”) for p in htlist:
print(“{0}	{1}”.format(p.getName(),p.getHeight()))

# Above we used a helper function use_ht() to sort based on height. But we did not need to do # that. Why? Because we already have
the function p.getHeight(), when p is a Bplayer object. # So let's try that now to order them in terms of
descending weight wtlist.sort(key=Bplayer.getWeight, reverse = True) # feed the sort function the #
name of the method directly print (“Name	Weight”) for p in wtlist:
print(“{0}	{1}”.format(p.getName(),p.getWeight()))

# order based on score =
height*weight scorelist.sort(key=Bplayer.getScore, reverse = True) # feed sort the name of the
method directly print (“Name	Score”) for p in scorelist:
print(“{0}	{1}”.format(p.getName(),p.getScore()))

main()

1) self.ht)*(self.wt

From:
http://courses.cs.purdue.edu/ - Computer Science Courses

Permanent link:
http://courses.cs.purdue.edu/cs17700:fall15:week11_examples1_save

Last update: 2015/11/03 12:43