Introduction to Python

Collections

Simple Statistics

```
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( "\nThe average of the numbers is", sum / count)

main()
```

Simple Statistics

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

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.

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.

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

Simple Statistics

- We need to keep track of all the values inserted by the user
- We do not know how many variables the user will provide.

Lists

- Python provides List to store sequences of values
- Lists in python are dynamic.
  - They grow/shrink on demand.
- Lists are mutable
  - Values can change on demand
  - Data type of individual items can change

List: Basic Examples

```
lst = [1,5,15,7]
print(lst)

lst[2] = 22
lst

lst[1] = “Hello”
lst

zeroes = [0] * 5
zerones = [0,1] * 3
zerones.append(2)
```

List: Operators

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: Basic Examples

```
lst = lst + [22, 3]
len(lst)

15 in lst
3 in lst

sum = 0
for x in zerones:
        sum += x
print(sum)

X = zerones
zerones.append(2)

Y = lst[1:3]
Z = lst[3:-1]
K = lst[1:-3]
```

List: Operators

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: Additional Examples

```
lst = [3, 1, 4, 1, 5, 9]
lst.append(2)
lst

lst.sort()
lst

lst.reverse()

lst.index(4)

lst.insert(4, "Hello")

lst.count(1)

lst.remove(1)

lst.pop(3)
```

Simple Statistics: Modifications

- Collect input from user
- Store in a list

Simple Statistics

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

Simple Statistics

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

Simple Statistics

- How do we compute the standard deviation?
- Do we re-compute the mean?
  - Inefficient for large collections
- Do we pass the mean as a parameter?
  - Forced to invoke both functions sequentially

Simple Statistics

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

Simple Statistics

- How do we compute the median?
- 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
```

Simple Statistics

```
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
```

Simple Statistics

```
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)
```

Range()

- “range” creates a list of numbers in a specified range
  - range([start,] stop[, step])
  - When step is given, it specifies the increment (or decrement).

```
range(5)

range(5, 10)

range(0, 10, 2)

for i in range(0, len(lst), 2):
  print lst[i]
```

Zipping Lists

```
k = zip(lst, zerones)

for (i,j) in k:
   print (i,j)
```

Tuples

```
data = [(“julius”, 3),
(“maria”, 2),
(“alice”, 4)]

for (n, a) in data:
  print(“I met %s %s times" % (n, a))

data.sort()
```

Structured Text Files

- Module CSV provides useful functions to handle structured text files
- CSV : Comma separated values
  - It supports other separators, e.g., tab delimited

Example: Import File

```
import csv
f = open("beers.txt")
x = 0
for row in csv.reader(f, delimiter='\t'):
    print(row)
    x += 1
    if (x > 10):
        break
```

Most rated beer

- Identify beer with most ratings
- Compute mean/median/stddev of ratings

Identify most ranked beer

```
cut -f 1 ../lab1/beers.txt | sort | uniq -c
| sort -n -r | head -1

grep “result” ../lab1/beers.txt > most-
popular.txt
```

Compute Statistics

```
p = open("most-popular.txt")
values = []
for row in csv.reader(p, delimiter='\t'):
    values.append(int(row[1]))

xbar = mean(values)
std = stdDev(values, xbar)
med = median(values)

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

Dictionaries

- Lookup tables
- They map from a “key” to a “value”
- Duplicate keys are not allowed

```
cities= {“A”: “Ancona”,
   “B”: “Bary”,
   “C”:“Como”}
```

Dictionaries

- Keys can be of any data type

```
element = {1: "hydrogen"
6: "carbon",
7: "nitrogen"
8: "oxygen",
}
```

Dictionaries

- Keys can also be tuples

```
nobel = {
(1979, "physics"): ["Glashow", "Salam",
  "Weinberg"],
(1962, "chemistry"): ["Hodgkin"],
(1984, "biology"): ["McClintock"],
}
```

Dictionaries: Accessing Elements

```
cities['A']

element[7]

nobel[(1979, "physics")]


cities['F']
cities.get("F","unknown")
```

Dictionaries: Useful methods

```
cities.keys

cities.values

cities['D']='Domodosola'

cities.update({"F": "Firenze", "G":
  "Genova"})

del cities['C']
```

Dictionaries: Exercise

- Construct a dictionary based on the beers.txt
- Each beer name is a key
- All ratings are the values
  - Stored as a list

Load all Ratings

```
import csv
f = open("../lab1/beers.txt")
dict = {}
for row in csv.reader(f,
delimiter='\t'):
    ratings = dict.get(row[0], [])
    ratings.append(row[1])
    dict[row[0]]= ratings

len(dict.keys())
```

Compute Statistics

```
stat = {}
for beer in dict.keys():
    ratings = dict.get(beer)
    m = mean(ratings)
    stat[beer] = {"count":
len(ratings), "mean": m}
```

Redefine Mean function

```
def mean(nums):
    sum = 0
    for num in nums:
        sum = sum + int(num)
    return sum / len(nums)
```

OR read file as int and not str

Produce Statistics

```
def countindex(num):
    return stat[num]["count"]

sorted(stat, key=countindex,
reverse=True)
```

Print Sorted Statistics

```
sortedstat = sorted(stat, key=countindex,
reverse=True)

for key in sortedstat:
    print("%s: %s" % (key, stat[key]))
```

Exercise

- Identify median of count of beer ratings
- Consider only beers with number of ratings above median
- Order beers based on mean rating

Exercise

- Consider 100 beers with most number of ratings received
- Order beers based on mean rating