Workshop 3

Let's look at some looping and branching in the interactive mode first:

```
>>> namelist = ['bob', 'sue', 'ray', 'ken', 'jim', 'malcomb']
>>> fullnamelist=[['bob','jones'],['sue','smith'],['ray','post','jr'],['ken','harry','johnson','III']]

>>> for name in namelist: print(name)
...
bob
sue
ray
ken
jim
malcomb

>>> for name in fullnamelist: print(name)
...
['bob', 'jones']
['sue', 'smith']
['ray', 'post', 'jr']
['ken', 'harry', 'johnson', 'III']

>>> for name in fullnamelist:
...     for item in name:
...             print(item)
...
bob
jones
sue
smith
ray
post
jr
ken
harry
johnson
III
```

```
>>> workerlist = [{'name': 'bob', 'age':40, 'job':'manager'}, {'name':'sue', 'age':28, 'job':'manager'},
... {'name':'ken', 'age': 32, 'job':'programmer', 'salary':50000}]

>>> for worker in workerlist: print(worker)
...
{'name': 'bob', 'age': 40, 'job': 'manager'}
{'name': 'sue', 'age': 28, 'job': 'manager'}
{'name': 'ken', 'age': 32, 'job': 'programmer', 'salary': 50000}

>>> for worker in workerlist:
...     keyslist = list(worker.keys())
...     print(keyslist)
...
['name', 'age', 'job']
['name', 'age', 'job']
['name', 'age', 'job', 'salary']

>>> for worker in workerlist:
...     print(worker)
...
{'name': 'bob', 'age': 40, 'job': 'manager'}
{'name': 'sue', 'age': 28, 'job': 'manager'}
{'name': 'ken', 'age': 32, 'job': 'programmer', 'salary': 50000}

>>> managers = []
>>> for worker in workerlist:
...     if worker['job'] == 'manager':
...             managers.append(worker)
...
>>> print(managers)
[{'name': 'bob', 'age': 40, 'job': 'manager'}, {'name': 'sue', 'age': 28, 'job': 'manager'}]

>>> youngpeople = []
>>> for worker in workerlist:
...     if worker['age'] < 40:
...             youngpeople.append(worker['name'])
...
>>> print(youngpeople)
['sue', 'ken']
>>>
```

Branching Statements (IF..ELIF..ELSE)

Control statements "control" which sections of code in a program are executed. For example, we might want a set of statements to execute only if certain conditions are met, otherwise, the statements would not be executed. There are three general types of control statements:

1. Sequential – The default ordering of execution
2. Decision (Branching or Conditional) – controls which block of code within several alternatives is executed.
3. Looping (Iterative) – controls how many times a block of code is executed.

Diagrams of the logic flow:

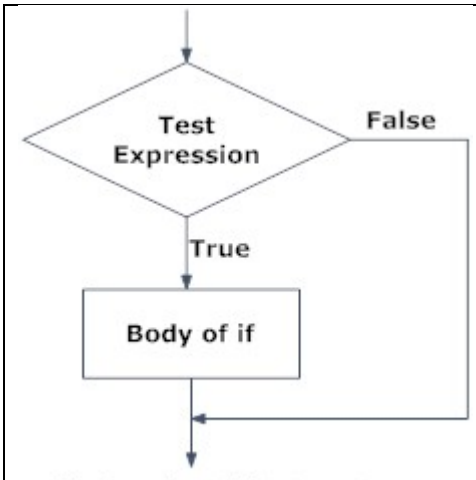
<table>
<tr>
<td> 

Fig: Operation of if statement</td>
<td>

```
if condition:
   statements
next program statement
```

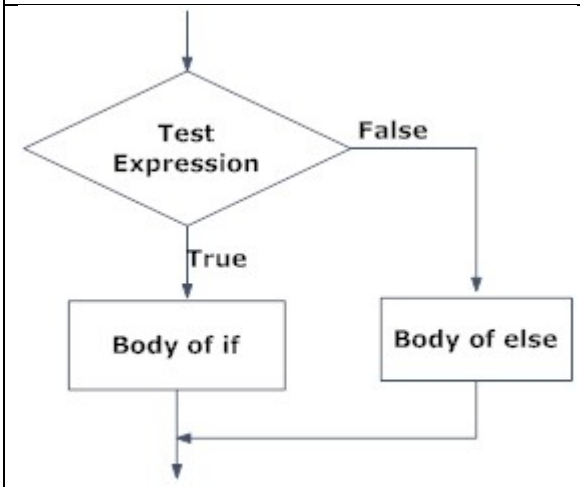
</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td> 

Fig: Operation of if...else statement</td>
<td>

```
if condition:
   statements
else:
   else statements
next program statement
```

</td>
</tr>
</table>

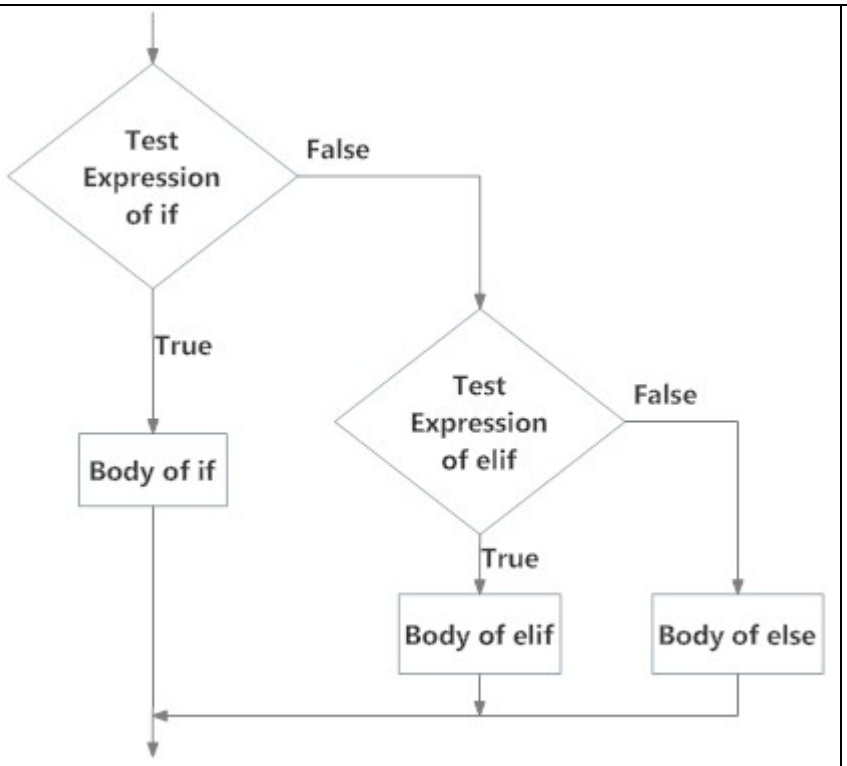


Fig: Operation of if...elif...else statement

```
if condition1:
   statements1
elif condition2:
   statements2
elif condition3:
   statements3
…
else:
   else statements
next program statement
```

Decision (Branching or Conditional) statements are a type of control statement and are often referred to as IF..ELIF..ELSE statements as can be seen in these examples:

Complexity	Conditional Code

If *condition* is true, execute *statements*. After that, execute *next program statement*.	`if condition:`
	`  statements`
	`next program statement`

If *condition* is true, execute *statements*, otherwise (*else*) execute the *else statements*. After that, execute *next program statement*.	`if condition:`
	`  statements`
	`else:`
	`  else statements`
	`next program statement`

If *condition1* is true, execute *statements1*, otherwise check if (*elif*) *condition2* is true, and if it is, execute *statements2*, otherwise check if (*elif*) *condition3* is true, and if it is, execute *statements3*, etc. and if none of the prior conditions are true (*else*) execute the *else statements*. After that, execute *next program statement*.	`if condition1:`
	`  statements1`
	`elif condition2:`
	`  statements2`
	`elif condition3:`
	`  statements3`
	`…`
	`else:`
	`  else statements`
	`next program statement`

In each case, the *condition* portion can be an arbitrarily complex boolean expression (using variables declared in the program) that will evaluate to either True or False. If *condition* evaluates to True then the *statements* will be executed. If all prior *conditions* evaluate to False, then the *else statements* will be executed if any are given.

In the following examples, a variable salary is defined and a value is assigned to it. A conditional statement then determines which message should be displayed.

Code Structure	Example Code

if *condition:*
  statements
next program statement
	salary = input("Enter salary category: ")
	if salary == 5:
	  print("Employees should be paid 25,000")
	print("Processing done.")

if *condition:*
  statements
else:
  else statements
next program statement
	salary = input("Enter salary category: ")
	if salary == 5:
	  print("Employees should be paid 25,000")
	else:
	  print("Employees should be paid 40,000")
	print("Processing done.")

if *condition1:*
  statements1
elif condition2:
  statements2
elif condition3:
  statements3
…
else:
  else statements
next program statement
	salary = input("Enter salary category: ")
	if salary == 1:
	  print("Employees should be paid 10,000")
	elif salary == 2:
	  print("Employees should be paid 15,000")
	elif salary == 3:
	  print("Employees should be paid 20,000")
	elif salary == 4:
	  print("Employees should be paid 22,000")
	elif salary == 5:
	  print("Employees should be paid 10,000")
	else:
	  print("Employees should be paid 40,000")
	print("Processing done.")

Example Code

choice = input('Enter choice:')
branch = {'spam': 1.25, 'ham': 1.99, 'eggs': 0.99, 'bacon': 1.10}
if choice in branch:
  print (branch[choice])
else:
  print('Invalid choice')

Looping Statements

Virtually all programming languages have a facility to allow a section of code to be repeated (iterated or looped). Python has two categories: FOR loops and WHILE loops.

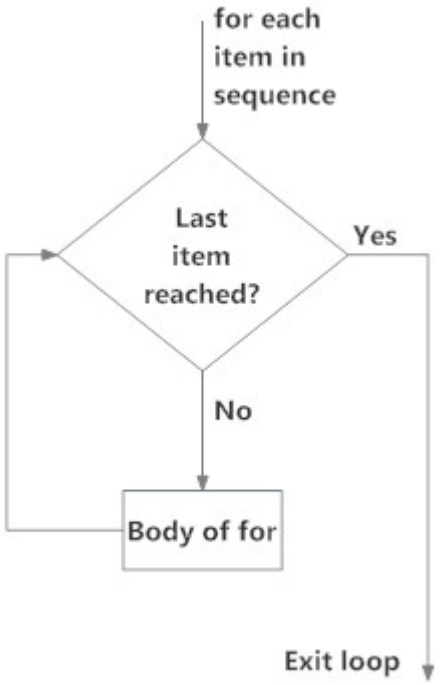


Fig: operation of for loop

```
for each item in sequence:
   statements
next program statement
```

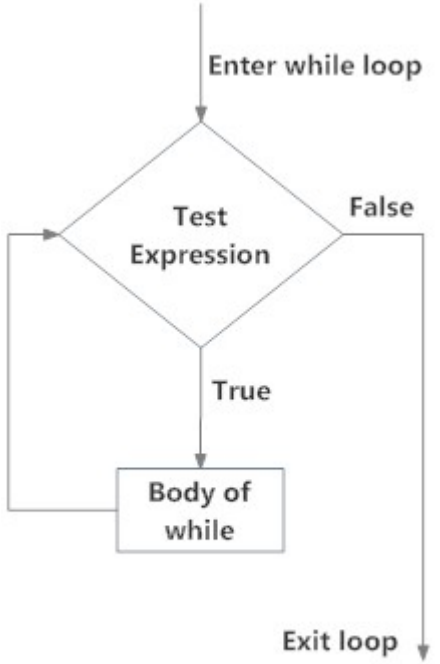


Fig: operation of while loop

```
while expression:
    statements
next program statement
```

FOR loops

Simple FOR loops iterate a specific number of times based on counting up (or down) on an integer variable. For example, if we want to do something 10 times, we can make a loop that counts up from 1 to 10 and put our work (what we want done) inside the loop. As another example, if we want something to be done for each item in a list, we can make a loop that traverses the list and put our work (what we want done) inside the loop.

Logic	FOR Loop
Execute these *statements* for values of j beginning with the value 0 and going up to the value of *n-1*.	`for j in range(n):` `  statements`
Execute these *statements* for each item in the list stored in L. (This way stores a number in j each time Python goes through the statements. The first time j is 0. The second time j is 1. … The last time j is equal to the length of L.)	`for j in range(len(L)):` `  statements`
Execute these *statements* for each item in the list stored in L. (This way stores the first item in L into j the first time Python goes through these statements. The second time it goes through them, it stores the second item into j. … The last time Python executes these statements, j will contain the last item in the list L.)	`for j in L:` `  statements`

The following examples show how the loop executes.

Code Structure	Example	Output

`for j in range(n):`
`  statements`
	`L = [10, 20, 30, 40, 50]`
	`for j in range(5):`
	`  print(j, L[j])`
		`1  10`
		`2  20`
		`3  30`
		`4  40`
		`5  50`

`for j in range(len(L)):`
`  statements`
	`L = [10, 20, 30, 40, 50]`
	`for j in range(len(L)):`
	`  print(j, L[j])`
		`1  10`
		`2  20`
		`3  30`
		`4  40`
		`5  50`

`for j in L:`
`  statements`
	`L = [10, 20, 30, 40, 50]`
	`for j in L:`
	`  print(j)`
	`end;`
		`10`
		`20`
		`30`
		`40`
		`50`

Some other examples:

Examples

```
#Initialize the dictionary of prices

prices = {'apple': 0.40, 'banana': 0.50, 'mango': 0.75, 'pear': 0.10}
my_purchase = {'apple' : 1,'mango' : 6}

#Initialize the grocery bill
grocery_bill = 0

#Loop through the items in my purchase, summing the total cost of each
for fruit in my_purchase:
    grocery_bill += prices[fruit] * my_purchase[fruit]

print ('I owe the grocer $%.2f' % (grocery_bill))
```

```
#This program works with a dictionary of stock information

import csv
def cmp (value, basis):
     if value > basis:
          return 1
     elif value < basis:
          return -1
     else:
          return 0

#write stocks as comma-separated values (csv)
with open('stocks.csv', 'w') as csvfile:
     #Establish the names of the fields
     fieldnames = ['symbol', 'firm_name', 'price', 'change', 'pct']
     #Establish writer as a csv Dict.writer object that has these field names
     writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
     #Write the data to the csv file.
     #If you open it in a text editor, it will look like text separated by commas
     writer.writerow({'symbol': 'GOOG', 'firm_name': 'Google Inc.', 'price':
505.24, 'change': 0.47, 'pct': 0.09})
     writer.writerow({'symbol': 'YHOO', 'firm_name': 'Yahoo! Inc.', 'price': 27.38,
'change': 0.33, 'pct': 1.22})
     writer.writerow({'symbol': 'CNET', 'firm_name': 'CNET Networks, Inc.',
'price': 8.62, 'change': -0.13, 'pct': -1.49})

#read the stock data into an object named stocks, print status messages
stocks = csv.reader(open('stocks.csv'))
#Set up labels to make output friendly. cmp is -1 if x<y, 0 if x == y, 1 if x>y
status_labels = {-1: 'down', 0: 'unchanged', 1: 'up'}

dictlist = []
for line in stocks:
     dictlist.append(dict(zip(['symbol','firm_name','price','change','pct'],line)))

for d in dictlist:
     test = float(d['change'])
     status = status_labels[cmp(test, 0.0)]
     print ('%s is %s %s%%' % (d['firm_name'], status, d['pct']))
```

Code Samples	Output

`# Iterating over a list`
`print("List Iteration")`
`l = ["geeks", "for", "geeks"]`
`for i in l:`
`  print(i)`
	List Iteration
	geeks
	for
	geeks

`# Iterating over a tuple (immutable)`
`print("\nTuple Iteration")`
`t = ("geeks", "for", "geeks")`
`for i in t:`
`  print(i)`
	Tuple Iteration
	geeks
	for
	geeks

`# Iterating over a String`
`print("\nString Iteration")`
`s = "Geeks"`
`for i in s :`
`  print(i)`
	String Iteration
	G
	e
	e
	k
	s

`# Iterating over dictionary`
`print("\nDictionary Iteration")`
`d = dict()`
`d['xyz'] = 123`
`d['abc'] = 345`
`for i in d :`
`  print("%s  %d" %(i, d[i]))`
	Dictionary Iteration
	xyz 123
	abc 345

Nested Loops (A Loop in a Loop)

Here's an example of a loop in a loop. This program will sort a list of numbers.

```
#old sort program

values = [2, 5, 6, 1, 4, 8, 3]

numvals = len(values)

for i in range(numvals):
    for j in range(i+1, numvals): #note how the range gets smaller each time
        if values[i] > values[j]: #if the outer loop value is larger, swap it
           temp = values[i]
           values[i] = values[j]
           values[j] = temp

print(values)
```

If you can follow how the sort above works, you probably understand loops (and list offsets).

WHILE loops

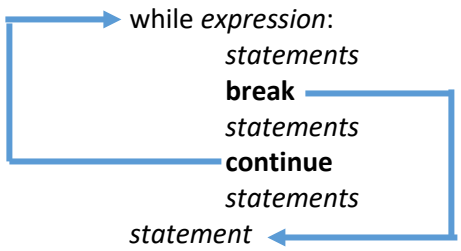
Like FOR loops, the WHILE loops are also used to repeat a section of code for some number of times. FOR loops basically specify some count; WHILE loops typically repeat while a *condition* is true. (In some cases, however, either approach can be used effectively.) The *condition* is a boolean expression that we can evaluate as either "true" or "false". As soon as the *condition* evaluates to "false" the loop ends.

Basic Structure:

Logic	While Loop
While an *expression* evaluates to true, execute the code in *statements*.	`while expression:` `    statements`

While an *expression* evaluates to true, execute the code in *statements1*, otherwise (else) execute the code in *statements2.*	`while expression:`
	`    statements1`
	`else:`
	`    statements2`

There are also break and continue statements in Python. The logic of them is as follows:



A break statement causes Python to leave the loop. A continue statement causes Python to go back to the beginning of the loop. Note: in the logic example above, code in *statements2* is only executed if the while loop ends naturally (i.e., without a break).

Examples

Code	Output

`# Python program to illustrate`
`# while loop`
`count = 0`
`while (count < 3):`
`    count = count + 1`
`    print("Hello Geek")`
	`Hello Geek`
	`Hello Geek`
	`Hello Geek`

`i = 1`
`while i < 4:`
`   print(i)`
`   i += 1`
`else:`
`   print("i is no longer less than 4")`
	`1`
	`2`
	`3`
	`i is no longer less than 4`

```
# Program to add natural
# numbers up to a number n entered by the user
# sum = 1+2+3+...+n

# Take input from the user
n = int(input("Enter n: "))

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i += 1    # update counter

# print the sum
print("The sum is", sum)
```

```
Enter n: 5
The sum is 15
```

WHILE Loop Example

```
#Create the REFRAIN of the song as a string that contains the format codes
needed
REFRAIN = '''
%d bottles of beer on the wall,
%d bottles of beer,
you take one down, pass it around,
%d bottles of beer on the wall!
'''

#Set the number of verses to sing
bottles_of_beer = 10
while bottles_of_beer > 1:
        print(REFRAIN % (bottles_of_beer,bottles_of_beer,bottles_of_beer-1)
        bottles_of_beer -= 1   #subtract 1 from number of bottles of beer
```

Comprehension

Comprehension is so cool. Comprehensions are ways of processing / creating sequences using expressions. Consider the following:

```
>>>S = ‘spam’
>>>[c for c in S]
```

The first statement creates an object S that contains a string ‘spam’. Let’s breakdown the second statement:

[indicates a list start
c for c in S can be read “make an occurrence of something I’m calling “c” for each occurrence of this thing called “c” that occurs in the string object named S”
] indicates a list end

Characters are the only thing that occurs in S. The characters are ‘s’, ‘p’, ‘a’, and ‘m’. So, the result is a list of the characters in S:

```
>>>S = ‘spam’
>>>[c for c in S]
[‘s’, ‘p’, ‘a’, ‘m’]

>>>L = [1,2,3,4,5]
>>>[n for n in L]
[1,2,3,4,5]
>>>[n*10 for n in L]
[10,20,30,40,50]
>>>L = [‘bob’, [‘sue’, ‘joe’], 23]
>>>[n for n in L]

>>>for x in [1, 2, 3]: print(x)
…
1
2
3
>>>newlist = [c*4 for c in ‘SPAM’]
>>>newlist
[‘SSSS’, ‘PPPP’, ‘AAAA’, ‘MMMM’]


>>>newlist = []
>>>for c in ‘SPAM’:
…         newlist.append(c*4)
…
>>>newlist
[‘SSSS’, ‘PPPP’, ‘AAAA’, ‘MMMM’]
```

Let's write some code!

Write a program that will "think" of a random number between 1 and 20 (I will show you how to get that) and give a user six chances to guess the number. The program should tell the user if the guess is too high or too low. The program should also use the user's name when conversing with the user.

```
#Guessing game

import random

#Initialize counter
tries = 0

#Simple way to input a name from the user
user_name = input('Hello! What is your name? ')

#Get a random number between 1 and 20
number = random.randint(1, 20)

#Start interaction with user.
print ('Well, %s, I am thinking of a number between 1 and 20.' % (user_name))

while tries < 6:

   guess = int(input('Take a guess: '))       #input is string, convert to integer

   tries += 1

   if guess < number:
      print ('Your guess is too low.')

   elif guess > number:
      print ('Your guess is too high.')

   else:
      print ('Good job, %s! You guessed my number in %s guesses!' % (user_name, tries))
      break

if guess != number: print ('Nope. The number I was thinking of was %s.' % (number))
```

The file Gamedata.csv contains data for Auburn men's basketball games from last season. Each line contains the following data: game date, game code, VH code, Auburn's score, opponent's name, and opponent's score. (Note: the columns in the spreadsheet are not labeled.) The game code is: C for conference game, N for nonconference game, or P for tournament game. The VH code is V if Auburn is the visiting team and H if Auburn is the home team.

Read the file and calculate the average points per game as follows.

Average points per game:

Home team, nonconference: xxx	No. of games: xxx
Home team, conference: xxx	No. of games: xxx
Home team, tournament: xxx	No. of games: xxx
Visiting team, nonconference: xxx	No. of games: xxx
Visiting team, conference: xxx	No. of games: xxx
Visiting team, tournament: xxx	No. of games: xxx
Overall home team average: xxx	No. of games: xxx
Overall visiting team average: xxx	No. of games: xxx
Overall average: xxx	No. of games: xxx

```
#This program reads the data in AU gamedata.csv and calculates averages

import csv

cvscore, chscore, cvnum, chnum = 0, 0, 0, 0
nvscore, nhscore, nvnum, nhnum = 0, 0, 0, 0
tvscore, thscore, tvnum, thnum = 0, 0, 0, 0

#Read the data from the file to be read
reader = csv.reader(open('Gamedata.csv', 'r'), delimiter=',')

#Process each row

for row in reader:
        datarow = list(row)
        score = int(datarow[3])
        if datarow[1] == 'C':
                if datarow[2] == 'V':
                        cvscore += score
                        cvnum += 1
                else:
                        chscore += score
                        chnum += 1
        elif datarow[1] == 'N':
                if datarow[2] == 'V':
                        nvscore += score
                        nvnum += 1
                else:
                        nhscore += score
                        nhnum += 1
        else:
                if datarow[2] == 'V':
                        tvscore += score
                        tvnum += 1
                else:
                        thscore += score
                        thnum += 1

cvavg = cvscore / cvnum
chavg = chscore / chnum
nvavg = nvscore / nvnum
```

```
nhavg = nhscore / nhnum
tvavg = tvscore / tvnum
thavg = thscore / thnum

print('Average points per game')
print('Home team, nonconference: %4.2f \tNo of games: %s' % (nhavg, nhnum))
print('Home team, conference: %4.2f \t\tNo of games: %s' % (chavg, chnum))
print('Home team, tournament: %4.2f \t\tNo of games: %s'% (thavg, thnum))
print('Visiting team, nonconference: %4.2f \tNo of games: %s' % (nvavg, nvnum))
print('Visiting team, conference: %4.2f \tNo of games: %s' % (cvavg, cvnum))
print('Visiting team, tournament: %4.2f \tNo of games: %s' % (tvavg, tvnum))

vscore = cvscore + nvscore + tvscore
vgames = cvnum + nvnum + tvnum
hscore = chscore + nhscore + thscore
hgames = chnum + nhnum + thnum
ovavg = vscore / vgames
ohavg = hscore / hgames
tscore = vscore + hscore
tgames = vgames + hgames
oavg = tscore / tgames

print('Overall home team average: %4.2f \tNo of games: %s' % (ohavg, hgames))
print('Overall visiting team average: %4.2f \tNo of games: %s' % (ovavg, vgames))
print('Overall average: %4.2f \t\t\tNo of games: %s' % (oavg, tgames))
```