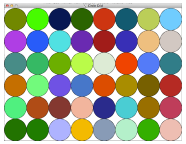
Iteration: while loops, for loops, iteration tables and nested loops



CS111 Computer Programming

Department of Computer Science
Wellesley College

Review: Python **for** loop

Recall that a Python `for` loop performs the loop body for each element of a sequence.

```
nums = range(5)          [0,1,2,3,4]

for num in nums:
    print num * 10
```

```
0
5
10
15
20
```

```
for n in nums:
    if (n % 2) == 0:
        print n
```

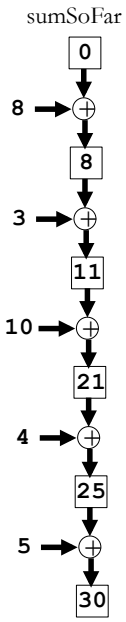
```
0
2
4
```


Accumulating a result with a **for** loop

It's common to use a **for** loop in conjunction with one or more variables ("accumulators") that accumulate results from processing the elements.

E.g., How can we define a sumList function that takes a list of numbers and returns a single number that is their sum?

```
In[ ]: sumList([8,3,10,4,5])
Out[ ]: 30
```



sumList in Python

```
def sumList(numbers):
    sumSoFar = 0
    for num in numbers:
        sumSoFar += num # or sumSoFar = sumSoFar + num
    return sumSoFar
```

initialize accumulator

update accumulator

return accumulator

`for` loop: concatAll

	Returns
`concatAll(['To','be','or','not','to','be'])`	`'Tobeornottobe'`
`beatles = ['John','Paul','George','Ringo']` `concatAll(beatles)`	`'JohnPaulGeorgeRingo'`
`concatAll([])`	`''`

What should the accumulator do in this case?

```
# Given a list of strings, returns the string that results
# from concatenating them all together
def concatAll(elts):
```


`for` loop: countOf

```
sentence = 'the cat that ate the mouse liked
            the dog that played with the ball'
```

`sentence.split()` ➔ `['the','cat','that','ate',… 'ball']`

	Returns
`countOf('the', sentence.split())`	4
`countOf('that', sentence.split())`	2
`countOf('mouse', sentence.split())`	1
`countOf('bunny', sentence.split())`	0
`countOf(3, [1,2,3,4,5,4,3,2,1])`	2

```
# Given a value val and a list elts, returns the
# number of times that val appears in elts
def countOf(val, elts):
```


What is Iteration?

Repeated execution of a set of statements

How does it stop repeating?

When some **stopping condition** is reached (or, alternatively, it continues while some **continuation condition** is true).

In most programming languages, iteration is expressed via **looping constructs**.

Python has **while** and **for** loops.

`while` Loops

`while` loops are a fundamental mechanism for expressing iteration

keyword indicating while loop

a boolean expression denoting whether to iterate through the body of the loop one more time.

```
while continuation_condition :
    statement1
    statement2
    …
    statementN
```

body of loop = actions to perform if the continuation condition is true

while Loop Example: printHalves

```
def printHalves(n):
    while n > 0:
        print(n)
        n = n/2
```

```
In[2]: printHalves(22)
```

What is printed here?

Your Turn: What's the output?

```
def printHalves2(n):
    while n > 0:
        print(n)
    n = n/2
```


Why don't computer scientists ever get out of the shower?

Because the shampoo bottle says:

- Lather
- Rinse
- Repeat

Accumulating a result with a **while** loop

It is common to use a **while** loop in conjunction with one or more variables ("accumulators") that accumulate results from processing the elements.

Define a sumHalves function that takes an nonnegative integer and returns the sum of the values printed by printHalves.

```
In[3]: sumHalves(22)
Out[3]: 41 # 22 + 11 + 5 + 2 + 1
```

```
def sumHalves(n):
    sumSoFar = 0
    while n > 0:
        sumSoFar = sumSoFar + n # or sumSoFar += n
        n = n/2
    return sumSoFar
```

initialize accumulator

update accumulator

return accumulator

Iteration Tables

An iteration is characterized by a collection of **state variables** that are updated during each step of the process. E.g the state variables of sumHalves are n and sumSoFar.

The execution of an iteration can be summarized by an **iteration table**, where columns are labeled by state variables and each row represents the values of the state variables at one point in time.

Example: iteration table for sumHalves(22):

step is not a state variable but a label that allows us to distinguish rows

step	n	sumSoFar
0	22	0
1	11	22
2	5	33
3	2	38
4	1	40
5	0	41

Iteration Rules

An iteration is governed by

- **initializing the state variables** to appropriate values;
- specifying **iteration rules** for how the next row of the iteration table is determined from the previous one;
- specifying the **continuation condition** (alternatively, stopping condition)

step	n	sumSoFar
0	22	0
1	11	22
2	5	33
3	2	38
4	1	40
5	0	41

initial values of state variables

continue while n > 0 (stop when n <= 0)

Iteration rules for sumHalves:

- next sumSoFar is current sumSoFar plus current n.
- next n is current n divided by 2.

Printing the iteration table in a loop

By adding a print statement to the top of a loop and after the loop, you can print each row of the iteration table.

```
def sumHalvesPrint(n):
    sumSoFar = 0
    while n > 0:
        print 'n:', n, '| sumSoFar:', sumSoFar
        sumSoFar = sumSoFar + n # or sumSoFar += n
        n = n/2
    print 'n:', n, '| sumSoFar:', sumSoFar
    return sumSoFar
```

```
In[4]: sumHalvesPrint(22)
n: 22 | sumSoFar: 0
n: 11 | sumSoFar: 22
n: 5 | sumSoFar: 33
n: 2 | sumSoFar: 38
n: 1 | sumSoFar: 40
n: 0 | sumSoFar: 41
Out[17]: 41
```


Your Turn: What is the result?

```
def sumHalves2(n):
    sumSoFar = 0
    while n > 0:
        n = n/2
        sumSoFar = sumSoFar + n
    return sumSoFar
```

step	n	sumSoFar
0	22	0
1	11	
2	5	
3	2	
4	1	
5	0	

Your Turn: What is the result?

```
def sumHalves3(n):
    sumSoFar = 0
    while n > 0:
        sumSoFar = sumSoFar + n # or sumSoFar += n
        n = n/2
        return sumSoFar
```


Your turn: **sumBetween** with while loop

```
In[6]: sumBetween(4,8)
Out[6]: 30 # 4 + 5 + 6 + 7 + 8
```

step	lo	hi	sumSoFar
0	4	8	0
1	5	8	4
2	6	8	9
3	7	8	15
4	8	8	22
5	9	8	30

```
# Returns the sum of the integers
# from lo up to hi (inclusive).
# Assume lo and hi are integers.
# sumBetween(4,8) returns 30
# sumBetween(4,4) returns 4
# sumBetween(4,3) returns 0
def sumBetween(lo, hi):
```


While loops and user input

```
name = raw_input('Please enter your name: ')
while (name.lower() != 'quit'):
        print 'Hi,', name
        name = raw_input('Please enter your name: ')
print('Goodbye')
```

```
Please enter your name: Ted
Hi, Ted
Please enter your name: Marshall
Hi, Marshall
Please enter your name: Lily
Hi, Lily
Please enter your name: quit
Goodbye
```

A while loop you may have encountered:

```
password = raw_input('Password: ')
while not isValid(password): # assuming isValid is written
        print 'Sorry, invalid password.'
        name = raw_input('Password: ')
print('Your password has been successfully updated.')
```


for loops are **while** loops in disguise!

```
# Sums the integers between lo and
# sumList([17,8,5,12]) returns 42
# sumList(range(1,11)) returns 55
def sumListFor(nums):
    sumSoFar = 0
    for n in nums:
        sumSoFar += n # or sumSoFar = sumSoFar + n
    return sumSoFar

# If Python did not have a for loop, the above for loop
# could be automatically translated to the while loop below
def sumListWhile(nums):
    sumSoFar = 0
    index = 0
    while index < len(nums):
        n = nums[index]
        sumSoFar += n # or sumSoFar = sumSoFar + n
        index += 1  # or index = index + 1
    return sumSoFar
```

Returning early from a loop

In a function, **return** can be used to exit the loop early (e.g., before it visits all the elements in a list).

```
def isElementOf(val, elts):
    for e in elts:
        if e == val:
            return True # return (and exit the function)
                        # as soon as val is encountered
    return False  # only get here if val is not in elts
```

```
In [1]: sentence = 'the cat that ate the mouse liked
                    the dog that played with the ball'

In [2]: isElementOf('cat', sentence.split())
Out[2]: True # returns as soon as 'cat' is encountered

In [3]: isElementOf('bunny', sentence.split())
Out[3]: False
```


Your turn

```
containsDigit('The answer is 42')          True
containsDigit('pi is 3.14159...')          True
containsDigit('76 trombones')              True
containsDigit('the cat ate the mouse')     False
containsDigit('one two three')             False
```

Returns

```
def containsDigit(string):
```


Terminating a loop early: **break**

The **break** command is used to exit from the **innermost** loop in which it is used (can be used within functions too)

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break  # exit the loop
    total += x
    x = x/2
```


Your turn: **areAllPositive**

```
# Given a list of numbers, check if all the elements
# in the list are positive
# areAllPositive([17, 5, 42, 16, 31]) returns True
# areAllPositive([17, 5, -42, 16, 31]) returns False
# areAllPositive([-17, 5, -42, -16, 31]) returns False
# areAllPositive([]) returns True
def areAllPositive(listOfNums):
```

Your turn: **indexOf**

```
# Given a value val and a list elts, returns
# the first index in elts at which val appears.
# If val does not appear in elts, returns -1.
# indexOf(8, [8,3,6,7,2,4]) returns 0
# indexOf(7, [8,3,6,7,2,4]) returns 3
# indexOf(5, [8,3,6,7,2,4]) returns -1
def indexOf(val, elts):
```


Loop Design: longestConsonantSubstring

(Extra practice for later)

```
# Given a string, returns the longest substring of
# consecutive consonants. If more than one such
# substring has the same length, returns the first
# to appear in the string.
# longestConsonantSubstring('strong') returns 'str'
# longestConsonantSubstring('strengths') returns 'ngths'
# longestConsonantSubstring('lightning') returns 'ghtn'
# longestConsonantSubstring('Program') returns 'Pr'
# longestConsonantSubstring('adobe') returns 'd'
def longestConsonantSubstring(s):
  # This is hard! Draw iteration tables first!
  # What state variables do you need?
```


Nested Loops

We can have one loop nested in the body of another loop.
An example with lists of numbers:

```
listA = [0, 1, 2, 3]
listB = [2, 5]

for A in listA:
  for B in listB:
      print(str(A) + '*' + str(B) + ' = ' + str(A*B))
```

```
0*2 = 0
0*5 = 0
1*2 = 2
1*5 = 5
2*2 = 4
2*5 = 10
3*2 = 6
3*5 = 15
```


Nested Loops

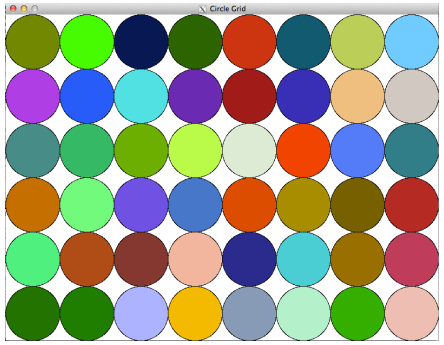
An example with strings:

```
for letter in 'cs':
    for letter2 in 'rocks':
       print letter + letter2
```

```
cr
co
cc
ck
cs
sr
so
sc
sk
ss
```

Nested Loops

In graphics, nested **for** loops can be used to create two-dimensional patterns. Here's a picture involving a grid of randomly colored circles with radius = 50 on a 800x600 canvas. An exercise for later: create this picture using two nested **for** loops and the **Color.randomColor()** function.



Yikes! Neglect to update state variable in loop

```
def printHalvesBroken(n):
    while n > 0:
        print(n)
    n = n/2 # n never changes in loop!
```

```
In[2]: printHalvesBroken(22)
22
22
22
22
22
22
22
22
...
```

An "infinite loop" (in Canopy, stop with Ctrl-C Ctrl-C)

Yikes! Variable update order matters

```
def sumHalvesBroken(n):
    sumSoFar = 0
    while n > 0:
        n = n/2 # updates n too early!
        sumSoFar = sumSoFar + n
    return sumSoFar
```

```
In[3]: sumHalvesBroken(22)
Out[3]: 19
```

step	n	sumSoFar
0	22	0
1	11	11
2	5	16
3	2	18
4	1	19
5	0	19

Yikes! Premature return

```
def sumHalvesBroken2(n):
    sumSoFar = 0
    while n > 0:
        sumSoFar = sumSoFar + n # or sumSoFar += n
        n = n/2
        return sumSoFar # wrong indentation!
                        # exits function after first
                        # loop iteration. Sometimes we
                        # want this, but not here!
```

```
In[4]: sumHalvesBroken2(22)
Out[4]: 22
```