Iteration: 
**while** loops, **for** loops, iteration tables

---

CS111 Computer Programming
Department of Computer Science
Wellesley College

---

**What is Iteration?**
Repeated execution of a set of statements

Keep repeating….

until **stopping** condition is reached

[5, 9, 7, 8]

Stopping condition of **for** loop: list is over

---

**High-level motivation for iteration**

Display time until no more time left

Hillary shimmy; Shaq shimmy;

Play until blocks too small to stack

Keep coding until all test cases passed

---

**How does a for loop work?**
Execution model of a **for** loop

nums = [5, 9, 7, 8]

sumSoFar = 0

for n in nums:
    sumSoFar += n

print sumSoFar

29 sumSoFar

8 n

29
Some **for** loop examples

A **for** loop performs the loop body for each element of a sequence.

```python
word = 'boston'
for i in range(len(word)):
    print(i, word[i])
```

We can also loop directly over the string if we don't need indices.

```python
word = 'boston'
for c in word:
    print(c)
```

More **for** loop examples

```python
nums = [2, -5, 1, 3]
for n in nums:
    print(n * 10)
```

```python
sumSoFar = 0
for n in nums:
    sumSoFar += n
print(sumSoFar)
```

What if we don't know in advance when something will be over?

- Stopping condition of **for** loop: list is over

```python
[5, 9, 7, 8]
```

- Example: repeatedly ask user for input until they say to stop

```
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
```

Another construct: **while** loops

**while** loops are a fundamental mechanism for expressing iteration.

```python
while continuation_condition:
    statement1
    statement2
    ...
    statementN
```

Keyword indicating while loop

- **while**
- **continuation_condition**
- **statement1**, **statement2**, ..., **statementN**

Obligatory boolean expression denoting whether to iterate through the body of the loop one more time.
**while** loops and user input

```python
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

**while** loops are not just for user input.

Useful for other problems too.

---

**while** Loop Example: `printHalves`

```python
def printHalves(n):
    '''Prints positive successive halves of n'''
    while n > 0:
        print(n)
        n = n/2
```

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

What is printed here?

---

A slight variation of `printHalves`:

```python
def printHalves2(n):
    '''Attempts to print positive successive halves of n'''
    while n > 0:
        print(n)
        n = n/2
```

What's the output? `printHalves2(22)`

```python
In[2]: printHalves2(22)
```

---

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

Because the shampoo bottle says:

- Lather
- Rinse
- Repeat

---
Accumulating a result with a \texttt{while} loop

It is common to use a \texttt{while} loop with “accumulators” that accumulate results from processing the elements.

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

\begin{verbatim}
def sumHalves(n):
    sumSoFar = 0
    while n > 0:
        sumSoFar = sumSoFar + n  # or sumSoFar += n
        n = n/2
    return sumSoFar
\end{verbatim}

\begin{verbatim}
In [3]: sumHalves(22)
Out[3]: 41 # 22 + 11 + 5 + 2 + 1
\end{verbatim}

Iteration Tables

An iteration is a step-by-step process characterized by a collection of \texttt{state variables} that determine the next step of the process from the current one. E.g the state variables of \texttt{sumHalves} are \texttt{n} and \texttt{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 \texttt{sumHalves(22)}:

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{step} & \textbf{n} & \textbf{sumSoFar} \\
\hline
0 & 22 & 0 \\
1 & 11 & 22 \\
2 & 5 & 33 \\
3 & 2 & 38 \\
4 & 1 & 40 \\
5 & 0 & 41 \\
\hline
\end{tabular}
\end{center}

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)

\begin{verbatim}
Iteration rules for \texttt{sumHalves}:
\begin{itemize}
    \item next \texttt{sumSoFar} is current \texttt{sumSoFar} plus current \texttt{n}.
    \item next \texttt{n} is current \texttt{n} divided by 2.
\end{itemize}
\end{verbatim}

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.

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
In[4]: sumHalvesPrint(22)
\end{verbatim}

\begin{verbatim}
Out[17]: 41
\end{verbatim}
What is the result? Fill in the table.

```
def sumHalves2(n):
    '''Prints positive successive halves of n'''
    sumSoFar = 0
    while n > 0:
        n = n/2
        sumSoFar = sumSoFar + n
    return sumSoFar
```

**sumHalves2(22)**

<table>
<thead>
<tr>
<th>step</th>
<th>n</th>
<th>sumSoFar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

```
sumBetween(lo, hi):
    '''Returns the sum of the integers from lo to hi (inclusive). Assume lo and hi are integers.'''
```

```
def sumList(nums):
    '''Returns the sum of the elements in nums'''
    sumSoFar = 0
    for n in nums:
        sumSoFar += n
    return sumSoFar
```

```
def sumListWhile(nums):
    '''Returns the sum of the elements in 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
```

```
def sumListFor(listOfNums):
    '''Returns the sum of elements in listOfNums'''
    sumSoFar = 0
    for n in listOfNums:
        sumSoFar += n
    return sumSoFar
```

Accumulating a result with a for loop

**sumList** should take any list of numbers and return the sum of the numbers

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

```
In [ ]: sumList([5,10,-20])
Out[ ]: -5
```

```
In [ ]: sumList([])
Out[ ]: 0
```

**sumBetween with while loop**

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

<table>
<thead>
<tr>
<th>step</th>
<th>lo</th>
<th>hi</th>
<th>sumSoFar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>8</td>
<td>30</td>
</tr>
</tbody>
</table>

**sumBetween(4,8) returns 30**

**sumBetween(4,4) returns 4**

**sumBetween(4,3) returns 0**
Accumulators with lists of strings

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

What should the accumulator do in this case?

```python
def concatAll(elts):
    '''Returns the string that results from concatenating all elements in elts'''
```

for loop: countOf

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

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

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

```python
def countOf(val, elts):
    '''Returns the number of times that val appears in elts'''
```

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).

```python
def isElementOf(val, elts):
    '''Returns True if val is found in elts; False otherwise'''
    for e in elts:
        if e == val:
            return True  # return (and exit the function)
        return False  # only get here if val is not in elts
```

Premature return done wrong (1)

```python
def isElementOfBroken(val, elts):
    '''Faulty version of returns True if val is found in elts; False otherwise'''
    for e in elts:
        if e == val:
            return True
        return False
```

Always returns after the 1st element without examining the rest of the list.

```python
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
```

```python
In [1]: isElementOfBroken(2, [2, 6, 1])
Out[1]: True
In [2]: isElementOfBroken(6, [2, 6, 1])
Out[2]: False
```
Premature return done wrong (2)

```python
def sumHalvesBroken2(n):
    '''Broken version of returns sum of halves of 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
```

Example of returning early

```python
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
```

Use the built-in `isdigit()` string predicate to check if a character is a digit. E.g.

'4'.isdigit() returns True
'h'.isdigit() returns False

```python
def containsDigit(string):
    '''Returns True if the string contains a number'''
```

```
areAllPositive
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):
    '''Returns True if all elements of listOfNums are positive'''
```

```
indexOf
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):
    '''Returns the first index in elts at which val appears. If val does not appear in elts, returns -1'''
```

**longestConsonantSubstring**

longestConsonantSubstring('strong') returns 'str'
longestConsonantSubstring('strengths') returns 'ngths'
longestConsonantSubstring('lightning') returns 'ghtn'
longestConsonantSubstring('Program') returns 'Pr'
longestConsonantSubstring('adobe') returns 'd'

```python
def longestConsonantSubstring(s):
    '''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.'''
```

Note: This is hard! Draw iteration tables first! What state variables do you need?

**Nested loops**

A **for** loop body can contain a **for** loop.

```
# print the multiplication table from 2 to 5

for i in range(2, 6):
    for j in range(2, 6):
        print i, 'x', j, '=', i*j
```

Inner loop gets executed for each value of i

**Nested Loops**

Here's a picture involving a grid of randomly colored circles with radius = 50 on a 800x600 canvas.

This picture is created using two nested **for** loops and the `Color.randomColor()` function.

What is printed?

```
for letter in ['g', 'p', 'd', 's']:
    for letter2 in ['ib', 'ump']:
        print letter + letter2
```
Variable update order matters

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

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

This table is the solution to slide 7-17.

Simultaneous update example:
Greatest Common Divisor algorithm

- The greatest common divisor (gcd) of integers a and b is largest integers that divides both a and b
  - Eg: gcd(84, 60) is 12
- Euclid (300 BC) wrote this algorithm to compute the GCD:
  - Given a and b, repeat the following steps until b is 0.
    - Let the new value of b be the remainder of dividing a by b
    - Let the new value of a be the old value of b
  - … this is a perfect opportunity for a while loop.

<table>
<thead>
<tr>
<th>step</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>84</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>60</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

Neither of the following two gcd functions works. Why?

```python
# Assume a >= b > 0
def gcdBroken1(a, b):
    while b != 0:
        a = b
        b = a % b
    return a

# Assume a >= b > 0
def gcdBroken2(a, b):
    while b != 0:
        b = a % b
        a = b
    return a
```

Fixing simultaneous update

```python
# Assume a >= b > 0
def gcdFixed1(a, b):
    while b != 0:
        prevA = a
        prevB = b
        a = prevB
        b = prevA % prevB
    return a

# Assume a >= b > 0
def gcdFixed2(a, b):
    while b != 0:
        prevA = a
        prevB = b
        b = prevA % prevB
        a = prevB
    return a
```

Python's simultaneous assignment is an even more elegant solution!

```python
# Assume a >= b > 0
def gcdFixed3(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```