Iteration – Part 1

CS111 Computer Programming
Department of Computer Science
Wellesley College

Iteration 1

Motivation for iteration

Display time until no more time left
Keep coding until all test cases passed
Play until blocks too small to stack

Iteration 1

What is Iteration?
Repeated execution of a set of statements

Keep repeating…. until stopping condition is reached

Stopping condition of for loop: no more elements in sequence

[5, 9, 7, 8]

Concepts in this slide: Definition of iteration; stopping condition.

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

sumSoFar = 0
for n in nums:
    sumSoFar += n
print sumSoFar

nums = [5, 9, 7, 8]
sumSoFar
n
Printed value

sumSoFar
5 14 21 29
n
5 9 7 8

To notice:
- The variable n takes values from the list nums.
- The statement += is repeated as many times as n can have a value.
- When there are no more values for n to take, the loop is exited.

Concepts in this slide: Model of the for loop execution.
Some **for** loop examples

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

```
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.

```
word = 'boston'
for c in word:
    print c
```

What if we don't know in advance when to stop?

- Stopping condition of **for** loop: no more elements in sequence

- 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
```

More **for** loop examples

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

```
sumSoFar = 0
for n in nums:
    sumSoFar += n
print sumSoFar
```

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

```
while continuation_condition:
    statement1
    ...
    statementN
```

- **while** loop keyword indicating a **while** loop
- **continuation_condition** a boolean expression denoting whether to iterate through the body of the loop one more time.
- **body** of loop = actions to perform if the continuation condition is true

**Concepts in this slide:**
- Introducing the need for more flexible stopping condition.
- Syntax of **while** loops.
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')
```

while loops are not just for user input.

Useful for other problems too.

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)

In[2]: printHalves2(22)

22
22
22
22
22
22
...    

while Loop Example: printHalves

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

In[2]: printHalves(22)

Accumulating Pattern

```python
nums = [5, 9, 7, 8]
sumSoFar = 0
for n in nums:
    sumSoFar += n
    print sumSoFar
```

It's common to use a loop in conjunction with one or more variables ("accumulators") that accumulate results from processing the elements. This is known as the "accumulating pattern", depicted in the picture on the right side.
Accumulating a result with a while loop

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

Below is defined the `sumHalves` function that takes a nonnegative integer and returns the sum of the values printed by `printHalves` (slide 9-10).

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

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

Iteration Tables [Model of execution]

An iteration is a step-by-step process characterized by a collection of state variables that determine the next step of the process from the current one. 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)`:

<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>22</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>41</td>
</tr>
</tbody>
</table>

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)

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.

```python
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)
Out[17]: 41
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>

Compare this function definition to that of `sumHalves` in 9-14. How do the two definitions differ?

---

**Interrupting Loops**

- Sometimes we want to interrupt a loop without iterating over all elements of a sequence. Examples:
  - When we have found an element we're looking for
  - When we're accumulating a value through a `for` loop and reached some desired value
- There are two situations when we can do this:
  - Within a function, via a `return` statement
  - Within a block of code, via a `break` statement

**Important:**
- We will not cover `break` in this lecture.
- `return` always exits the body of a function.
- We’ll use `break` when we want to exit a loop, but not the function.

---

**for loops are disguised while loops!**

```
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

sumListWhile([17,8,5,12]) returns 42
sumListWhile(range(1,11)) returns 55
```

**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):
    '''Returns True if val is found in elts; False otherwise'''
    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
```
Premature return done wrong (1)

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

The function always returns after the 1st element without examining the rest of the list.

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 within the loop.
Function returns after first iteration

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

Test your knowledge

1. Given a for loop over a sequence, how many times will the statements within the loop executed?
2. How does a for loop differ from a while loop? How are they similar?
3. Can you translate into English the line: `while continuation_condition:`?
4. Can you think of everyday activities in your life that are basically loops?
5. Can you think of examples of the accumulating pattern in everyday life? What are the equivalents for the “accumulators”?
6. What is an infinite loop?
7. Can a for loop be infinite? How? Can a while loop be infinite? How?
8. What errors in the Python code could lead to an infinite loop?
9. What do the columns in the iteration table represent? What do the rows represent?
10. Was the result of the iteration table in slide 9-17 the same or different from that in slide 9-14? What can you learn from these two examples?
11. Only by reasoning about the problems in 9-23 (no need to write code yet), which of them needs to be solved with the accumulating pattern?