Module 8

Topics:

•Iterative structure in Python

Readings: ThinkP 7

Python can be like Scheme ...

```
## count_down: int[>=0] ->
##             (listof int[>=0])
## Produces the list
## [x, x-1, x-2, ..., 1,0]
def count_down(x):
    if x == 0:
        return [0]
    else:
        return [x] + count_down(x-1)
```


But it can be different → Iteration

```
def count_down(x):                  #L1
    answer = []                     #L2
    while x >= 0:                   #L3
        answer.append(x)            #L4
        x = x – 1                   #L5
    return answer                   #L6
```

What happens when we call **`count_down(3)`**?

Calling **`count_down(3)`**

- **`L1, L2`**: **`x ← 3, answer ← []`**
- **`L3`**: Since **`x>=0`**, execute **`L4, L5`**:
  - **`answer ← [3], x ← 2`**
- Now, return to **`L3:`** since **`x>=0`**, execute **`L4, L5`**:
  - **`answer ← [3,2], x ← 1`**
- Now, return to **`L3:`** since **`x>=0`**, execute **`L4, L5`**:
  - **`answer ← [3,2,1], x ← 0`**
- Now, return to `L3:` since **`x>=0`**, execute **`L4, L5`**:
  - **`answer ← [3,2,1,0], x ← -1`**
- Now, return to **`L3:`** since **`x<0`**, do not execute **`L4, L5`**
- **`L6`**: **`return [3,2,1,0]`**

`while` loop basics

- If the continuation test is **`True`**,
  - Execute the loop body
- If the continuation test is **`False`**,
  - Do not execute the loop body
- After completing the loop body:
  - Evaluate the continuation test again
- The body usually includes a mutation of variables used in the continuation test

`while` loop template

```
## initialize loop variables
while test:
  ## body, including statements to:
  ## - update variables used in test
  ## - update value being calculated
## additional processing
```

Steps for writing a **while** loop

You must determine
- how to initialize variables outside the loop
- when the loop body should be executed, or, when it should stop
- what variables must be updated in the loop body so the loop will eventually stop
- what other actions are needed within the loop body

Note: these can be determined in any order – just fill in the template!

Beware of "infinite loops"

```
while 0 == 0:
  print 'runs forever'

x = -5
total = 0
while x < 0:
  total = 2.0 ** x
  x = x-1
print total
```

Notes:
- *it is impossible to write a program that identifies if a loop will run indefinitely (more in CS360)*
- *The code will eventually be terminated in WingIDE with an error – it isn't really "infinite"*

Example: Checking Primality

A number **n>=2** is prime if it has no factors other than 1 and itself.

To test if a number **n** is prime:
- Check every number from 2 to **n-1**
- If you find a factor of **n**, stop and return **False**
- If none of them are, stop and return **True**

Implementation of **`prime`**

```
## is_prime: int[>=2] -> bool
def is_prime (n):
    test_factor = 2
    while test_factor < n:
        if n % test_factor == 0:
            return False
        else:
            test_factor = test_factor + 1
    ## tried all the numbers from 2 to n-1
    return True
```


Testing a **`while`** loop

Include tests, when possible, for which the body executes

- zero times
- exactly one time
- a "typical" number of times
- the maximum number of times

Also, if the continuation test involves multiple conditions, test each way that the loop may terminate

Testing **`is_prime`**

Consider the following test cases:

- **`n=2`** (loop body does not execute)
- **`n=3`** (loop body executes once, terminates because **`test_factor`** equals **`n`**)
- **`n=4`** (loop terminates because 2 is a factor)
- **`n=5`** (maximum iterations, no factors found)
- **`n=77`** (larger composite number)
- **`n=127`** (larger prime number)

Exercise: `factorial`

Write a Python function to calculate **`n!`**

- Use a **`while`** loop that counts from 1 to **`n`**
- Use a **`while`** loop that counts down from **`n`** to 1

Why use loops instead of recursion?

- Iteration, like accumulative recursion, may allow for a more “natural” solution
- Python won’t let us recurse thousands of times
- Iteration is memory efficient
  - for each recursive call, we need memory for parameters
  - for an iterative call, we may just need to update an existing variable
- Iteration will generally run faster

Another type of loop: `for`

- While loops are called *guarded* iteration:
  - If the test evaluates to **`True`**, execute the body
- Another approach:
  - Iterate over all members in a collection
  - Called *bounded* iteration

```
for item in collection:
  loop_body
```

for loop examples

```
for food in ['avocado', 'banana',
  'cabbage']:
    print food.upper()

for base in 'ACGGGTCG':
    print base

for i in range(2,5):
    print i*i

for j in range(10,2,2):
    print j
```


for and while

while

- Loop counter should be initialized outside loop
- Includes continuation test before body
- Should update loop variables in body of loop
- Body contains steps to repeat

for

- Loop counter initialized automatically
- Continues while more elements in collection
- Loop variable updated automatically – do not update in loop
- Body contains steps to repeat

Multidimensional Lists and Nested Loops

```
M = [[1,2,3], [[4],[5, 6]], [7,8,9],
     [10, []]]
print M[0]
print M[0][2]
print M[1][1][1]
print M[3][0]
print len(M)
print len(M[0])
print len(M[1])
print len(M[1][1])
print len(M[3][1])
```

```
## nested_max:
##(listof (listof int)[nonempty])[nonempty]
##        -> int
## find the largest value in a list of lists
## Examples:
## nested_max([[1,5,3], [3], [35,1,2]]) =>  35
def nested_max(alol):
    ## set the initial value
    cur_max = alol[0][0]
    for L in alol:      # each list in alol
        for elem in L:  # each value in L
            if elem > cur_max:
                cur_max = elem
    return cur_max
```


Determining Run-time of code involving loops

- Determine the number steps performed outside the loop
- Determine the number of iterations of the loop
- Determine the number of steps performed in each iteration
- Add everything together to get the overall running time

What is the run-time of this function for a list of length n?

```
def mystery(L):
    M = []
    index_range = range(len(L))
    for index in index_range:
        rest = L[index:]
        new_val = max(rest)
        M = M + [new_val]
    return M
```

Application: Searching a list

Suppose you have a list **L**. How could you determine if a particular value is in that list, if **L** is in no particular order?

Algorithm (called Linear Search)

- Check the first element in **L**: is it the one?
  - If Yes, return True
  - Else, check the next value
- The value is not in the list if you don't find it

Implementing Linear Search

```
## linear_search: (listof X) X
##                -> bool
## produces True if target is in L,
## False otherwise
## Note: equivalent to: target in L
def linear_search (L, target):
    for val in L:
        if val == target:
            return True
    return False
```


Running Time of `linear_search`

- Let `n = len(L)`
- Best Case:
  - If target is in first position, we find it right away → `O(1)`
- Worst Case:
  - It target is not in **L**, we have to check all `n` elements → `O(n)`
  - What is the other worst case?

Alternatives to Linear Search

- If `L` is unsorted, we can't do any better than Linear Search.
- How could we improve Linear Search if `L` was sorted into increasing order?
  - Are there situations in which we could stop earlier?
  - Is this any faster in the worst case?

A better approach: Binary Search

- Suppose **`L`** is a listing of the taxpayers in Canada, sorted by Social Insurance numbers.
- Approximately 22,000,000 entries
- Look at **`L[11000000]`**
- Is it the **`target`** taxpayer?
  - If yes, stop.
  - If not, is **`target < L[11000000]`** ?
    - If yes, then **`target`** is in the first half of **`L`**
    - If not, then **`target`** is in the second half of **`L`**
  - Repeat this process for the half containing **`target`**

Developing **`binary_search`**

- We need to determine how to keep track of the section of the list still being searched
  - →Variables **`beginning`**, **`end`**
  - →Determine their initial values
- Determine the **`middle`** position
- If **`L[middle]`** is target, return **`True`**
- Otherwise, update **`beginning`** and **`end`**
- Determine when we to continue (or stop) searching

Starting the implementation

```
def binary_search(L, target):
    beginning = …
    end = …
   while …:
        middle = …
        if L[middle] == target:
            return True
        elif L[middle] > target :
            …
        else:
            …
    return False
```


`binary_search` tests should include

- empty list
- list of length 1: target in list and not in list
- small list, both even and odd lengths
- larger list
  - **`target`** "outside" list, i.e. **`target< L[0]`** or **`target > L[len(L)-1]`**
  - **`target`** in the list, various positions (first, last, middle)
  - **`target`** not in the list, value between two list consecutive values

Worst Case running time of **`binary_search`**

Suppose n = 2^k:

- First comparison reduces list size to 2^{k-1}
- Second comparison reduces list size to 2^{k-2}
- Third comparison reduces list size to 2^{k-3}
- …
- m^{th} comparison reduces list size to 2^{k-m}
- How many comparisons until we reduce list size to 1?

Comments on running time for **`binary_search`**

- Worst case running time is **`O(log n)`**
  - For n ~ 1000, will consider at most 11 elements (2^{10} = 1024)
  - For n ~100,000, will consider at most 17 elements (2^{17} = 131072)
  - For n ~ 22,000,000, will consider at most 25 elements (2^{25}=33,554,432)
  - Doubling the size of list requires 1 more comparison worst-case!!!!

Comments and Questions on running time for **`binary_search`**

- Binary search isn't faster than linear search all the time *(Q1: in what situation is linear faster?)*
- Could modify it to produce something other than a boolean *(Q2: what would be a good value?)*
- Could be written recursively instead in Python and still have worst case run-time of **`O(log n)`** *(Q3: would be the worst cast for a recursive implementation in Scheme still be **`O(log n)`**?)*

Didn't we do something like this in CS115?

- In CS115, you studied searching in binary search trees (bst)
- You looked at the root value, and either searched the left or right
- The technique is the similar, but
  - In binary search tree search, there is no guarantee that the left and right subtrees were about the same size, so its worst case is not **`O(log n)`**

Goals of Module 8

- Understand that iteration is central to Python
- Understand the difference between **`while`** and **`for`** loops
- Be able to write a loop to solve a problem
- Understand how binary search works and why it is much faster than linear search
- Understand the basics of determining run-time of code involving loops