Module 06

Topics:
• Iterative structure in Python

Readings: ThinkP 7
In Python, repetition can be recursive

```python
## count_down: Nat -> (listof Nat)
## Returns the list
## \[x, x-1, x-2, \ldots, 1,0\]
def count_down(x):
    if x == 0:
        return [0]
    else:
        return [x] + count_down(x-1)
```
def count_down(x):
    answer = []
    while x >= 0:
        answer.append(x)
        x = x - 1
    return answer

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

- **L1, L2:** \(x \leftarrow 3, \text{ answer } \leftarrow []\)

- **L3:** Since \(x \geq 0\), execute **L4, L5**:
  - \(\text{answer } \leftarrow [3], \ x \leftarrow 2\)

- **Now, return to L3:** since \(x \geq 0\), execute **L4, L5**:
  - \(\text{answer } \leftarrow [3, 2], \ x \leftarrow 1\)

- **Now, return to L3:** since \(x \geq 0\), execute **L4, L5**:
  - \(\text{answer } \leftarrow [3, 2, 1], \ x \leftarrow 0\)

- **Now, return to L3:** since \(x \geq 0\), execute **L4, L5**:
  - \(\text{answer } \leftarrow [3, 2, 1, 0], \ x \leftarrow -1\)

- **Now, return to L3:** since \(x < 0\), do not execute **L4, L5**

- **L6:** \text{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!
Example: Checking Primality

A number $n \geq 2$ 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 $\text{False}$
• If none of them are, stop and return $\text{True}$
Implementation of prime

```python
## is_prime: Nat -> Bool
## requires: n >= 2
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 body executes once, terminates because 2 is a factor)
- $n=5$ (maximum iterations, no factors found)
- $n=77$ (larger composite number)
- $n=127$ (larger prime number)
Beware of “infinite loops”

```python
while True:
    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”*
Exercise: factorial

Write a Python function to calculate \( n! \)

• Use a \textbf{while} loop that counts from 1 to \( n \)
• Use a \textbf{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 more 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

```python
for item in collection:
    loop_body
```
for loop examples

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

for base in 'ACCGGGTCG':
    print(base)
**for** loop examples using `range`

```python
def sum_all():
    sum_all = 0
    for i in range(2,5):
        sq = i*i
        sum_all = sum_all + sq
    print(sum_all)

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

• `range` is an iterator, it can generate a collection
  – the next value in the `range` is computed automatically with each pass through the `for` loop.
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, or more values in iterator
• Loop variable updated automatically – do not update in loop
• Body contains steps to repeat
Nested Lists and Loops

In Module 04, we considered simple nested lists like:
\[ L = [[1,2], [], [7,8,9,10]] \]

What is printed by the following?
for \( m \) in \( L \):
    print(sum(m))

What if we want to access all values in a list like \( L \)?
# nested_max(alol) returns the largest value in alol
# nested_max: (listof (listof Int)) -> Int
# requires: alol is nonempty
# Lists in alol are nonempty
# Example:
# 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
Revisiting `multiply_by` example

The function `multiply_by` consumes a list of integers (called `values`) and an integer (called `factor`) and mutates `values` by multiplying each entry in `values` by `factor`. The function returns `None`.

Implement `multiply_by` using a loop.
Question: What is the value of $L$ after the following `for` loop terminates?

$L = [2, 4, 6, 8, 10]$  
`for x in L:`  
    `if x%2==0:`  
        `L.remove(x)`

*Warning*: Do not add/remove entries in a list that you are looping over using a `for` loop
What does this function do?

def mult_table(n):
    table = []
    for r in range(n):
        row = []
        for c in range(n):
            row.append(r*c)
        table.append(row)
    return table

How many total iterations would \texttt{mult_table(5)} involve? \texttt{mult_table(n)} for any Nat \texttt{n}?
What does this function do?

def smaller(L,x):
    p = 0
    while p < len(L):
        if L[p] < x:
            return p
        else:
            p = p+1
    return False

How many iterations would smaller([10,8,6],3) involve? smaller([7,10,2], 8)? smaller(L,x) for any L and x?
Goals of Module 06

• Understand that iteration is central to Python
• Understand the difference between \texttt{while} and \texttt{for} loops
• Be able to use a loop to solve a problem