COMP10001 Foundations of Computing
Iteration, Lists and Sequences

Semester 2, 2018

Chris Leckie & Nic Geard

THE UNIVERSITY OF MELBOURNE

Lecture Agenda

- Last lecture:
  - Conditionals
  - Functions
- This lecture:
  - Iteration
  - Lists
  - Mutability

The Power of `return` I

- In order to use the output of a function (eg to assign it to a variable), we need to `return` a value:

```
def count_digits(n):
    s = str(abs(n))
    return(len(s) - ('.' in s))

print(count_digits(-123.123))
```

- Convert from Celsius to Fahrenheit:

```
def C2F(n):
    return(9*n/5 + 32)

print(C2F(21))
```

The Power of `return` II

- `return` is also a way of (unconditionally and irrevocably) terminating a function:

```
def safe_divide(x,y):
    if y:
        return(x/y)

    print("ERROR: denom must be non-zero")
```

Quiz

What is printed to the screen here?

```
def bloodify(word):
    return(word[:3] + '-bloody-' + word[3:])

w = bloodify('fantastic')
print(w)
```

What is printed to the screen here?

```
def bloodify(word):
    return(word[:3] + '-bloody-' + word[3:])

w = bloodify('andrew')
print(w)
```

Variables and "Scope" I

- Each function (call) defines its own local variable "scope". Its variables are not accessible from outside the function (call)

```
def subtract_one(k):
    k = k - 1
    return(k)

i = 0
n = subtract_one(i)
print(i)
print(n)
print(k)
```

Variables and "Scope" II

- Are the semantics different to the previous slide?

```
def subtract_one(i):
    i = i - 1
    return(i)

i = 0
n = subtract_one(i)
print(i)
print(n)
print(k)
```

Variables and "Scope" III

- Functions can access variables defined outside functions ("global" variables), although they should be used with extreme caution (perhaps never!)

```
def fun1(j):
    fun2(j)
    return(1)
def fun2(k):
    global i,j  # global variables
    print(i,j,k)
    return(2)
i,j,k = 1,2,3
fun1(i)
```

Reasons for Using Functions

- "Archiving" code in libraries
- Removing redundancy
- Ease of testing
- Increasing modularity
- Increasing readability

Iteration

One final, essential tool to make the computer do something over and over again.

- Repeat something forever (eg Windows)
- Repeat something a fixed number of times
  - move Mario forward 10 pixels
  - print 7 copies
  - play Nyan Cat 15 times
- Repeat something until something happens (eg scroll while button held)

Iteration: `while` Loops I

A conditional loop.

- The general idea is that we continue repeating a block of code as long as a given condition holds
- Basic form:

```
while␣<condition>:
␣␣␣␣statement␣block
```

- We use the notion of "block" as in `if` statements, but here, potentially the code block is repeated

```
text = ''
while len(text) != 3:
  text = input('Enter 3-digit code: ')
```

Iteration: `while` Loops II

- Another way to end `while` loops (and bypass the condition in the `while` statement) is via a `break` in the block of code

```
text = ''
while True:
  text = input('Enter 3-digit code: ')
  if len(text) == 3:
    break
  print('Sorry, invalid code.')
```

This prematurely and unconditionally exits from the loop

Iteration: `for` Loops I

A loop over sequences.

- The general idea is that we work our way through (all of) an "iterable" (eg `str`, `tuple`) of items one item at a time, in sequence
- Basic form:

```
for␣<var>␣in␣<iterable>:
␣␣␣␣statement␣block
```

- Note: `in` here is not (quite) the same as the comparison operator of the same name

Iteration: `for` Loops II

- Simple example:

```
sum = 0
for i in (1,2,3):
    sum = sum + i
print(sum)
```

is equivalent to:

```
sum = 0
sum = sum + 1
sum = sum + 2
sum = sum + 3
print(sum)
```

Iteration: `for` Loops III

- More interesting example:

```
vowels = 0
for char in "rhythm":
    if char in "aeiou":
        vowels = vowels + 1
print(vowels)
```

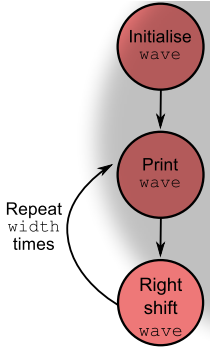
A Useful Function for creating a sequence

- `range(start=0,end,step=1)`: generate a sequence of `int` values from `start` (inclusive) to `end` (non-inclusive), counting `step` at a time

```
>>> for i in range(5):
...     print(i, end=" ")
0 1 2 3 4
>>> for i in range(0,10,2):
...     print(i, end=" ")
0 2 4 6 8
>>> for i in range(10,0,-1):
...     print(i, end=" ")
10 9 8 7 6 5 4 3 2 1
```

`for` Loop Practice: Mexican Wave

- Given the string `wave` made up of a `"Y"` and `width-1` repeats of `"x"`, how can we use a `for` loop to move the `"Y"` across one position to the right at a time?



Choosing between `for` and `while`

- If you need to iterate over all items of an iterable, use a `for` loop
- If there is a well defined end-point to the iteration which doesn't involve iterating over all items, use a `while` loop
- With a `for` loop, avoid modifying the object you are iterating over within the block of code
- Given a choice between the two, `for` loops are generally more elegant/safer/easier to understand

Class Exercise

- Assuming an unlimited number of coins of each of the following denominations:

  $$(1, 2, 5, 10, 20)$$

  calculate the number of distinct coin combinations which make up a given amount N (in cents).

Lists: An Introduction

- To date, we have discussed data types for storing single values (numbers or strings), and tuples for storing multiple things. There is another way to store multiple things: a "list".

```
["head","tail","tail"] # list of strings
[5,5,30,10,50] # list of ints
[1,2,"buckle my shoe",3.0,4.0] # allsorts
```

- As with all types, we can assign a list to a variable:

```
fruit = ["orange","apple","apple"]
```

List Indexing and Splitting

- To access the items in a list we can use indexing (just like we do with strings and tuples):

```
>>> listOfStuff = ["12", 23, 4, 'burp']
>>> listOfStuff[-1]
'burp'
```

- We can similarly slice a list:

```
>>> listOfStuff[:2]
['12', 23]
```

and calculate the length of a list with `len()`

```
>>> len(listOfStuff)
4
```

Class Exercise

- Write code to extract the middle element from the list `l`:

```
>>> l = [1,2,3]
>>> middle(l)
[2]
>>> l = [1,2]
>>> middle(l)
[]
```

- What are the values of `l1` and `l2` after execution of the following code:

```
l1 = [1,2,3,4]
l2 = l1[::-1]
```

But what's the difference?

It seems that tuples and lists are the same, why have both? Important difference: **mutability**

```
>>> mylist = [1,2,3]
>>> mytuple = (1,2,3)
>>> mylist[1] = 6 ; print(mylist)
[1,6,3]
>>> mytuple[1] = 6 ; print(mytuple)
TypeError: 'tuple' object does not support item assignment
```

- Tuples are immutable - they cannot be changed once created
- Lists are mutable - individual elements can be changed

Mutability

Types in Python can be either:

- "immutable": the state of objects of that type cannot be changed after they are created
- "mutable": the state of objects of that type **can** be changed after they are created

Quiz

- Are strings mutable?
- Are lists mutable?
- Are tuples mutable?

Function Arguments I

A key place where mutability is important is when passing arguments to functions.

```
def f(l):
    l[1] = 6

mylist = [1,2,3,4,5]
f(mylist)
print(mylist)

mytuple = (1,2,3,4,5)
f(mytuple)
print(mytuple)
```

Function Arguments II

```
def f(l):
    if type(l) is list:
        l = l + [6]
    else:
        l = l + (6,)

mylist = [1,2,3,4,5]
f(mylist)
print(mylist)

mytuple = (1,2,3,4,5)
f(mytuple)
print(mytuple)
```

Function Arguments III

```
def f(l):
    if type(l) is list:
        l.append(6)
    else:
        l = l + (6,)

mylist = [1,2,3,4,5]
f(mylist)
print(mylist)

mytuple = (1,2,3,4,5)
f(mytuple)
print(mytuple)
```

Lecture Summary

- What is a list?
- What are mutable types?