Week 3: Control Flows

NYC Data Science Academy

OVERVIEW

- Conditionals
  - For loops
  - List Comprehensions
  - While loops
  - Errors and Exceptions

Conditionals in functions

- We have seen boolean functions used in the filter operator. They can also be used inside functions, to do different calculations depending upon properties of the input.
- For example, recall the function `firstelt` that returns the first element of a list:

```
def firstelt(L):
    return L[0]
```

It "crashes" if its argument is the empty list. Suppose we would like it to instead return `None` in that case; `None` is a special value in Python that is often used for this kind of thing. We can do that with a conditional:

```
def firstelt(L):
    if L == []:
        return None
    else:
        return L[0]
```

Conditionals in functions

- The syntax for a conditional in a function is:

```
if condition:                # any boolean expression
    return expression        # return is indented from if
else:                        # else starts in same column as if
    return expression        # return is indented from else
```

- The syntax in lambda definitions is different:

```
lambda x: expression if condition else expression
```

  - For example, here is firstelt in lambda syntax:

```
Firstelt = lambda L: None if L==[] else L[0]
```

Conditionals in functions

- Conditionals can be nested arbitrarily:
  - Return A if c1 is true, B if c1 is false but c2 is true, and C if both are false:

```
if c1:
   return A
else:
   if c2:
      return B
   else:
      return C
```

Conditionals in functions

- Having an if follow an else is so common there is special syntax for it:

```
if c1:
   return A
elif c2:
   return B
else:
   return C
```

Conditionals in functions

- Return A if c1 and c2 are true, B if c1 is true but not c2, C if c1 is false but c3 is true, and D if c2 and c3 are both false:

```
if c1:
   if c2:
      return A
   else:
      return B
elif c3:
   return C
else:
   return D
```

Conditionals in functions

- The nesting of ifs can be as deep as you want.
- The if and its corresponding else must start at the same column. Nested ifs or returns within the true or false branch must be indented.

```
if condition:
    true branch
else:
    false branch
```

- An elif must be at the same indentation level as its corresponding if; the elif itself has a matching else that must be at the same indentation level.

```
if condition1:
    condition1 true branch
elif condition2:
    condition2 true branch
else:
    false branch
```

Exercise 1: Conditionals

- Define these functions using if conditionals:

1. absolute(x) returns the absolute value of x. (Don't use the built-in abs function.)
2. choose(response, choice1, choice2) returns choice1 if response is the string 'y' or 'yes', and choice2 otherwise.
3. leap_year(y) returns true if y is divisible by 4, except if it is divisible by 100; but it is still true if y is divisible by 400. Thus, 1940 is a leap year, but 1900 isn't, but 2000 is.
4. Use filter to define a function leap_years that selects from a list of numbers those that represent leap years.

OVERVIEW

- Conditionals
- For loops
- List Comprehensions
- While loops
- Errors and Exceptions

Loops

- Loops allow the repetition of a statement or list of statements. These statements are called the *body* of the loop.
- Some variables must change in those statements; otherwise, the loop would do the exact same thing each time.
- In addition to a body, a loop has a *header* which says how often the loop should execute. There are two types of loops:
  - *for* loops: The header says exactly what will vary each time the body executes (each *iteration* of the loop), and exactly how many times it will execute (how many iterations).
  - *while* loops: The header has a condition saying when to stop; the number of iterations is not known when the loop starts.

For loops

- A for loop steps through each of the items in a list or tuple (technically, any “iterable” object):

```
for name in collection:
    statements
```

- Each time *statements* is executed - in each *iteration* of the loop - the variable *name* is bound to the next element of the list.
- *You can think of a for loop as being like a map, except that instead of calculating an expression for each element of a list, you perform an action (i.e. execute a set of statements) for each element.*

For loops for printing

- A simple example is printing the elements of a list.

```
words = ['a', 'b', 'c', 'd', 'e']
for w in words:
    print w,     # comma suppresses the newline
a b c d e
```

- Recall that the range function generates a list of numbers:

```
for i in range(len(words)):
    print i, words[i]
0 a
1 b
2 c
3 d
4 e
```

For loops for printing

- There is a function called `enumerate()` in Python that would return both the index and element from the list at the same time.

```
words = ['a', 'b', 'c', 'd', 'e']
for i, e in enumerate(words):
    print i, e
```

```
0 a
1 b
2 c
3 d
4 e
```

Modifying variables in a loop

- In addition to the iteration variable taking on values in a list, you may want other variables to take on different values in each iteration. You can accomplish this by "self-assigning" to those variables. This loop sums the elements of a list:

```
primes = [2, 3, 5, 7, 11]
sum_ = 0
for p in primes:
    sum_ = sum_ + p
sum_
28
```

- The loop body (consisting of the assignment to sum) is executed 5 times. Here are the values of primes and sum at the *start* of each iteration:

sum	0	2	5	10	17
p	2	3	5	7	11

Exercise 2: For loops

- Print the list of prime numbers along with the running sums of those numbers:

```
primes = [2, 3, 5, 7, 11]
sum = 0
for ...      # fill in for loop
2 2
3 5
5 10
7 17
11 28
```

Exercise 2: For loops

- Print a list of strings with numbers determined by the lengths of the strings:

```
names = ['don', 'mike', 'vivian', 'saul']
i = 0
for ...     # fill in for loop

3 don
7 mike
13 vivian
17 saul
```

For loops for mapping

- Using append, we can make a copy of a list:

```
names = ['don', 'mike', 'vivian', 'saul']
copy = []
for name in names:
   copy.append(name)
copy
['don', 'mike', 'vivian', 'saul']
```

- We can get the effect of mapping a function `f` over the list just by changing the body of the loop to: `copy.append(f(name))`.

For loops for mapping

- You should prefer map if you have a choice, because it is more concise and more efficient. But some things are hard to do. For example, doing running sums with a map is hard. So this loop would be hard to write with map:

```
primes = [2, 3, 5, 7, 11]
prime_sums = []
sum = 0
for p in primes:
    sum = sum + p
    prime_sums.append(sum)
prime_sums
[2, 5, 10, 17, 28]
```

Exercise 3: For loops

1. Write a function `map_uc(l)` that takes a list of strings and returns a list of those same strings in all upper-case. You know how to do that using map; do it this time using a for loop. You'll need to create a copy, as you did in the loop, and return that.
2. In the previous exercise, you wrote a loop that produced this output:

```
3 don
7 mike
13 vivian
17 saul
```

   For this exercise, modify that loop to put pairs of these values in a list, instead of printing them, producing:

```
[[3, 'don'], [7, 'mike'], [13, 'vivian'], [17, 'saul']]
```

For loops for files

- We learned how to write string to a file last week when you were talking about file I/O. However, you need to write a lot of string to a file instead of just one.
- Remember this is the syntax for writing a string s to file.
  - Open file for output: `f = open(filename, 'w')`
  - Write a string to the file: `f.write(s)`
  - Close the file: `f.close()`

For loops for files

- We could loop through the list that contains all the strings we want and write each string to the file.
- Suppose we want to write the following output to a file instead of just printing it out.

```
words = ['a', 'b', 'c', 'd', 'e']
for i, e in enumerate(words):
    print i, e

0 a
1 b
2 c
3 d
4 e
```

For loops for files

- In the for loop, if we just call `f.write(i, e)`, will it work?
- In this example, i is an integer and e is a string. So we need to convert i into a string and then concatenate them together.

```
f = open('loop.txt', 'w')
for i, e in enumerate(words):
    s = '%d' % i + e
    f.write(s)
f.close()
```

- But seems like it is not exactly the same as we want. We need to append a newline character at the end of string.

Exercise 4: For loops

- For this exercise, we want to write the key and value pairs from the following dictionary to a file:

```
inventory = {'pumpkin' : 3.99, 'potato': 2,
             'apple' : 2.99}
potato 2
apple 2.99
pumpkin 3.99
```

- Use `.items()` to get the key and value pair of a dictionary.
- The values of the dictionary are of different types, you can use str() function to convert either a float or integer to a string.

OVERVIEW

- Conditionals
- For loops
- List Comprehensions
- While loops
- Errors and Exceptions

List comprehensions

- List comprehensions are another notation for defining lists. They are meant to mimic the mathematical notation of "set comprehensions."
- A list comprehension has the form:

  [*expression* `for` *x* `in` *list* `if` *x satisfies a condition*]

```
[ x * x for x in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]

nested_list1 = [[], ["ab", "cd"], [3, 4, 5]]
[len(l) for l in nested_list1]
[0, 2, 3]

[l[0] for l in nested_list1 if l != []]
['ab', 3]
```

Exercise 5: List comprehensions

- Write list comprehensions to create the following lists:
  1. The square roots of the numbers in [1, 4, 9, 16]. (Recall that math.sqrt is the square root function.)
  2. The even numbers in a numeric list L. Define several lists L to test your list comprehension. (*n* is even if *n* % 2 == 0.)

OVERVIEW

- Conditionals
- For loops
- List Comprehensions
- While loops
- Errors and Exceptions

While loops

- while loops are used when you do not know ahead of time how many iterations you will need:
  - Sum the elements of a list up to the first zero.
  - Newton's method is used to find a zero of an equation. It works by finding values that are closer and closer to the zero, until it finds a value "close enough." But there is no way to know how many times it will have to calculate a new value to get close enough.
  - Get input from a user until the user enters 'quit'.
- With a while loop, you iterate until a given condition becomes false:

```
while condition:
    statements
```

For loops as while loops

- As a first example, this loop prints integers from 0 to 9:

```
i = 0
while i < 10:
    print i
    i = i + 1
```

- This for loop does the same thing:

```
for i in range(0, 10):
    print i
```

- In both loops, the iteration variable is i. The for loop is simpler, but the while loops allows you to do some things that you can't do with the for loop, as we will see.

For loops as while loops

- Let's go through this loop in detail:

```
i = 0
while i < 10:
    print i
    i = i + 1
```

- Before starting the loop, we set i to 0. Here are the steps of the loop:
  - Test the condition: Since i = 0 < 10, execute the body. Print 0 and increment i to 1.
  - Test the condition: Since i = 1 < 10, execute the body. Print 1 and increment i to 2.
  - Test the condition: i = 2 < 10, so print 2 and increment i to 3.
  - Continue until i = 10. Then i < 10 is false, so the loop is terminated.

While loops

- One thing we can do with while loops that is hard to do with for loops is to terminate early. This loops adds up integers starting from 1 until the sum exceeds n:

```
n = 20
i = 1
sum = 0
while sum <= n:
    sum = sum + i
    i = i + 1
sum
21
```

While loops

- This loop is similar, but sums the numbers in a list:

```
n = 20
i = 1
sum = 0
while sum <= n:
    sum = sum + L[i]
    i = i + 1
```

- When we iterate over a list like this, we should also test that we aren't going out of bounds:

```
i = 1
sum = 0
while sum <= n and i < len(L):
    sum = sum + L[i]
    i = i + 1
```

Exercise 6: While loops

- Now we'll do similar loops, but terminate under different conditions. If we're iterating over a list, remember to check that the list index is not out of bounds.
  a. Print the elements of a numeric list, up to the first even number.
  b. Print the elements of a list of strings, up to the first string whose length exceeds 10.
  c. Sum the *even* elements of a numeric list. This loop is different in that it contains an if statement (without an else).

break and continue statements

- The `break` statement immediately terminates the (for or while) loop it is in. This provides a way to terminate the loop from within the middle of the body.
- The `continue` statement terminates *the current iteration* of the loop and goes back to the header.
- This loop adds the values in a list, but ignores negative numbers, and stops if the number exceeds 100:

```
sum = 0
for x in L:
   if x < 0:
      continue
   sum = sum + x
   if sum > 100:
      break
```

Exercise 7: While loops

- Calculate the sum of integers that can be divided by 7 and less than 100. In the following example code, we use `while True`, which means the while loop will keep running until you break it.

```
i = 0
sum = 0
while True:

    # Type your code here

print sum
```

OVERVIEW

- Conditionals
- For loops
- List Comprehensions
- While loops
- Errors and Exceptions

Exceptions

- Exceptions are a language mechanism in Python (and many other languages) for handling unexpected and undesirable situations. Typical examples are:
  - Opening a file that does not exist
  - Dividing by zero
- The exception mechanism allows a program to handle such situations gracefully, without creating a lot of extra code.

Exceptions

- The mechanisms has two parts: signal the exception; and catch the exception.
  - Signal an exception

```
raise Exception
```

  - Catch exception: try

```
try:
    commands
except Exception:
    handle exception
```

An example

- Many predefined functions, or functions you import from modules, can throw exceptions. For example, the function open below raise an error when a file indicated by `filename` does not exist.
- So we focus on handling the errors first:

```
def openfile(filename, mode):
   try:
      f = open(filename, mode)
   except:
      print 'Error:', filename, 'does not exist'
      return
   ... use f ...
```

Built-in Exceptions

- The previous except clause - with no specific exception named - catches all exceptions. However, it is best to be specific about the exceptions you to avoid responding inappropriately.
- For example, the problem of the code below is that we specify a mode that does not exist, but the error message we print out is not true -- existent.txt does exist.

```
def openfile(filename, mode):
    try:
        f = open(filename, mode)
    except:
        print 'Error:', filename, 'does not exist'

openfile('existent.txt', 'no_such_mode')
Error: existent.txt does not exist
```

Exception types

- There are many different exceptions. Here are some of the most common:
  - `Exception`: the most general exception.
  - `TypeError`: the error when you give the wrong type to a function, e.g. 3 + []
  - `ValueError`: the exception when you give a bad value (of the correct type), e.g. int('abc')
  - `IndexError`: when your list subscript is out of bounds, e.g. [][0]
  - `IOError`: when you try to open a non-existent file
- The complete list is here: docs.python.org/2/library/exceptions.html.

Built-in Exceptions

- We can see the type of an error as below:

```
def openfile(filename, mode):
    try:
        f = open(filename, mode)
    except Exception as e:
        print type(e)

openfile('nonexistent.txt', 'r')
openfile('loop.txt', 'no_such_mode')
<type 'exceptions.IOError'>
<type 'exceptions.ValueError'>
```

try statements - general form

- The general form of the try statement, and the meaning of the various parts, is:

```
try:
    statements          # start by executing these
except name:
    statements          # execute if exception "name" was raised
...                     # more named except clauses
except:
    statements          # execute if an exception was raised that
                        # is not named above
else:
    statements          # execute if no exception was raised
finally:
    statements          # execute no matter what
```

try statements - general form

- In our example:

```
def openfile(filename, mode):
    try:
        f = open(filename, mode)
    except IOError:
        print 'File doesn\'t exist in this case.'
    except ValueError:
        print 'Likely to be wrong mode in this case.'
    except:
        print 'Some other error.'
    else:
        print 'No error'
    finally:
        print 'Everybody should have this!'
```

Handling Exceptions

- What we have learned previously is trying to catch an error when you execute the code. However, sometimes we want to raise the error that won't get triggered by the code itself, but it doesn't fit into the context of your function. For example:

```
def cal_volume(x,y,z):
    if x <= 0 or y <= 0 or z <= 0:
        raise ValueError('The value of each dimension \
                          should be greater than 0!')
    else:
        return x*y*z
```

Summary

- We have now seen the major language features of Python:
  - Data structures: lists, tuples, dictionaries, strings
  - Functional operators: map, filter
  - Loops: for, while
  - Exceptions
  - Functions
- All that is left now is to get a lot more practice.
- Next week, we will learn about some popular modules for Python - numpy, matplotlib, and pandas - and also get more practice programming.