logo_CDM

CSC 241 Notes  
Yosef Mendelsohn

With many thanks to Dr. Amber Settle for the original version of these notes.

Prof Settle's notes in turn are based on the Perkovic text.

# Loop patterns: Nested loops

Some problems can only be solved using multiple loops together.

**Practice problem**: Write a function multmult that takes two lists of integers as parameters and returns a list containing the products of integers from the first list with the integers from the second list.

For example, it would be used as follows:

>>> multmult([3, 4, 1], [2, 0])

[6, 0, 8, 0, 2, 0]

**Another Example:**

Imagine a function **lines**() that takes one positive integer n as a parameter and prints on the screen the following n lines:

0

0 1

0 1 2

0 1 2 3

…

0 1 2 3 … n-1

For example:

>>> lines(5)

0

0 1

0 1 2

0 1 2 3

0 1 2 3 4

This is a situation where a nested loop can do the job nicely with minimal code.

def lines(n):

for j in range(n+1):

for i in range(j):

print (i, end = " ")

print()

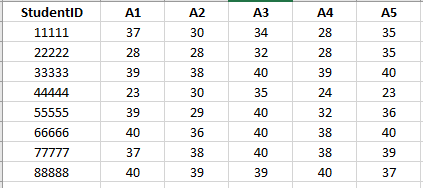
## More on lists: Multi-dimensional lists

A 2-dimensional table has many real-world representations. For

example, think of just about any table or spreadsheet you may have seen:

* Sales of 5 people in your department over 12 months
* Statistics of 10 basketball players (shooting percentage, free throw percentage, fouls committed, etc)

Here is an example of a table showing the scores (out of 40) on 5 homework assignments for a group of students:



Tables such as these are frequently analyzed in data science and other fields using programming code. In Python, 2-dimensional tables can be easily stored as a list of lists.

The lists we’ve seen so far have been one-dimensional. We might think of them as a one-dimensional table.

For example, a list such as:

lst1 = [3, 5, 7] can be viewed as the table:

|  |  |  |
| --- | --- | --- |
| 3 | 5 | 7 |

A two-dimensional table like the following:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 7 | 2 | |
| 5 | 1 | 9 |
| 8 | 3 | 6 |

can be viewed as a list of three rows, with each row being a one-dimensional list:

|  |  |  |
| --- | --- | --- |
| 4 | 7 | 2 |

|  |  |  |
| --- | --- | --- |
| 5 | 1 | 9 |

|  |  |  |
| --- | --- | --- |
| 8 | 3 | 6 |

Can you see what I'm getting at? We can use Python to represent such a two-dimensional table by making it a **list of lists**:

table = [ [4, 7, 2] , [5, 1, 9] , [8, 3, 6] ]

**Note**: In the above line of code I added several (unnecessary) extra spaces to help visualize what is happening. In the real world, we shouldn't space things out this much.

In this example:

* table[0] is holding the list [4,7,2]
* table[1] is holding the list [5,1,9]
* table[2] is holding the list [8,3,6]

To see how to use nested loops to process a two-dimensional list, we need to think about how to **access elements in a two-dimensional list**.

#Recall our original list:

>>> lst = [[4, 7, 2], [5, 1, 9], [8, 3, 6]]

If you are working with a list of lists, then accessing that object with a singleindex gives you an entire sub-list:

>>> lst[0]

[4, 7, 2]

>>> lst[1]

[5, 1, 9]

Accessing a list of lists with two indices gives you a specific element in the specified row.

Here are some examples using the object 'table' we created above:

>>> table[0][0] #first row, first element

4

>>> table[1][2] #second row, third element

9

>>> table[2][1] #third row, second element

3

To summarize:

To access an element in **row i and column j**, we would write: **table[i][j]**

**Example:** If we wanted to get the average of the first number in each row we could do something like:

avg = (table[0][0]+table[1][0]+table[2][0]) / 3.0

But imagine if instead of 3 rows, we had 30, or perhaps 3,000,000? Clearly writing out this code would be impossible. However, now that we are more familiar with nested loops, we can use this technique to analyze this data.

**Aside:** For those of you who have been hearing all the buzz about the field of "data science", this is one of the bread-and-butter techniques used by data scientists to analysze tabular data – which can reach into many, many millions of records.

**Practice problem**: Write a function add2D that takes a 2-dimensional list of integers, adds 1 to each entry in the list, and returns the resulting modified list.

For example, it would be used as follows:

>>> lst = [[4, 7, 2], [5, 1, 9], [8, 3, 6]]

>>> **add2D(lst)**

>>> print(lst)

[[5, 8, 3], [6, 2, 10], [9, 4, 7]]

**Hint**: The indices for the row will be 0 to len(lst), and the indices for the columns will be 0 to len(lst[i]).

# while loop

There is another, more general, looping structure in Python: the while loop.

Recall that a 'for' loop typically executes by using a counter:

eg. going from **i=0** until **i=length of something** such as a list or a range).

Sometimes, though, we don't know how many times a loop should be executed. For example, perhaps you want to repeat a procedure until something happens (e.g. simulate a roll of the dice, and keep going until the program rolls a 7). Or perhaps you need the user to enter a positive number, so you want to looping that code until the user gives you that positive number.

**In a while loop**, there is always a **condition**. The body of the loop gets executed over and over again, as long as the condition is 'True'. The loop will keep repeating on and on untilm at some point, the condition becomes 'False'. The loop will only end when the condition becomes false.

Here is how it works:

1. When Python encounters a '**while**' loop, the very first thing it does is evaluate the conditional. If the conditional is 'True', then Python enters the loop and completes the block of code.
2. After the body of the loop has been executed the program returns to the condition to check if it’s still true.
3. If the condition is still true, the body of the loop gets executed again. As long as the condition remains true, the body continues to be executed.
4. We repeat the above steps over and over and over again until the condition becomes false. When the condition becomes false, the program jumps to the end of the loop, and rest of the program is executed.

while <condition>:

<body>

<rest of program>

Draw the flowchart.

As an example, let's write a function called printSevens that accepts one argument 'x' and outputs all multiples of 7 until x. For example, printSevens(40) would output 7 14 21 28 35.

To do this we start by assigning some counter variable a value 7 and then increment that counter by 7 inside the body of a while loop.

def printSevens(num):

counter=7

while counter<=num:

print(counter, end=" ")

counter+=7

In every iteration the condition counter <= num is checked. When the condition evaluates to False, we are done.

>>> printSevens(40)

7 14 21 28 35

You may have noticed that for this particular example, we could have just as easily used a for loop. Remember that for loops work as long as we have a known starting position, a known stopping position, and a known value to increment the counter. When you have this situation, a for loop is often the best choice.

So let's now look at some examples where a for loop would not work.

**Practice problem 1**: Write a function **enterPos**() that prompts the user to enter a positive number. As soon as the user enters a positive number, the function returns (not prints!) the value. As long as the user continues to enter a negative number, the function re-prompts the user.

Here is what it should look like:

>>> val = enterPos()

Enter a positive whole number (> 0): -3

That number is not as requested. Please try again.

Enter a positive whole number (> 0): -2

That number is not as requested. Please try again.

Enter a positive whole number (> 0): -1

That number is not as requested. Please try again.

Enter a positive whole number (> 0): 5

>>> val

5

>>> enterPos()

Enter a positive whole number (> 0): 2

2

**Review:** If the user enters a floating point number or types in a string, the function should recover gracefully, i.e. your program must not crash. See the interaction below for an example:

>>> val = enterPos\_v2()

Enter a positive whole number (> 0): 3.5

Please enter a whole number using digits.

Enter a positive whole number (> 0): five

Please enter a whole number using digits.

Enter a positive whole number (> 0): [1, 2, 3]

Please enter a whole number using digits.

Enter a positive whole number (> 0): 4

>>> val

4

**Important:** The technique demonstrated in this last example is something you will want to become familiar with. It's a great way to check to make sure that the user gives you valid input. All too frequently users will enter input that can cause your programs to crash. For this reason, programmers in the real world often spend a significant amount of time making sure that the program can handle these situations gracefully and without crashing.

**Practice problem 2**: The code below generates a random number between 1 and 6 and assigns it to the variable diceRoll:

import random

rollResult = random.randrange(1,7)

We will expain this 'random' code in an upcoming lecture. (It's actually kind of fun…)

Using the two lines of code above, let's write a function called rollSix() that keeps outputting the value of rollResult and only stops when the result if a 6.

For example:

**>>> rollSix()**

You rolled a: 4

You rolled a: 1

You rolled a: 6

**>>> rollSix()**

You rolled a: 4

You rolled a: 5

You rolled a: 1

You rolled a: 4

You rolled a: 4

You rolled a: 2

You rolled a: 6

**>>> rollSix()**

You rolled a: 1

You rolled a: 5

You rolled a: 1

You rolled a: 3

You rolled a: 6

**Practice problem 4**: Write a function safeOpen() that asks the user to enter the name of a file. The function then attempts to open the file, and if it succeeds returns the opened file. If it fails to open the file, it again asks the user for the name of a file and tries to open and return it. It continues this process until the user provides a file that can be correctly opened and returned.

The following shows some sample interactions, assuming that the files **none1.txt** and **none2.txt** do not exist in the current directory, but the file **small\_text\_file.txt** does:

>>> inf = safeOpen()

Enter a file name: none1.txt

none1.txt could not be opened.

Please try again.

Enter a file name: none2.txt

none2.txt could not be opened.

Please try again.

Enter a file name: small\_text\_file.txt

>>> print(inf.read())

…

... 🡪 content will be outputted ...

>>> inf.close()

**Note:** Note that in this example, we had to store the refererence to our file in a variable 'inf'. Without it, we would not have been able to output the contents, or to close the file.

As always, my version of the solution to these examples can be found in this week’s solutions file.