DePaul University
College of Computing and Digital Media

CSC 243 NOTES
Fall 2020: Amber Settle

Monday, September 28, 2020

Announcements

- Questions about the second quiz or the first assignment?
- The second assignment is due Thursday, October 1st at 9 pm – questions?

Iteration

Sometimes we need to repeat a block of code multiple times, which we achieve by using a loop.

Here we discuss various looping patterns and their usage. Then we will see the while loop.

Loop patterns: Indexed loops

A very common reason to iterate over a sequence of consecutive integers is to **generate indices of values in a list or characters in a string**.

Iterating through indices is more complex and less intuitive than using the for loop to move through elements of a list.

Why would we use it?

Sometimes it makes more sense to iterate through a sequence by index rather than by value, especially when the location of the item in a list makes a difference for the problem.

Practice problem: Write a function **arithmetic**() that takes a list of integers and returns True if they form an arithmetic progression (i.e. the difference between adjacent items of the list is always the same).

It would be used as follows:

```
>>> arithmetic([3, 6, 9, 12, 15])
True
>>> arithmetic([3, 6, 9, 11, 14])
False
>>> arithmetic([5, 12, 19, 26, 33])
True
>>> arithmetic([5, 12, 19, 26, 32])
False
```

See the solution in the file **loopSols.py**.

Loop patterns: Accumulator loop

A common pattern in loops is to accumulate something in every iteration of the loop.

For example, given a list of numbers we might want to **sum the numbers**.

To do this we need to introduce a variable that will contain the sum. This variable will need to be initialized to 0, and then a for loop is used to add every number in the list to the variable.

```
>>> l = [3, 2, 7, -1, 9]
>>> s = 0
>>> for number in l:
      s = s + number

>>> s
20
```

In this example, **s is the accumulator**. The assignment s = s + n increments the value of s by n.

This operation is so common that there is a shortcut for it: s += n

Recall from the textbook that there is a built-in function sum that can be used to add up the values in a list.

```
>>> l = [3, 2, 7, -1, 9]
>>> sum(l)
20
```

So in this case a solution using a loop wasn't necessary.
But a built-in function may not always be available.

For example, suppose we want to **multiply all the numbers** in the list?

A similar approach as the one we used for the sum would work:

```
>>> l = [3, 2, 7, -1, 9]
>>> p = 0
>>> for i in l:
      p = p * i

>>> p
0
```

Clearly something went wrong. What is the problem?

Anything times 0 is 0, so we won't see the product. Instead we need to initialize p to 1, which is neutral for multiplication.

```
>>> l = [3, 2, 7, -1, 9]
>>> p = 1
>>> for i in l:
      p = p * i

>>> p
-378
```

In the previous two examples the accumulators were of a numerical type.

There can be other types of accumulators. In the following examples, the accumulators are strings and lists.

Practice problem 1: Write a function **upAbbrev** that takes a string as an argument and returns a string consisting of all upper-case characters in the string.

It would be used as follows:

```
>>> upAbbrev("Amber Settle, Ph.D.")
'ASPD'
```

Practice problem 2: An acronym is a word formed by taking the first letters of the words in a phrase and making a word from them. For example, RAM is an acronym for "random access memory".

Write a function **acronym** that takes a phrase as a parameter and returns the acronym for that phrase. Note: The acronym should be all uppercase, even if the words in the phrase are not capitalized.

It would be used as follows:

```
>>> acronym('random access memory')
'RAM'
>>> acronym('internal revenue service')
'IRS'
```

Practice problem 3: Write a function **divisors** that takes a positive integer n as an argument and returns the list of all positive divisors of n.

It would be used as follows:

```
>>> divisors(6)
[1, 2, 3, 6]
>>> divisors(11)
[1, 11]
>>> divisors(12)
[1, 2, 3, 4, 6, 12]
```

See the solutions in **loopSols.py**.

Modules

A module in Python is just a file containing Python code.

The purpose of modules is the same as the purpose of developing and storing programs in a file, namely code reuse. Modules provide a way to organize components (smaller programs) of software systems (larger programs).

The math module

The core Python language supports only basic mathematical operators.

Those include:

- Comparison operators: <, <=, >, >=, ==, !=
- Algebraic operators: +, -, *, /

For other mathematical operators, such as the tangent function, we need the **math module**. The math module is a library of mathematical constants and functions.

In order to use the math module, the module must be explicitly imported into the execution environment. Otherwise we get an error, as can be seen below:

```
>>> sqrt(3)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    sqrt(3)
NameError: name 'sqrt' is not defined
```

The module is imported using the import statement with the general format:

```
import <module>
```

Unfortunately just importing the module isn't sufficient, as the following example illustrates:

```
>>> import math
>>> sqrt(3)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    sqrt(3)
NameError: name 'sqrt' is not defined
```

The square root function is defined in the math module, but the execution environment isn't aware of this. We must tell it explicitly where (i.e. which module) it belongs to, as illustrated below:

```
>>> math.sqrt(3)
1.7320508075688772
```

In the math module, there are two well-known constants defined:

```
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

To summarize, the math module is just a file containing a bunch of functions and constants. As shown above, the module is imported using an import statement which allows us to access the functions defined in the module, although we must use the name of the module in front of the function and constant names.

Loop patterns: Accumulator loop

A common pattern in loops is to accumulate something in every iteration of the loop.

Practice problem 3 redux: Write a function **fdivisors** that takes a positive integer n as an argument and returns the list of all positive divisors of n. Only loop to $\sqrt{n}$.

It would be used as follows:

```
>>> fdivisors(6)
[1, 2, 3, 6]
>>> fdivisors(11)
[1, 11]
>>> fdivisors(12)
[1, 2, 3, 4, 6, 12]
>>>
```

See the solutions in **loopSols.py**.

Character encoding and strings

For many years the standard encoding for characters in the English language was the ASCII encoding (see http://en.wikipedia.org/wiki/ASCII).

ASCII defines a numeric code for 128 characters, punctuation, and a few other symbols common in the American English language.

There are two functions that allow you to manipulate ASCII encoding:

The function **ord**() takes as a parameter a character and returns the ASCII encoding of the character:

```
>>> ord('a')
97
>>> ord('+')
43
```

The function **chr**() is the inverse function of ord(). It takes as a parameter a numeric code and returns the character corresponding to it:

```
>>> chr(97)
'a'
>>> chr(45)
'-'
```

Practice problem: Write a function char(low, high) that prints the characters corresponding to ASCII decimal code i for all values of i from low up to and including high.

See **encodings.py** for the solution.

Loop patterns: Nested loops

Some problems can only be solved using multiple loops together.

To see how to use multiple loops, consider the following (somewhat artificial) problem: Suppose we want to write a function **nested**() that takes a positive integer n and prints to the screen the following n lines:

0 1 2 3 … n-1
0 1 2 3 … n-1
…
0 1 2 3 … n-1

For example:

```
>>> nested(5)
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
```

We've already seen that in order to print one line we can write the following:

```
>>> for i in range(n):
      print(i, end=" ")

0 1 2 3 4
```

In order to get n such lines (5 lines in this case), we need to repeat the above loop n times (5 times in this case). We can do that with an additional outer for loop which will repeatedly execute the above inner for loop:

```
>>> for j in range(n):
      for i in range(n):
            print(i, end = " ")
```

```
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3
4
```

Except this doesn't produce what we wanted! Setting end to a space forces all of the numbers onto a single line, which is not what we want.

We would like to have a new line printed between each line printed by the inner for loop. We can do that as follows:

```
>>> for j in range(n):
      for i in range(n):
            print(i, end = " ")
      print()

0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
```

To see another example of nested loops, consider writing a function **nested2**() 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:

```
>>> nested2(5)

0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
```

What needs to change in the **nested**() function to create the output we want?

nested() prints the complete line 0 1 2 3 … n-1 for each value of j.

What we want to print is:
0 when j is 0
0 1 when j is 1

0 1 2 when j is 2
…
0 1 2 … n-1 when j is n-1

This suggests the following solution:

```
def nested2(n):
    for j in range(n+1):
        for i in range(j):
            print (i, end = " ")
        print()
```

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]
```

See the solution in **solutions.py**.

More on lists: Multi-dimensional lists

As an application area for nested loops, consider a new type of list.

The lists we've seen so far have been one-dimensional, and each can be viewed as a one-dimensional table.

For example, the list `lst = [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 just a one-dimensional list:

4	7	2

5	1	9

8	3	6

Python represents a **two-dimensional table as a list of lists**:

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

This isn't new. But we need to use nested loops to access such lists, which is why two-dimensional lists are an application of nested loops.

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**.

Accessing with just a **single index gives you a row**:

```
>>> lst = [[4, 7, 2], [5, 1, 9], [8, 3, 6]]
>>> lst[0]
[4, 7, 2]
>>> lst[1]
[5, 1, 9]
```

Accessing with two indices gives you a specific element in the specified row and column:

```
>>> lst[0][0]
4
>>> lst[1][2]
9
```

In general, to access an element in **row i and column j**, we would write: **lst[i][j]**

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)
[[5, 8, 3], [6, 2, 10], [9, 4, 7]]
>>> lst
[[5, 8, 3], [6, 2, 10], [9, 4, 7]]
```

See the solution in **solutions.py**.

while loop

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

In a while loop, the body is executed if the condition is true.

But then, after the body has been executed the program goes back to the condition to check if it's still true. If it is, it executes the body again. As long as the condition remains true, the body continues to be executed.

When the condition becomes false, then the rest of the program is then executed.

```
while <condition>:
      <body>
<rest of program>
```

Practice problem: Write a function **greater**() that takes as input two arguments: a list of integers lst and a number num. It should return the index of the first number in the list greater than num. If no element of the list is greater than num, len(lst) should be returned.

An **example** is given here:

```
>>> greater([4, 2, 7, 5, 3, 8, 9, 7], 7)
5
>>> greater([1, 2, 3, 0], 5)
4
```

The solutions can be found in **solutions.py**.