CptS 111, Spring 2023
Lect. #17, Mar. 22, 2023
Class Notes

Today's Agenda:

1. Demo of Lab 8, Task 3
2. `for` -loops vs `while` -loops
3. Nested `for` -loops
4. Lists of lists (nested lists)

---

1. Demo of CptS 111 Fitness App

Ch. 6 (cont.)

Loops (cont.)

2. `for` -loops vs `while` -loops

Thus far in Ch. 6, we've studied counting `for` -loops, iterating `for` -loops, and `while` -loops, but you might be asking: When should we use a `for` -loop and when should we use a `while` -loop?

In general, it's always safer to use a `for` -loop because the number of iterations is fixed using either the `range()` function (counting `for` -loop) or by the finite length of an iterable (iterating `for` -loop). Thus, when the number of iterations is "known," we use a `for` -loop, and we don't have to worry about a loop iterating an infinite number of times as is possible with a `while` -loop.

Consider, for example, the following:

In [1]:
```
# while-loop with counter

i = 0
while i <= 5:               # A while-loop always uses a test expression;
    print(i, end=' ')       # a test expression can be very simple, e.g., T
    i += 1
```

```
0 1 2 3 4 5
```

If we forget to include the `i += 1` statement, the loop will continue on forever:

In [2]:
```
# while-loop that is an infinite loop

i = 0
while i <= 5:
    print(i, end=' ')
```

```
File /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.1
0/site-packages/ipykernel/iostream.py:518, in OutStream.write(self, s
tring)
    509         content = {"name": self.name, "text": data}
    510         self.session.send(
    511             self.pub_thread,
    512             "stream",
   (...)
    515             ident=self.topic,
    516         )
--> 518 def write(self, string: str) -> Optional[int]:  # type:ignore
[override]
    519     """Write to current stream after encoding if necessary
    520
    521     Returns
   (...)
    525
    526     """
```

This can, literally, be very dangerous if, e.g., the loop is controlling some important system in an airplane or is used in a safety critical system!

However, there are times when a `while` -loop is useful, e.g., when you don't know in advance how many times a loop will execute as in PA #5.

In [3]:
```
# while-loop with graceful exit

prompt = 'Enter a fave [or return to stop]: '
faves = []
fave = input(prompt)
while fave != '':               # '' is an empty string
    faves.append(fave)
    fave = input(prompt)
print(faves)
```

```
Enter a fave [or return to stop]: chocolate croissant
Enter a fave [or return to stop]: scones
Enter a fave [or return to stop]: cookies
Enter a fave [or return to stop]: vanilla sweet cream cold brews
Enter a fave [or return to stop]:
['chocolate croissant', 'scones', 'cookies', 'vanilla sweet cream cold
brews']
```

In PA #5, we use the following (or something similar) for our `while` -loop header:

```
while True:
```

Because `True` is always true, we have to insert a `break` statement at some point so we can break out of the loop. Recall that we can actually use almost anything in place of `True` because

In [4]:
```
if 42:
    print('This is a true statement.')
```

```
This is a true statement.
```

In [5]:
```
if 'false':
    print('This is a true statement.')
```

```
This is a true statement.
```

In [6]:
```
if False:
    print('This is a false statement.')
```

In [7]:
```
if []:
    print('This is a false statement.')
```

3. Nested `for` -loops

We can nest loops as many times as necessary to accomplish what we want to do. However, for this class, we'll consider only two nested loops consisting of a single outer loop and a single inner loop. For example,

```
for i in range(4):         # outer loop
    for j in range(3):     # inner loop
```

For nested loops, we start with the outer loop, and then we loop through the inner loop until we've finished every iteration (here there are three of them). Only then do we return to the outer loop to begin the *next* iteration of the outer loop. We continue this until we've completed all iterations of the outer loop (here there are four of them).

```
    i is assigned to 0       # in outer loop     (i is 0)
        j is assigned to 0       # in inner loop (i is still 0)
        j is assigned to 1       #  "    "    "   (i is still 0)
        j is assigned to 2       #  "    "    "   (i is still 0)
```

In [8]:
```
# Inner loop executes 3 times for each execution of outer loop

for i in range(4):           # outer loop gives rows
    print('(i,j) =', end=' ')
    for j in range(3):       # inner loop gives columns
        print(f'({i},{j})', end=' ')
    print()                  # Outside inner loop, but inside outer loop
```

```
(i,j) = (0,0) (0,1) (0,2)
(i,j) = (1,0) (1,1) (1,2)
(i,j) = (2,0) (2,1) (2,2)
(i,j) = (3,0) (3,1) (3,2)
```

How many times is `print()` executed?

In [ ]:
```
# zCA 6.11.3: Want, e.g., 1A 1B 1C 2A 2B 2C for num_rows = 2, num_cols =

num_rows = int(input())
num_cols = int(input())

num = 65
for i in range(num_rows):       # outer loop gives rows
    for j in range(num_cols):   # inner loop gives columns
        print(f'{i+1}{chr(num+j)}', end=' ')
print()
```

4. Lists of Lists (Nested Lists)

A. Indexing

Recall that a simple list is indexed by a single pair of brackets. For example,

```
listA = [1, 2, 3, 4, 5, 6]
```

is indexed by `list1[i]` where `i` varies between 0 and 5.

For nested lists, we use as many pairs of brackets as the depths of the embedded lists. For example,

```
listB = [[1, 2], [3, 4], [5, 6]]
```

`listB` has three elements, i.e., its length is 3. However, each element is itself a list. We can think of these elements as inner lists. `listB` is indexed by `listB[i][j]` where `i` varies between 0 and 2 (because the length of `listB` is 3) and `j` varies between 0 and 1 (because the length of each nested (inner) list is 2).

Another way of looking at indexing is to first consider the index for each element in `listB`. For example, if we use the index for the first element, we get:

```
listB[0] == [1, 2]
```

But suppose now we want to specify the `2`. How do we do this? We need to add an index for it. We do this by adding a second set of brackets, and because the index for `2` in this "inner" list is 1, we end up with:

```
listB[0][1] == 2
```

In [9]:
```
# Let's prove this indexing works!

listB = [[1, 2], [3, 4], [5, 6]]
print('listB is', listB)
print(f'listB[0][1] is {listB[0][1]}.')
print(f'listB[1][0] is {listB[1][0]}.')
print(f'The length of listB is {len(listB)}.')
print(f'The length of listB[-1] is {len(listB[-1])}.')
print(f'listB[-1] is', listB[-1])
```

```
listB is [[1, 2], [3, 4], [5, 6]]
listB[0][1] is 2.
listB[1][0] is 3.
The length of listB is 3.
The length of listB[-1] is 2.
listB[-1] is [5, 6]
```

B. Code Readability

In coding, it's important for our code to be readable by others. To improve readability, we do the following:

- Write blocks of code. Each block of code must use proper indentation.
- Default indentation is 4 spaces, but if there's lots of nesting, decrease to 2. Be consistent or the code won't run.
- Add whitespace and line breaks (\).
- Write long lists, etc., using multiple lines, i.e., can start on one line and continue on others.

This is especially helpful for lists of lists (nested lists) and can be helpful for dictionaries as well. Consider the following list of lists.

In [10]:
```
# 3x3 matrix written as list of lists (nested lists)

table = [
        [1, 2, 3],
        [2, 4, 6],
        [3, 6, 9]
        ]
print(table)
```

```
[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

Suppose we want to print the list of lists above so that it results in the following (zCA6.12.1):

```
1 | 2 | 3
2 | 4 | 6
3 | 6 | 9
```

We can again use the `range(len() - 1)` approach we used for a simple list. We just have to be careful about it and think a little!

In [11]:
```
# zCA 6.12.1: Nested counting for-loops

for i in range(len(table)):                # len() gives number of rows
    for j in range(len(table[i]) - 1):     # len() gives number of columns
        print(table[i][j], '|', end=' ') # print row[i] except for last v
    print(table[i][-1])                    # print last value in row[i]
```

```
1 | 2 | 3
2 | 4 | 6
3 | 6 | 9
```

Note that we had to use `table[i]` in the inner `for` -loop because we need the length of each inner list.

There are a number of other ways to perform the same task:

In [12]:
```
# Using combo of iterating and counting for-loops:

for row in table:
    for i in range(len(row) - 1):
        print(row[i], '|', end=' ')
    print(row[-1], end='')
    print()
```

```
1 | 2 | 3
2 | 4 | 6
3 | 6 | 9
```

In [13]:
```
# Using iterating for-loop, enumerate(), and if-else:

for row in table:
    for i, col in enumerate(row):
        if i != len(row) - 1:
            print(col, '|', end=' ')
        else:
            print(col)
```

```
1 | 2 | 3
2 | 4 | 6
3 | 6 | 9
```

If we want to print the table without the separators, we can simply use nested iterating `for` -loops:

In [14]:
```
# Table without separators

for row in table:
    for col in row:
        print(col, end=' ')
    print()                    # Use so don't end up with a single row.
```

```
1 2 3
2 4 6
3 6 9
```

In [15]:
```
# What happens if we don't include print() in the outer for-loop?

for row in table:
    for col in row:
        print(col, end=' ')
```

```
1 2 3 2 4 6 3 6 9
```

C. A Bit More on Nested Lists

As we've shown above, nested lists can be used to create 2-D tables. I want to emphasize that the number of rows is equal to the length of the outer list, and the number of columns is equal to the length of the inner lists. Thus, for example,

```
nums = [               # 2 rows by 3 columns
       [1, 2, 3],
       [4, 5, 6]
       ]
```

can be used to create a 2x3 table, i.e., one with two rows and three columns, and

```
table = [
        [1, 2],        # row 1 = table[0]
        [3, 4],        # row 2 = table[1]
        [5, 6],        # row 3 = table[2]
        [7, 8]         # row 4 = table[3]
        ]
```

can be used to create a 4x2 table, i.e., one with four rows and two columns.

D. A Bit More on Nested Loops

As we also showed above, the way to create a table from a list of lists is to use nested loops. Importantly, the outer loop is used to work with the rows, and the inner loop is used to work with the columns. For each iteration of the outer loop, the inner loop iterates as many times as there are values in each row.

Let's look at one more example of how to work with a list of lists and nested `for` -loops:

In [18]:
```
# Another example in an attempt to provide insight, 4x3 matrix
# 4 rows
# 3 columns

letters = [
          ['a', 'b', 'c'],
          ['d', 'e', 'f'],
          ['g', 'h', 'i'],
          ['j', 'k', 'l']
          ]

for i, row in enumerate(letters):   # i is index for row
    print(f'i={i+1}')
    for j, col in enumerate(row):   # j is index for letter
        print(f'j={j+1}: {col} ', end=' ')
    print()
```

```
i=1
j=1: a  j=2: b  j=3: c
i=2
j=1: d  j=2: e  j=3: f
i=3
j=1: g  j=2: h  j=3: i
i=4
j=1: j  j=2: k  j=3: l
```