09.1_Iteration(2)

October 16, 2023

1 Introduction to Python for Open Source Geocomputation

- Instructor: Dr. Wei Kang

Content:

- Iterations
  - `while` statement (comparison with `for` statement)
  - `break` and `continue`

2 Iteration with a `while` statement

```
while condition:
    print("sth")
```

- to automate repetitive tasks
- similar to a `for` statement, but explictly checking whether a condition is true
- the flow of execution for a `while` statement:
  1. Determine whether the condition is true or false.
  2. If false, exit the `while` statement.
  3. If the condition is true, run the body and then go back to step 1.

[1]:
```
names=['waldo','rick','reg']
```

[2]:
```
while names:
    name = names.pop()
    print(name)
```

```
reg
rick
waldo
```

Here the condition is that the list **names** is not empty.

In each iteration, `names.pop()` removes the last element from names. So after three iterations, `names` becomes an empty list, and the condition is false, which stops the `while` statement.

[3]:
```
names
```

[3]:
```
[]
```

2.0.1 Group Exercise

Discuss with your group members about the following `while` statement. What is the condition? how is the condition changed by each iteration? What does `while` statement do?

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

> When you are done, raise your hand! I will randomly pick one group to present your answers.

[4]:
```
n = 10
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

```
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

While n is greater than 0, display the value of n and then decrement n. When you get to 0, display the word Blastoff!

2.1 Caveats with `while` statements

- The body of the `while` statement should change the value of one or more variables so that the condition becomes false eventually and the loop terminates.

- Otherwise the loop will repeat forever, which is called an infinite loop.

3 Control iterations with `break` or `continue`

- `break` statement
  - terminate the whole loop
- `continue` statement
  - terminate the current iteration

Both are effective to `while` and `for` statements.

[5]:
```
n = 10
while n > 0:
    n = n - 1
    print(n)
    if n == 5:
        break
print('Blastoff!')
```

```
9
8
7
6
5
Blastoff!
```

[6]:
```
n = 10
while n > 0:
    n = n - 1
    print(n)
    if n == 5:
        continue
print('Blastoff!')
```

```
9
8
7
6
5
4
3
2
1
0
Blastoff!
```

[7]:
```
n = 10
while n > 0:
    n = n - 1
```

```
    if n == 5:
        continue
    print(n)

print('Blastoff!')
```

```
9
8
7
6
4
3
2
1
0
Blastoff!
```

5 is not printed out if the `print` function is called after the `if/continue` statement.

3.0.1 Group Exercise

Using `while` loop and `if` statement to iterate through an integer list and if there is a 100, print it with its index number. The `while` loop stops at the first 100.

`list_a = [1,2, 100, 10, 11, 100]`

When you are done, raise your hand!

[8]:
```
list_a = [1,2, 100, 10, 11, 100]
```

[9]:
```
index = 0
while index < len(list_a):
    if list_a[index] == 100:
        print(index, 100)
        break
    index = index + 1
```

```
2 100
```

[10]:
```
list_a = [1,2, 10, 11]
index = 0
while index < len(list_a):
    if list_a[index] == 100:
        print(index, 100)
        break
    index = index + 1
```

[ ]:
```

```

We can also use a `for` statement to complete the task:

[11]:
```
for index in range(len(list_a)):
    value = list_a[index]
    if value == 100:
        print(100, index)
        break
```

3.0.2 Group Exercise

Using `while` loop and `if` statement to iterate through an integer list and if the integer is not 100, print the integer with its index number.

`list_a = [1,2, 100, 10, 11, 100]`

When you are done, raise your hand!

[12]:
```
list_a = [1,2, 100, 10, 11, 100]
index = 0
while index < len(list_a):
    if list_a[index] != 100:
        print(index, list_a[index])
    index = index + 1
```

```
0 1
1 2
3 10
4 11
```

[13]:
```
index = 0
while index < len(list_a):
    value = list_a[index]
    index = index + 1
    if value == 100:
        continue
    print(index, value)
```

```
1 1
2 2
4 10
5 11
```

[14]:
```
index = 0
while index < len(list_a):
    integer = list_a[index]
    if integer != 100:
        print(integer, index)
    index = index + 1
```

```
1 0
2 1
```

```
10 3
11 4
```

[16]:
```
# infinite loop, why?
index = 0
while index < len(list_a):
    integer = list_a[index]

    if integer == 100:
        continue
    print(integer, index)
    index = index + 1
```

```
1 0
2 1
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[16], line 2
      1 index = 0
----> 2 while index < len(list_a):
      3     integer = list_a[index]
      5     if integer == 100:

KeyboardInterrupt:
```

We run into infinite loops as the update of `index` occurs after the `if-continue` statement. So when `integer` has the value of 100, the iteration stopped and we go back to check the condition - it will always be `True` (we are stuck in the first 100).

One potential fix is to update `index` before the `if-continue` statement

[19]:
```
index = 0
while index < len(list_a):
    integer = list_a[index]
    index = index + 1
    if integer == 100:
        continue
    print(integer, index-1)
```

```
1 0
2 1
10 3
11 4
```

[20]:
```
for index in range(len(list_a)):
    integer = list_a[index]
```

```
    if integer != 100:
        print(integer, index)
```

```
1 0
2 1
10 3
11 4
```

[21]:
```
for index in range(len(list_a)):
    integer = list_a[index]
    if integer == 100:
        continue
    print(integer, index)
```

```
1 0
2 1
10 3
11 4
```

3.1 Nested loops

```
for i in list_a:
    for j in list_b:
        body of the inner loop
    body of the outer loop
```

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop".
- Appropriate indentation is important - this is how python recognize which is the body of the inner loop and which is the body of the outer loop

[22]:
```
persons = [ "John", "Marissa", "Pete", "Dayton" ]
restaurants = [ "Japanese", "American", "Mexican", "French" ]
```

[23]:
```
for person in persons:
    for restaurant in restaurants:
        print(person + " eats " + restaurant)
```

```
John eats Japanese
John eats American
John eats Mexican
John eats French
Marissa eats Japanese
Marissa eats American
Marissa eats Mexican
Marissa eats French
Pete eats Japanese
Pete eats American
Pete eats Mexican
```

```
Pete eats French
Dayton eats Japanese
Dayton eats American
Dayton eats Mexican
Dayton eats French
```

[24]:
```
print?
```

[25]:
```
for i in range(1,4):
    for j in range(1,4):
        print(i*j, end=" ")
    print()
```

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

Using Nested `for` loop to access each individual element from a list within a list

[26]:
```
nested_list = [["university", "city", "state"],
               ["UNT", "Denton", "TX"]]
```

[27]:
```
for i in nested_list:
    for j in i:
        print(j)
```

```
university
city
state
UNT
Denton
TX
```

[28]:
```
a = [[1,0],[1,2]]
```

[29]:
```
a[0][0] = 0
```

[30]:
```
a
```

[30]:
```
[[0, 0], [1, 2]]
```

3.2 Group exercise

Using nested `for` loop to conduct element-wise math calculations on a nested list `list_num = [[1,2], [3,4]]`. Each number inside `list_num` will be multiplied by 10. `list_num` will be updated to be `[[10,20], [30,40]]`

When you are done, raise your hand!

[31]:
```
list_num = [[1,2], [3,4]]
```

[32]:
```
for outer_index in range(2):
    for inner_index in range(2):
        list_num[outer_index][inner_index] = list_num[outer_index][inner_index]␣
 ↪* 10
```

[33]:
```
list_num
```

[33]:
```
[[10, 20], [30, 40]]
```

[34]:
```
list_num = [[1,2,3], [3,4,5]]
```

[35]:
```
for i in range(len(list_num)):
    for j in range(len(list_num[0])):
        list_num[i][j] = list_num[i][j]*10
list_num
```

[35]:
```
[[10, 20, 30], [30, 40, 50]]
```

4 Next Class

- Object oriented programming

4.1 Readings

- Chapters 15, 16!

[ ]:
```

```