Notebook_01_intro_loops

March 17, 2020

1 Repition in Python

1.1 An Introduction to Loops

One reason computers are so useful is their ability to do repititve tasks (the same thing over and over again) both very accurately and very quickly. If you find that you need a program to do the same thing over and over again, you likely need to use a **loop**.

There are two main looping constructs in Python: 1. *while* loops 2. *for* loops

`while` loops are general. You do not necessarily need to know how long (or for how many iterations) the loop must continue. A `while` loop will continue itertaing as long as the logical expression at the start of the loop evaluates to `True`.

`for` loops, on the other hand, are designed for stepping through the items in a sequence (ex. string, list) or other iterable object (ex. np.array or tuple). In a `for` loop, you will have to specify how many iterations you want to execute.

1.2 Syntax of a `while` loop

```
a=0
b=1
while a < b :   # This line is called the "header" line
    #consequent line 1  # Notice the indention
    #consequent line 2
    #and so on
```

The `while` loop is started with a "header line", which include the word "while" followed by a logical (Boolean) expression. This expression is then followed by a colon ":". This syntax is very similar to the `if` statements we talked about previously.

If the logical expression evaluates to `True`, then the "consequent" is executed (or continues to be executed). If the logical expression evaluates to `False` then the "consequent" is not executed (or it ceases to execute).

Each time we reach last line of the consequent, the logical expression is re-evaluated because **the objects that make up the logical expression could have been changed in the loop.**

Here is a basic example of an infinite loop.

[10]:
```
# It will not stop until the user manually breaks it.
import time    # To put in a delay (so we don't get a million lines printed)
while True:
```

```
    print('This is the loop that never ends...')  #You must manually Ctrl-C to␣
↪break it.
    time.sleep(2)    # Pause the code execution for 2 seconds
```

```
This is the loop that never ends...
```

```
      ␣
↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-10-bce28fe280de> in <module>
         3 while True:
         4     print('This is the loop that never ends...')  #You must manually␣
↪Ctrl-C to break it.
    ----> 5     time.sleep(2)    # Pause the code execution for 2 seconds


       KeyboardInterrupt:
```

Here is another example of a `while` loop.

[11]:
```
#In this example, the loop terminates on its own
count = 0
while count < 5:
    print(count)
    count = count + 1
    time.sleep(1)    #Again, just to slow down the code
print('The while loop has ended')
```

```
0
1
2
3
4
The while loop has ended
```

What happens in the above code if "count" is defined as 5 to start? How else can "count" be incremented in this loop?

In the next example, we will use a `while` loop to chop a string down until nothing is left.

[12]:
```
mystr = "Happy St. Patty's Day!"
while mystr:
    print(mystr)
    mystr = mystr[1:]
```

```
Happy St. Patty's Day!
appy St. Patty's Day!
ppy St. Patty's Day!
py St. Patty's Day!
y St. Patty's Day!
 St. Patty's Day!
St. Patty's Day!
t. Patty's Day!
. Patty's Day!
 Patty's Day!
Patty's Day!
atty's Day!
tty's Day!
ty's Day!
y's Day!
's Day!
s Day!
 Day!
Day!
ay!
y!
!
```

In this previous example, a string with any number of characters evaluates to `True` while an empty string evaluates to `False`.

1.3 Control Flow: `break` and `continue`

There are several keywords you may use to alter the natural progression of a loop. These are called "control flow" statements.

- `break`: Jumps out of the closest enclosing loop and past the entire loop
- `continue`: Jumps to the top of the closest enclosing loop (to the header line)

Here is an example of the use of `break` and `continue`. This loop gets 3 tries to guess my favorite color (at random). However, if the random guess is close to the right answer, it won't count against the number of guesses.

[13]:
```
import numpy as np
import time

mycolors =␣
 ↪['red','blue','green','orange','purple','teal','maroon','yellow','magenta','pink']
fav_color = 'red'   #My favorite color
close_colors = ['pink','maroon','magenta']   #These are close to red, so I␣
 ↪won't count them
guesses_left = 3

while guesses_left>0:
```

```
    print('\nGuesses Remaining: ',guesses_left)
    time.sleep(3)  #Just slowing the code down
    index = np.random.randint(low=0,high=9)  #Draw a random integer to use as␣
↪an index
    guess = mycolors[index]
    print('Current guess: ',guess)

    #If the guess is correct, use break to exit the while loop
    if guess == fav_color:
        print('Correct! Ending the loop\n')
        break

    #If the guess is close, use continue to jump back to the top without using␣
↪a guess
    if guess in close_colors:
        print("That's close. I won't count that as a guess\n")
        continue

    #If neither break nor continue have been reached by this point, subtract a␣
↪guess
    guesses_left -= 1

print("That's the end of the guesssing game!")
```

```
Guesses Remaining:  3
Current guess:  blue

Guesses Remaining:  2
Current guess:  maroon
That's close. I won't count that as a guess


Guesses Remaining:  2
Current guess:  magenta
That's close. I won't count that as a guess


Guesses Remaining:  2
Current guess:  yellow

Guesses Remaining:  1
Current guess:  yellow
That's the end of the guesssing game!
```

[ ]: