Week 3 Notebook 2 - While loops data entry and error trapping

August 21, 2018

1 While Loops, Data Input and Basic Error Trapping

As you might guess, while loops are another form of looping. While (no pun intended) for loops have a definite beginning and end based on the parameters in the range() statement, while loops are based on conditionals much like if statements. As long as a specific condition is True, the while loop will continue looping. When the condition becomes False, the while loop ends. Here is a template for while loops:

```
# specify and set the initial condition to True
running = True

# here is the while loop
while running:
    # do some stuff here
    # and do some stuff here if needed
    # at some point you would need
    # to set running = False
    # otherwise the loop will never end

# once the loop ends, the program continues here
```

- The while loop needs to start with a True condition
- The while statement itself contains the condition test
- As long as the condition remains True, the loop continues
- As soon as the condition changes to False, the loop ends
- Once the loop ends, the program continues with the lines after the loop

One common error is to create a loop where the condition NEVER becomes False. In that event, the loop continues. We call this an infinite loop… or as my math professor friend calls it (and my favorite expression), an eternal loop. You can usually break out of eternal loops by using Ctrl-C or clicking on a Stop button in Jupyter (the black square in the icon menu) or Cancel Build in Sublime Text or something similar.

```
In [ ]: # here is a countdown
        count = 10
```

```
# while loop with condition count > 0
while count > 0:
    print(count)

    # this is the decrement function
    count -= 1

# this line prints after the loop is over, i.e. count = 0
print("\nBlast Off!")
```

1.0.1 Counting forwards and backwards

I probably should not have used the variable count, but it seemed descriptive. Any variable will work!

The count -= 1 statement subtracts 1 from the variable count each time the statement is executed. Counters are very useful in programming. You can count forward by 1 by using count += 1. You can count by any value, even non-integers if you wish. This method is not limited to addition and subtraction. You can also use * and / if you wish by using count *= (for example) or count /= 2.

You can also count by using:

```
count = count + 1
```

This seems like an odd statement! How can count be equal to itself plus 1? However, the = sign is the assignment symbol (the == is the logical equals). You read this statement from right to left. The value of the right side is assigned to the left side variable. Which count version is best? I think the += symbol is a bit more efficient, but it's your choice.

1.0.2 Try It Yourself

Let's have some fun. Copy the blastoff program into the code cell below. Then...

- Change the count -= 1 to count -= 0.1. Does the program still work? What do you notice?
- Change the count -= 1 to count *= 2. Be ready to click the square stop button! Look at the output (scroll down if needed)
- Change the count -= 1 to count /= 2. Let the program run. What is the smallest value you see? After several decimal values, the remaining numbers are in scientific notation. For example '1.23889e-26' is the same as

$$1.23889 \times 10^{-26}$$

and 1.5678e+28 is the same as

$$1.5678 \times 10^{28}$$

1.1 Input data from the keyboard and trapping input errors

There are times when you need some interaction with the user or customer. This is easy to accomplish by using the the input() statement. Here is an example:

```
In [ ]: # have the user type in an integer between 1 and 10 inclusive
        value = input("Please type in an integer between 1 and 10 inclusive ")

        # print the value
        print("You typed in " + str(value))
```

That's nice, but what prevents a user from typing in a value outside the specified range... or from entering a floating point value? It's important to provide data entry error checking in all programs. Error checking helps prevent conditions such as dividing by zero (always a bad thing!) and other garbage input that can lead to garbage output. Let's give this concept a try... maybe a while loop will help?

Error trapping is an intermediate to advanced topic, but let's take a look. There are a couple of types of error trapping going on in the next program. First, the new keywords try: and except:

```
try:
    # try out the python statement to see if it generates
    # an error (or throws and exception... same thing)
except:
    # if the statement in the try: block causes an error or throws an exception
    # instead of halting the program and showing the error, go to this block
    # of code instead
```

The try: and except: blocks are very useful for catching errors, particularly errors that may occur at random or in unexpected instances (such as accidently dividing by zero).

The second error trap is the

```
if value >= 1 and value <= 10:
```

statement in the try: block. If the first line in the try: block is OK, that means the value is an integer. We then can test to see if the integer is in the specified range.

Try running the program several times and test the error trapping. The error trapping keeps the while loop working until the user types in the correct input.

Note: If the program seems to hang or does not accept input, check to see if there is an asterisk in the bracket next to the In preceding the code block... something like In[*]. If so, you need to click the black stop button or select the Kernel menu and Interrupt or any of the Restart entries (most likely Restart & Clear Output)... whatever it takes to clear the asterisk and return control back to you so you can run the program again.

```
In [ ]: # here is the condition
        key_in = True

        while key_in:

            # trap input errors using try and except
            try:
                # this statement checks to make certain the entered data is an integer
                value = int(input("Please type in an integer between 1 and 10 inclusive "))
```

```
            # at this point, we know we have an integer... but is it in the range?
            if value >= 1 and value <= 10:
                key_in = False
            else:
                print("\nNo, you are out of the input range!  Try again...")
        except:
            # if the try: statement is false, i.e. not an integer, then the except:
            # block of code executes next.
            print("\nNo, you typed in a string or decimal value!  Try again...\n")

    # print the value
    print("\nYou typed in " + str(value))
```

Python normally assumes that a variable entered via input() is a string. We can convert an intended number to a value by the method used above. If we need a floating point value, we would replace the

```
value = int(input("Please type in an integer between 1 and 10 inclusive "))
```

with

```
value = float(input("Please type in an integer between 1 and 10 inclusive "))
```

We've used error trapping for keyboard input, but as previously stated, error trapping is useful in any situation where errors, whether input or programmed, might possibly occur.

1.1.1 Using a while loop to slice strings

String variables can be considered as a list of characters. As such, we can slice strings just as we did with lists.

```
In [ ]: # enter a string
        str_var = input("Type in the name of your favorite city - ")

        # print the name one character at a time, but first, get the string length
        length = len(str_var)

        # start counting with the 0 or first character
        count = 0

        # print the individual letters in a column
        while count < length:
            print(str_var[count])
            count += 1

        # space
        print('')
```

```
# print the individual letters in a column in reverse

# this looks a bit odd, but remember we start counting from 0
# so the last character is numbered one less than the length
# i.e. if the string is 6 characters long, the last position = 5

count = length - 1

# we've got to to all the way to 0
while count >= 0:
    print(str_var[count])
    count -= 1

# print the last two characters in the string
print("\nThe last two characters are " + str_var[-2:])

print('')

# now print the name on a single line one character at a time


# start counting with the 0 or first character
count = 0

# print the individual letters in a line
while count < length:
    print(str_var[count], end = '')
    count += 1

# we've stripped the newline from the end of the print statement with end = ''
# if we don't add an empty print statement or add a newline we'll keep
# printing on the same line.

print('')

# and now reverse the letters in a line
count = length - 1

# we've got to to all the way to 0
while count >= 0:
    print(str_var[count], end = '')
    count -= 1
```

1.1.2 Try It Yourself

Now for a little challenge...

- Write a Python program that accepts string input from the keyboard.

- Print the input, whether a single name or sentence, in reverse order one letter at a time.
- Have the program continue to accept input (looping) until the user types 'quit'.
- No error trapping needed. Accept any input. You will see that numbers work as well as letters.

1.1.3 More functions

Let's write a function to determine whether or not an integer is prime. This script looks like a longer program and it is, but I've used a lot of remark statements to help me remember how the program was written and what I was thinking at the time.

```
In [ ]: # isprime function

        def isprime(n):

            # prime acts as a flag to let us know if we have a prime or not
            # we start out with prime = True

            prime = True

            # we need to divide by all integers up to a certain point
            # to see if any are divisors of our test value.  We know
            # all numbers > 2 can't be prime, so we start with 3 and
            # then use odd numbers

            divisor = 3

            # we will check divisors up to one less than half the test value
            # actually, we only need to check up to the square root of the
            # test value... n**(1/2) Why?

            while divisor < (n/2):

                # the % is the modular or remainder division symbol
                # it returns only the remainder in a division problem
                # if the remainder == 0, then we have a divisor and the
                # test value is NOT prime

                if n%divisor == 0:

                    # if we have a divisor, flag it by changing noprime to True
                    prime = False

                    # get out of the while loop... we are done
                    break

                else:
                    # add two to the divisor and try again.  Since we started with
```

```
            # 3, this statement runs through the odd numbers
            divisor += 2

        # now the moment of truth. If we went through all of the potential divisors
        # and did not change the value of prime from True to False, we have a prime number

        if prime == True:
            print(str(n) + " is prime.")

        # and if we DID find a divisor, prime was changed to False
        # and we'll print accordingly

        else:
            print(str(n) + " is composite.")


    # test value

    isprime(45587)
```

```
In [ ]: # try your own test value here
```

Note: We can use the % (mod or remainder division) to determine whether or not a number is even or odd. Here is an example code block for that purpose:

```
# assume the variable n is some integer
if n%2 == 0:
    print('even')
else:
    print('odd')
```

Now let's write a square root function. I know Python can do this on its own, but perhaps we can put one together using what we've already learned. The following algorithm (computer code that performs a series of calculations) is based on Newton's method. It is several hundred years old (Newton's method, not the code), but is still useful.

```
In [ ]: # define a square root function using Newton's method
        # you can look this up online for a better explanation

        def sqr(n):

            # start with a guess of 0.5*n or 1/2 of the number
            guess = 0.5 * n

            # make a better guess by taking the average of
            # the guess and the number divided by the guess
            better = 0.5 * (guess + n/guess)

            # as long as the guess and the better value are
```

```
        # different (not equal or !=), keep on working
        while better != guess:

            # replace guess with the better value and try again
            guess = better

            # make a better guess by taking the average of
            # guess and the number divided by guess
            better = 0.5 * (guess + n/guess)

        return guess

    # test
    sqr(2)
```

```
In [ ]: # try your own test values here
```

1.1.4 Go to Assignment 3