

Loops in Python

ENGR 298: Engineering Analysis and Decision Making with Python



Gains over time

- Our previous compound interest equation provides the result of some deposit over time, but how could we see the final balance at the end of each year?

$$A = P(1 + (r/100))^n$$

```
P =100; r = 5.0;  
n=0;  A = P * (1+r/100)**n;  print(n,A)  
n=1;  A = P * (1+r/100)**n;  print(n,A)  
...  
n=9;  A = P * (1+r/100)**n;  print(n,A)  
n=10; A = P * (1+r/100)**n;  print(n,A)
```

- For 10 years, we could write 10 lines of code to automatically update/refresh the value of final deposit (A).

How would this approach extend to 100 years? Or for an indefinite amount of time?

Repeating Yourself Through Loops

- Frequently a piece of code must be repeated to accumulate or finalize some result. A **decision** is being made by the computer to execute the code a certain number of times.
- A **loop** can be used to execute a block of code until some **condition** is met.
- Python supports **for** and **while** looping. Each can achieve the same function but have different styles and may be easier to write.

A program is a set of instructions for **manipulating**, **moving**, and **making decisions** on **data** in your computer.

While Loop – Part 1

- A while loop tests a single **condition** and then begins the code within its loop. At the of the loop, if the condition is **TRUE**, the loop repeats. Otherwise, if the statement is **FALSE**, the loop exits and proceeds with the program.
- The condition evaluated by the loop must be a **Boolean expression** that is TRUE or FALSE.
 - (3 < 4) is TRUE
 - (3 == 3) is TRUE
 - (4 <= 3) is FALSE
 - (3 = 3) is not a Boolean expression but will evaluate as TRUE

While Loop – Part 2

- The variables in a loop condition are typically a **counter** (that is incremented to some value) or a **flag** (that set true once an operation completes)

```
# counter variable to keep track
counter = 0

# while loop to see when counter is more than 0
while counter < 10:
    # do something now we're in the loop

    # Important! Update the counter variable
    counter = counter + 1
```

While loop with counter

```
# create a flag. We will loop until this is True
done = False

# Dangerous infinite loop but it may work
while not done:

    # Call a function, it will say when we're done
    done = myFunction()
```

While loop with flag

While Loop – Part 2

```
# counter variable to keep track
counter = 0

# while loop to see when counter is more than 0
while counter < 10:
    # do something now we're in the loop

    # Important! Update the counter variable
    counter = counter + 1
```

The major error in this format is failing to update the counter variable AND/OR having a comparison that will never work

```
# create a flag. We will loop until this is True
done = False
```

```
# Dangerous infinite loop but it may work
while not done:
```

```
# Call a function, it will say when we're done
done = myFunction()
```

It is very easy for this to become an 'infinite loop'. Must have strong assurance that at some point myFunction will return True. Often used when waiting for some external input (ctrl-c) to end a program.

First Attempt: Why did this approach run forever?

```
# an initial deposit
balance = 1000

# interest rate (as a fraction)
rate = 0.023

# year to run calculation
n = 10

# a counter variable to keep track of the loop
counter = 0

while counter < n:
    balance = balance * (1+rate)
    print("At the end of Year " + str(counter)
          + " the balance is " + str(balance))
```

This is better now...

```
# an initial deposit
balance = 1000

# interest rate (as a fraction)
rate = 0.023

# year to run calculation
n = 10

# a counter variable to keep track of the loop
counter = 0

while counter < n:
    balance = balance * (1 + rate)
    counter = counter + 1

    print("At the end of Year " + str(counter)
          + " the balance is " + str(balance))
```

```
At the end of Year 1 the balance is 1022.9999999999999
At the end of Year 2 the balance is 1046.5289999999998
At the end of Year 3 the balance is 1070.5991669999996
At the end of Year 4 the balance is 1095.2229478409995
At the end of Year 5 the balance is 1120.4130756413424
At the end of Year 6 the balance is 1146.1825763810932
At the end of Year 7 the balance is 1172.5447756378583
At the end of Year 8 the balance is 1199.513305477529
At the end of Year 9 the balance is 1227.102111503512
At the end of Year 10 the balance is 1255.3254600680928
```


How do we know if that program is correct?

Let's compare output with the previous one.

```
Initial principal is $1000
Interest rate is 0.023
Hold for 10 years
Final value after is $1255.33
week2-formulas.py
```

```
At the end of Year 1 the balance is 1022.9999999999999
At the end of Year 2 the balance is 1046.5289999999998
At the end of Year 3 the balance is 1070.5991669999996
At the end of Year 4 the balance is 1095.2229478409995
At the end of Year 5 the balance is 1120.4130756413424
At the end of Year 6 the balance is 1146.1825763810932
At the end of Year 7 the balance is 1172.5447756378583
At the end of Year 8 the balance is 1199.513305477529
At the end of Year 9 the balance is 1227.102111503512
At the end of Year 10 the balance is 1255.3254600680928
```

week2-while-loop.py

The results appear to be the same. However, note that $1255.33 \neq 1255.3254\dots$. The internal value on one has been truncated when the result is output.

Never perform `==` comparison on decimal values. This will compare every decimal point. There may be rounding errors between CPU runs.

A little practice on Boolean expressions...

- What values for t make the expressions TRUE and FALSE?

$t == 40$

$t != 40$

$t \geq 40$

$t > 40$

$t < 40$

A little practice on Boolean expressions...

- What values for t make the expressions TRUE and FALSE?

$t == 40$

True: Must exactly equal 40 (integer). **False:** any other value.

$t != 40$

True: Any value that is not 40 (integer). **False:** Must exactly equal 40 (integer).

$t >= 40$

True: Any value ≥ 40 . **False:** Any value less than 40

$t > 40$

True: Any value > 40 . **False:** Any value less than or equal to 40

$t < 40$

True: Any value < 40 . **False:** Any value greater than or equal to 40

```
# some random variable to compare
```

```
t = 30
```

```
print("Using t = "+str(t))
```

```
# perform boolean equality comparison
```

```
print(str(t)+" == 40 is " + str(t == 40))
```

```
# perform boolean not equality
```

```
print(str(t)+" != 40 is " + str(t != 40))
```

```
# perform greater than or equal
```

```
print(str(t)+" >= 40 is " + str(t >= 40))
```

```
# perform greater than
```

```
print(str(t)+" > 40 is " + str(t > 40))
```

```
# perform less than
```

```
print(str(t)+" < 40 is " + str(t < 40))
```

Using t = 30

30 == 40 is False

30 != 40 is True

30 >= 40 is False

30 > 40 is False

30 < 40 is True

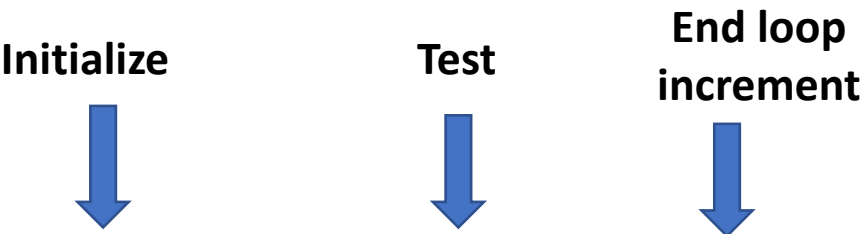
Final comment on Boolean expressions...

- Each expression ($A \neq B$) can be chained together to form a longer expression using **AND**, **OR**, and **NOT** operators.
- For **AND**, both A and B must be TRUE, otherwise the result is FALSE.
- For **OR**, either A or B can be TRUE, otherwise the result is FALSE.
- **NOT**, returns that opposite of a result. **NOT True == False**.

For Loops in Python

- Another common loop is a **For** loop. In traditional languages (C, C++, Java) the For loop has complicated syntax that performs an *initialization, conditional test, and end loop statement*.

Initialize Test End loop
 increment



```
for (int i = 0; i < LIMIT; i++)  
{  
    sum = result[i] = (10 * data[i]);  
}
```

- In Python, for loops are greatly simplified as they only *iterate* over a list or range.

Basic FOR loop examples

Like while loop, the for loop executes the code in its body until the list or collection is exhausted. It “iterates” over each element and then performs the body code.

Can read each example as asking “For <some variable> in <some collection>, do the following things...”

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Iterate over list of words and print out the word with its length

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Iterate over a range of 5 values,
print out each value

Calculating Return for Various Deposits

- Consider you have a list of all deposits held in a CD. It is the end of year and the interest must be *compounded*.

```
# create a random list of deposits
accounts = [1000, 1235, 1982, 29462, 189263, 102372, 27]

# set an interest rate to be applied (as fraction)
rate = 0.05

# iterate through deposits and determine new value
for deposit in accounts:
    balance = deposit * (1+rate)
    print("Balance was "+str(deposit)+", now is "+str(balance))
```


Basic For with Break and Continue

- There may be times where you wish to end a loop 'early'. Possibly you have found the item of interest or some other process is indicating to stop.
- "Break" can be used to immediately exit the loop. Will start back right below for().
- "Continue" will automatically restart and iterate to the next element. Useful when you are scanning through a list, but only want to perform operations on 'certain' elements.

```

# generate a long random list of integers
randoms = generate_random_int_list(1000, 100000000)

# iterate through the list check for all prime numbers
# if you see the magic number '100', gremlins are present and
# exit the loop. Keep track of how many we found
primes_found = 0
for p in randomness:

    # anything less than 2 is not prime
    if p < 2:
        continue

    # check to specific, dangerous numbers
    if p == 100:
        print("Danger!!")
        break

    # check to see if p is prime or not
    primality_test = primality.isprime(p)

    if primality_test == True:
        print("The value "+str(p)+ " is prime!")
        primes_found = primes_found + 1

# loop is now done. Print out the results....
print("The list contained " + str(len(randoms)) + " elements. We found " + str(primes_found) + " primes!")

```

Create your own Python scripts to answer the following questions. Submit to Canvas by end of week.

- **Odds or Evens:** download the file template from GitHub ([link](#)). Iterate through the list and determine how many values were ODD and how many were EVEN. Print out the count for each type at the end of the program and what percentage it was of the total.
 - **Food for thought:** when sampling a random distribution there should be equal numbers of odd and even values. Increase the length of the list and maximum value. As these are increased the number of ODDS and EVENS should balance.
- **Spread:** download the file template from GitHub ([link](#)). The two lists are samples from a random distribution. Print out whether List A or B has the larger standard deviation.
 - **Food for thought:** consider how a distribution comparison could be used to determine if a machine or process is out of specification? A simple t-test can be used to see if processes/results are “different” from one another. How might an engineer utilize this information?
- **I Like Pi:** Use the [Gauss-Legendre Algorithm](#) to estimate Pi. Perform 10 approximation loops. At the end of each loop, print out your current PI estimate and its error from [PI](#) as encoded in Python.
 - **Food for thought:** while this is a silly example it will walk you through numerical approximation methods. This is the basis for any simulation or FEA tool. We will utilize Euler’s Method later in the course to solve first order differential equations.