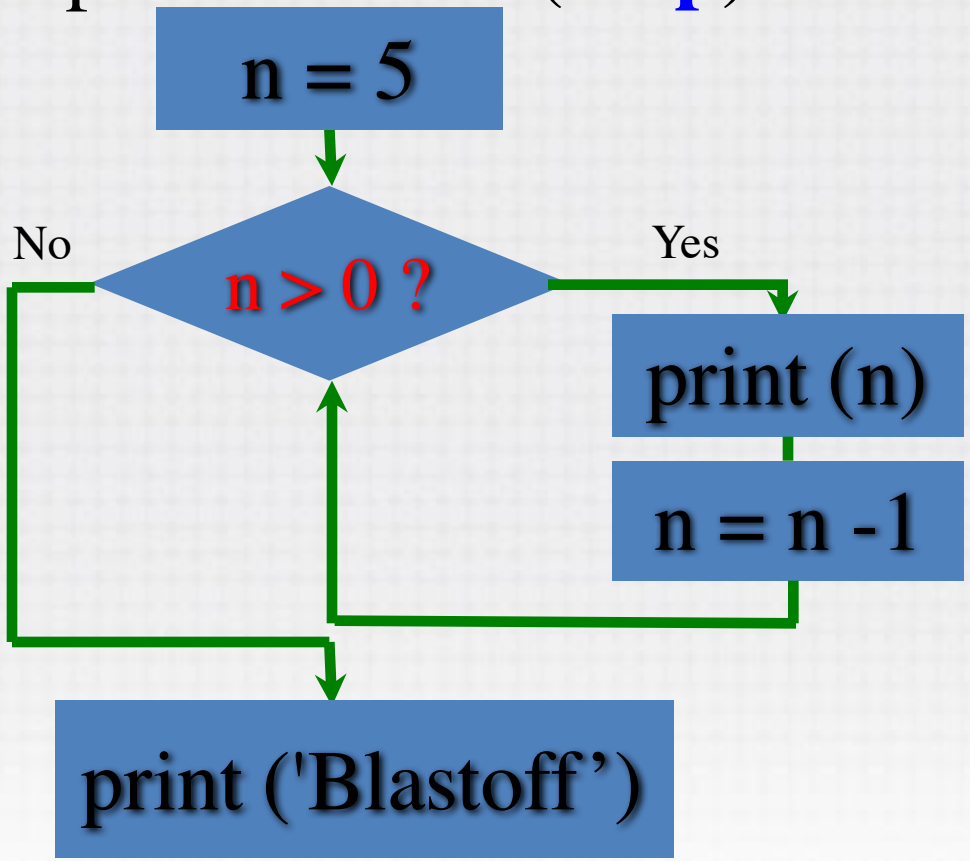
Essential Data Skills for Business Analytics

Lecture 4: Loop Structure

Decision, Operations & Information Technologies
Robert H. Smith School of Business
Spring, 2020

Repeated steps

- Computers are often used to automate repetitive tasks (**loop**)



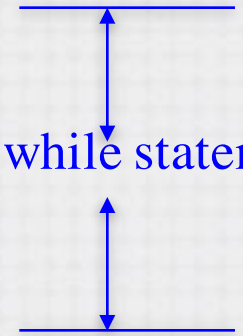
Output:

5
4
3
2
1
Blastoff!

The *while* statement

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

- Syntax:




```
while expression:
    statement 1
    statement 2
    …
    statement N
next statement
```

- The flow of execution
  a) Evaluate the expression, yielding True or False
  b) If the expression is False, exit the entire while statement and continue execution at the next statement
  c) If the expression is True, execute each of the statements in the body and then go back to step (a)

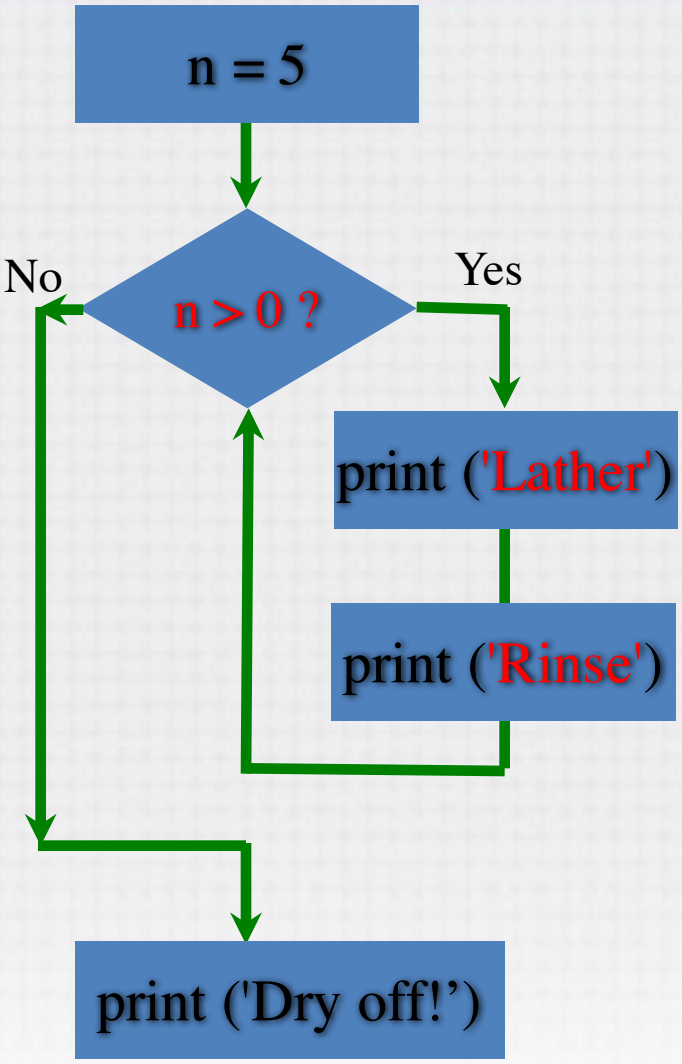
Example


```
test1.py

x = 5
while x > 3:
   print (x)
   x = x - 1
print (x+1)

python test1.py
5
4
4
```

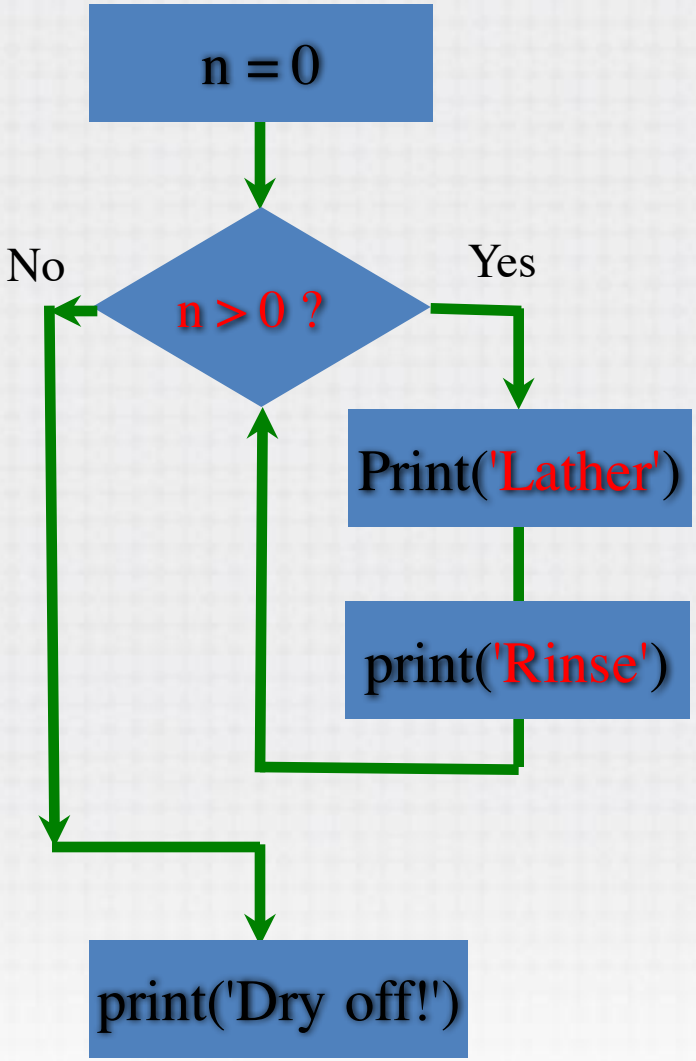
An infinite loop




```
n = 5
while n > 0 :
    print ('Lather' )
    print ('Rinse')
print ('Dry off!')
```

What is wrong with this loop?

Another loop




```
n = 0
while n > 0 :
    print('Lather')
    print('Rinse')
print('Dry off!')
```

What does this loop do?

Example


```
test2.py

x = 5
while x != 1:
    print (x)
    if x%2 == 0:
        x = x / 2
    else:
        x = x*3 + 1

python test2.py
5
16
8
4
2
```

The nested *while* statement

- Syntax:

```
while expression:
    statement 1
    statement 2
    …
    while expression:
        statement 1
        statement 2
        …
        statement N
    statement N
next statement
```

Outer while statement

inner while statement

Example


```
test3.py

x = 5
while x != 1:
   print (x)
   while x > 3:
      print ('x>3')
      x = x - 1
   x = x - 1

python test3.py
```

```
Output:
5
x>3
x>3
3
2
```

Indefinite loop

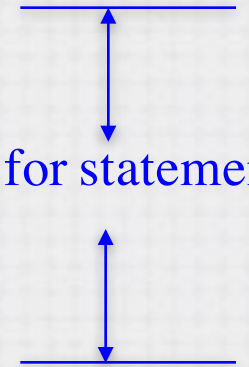
- While loops are called "indefinite loops" because they keep going until a logical expression becomes False
- The loops we have seen so far are easy to examine to see if they will terminate or if they are "infinite loops"
- Sometimes it is harder to be sure if a loop will terminate

Definite loop

- Quite often we have a list of items – effectively a finite set of things
- We can write a loop to run the loop once for each of the items in a set using the Python for construct
- These loops are called "definite loops" because they execute an exact number of times
- We say that "definite loops iterate through the members of a set"

The *for* statement

- Syntax:

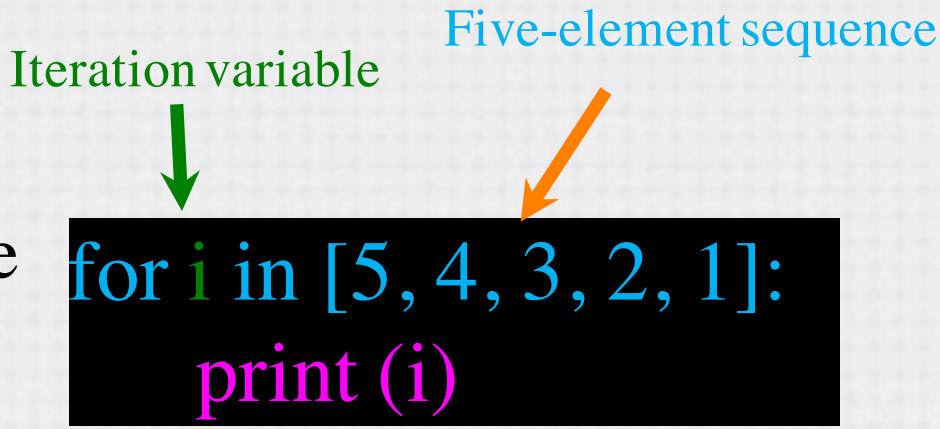


for iterator *in* expression_list**:**
statement 1
statement 2
…
statement N

- The flow of execution
  - The expression list is evaluated once; it should yield an iterable object (e.g., list, tuple, etc.)
  - For each member in the expression_list, execute all statements in the for body.

The *for* statement

- The iteration variable "iterates" though the sequence (ordered set)
- The block (body) of code is executed once for each element in the sequence
- The iteration variable moves through all of the values in the sequence



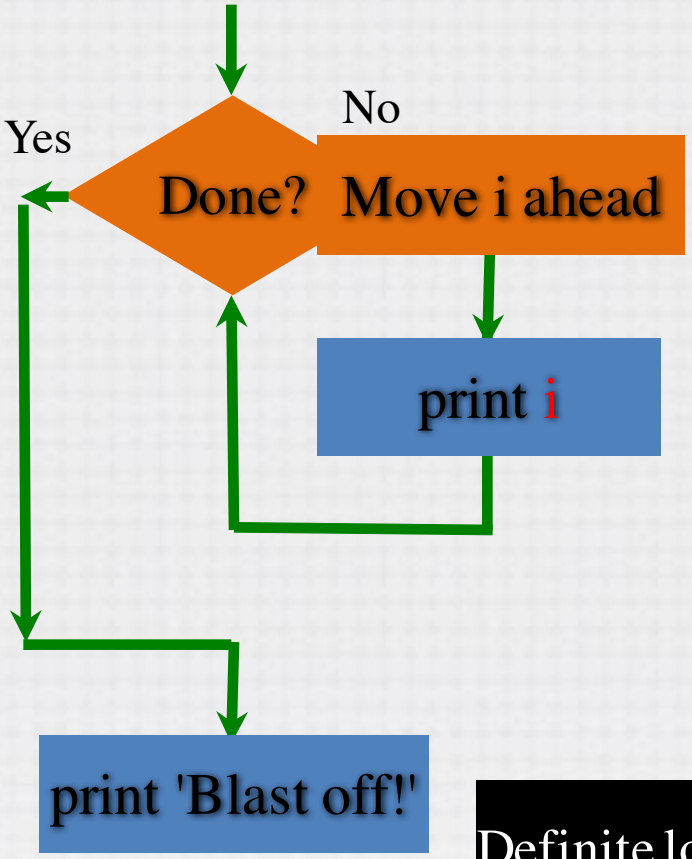
Example (1)

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
for i in [5, 4, 3, 2, 1]:
    print (i)
print ('Blastoff!')
```

```
Output
5
4
3
2
1
Blastoff!
```

The *for* statement




```
for i in [5, 4, 3, 2, 1]:
    print (i)
print ('Blastoff!')
```

```
5
4
3
2
1
Blastoff!
```

Definite loops (for loops) have explicit iteration variables that change each time through a loop. These iteration variables move through the sequence or set.

Example (2)

```
for i in [5, 4, 3, 2, 1]:
    if i % 2 == 0:
        print (i, ": even")
    else:
        print (i, ": odd")
print ('Blastoff!')
```

```
Output
5: odd
4: even
3: odd
2: even
1: odd
Blastoff!
```

Nested *for* statement

- Syntax:

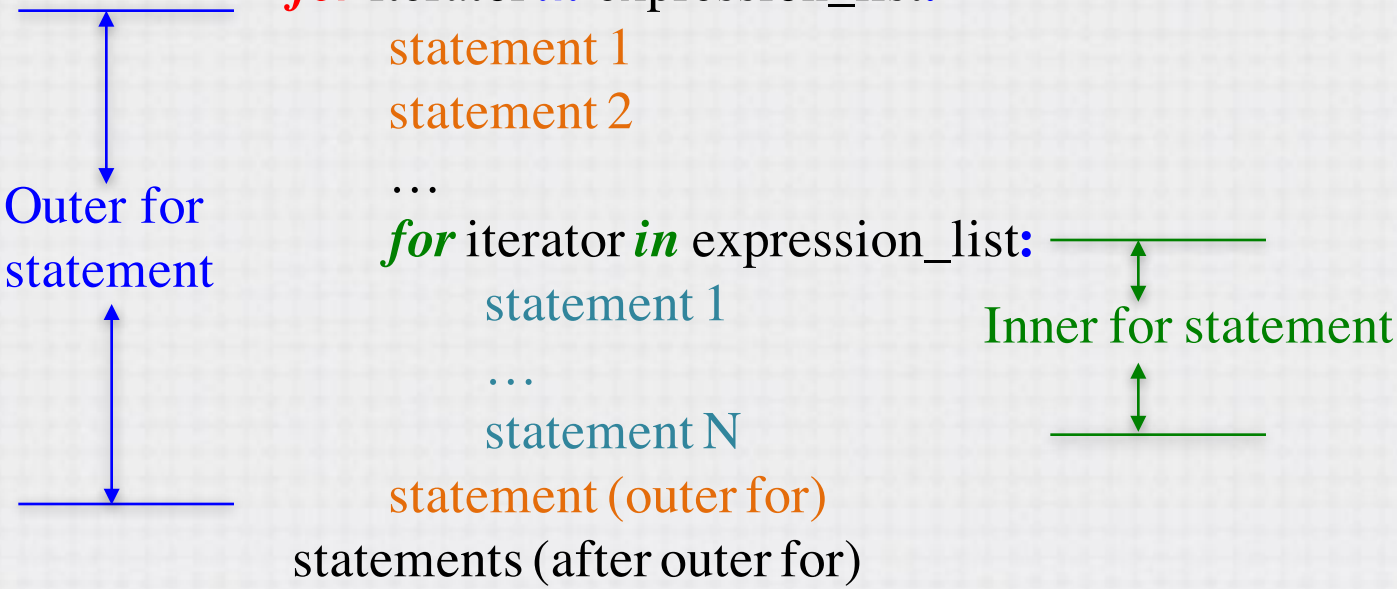
```
for iterator in expression_list:
    statement 1
    statement 2
    …
    for iterator in expression_list:
        statement 1
        …
        statement N
    statement (outer for)
statements (after outer for)
```

Outer for statement

Inner for statement

Nested *for* statement

- Syntax:




```
for iterator in expression_list:
        statement 1
        statement 2
        …
        for iterator in expression_list:
                statement 1
                …
                statement N
        statement (outer for)
statements (after outer for)
```

- The flow of execution
  - ❑ Consider the "inner for loop" as "one statement" within the outer loop body
  - ❑ For each member in the "outer loop", execute all statements
  - ❑ When execute inner for loop statement, consider it as a real loop

Example (1)

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
for i in [1, 2, 3] :
    for j in [1, 2, 3]:
        print (i*j)
print ('Blastoff!')
```

```
Output
1
2
3
2
4
6
3
6
9
Blastoff!
```

Example (2)

```
for i in [1, 2, 3]:
    j = 1
    while j<=i:
        print (i)
        j = j+1
print ('Blastoff!')
```

```
Output
1
2
2
3
3
3
Blastoff!
```

Making "smart" loops

- The trick is "knowing" something about the whole loop when you are stuck writing codes that only sees one entry at a time

Set some variables to initial values

for element in set:

1. Look for something or do something to each element separately.
2. Update a variable.

Look at the variables.

What is the largest number

3 41 12 9 74 15

largest_so_far -1 3 41 74

What is the largest number

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
largest_so_far = -1
for current in [3, 41, 12, 9, 74, 15]:
    if current > largest_so_far:
        largest_so_far = current
print (largest_so_far)
```

Counting in a loop


```
i = 0
print ('Before', i)
for thing in [9, 41, 12, 3, 74, 15] :
    i= i+ 1
    print (i, thing)
print ('After', i)
```

```
python countloop.py
Before 0
1 9
2 41
3 12
4 3
5 74
6 15
After 6
```

To count how many times we execute a loop we introduce a counter variable that starts at 0 and we add one to it each time through the loop.

Summing in a loop

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

```
sum = 0
print ('Before', sum)
for thing in [9, 41, 12, 3, 74, 15] :
    sum= sum+ thing
    print (sum, thing)
print ('After', sum)
```

```
python sumloop.py
Before 0
9 9
50 41
62 12
65 3
139 74
154 15
After 154
```

To add up a value we encounter in a loop, we introduce a sum variable that starts at 0 and we add the value to sum each time through the loop.

Finding the average in a loop


```
count = 0
sum = 0
print ('Before', count, sum)
for value in [9, 41, 12, 3, 74, 15] :
    count = count+1
    sum= sum+ value
    print (count, sum, value)
print ('After', count, sum, sum/count)
```

```
python avgloop.py
Before 0 0
1 9 9
2 50 41
3 62 12
4 65 3
5 139 74
6 154 15
After 6 154 25
```

An average just combines the counting and sum patterns and divides when the loop is done.

Search in a loop


```
found = False
print ('Before', found)
for value in [9, 41, 12, 3, 74, 15] :
    if value == 3:
        found = True
    print (found, value)
print ('After', found)
```

```
python searchloop.py
Before False
False 9
False 41
False 12
True 3
True 74
True 15
After True
```

If we just want to search and know if a value was found, we use a variable that start at False and is set to True as soon as we find the value.

Another example

```
i = 1
height = 5
while i <= height:
    j = 1
    line = ''
    while j <= i:
        line += str(i*j) + '\t'
        j = j+1
    print (line)
    i = i+1
```

```
python nestedloop.py

1
2    4
3    6    9
4    8    12   16
5    10   15   20   25
```

Breaking out of a loop

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

- The break statement ends the **current innermost** loop and jumps to the statement immediately following the loop.
- It can happen anywhere in the body of the loop, depending on your needs.

```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

Texts in green here are received from the keyboard

Breaking out of a loop

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

- The break statement ends the **current innermost** loop and jumps to the statement immediately following the loop.
- It can happen anywhere in the body of the loop, depending on your needs.

```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

Texts in green here are received from the keyboard

Breaking out of a loop

- All statements in the loop body and after break will NOT be executed if break happens.

```
x = 5
while x > 0:
    print (x)
    if x == 3:
        break
    x = x – 1
print x
```

```
Output:

5
4
3
3
```

Breaking out of a loop

- The break statement ends the current loop and jumps to the statement immediately following the loop.
- All statements in the loop body and after break will NOT be executed if break happens.

```
x = 5
while x > 2:
    print (x)
    while True:
        print('x > 3')
        if  x == 3 :
            break
        x = x - 1
    x = x – 1
print (x)
```

Innermost loop

```
Output:

5
x>3
x>3
x>3
2
```

The *continue* statement

- The continue statement ends the current iteration of the innermost loop and jumps to the top of the loop and starts the next iteration.
- It can happen anywhere in the body of the loop, depending on your needs.

```
while True:
    line = input('> ')
    if line == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> #
> print this!
print this!
> done
Done!
```

Texts in green here are received from the keyboard

The *continue* statement

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration.
- It can happen anywhere in the body of the loop, depending on your needs.

```
while True:
    line = input('> ')
    if line == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> #
> print this!
print this!
> done
Done!
```

Texts in green here are received from the keyboard

Example

```
x = 5
while x > 0:
    x = x – 1
    if x == 3:
        continue
    print (x)
print(x)
```

```
Output:

4
2
1
0
0
```

Example

```
x = 5
while x > 2:
        print (x)
        while x > 0:
                x = x - 1
                if  x < 3 :
                        continue
                        print ('x < 3')
                else:
                        print ('x >= 3')
        x = x – 1
print (x)
```

Innermost loop

Output:

```
5
x>=3
x>=3
-1
```