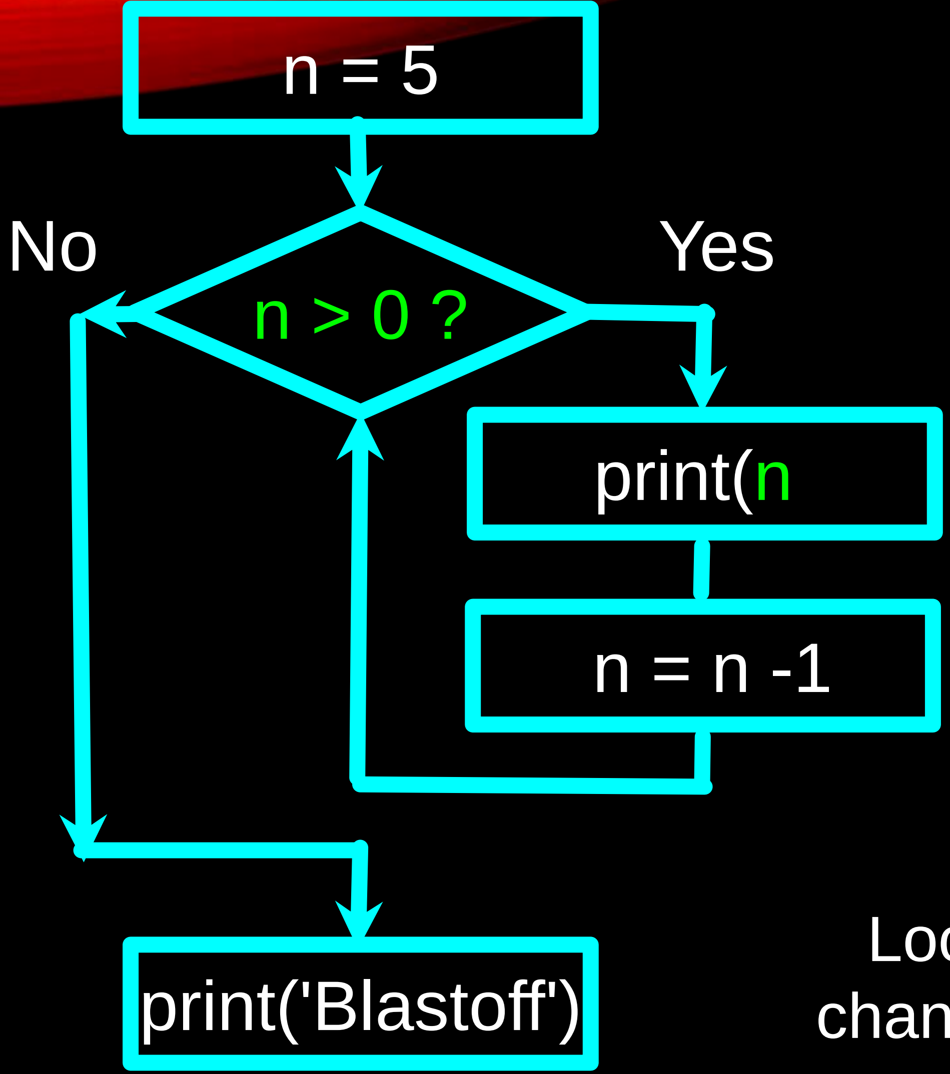
PYTHON – WHILE LOOP

Repeated Steps



Program:

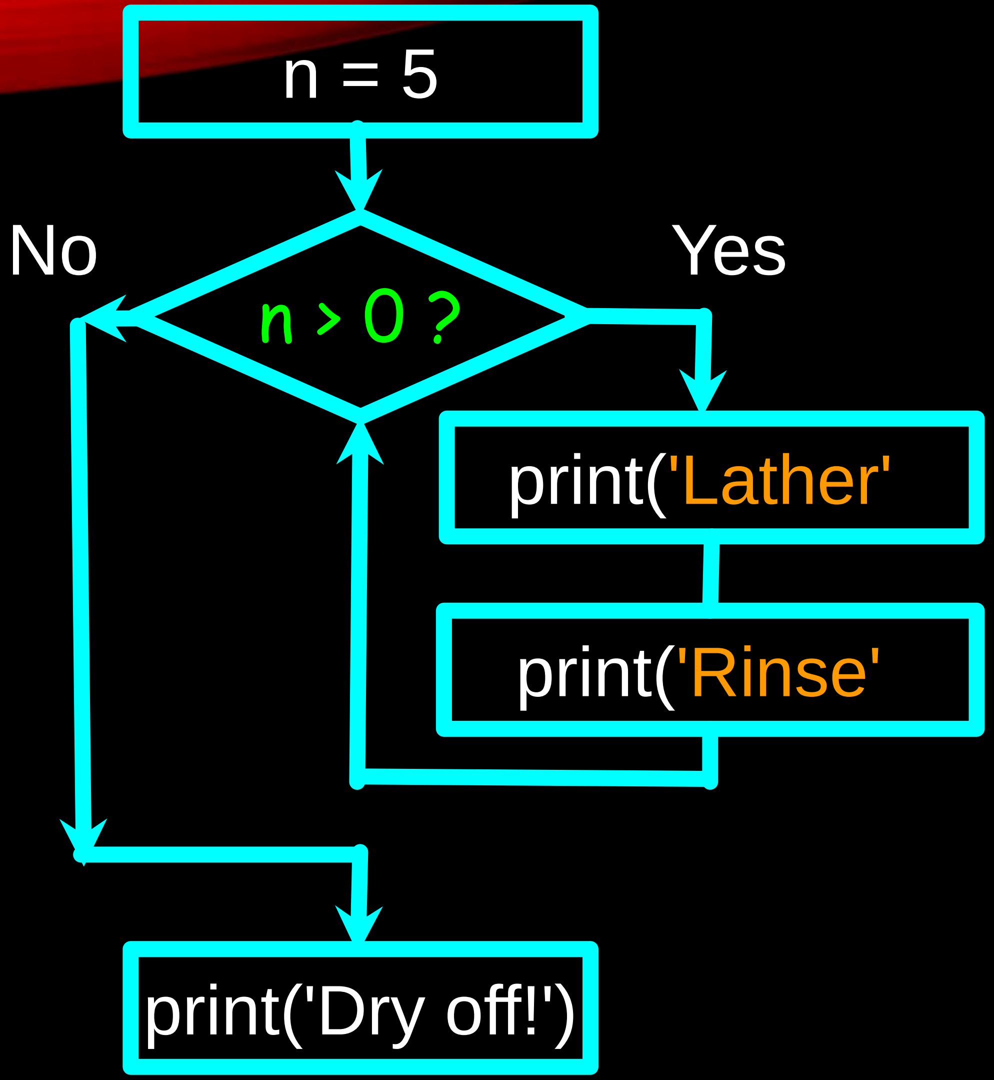
```
n = 5
while n > 0 :
    print n
    n = n – 1
print 'Blastoff!'
print n
```

Output:

```
5
4
3
2
1
Blastoff!
0
```

Loops (repeated steps) have iteration variables that change each time through a loop. Often these iteration variables go through a sequence of numbers.

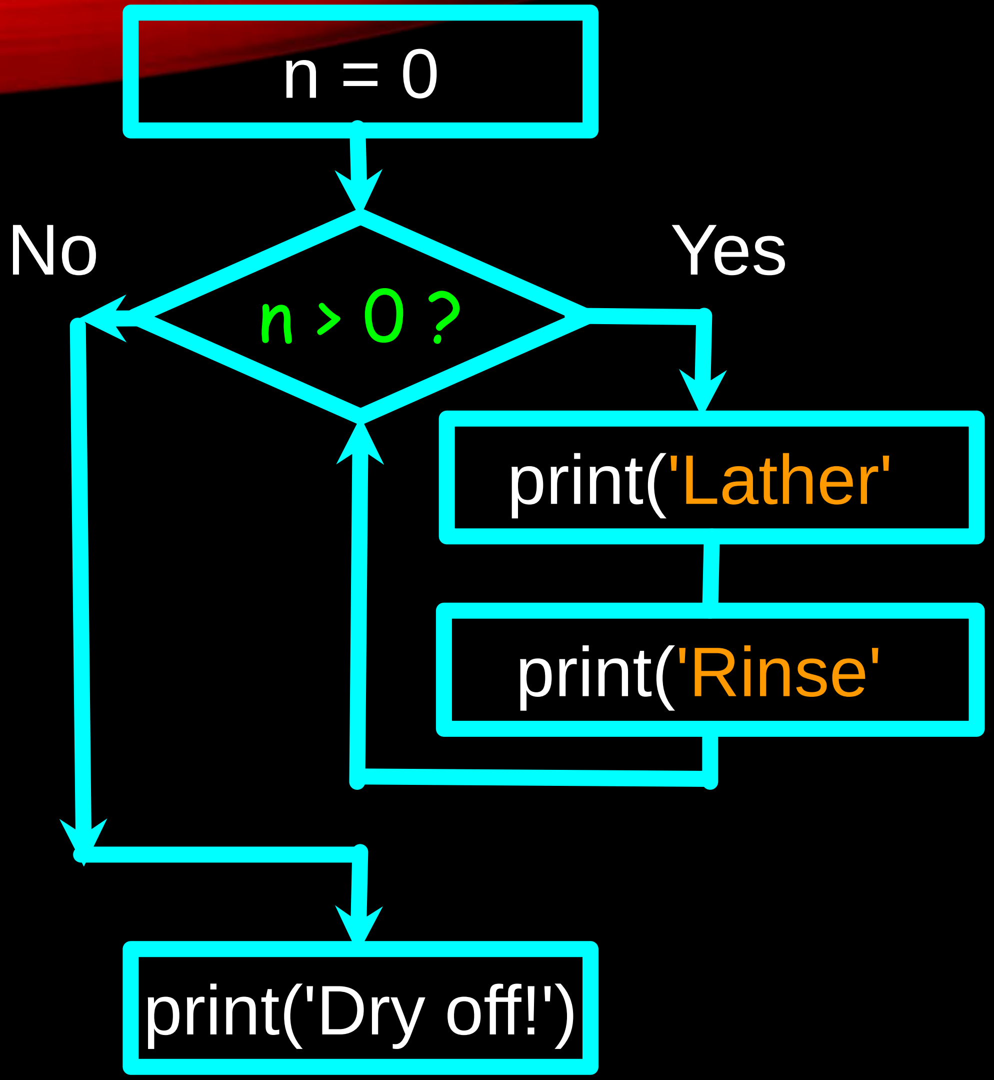
An Infinite Loop



```
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 is this loop doing?

Breaking Out of a Loop

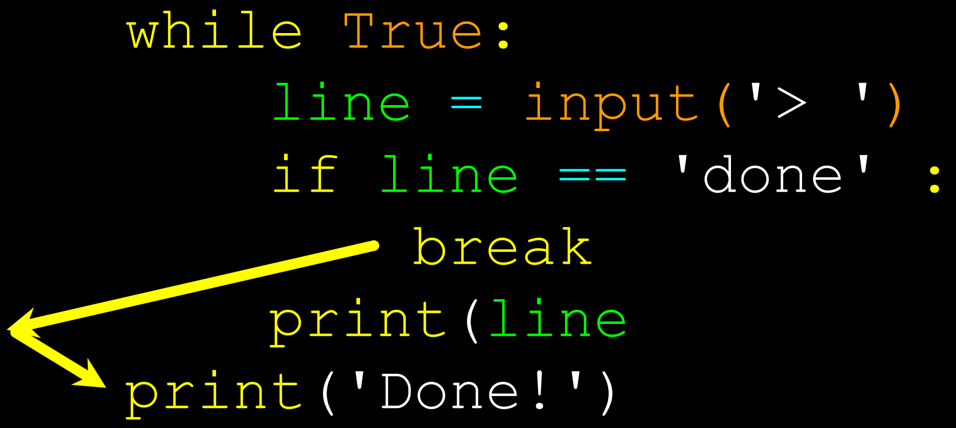
- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

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

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

Breaking Out of a Loop

- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

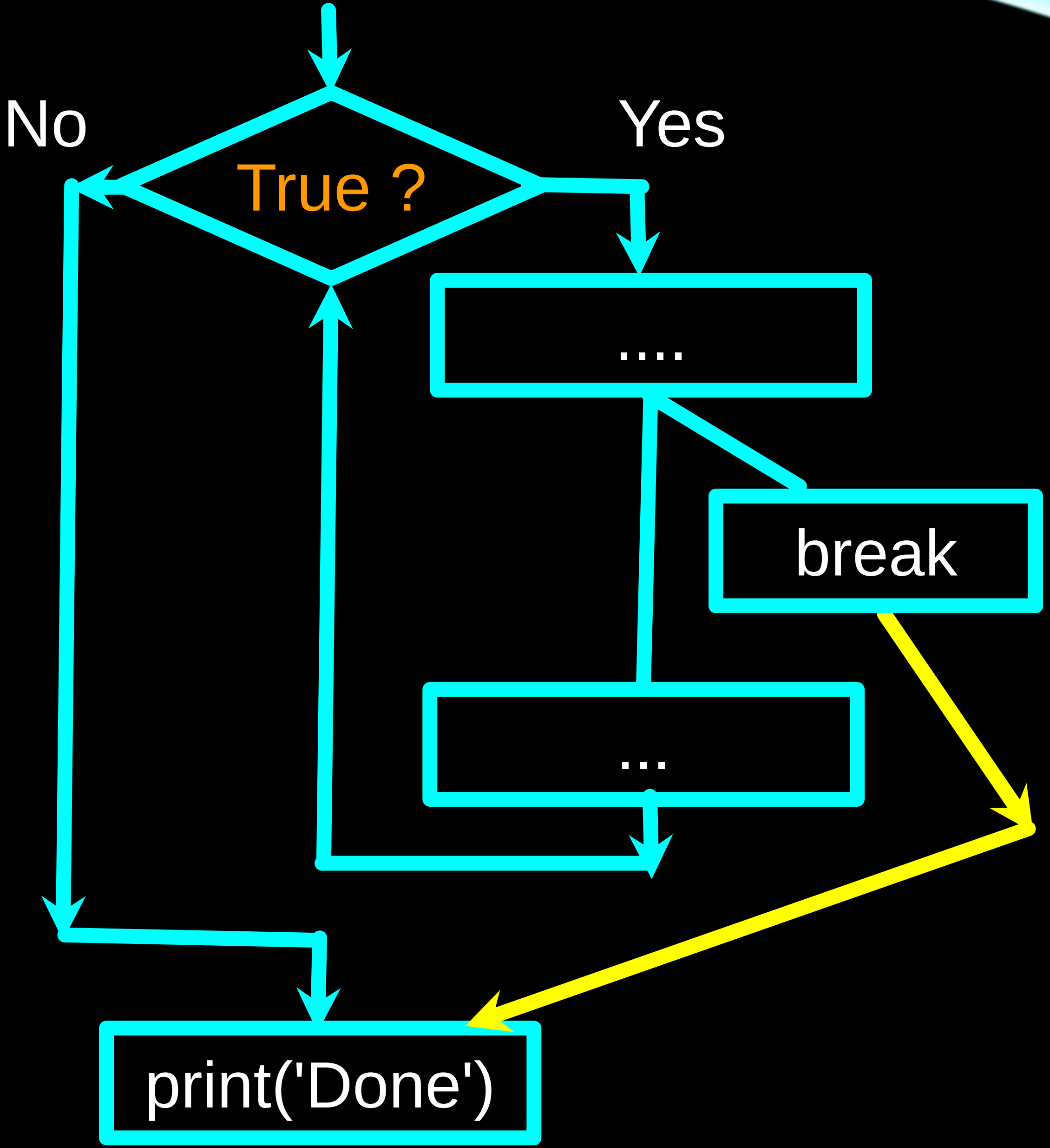


```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line
print('Done!')
```

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

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

http://en.wikipedia.org/wiki/Transporter_(Star_Trek)



Finishing an Iteration with continue

The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

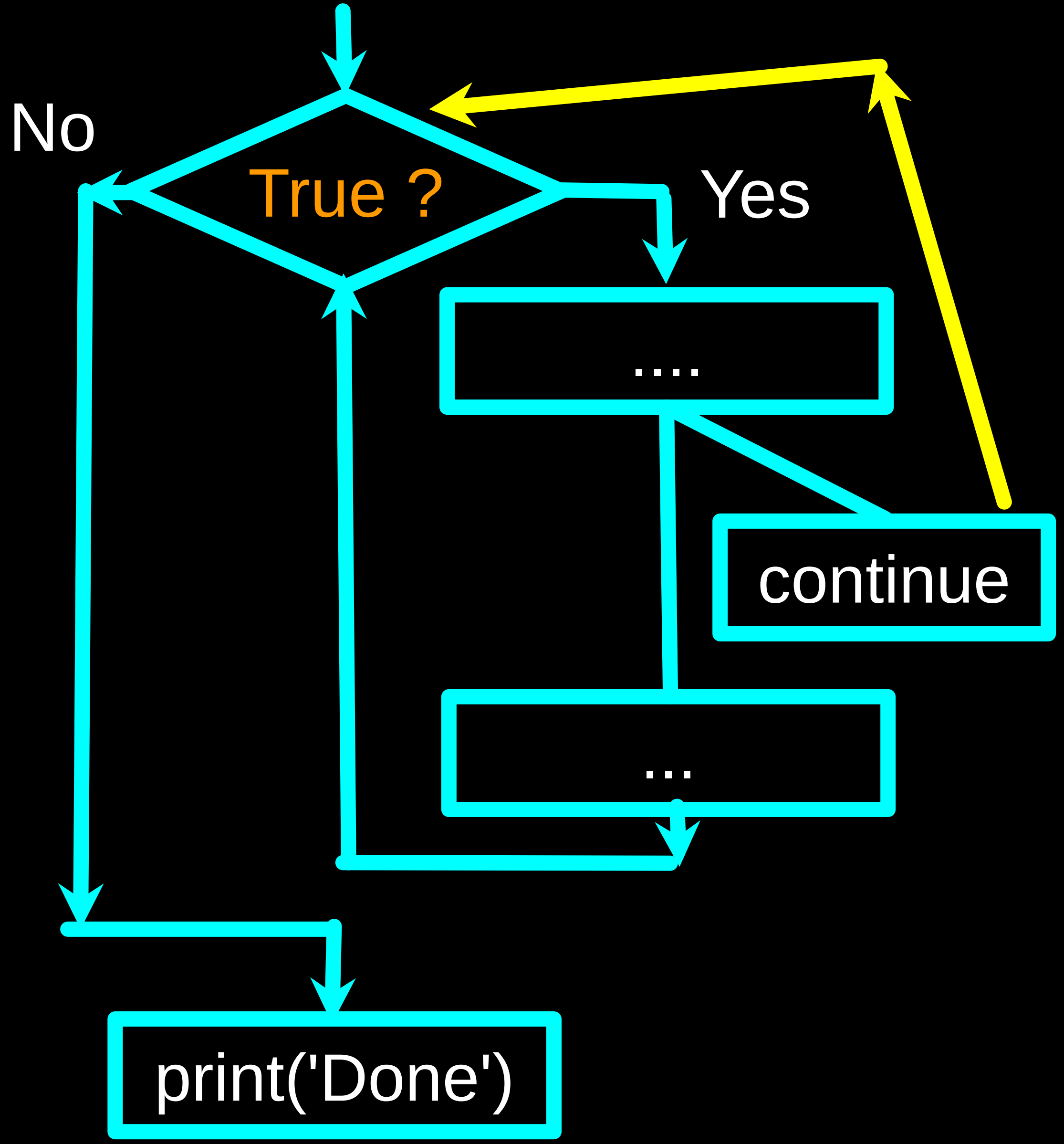
Finishing an Iteration with continue

The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

```
while True:
   line = raw_input('> ')
   if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line
print('Done!')
```



Indefinite Loops

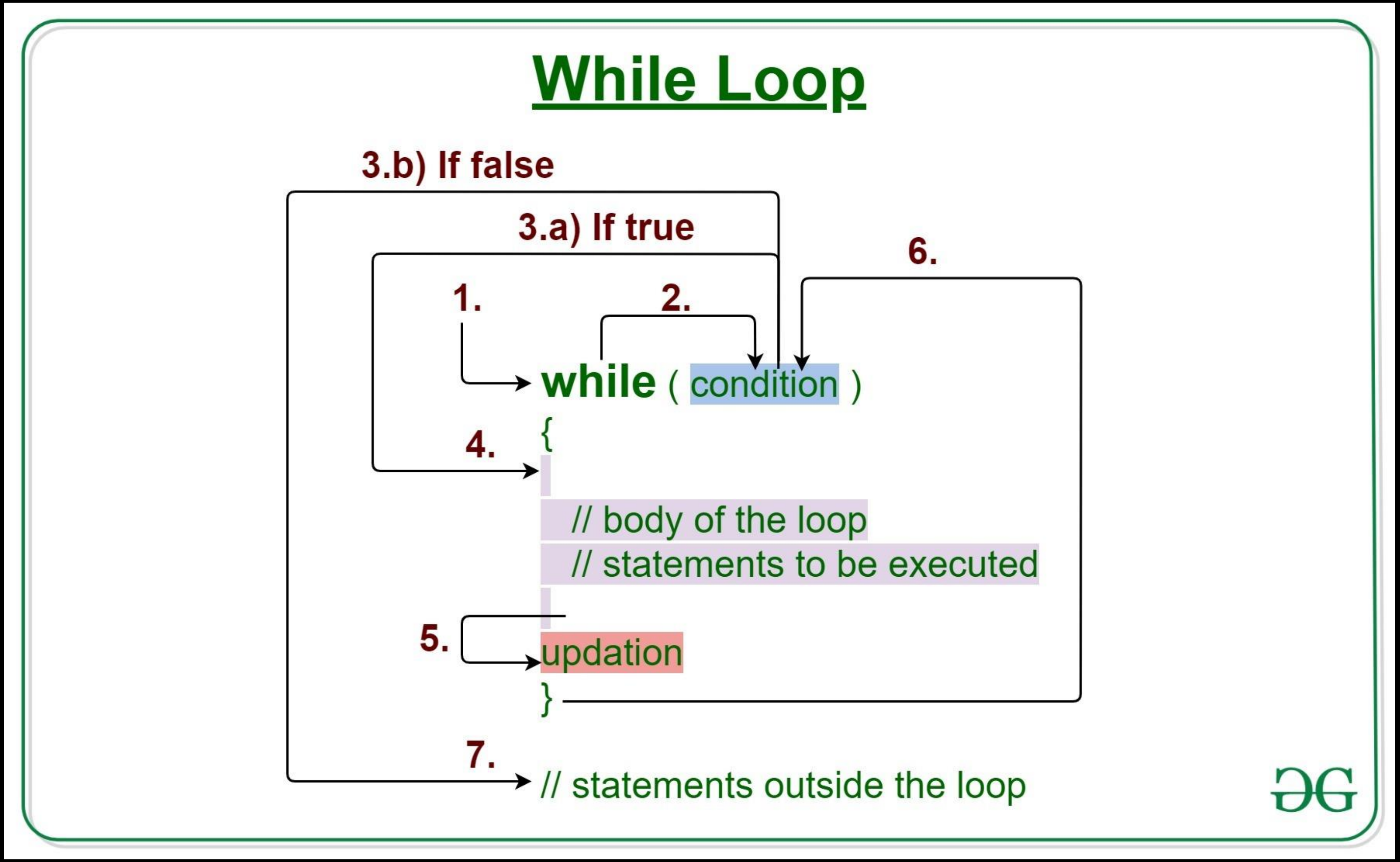
- While loops are called “indefinite loops” because they keep going until a logical condition becomes False
- The loops we have seen so far are pretty easy to examine to see if they will terminate or if they will be “infinite loops”
- Sometimes it is a little harder to be sure if a loop will terminate

WHILE LOOP

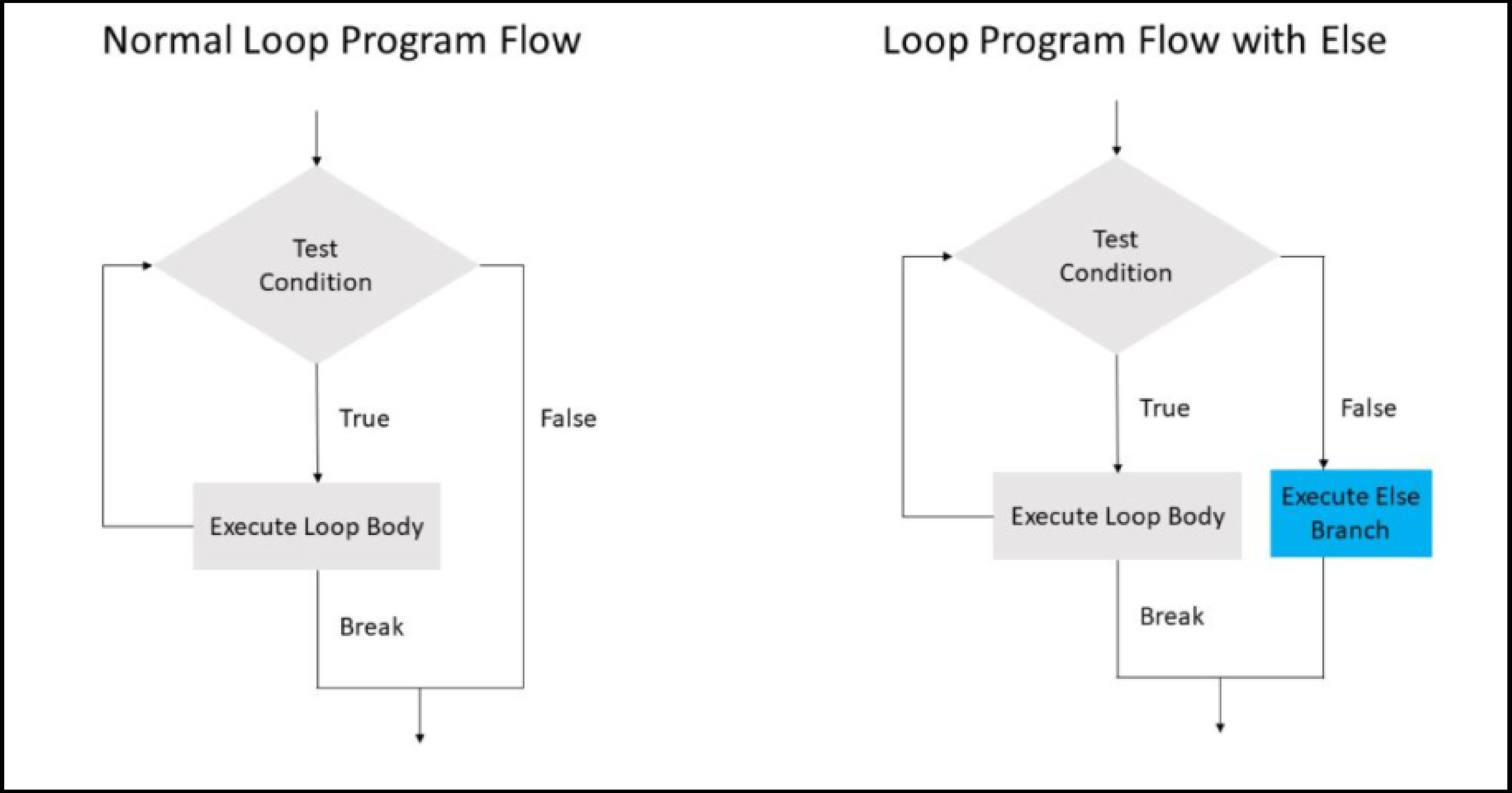
- Repeat a specific block of code
- Used to iterate over a block of code as long as the test expression (condition) is true
- While loop when we don't know the number of times to iterate beforehand

```
while test_expression:
    Body of while
```

WHILE LOOP


While Loop
3.b) If false
3.a) If true
6.
1.
2.
while (condition)
{
4.
// body of the loop
// statements to be executed
5.
updation
}
7.
// statements outside the loop

WHILE – ELSE



Normal Loop Program Flow
Test Condition
True
False
Execute Loop Body
Break
Loop Program Flow with Else
Test Condition
True
False
Execute Loop Body
Execute Else Branch
Break

PYTHON – BREAK AND CONTINUE

BREAK & CONTINUE

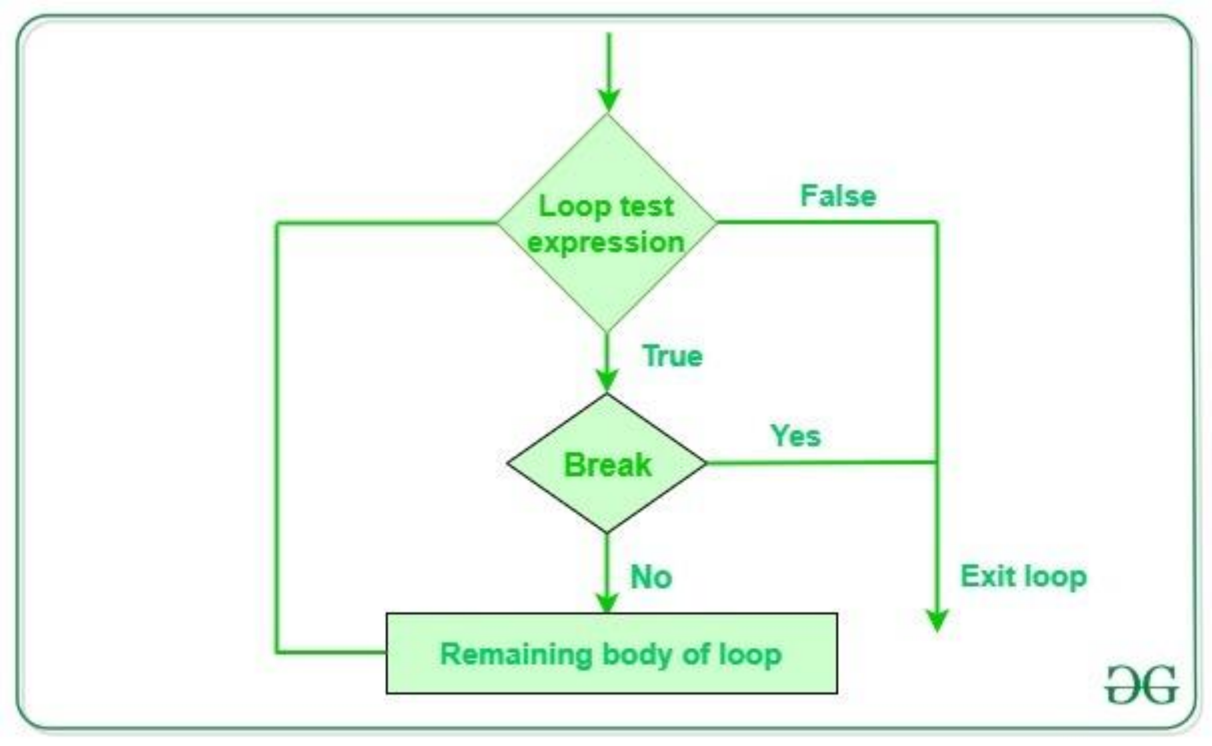
- To alter the flow of a loop
- Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression
- Break and continue

BREAK

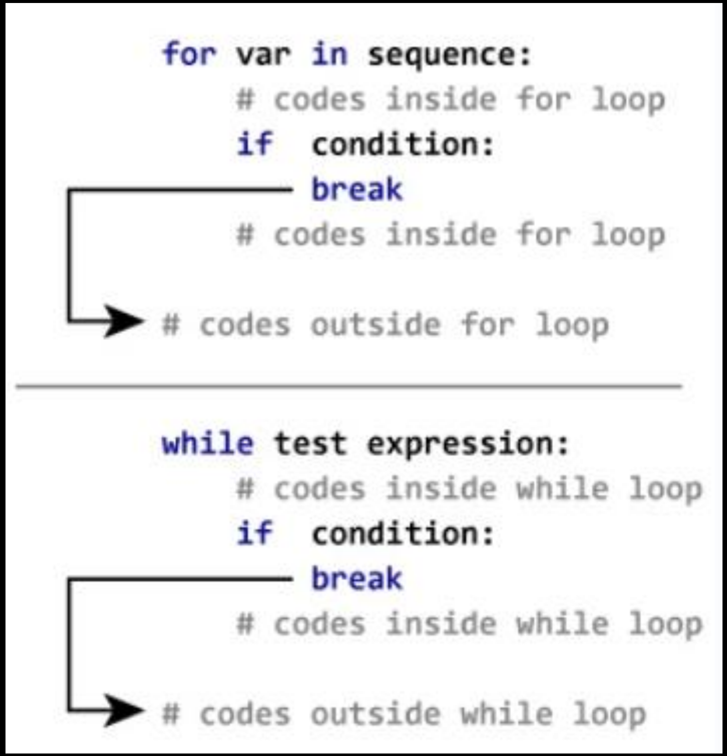
- terminates the loop containing it
- Control of the program flows to the statement immediately after the body of the loop
- If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop

```
break
```

BREAK – WORK FLOW



BREAK – FOR AND WHILE LOOPS



```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

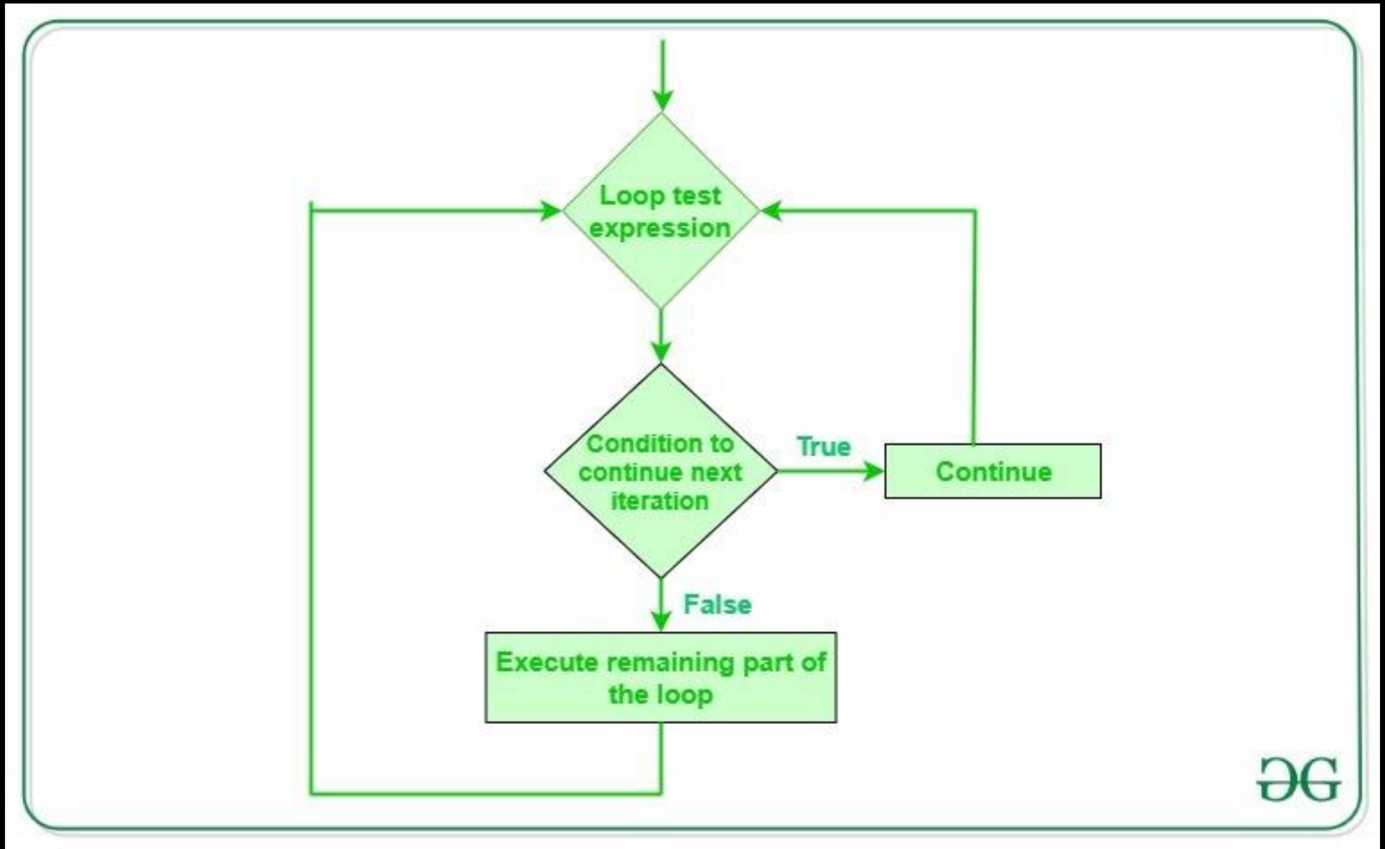
```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

CONTINUE STATEMENT

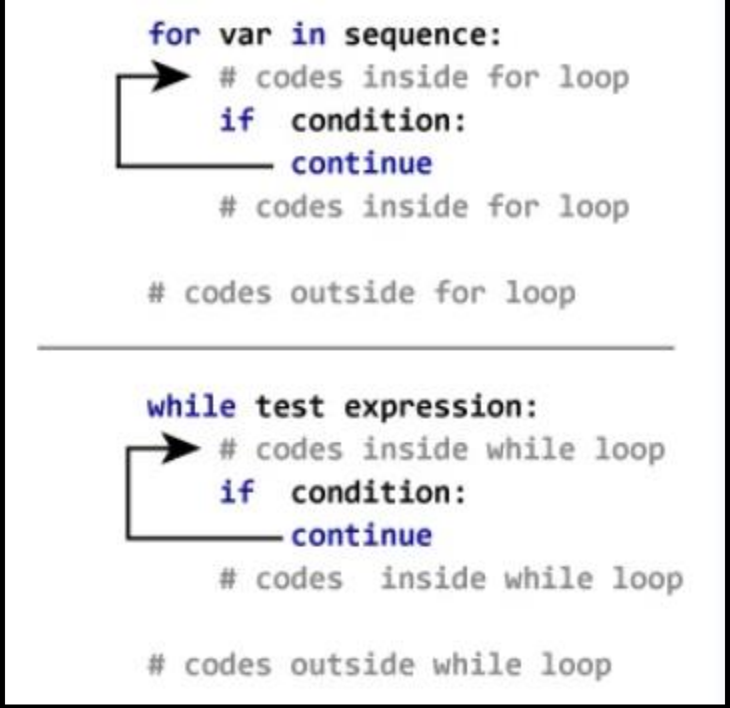
- The continue statement is used to skip the rest of the code inside a loop for the current iteration only.
- Loop does not terminate but continues on with the next iteration

```
continue
```

CONTINUE – WORK FLOW


Loop test expression
Condition to continue next iteration
True
Continue
False
Execute remaining part of the loop

CONTINUE STATEMENT FOR FOR AND WHILE LOOP



```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

PYTHON – PASS STATEMENT

PASS STATEMENT

- Null statement
- Difference with comment – not ignored and it's no operation
- Placeholder – future presprective

pass

PYTHON – FOR LOOP

Definite Loops

- Quite often we have a list of items of the lines in a file - 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”

A Simple Definite Loop

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

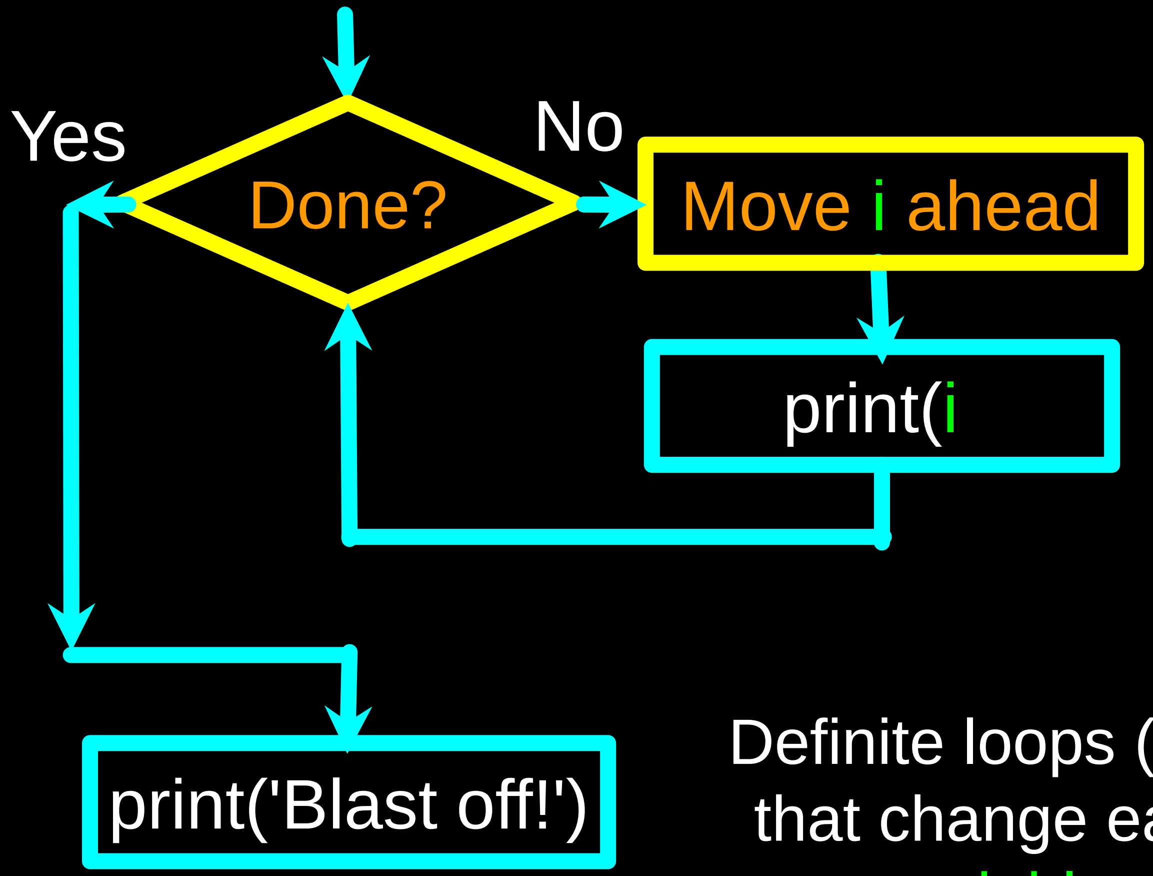
```
5
4
3
2
1
Blastoff!
```

A Definite Loop with Strings

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends :
   print('Happy New Year:', friend
print('Done!'
```

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally

Done!
```

A Simple Definite Loop



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

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

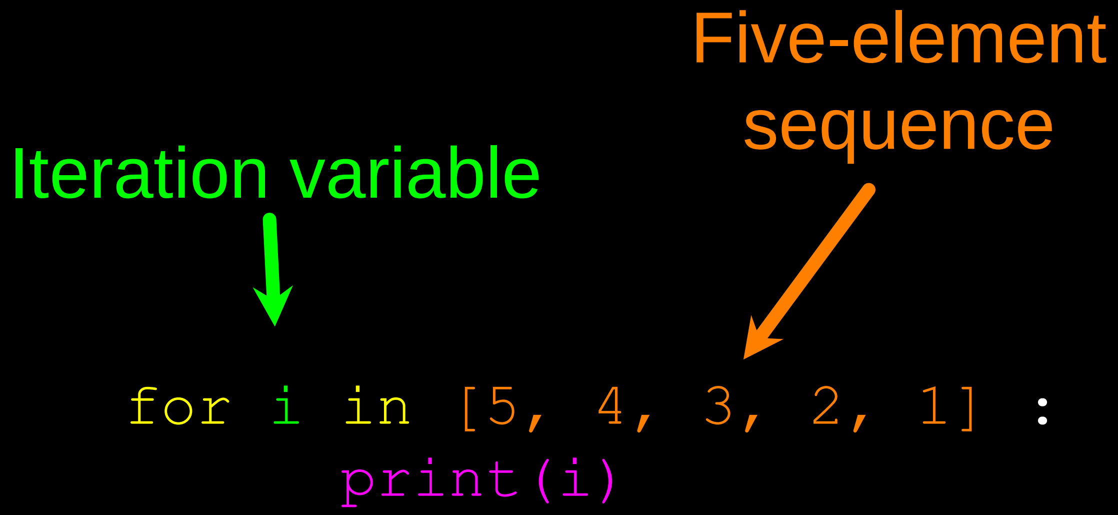
Definite loops (for loops) have explicit iteration variables that change each time through a loop. These iteration variables move through the sequence or set.

Looking at in...

- The iteration variable “iterates” through the sequence (ordered set)

- The block (body) of code is executed once for each value in the sequence

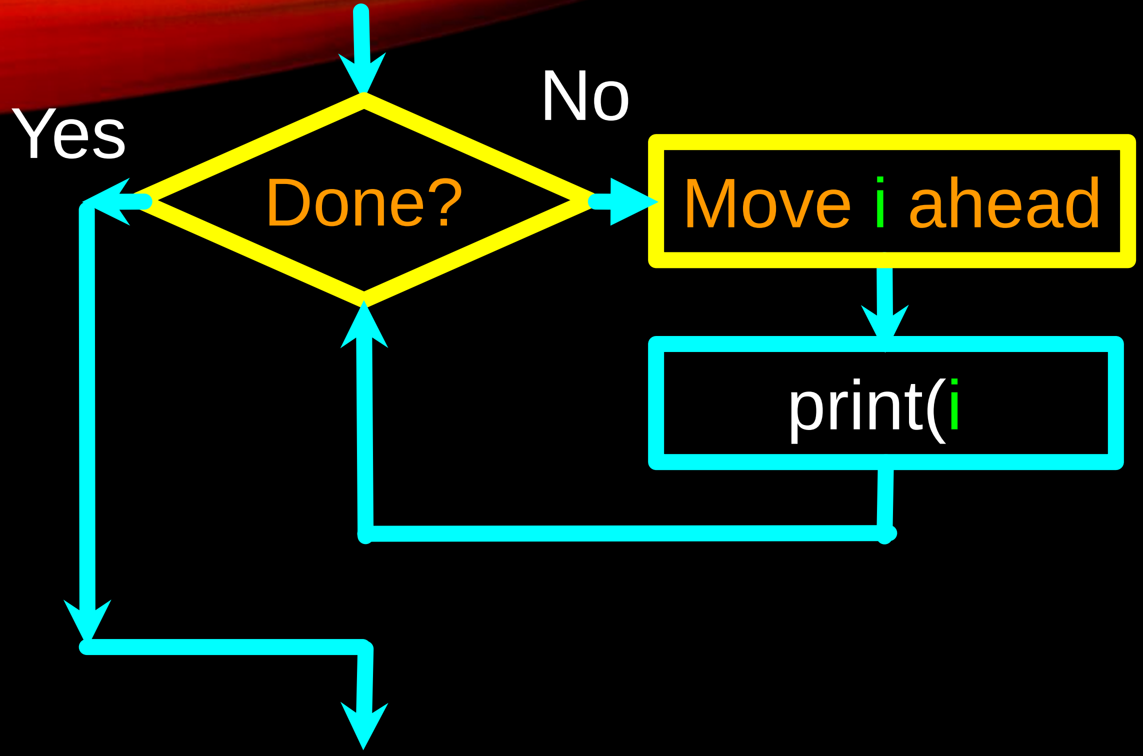
- The iteration variable moves through all of the values in the sequence



Five-element sequence

Iteration variable

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



Yes

No

Done?

Move i ahead

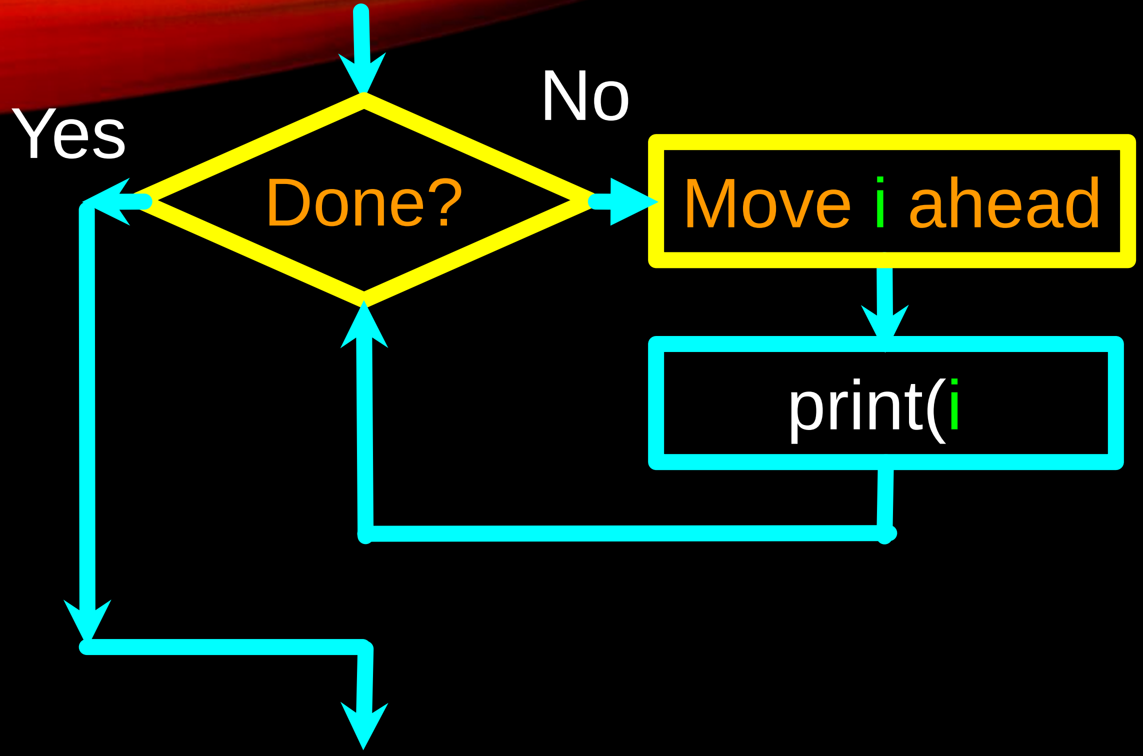
print(i

```
for i in [5, 4, 3, 2, 1] :
    print(i
```

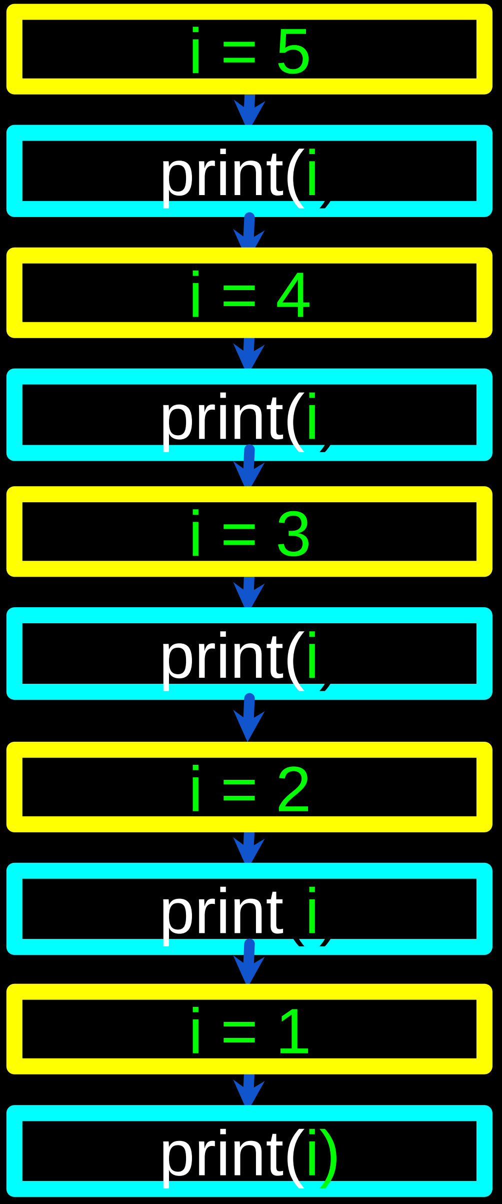
- The iteration variable “iterates” through the sequence (ordered set)

- The block (body) of code is executed once for each value in the sequence

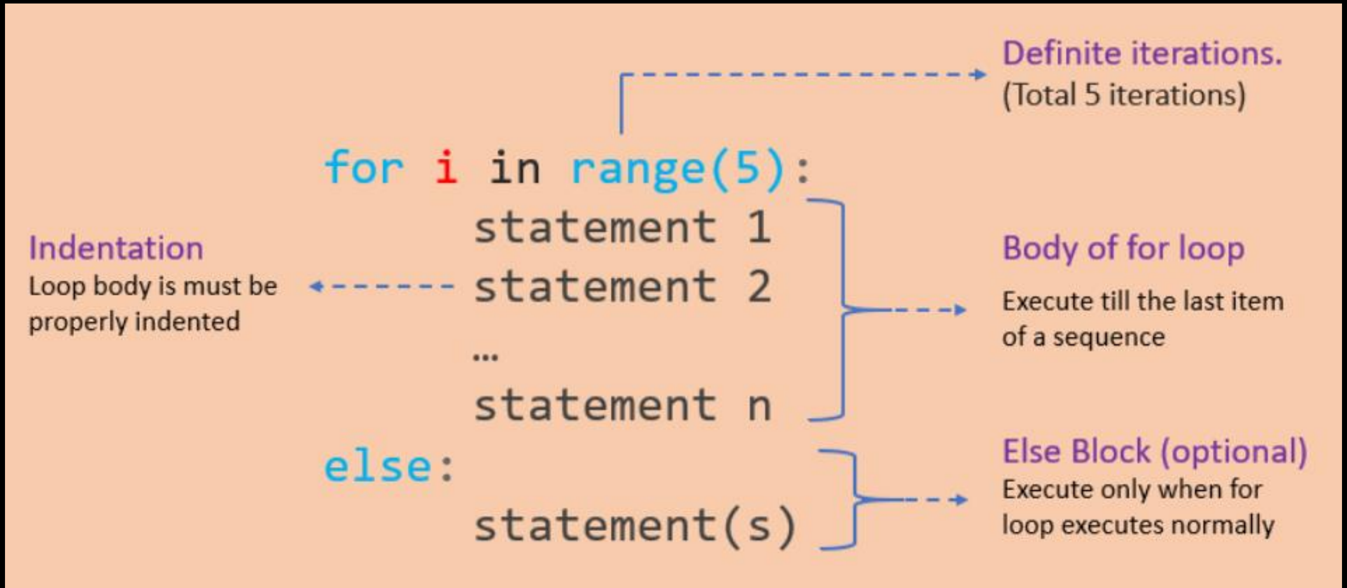
- The iteration variable moves through all of the values in the sequence



```
for i in [5, 4, 3, 2, 1] :
    print(i
```



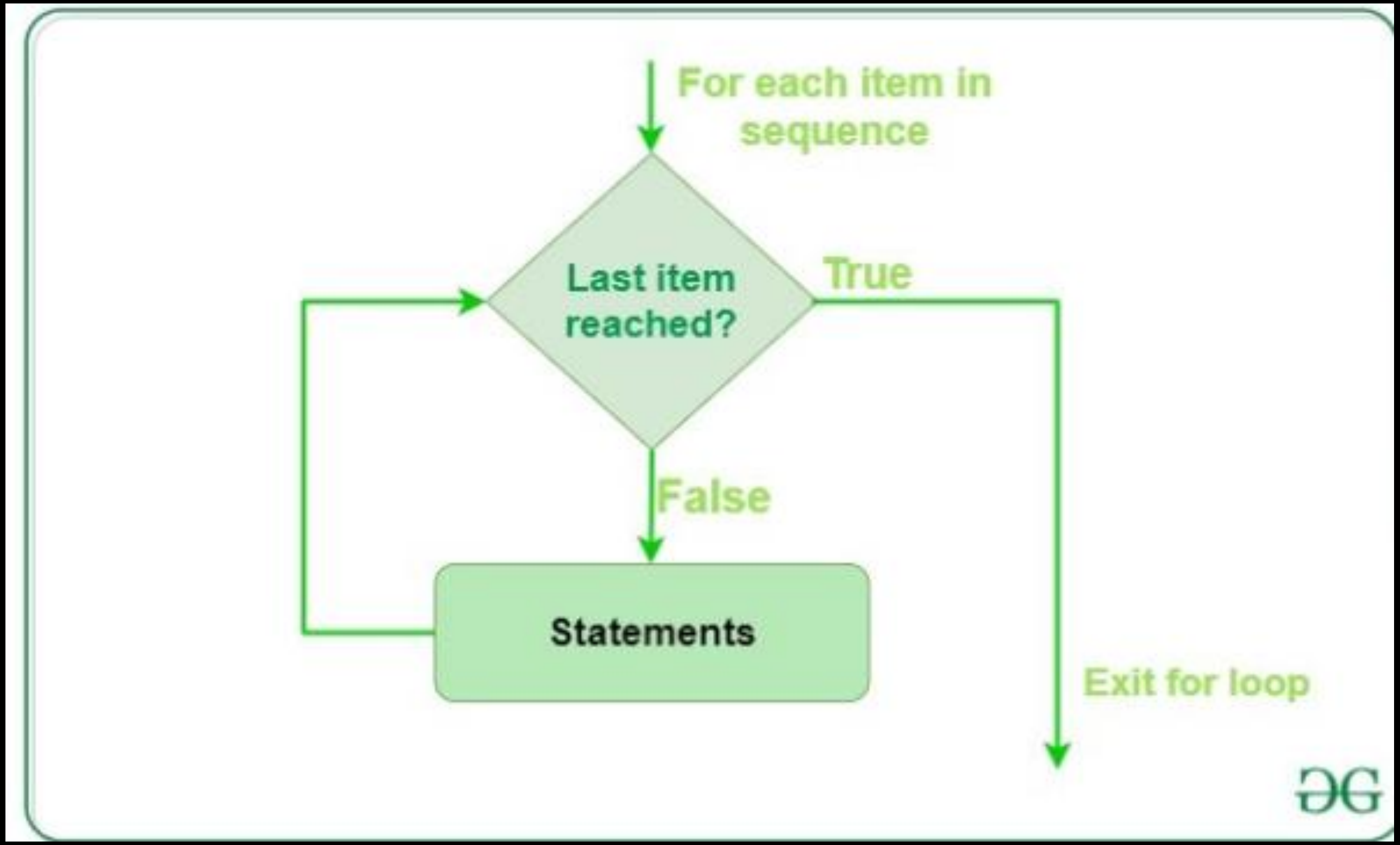
FOR LOOP



```
for i in range(5):
        statement 1
        statement 2
        ...
        statement n
else:
        statement(s)
```

- Used to iterate over a sequence like list, tuple, string or other
- Loop continues until we reach the last item in the sequence
- Identation

FOR LOOP – FLOW CONTROL



RANGE()

- Generate a sequence of numbers using range() function
- range(start, stop,step_size)
- Not an iterator since it supports in, len and __getitem__ operations

FOR…ELSE STATEMENT

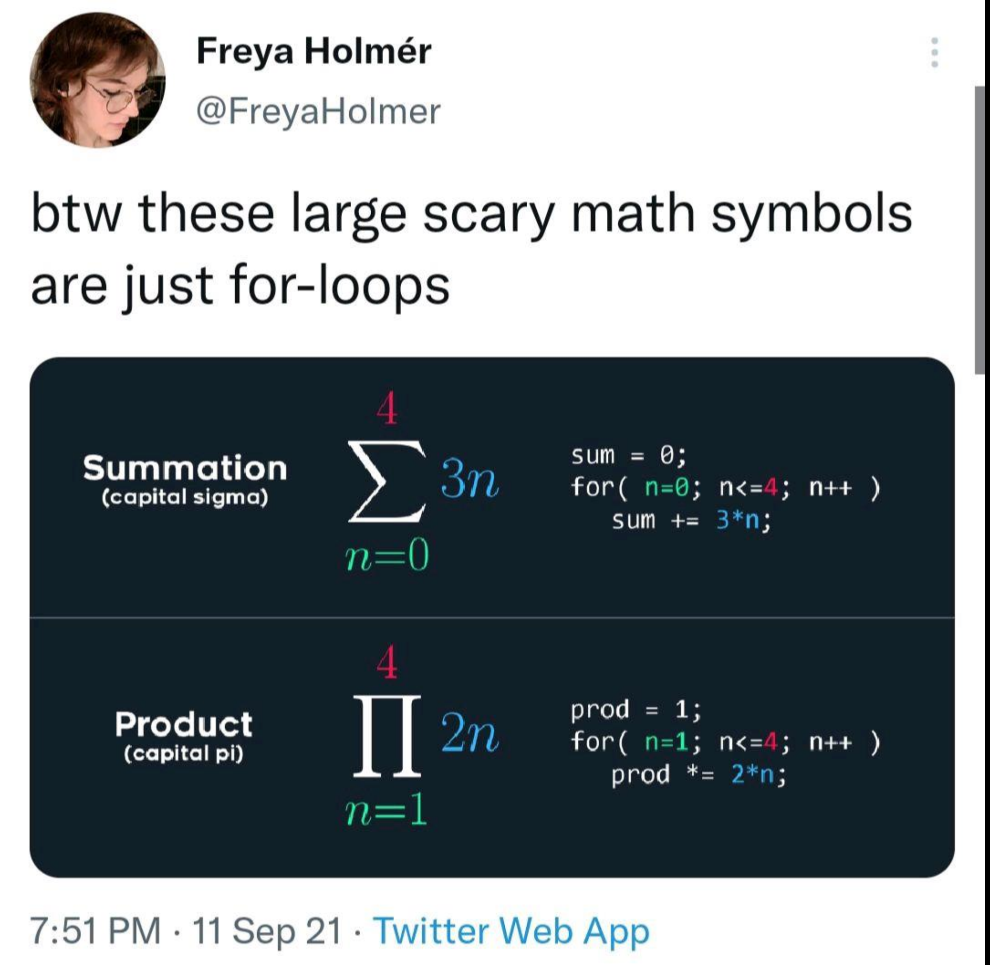
- for...else statement can be used with the break keyword to run the else block only when the break keyword was not executed

Making “smart” loops

The trick is “knowing” something about the whole loop when you are stuck writing code that only sees one entry at a time

```
Set some variables to initial values

for thing in data:
    Look for something or do something to each entry separately, updating a variable

Look at the variables
```

HOW EFFECTIVE OF USING FOR LOOP – MEME EXPLANATION



Freya Holmér
@FreyaHolmer

btw these large scary math symbols are just for-loops

Summation
(capital sigma)

$$\sum_{n=0}^{4} 3n$$

```
sum = 0;
for( n=0; n<=4; n++ )
    sum += 3*n;
```

Product
(capital pi)

$$\prod_{n=1}^{4} 2n$$

```
prod = 1;
for( n=1; n<=4; n++ )
    prod *= 2*n;
```

7:51 PM · 11 Sep 21 · Twitter Web App

Counting in a Loop

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

```
$ 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

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

```
$ python countloop.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 the 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 averageloop.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 25.666
```

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

Filtering in a Loop

```
print 'Before'
for value in [9, 41, 12, 3, 74, 15] :
    if value > 20:
       print('Large number',value)
print 'After'
```

```
$ python search1.py
Before
Large number 41
Large number 74
After
```

We use an if statement in the loop to catch / filter the values we are looking for.

Search Using a Boolean Variable

```
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 search1.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 starts at False and is set to True as soon as we find what we are looking for.

HOW TO FIND THE SMALLEST VALUE

```
largest_so_far = -1
print 'Before', largest_so_far
for the_num in [9, 41, 12, 3, 74, 15] :
   if the_num > largest_so_far :
      largest_so_far = the_num
   print largest_so_far, the_num

print 'After', largest_so_far
```

```
$ python largest.py
Before -1
9  9
41  41
41 12
41  3
74  74
74  15
After 74
```

How would we change this to make it find the smallest value in the list?

FINDING THE SMALLEST VALUE

```
smallest_so_far = -1
print 'Before', smallest_so_far
for the_num in [9, 41, 12, 3, 74, 15] :
   if the_num < smallest_so_far :
      smallest_so_far = the_num
   print smallest_so_far, the_num

print 'After', smallest_so_far
```

We switched the variable name to smallest_so_far and switched the > to <

FINDING THE SMALLEST VALUE

```
smallest_so_far = -1
print 'Before', smallest_so_far
for the_num in [9, 41, 12, 3, 74, 15] :
   if the_num < smallest_so_far :
      smallest_so_far = the_num
   print smallest_so_far, the_num

print 'After', smallest_so_far
```

```
$ python smallbad.py
Before -1
-1  9
-1  41
-1 12
-1  3
-1  74
-1  15
After -1
```

We switched the variable name to smallest_so_far and switched the > to <

FINDING THE SMALLEST VALUE

```
smallest = None
print 'Before'
for value in [9, 41, 12, 3, 74, 15] :
    if smallest is None :
        smallest = value
    elif value < smallest :
        smallest = value
    print smallest, value
print 'After', smallest
```

```
$ python smallest.py
Before
9 9
9 41
9 12
3 3
3 74
3 15
After 3
```

We still have a variable that is the smallest so far. The first time through the loop smallest is None, so we take the first value to be the smallest.

The is and is not Operators

```
smallest = None
print('Before')
for value in [3, 41, 12, 9, 74, 15] :
    if smallest is None :
        smallest = value
    elif value < smallest :
        smallest = value
    print(smallest, value)

print('After', smallest
```

- Python has an is operator that can be used in logical expressions

- Implies “is the same as”

- Similar to, but stronger than ==

- is not also is a logical operator