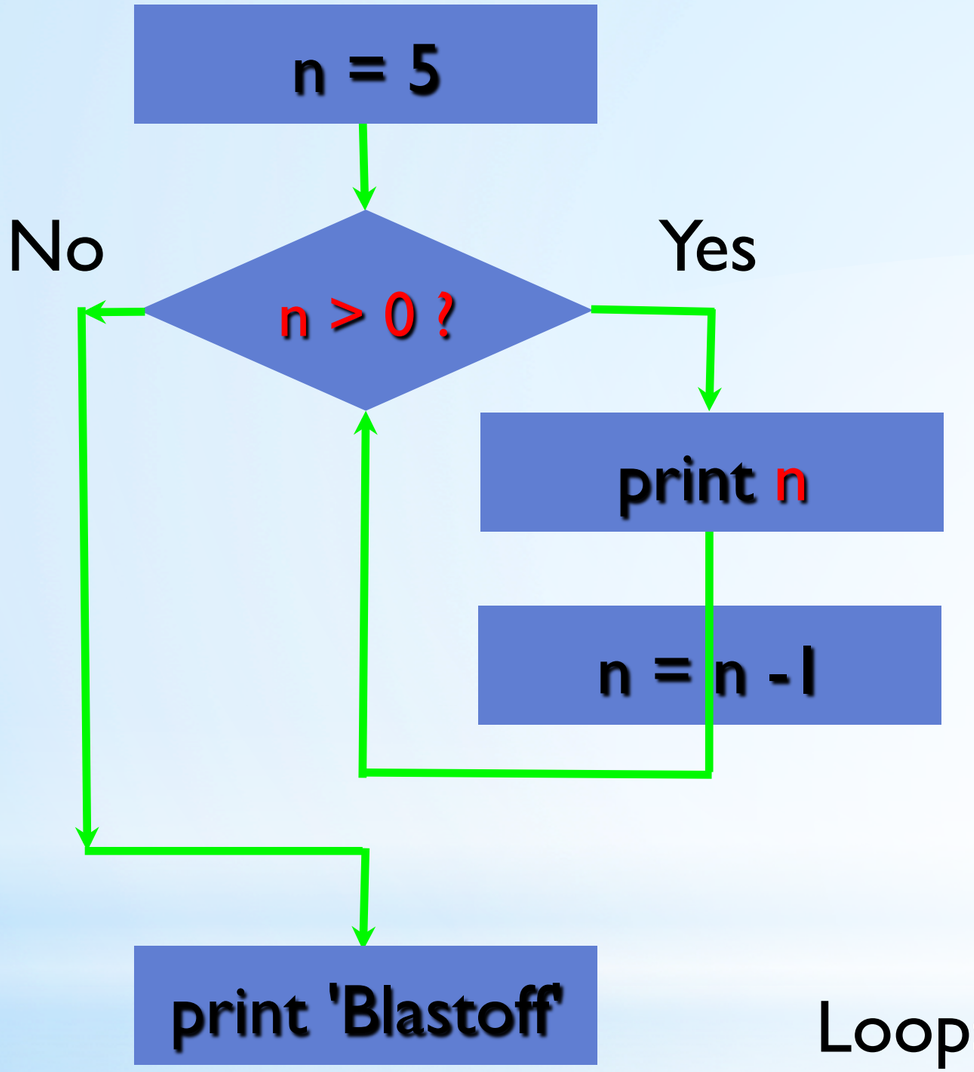
Computer Applications to Behavioral Sciences

Lecture 5. Loops &Iterations

* Dr. Jibo He
* Wichita State University
* hejibo@gmail.com

https://psychology-courses.appspot.com/

*Repeated Steps



Program:

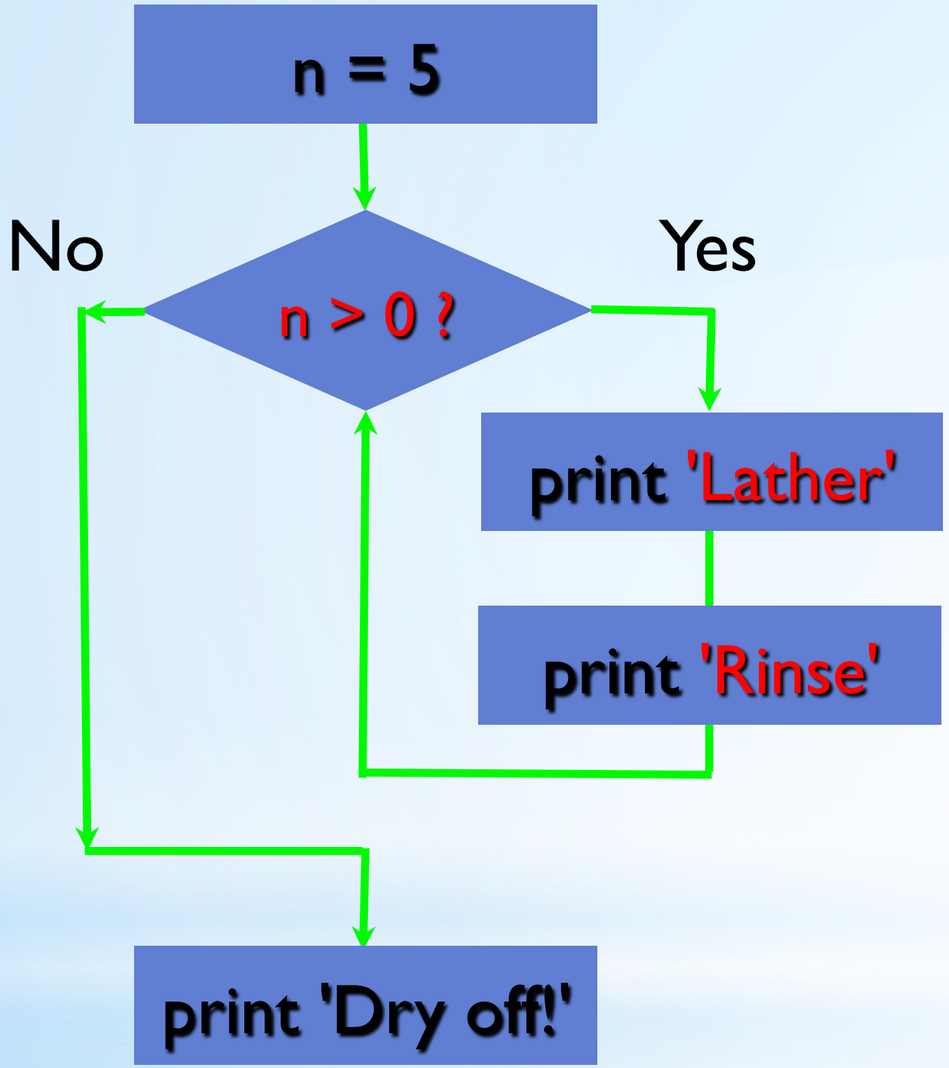
```
n = 5
while n > 0 :
    print n
    n = n – 1
print 'Blastoff!'
print n
```

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.

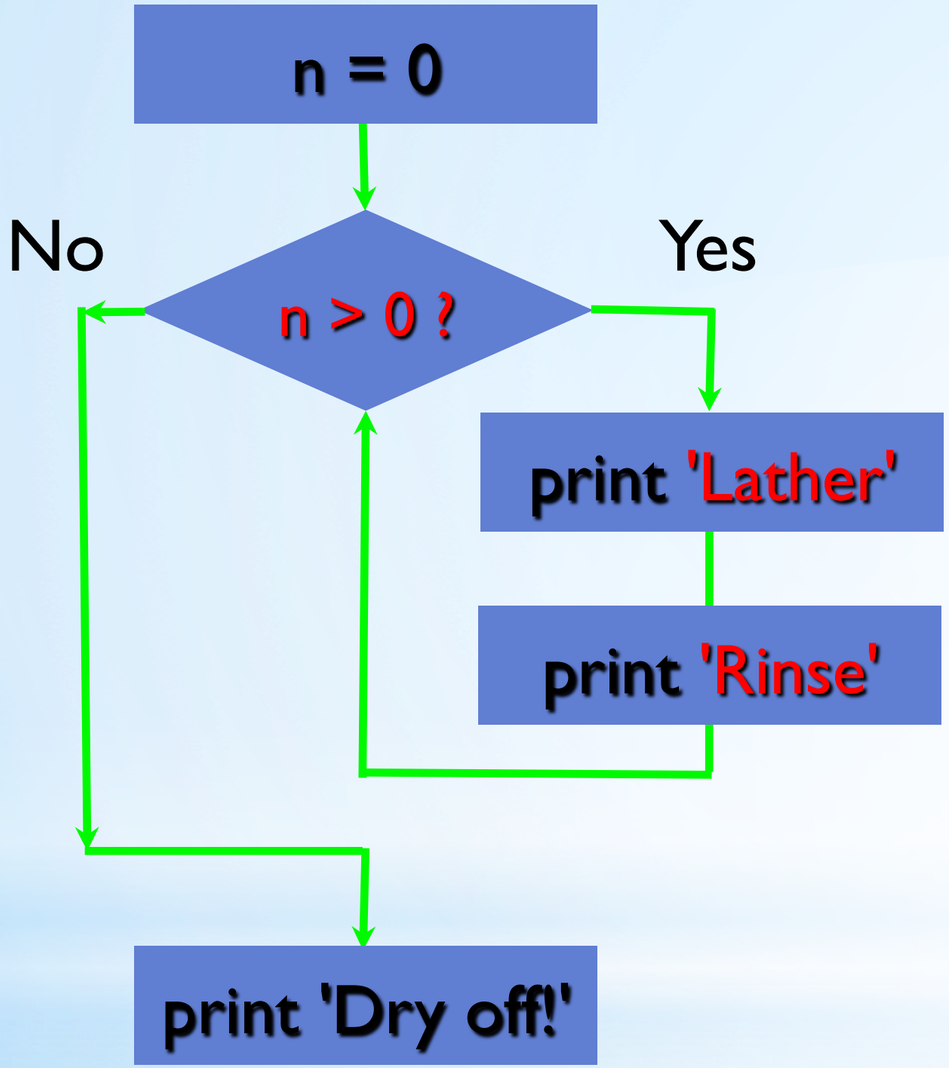
*An Infinite Loop




```
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 does this loop do?

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 = raw_input('> ')
    if line == 'done' :
        break
    print line
print 'Done!'
```

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

Breaking Out of a Loop

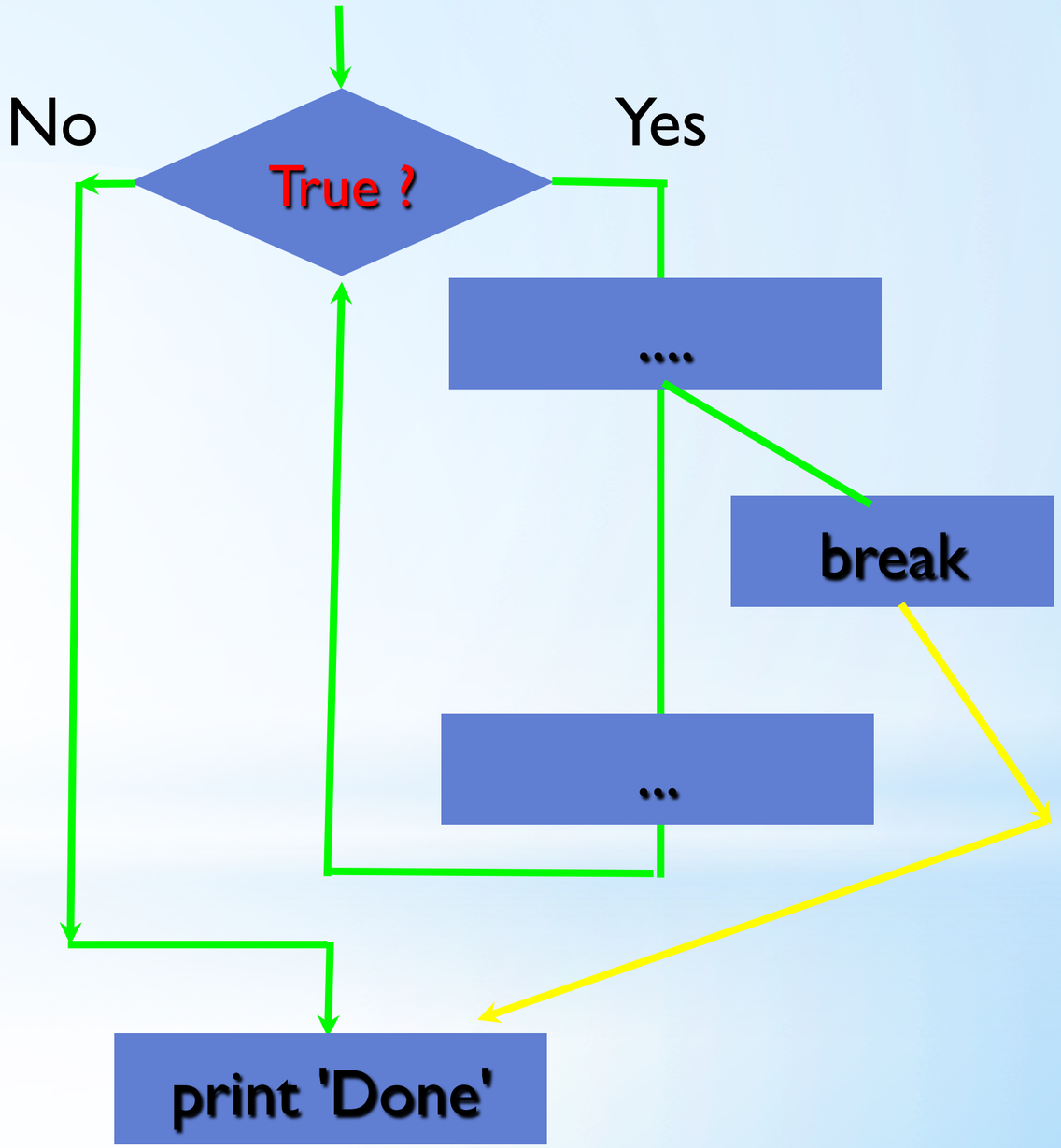
- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

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

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

```
while True:
    line = raw_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 = raw_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!
```

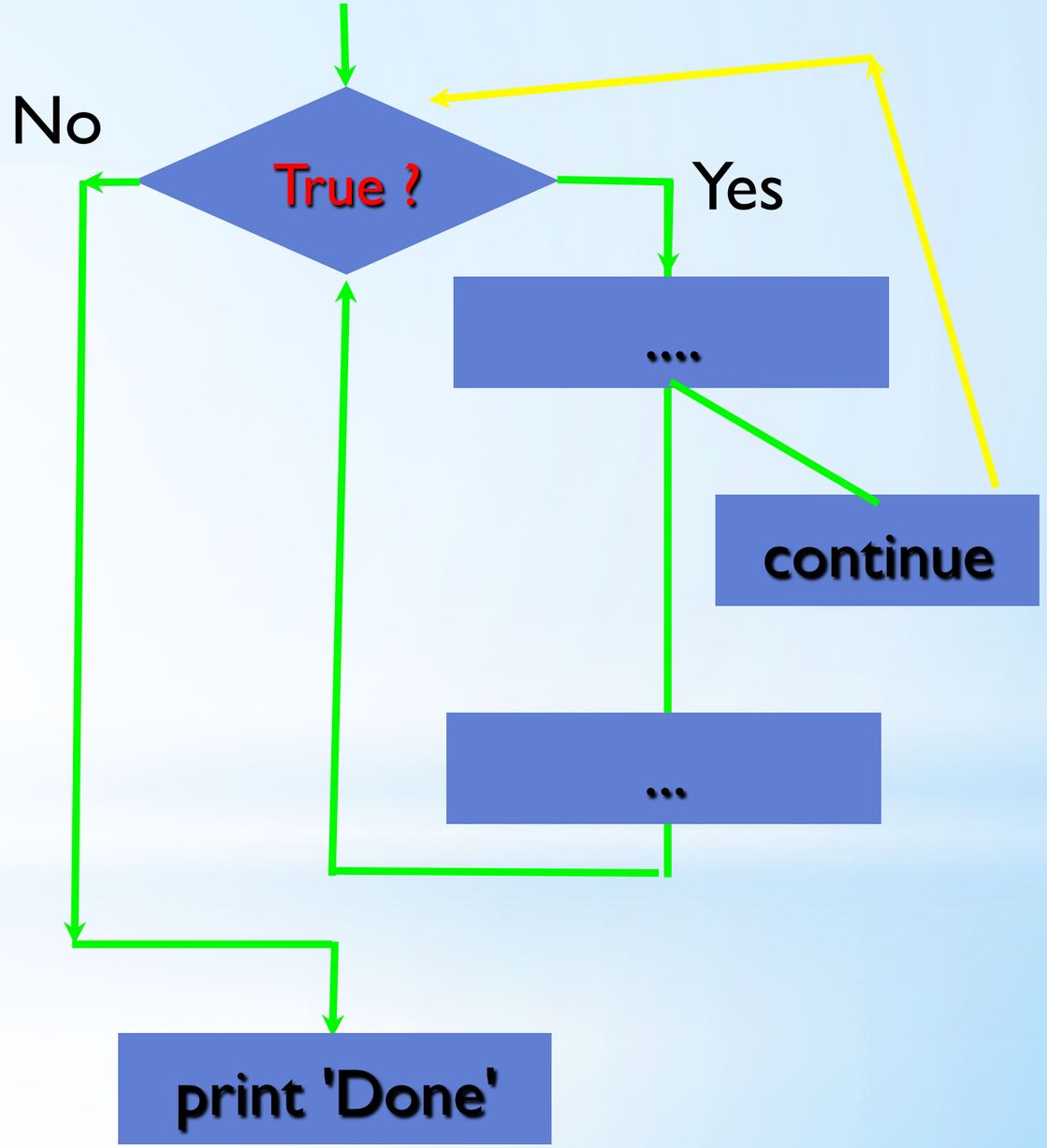
Finishing an Iteration with continue

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

```
while True:
    line = raw_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

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

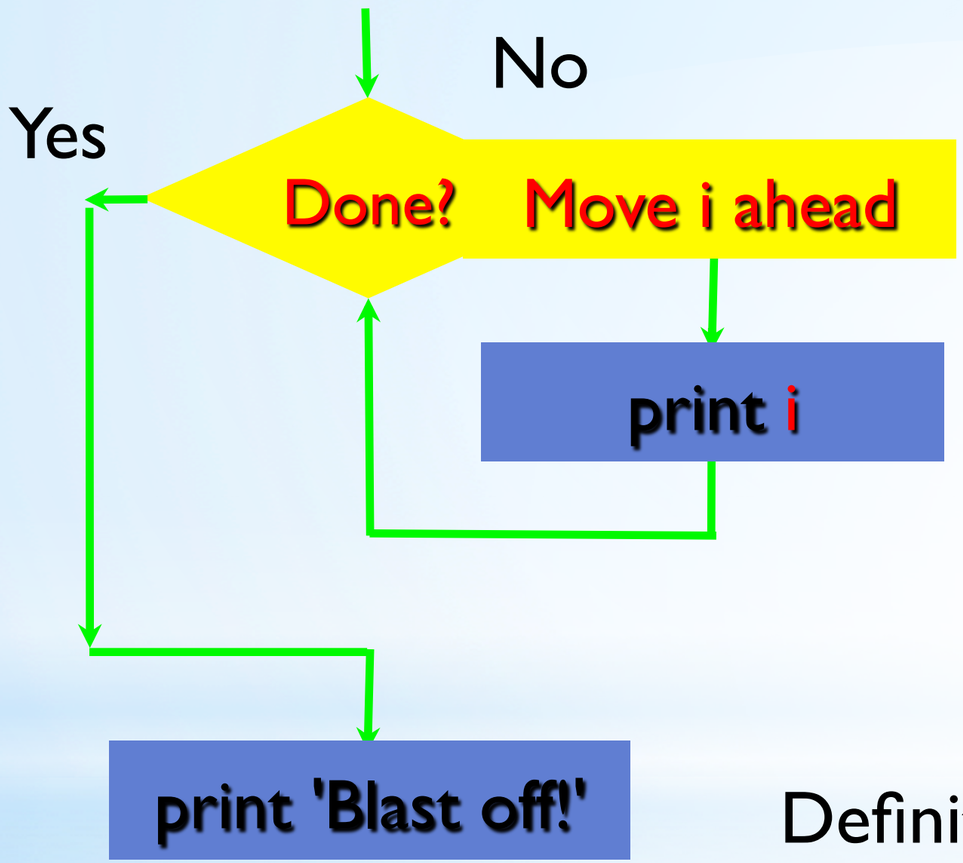
```
for i in [5, 4, 3, 2, 1] :
    print i
print 'Blastoff!'
```

```
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!'
```

*A Simple Definite Loop



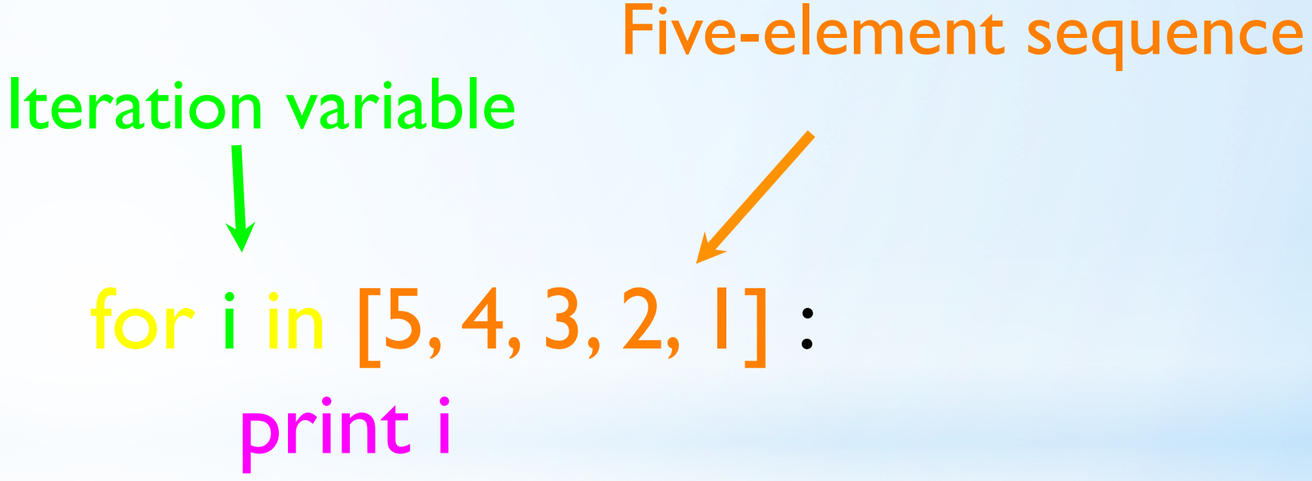

```
for i in [5, 4, 3, 2, 1] :
    print i
print 'Blastoff!'
```

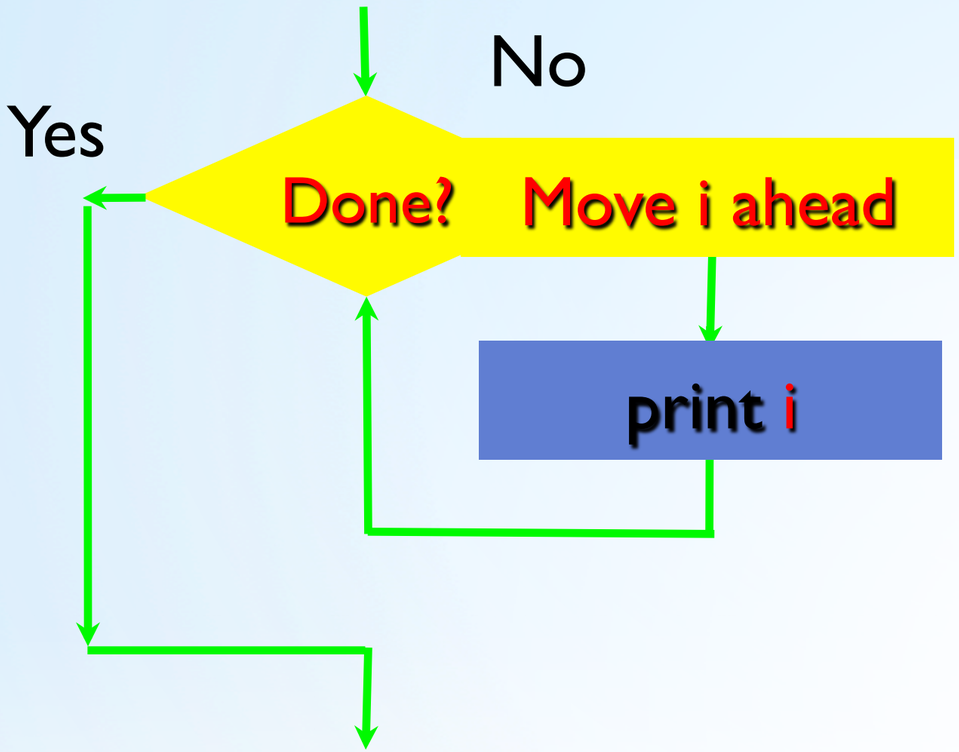
```
5
4
3
2
1
Blastoff!
```

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…

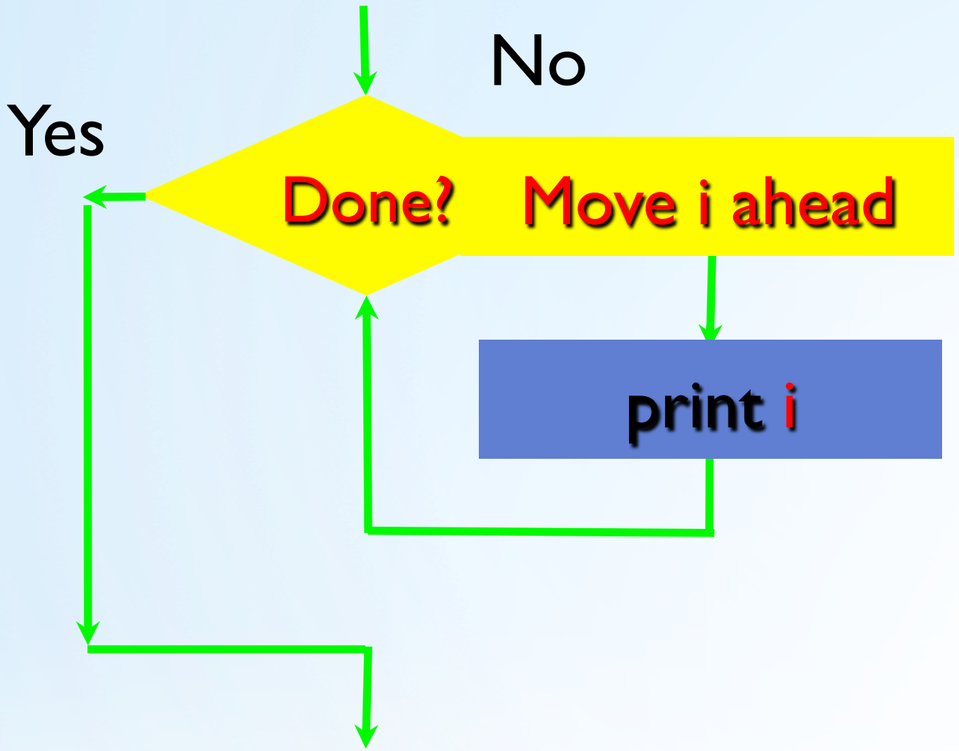
- The iteration variable "iterates" though the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence



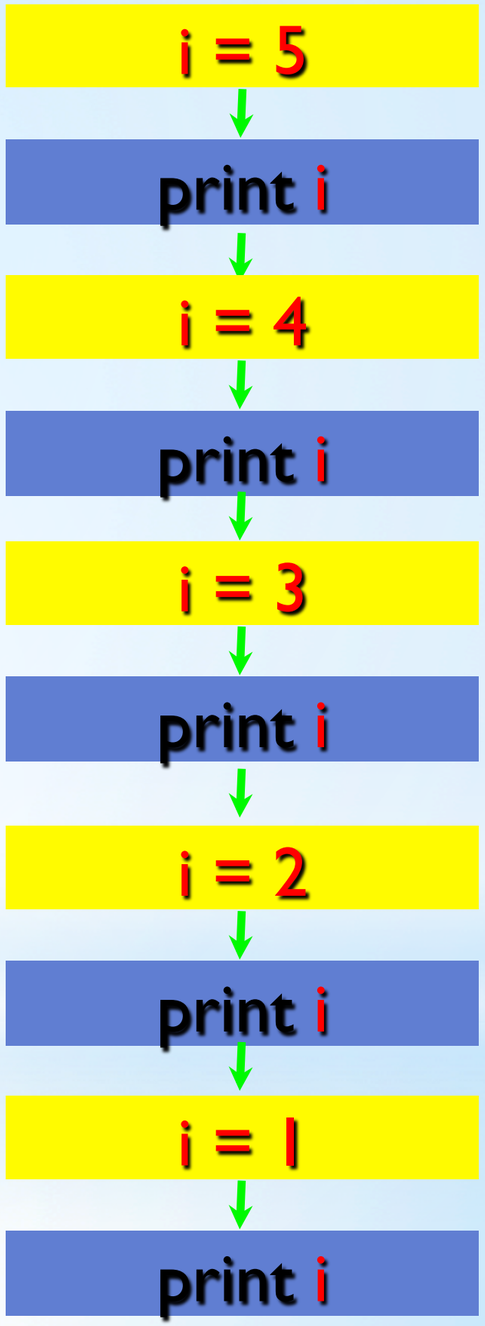



```
for i in [5, 4, 3, 2, 1]
:    print i
```

- The iteration variable "iterates" though 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
```



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"

*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.

*Looping through a Set

```
print 'Before'
for thing in [9, 41, 12, 3, 74, 15] :
    print thing
print 'After'
```

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

*What is the Largest Number?

*What is the Largest Number?

3 41 12 9 74 15

largest_so_far

-1 3 41 74

*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
```

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.

* What is the Smallest Number?

*What is the Smallest Number?

9 41 12 3 74 15

smallest_so_far | -1

*What is the Smallest Number?

9 41 12 3 74 15

largest_so_far

None 9 3

* 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" operaror that can be used in logical expressions
* Implies 'is the same as'
* Similar to, but stronger than ==
* 'is not' also is a logical operator

Summary

- While loops (indefinite)
- Infinite loops
- Using break
- Using continue
- For loops (definite)
- Iteration variables
- Largest or smallest