

CMPE 102

Programming Logic and Design

Module 6

Iteration/Looping in Python

In this module you will learn how to rerun parts of your program without duplicating instructions

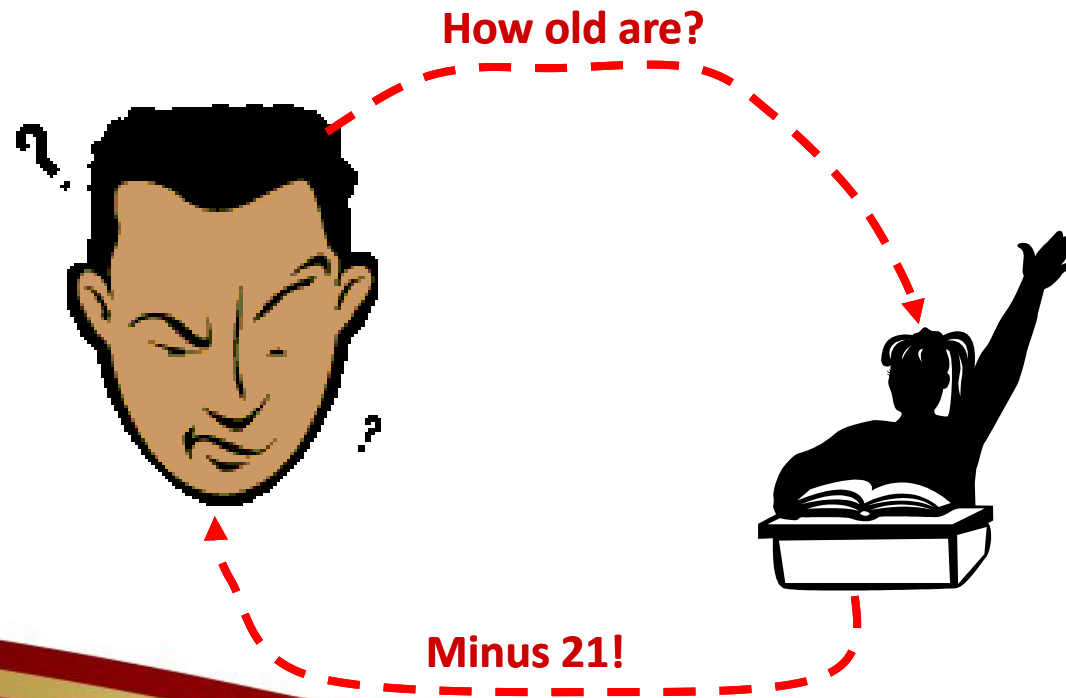
Repetition Flow Control

- Computers are often used to do *repetitive tasks* because humans don't like to do the same thing over and over again.
- In computer programs, **repetition flow control** is used to execute a group of instructions repeatedly.
- **Repetition flow control** is also called **iteration** or **loop**.
- In Python, repetition flow control can be expressed by a **for-statement** or a **while-statement** that allow us to execute a code block repeatedly.

Repetition: Computer View

- Continuing a process as long as a certain condition has been met.

Ask for age as long as the answer is negative (outside allowable range)



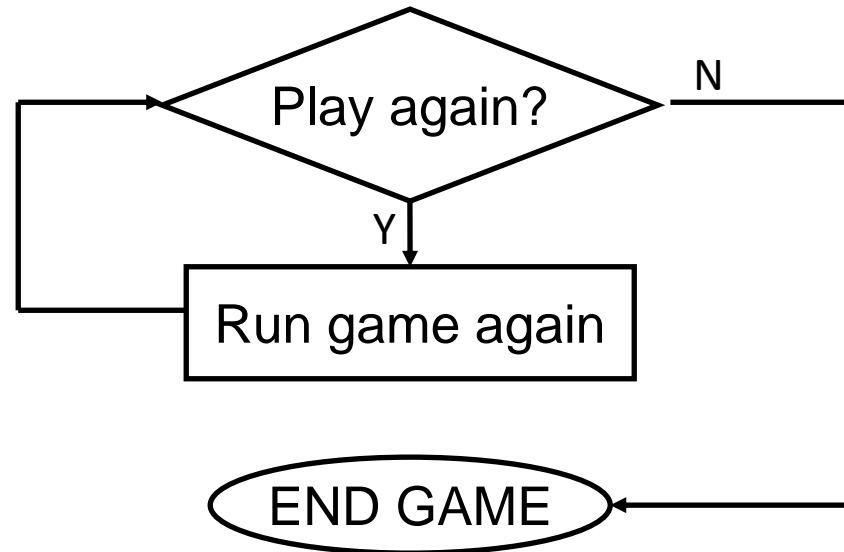
How To Determine If Loops Can Be Applied

- Something needs to occur multiple times (generally it will repeat itself as long as some condition has been met).
- Example 1:



Re-running the entire program

Flowchart



Pseudo code

While the player wants to play
Run the game again

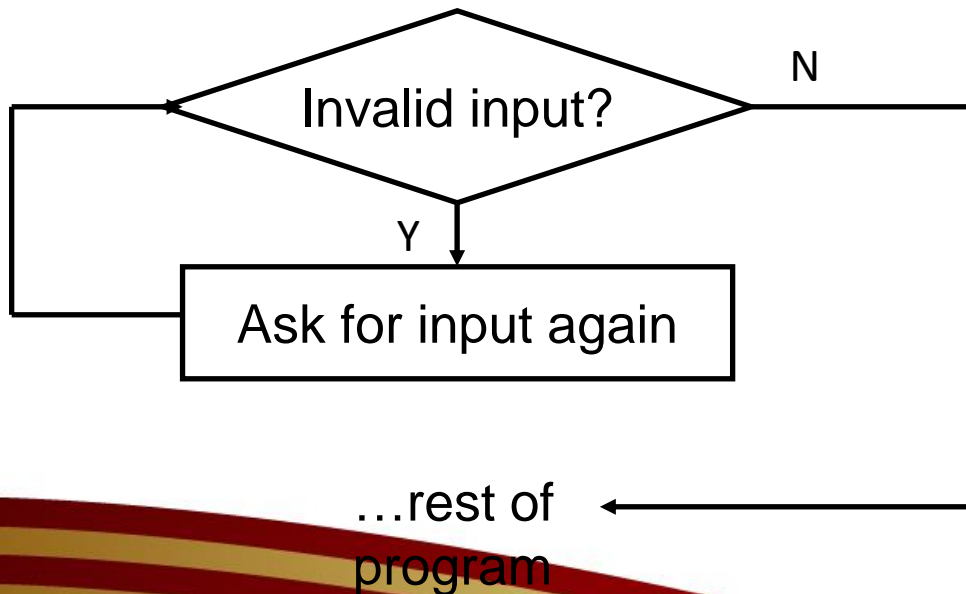
How To Determine If Loops Can Be Applied

- Example 2:

```
Enter your age (must be non-negative): -1
Enter your age (must be non-negative): 27
Enter your gender (m/f): 
```

Re-running specific parts of the program

Flowchart



Pseudo code

While input is invalid
 Prompt user for input

Basic Structure Of Loops

Whether or not a part of a program repeats is determined by a loop control (typically the control is just a variable).

- Initialize the control to the starting value
- Testing the control against a stopping condition (Boolean expression)
- Executing the body of the loop (the part to be repeated)
- Update the value of the control

Types Of Loops

1.Pre-test loops

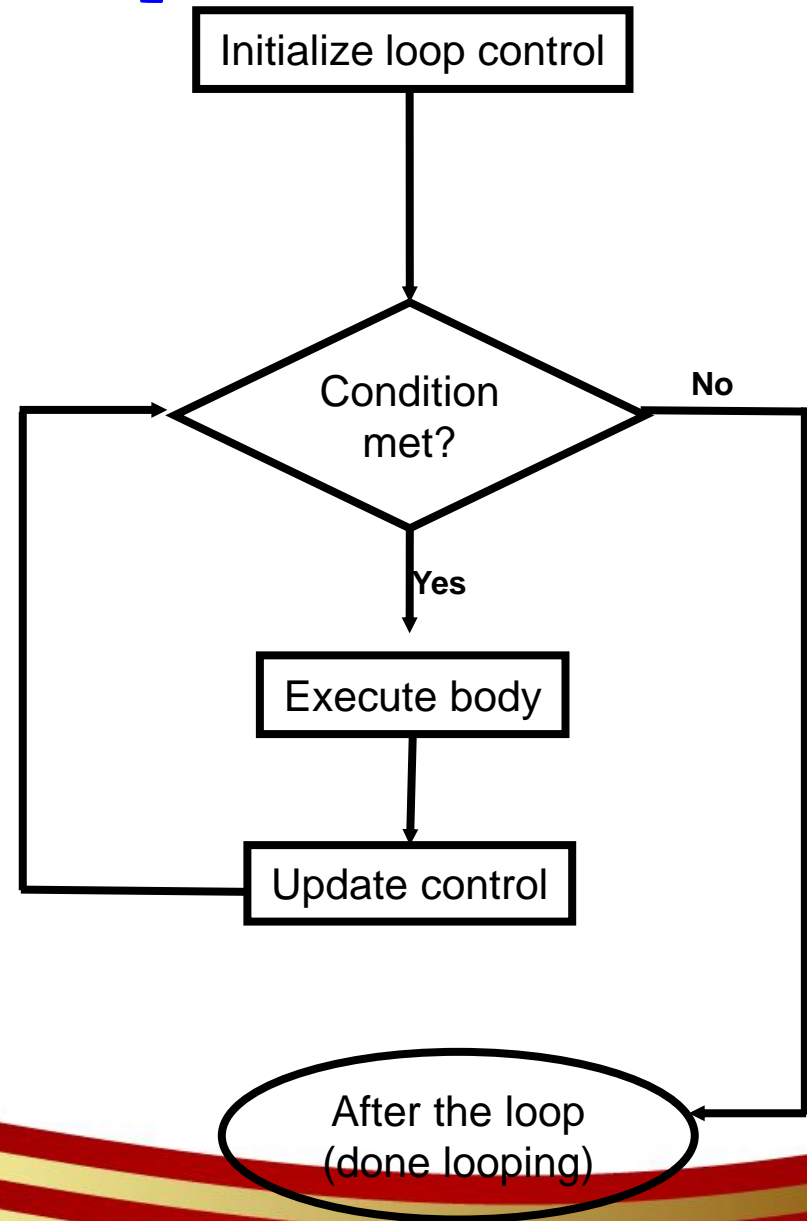
- Check the stopping condition *before* executing the body of the loop.
- The loop executes *zero or more* times.

2.Post-test loops

- Checking the stopping condition *after* executing the body of the loop.
- The loop executes *one or more* times.

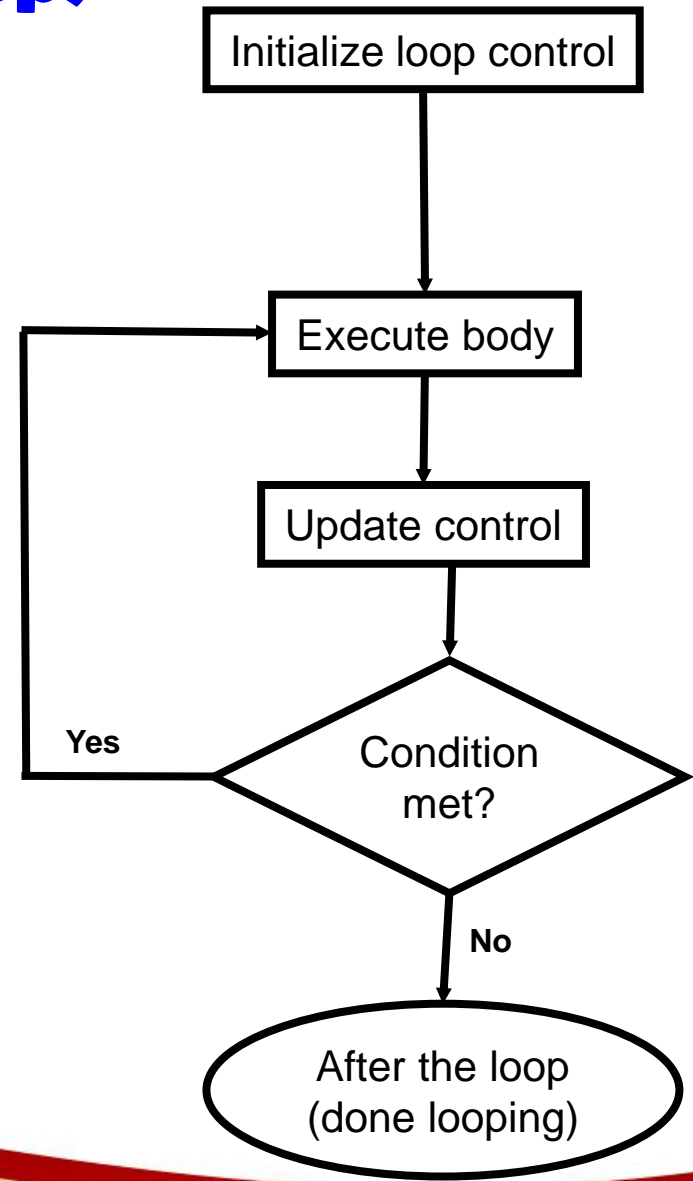
Pre-Test Loops

1. Initialize loop control
2. Check if the repeating condition has been met
 - a. If it's been met then go to Step 3
 - b. If it hasn't been met then the loop ends
3. Execute the body of the loop (the part to be repeated)
4. Update the loop control
5. Go to step 2



Post-Test Loops

1. Initialize loop control
(sometimes not needed because initialization occurs when the control is updated)
2. Execute the body of the loop
(the part to be repeated)
3. Update the loop control
4. Check if the repetition condition has been met
 - a. If the condition has been met then go through the loop again (go to Step 2)
 - b. If the condition hasn't been met then the loop ends.



Pre-Test Loops In Python

1. While

2. For

Characteristics:

1. The stopping condition is checked *before* the body executes.
2. These types of loops execute zero or more times.

Post-Loops In Python

- Note: this type of looping construct has not been implemented with this language.
- But many other languages do implement post test loops.

Characteristics:

- The stopping condition is checked *after* the body executes.
- These types of loops execute one or more times.

The *while* Statement

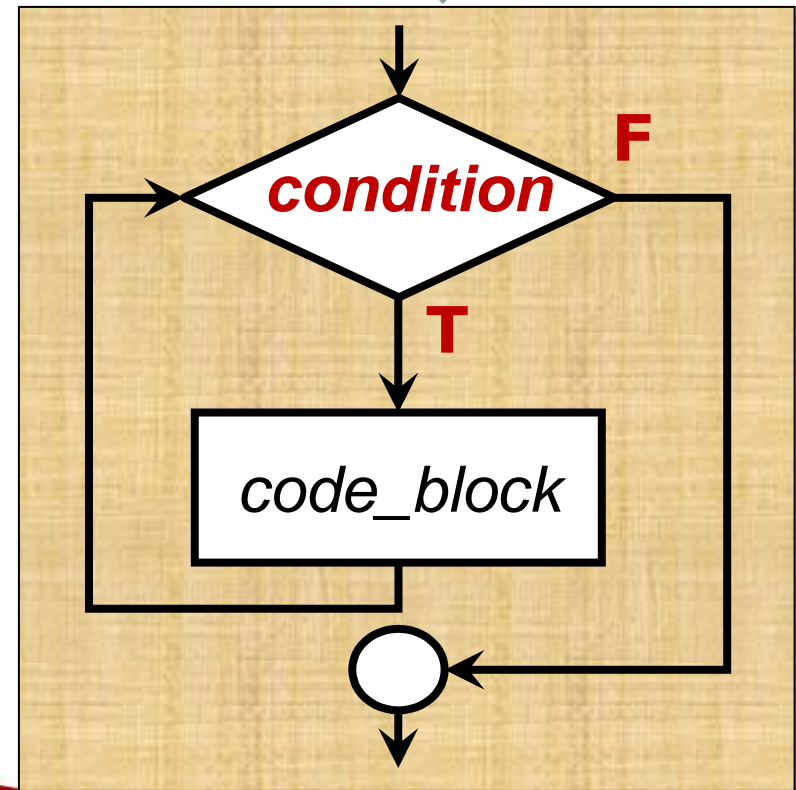


Python Syntax

```
while condition:  
    code_block
```

- *condition* is a Boolean expression.
- *code_block* is, as usual, an indented sequence of one or more statements.

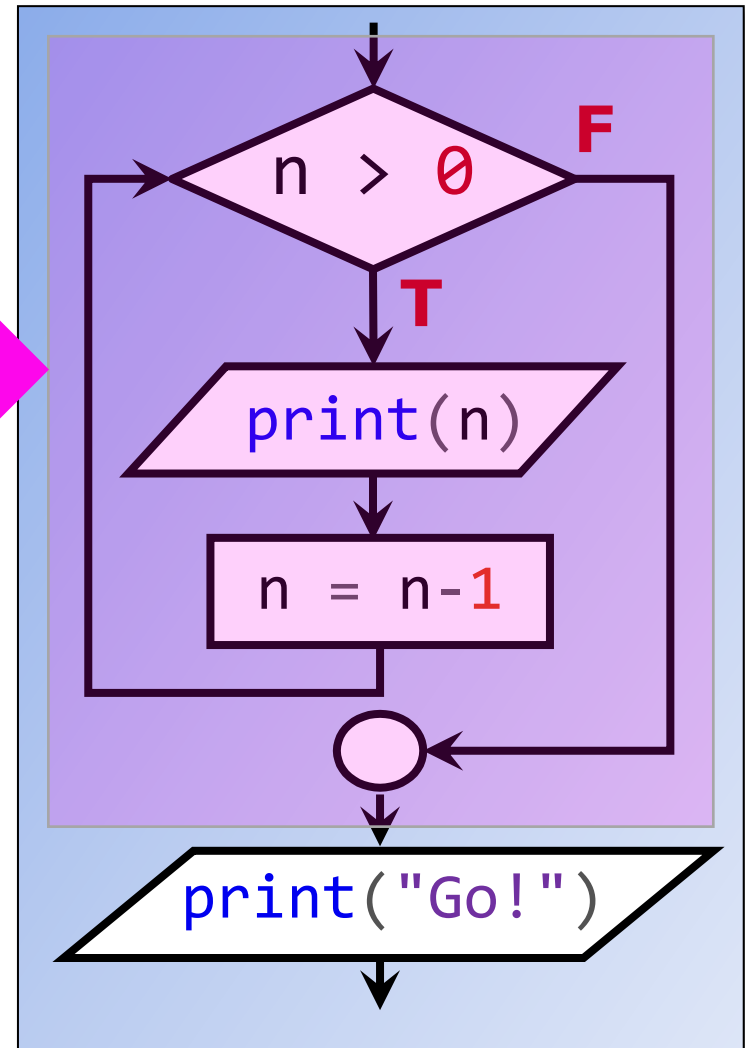
Semantics



Example

```
n = int(input("Enter a number"))
while n > 0:
    print(n)
    n = n-1
    print("Go!")
```

```
>>> countdown
Enter a number: 4
4
3
2
1
Go!
>>> countdown
Enter a number: 0
Go!
```



This means, in this case, that the loop body doesn't get executed at all. **Why?**

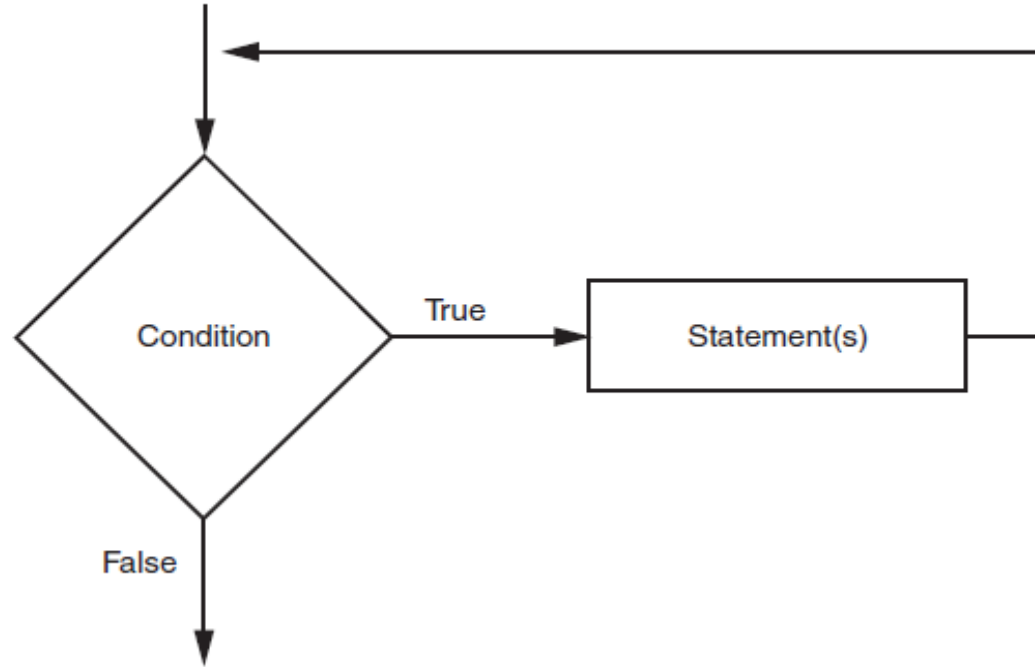
The *while* Loop: a Condition-Controlled Loop

- while loop: while condition is true, do something
 - Two parts:
 - Condition tested for true or false value
 - Statements repeated as long as condition is true
 - In flow chart, line goes back to previous part
 - General format:

```
while condition:  
    statements
```

The *while* Loop: a Condition-Controlled Loop

Figure 4-1 The logic of a while loop



The *while* Loop: a Condition-Controlled Loop

- In order for a loop to stop executing, something has to happen inside the loop to make the condition false
- Iteration: one execution of the body of a loop
- `while` loop is known as a *pretest* loop
 - Tests condition before performing an iteration
 - Will never execute if condition is false to start with
 - Requires performing some steps prior to the loop

The *while* Loop: a Condition-Controlled Loop

- This type of loop can be used if it's *not known* in advance how many times that the loop will repeat (most powerful type of loop, any other type of loop can be simulated with a while loop).
 - It can repeat so long as some arbitrary condition holds true.

- **Syntax:**

(Simple condition)

```
while (Boolean expression):  
    body
```

(Compound condition)

```
while (Boolean expression) Boolean operator (Boolean expression):  
    body
```

The *while* Loop: a Condition-Controlled Loop

- Program name: while1.py

```
i = 1
```

```
while (i <= 3):
```

```
    print("i =", i)
```

```
    i = i + 1
```

```
print("Done!")
```

1) Initialize control

2) Check condition

3) Execute body

4) Update control

The while Loop: a Condition-Controlled Loop

- **Program name:** `while1.py` **Tracing:**

```
i = 1
while (i <= 5):
    print("i =", i)
    i = i + 1
print("Done!")
```

Execution	Variable
>python while1.py	i

Countdown Loop

- **Program name:** while2.py **Tracing:**

```
i = 3
while (i >= 1):
    print("i =", i)
    i = i - 1
print("Done!")
```

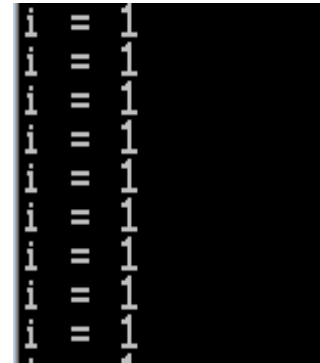
Execution	Variable
>python while2.py	i

Common Mistakes: While Loops

- Forgetting to include the basic parts of a loop.

- Updating the control

```
i = 1
while(i <= 4):
    print("i =", i)
```




Illustrative example

average2.py ×

```
1 # average2.py
2 #     A program to average a set of numbers
3 #     Illustrates interactive loop with two accumulators
4
5 moredata = "yes"
6 sum = 0.0
7 count = 0
8 while moredata[0] == 'y':
9     x = int(input("Enter a number >> "))
10    sum = sum + x
11    count = count + 1
12    moredata = input("Do you have more numbers (yes or no)? ")
13 print ("\nThe average of the numbers is: ", sum / count)
14
```

Definite Loops: the *for* Statement

Python
Syntax



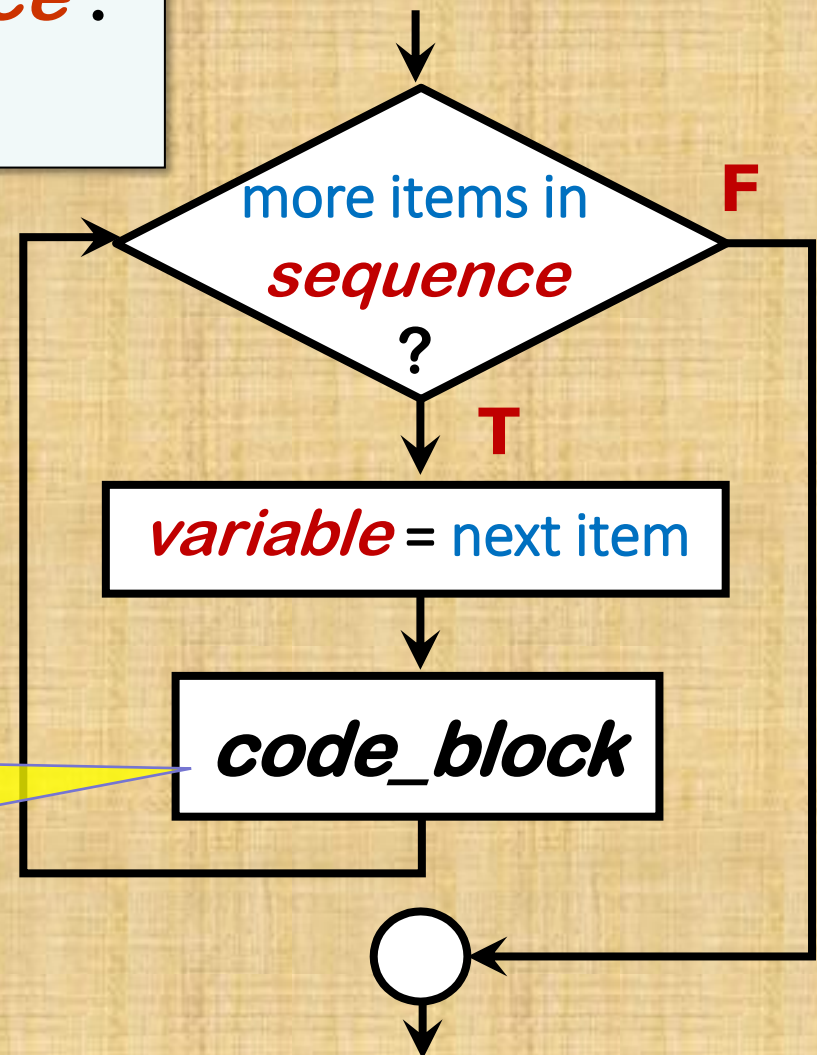
```
for variable in sequence:  
    code_block
```

- *variable* after **for** is called the *loop index*. It takes on each successive value in *sequence* in the order they appear in the sequence.
- The *sequence* can be one of the Python sequence objects such as *a string*, *a list*, *a tuple*, or *a range of integers* from the built-in function **range()**.

How the *for* statement works

```
for variable in sequence:  
    code_block
```

The number of times the *code_block* is executed is precisely the number of items in the *sequence*.



The `range()` function

Python
Syntax

`range(start, stop, step)`

- In its most general form, the `range()` function takes three integer arguments: *start*, *stop*, and *step*.
- `range(start, stop, step)` produces the sequence of integers:
start, *start + step*, *start + 2*step*, *start + 3*step*, ...

If *step* is positive,
the last element is
the largest integer
less than *stop*.

If *step* is negative,
the last element is
the smallest integer
greater than *stop*.

Hands-on Example

To see the type of object `range()`

```
>>> type(range(-4,14,3))  
<class 'range'>
```

This doesn't show the sequence generated by `range()`.

```
>>> range(-4,14,3)  
range(-4, 14, 3)
```

Use the built-in function `list()` to show the sequence.

```
>>> list(range(-4,14,3))  
[-4, -1, 2, 5, 8, 11]
```

Try a negative step.

```
>>> list(range(14,-4,-3))  
[14, 11, 8, 5, 2, -1]
```

produces the empty sequence because *step* is positive and *start* is not less than *stop*.

```
>>> list(range(5,3,1))  
[]
```

produces the empty sequence because *step* is negative and *start* is not greater than *stop*.

```
>>> list(range(3,5,-1))  
[]
```

```
>>>
```

Note that the *stop* value will never appear in the generated sequence.

Hands-on Example

```
>>> list(range(3,8,1))  
[3, 4, 5, 6, 7]  
>>> list(range(3,8))  
[3, 4, 5, 6, 7]  
>>> list(range(0,5))  
[0, 1, 2, 3, 4]  
>>> list(range(5))  
[0, 1, 2, 3, 4]  
>>> for i in range(4):  
    print('i is', i)
```

If *step* is omitted, the *default step* is 1.

If *start* is omitted, the *default start* is 0.

So `range(4)` is the same as `range(0,4)` and `range(0,4,1)`.

```
i is 0  
i is 1  
i is 2  
i is 3
```

The *for* Loop

- In Python a `for`-loop is used to step through a sequence e.g., count through a series of numbers or step through the lines in a file.

- **Syntax:**

*for <name of loop control> in <something that can be iterated>:
 body*

- **Program name:** `for1.py`

```
total = 0
for i in range (1, 4, 1):
    total = total + i
    print("i=", i, "\ttotal=", total)
print("Done!")
```

1) Initialize control

2) Check condition

4) Update control

3) Execute body

The **for** Loop

- In Python a for-loop is used to step through a sequence
- **Syntax:**
*for <name of loop control> in <something that can be iterated>:
 body*
- **Program name:** for1.py

```
i = 0
total = 0
for i in range (1, 4, 1):
    total = total + i
    print("i=", i, "\ttotal=", total)
print("Done!")
```

```
i= 1      total= 1
i= 2      total= 3
i= 3      total= 6
Done!
```

Counting Down With A *for* Loop

- **Program name:** for2.py

```
i = 0
total = 0
for i in range (3, 0, -1):
    total = total + i
    print("i = ", i, "\t total = ", total)
print("Done!")
```

for Loop: Stepping Through A Sequence Of Characters

- Recall: A for-loop in Python can step through any looping sequence (number sequence, characters in a string, lines in a file).
- Example: `for3.py`

```
bath.py * x
1 activity = input("What are you doing with dog now: ")
2 print("We are taking the dog for a '", end="")
3 for ch in activity:
4     print(ch + "-", end="")
5 print("'")
6
7

Shell x
Python 3.7.2 (bundled)
>>> %Run bath.py

What are you doing with dog now: bath
We are taking the dog for a 'b-a-t-h-'
>>> |
```

Erroneous *for* Loops

- The logic of the loop is such that the end condition has already been reached with the start condition.

- **Example:** `for_error.py`

```
for i in range (5, 0, 1):  
    total = total + i  
    print("i = ", i, "\t total = ", total)  
print("Done!")
```


Loop Increments Need Not Be Limited To One

- **While:** while_increment5.py

```
i = 0
while (i <= 100):
    print("i =", i)
    i = i + 5
print("Done!")
```

- **For:** for_increment5.py
- ```
for i in range (0, 105, 5):
 print("i =", i)
print("Done!")
```

```
i = 0
i = 5
i = 10
i = 15
i = 20
i = 25
i = 30
i = 35
i = 40
i = 45
i = 50
i = 55
i = 60
i = 65
i = 70
i = 75
i = 80
i = 85
i = 90
i = 95
i = 100
Done!
```

# Sentinels

- **Sentinel**: special value that marks the end of a sequence of items
  - When program reaches a sentinel, it knows that the end of the sequence of items was reached, and the loop terminates
  - Must be distinctive enough so as not to be mistaken for a regular value in the sequence
  - Example: when reading an input file, empty line can be used as a sentinel

# Illustrative Example #1

average3.py ×

```
1 # average3.py
2 # A program to average a set of numbers
3 # Illustrates sentinel loop using negative input as sentinel
4
5 sum = 0.0
6 count = 0
7 x = int(input("Enter a number (negative to quit) >> "))
8 while x >= 0:
9 sum = sum + x
10 count = count + 1
11 x = int(input("Enter a number (negative to quit) >> "))
12 print ("\nThe average of the numbers is", sum / count)
13
```

Shell ×

```
>>> %Run average3.py
Enter a number (negative to quit) >> 1
Enter a number (negative to quit) >> 2
Enter a number (negative to quit) >> 3
Enter a number (negative to quit) >> -1

The average of the numbers is 2.0
>>> |
```

# Illustrative Example #2

sumsenti.py ×

```
1 # sumsenti.py
2 # A program to get the sum of numbers
3 # Illustrates sentinel loops using <ENTER> key
4 sum = 0.0
5 data = input("Enter a number or just enter to quit: ")
6 while data != "":
7 number = float(data)
8 sum += number
9 data = input("Enter a number or just enter to quit: ")
10 print("The sum is: ", sum)
```

Shell ×

Python 3.7.2 (bundled)

>>> %Run sumsenti.py

Enter a number or just enter to quit: 1

Enter a number or just enter to quit: 2

Enter a number or just enter to quit: 3

Enter a number or just enter to quit: 4

Enter a number or just enter to quit: 5

Enter a number or just enter to quit:

The sum is: 15.0

>>> |

# Sentinel Controlled Loops

- The stopping condition for the loop occurs when the 'sentinel' value is reached.
- **Program name:** sum.py

```
total = 0
temp = 0
while(temp >= 0):
 temp = input ("Enter a non-negative integer (negative to end
 series): ")
 temp = int(temp)
 if (temp >= 0):
 total = total + temp
print("Sum total of the series:", total)
```

```
Enter a positive integer (negative to end series):1
Enter a positive integer (negative to end series):2
Enter a positive integer (negative to end series):3
Enter a positive integer (negative to end series):-1
Sum total of the series: 6
```

Q: What if the user just entered a single negative number?

# Sentinel Controlled Loops

- Sentinel controlled loops are frequently used in conjunction with the error checking of input.
- Example (sentinel value is one of the valid menu selections, repeat while selection is not one of these selections)

```
selection = " "
while selection not in ("a", "A", "r", "R", "m", "M", "q", "Q"):
 print("Menu options")
 print("(a)dd a new player to the game")
 print("(r)emove a player from the game")
 print("(m)odify player")
 print("(q)uit game")
 selection = input("Enter your selection: ")
 if selection not in ("a", "A", "r", "R", "m", "M", "q", "Q"):
 print("Please enter one of 'a', 'r', 'm' or 'q' ")
```

# Recap: What Looping Constructs Are Available In Python/When To Use Them

| Construct                                               | When To Use                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Pre-test loops                                          | You want the stopping condition to be checked before the loop body is executed (typically used when you want a loop to execute zero or more times).                                                                                                                                                                                                                     |
| <ul style="list-style-type: none"><li>• While</li></ul> | <ul style="list-style-type: none"><li>• The most powerful looping construct: you can write a 'while' loop to mimic the behavior of any other type of loop. In general it should be used when you want a pre-test loop which can be used for most any arbitrary stopping condition e.g., execute the loop as long as the user doesn't enter a negative number.</li></ul> |
| <ul style="list-style-type: none"><li>• For</li></ul>   | <ul style="list-style-type: none"><li>• In Python it can be used to step through some sequence</li></ul>                                                                                                                                                                                                                                                                |
| Post-test: None in Python                               | You want to execute the body of the loop before checking the stopping condition (typically used to ensure that the body of the loop will execute at least once). The logic can be simulated with a while loop.                                                                                                                                                          |



# The Break Instruction

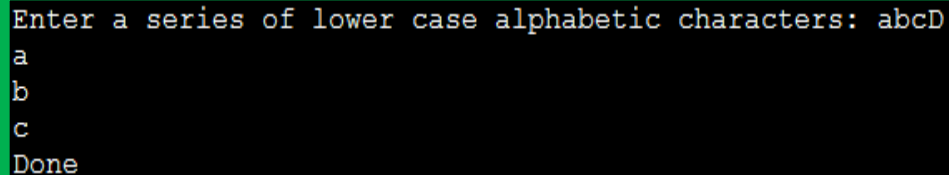
- It is used to terminate the repetition of a loop which is separate from the main Boolean expression (it's another, separate Boolean expression).

- General structure:**

```
for (Condition 1): while (Condition 1):
 if (Condition 2): if (Condition 2):
 break break
```

- Specific example (mostly for illustration purposes at this point): break.py

```
str1 = input("Enter a series of lower case alphabetic characters: ")
for temp in str1:
 if (temp < 'a') or (temp > 'z'):
 break
 print(temp)
print("Done")
```



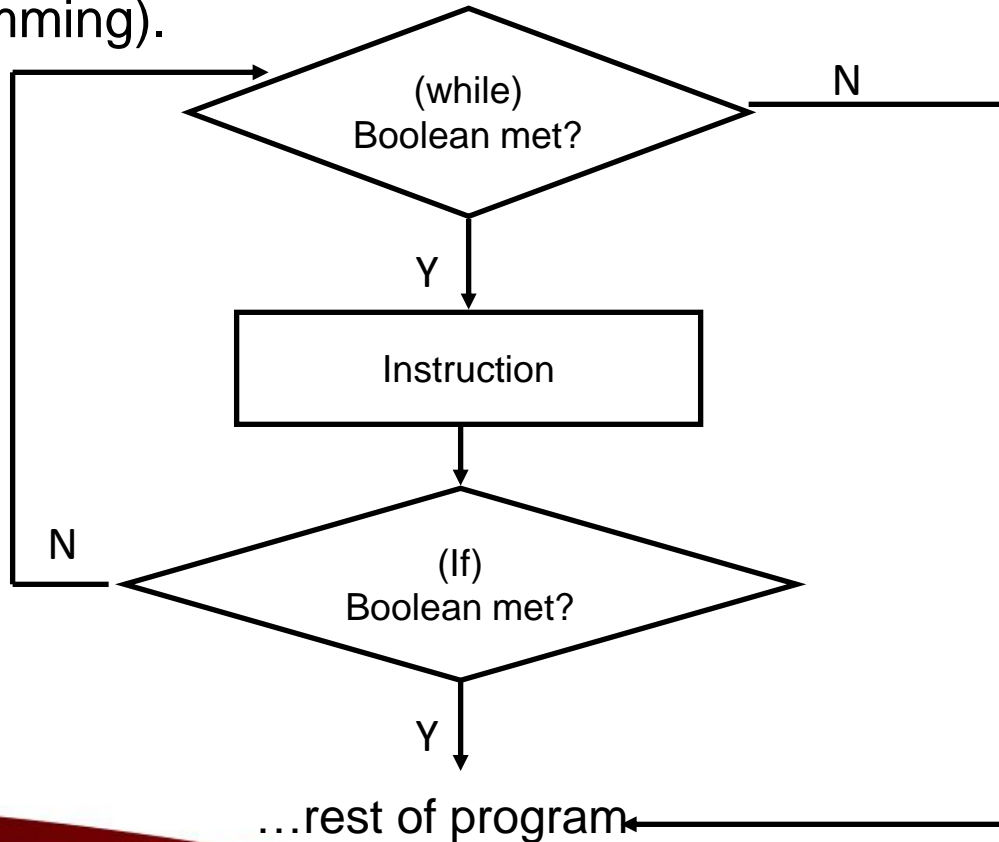
```
Enter a series of lower case alphabetic characters: abcD
a
b
c
Done
```

Q: What if the user just typed 'abc' and hit enter?



# The Break Should Be Rarely Used

- Adding an extra exit point in a loop (aside from the Boolean expression in the while loop) may make it harder to trace execution (leads to 'spaghetti' programming).



**While adding a single break may not always result in 'spaghetti' it's the beginning of a bad habit that may result in difficult to trace programs**

# An Alternate To Using A 'Break'

- Instead of an 'if' and 'break' inside the body of the loop

```
while (BE1):
 if (BE2):
 break
```

- Add the second Boolean expression as part of the loop's main Boolean expression

```
while (BE1) and not (BE2):
```

# Another Alternative To Using A 'Break'

- If the Boolean expressions become too complex consider using a 'flag'

```
flag = true
while (flag == true):
 if (BE1):
 flag == false
 if (BE2)
 flag == false
Otherwise the flag remains set to true
```

- Both of these approaches still provide the advantage of a single exit point from the loop.

# Nested Loops

- One loop executes inside of another loop(s).

- Example structure:

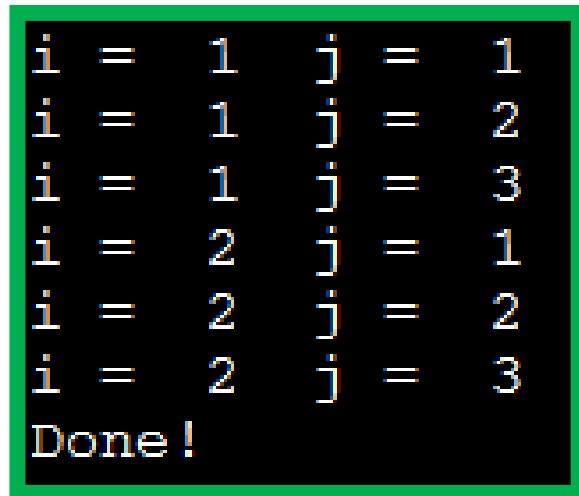
Outer loop (runs n times)

Inner loop (runs m times)

Body of inner loop (runs n x m times)

- Program name: nested.py

```
i = 1
while (i <= 2):
 j = 1
 while (j <= 3):
 print("i = ", i, " j = ", j)
 j = j + 1
 i = i + 1
print("Done!")
```



```
i = 1 j = 1
i = 1 j = 2
i = 1 j = 3
i = 2 j = 1
i = 2 j = 2
i = 2 j = 3
Done!
```

# Nested Loops

Write a Python that will generate a pyramid of stars. The program should ask the user to enter an integer to be use as a basis of generating the pyramid of stars.

Sample output:

Enter an integer: 4

```
*
* *
* * *
* * * *
```

Enter an integer: 6

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

# Prg A

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

# Prg B

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

# Prg C

\*

\*

\*\*                \*\*

\*\*\*            \*\*\*

\*\*\*\*\*       \*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*       \*\*\*\*\*

\*\*\*            \*\*\*

\*\*               \*\*

\*                   \*

# Nested Loops

starpyramid.py ×

```
1 n = int(input("Enter an integer: "))
2 for i in range(0, n):
3
4 # inner loop to handle number of columns
5 # values changing acc. to outer loop
6 for j in range(0, i+1):
7
8 # printing stars
9 print("* ",end="")
10
11 # ending line after each row
12 print("\r")
```

Shell ×

```
Python 3.7.2 (tags/builtins)
>>> %Run starpyramid.py

Enter an integer: 5
*
* *
* * *
* * * *
* * * * *

>>> %Run starpyramid.py

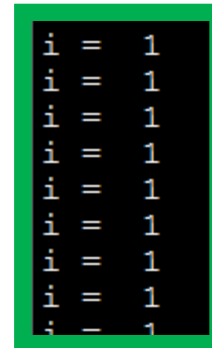
Enter an integer: 4
*
* *
* * *
* * * *
```



# Infinite Loops

- Infinite loops never end (the stopping condition is never met).
- They can be caused by logical errors:
  - The loop control is never updated (Example 1 – below).
  - The updating of the loop control never brings it closer to the stopping condition (Example 2 – next slide).
- **Example 1:** infinite1.py

```
i = 1
while (i <= 10):
 print("i = ", i)
 i = i + 1
```

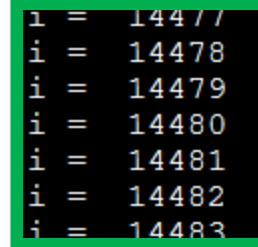


To stop a program with an infinite loop in Thonny simultaneously press the <ctrl> and the <c> keys

# Infinite Loops

- **Example 2:** infinite2.py

```
i = 10
while (i > 0):
 print("i = ", i)
 i = i + 1
print("Done!")
```



```
i = 14477
i = 14478
i = 14479
i = 14480
i = 14481
i = 14482
i = 14483
```

To stop a program with an infinite loop in Unix simultaneously press the <ctrl> and the <c> keys