



# Basic Program Structure

## Repetition (Looping) in Python



**At the end of this lesson, you should be able to:**

- Write programs for executing statements repeatedly using a ***while*** loop in Python
- Describe how to control a loop with sentinel value in Python
- Write loops using ***for*** statements in Python
- Discuss how to generate a sequence of numbers using the ***range()*** function
- Apply ***for*** loop to iterate through a ***range*** sequence
- Write nested loops in Python
- Implement program control with ***break***, ***pass***, and ***continue*** in Python

# Topic Outline



**while** loop

**while-else** statement

**break** statement

**continue** statement

**for** loop

**range** function

**for-else-break-continue**

nested loops

**pass** statement

# Loops in Python

## while

Usually for sentinel-controlled loops  
but may also be used to implement  
counter-controlled loops

```
while <boolean expression>:  
    Suite
```

## Ways to implement loops

## for

Usually for  
counter-controlled loops

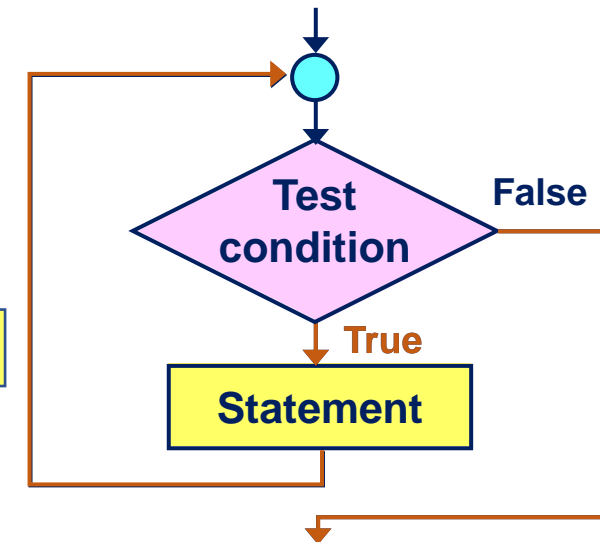
```
for element in collection:  
    Suite
```

# Loops in Python: **while**

- The **while** statement allows repetition a suite of Python codes as long as a condition (Boolean expression) is True.
- It is structurally similar to an **if** statement but repeats the block until the condition becomes False.

```
while <boolean expression>:
```

```
    Suite (one or more indented statements)
```



- When the condition becomes False, repetition ends and control moves on to the code following the repetition.



# Loops in Python: **while** - Syntax

The whole  
**While** structure

```
while <boolean expression>:
    Suite (one or more indented statements)
```

indentation

It **must** use a **colon**  
followed by a proper  
**indentation**

# Loops in Python: **while** - Syntax (Cont'd)

## Example

```
# simple while

x = 0 # initialize #loop-control variable

# test loop-control variable at beginning of loop
while x < 10:
    print(x) # print the value of x int each time through the while loop
    print()
print("Final value of x int: ", x) # bigger than value printed in loop
```



Output:

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

Final value of x int: 10

The whole **while** structure



# Loops in Python: **while** - Syntax (Cont'd)

## More on `print()`

```
x = 0
while x < 10:
    print(x, end = ' ')
    x = x + 1
print()
print("Final value of x int: ", x)
```

Output:

0 1 2 3 4 5 6 7 8 9

Final value of x int: 10



Separate the numbers using @?

```
print(x, end = '@')
```

- The `end = ' '` in the print statement indicates that the print ends with an empty string rather than the default new line.
- This means that the output from multiple calls to print will occur on the same output line.

Output

0@1@2@3@4@5@6@7@8@9

Final value of x int: 10



What is the output of the following code?

```
count = 5
while count < 9:
    count = count + 1
print("count= ", count, end=" ")
```



# Quick Check: Answer



What is the output of the following code?

```
count = 5
while count < 9:
    count = count + 1
print("count= ", count, end=" ")
```

Answer

count = 9



What is the output of the following code?

```
count = 0
number = 9

while number > 0:
    if number % 2 == 0:
        number //=2
    elif number % 3 ==0:
        number //=3
    else:
        number -=1
    count = count + 1

print ('count is:', count, 'number is:', number)
```

# Quick Check: Answer



What is the output of the following code?

```
count = 0
number = 9

while number > 0:
    if number % 2 == 0:
        number //=2
    elif number % 3 ==0:
        number //=3
    else:
        number -=1
    count = count + 1

print ('count is:', count, 'number is:', number)
```

Answer

count is: 3 number is: 0



If **line 1** and **line 2** were removed, what effect would that have on the program?

```
count = 0
number = 9

while number > 0:
    if number % 2 == 0:
        number //=2
    elif number % 3 ==0:
        number //=3
    else:                                #line 1
        number -=1                      #line 2
    count = count + 1

print ('count is:', count, 'number is:', number)
```

# Quick Check: Answer



If **line 1** and **line 2** were removed, what effect would that have on the program?

```
count = 0
number = 9

while number > 0:
    if number % 2 == 0:
        number //=2
    elif number % 3 ==0:
        number //=3

    count = count + 1
print ('count is:', count, 'number is:', number)
```

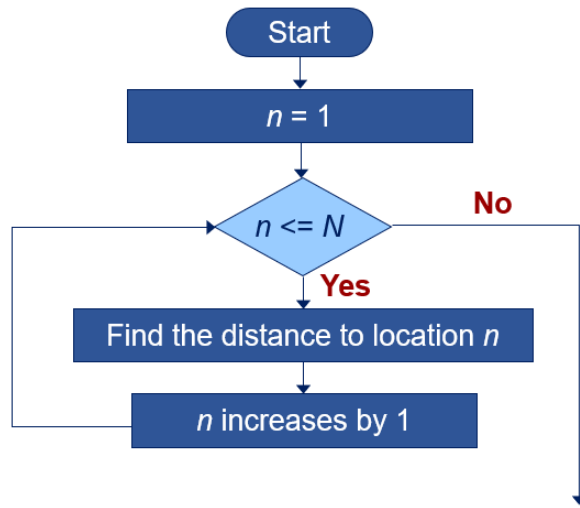
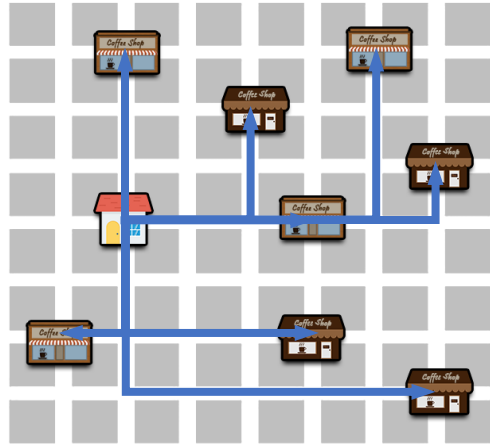


Answer

The **while** loop  
will not end.



# Application of Looping: Find the Distance to N Locations



## Pseudocode

```
Count = 0
WHILE count < N
    Read horizonDist
    Read vertDist
    dist = horizonDist + vertDist
    Display distance traveled between two points
    Increase count by 1
END WHILE
```

# Find the Distance to N Locations: Loop Design Strategies

## Step 1 Identify the statements that need to be repeated

```
horizonDist = int(input("Read horizonDist in meters"))  
vertDist = int(input("Read vertDist in meters"))  
dist = horizonDist + vertDist  
print("dist from home to coffee shop ", count, " is ", dist, "m")
```



# Find the Distance to N Locations: Loop Design Strategies

(Cont'd)

**Step 2** Wrap these statements in a loop

```
while (true) {  
  Statements;  
}
```

**while True:** ← colon



```
horizonDist = int(input("Read horizonDist in meters"))  
vertDist = int(input("Read vertDist in meters"))  
dist = horizonDist + vertDist  
print("dist from home to coffee shop ", count, " is ", dist, "m")
```

indentation

# Find the Distance to N Locations: Loop Design Strategies

(Cont'd)

## Step 3

Code the loop-continuation-condition and add appropriate statements for controlling loop

User-friendly message  
indicating index of current input/  
loop control variable

```
N = int(input("How many coffee shops?"))
count = 1
while count <= N:
    print("input information of coffee shop ", count)
    horizonDist = int(input("Read horizonDist in meters"))
    vertDist = int(input("Read vertDist in meters"))
    dist = horizonDist + vertDist
    print("dist from home to coffee shop ", count, " is ", dist, "m")
    count = count + 1
```

Initialize the loop control variable

Loop-continuation-condition

Update

# Find the Distance to N Locations: Check Loop Iteration

```

N = int(input("How many coffee shops?"))
count = 1
while count <= N:
    print ("input information of coffee shop ", count)
    horizonDist = int(input("Read horizonDist in meters"))
    vertDist = int(input("Read vertDist in meters"))
    dist = horizonDist + vertDist
    print("dist from home to coffee shop ", count, " is ", dist, "m")
    count = count + 1
print ("Thank you for using the application")

```

Check Loop Iteration: Table (N = 2)

Count	Count <= N	Output
1	True	Dist to shop 1
2	True	Dist to shop 2
3	False	

## In the loop:

- What is the loop control variable? → **count**
- Counter-controlled** or sentinel-controlled?

# Outputs of Sample Runs

## Output (Program instructions: 2 times)

How many coffee shops? 2  
input information of coffee shop 1  
Read horizonDist in meters 332  
Read vertDist in meters 432  
dist from home to coffee shop 1 is 764 m  
input information of coffee shop 2  
Read horizonDist in meters 123  
Read vertDist in meters 256  
dist from home to coffee shop 2 is 379 m  
Thank you for using the application

## Output (Program instructions: 1 time)

How many coffee shops? 1  
input information of coffee shop 1  
Read horizonDist in meters 55  
Read vertDist in meters 778  
dist from home to coffee shop 1 is 833 m  
Thank you for using the application

## Output (Program instructions: 0 time)

How many coffee shops? -2  
Thank you for using the application

# Potential Issues in the Program

```
N = int(input("How many coffee shops?"))
count = 1
while count <= N:
    print ("input information of coffee shop ", count)
    horizonDist = int(input("Read horizonDist in meters"))
    vertDist = int(input("Read vertDist in meters"))
    dist = horizonDist + vertDist
    print("dist from home to coffee shop ", count, " is ", dist, "m")
    count = count + 1
print ("Thank you for using the application")
```



How many coffee shops? -2  
Thank you for using the application

Something to  
think about:

- Any potential issues in the program?
- If the user enters a value smaller than 1, what would happen?
- How about if the user enters '999999999999'?





# Potential Issues in the Program: Solution

Force the user to input again if the input is out of a reasonable range.



**HOW?**

Keep asking until the user enters a number that is at least 1 and at most a certain reasonable limit.



Add a **while** loop

# Implementation of the Solution (New While Loop)

New  
**while**

```
N = int(input("How many coffee shops?"))
while (N < 1 or N > 10):
    print("warning: input value should range from 1 to 10");
    N = int(input("How many coffee shops?"))
count = 1
while count <= N:
    print("input information of coffee shop ", count)
    horizonDist = int(input("Read horizonDist in meters"))
    vertDist = int(input("Read vertDist in meters"))
    dist = horizonDist + vertDist
    print("dist from home to coffee shop ", count, " is ", dist, "m")
    count = count + 1
print("Thank you for using the application")
```

() is okay  
but  
redundant

## In the loop:

- What is the loop control variable? → **N**
- Counter-controlled or **sentinel-controlled**?



**while** loop is useful for the purpose of **repeatedly** asking the user for an input until the input fulfills the requirement

# Another Example of Sentinel-Controlled Loop

Write a program to read numbers from a user; sum them up until the input is -1.

```
=====
>>>
Enter value: 2
Enter value: 1.2
Enter value: 4
Enter value: 10
Enter value: -1
total= 17.2
>>>
```

Indicates the end



Note: -1 is called a sentinel value.

# Common Sentinel Loop Implementation

```
value = some value
while value != sentinel value:
    # process value
    # get another value
```

**while**  
structure {

```
total = 0.0
item = float( input("Enter value: ") )
while item != -1:
    total += item
    item = float( input("Enter value: ") )
print("total=" , total )
```



Make sure  
**sentinel value**  
will not appear  
in normal inputs.



Do you foresee any issue to ensure for this design?

# Loops in Python: **while-else** (Syntax)

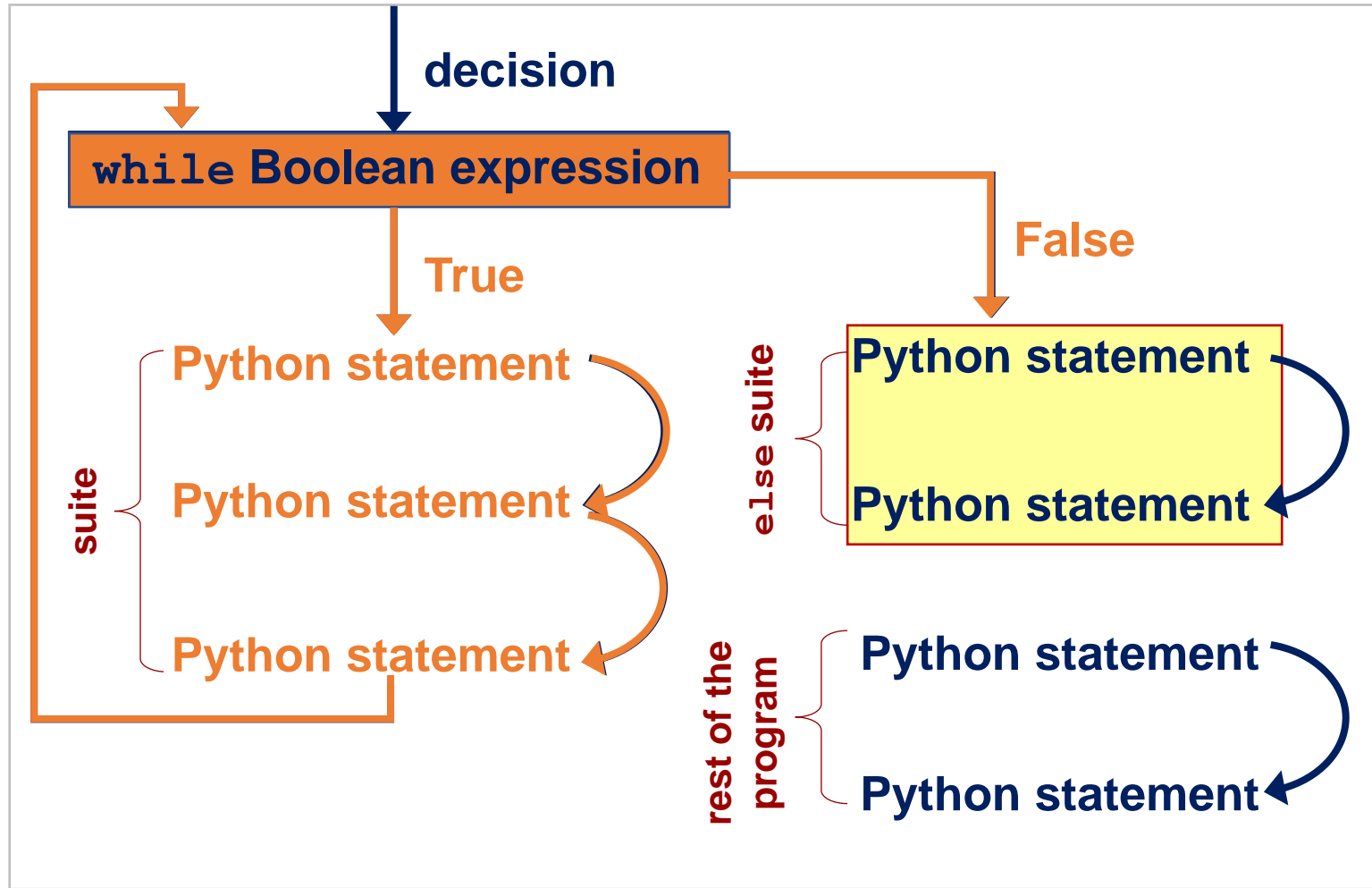
The whole  
**while** structure

```
while boolean expression:  
    handle_true()  
else:  
    handle_false() ←  
rest of the programme
```

Condition is false now,  
handle and go on with  
the rest of the program

- **while** loop, can have an associated **else** statement
- **else** statement is executed when the loop finishes under normal conditions
  - the last thing the loop does as it exits

# Loops in Python: **while-else** (Syntax)



- It is entered after the **while** loop's Boolean expression becomes False.
- This entry occurs even when the expression is initially False and the **while** loop has never run.
- A handy way to perform some final tasks when the loop ends normally.

# while-else: Example

```
total = 0.0
count = 0
item = float(input("Enter value:"))
while item != -1:
    total += item
    count += 1
    item = float(input("Enter value:"))
else:
    print (count, "numbers read.", "Total= ", total),
```



Enter value:-1  
0 numbers read. Total= 0

Enter value:55  
Enter value:21  
Enter value:1  
Enter value:-1  
3 numbers read. Total= 77.0





What is the output of the following code?

```
value = 1
print ("before while", value)

while value <=3:
    value = value + 1
    print ("while", value)
else:
    print ("else", value)
```



# Quick Check: Answer



What is the output of the following code?

```
value = 1
print ("before while", value)

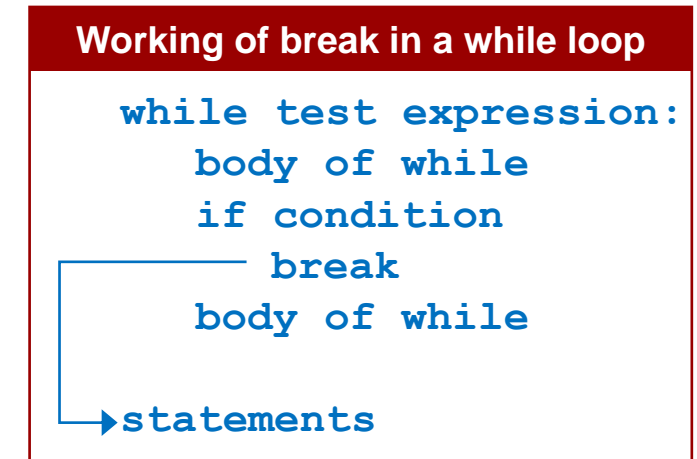
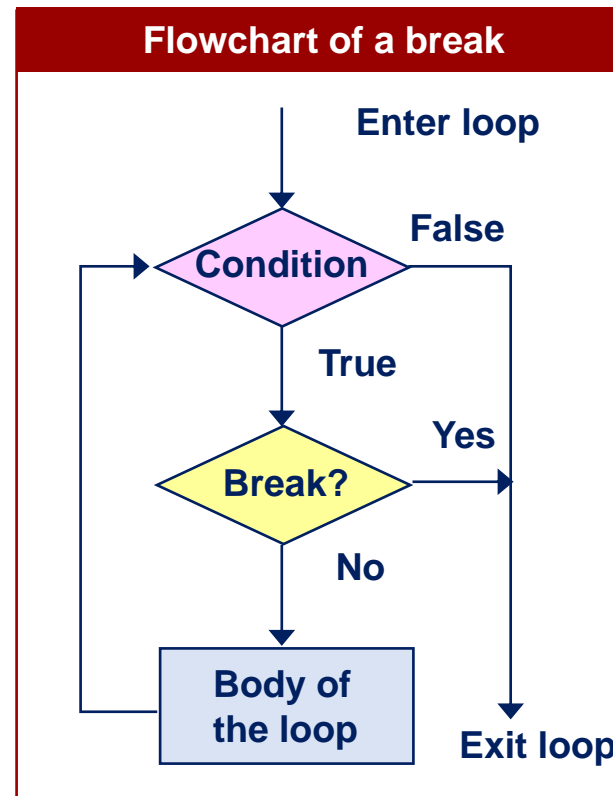
while value <=3:
    value = value + 1
    print ("while", value)
else:
    print ("else", value)
```

Answer

before while 1  
while 2  
while 3  
while 4  
else 4

# while-else: Break Statement

- The **break** statement can be used to **immediately** exit the execution of the **current** loop and skip past all the remaining parts of the loop suite.
  - It is important to note that “skip past” means to skip the **else** suite (if it exists) as well.
- The **break** statement is useful for stopping computation when the “answer” has been found or when continuing the computation is otherwise useless.





What do you think is the output of the following code?

```
value = 1
print ("before while", value)

while value <=3:
    value = value + 1
    print ("while", value)
    if value == 2:
        break;
else:
    print ("else", value)
print ("after while", value)
```



Compared to quick check question:  
added **if** and **break**.

- Executing break exits the immediate loop that contains it.
- It goes after the whole enclosing loop.

# Quick Try: Answer



What do you think is the output of the following code?

```
value = 1
print ("before while", value)

while value <=3:
    value = value + 1
    print ("while", value)
    if value == 2:
        break;
else:
    print ("else", value)

print ("after while", value)
```

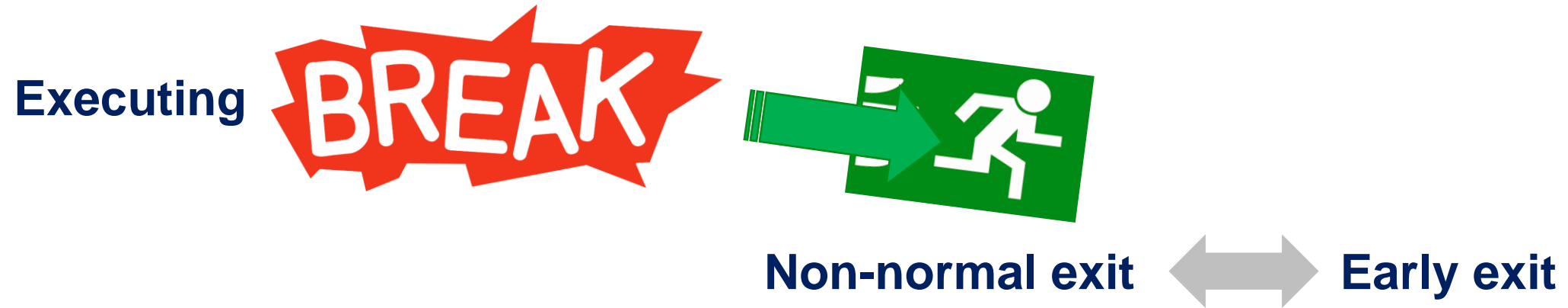


Answer

before while 1  
while 2  
after while 2

No more:  
while 3  
while 4  
else 4

# **while-else:** Break Statement (Cont'd)



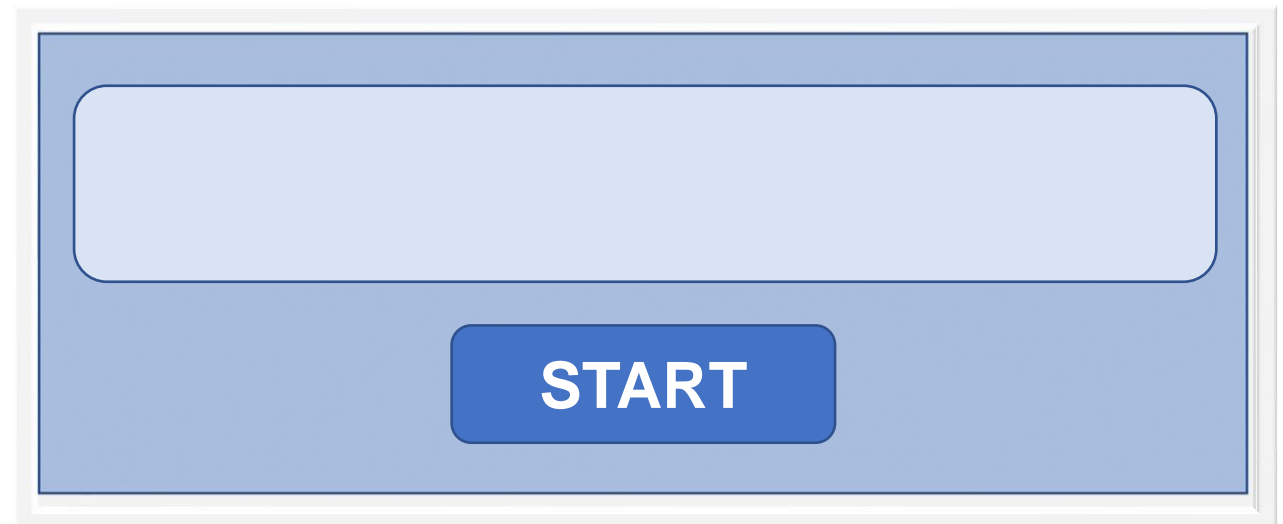
# while-else: Break Statement – Example

## “Hi-Low” Number Guessing Game

The program starts by generating a random number hidden from the user that is between 0 and 100. The user then attempts to **guess the number**, getting **hints** as to which direction (bigger or smaller, higher or lower) to go with the next guess.

**The game can end in one of two ways:**

- The correctly guessed the number.
- The user can quit playing by entering a number out of the range of 0 - 100.





# while-else: Break Statement – Example (Cont'd)

## “Hi-Low” Number Guessing Game

The program starts by generating a random number hidden from the user that is between 0 and 100. The user then attempts to **guess the number**, getting **hints** as to which direction (bigger or smaller, higher or lower) to go with the next guess.

**The game can end in one of two ways:**

- The user correctly guesses the number.
- The user can quit playing by entering a number out of the range of 0 - 100.

### ALGORITHM

Choose a random number.

Prompt for a guess.

While the guess is in range:

    Check whether the guess is correct;

        If it is a win, print a win message and exit.

        Otherwise, provide a hint and prompt for a guess.

Else:

    User quits



# while-else: Example 2 Python Code

```

Number = 67
print("Hi-Lo Number Guessing Game: between 0 and 100 inclusive.")
guess= (int)(input("Guess a number: "))
while 0 <= guess <= 100:
    if guess > Number:
        print("Guessed Too High.")
    elif guess < Number:
        print("Guessed Too Low.")
    else: # correct guess, exit with break
        print("You guessed it. The number was:", Number)
        break
    # keep going, get the next guess
    guess= (int)(input("Guess a number: "))
else:
    print("A pity. You quit the game early, the number was:", Number)

```

The else clause is often used in conjunction with the break statement.

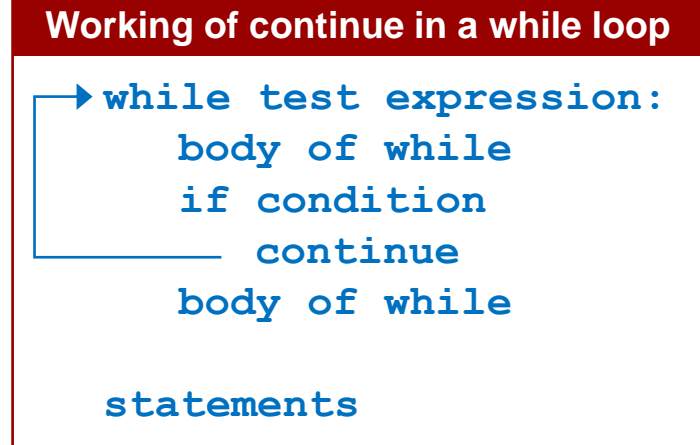
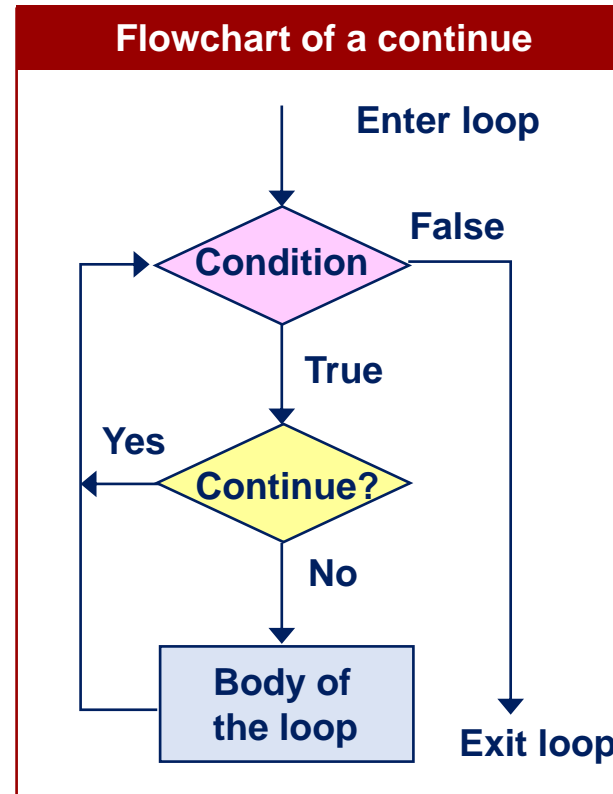
# while-else: Example 2 Python Code (Cont'd)

```
Number = 67
print("Hi-Lo Number Guessing Game: between 0 and 100 inclusive.")
guess= (int)(input("Guess a number: "))
while 0 <= guess <= 100:
    if guess > Number:
        print("Guessed Too High.")
    elif guess < Number:
        print("Guessed Too Low.")
    else: # correct guess, exit with break
        print("You guessed it. The number was:",Number)
        break
    # keep going, get the next guess
    guess= (int)(input("Guess a number: "))
else:
    print("A pity. You quit the game early, the number was:",Number)
```

Hi-Lo Number Guessing Game: between 0 and 100 inclusive.  
 Guess a number: 55  
 Guessed Too Low.  
 Guess a number: 88  
 Guessed Too High.  
 Guess a number: 234  
 A pity. You quit the game early, the number was: 67

# while-else: Continue Statement

- Skip some portion of the **while** suite we are executing and have control flow back to the beginning of the **while** loop.
- Exit early from this iteration of the loop (not the loop itself), and keep executing the **while** loop.
- The **continue** statement continues with the next iteration of the loop.



# while-else: Continue Statement - Sample Problem

## Algorithm

Prompt the user for a number.

Convert the input string to an int.

If the input is even, add it to the running sum.

If the input is not even (odd), print an error message – don't add it to the sum, just continue on.

If the input is the special character ".", end and print the final sum.

# while-else: Continue Statement – Solution to the Problem

```
# sum up a series of even numbers
# make sure only even numbers contribute to sum

print ("Allow the user to enter a series of even integers. Sum them.")
print ("Ignore non-even input. End input with a '. '")

# initialize the input number and the sum
number_str = input("Number: ")
the_sum = 0

# Stop if a period ( . ) is entered .
# remember , number_str is a string until we convert it
while number_str != ". " :
    number = int(number_str)
    if number % 2 == 1: # number is not even ( it is odd)
        print ("Error, only even numbers please.")
        number_str = input("Number: ")
        continue # if the number is not even , ignore it
    the_sum += number
    number_str = input("Number: ")
print ("The sum is:",the_sum)
```



**Control flow knowledge:**  
Go immediately to test  
in the enclosing loop

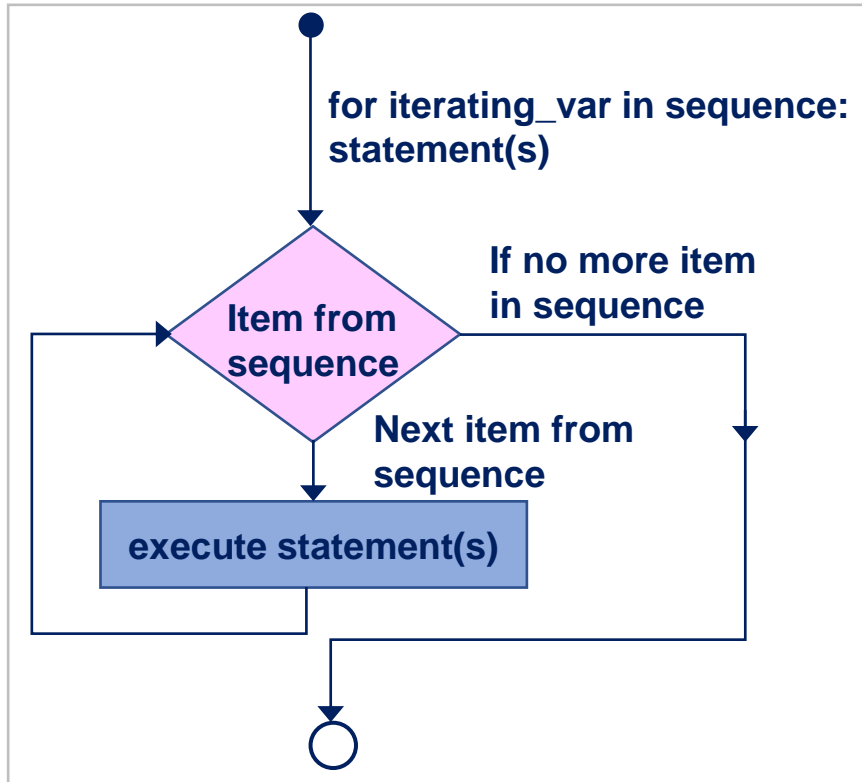
➔ **Output:**

Allow the user to enter a series of even integers. Sum them.  
Ignore non-even input. End input with a '. '  
Number: 7  
Error, only even numbers please.  
Number: 8  
Number: 9  
Error, only even numbers please.  
Number: 4  
Number: .  
The sum is: 12

For **continue**, make sure loop control variable can be updated. Else, **infinite loop**.

## Syntax

```
for iterating_var in sequence:  
    Suite (one or more indented statements)
```



- It has the ability to iterate over the items of any sequence, such as a list or a string.
- Each item in the sequence is assigned to the iterating variable `iterating_var`. Next, the `statement(s)` block is executed.
- The **for** loop completes when the last of the elements has been assigned to the `iterating_var`, i.e., the entire sequence is exhausted.

# Loops in Python: **for** - Syntax

```
for iterating_var in <sequence>:  
    Suite (one or more indented statements)
```

indentation

It must use a **colon** followed by proper **indentation**.

- Has a header and an associated suite.
- Keywords: **for** and **in**.
- The keyword **in** precedes the sequence.
- The variable `iterating_var` is a variable associated with the **for** loop that is assigned the value of an element in the sequence.
  - The variable `iterating_var` is assigned a different element during each pass of the **for** loop.
  - Eventually, `iterating_var` will be assigned to each element in the sequence.



# Loops in Python: **for** - Example

Two **for** structures

It can also be a list

```
for the_char in 'thinking':
    print("current char:", the_char)

for the_drink in ["coffee", "tea", "milo"]:
    print("current drink:", the_drink)

print("enjoy")
```



**Output**

```
current char: t
current char: h
current char: i
current char: n
current char: k
current char: i
current char: n
current char: g
current drink: coffee
current drink: tea
current drink: milo
enjoy
```



Write a Python program to find the sum of the numbers from 1 to 10 using **for** loop.

# Quick Check: Answer



Write a Python program to find the sum of the numbers from 1 to 10 using **for** loop.

Answer

```
sum = 0

for i in [1,2,3,4,5,6,7,8,9,10]:
    print("current i is :", i)
    sum += i

print ("the sum of the numbers from 1 to 10 is ", sum)
```



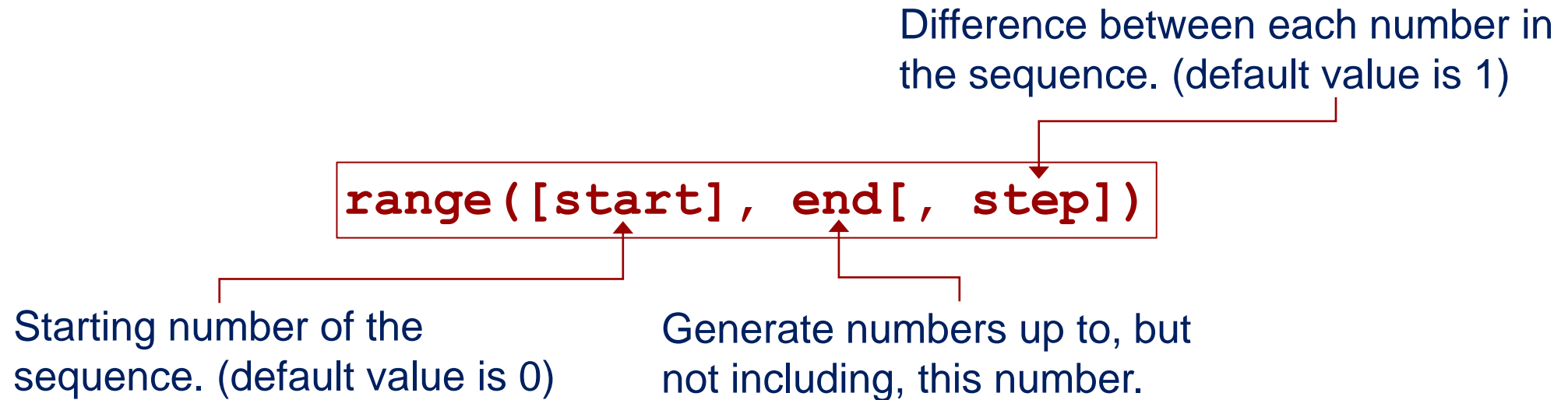
Output

```
current i is : 1
current i is : 2
current i is : 3
current i is : 4
current i is : 5
current i is : 6
current i is : 7
current i is : 8
current i is : 9
current i is : 10
the sum of the numbers from 1 to 10 is 55
```

# Python: The Range Function

## `range()`

- It is a useful built-in function in Python.
- It generates a list of **integers** from **start** up to **end** (but excluding **end**) with step-size **step**.



Note: All parameters must be integers, positive or negative.

# Python: The Range Function - Variance

## Syntax Three different Forms:

### `range(end)`

- Python puts **start** to 0 and **step** to 1.
- **Example:** `range(11)`  $\Rightarrow$  [0,1, 2, 3,..., 10]

### `range(start, end)`

- Python puts **step** to 1 (default value).
- **Example:** `range(1, 11)`  $\Rightarrow$  [1, 2, 3,..., 10]

### `range(start, end, step)`

#### Examples:

`range(1, 11, 2)`  $\Rightarrow$  [1, 3, 5, 7, 9]

`range(11, 2, -3)`  $\Rightarrow$  [11, 8, 5]



What is the output of the following code?

```
sum = 0

for number in range(1,10):
    sum = sum + number

print (sum)
```



# Quick Check: Answer



What is the output of the following code?

```
sum = 0

for number in range(1,10):
    sum = sum + number

print (sum)
```

Answer

45



Note: The range in Python excludes the ending element.



What is the output of the following code?

```
sum1 = sum2 = 0

for number in range(1,5,1):
    sum1 += number
for number in range(5,1,-1):
    sum2 += number

print ("sum1:", sum1, "sum2:", sum2)
```



# Quick Check: Answer



What is the output of the following code?

```
sum1 = sum2 = 0

for number in range(1, 5, 1): → [1, 2, 3, 4]
    sum1 += number
for number in range(5, 1, -1): → [5, 4, 3, 2]
    sum2 += number

print ("sum1:", sum1, "sum2:", sum2)
```



Note:

- The sequence is not symmetric.  
 $\text{range}(1, 5, 1) \neq \text{range}(5, 1, -1)$
- Variable number in the example “**for**” loops will take different values in different loop iterations.

Answer

sum1: 10 sum2: 14

# range() : Problem Example

## Problem: Print a Multiplication Table of a Certain Number (Multiplier)

**Example: 9** (multiplicands: from 1 to 10) 

- First, ask the user for a number.
- Then loop from 1 to 10, compute the multiplication, and display the formulae.

=====

>>>

Enter a number between 1 to 9: 9

1 × 9 = 9

2 × 9 = 18

3 × 9 = 27

4 × 9 = 36

5 × 9 = 45

6 × 9 = 54

7 × 9 = 63

8 × 9 = 72

9 × 9 = 81

10 × 9 = 90

# range() : Problem Example - Implementation in Python

In Python  $\Rightarrow$  just a few lines

```
num = int(input("Enter a number between 1 to 9: "))  
  
for x in range (1, 11):  
    print (x , 'x' , num , '=' , x*num)
```



Use comma to separate  
different elements for **print**.

End with 11 instead of 10 if we  
want the last value to be 10.

# for-else-break-continue

```
for target in object:
```

```
    # statement suite1
```

```
    if boolean expression1:
```

```
        break # Exit loop now; skip else
```

```
    if boolean expression2:
```

```
        continue # Go to top of loop now
```

```
else:
```

```
    # statement suite2
```

## Working of break in a for loop

```
for var in sequence:
    body of for
    if condition
        break
    body of for
statements
```

## Working of continue in a while loop

```
while test expression:
    body of while
    if condition
        continue
    body of while
statements
```



What is the output of the following code?

```
for letter in "Python":  
    if letter == 'h':  
        break;  
print (letter, end = '')
```



# Quick Check: Answer



What is the output of the following code?

```
for letter in "Python":  
    if letter == 'h':  
        break;  
print (letter, end = '')
```



Answer

Pyt



What is the output of the following code?

```
for letter in "Python":  
    if letter == 'h':  
        continue;  
    print (letter, end = ' ')
```



# Quick Check: Answer



What is the output of the following code?

```
for letter in "Python":  
    if letter == 'h':  
        continue;  
    print (letter, end = '')
```

Answer

Pyton



# for-else-break: Computational Thinking Example

## Prime or Non-Prime

Given a number  $k$ , how to determine if it is a prime?

- A prime number does not have other divisors except 1 and itself
- Find out whether a number is prime:
  1. Divide it by numbers smaller than it.
  2. If any of them have a zero remainder, then the number is not prime.

This is a job is for a **for** loop.

Logic

```
isprime = True
for n in range (1, k):
    if k%n == 0:
        isprime = False
```

The for loop ends at  $k/2$ .

**Fact:** It is not possible for any number larger than  $k/2$  to divide evenly into  $k$ .

Reason

Is there a need to divide it by all the numbers smaller than it? → **No**

# for-else-break: Computational Thinking Example

```
k = int(input("Please enter an integer: "))
# Divide k by all numbers 2<= n <= k/2
for n in range (2, int(k/2)+1):
    # If the remainder is 0 then n is a factor of k
    if k%n == 0:
        isprime = "no" # K is not a prime
        break #exit loop
    else:
        isprime = "yes" # Loop not exited: it is prime
print(k, "is a prime? ", isprime)
```

Please enter an integer: 31  
31 is a prime? yes

Please enter an integer: 32  
32 is a prime? no

## **Nested Loop** a loop inside another loop

- Just as it is possible to have if statements nested within other if statements, a loop may appear inside another loop.
  - An outer loop may enclose an inner loop.

### **What can be nested?**

- Nest as many levels of loops as the system allows.
- Nest different types of loops.

# Nested Loops: Example 1

```
for y in range (1,4):#outer loop
    for x in range (1,6):#inner loop
        print('(', x, ', ', y, ')', end = ' ')#inner loop body
    #end inner loop
print()
#end outer loop
```

This is the entire suite/ block inside the outer **for** loop.

Only one statement inside the inner **for** loop.

This additional argument is used in Python 3 to avoid the default ending `\n`.

( 1 , 1 )	( 2 , 1 )	( 3 , 1 )	( 4 , 1 )	( 5 , 1 )
( 1 , 2 )	( 2 , 2 )	( 3 , 2 )	( 4 , 2 )	( 5 , 2 )
( 1 , 3 )	( 2 , 3 )	( 3 , 3 )	( 4 , 3 )	( 5 , 3 )



- How many times each **print** is executed?
- What is the control flow?

# Nested Loops: Example 2 - Printing Full Multiplication Table

```
m = int (input ("Enter a number between 1 to 9: ") )
n = int (input ("Enter another number between 1 to 9: "))
for x in range (1,m+1):
    for y in range (1,n+1):
        print (x, 'x', y, '=', x*y, end = ' ')
    print()
```



Enter a number between 1 to 9: 3


Enter another number between 1 to 9: 4

1 x 1 = 1 1 x 2 = 2 1 x 3 = 3 1 x 4 = 4

2 x 1 = 2 2 x 2 = 4 2 x 3 = 6 2 x 4 = 8


3 x 1 = 3 3 x 2 = 6 3 x 3 = 9 3 x 4 = 12

# In-Depth Nested Loops

- Loops can be nested to a greater depth if necessary, and **while** and **for** loops can be nested interchangeably.
- Is it ever useful to do this?  **Yes, very often.**

## Application

**Test multiple numbers whether they are prime or not.**

- How many numbers to check?  **Unknown**
- User enters '#' to end the checking.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

# Test Multiple Numbers - Prime or Not: Implementation

```

k_str = input("Please enter an integer: ")
while k_str != '#':
    k = int(k_str)
    # Divide k by all numbers 2<= n <= k/2
    for n in range (2, int(k/2)+1):
        # If the remainder is 0 then n is a factor of k
        if k%n == 0:
            isprime = "no" # k is not a prime
            break #exit for loop
        else:
            isprime = "yes" # loop not exited: it is prime
    print(k, "is a prime? ", isprime)
    k_str = input("Please enter another integer: ")

```



→ Output of a sample run

```

Please enter an integer: 31
31 is a prime? yes
Please enter another integer: 32
32 is a prime? no
Please enter another integer: 13
13 is a prime? yes
Please enter another integer: #

```



Note: **break** affects only that inner loop that contains it.

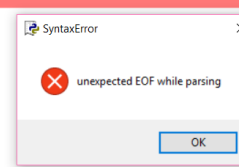
# Interesting Pass Statement

- It has no effect (does nothing) but helps in **indicating** an empty statement/ suite/ block.
- Use when you have to put something in a statement (syntactically, you cannot leave it blank or Python will complain) but what you really want is nothing.

Python has the syntactical requirement that code blocks (after **if**, **for**, **while**, **except**, **def**, **class**, etc.) cannot be empty.

## Example

```
for x in range (1,10):
```




```
for x in range (1,10):  
    pass
```

## Some uses of pass

- It can be used to **test a statement** (say, opening a file or iterating through a collection) just to see if it works.
- You can use **pass** as a **place holder**.





What is the output of the following code?

```
for letter in "Python":  
    if letter == 'h':  
        pass;  
    print (letter, end = ' ')
```



# Quick Check: Answer



What is the output of the following code?

```
for letter in "Python":  
    if letter == 'h':  
        pass;  
    print (letter, end = ' ')
```



Answer

Python

