UNIT: II	Decision Control Statements	Lectures: 08 Hrs

- **Decision Control Statements:** Decision control statements
- **Selection/conditional branching Statements:** if, if-else, nested if, if-elif-else statements .
- **Basic loop Structures/Iterative statements:** while loop, for loop, selecting appropriate loop. Nested loops, The *break, continue, pass, else statement used with loops.*

Decision Control Statements

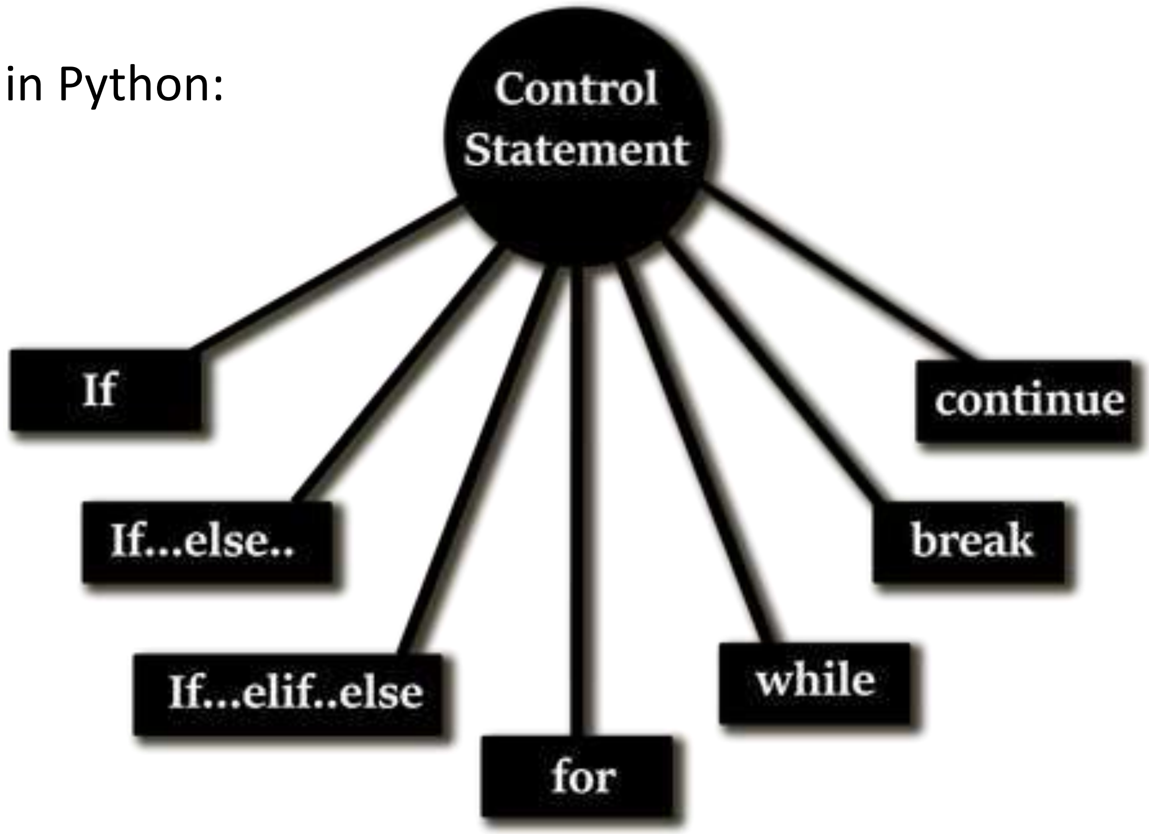
- Decision making statements in programming languages decides the direction of flow of program execution.
- In our daily life we do many things which depends on some kind of conditions for **e.g.** If I study I will pass exams. If he is hungry he will eat etc.
- Sometimes in the program, we need the statement to execute under some condition like **if the value is equal to this**, then this will happen and **if not equal to this** then this will happen.
- This can be achieved in Python by using the Decision Control statements in python.

Decision Control Statements

- We have the **two** types of Control statement in Python:
- **Decision Control Statement:**

  If, If..else, if…elif….else.
- **Flow Control Statement:**

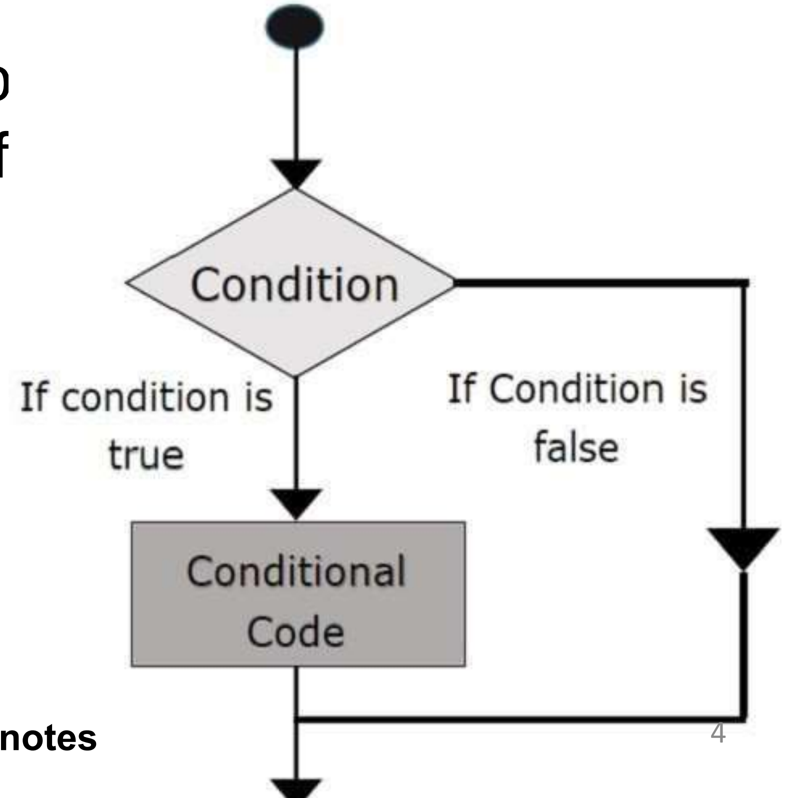
  for, while break, continue



IF Statement

The **If** the statement is similar to other languages like in Java, C, C++, etc. It is used when we have to take some decision like the comparison between anything or to check the presence and gives us either TRUE or FALSE.
The if statement tells your script, “If this Boolean expression is True, then run the code under it, otherwise skip it.”
A colon at the end of a line is how you tell Python yo
to create a new “block” of code, and then indenting f
tells Python what lines of code are in that block.

The **syntax** for **IF** Statement is as:
if (expression):
# code to be executed under the condition
statement1
statement2



IF Statement Example

```
var1 = 10

var2 = 12

if var2 > var1:        #here it return the TRUE value hence the if block is executed

        print( var2,'is greater then', var1)

if var1 > var2:            #here it return the FALSE value hence the if block is not executed

        print(var1,'is smaller than', var2)
```

IF...else Statement

The **else** statement can be used with the **if** statement. It usually contains the code which is to be executed at the time when the expression in the **if** statement returns the FALSE. There can only be one **else** in the program with every single **if** statement

It is optional to use the **else** statement with **if** statement it depends on your condition.

The **syntax** for **If….Else** Statement is as:

```
If(expression):
        statement  #body of if/ if block
        Statement
else:
        statement #body of else/ else block
        statement
```

IF...else Statement Example

```
var1 = 10
var2 = 12
if var2 > var1:
        print( var2,'is greater then', var1)
else:                   #it is not executed because the if expression is true
        print(var2,'is samller then', var1)

if var1 > var2:
        print(var1,'is greater than', var2)
else:                   #it is executed because if expresion is false
        print(var1, 'is samller then', var2)
```

The Elif Statement

The **elif** statement in the Python is used to check the multiple expression for TRUE and execute a block of code as soon as one of the conditions returns to TRUE.
elif – is a keyword used in Python in replacement of else if to place another condition in the program. This is called chained conditional. Chained conditions allows than two possibilities and need more than two branches.

The **syntax** for **elif** Statement is as:

if expression: *#body to be executed in if*

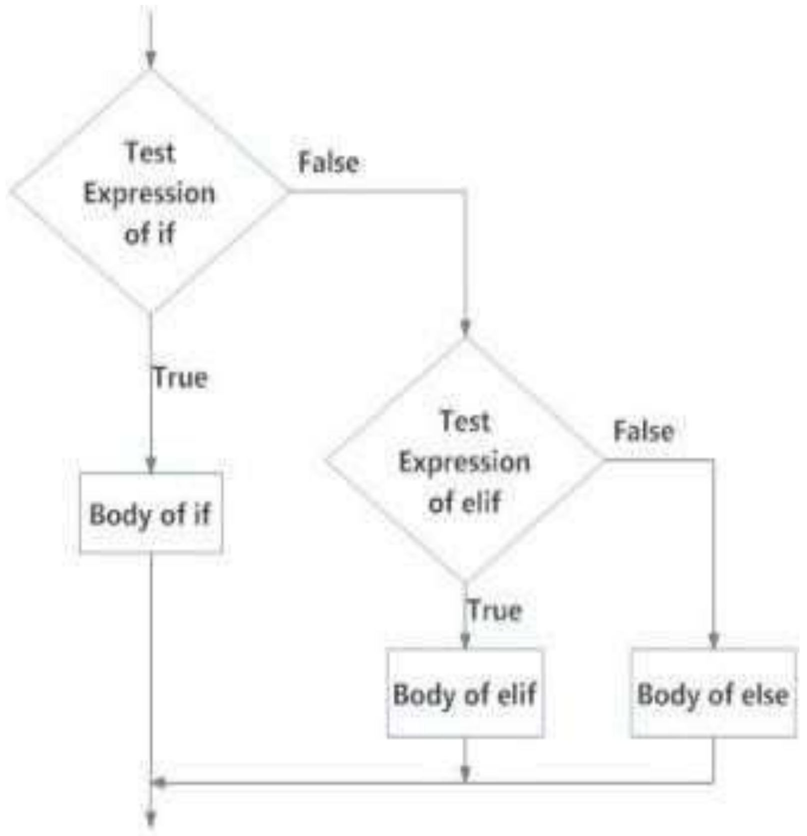
statement1

elif *#body to be executed in elif*

statement1

else *#body to be executed in else*

statement1



if...elif...else statement

The Elif Statement Example

```
age1 = 15

#check your age

#age1 = 12

#age1 = 19

if age1 < 12: #executed when the age1=12

print('my age is ', age1)

elif (age1 >= 12) and (age1 < 18): #executed when the age1=15

print('my age is ', age1)

else: #executed when the age1=19

#age1 >=18

    print('my age is ', age1)
```

Python Nested if Example

```
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if


num = float(input("Enter a number: "))
if num >= 0:
if num == 0:
print("Zero")
else:
print("Positive number")
else:
print("Negative number")
```

Example: largest among three numbers

```
a = int(input(“Enter 1st number:”))
b= int(input(“Enter 2nd number:”))
c= int(input(“Enter 3rd number:”))
if (a > b) and (a > c):
print("a is greater")
elif (b < a) and (b < c):
print(“b is greater")
else:
print(“c is greater")
```

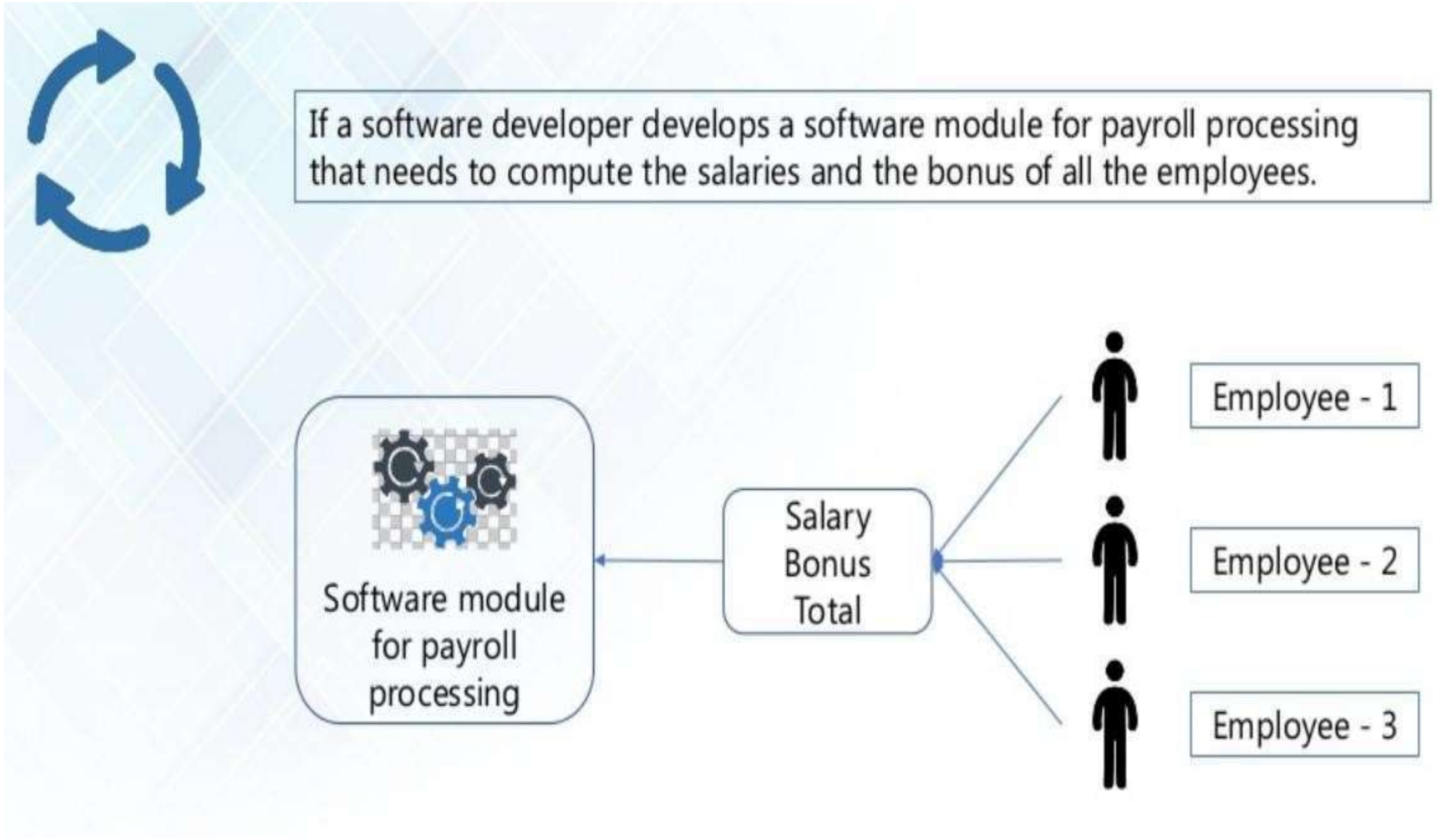
CONTROL STATEMENT (Looping Statement)

Program statement are executed sequentially one after another. In some situations, a block of code needs to execute number of times.

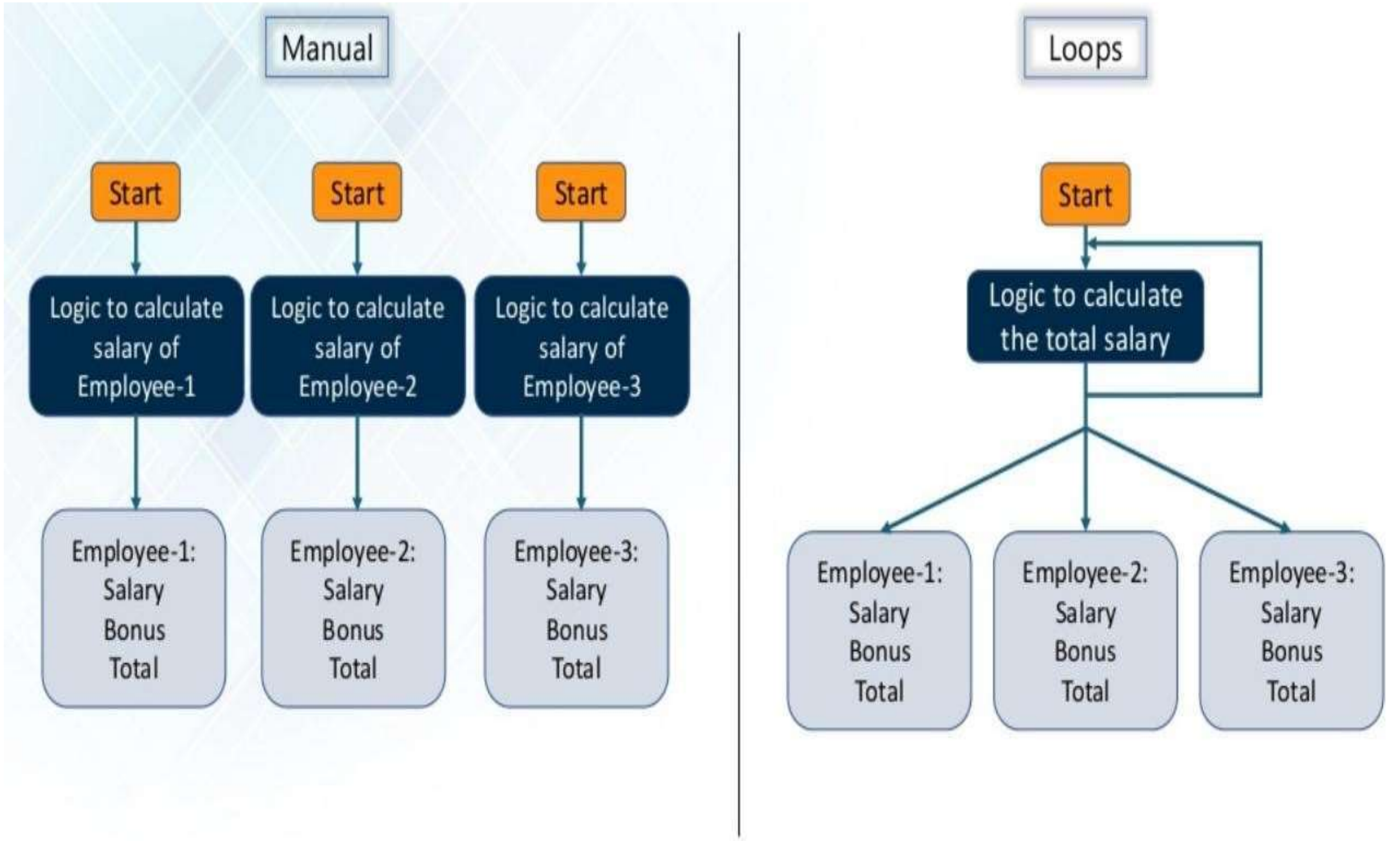
These are repetitive program codes, the computers have to perform to complete tasks. The following are the loop structures available in python.

- **while statement**
- **for loop statement**
- **Nested loop statement**

Why we Use Loop

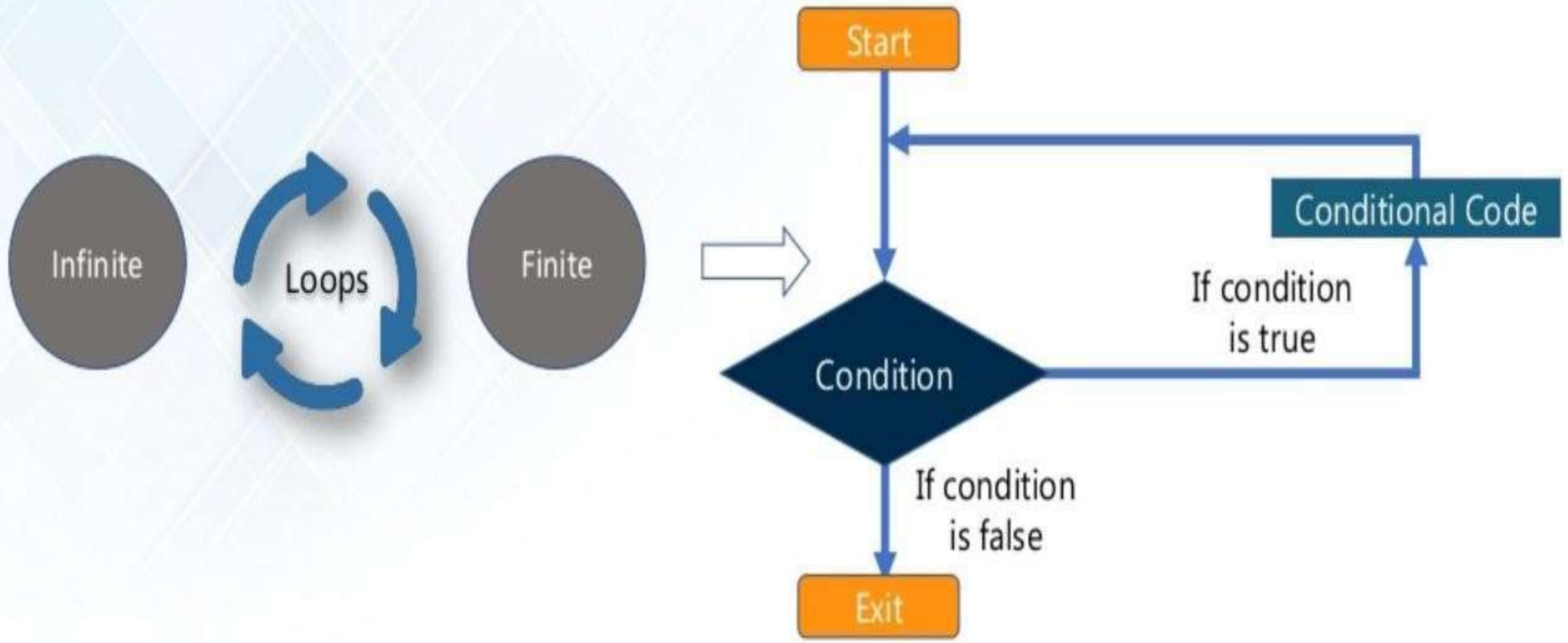

If a software developer develops a software module for payroll processing that needs to compute the salaries and the bonus of all the employees.

Why we Use Loop


Manual
Start
Start
Start
Logic to calculate salary of Employee-1
Logic to calculate salary of Employee-2
Logic to calculate salary of Employee-3
Employee-1: Salary Bonus Total
Employee-2: Salary Bonus Total
Employee-3: Salary Bonus Total
Loops
Start
Logic to calculate the total salary
Employee-1: Salary Bonus Total
Employee-2: Salary Bonus Total
Employee-3: Salary Bonus Total

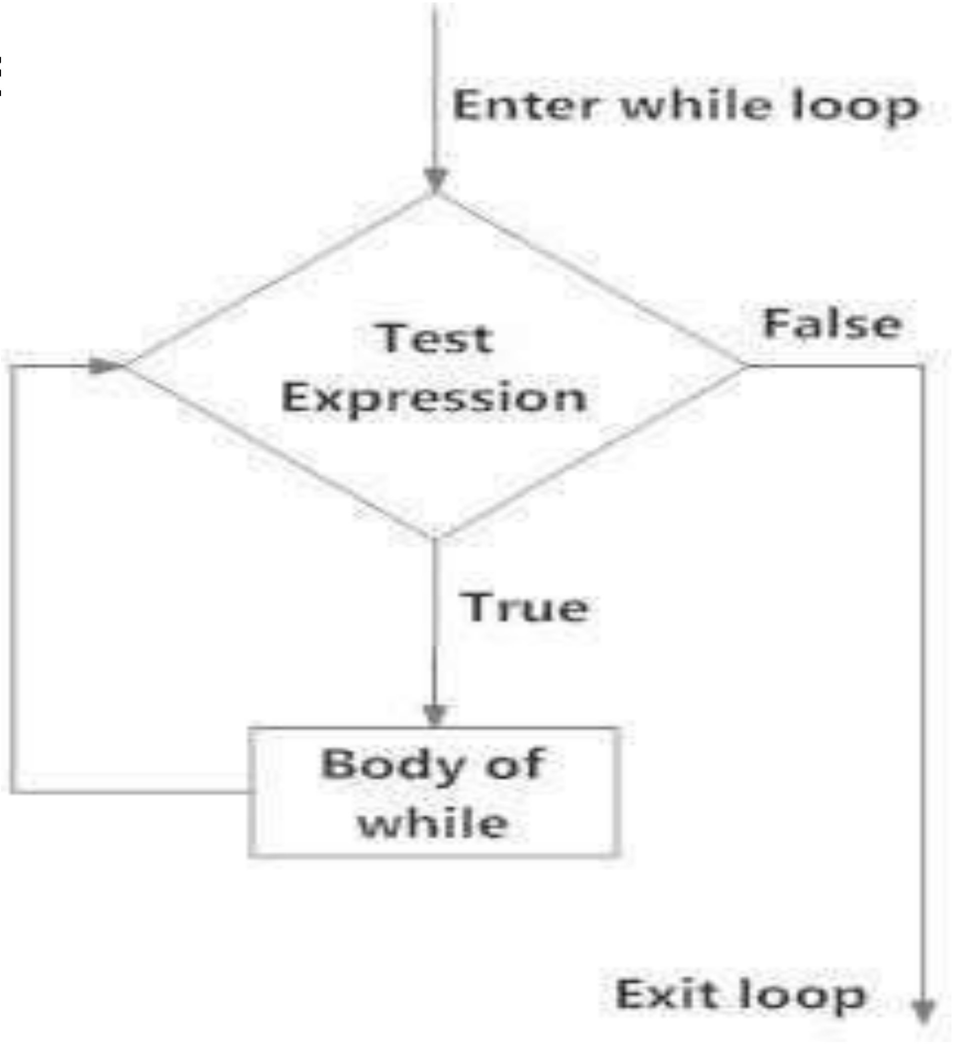
Why we Use Loop

- Loops allows the execution of a statement or a group of statement multiple times.
- In order to enter the loop there are certain conditions defined in the beginning.
- Once the condition becomes false the loop stops and the control moves out of the loop.



While loop statement

the while-loop. A whileloop will keep executing the code block under it as long as a Boolean expression is True.
they jump back to the "top" where the while is
and repeat. A while-loop runs until the
expression is False.



Syntax of while loop:

```
while expression:
        statement(s)
```

- Here’s the problem with while-loops: Sometimes they do not stop. To avoid these problems, there are some rules to follow:

1. Make sure that you use while-loops sparingly(use infrequently). Usually a for-loop is better.

2. Review your while statements and make sure that the Boolean test will become False at some point.

3. When in doubt, print out your test variable at the top and bottom of the while-loop to see what it’s doing.

While Loop

While

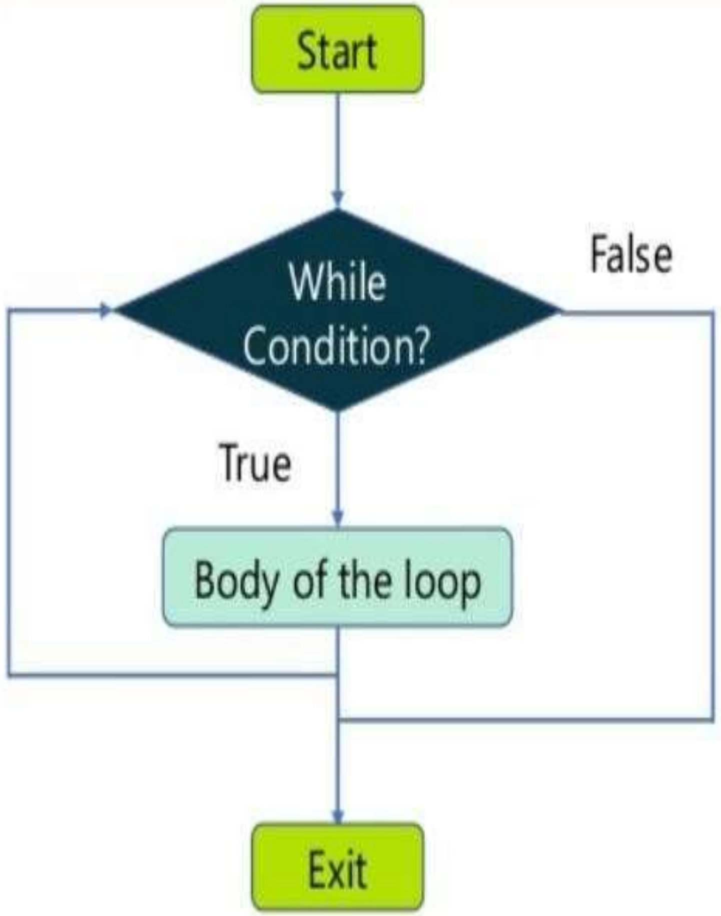
For

Nested

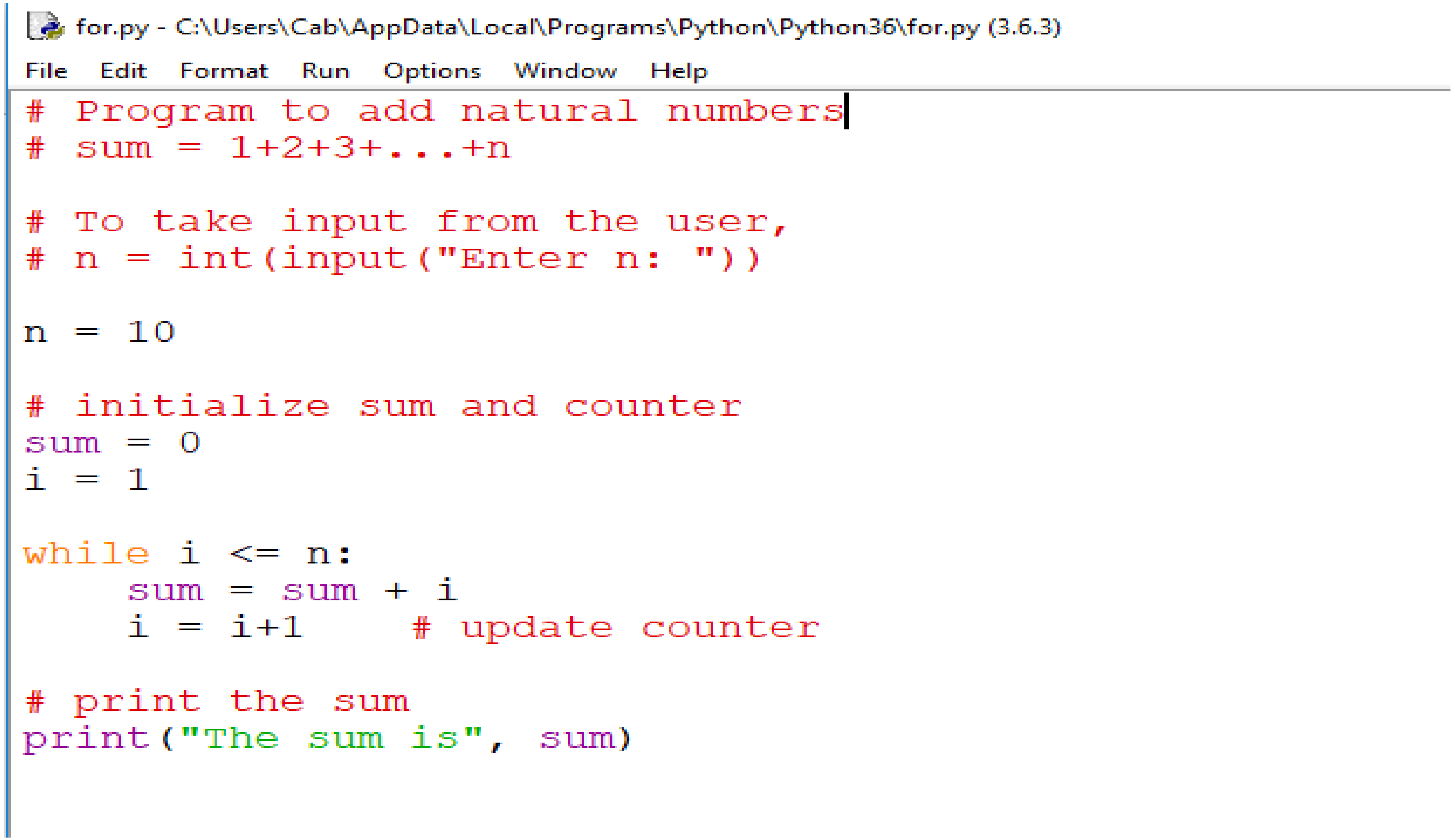
While loops are known as indefinite or conditional loops. They will keep iterating until certain conditions are met. There is no guarantee ahead of time regarding how many times the loop will iterate.

Syntax:

```
while expression:
    statements
```



While loop statement


for.py - C:\Users\Cab\AppData\Local\Programs\Python\Python36\for.py (3.6.3)
File Edit Format Run Options Window Help

```
# Program to add natural numbers
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1     # update counter

# print the sum
print("The sum is", sum)
```

For loop statement

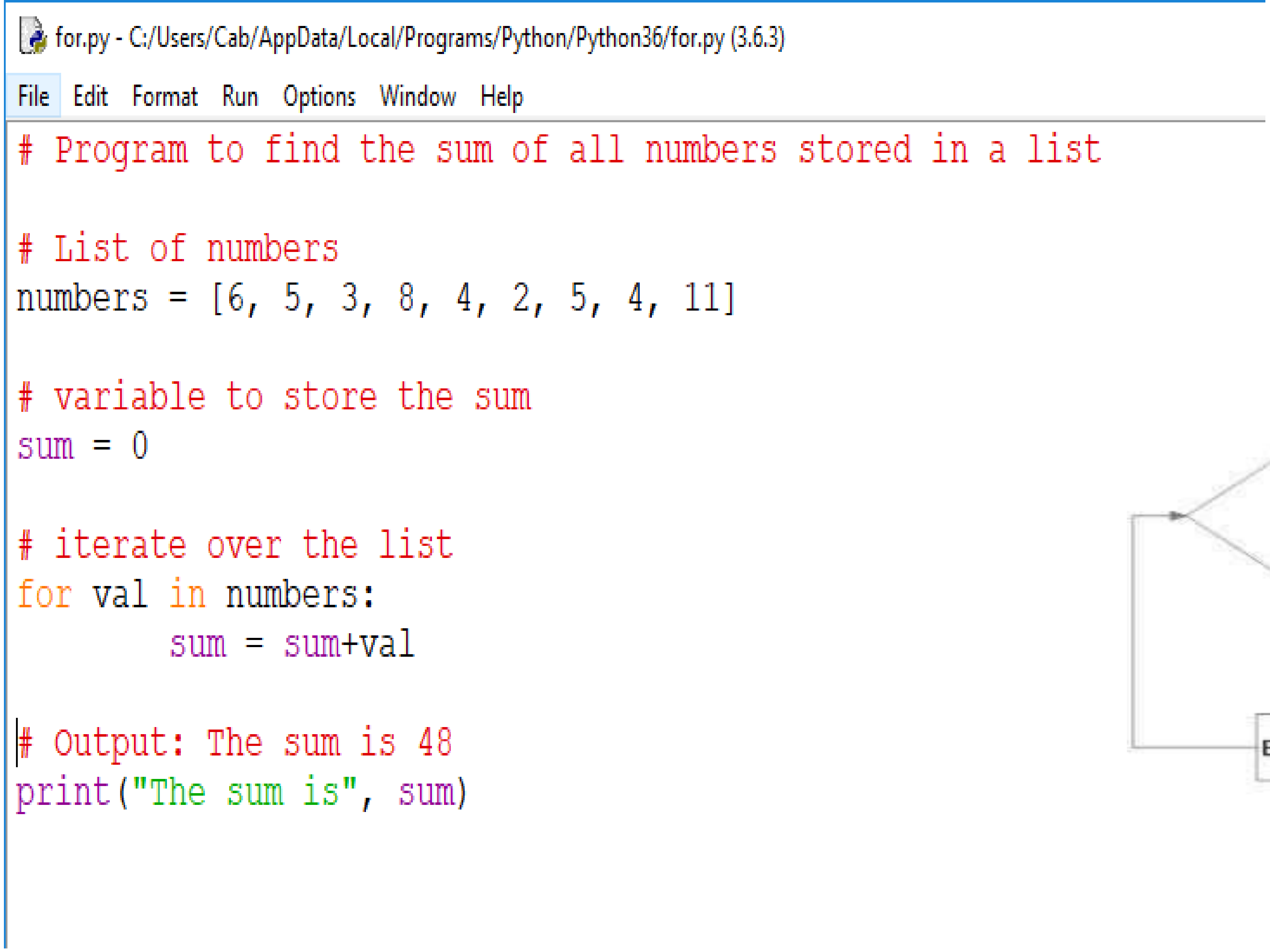
The for loop is another repetitive control structure, and is used to execute a set of instructions repeatedly, until the condition becomes false.
The for loop in Python is used to iterate over a sequence **(list, tuple, string)** or other iterable objects. Iterating over a sequence is called traversal.

For Loops

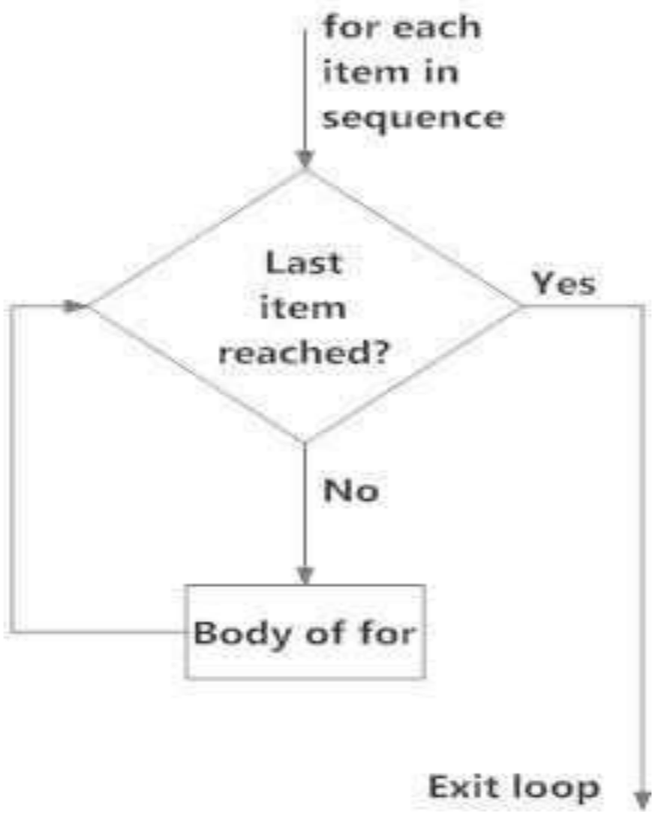
For loops are a pre-test loop
In order to utilize a for loop you need 3 things:
1. Needs to ***initialize a counter***
2. Must **check/*test the counter variable***
3. It must ***update the counter variable***

Syntax of for Loop:
for val in sequence:
Body of for loop

Here, **val** is the variable that takes the value of the item inside the **sequence** on each iteration
Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

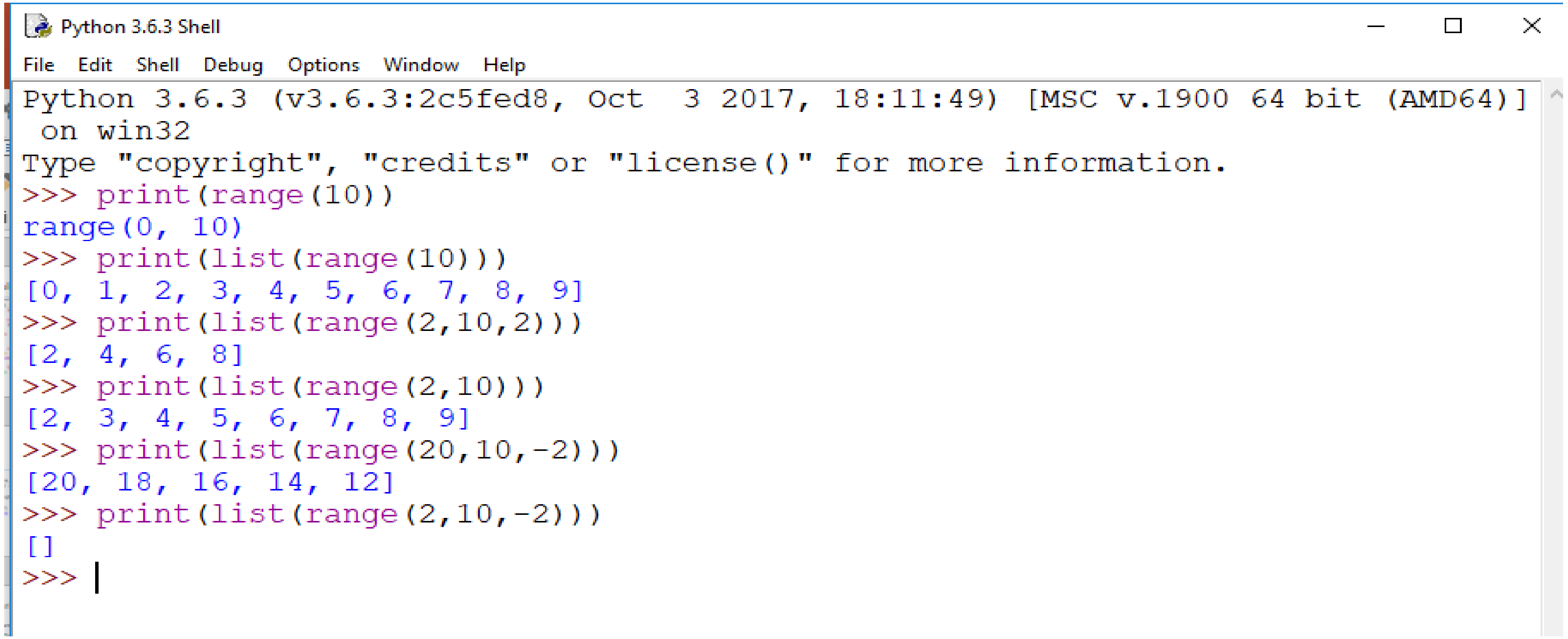

for.py - C:/Users/Cab/AppData/Local/Programs/Python/Python36/for.py (3.6.3)
File Edit Format Run Options Window Help

```python
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
        sum = sum+val

# Output: The sum is 48
print("The sum is", sum)
```



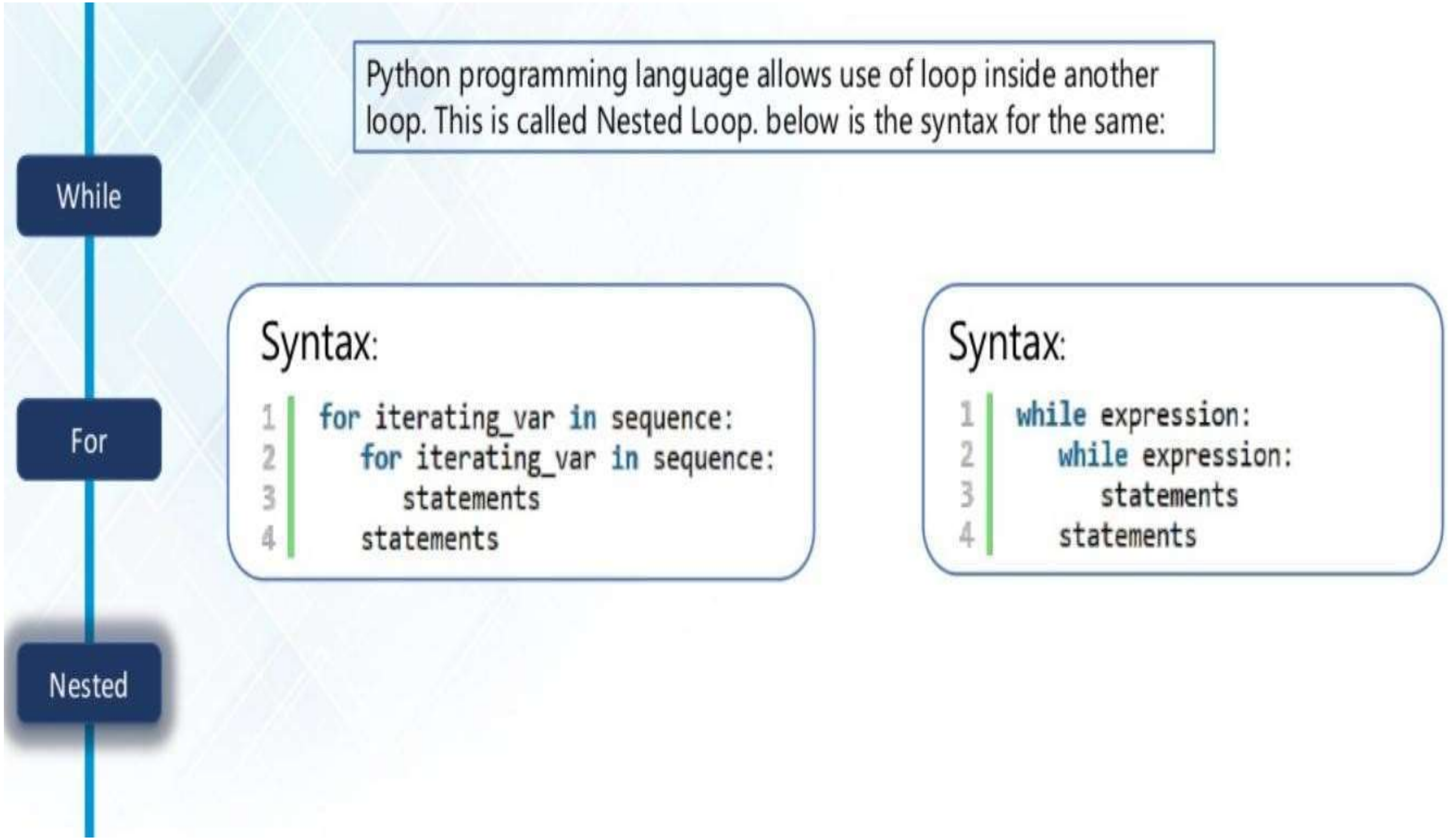
The range() function

- Can generate a sequence of numbers using range() function
- range(10) will generate numbers from 0 to 9 (10 numbers)
- Can also define the start, stop and step size as range(start,stop,stepsize)
- Step size defaults to 1 if not provided.
- Does not store all the values in memory, it would be inefficient
- So it remembers the start, stop, step size and generates the next number on the go



```
Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 18:11:49) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> print(range(10))
range(0, 10)
>>> print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(list(range(2,10,2)))
[2, 4, 6, 8]
>>> print(list(range(2,10)))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> print(list(range(20,10,-2)))
[20, 18, 16, 14, 12]
>>> print(list(range(2,10,-2)))
[]
>>> |
```

Nested Loop



Python programming language allows use of loop inside another loop. This is called Nested Loop. below is the syntax for the same:

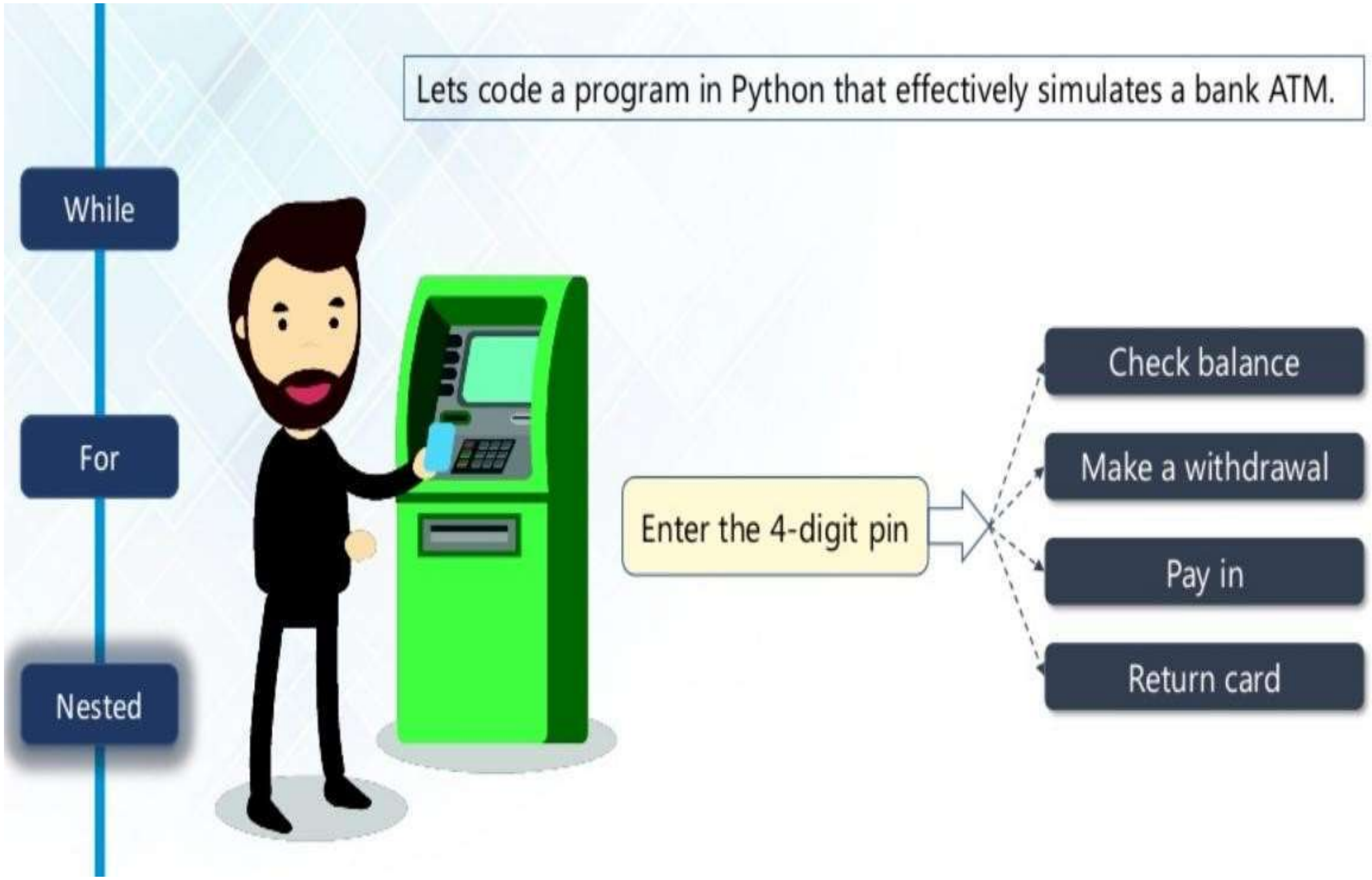
Syntax:

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements
   statements
```

Syntax:

```
while expression:
   while expression:
      statements
   statements
```

Nested Loop Example



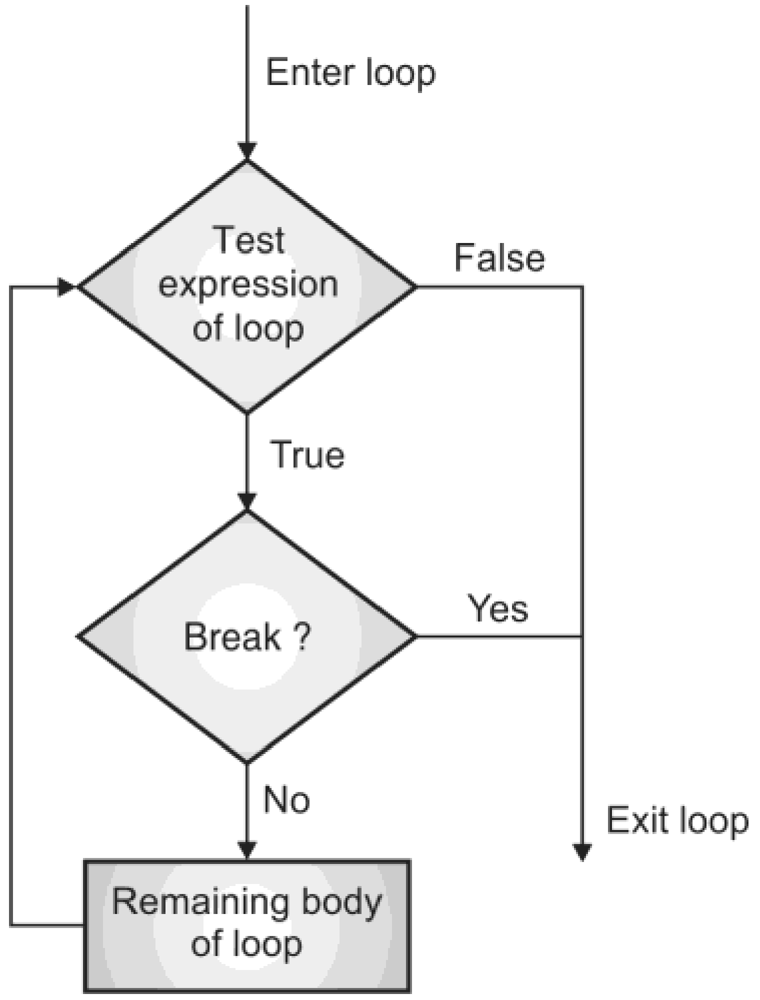
Break Statement :

- The break is a keyword in python which is used to bring the program control out of the loop.
- The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

Break Statement :

- ➢If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.
- ➢In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.
- ➢The break is commonly used in the cases where we need to break the loop for a given condition.

Flowchart of break :



Break Loop Example :

```
for letter in 'Python':
  if letter == 'h':
    break
  print 'Current Letter :', letter
```

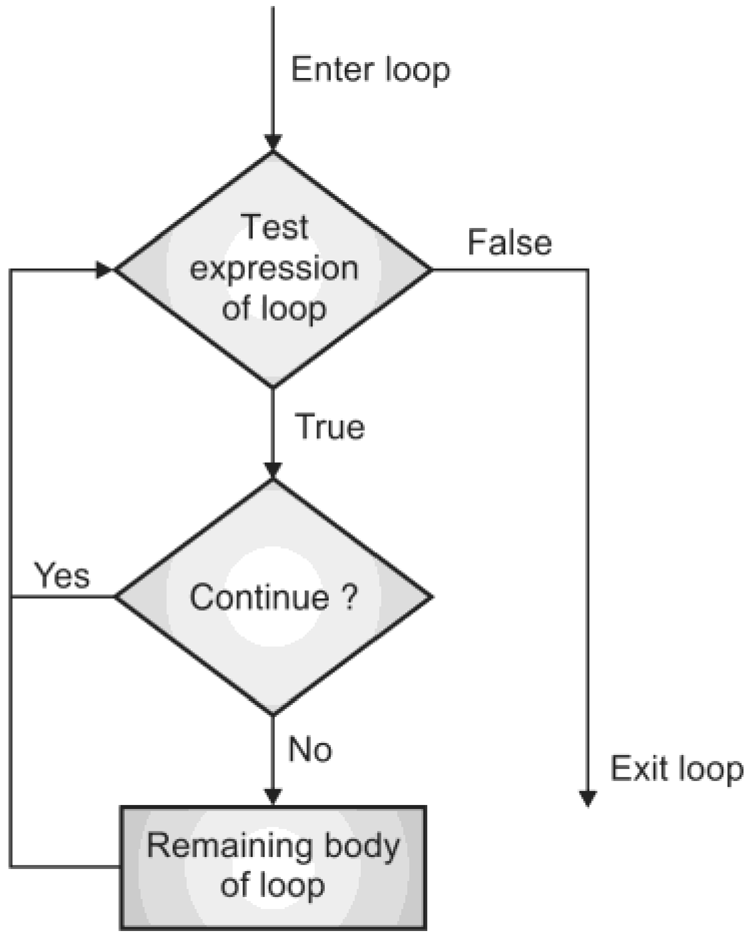
Output :

```
Current Letter : P
Current Letter : y
Current Letter : t
```

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.
- ➢It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

Flowchart of continue :



Continue Statement Example :

```
for letter in 'Python':
   if letter == 'h':
      continue
   print ('Current Letter :', letter)
```

Output :

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

pass Loop :

- In Python, **pass** keyword is used to execute nothing; it means, when we don’t want to execute code, the pass can be used to execute empty.
- It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.
- In Python programming, pass is a null statement.

pass Loop :

- In Python programming, pass is a null statement.
- The difference between a comment and *pass* statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.
- However, nothing happens when *pass* is executed. It results into no operation (NOP).
- It is generally used as a dummy statement in a code block, for example in the if or else block.

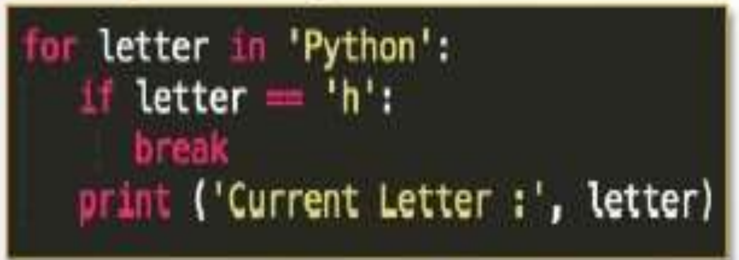
pass Loop Example :

```
for letter in 'Python':
   if letter == 'h':
      pass
      print ("This is pass block")
   print("Current Letter :", letter)

print ("Good bye!")
```

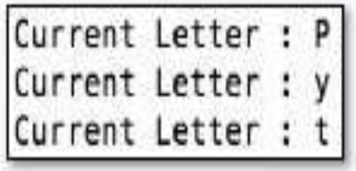
pass Loop Example :

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

- **break** :Terminates the loop statement and transfers execution to the statement immediately following the loop.

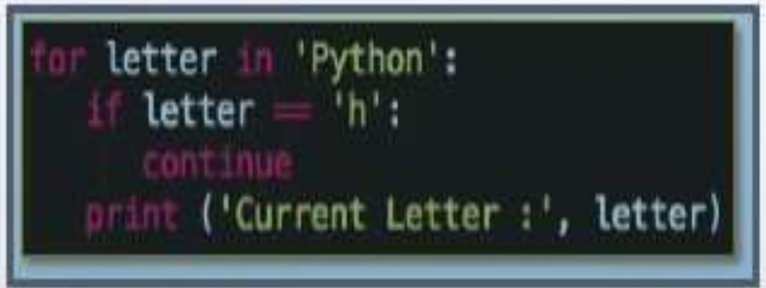


```python
for letter in 'Python':
   if letter == 'h':
      break
   print ('Current Letter :', letter)
```

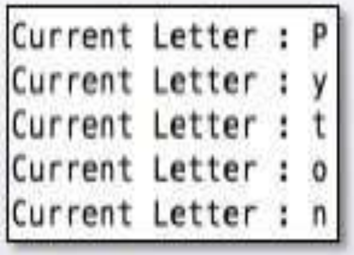


```
Current Letter : P
Current Letter : y
Current Letter : t
```

- **continue** :Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

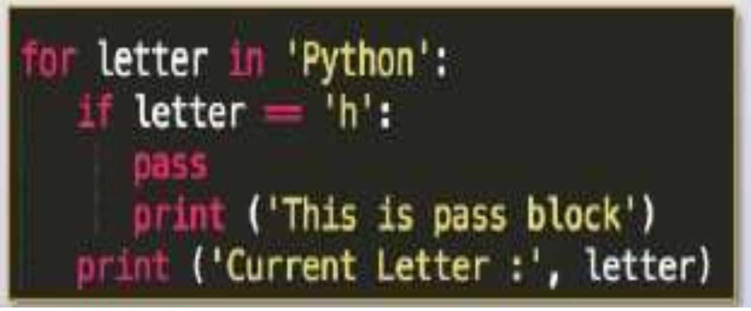


```python
for letter in 'Python':
   if letter == 'h':
      continue
   print ('Current Letter :', letter)
```

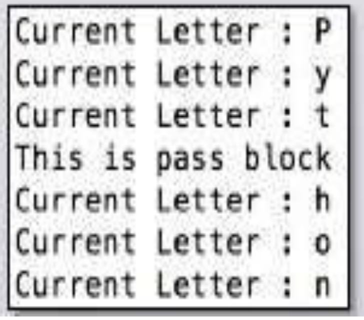


```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

- **pass** :Used when a statement is required syntactically but you do not want any command or code to execute.



```python
for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :', letter)
```



```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
```

Lists :

- Python offers a range of compound datatypes often referred to as sequences. List is one of the most frequently used and very versatile datatype used in Python.
- In Python, the list is a collection of items of different data types. It is an ordered sequence of items.
- A list object contains one or more items, not necessarily of the same type, which are separated by comma and enclosed in square brackets [].

Lists :

Create a List:

```
mylist = ["apple", "banana", "cherry“]
print(mylist)
```

Output :

```
['apple', 'banana', 'cherry‘]
```

Access List Items :

```
mylist  = ["apple", "banana", "cherry“]
print(mylist[1])
```

Output :

```
banana
```

Tuples :

- ➢A tuple in Python is similar to a list.
- ➢The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas, in a list, elements can be changed.
- ➢Tuple is a collection of items of any Python data type, same as the list type. Unlike the list, tuple is immutable.
- ➢The tuple object contains one or more items, of the same or different types, separated by comma and enclosed in parentheses ().

Tuples Example:

Create a Tuple:

```
mytuple = ("apple", "banana", "cherry")
print(mytuple)
```

Output :

```
('apple', 'banana', 'cherry')
```

Accessing Tuple Items :

```
mytuple = ("apple", "banana", "cherry")
print(mytuple[1])
```

Output :

```
banana
```

Tuples Example:

```
mytuple = ("apple", "banana", "cherry")
print(mytuple[-1])
```

Output :

```
cherry
```

```
mytuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(mytuple [2:5])
```

Output :

```
('cherry', 'orange', 'kiwi')
```

Tuples Example:

```
mytuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(mytuple [-4:-1])
```

Output :

```
('orange', 'kiwi', 'melon')
```

Dictionary :

- ➢Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.
- ➢Dictionaries are optimized to retrieve values when the key is known.
- ➢One or more key:value pairs separated by commas are put inside curly brackets to form a dictionary object.

Create and print a dictionary:

```
mydict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(mydict)
```

Output :

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Accessing Items :

```
mydict =     {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Get the value of the "model" key :

```
x = mydict["model"]
```

Output :

```
Mustang
```

Accessing Items :

There is also a method called get() that will give the same result.

```
mydict =        {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Get the value of the "model" key:

```
x = mydict.get("model")
```

Output :

```
Mustang
```

Dictionary Methods :

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair

Dictionary Methods :

keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary