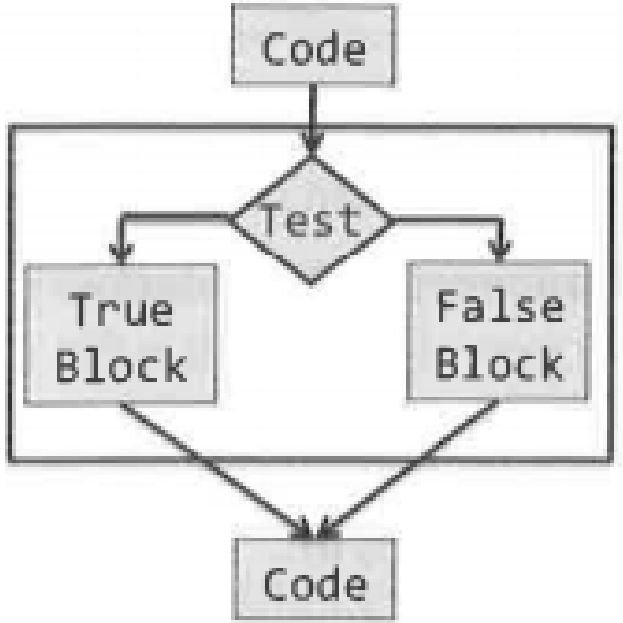
Module II CONTROL STRUCTURE AND FUNCTION 9

Conditionals: If-Else Constructs – Loop Structures/Iterative Statements – While Loop – For Loop – Break Statement – Function Call and Returning Values – Parameter Passing – Local and Global Scope – Recursive Functions.

Branching Programms

- The simplest branching statement is a conditional.
- The conditional statement has three parts:

1. a test (an expression that evaluates to either True or False)
2. a block of code that is executed if the test evaluates to True and
3. an optional code that is executed if the test evaluates to False.



- Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:
  - **if statement**
  - **if..else statement**
  - **nested if statements**
  - **if-elif ladder**
  - **ShortHand if statement**
  - **Short Hand if-else statement**

if statement

- simple decision making statement.
- It is used to decide whether a certain statement or block of statements will be executed or not

Syntax:

```
if condition:
    # Statements to execute if
    # condition is true
```

```
if condition:
statement1
statement2
```

```
if(off==0):
    print("room is dark")
```

```
>>> if 15>5:
...     print("True")
...     print("The program continuous here....")
...
True
The program continuous here....
```

```
>>> a=int(input("Enter a number:"))
Enter a number:7
>>> if(a>5):
...     print("The number your entered is greater than 5!")
...     print("Thanks for the input")
...
The number your entered is greater than 5!
Thanks for the input
```

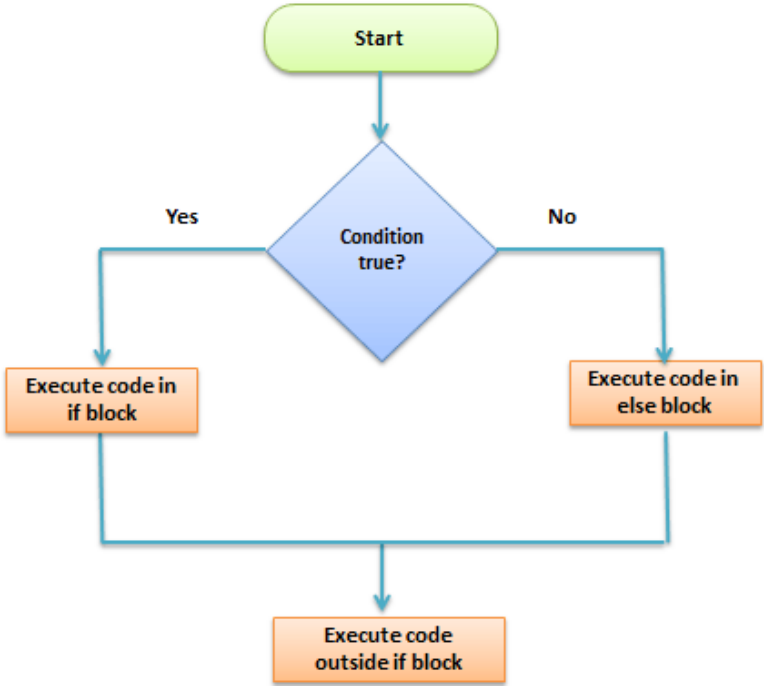
if- else

- The if statement alone tells us that if a condition is true
- We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):
  # Executes this block  if
     # condition is true
else:
  # Executes this block if
    # condition is false
```

```
if(result > 45)
        print("Pass")
Else:
        print("Fail")
```



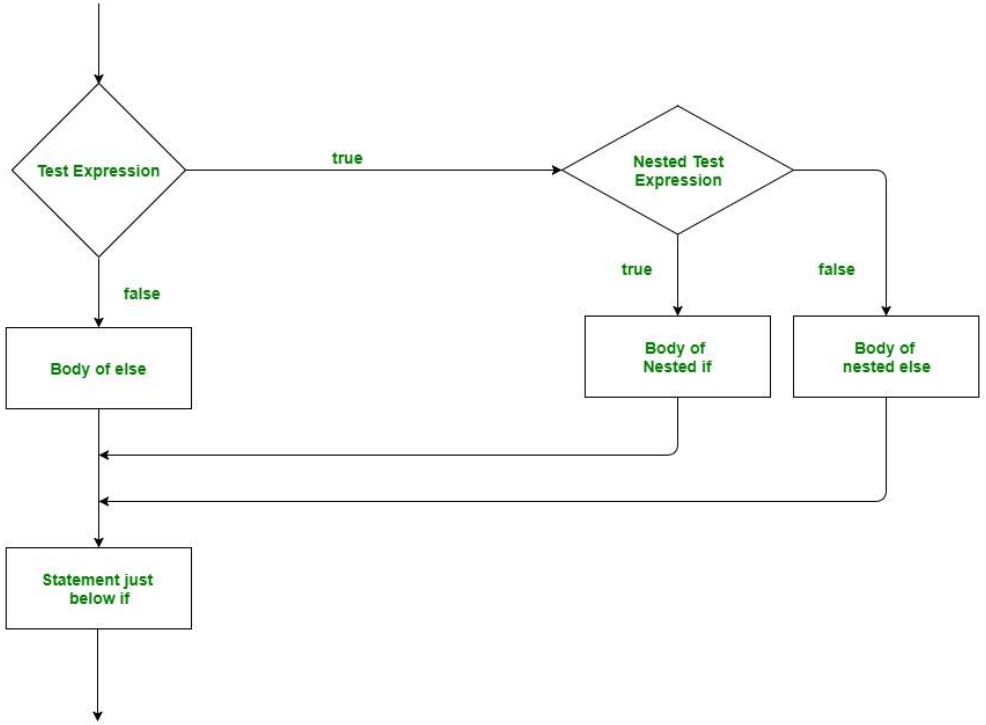
```
>>> a=10
>>> b=20
>>> if(a<b):
...     print("a is small")
... else:
...     print("b is big")
...
a is small
```

```
>>> x=int(input("Enter the number"))
Enter the number10
>>> y=int(input("Enter the second number"))
Enter the second number20
>>> if(x<y):
...     print("y is big")
... else:
...     print("x is big")
...
y is big
```

nested-if

- A nested if is an if statement that is the target of another if statement.

```
if (condition1):
  # Executes when condition1 is true
if (condition2):
  # Executes when condition2 is true
  # if Block is end here
  # if Block is end here
```

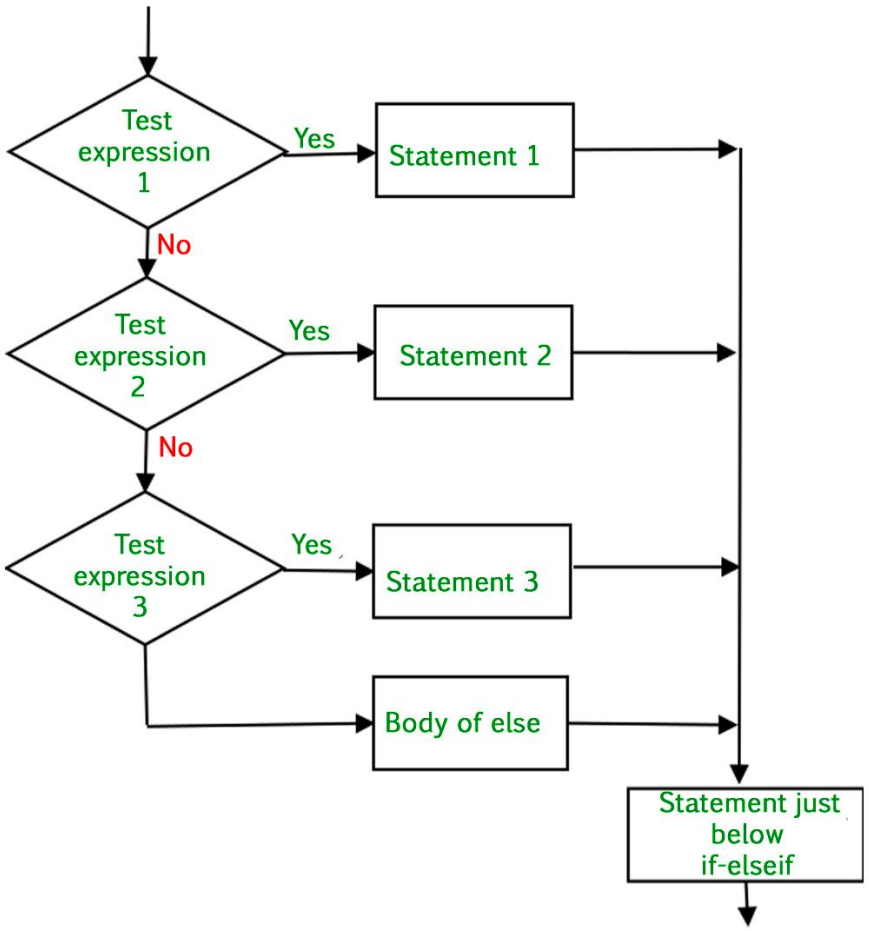


```
>>> a=33
>>> b=33
>>> if(a>b):
...     print("a is big")
... elif(a==b):
...     print("a and b equal")
...
a and b equal
```

if-elif-else ladder

```
if (condition):
        statement
elif(condition):
        statement . .
 else:
        statement
```

```
i = 20
if (i == 10):
    print ("i is 10")
elif (i == 15):
    print ("i is 15")
elif (i == 20):
    print ("i is 20")
else:
    print ("i is not present")
```



Short Hand if statement

Syntax:

```
if condition: statement
```

```
i = 10
if i < 15: print("i is less than 15")
```

Short Hand if-else statement

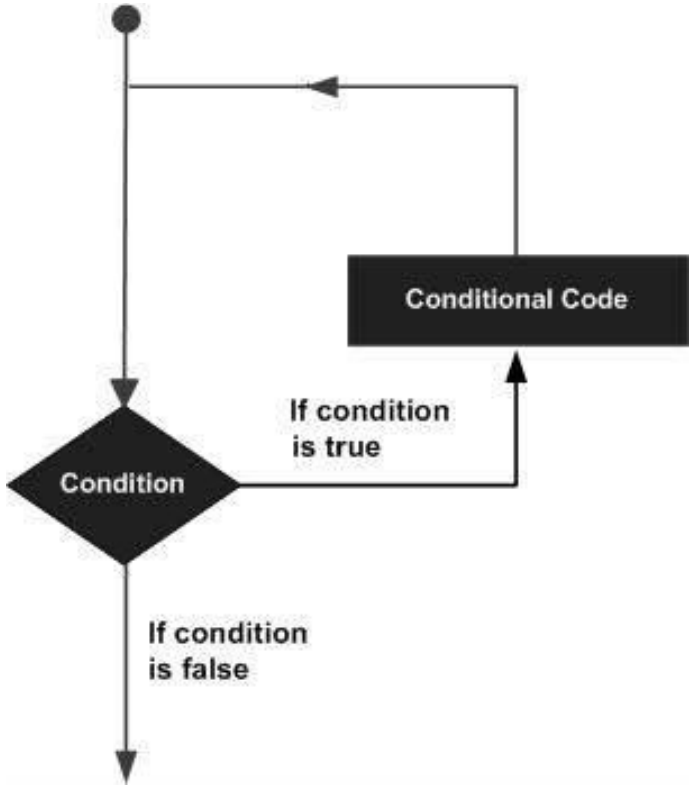
Syntax:

```
statement_when_True if condition else statement_when_False
```

```
i = 10
print(True) if i < 15 else print(False)
```

Looping statement

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement



for

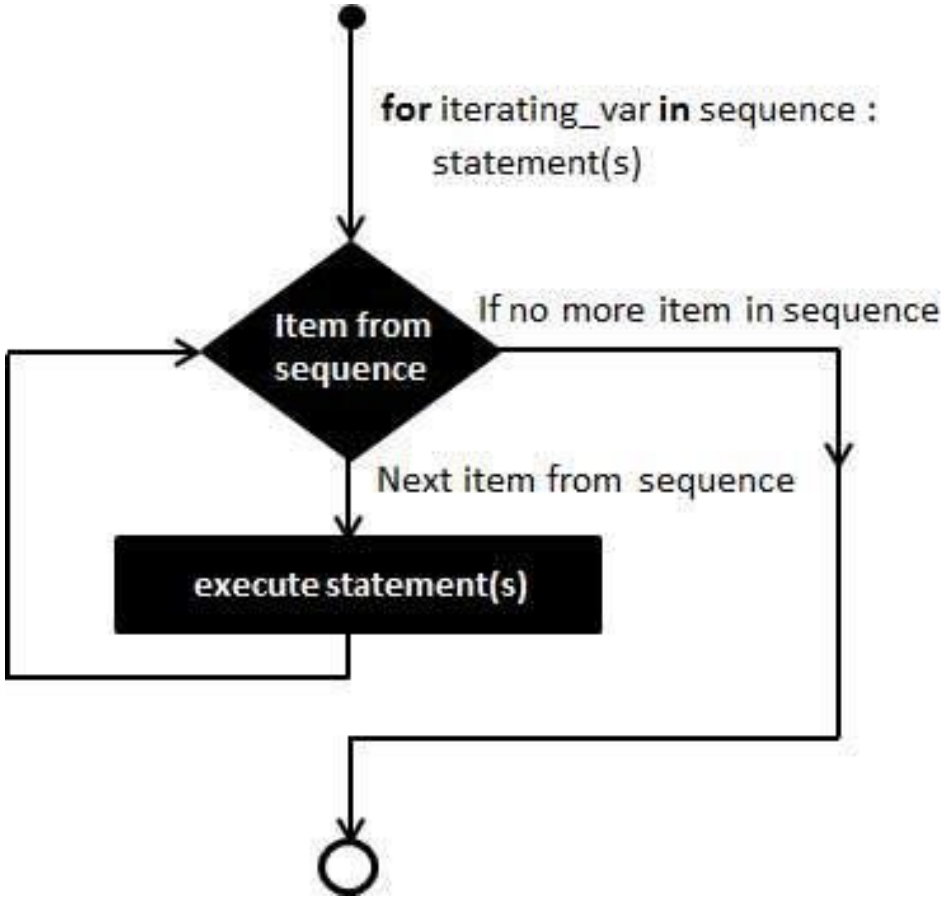
- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

while

- Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

for

- It has the ability to iterate over the items of any sequence, such as a list or a string.
- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).



Syntax

for iterating_var in sequence: statements(s)

```
>>> fruits=["apple","banana","cherry"]
>>> for x in fruits:
...     print(x)
...
apple
banana
cherry
```

```
>>> for x in "apple":
...     print(x)
...
a
p
p
l
e
```

```
>>> fruits=["apple","banana","cherry"]
>>> for index in range(len(fruits)):
...     print("current fruit:", fruits[index])
...
current fruit: apple
current fruit: banana
current fruit: cherry
```

```python
for x in range(5):
  print(x)
```

```python
for x in range(3, 9):
  print(x)
```

```python
for x in range(3, 30, 3):
  print(x)
```

```python
for x in range(5):
  print(x)
else:
  print("Finally finished!")
```

```python
for x in range(5):
  if x == 2: break
  print(x)
else:
  print("Finally finished!")
```

```python
for x in [0, 1, 2]:
  pass
```

```python
num=[1,2,3,4]
sum=0
for val in num:
    sum=sum+val
```

```python
for i in range(1,5):
    for j in range(i):
        print(i,end= ‘ ’)
    print()
```

```
1
22
333
4444
```

while

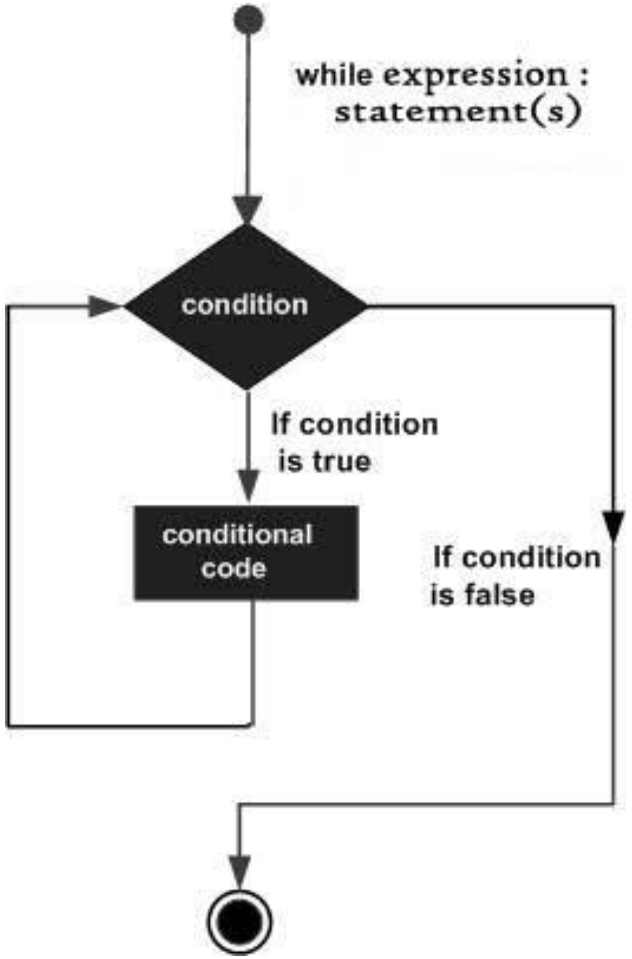
- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

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

```
>>> count =0
>>> while(count<9):
...     print("The count is:",count)
...     count=count+1
...
```

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
```



```
>>> var=1
>>> while(var==1):
...     num=int(input("Enter the value"))
...     print("you entered :", num)
...
Enter the value10
you entered : 10
```

```
>>> count=0
>>> while(count<5):
...     print(count,"is less than 5")
...     count=count+1
... else:
...     print(count, "is not less than 5")
...
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

```
i = 1
while i < 5:
  print(i)
  if i == 2:
    break
  i += 1
```

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
i=0
while i<10:
    print(i)
    i+=1
    break
else:
    print("hello")
```

```
i=0
j=0
while i<5:
    while j<3:
        print(j)
        j+=1
    print(i)
    i+=1
```

Nested Loop

- Python programming language allows to use one loop inside another loop.

Syntax

```
for iterating_var in sequence:
for iterating_var in sequence:
statements(s) statements(s)
```

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

```
>>> for i in range(1,5):
...     for j in range(i):
...             print(i,end=' ')
...     print()
...
1
2 2
3 3 3
4 4 4 4
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

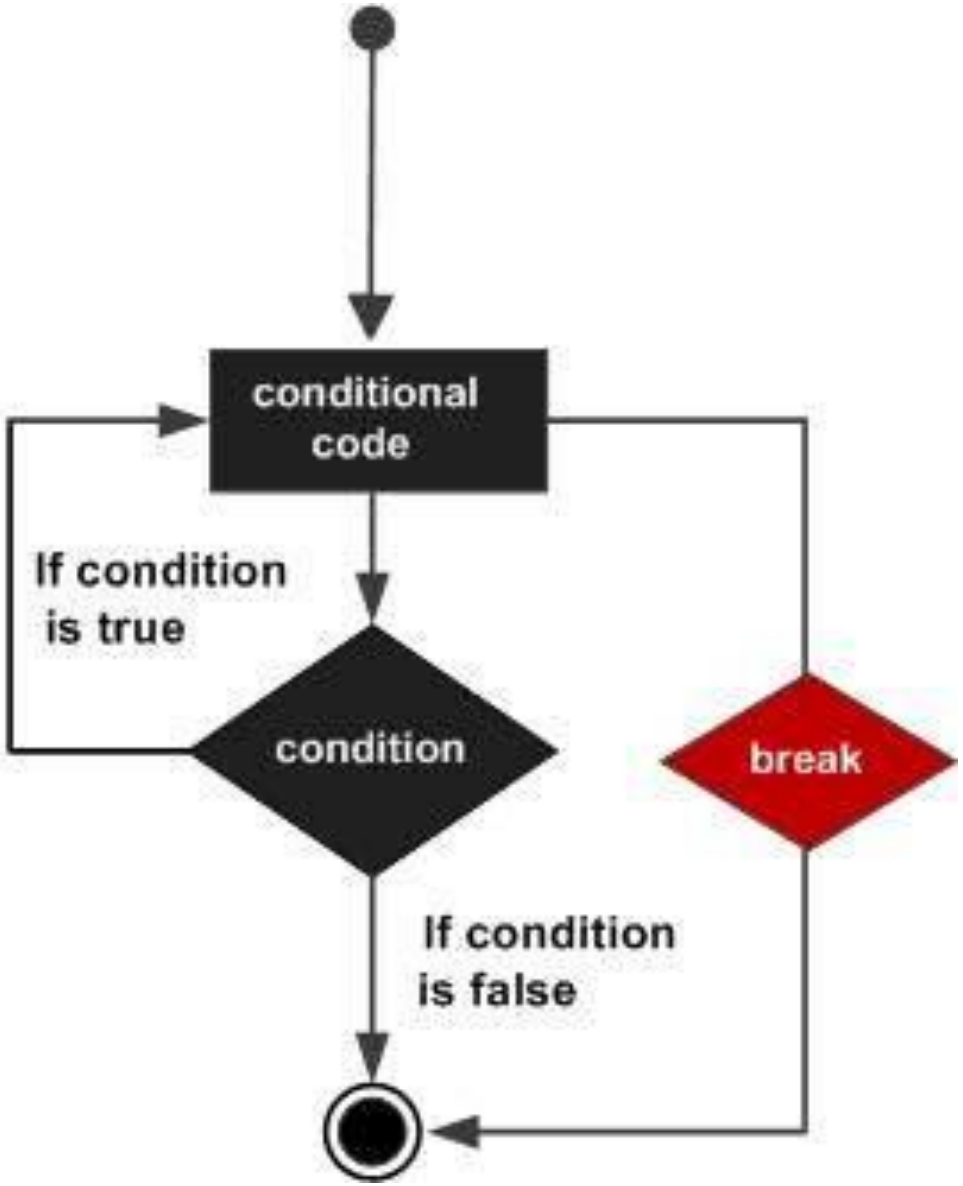
- **break statement** - Terminates the loop statement and transfers execution to the statement immediately following the loop.

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

- **pass statement -** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

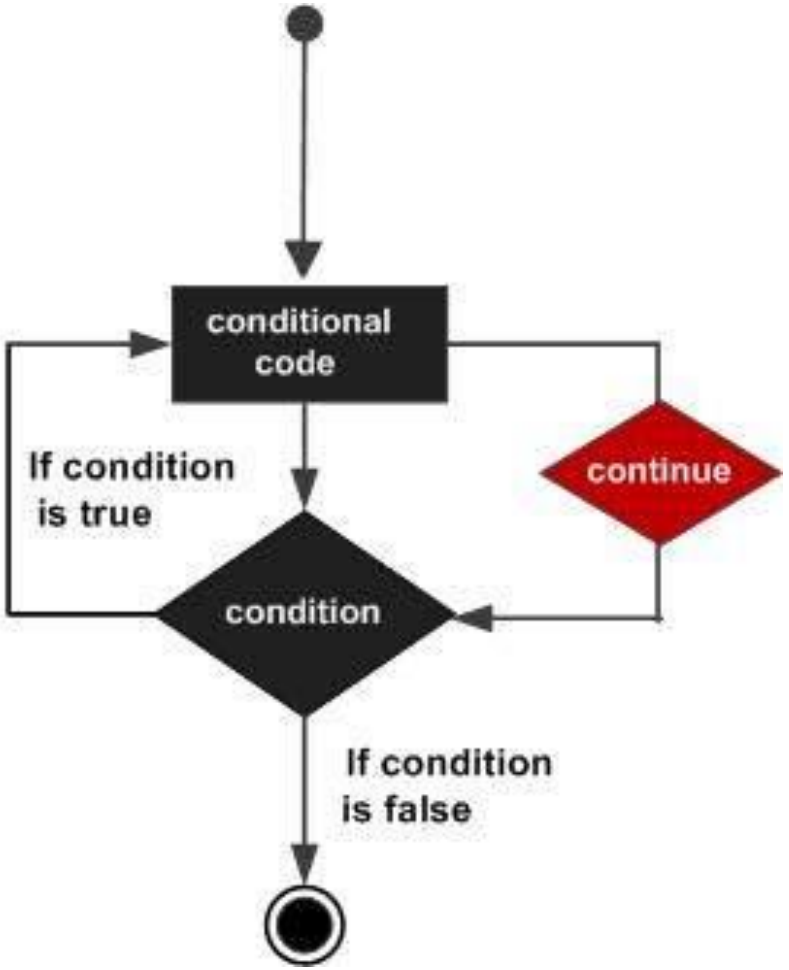
Break Statements

```
>>> for letter in "python":
...     if letter=="h":
...             break
...     print("current letter:", letter)
...
current letter: p
current letter: y
current letter: t
```



Continue Statements

```
>>> 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 Statements

- The **pass** statement is a *null* operation; nothing happens when it executes.

```
>>> 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
```

Functions

- A function is a block of code which only runs when it is called.
- pass data, known as parameters, into a function.
- A function can return data as a result.
- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.

Syntax

```
def functionname( parameters ):
"function_docstring"
function_suite return
[expression]
```

```
>>> def myfun():
...     print("hello")
...
>>> myfun()
hello
```

Types:

1. Built-in function
2. User defined function

Math

Function	Description
`abs()`	Returns absolute value of a number
`divmod()`	Returns quotient and remainder of integer division
`max()`	Returns the largest of the given arguments or items in an iterable
`min()`	Returns the smallest of the given arguments or items in an iterable
`pow()`	Raises a number to a power
`round()`	Rounds a floating-point value
`sum()`	Sums the items of an iterable

Type Conversion

Function	Description
`ascii()`	Returns a string containing a printable representation of an object
`bin()`	Converts an integer to a binary string
`bool()`	Converts an argument to a Boolean value
`chr()`	Returns string representation of character given by integer argument
`complex()`	Returns a complex number constructed from arguments
`float()`	Returns a floating-point object constructed from a number or string
`hex()`	Converts an integer to a hexadecimal string
`int()`	Returns an integer object constructed from a number or string

`oct()`	Converts an integer to an octal string
`ord()`	Returns integer representation of a character
`repr()`	Returns a string containing a printable representation of an object
`str()`	Returns a string version of an object
`type()`	Returns the type of an object or creates a new type object

Iterables and Iterators

Function	Description
`all()`	Returns `True` if all elements of an iterable are true
`any()`	Returns `True` if any elements of an iterable are true
`enumerate()`	Returns a list of tuples containing indices and values from an iterable
`filter()`	Filters elements from an iterable
`iter()`	Returns an iterator object
`len()`	Returns the length of an object
`map()`	Applies a function to every item of an iterable
`next()`	Retrieves the next item from an iterator

`range()`	Generates a range of integer values
`reversed()`	Returns a reverse iterator
`slice()`	Returns a `slice` object
`sorted()`	Returns a sorted list from an iterable
`zip()`	Creates an iterator that aggregates elements from iterables

Composite Data Type

Function	Description
`bytearray()`	Creates and returns an object of the `bytearray` class
`bytes()`	Creates and returns a `bytes` object (similar to `bytearray`, but immutable)
`dict()`	Creates a `dict` object
`frozenset()`	Creates a `frozenset` object
`list()`	Creates a `list` object
`object()`	Creates a new featureless object
`set()`	Creates a `set` object
`tuple()`	Creates a `tuple` object

Classes, Attributes, and Inheritance

Function	Description
`classmethod()`	Returns a class method for a function
`delattr()`	Deletes an attribute from an object
`getattr()`	Returns the value of a named attribute of an object
`hasattr()`	Returns `True` if an object has a given attribute
`isinstance()`	Determines whether an object is an instance of a given class
`issubclass()`	Determines whether a class is a subclass of a given class
`property()`	Returns a property value of a class
`setattr()`	Sets the value of a named attribute of an object
`super()`	Returns a proxy object that delegates method calls to a parent or sibling class

Input/Output

Function	Description
`format()`	Converts a value to a formatted representation
`input()`	Reads input from the console
`open()`	Opens a file and returns a file object
`print()`	Prints to a text stream or the console

Variables, References, and Scope

Function	Description
`dir()`	Returns a list of names in current local scope or a list of object attributes
`globals()`	Returns a dictionary representing the current global symbol table
`id()`	Returns the identity of an object
`locals()`	Updates and returns a dictionary representing current local symbol table
`vars()`	Returns `__dict__` attribute for a module, class, or object

Miscellaneous

Function	Description
`callable()`	Returns `True` if object appears callable
`compile()`	Compiles source into a code or AST object
`eval()`	Evaluates a Python expression
`exec()`	Implements dynamic execution of Python code
`hash()`	Returns the hash value of an object
`help()`	Invokes the built-in help system
`memoryview()`	Returns a memory view object
`staticmethod()`	Returns a static method for a function
`__import__()`	Invoked by the `import` statement

Function Arguments

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

- Required arguments are the arguments passed to a function in correct positional order.

```
>>> def samp(str):     # formal argument
...     print(str)
...     return;
...
>>> samp("hello")      #actual argument
hello
```

Keyword arguments

- Keyword arguments are related to the function calls.
- It use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

```
>>> def sample(str):
...     print(str)
...     return;
...
>>> sample(str="hello")
hello
```

```
>>> def stud(rrn,name):
...     print("RRN:", rrn)
...     print(“Name:",name)
...     return;
...
>>> stud(rrn=2001,name="abdul")
RRN: 2001
Name: abdul
```

Default arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
>>> def samp(str):    # formal argument
...     print(str)
...     return;
...
>>> samp("hello")     #actual argument
hello
```

```
>>> def add(x,y):   # formal argument
...     if(x>y):
...             return x
...     else:
...             return y
...
>>> add(4,5)        #actual argument
5
```

Variable-length arguments

- The process a function for more arguments than you specified while defining the function.
- These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

```
def functionname([formal_args,] *var_args_tuple ):
"function_docstring"
function_suite return [expression]
```

- An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments.

```
>>> def stud(rrn,*age):
...     print ("Output is:")
...     print (rrn)
...     for a in age:
...             print (a)
...     return;
...
>>> print(2001,19)
2001 19
```

The Anonymous Functions

- The functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.
- we can use the *lambda* keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

```
>>> l=lambda x, y : x * y
>>> print(l(5,5))
25
```

```
>>> l=lambda x, y,z : x + y+z
>>> print(l(5,5,5))
15
```

```
>>> l=lambda x, y,z : x + y*z
>>> print(l(5,5,5))
30
```

```
>>> l=lambda x:x+5
>>> print(l(5))
10
```

```
>>> l=lambda x, y,z : x + y%z
>>> print(l(5,5,5))
5
```

```
>>> l=lambda x, y,z : x + y-z
>>> print(l(5,5,5))
5
```

```
>>> l=lambda x, y,z : x + y/z
>>> print(l(5,5,5))
6.0
```

```
>>> add=lambda a,b: a+b;
>>> print("The total is");
The total is
>>> add(10,10)
20
```

```
>>> add=lambda a,b : a+b
>>> sub=lambda x,y : x-y
>>> add(10,10)
20
>>> sub(20,10)
10
```

The *return* Statement

- The statement return [expression] exits a function, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return None.

```
>>> def add(a,b):
...     print(a)
...     print(b)
...     return;
...
>>> add(a=10,b=10)
10
10
```

Scope

Local Scope - A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Global Scope - A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Local Scope

```
>>> def fun():
...     x=10
...     print(x)
...
>>> fun()
10
```

```
>>> def fun():
...     x=10
...     def infun():
...             print(x)
...     infun()
...
>>> fun()
10
```

Global Scope

```
>>> x=10
>>> def fun():
...     x=20
...     print(x)
...
>>> fun()
20
>>> print(x)
10
```

Global Keyword

- The global keyword makes the variable global.

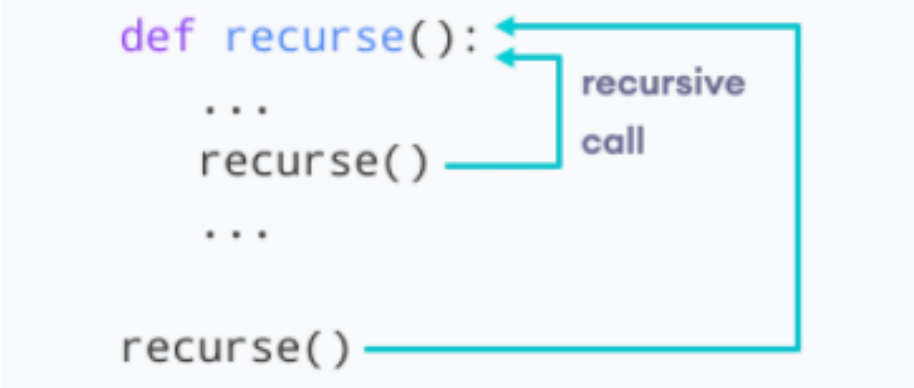
```
>>> def fun():
...     global x
...     x=10
...
>>> fun()
>>> print(x)
10
```

- To make a change to a global variable inside a function

```
>>> x=10
>>> def fun():
...     global x
...     x=20
...
>>> fun()
>>> print(x)
20
```

Recursion

- a defined function can call itself.
- Recursion is a common mathematical and programming concept



Recursive Function in Python

```
>>> def factorial(x):
...     if(x==1):
...             return 1
...     else:
...             return(x*factorial(x-1))
...
>>> num=4
>>> print(factorial(num))
24
```

Thank You