**UNIT – III**

# 1. Control flow :

In order to control the flow of execution of a program there are three categories of statements in python.They are:

1. Selection statements
2. iteration statements
3. Jump statements

## 1. 1. Selection Statements

Decision making is valuable when something we want to do depends on some user input or some other value that is not known when we write our program. This is quite often the case and Python, along with all interesting programming languages, has the ability to compare values and then take one action or another depending on that outcome.

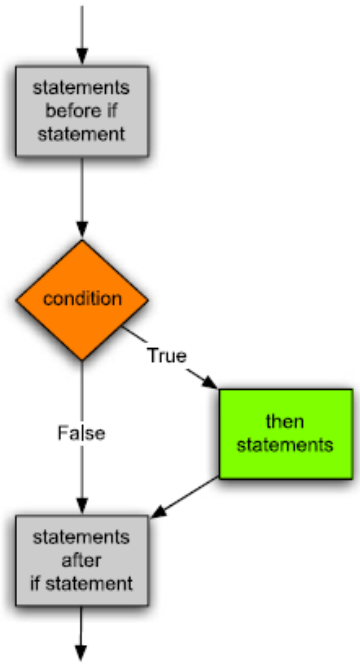


**Figure: If Statement**

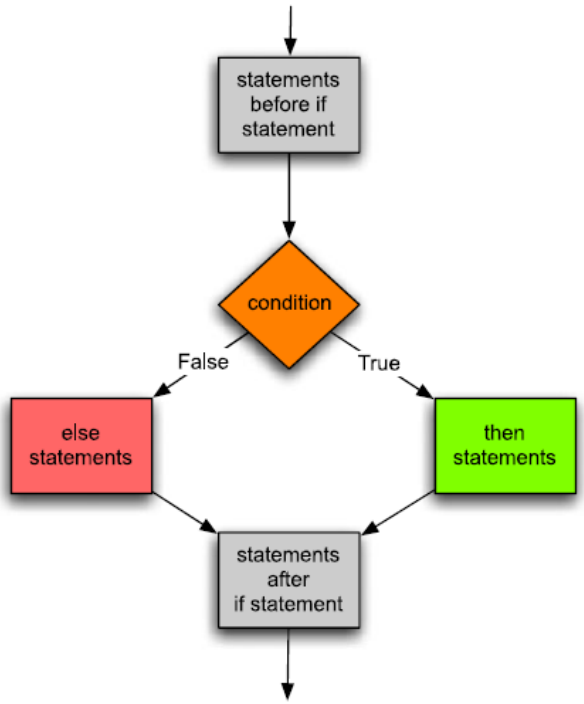


**Figure: if-else Statement**

For instance, you might write a program that reads data from a file and takes one action or another based on the data it read. Or, a program might get some input from a user and then take one of several actions based on that input. To make a choice in Python you write an *if* statement. An if statement takes one of two forms. It may be just an *if* statement. In this case, if the condition evaluates to true then it will evaluate the *then statements*. If the condition is not true the computer will skip to the statements after the if statement.

<statements before if statement>
if <condition>:
    <then statements>
<statements after if statement>

Figure depicts this graphically. An if statement evaluates the conditional expression and then goes to one of two places depending on the outcome. Notice the indentation in the *if* statement above. The indentation indicates the *then statements* are part of the *if* statement. Indentation is very important in Python. Indentation determines the control flow of the program.

If the condition evaluates to true, a detour is taken to execute the *then statements* before continuing on after the *if* statement. Generally, we want to know if some value in our program is equal to, greater, or less than another value. The comparison operators, or relational operators, in Python allow us to compare two values. Any value in your program, usually a variable, can be compared with another value to see how the two values relate to each other.

Sometimes, you may want your program to do one thing if a condition is true and something else if a condition is false. Notice that the if statement does something only when the condition evaluates to true and does not do anything otherwise. If you want one thing to happen when a condition is true and another to happen if the condition is false then you need to use an if-else statement. An if-else statement adds a keyword of else to do something when the condition evaluates to false. An if-else statement looks like this.

<statements before if statement>
if <condition>:
    <then statements>
else:
    <else statements>
<statements after if statement>

If the condition evaluates to true, the then statements are executed. Otherwise, the else statements are executed. Figure depicts this graphically. The control of your program branches to one of two locations, the then statements or the else statements depending on the outcome of the condition.

Again, indentation is very important. The else keyword must line up with the if statement to be properly paired with the if statement by Python. If you don't line up the if and the else in exactly the same columns, Python will not know that the if and the else go together. In addition, the else is only paired with the closest if that is in the same column. Both the then statements and the else statements must be indented and must be indented the same amount. Python is very picky about indentation because indentation in Python determines the flow of control in the program.

In either case, after executing the *if* or the *if-else* statement control proceeds to the next statement after the *if* or *if-else*. The statement after the *if-else* statement is the next line of the program that is indented the same amount as the *if* and the *else*.

***Example**:* Consider a program that finds the maximum of two integers. The last line before the *if-else* statement is the *y* = assignment statement. The first line after the *if-else* statement is the *print*("*Done*.") statement.

```
x = int(input("Please enter an integer: "))
y = int(input("Please enter another integer: "))
i f x > y:
        print(x,"is greater than",y)
e l s e :
        print(y,"is greater than or equal to",x)
print("Done.")
```

**Output:**
**Please enter an integer: 20**
**Please enter another integer: 48**
**48 is greater than or equal to 20**

Example: Finding the Max of Three Integers

```
x = int(input("Please enter an integer: "))
y = int(input("Please enter another integer: "))
z = int(input("Please enter a third integer: "))
i f y > x:
        i f z > y:
                print(z, "is greatest.")
        e l s e :
                print(y, "is greatest.")
e l s e :
        i f z > x:
                print(z, "is greatest.")
        e l s e :
                print(x, "is greatest.")
print("Done.")
```

Output:
**Please enter an integer: 30**
**Please enter another integer:20**
**Please enter a third integer: 5**
**30 is greatest.**
**Done**

The need to select between several choices that Python has a special form of the if statement to handle this. It is the if-elif statement. In this statement, one, and only one, alternative is chosen. The first alternative whose condition evaluates to True is the code that will be executed. All other alternatives are ignored. The general form of the if-elif statement is:

```
<statements before if statement>
if <first condition>:
        <first alternative>
elif <second condition>:
```

```
        <second alternative>
elif <third condition>:
        <third alternative>
else:
        <catch-all alternative>
<statements after the if statement>
```

There can be as many alternatives as are needed. In addition, the else clause is optional so may or may not appear in the statement. If we revise our example using this form of the if statement it looks a lot better. Not only does it look better, it is easier to read and it is still clear which choices are being considered. In either case, if the conditions are not mutually exclusive then priority is given to the first condition that evaluates to true. This means that while a condition may be true, its statements may not be executed if it is not the first true condition in the if statement.

## 1.2 Iteration Statements

Iteration refers to repeating the same thing over and over again. In the case of string sequences, you can write code that will be executed for each character in the string. The same code is executed for each character in a string. However, the result of executing the code might depend on the current character in the string. To iterate over each element of a sequence you may write a for loop. A for loop looks like this:

### for loop

```
<statements before for loop>
for <variable> in <sequence>:
        <body of for loop>
<statements after for loop>
```

In this code the <variable> is any variable name you choose. The variable will be assigned to the first element of the sequence and then the statements in the body of the for loop will be executed. Then, the variable is assigned to the second element of the sequence and the body of the for loop is repeated. This continues until no elements are left in the sequence. If you write a for loop and try to execute it on an empty sequence, the body of the for loop is not executed even once. The for loop means just what is says: for each element of a sequence. If the sequence is zero in length then it won't execute the body at all. If there is one element in the sequence, then the body is executed once, and so on.

For loops are useful when you need to do something for every element of a sequence. Since computers are useful when dealing with large amounts of similar data, for loops are often at the center of the programs we write.

*Example :* Consider the following program.

```
s = input("Please type some characters and press enter:")
for  c in s:
        print(c)
print("Done")
```

**Output:**

If the user enters ***how are you?*** the output is:

```
h
o
w

a
r
e

y
o
u
?
Done
```

Each character of the sequence is printed on a separate line. Notice that there are blank lines, or what appear to be blank lines, between the words. This is because there are space characters between each of the words in the original string and the for loop is executed once for every character of the string including the space characters. Each of these blank lines really contains one space character.

**Example:** Use for loop to find sum of 1 to 5 numbers

```
sum=0
for i in range(1,6):
        sum=sum+i

print("sum of 1 to 5 numbers=",sum)
```

**Output**:
sum of 1 to 5 numbers= 15

**Example:** Use for loop to find the sum of numbers of a list.

```
sum=0
a =[1,2,4,5]
for i in a:
        sum=sum+i
print("sum of  numbers of a list=",sum)
```

**Ouput:**
sum of numbers of a list= 12

## While Loops

Python's while statement is its most general iteration construct. In simple terms, it repeatedly executes a block of indented statements, as long as a test at the top keeps evaluating to a true value. When the test becomes false, control continues after all the statements in the while, and the body never runs if the test is false to begin with.

The while statement is one of two looping statements. We call it a *loop*, because control keeps looping back to the start of the statement, until the test becomes false. The net effect is that the loop's body is executed repeatedly while the test at the top is true. Python also provides a handful of tools that implicitly loop (iterate), such as the map, reduce, and filter functions, and the in membership test;

***General Format***

In its most complex form, the while statement consists of a header line with a test expression, a body of one or more indented statements, and an optional else part that is executed if control exits

the loop without running into a break statement. Python keeps evaluating the test at the top, and executing the statements nested in the
while part, until the test returns a false value.

```
while <test>:                  # loop test
        <statements1>           # loop body
else:                           # optional else
        <statements2>           # run if didn't exit loop with break
```

Example:

```
count = 0
while (count < 9):
        print ('The count is:', count )
        count = count + 1
print ("Good bye!" )
```

output:

```
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
Good bye!
```

Python supports to have an else statement associated with a loop statement. If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list. If the else statement is used with a while loop, the else statement is executed when the condition becomes false

Example:

```
count = 0
while count < 5:
        print (count, " is less than 5" )
        count = count + 1
else:
        print( count, " is not less than 5“)
```

Output:

```
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
```

**Single Statement Suites**: If there is only one executable statement in while loop, then we can use this format.
Example:

```
flag = 1
while (flag): print ('Given flag is really true!' )
print( "Good bye!" )
```

The above program repeatedly prints "Given flag is really true" . To stop use CTRL+C.

## 1.3 Jump Statements

Now that we've seen a few Python loops in action, it's time to take a look at two simple statements that have a purpose only when nested inside loops—the break and continue statements.
In Python:

**pass**
Does nothing at all: it's an empty statement placeholder

**break**
Jumps out of the closest enclosing loop (past the entire loop statement)

**continue**
Jumps to the top of the closest enclosing loop (to the loop's header line)

### 1.3.1 pass

Simple things first: the pass statement is a no-operation placeholder that is used when the syntax requires a statement, but you have nothing useful to say. It is often used to code an empty body for a compound statement. For instance, if you want to code an infinite loop that does nothing each time through, do it with a pass:

```
while True: pass                # Type Ctrl-C to stop me!
```

Because the body is just an empty statement, Python gets stuck in this loop. pass is roughly to statements as None is to objects—an explicit nothing. Notice that here the while loop's body is on the same line as the header, after the colon; as with if statements, this only works if the body isn't a compound statement. This example does nothing forever. A pass is also sometime coded to mean "to be filled in later," to stub out the bodies of functions temporarily:

```
def func1():
        pass            # Add real code here later

def func2():
        pass
```

We can't leave the body empty without getting a syntax error, so we say pass instead. Python 3.X (but not 2.X) allows ellipses coded as ... (literally, three consecutive dots) to appear any place an expression can. Because ellipses do nothing by themselves, this can serve as an alternative to the pass statement, especially for code to be filled in later.

```
def func1():
        ...        # Alternative to pass

def func2():
        ...

func1()          # Does nothing if called
```

Ellipses can also appear on the same line as a statement header and may be used to initialize variable names if no specific type is required:

```
def func1(): ...          # Works on same line too
def func2(): ...
>>> X = ...               # Alternative to None
```

Example:

```
i=1
while(i<=10):
        if(i==6):
                pass
        else:
                print(i)
        i=i+1
```

Output:

1
2
3
4
5
7
8
9
10

Example:

```
i=1
while(i<=10):
        if(i==6):
                print("hello")
                pass
        else:
                print(i)
        i=i+1
```

Output:

1
2
3
4
5
hello
7
8
9
10

### 1.3.2 break

The break statement causes an immediate exit from a loop. Because the code that follows it in the loop is not executed if the break is reached. The break statement can be used in both while and for loops. In nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax: **break**

Example:

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

Output:

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

Example:

```
var = 10
while var > 0:
        print 'Current variable value :', var
        var = var -1
        if var == 5:
                break
```

Output:

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
```

### 1.3.3 continue

The continue statement causes an immediate jump to the top of a loop. It also sometimes lets you avoid statement nesting. The continue statement can be used in both while and for loops. The next example uses continue to skip odd numbers. This code prints all even numbers less than 10 and greater than or equal to 0. Remember, 0 means false and % is the remainder of division (modulus) operator, so this loop counts down to 0, skipping numbers that aren't multiples of 2—it prints 8 6 4 2 0:

```
x = 10
while x:
        x = x−1                          # Or, x -= 1
        if x % 2 != 0: continue          # Odd? -- skip print
        print(x, end=' ')
```

Ouput: 8 6 4 2 0

Because continue jumps to the top of the loop, you don't need to nest the print statement here inside an if test; the print is only reached if the continue is not run.

**Example:**

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

**Output:**

```
Current Letter : P
```

**Example:**

```
var = 10
while var > 0:
        var = var -1
        if var == 5:
                continue
        print 'Current variable value :', var
```

**Output:**

```
Current variable value : 9
```

| | |
|---|---|
| Current Letter : y<br>Current Letter : t<br>Current Letter : o<br>Current Letter : n | Current variable value : 8<br>Current variable value : 7<br>Current variable value : 6<br>Current variable value : 4<br>Current variable value : 3<br>Current variable value : 2<br>Current variable value : 1<br>Current variable value : 0 |

## 2. Functions

In simple terms, a function is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name. Functions also can compute a result value and let us specify parameters that serve as function inputs and may differ each time the code is run. Coding an operation as a function makes it a generally useful tool, which we can use in a variety of contexts.

More fundamentally, functions are the alternative to programming by cutting and pasting—rather than having multiple redundant copies of an operation's code, we can factor it into a single function. In so doing, we reduce our future work radically: if the operation must be changed later, we have only one copy to update in the function, not many scattered throughout the program. Functions are also the most basic program structure Python provides for maximizing code reuse, and lead us to the larger notions of program design. As we'll see, functions let us split complex systems into manageable parts. By implementing each part as a function, we make it both reusable and easier to code.

### Why Use Functions?

Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called subroutines or procedures. As a brief introduction, functions serve two primary development roles:

#### *Maximizing code reuse and minimizing redundancy:*

As in most programming languages, Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Functions allow us to group and generalize code to be used arbitrarily many times later. Because they allow us to code an operation in a single place and use it in many places, Python functions are the most basic factoring tool in the language: they allow us to reduce code redundancy in our programs, and thereby reduce maintenance effort.

#### *Procedural decomposition*

Functions also provide a tool for splitting systems into pieces that have well-defined roles. For instance, to make a pizza from scratch, you would start by mixing the dough, rolling it out, adding toppings, baking it, and so on. If you were programming a pizza-making robot, functions would help you divide the overall "make pizza" task into chunks—one function for each subtask in the process. It's easier to implement the smaller tasks in isolation than it is to implement the entire process at once. In general, functions are about procedure—how to do something, rather than what you're doing it to.

## def Statements

The def statement creates a function object and assigns it to a name. Its general format is as follows:

```
def name(arg1, arg2,... argN):
        statements
```

As with all compound Python statements, def consists of a header line followed by a block of statements, usually indented (or a simple statement after the colon). The statement block becomes the function's body—that is, the code Python executes each time the function is later called.

The def header line specifies a function name that is assigned the function object, along with a list of zero or more arguments (sometimes called parameters) in parentheses. The argument names in the header are assigned to the objects passed in parentheses at the point of call.
Function bodies often contain a return statement:

```
def name(arg1, arg2,... argN):
        ...
        return value
```

The Python return statement can show up anywhere in a function body; when reached, it ends the function call and sends a result back to the caller. The return statement consists of an optional object value expression that gives the function's result. If the value is omitted, return sends back a None. The return statement itself is optional too; if it's not present, the function exits when the control flow falls off the end of the function body.

## def Executes at Runtime

The Python def is a true executable statement: when it runs, it creates a new function object and assigns it to a name. Because it's a statement, a def can appear any where a statement can—even nested in other statements. For instance, although defs normally are run when the module enclosing them is imported, it's also completely legal to nest a function def inside an if statement to select between alternative definitions:

```
if test:
        def func():      # Define func this way
                ...
else:
        def func():      # Or else this way
                ...
...
func()                   # Call the version selected and built
```

One way to understand this code is to realize that the def is much like an = statement: it simply assigns a name at runtime. Unlike in compiled languages such as C, Python functions do not need to be fully defined before the program runs. More generally, defs are not evaluated until they are reached and run, and the code inside defs is not evaluated until the functions are later called. Because function definition happens at runtime, there's nothing special about the function name. What's important is the object to which it refers:

```
othername = func        # Assign function object
othername()             # Call func again
```

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just objects; they are recorded explicitly in memory at program execution time. In fact, besides calls, functions allow arbitrary attributes to be attached to record information for later use:

```
def func(): ...         # Create function object
func()                  # Call object
func.attr = value       # Attach attributes
```

The def statement makes a function but does not call it. After the def has run, you can call (run) the function in your program by adding parentheses after the function's name. The parentheses may optionally contain one or more object arguments, to be passed (assigned) to the names in the function's header:

## Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function:

**Example:**

```
def printme( str ):
        "This prints a passed string into this function"
        print str;
        return;
printme("I'm first call to user defined function!");
printme("Again second call to the same function");
```

**Output**:

```
I'm first call to user defined function!
Again second call to the same function
```

## Passing by Reference Versus Passing by Value:

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

```
def changeme( mylist ):
        "This changes a passed list into this function"
```

```
    mylist.append([1,2,3,4]);
    print ("Values inside the function: ", mylist)
    return
mylist = [10,20,30];
changeme( mylist );
print("Values outside the function: ", mylist)
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result:

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist = [1,2,3,4]; # This would assig new reference in mylist
    print ("Values inside the function: ", mylist)
    return
mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result:

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

## 2.1 Function Arguments

You can call a function by using the following types of formal arguments:

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

### *1. Required arguments*

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows:

```
def printme( str ):
        "This prints a passed string into this function"
        print str;
        return;

        # Now you can call printme function
printme();
```

When the above code is executed, it produces the following result:

```
Traceback (most recent call last):
File "test.py", line 11, in <module>
printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

### 2. Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways:

```
def printme( str ):
        "This prints a passed string into this function"
        print str;
        return;
# Now you can call printme function
printme( str = "My string");
```

When the above code is executed, it produces the following result:
**My string**

The following example gives more clear picture. Note that the order of parameters does not matter.

```
def printinfo( name, age ):
        "This prints a passed info into this function"
        print "Name: ", name;
        print "Age ", age;
        return;
# Now you can call printinfo function
printinfo( age=50, name="miki" );
```

When the above code is executed, it produces the following result:

```
Name: miki
```

Age 50

### 3. 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. The following example gives an idea on default arguments, it prints default age if it is not passed:

```
def printinfo( name, age = 35 ):
        "This prints a passed info into this function"
        print "Name: ", name;
        print "Age ", age;
        return;
# Now you can call printinfo function
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

When the above code is executed, it produces the following result:
Name: miki
Age 50
Name: miki
Age 35

### 4. Variable-length arguments

You may need to 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.
Syntax for a function with non-keyword variable arguments is this:

```
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. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example:

```
def printinfo( arg1, *vartuple ):
        "This prints a variable passed arguments"
        print "Output is: "
        print arg1
        for var in vartuple:
                print var
        return;

# Now you can call printinfo function
printinfo( 10 );
printinfo( 70, 60, 50 );
```

When the above code is executed, it produces the following result:
Output is:
10

Output is:
70
60
50

## 2.2 Anonymous Functions

Besides the def statement, Python also provides an expression form that generates function objects. Because of its similarity to a tool in the Lisp language, it's called lambda. Like def, this expression creates a function to be called later, but it returns the function instead of assigning it to a name. This is why lambdas are sometimes known as anonymous (i.e., unnamed) functions. In practice, they are often used as a way to inline a function definition, or to defer execution of a piece of code.

### *lambda Basics*

The lambda's general form is the keyword lambda, followed by one or more arguments (exactly like the arguments list you enclose in parentheses in a def header), followed by an expression after a colon:

lambda argument1, argument2,... argumentN : expression using arguments

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs, but there are a few differences that make lambdas useful in specialized roles:
• lambda is an expression, not a statement. Because of this, a lambda can appear in places a def is not allowed by Python's syntax—inside a list literal or a function call's arguments, for example. With def, functions can be referenced by name but must be created elsewhere. As an expression, lambda returns a value (a new function) that can optionally be assigned a name. In contrast, the def statement always assigns the new function to the name in the header, instead of returning it as a result.

• lambda's body is a single expression, not a block of statements. The lambda's body is similar to what you'd put in a def body's return statement; you simply type the result as a naked expression, instead of explicitly returning it. Because it is limited to an expression, a lambda is less general than a def—you can only squeeze so much logic into a lambda body without using statements such as if. This is by design, to limit program nesting: lambda is designed for coding simple functions, and def handles larger tasks.
Apart from those distinctions, defs and lambdas do the same sort of work. For instance, we've seen how to make a function with a def statement:

```
>>> def func(x, y, z): return x + y + z
>>> func(2, 3, 4)
9
```

But you can achieve the same effect with a lambda expression by explicitly assigning its result to a name through which you can later call the function:

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Here, f is assigned the function object the lambda expression creates; this is how def works, too, but its assignment is automatic. Defaults work on lambda arguments, just like in a def:

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

```
>>> def knights():
        title = 'Sir'
        action = (lambda x: title + ' ' + x) # Title in enclosing def scope
        return action # Return a function object

>>> act = knights()
>>> msg = act('robin') # 'robin' passed to x
>>> msg
'Sir robin'
```

## *Why Use lambda?*

Generally speaking, lambda comes in handy as a sort of function shorthand that allows you to embed a function's definition within the code that uses it. They are entirely optional—you can always use def instead, and should if your function requires the power of full statements that the lambda's expression cannot easily provide—but they tend to be simpler coding constructs in scenarios where you just need to embed small bits of executable code inline at the place it is to be used.

To nest selection logic in a lambda, you can use the if/else ternary expression. As you learned earlier, the following statement:

```
if a:
b
else:
c
```

can be emulated by either of these roughly equivalent expressions:

```
b if a else c
((a and b) or c)
```

Because expressions like these can be placed inside a lambda, they may be used to implement selection logic within a lambda function:

```
>>> lower = (lambda x, y: x if x < y else y)
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
```

'aa'

Furthermore, if you need to perform loops within a lambda, you can also embed things like map calls and list comprehension expression.

### *Scopes: lambdas Can Be Nested Too*

lambdas are the main beneficiaries of nested function scope lookup. As a review, in the following the lambda appears inside a def—the typical case—and so can access the value that the name x had in the
enclosing function's scope at the time that the enclosing function was called:

```
>>> def action(x):
        return (lambda y: x + y) # Make and return function, remember x

>>> act = action(99)
>>> act
<function action.<locals>.<lambda> at 0x00000000029CA2F0>

>>> act(2) # Call what action returned
101
```

A lambda also has access to the names in any enclosing lambda. This case is somewhat obscure, but imagine if we recoded the prior def with a lambda:

```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

Here, the nested lambda structure makes a function that makes a function when called. In both cases, the nested lambda's code has access to the variable x in the enclosing lambda. This works, but it seems fairly convoluted code; in the interest of readability, nested lambdas are generally best avoided.

## 2.3 Fruitful Functions (Function Returning Values)

The function which returns values are called the fruitful function. 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.

You can return a value from a function as follows:

```
def sum( arg1, arg2 ):
    # Add both the parameters and return them.
    total = arg1 + arg2
    print "Inside the function : ", total
    return total;
```

```
# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result:
Inside the function : 30
Outside the function : 30

### *void function*

```
def sum( arg1, arg2 ):
        # Add both the parameters and return them.
        total = arg1 + arg2
        print "Inside the function : ", total

# Now you can call sum function
sum( 10, 20 );
```

When the above code is executed, it produces the following result:
Inside the function : 30

## 2.4 Variable Scope

The scope of an identifier is defined to be the portion of the program where its declaration applies, or what we refer to as "variable visibility." In other words, it is like asking yourself in which parts of a program do you have access to a specific identifier. Variables either have local or global scope.

### Global versus Local Variables

Variables defined within a function have local scope, and those at the highest level in a module have global scope. "The portion of the program to which a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be local to the procedure if it is in the scope of a declaration within the procedure; otherwise, the occurrence is said to be nonlocal."
One characteristic of global variables is that unless deleted, they have a lifespan that lasts as long as the script that is running and whose values are accessible to all functions, whereas local variables, like the stack frame they reside in, live temporarily, only as long as the functions they are defined in are currently active. When a function call is made, its local variables come into scope as they are declared. At that time, a new local name is created for that object, and once that function has completed and the frame deallocated, that variable will go out of scope.

```
global_str = 'foo'
def foo():
        local_str = 'bar'
        return global_str + local_str
```

In the above example, global_str is a global variable while local_str is a local variable. The foo() function has access to both global and local variables while the main block of code has access only to global variables.

When searching for an identifier, Python searches the local scope first. If the name is not found within the local scope, then an identifier must be found in the global scope or else a NameError exception is raised. It is possible to "hide" or override a global variable just by creating a local one. Recall that the local namespace is searched first, being in its local scope. If the name is found, the search does not continue to search for a globally scoped variable, hence overriding any matching name in either the global or built-in namespaces.
Also, be careful when using local variables with the same names as global variables. If you use such names in a function (to access the global value) before you assign the local value, you will get an exception (NameError or UnboundLocalError), depending on which version of Python you are using.

## global Statement

Global variable names can be overridden by local variables if they are declared within the function. Here is another example, similar to the first, but the global and local nature of the variable are not as clear.

```
def foo():
        print "\ncalling foo()..."
        bar = 200
        print "in foo(), bar is", bar
bar = 100
print "in __main__, bar is", bar
foo()
print "\nin __main__, bar is (still)", bar
```

It gave the following output:

```
in __main__, bar is 100
calling foo()...
in foo(), bar is 200
in __main__, bar is (still) 100
```

Our local bar pushed the global bar out of the local scope. To specifically reference a named globalvariable, one must use the global statement. The syntax for global is:

**global var1[, var2[, ... varN]]]**

Modifying the example above, we can update our code so that we use the global version of is_this_global rather than create a new local variable.

```
is_this_global = 'xyz'
def foo():
   global is_this_global
   this_is_local = 'abc'
   is_this_global = 'def'
   print (this_is_local + is_this_global)
foo()
print( is_this_global)
```

**Output:**

```
abcdef
def
```

## Number of Scopes

Python syntactically supports multiple levels of functional nesting, and as of Python 2.1, matching statically nested scoping. However, in versions prior to 2.1, a maximum of two scopes was imposed: a function's local scope and the global scope. Even though more levels of functional nesting exist, you could not access more than two scopes:

```
def foo():
        m = 3
        def bar():
                n = 4
                print( m + n)
        print(m)
        bar()
foo()
```

Although this code executes perfectly fine today ...

```
>>> foo()
37
```

... executing it resulted in errors in Python before 2.1:

```
>>> foo()
Traceback (innermost last):
File "<stdin>", line 1, in ?
File "<stdin>", line 7, in foo
File "<stdin>", line 5, in bar
NameError: m
```

The access to foo()'s local variable m within function bar() is illegal because m is declared local to foo(). The only scopes accessible from bar() are bar()'s local scope and the global scope. foo()'s local scope is not included in that short list of two. Note that the output for the "print m" statement succeeded, and it is the function call to bar() that fails. Fortunately with Python's current nested scoping rules, this is not a problem today.