x**y

Lecture notes on

Computer Programming & Software

PS03EMTH32/PS04EMTH32

x=2; x*=3

JAY MEHTA
Department of Mathematics,
Sardar Patel University.

\n

def f(x):

global x

SEMESTER - IV
2018-19

Preface and Acknowledgments

These are the lecture notes of the course ‘Computer Programming and Software’ which covers theory part of Python programming offered to the M.Sc. (Semester - III/Semester - IV) students at Department of Mathematics, Sardar Patel University, 2018-19. These notes are tailored for the Computer Programming and Software (PS03EMTH32/PS04EMTH32) syllabus of M.Sc. (Semester-III/Semester-IV) of the University and do not cover all the topics of Python.

These are prepared from the recommended reference books and other available literature. We completely followed the free e-book “A Byte of Python” by Swaroop, C. H. and no part of it is my original material.

JAY MEHTA

Contents

Syllabus

PS04EMTH32: Computer Programming & Software

Note: 50 Marks (35 marks for external examination and 15 marks for the internal examination) for theory and 50 Marks for practical on computers. External examination will be of two hours for theory and three hours for practical.

Unit I:	The Basics: Literal constants, numbers, strings, variables, identifier naming, data types, objects, logical and physical lines, indentation. Operators, operator precedence, expressions. Control flow: the if statement, the while statement, the for loop, the break statement, the continue statement.
Unit II:	Functions: Defining a function, local variables, default argument values, keyword arguments, the return statement, DocStrings. Modules: using the sys module, the from import statement, creating modules, the dir() function.
Unit III & IV:	List of computer practical.

No.	Purpose of the program
1	To find the minimum/maximum of a given list of numbers.
2	To check whether a given number is odd or even. To check whether a given year is a leap year or not.
3	To find the real roots of a quadratic equation.
4	To compute $n!$, a^n, sum and average of a list of numbers. To prepare the result of a student.
5	Primality lists: To check whether a given number is prime or not, to list all the prime numbers within a given range, to factorize a number.
6	Manipulation of numbers: to check whether a given number is perfect or not, to check whether a given number is palindrome or not, to compute the sum of digits of a given number, to compute the sum of squares of the digits, to print a given number in reverse order of its digits.
7	To compute GCD and LCM of two numbers, to evaluate the functions $\sigma(n)$, $\tau(n)$, $\phi(n)$, $\mu(n)$ for a given positive integer n.
8	To generate Fibonacci sequence and Lucas sequences; to compute the sum of the series and hence evaluate e^x, $\sin(x)$, $\cos(x)$, $\tan(x)$, $\sinh(x)$, $\cosh(x)$ (terminate the program after n terms of the series or terminate the program at the desired level of accuracy).
9	Basics of Scilab 1: Sum of matrices, determinant of a matrix, product of matrices, inverse of a matrix, row reduced echelon form.
10	Basics of Scilab 2: Plotting Cartesian, polar and parametric curves, commands for plotting functions.

Text Book

1. Swaroop C. H., A byte of Python.
   Chapters: 4, 5, 6, 7, 8.

Reference Books

1. James Payne, Beginning Python: Using Python 2.6 and Python 3, Wiley India, 2010.
2. Amit Saha, Doing Math with Python, No Starch Press (2015).
3. SCILAB - A Free software to MATLAB by Er. Hema Ramachandran and Dr. Achuthsankar S. Nair., S. Chand and Company Ltd. (2008).

CHAPTER 1

Basics of Python and Control Flow

1.1 Basics

1.1.1 Comments

Any text to the right of the # symbol is called a *comment*. Comments are not printed or executed during the execution of a Python program. They are mainly used as information or notes for the reader of the program. For example,

```
print('Namaste! world') # Note that print is a function
```

or

```
# Note that print is a function
print('Namaste! world')
```

In above example the comment is `Note that print is a function`.

The following are some of the uses of comments in the program:

- to explain assumptions
- to explain important decisions
- to explain important details
- to explain problems you're trying to solve
- to explain problems you're trying to overcome in your program, etc.

1.1.2 Literal Contants

An example of a literal constant is a number like `5` , `1.23` , or a string like `‘This is a string’` or `“It’s a string!”` .

It is called a literal because it is indeed literal, i.e. one uses its value literally. The number `2` always represents itself, i.e. the value 20 and nothing else. It is a constant because its value does not change. Thus, all these are referred to as literal constants.

Numbers

Numbers are mainly of two types:

1. Integers
   An example of an integer is `2` which is just a whole number.

2. Floats
   Examples of floating point numbers (or floats) are `3.23` and `52.3E-4` . The `E` notation indicates powers of 10. In this case, `52.3E-4` means `52.3 * 10 ^ -4`, i.e. 52.3×10^{-4}.

Strings

A string is a sequence of characters. It is just a collection of words. In Python, strings are specified in the following three ways.

Single Quote A string can be specified using single quotes as `‘This is a string’` . All the white spaces, tabs, symbols, etc. within the quotes are preserved as it is.

Double Quotes Strings can be specified using double quotes in the same way as single quotes. For example, `“What’s your name?”` .

Triple Quotes Triple quotes are used to specify a multi-line string. In triple quotes the quote can either be single or double, i.e. `"""` or `’’’` . For example

```
“‘This is a multi-line string. This is the first line.
This is the second line.
“What’s your name?”, I asked.
He replied “Bond, James Bond.”
’’’
```

This is same as

```
"""This is a multi-line string. This is the first line.
This is the second line.
“What’s your name?”, I asked.
He replied “Bond, James Bond.”
"""
```

1.1.3 The format method

The `format( )` method is used to construct strings from other information. Consider the following example.

```
rollno = 23
name = ‘Shailesh’
sem = 3
print(‘{0} is in Semester {1} and his roll number is {2}’.format(name, sem, rollno))
```

Output:

```
Shailesh is in Semester 3 and his roll number is 23
```

How it works:
A string can use certain specifications for which the `format` method can be used. The `format( )` command can be called to substitute those specifications with corresponding arguments to the `format` method.

Observe that in the first usage we use `{0}` and this corresponds to the variable `name` which is the first argument to the format method. Similarly, the second specification is `{1}` corresponding to the variable `sem` which is the second argument to the format method. Finally, the third specification is `{2}` which corresponds to the value of the variable `rollno` which is the final argument in the format command.

Observe that in Python, the counting or the numbering starts from 0, i.e. that first position is at index 0, second position is at index 1, and so on . . ..

Note that the above can also be achieved by adding strings as follows:

```
name + ‘ is ’ + str(age) + ‘ years old’
```

The advantages of using format method over the above method are:

- The chances of error with format method is minimal.

- The conversion to string would be done automatically by the format method instead of the explicit conversion to strings needed in this case (i.e. `str(age)` here).
- With the format method, the message can be changed easily without dealing with the variables used and vice-versa.

Note that the numbers written in curly braces in the above example are optional, i.e. we can also write as:

```
rollno = 23
name = 'Shailesh'
sem = 3
print('{} is in Semester {} and his roll number is {}'.format(name, sem, rollno))
```

This too will give exactly the same output as in the case of previous example. In format method, Python substitutes the value of each argument into the place of its specification. Some of the more detailed specifications are as follows:

```
# decimal (.) precision of 3 for float '0.333'
print('{0:.3f}'.format(1.0/3))
# fill with underscores (_) with the text at the center
# ( ^ ) to 11 width '___hello___'
print('{0:_^11}'.format('hello'))
# keyword-based 'Semester 4 is offered the Computers course'
print('{sem} is offered the {c} course'.format(sem='Swaroop', c='Computers'))
```

Output:

```
0.333
___hello___
Semester 4 is offered the Computers course
```

There are other methods of formatting too which do not use the format command. For example formatting the ending of a print statement.

1.1.4 Ending of a print statement

Note that the `print` statement always ends with an invisible 'new line' character (`\n`). So, repeated calling of `print` will give output each on a separate line. To avoid this default new line character from being printed, we can specify how a print statement in Python should end. To `end` a print statement with a a blank as:

```
print('a', end='')
print('b', end='')
```

The output will be

```
ab
```

We can also `end` it by a comma or a space as:

```
print('a', end=', ') # Note that there is a white space after comma
print('b', end=', ') # Note that there is a space after comma
print('c', end=' ') # Only one white space is given
print('d')
```

Output:

```
a, b, c  d
```

1.1.5 Escape Sequences

Suppose we want to print a string which contains a single quote (`'`). For example, the string is: `What's your name?`. Certainly, we cannot specify it with single quotes as Python will give error regarding starting and ending of the string. So, we have to specify the string in a way it indicates that the single quote in `What's` is not the end of the string. Of course, one way to do this is using a double quotes as: `"What's your name?"`. However, this can be done in a better way using what is called an *escape sequence*. The single can be specified quote as `\'`, i.e. `'What\'s your name?'`.

Similarly, a string having double quotes can be either specified by single quotes or by using an escape sequence for double quotes which is `\"`.

The following are some (not all) of the escape sequences and their use:

- `\\` - To specify the backslash itself using the escape sequence.
- `\n` - To specify the beginning of a new line.

  Suppose we want to specify a two-line (or in general a multi-line) string. One way, we know, to do this is using triple quotes. Another way to do this is using the escape sequence of the new line character - `\n` which indicates the start of a new line or the next line. An example is:

```
'This is the first line\nThis is the second line'
```

- `\t` - To specify tab or white spaces given by tab.
- `\b` - To specify backspace.

For example,

```
print('Hello\b\b')
print('Hello\t again')
print('Hello\nagain')
print('Hello\n again')
```

gives the following output:

```
Hel
Hello      again
Hello
again
Hello
 again
```

There are many other escape sequences but only few are listed here.

Single backslash

Note that, in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added. For example

```
"This is the first sentence. \
This is the second sentence."
```

is equivalent to

```
"This is the first sentence. This is the second sentence."
```

1.1.6 Raw String

Raw string is used to specify a string as it is, i.e. without any processing such as handling of escape sequences, etc. A raw string is specified by prefixing `r` or `R` to the string. For example,

```
print(r"A line break is indicated by \n which is an escape sequence")     # printing a raw sting
print("A line break is indicated by \n which is an escape sequence")     # printing a usual string
```

gives the following output:

```
A line break is indicated by \n which is an escape sequence
A line break is indicated by
 which is an escape sequence
```

1.1.7 Variables

We need some way to store information and then manipulate them when we want. For this purpose, variables are used. Variables are exactly as their names implies, i.e. their values can vary. Variables are just parts of computer's memory where we store some information. Unlike literal constants, we need some method of assessing these variables and hence we give them names.

Identifier Naming

Identifiers are names given to identify something. Variables are examples of identifiers. The following are some rules for naming the identifiers:

- The first character of the identifier must be a letter of the alphabet (uppercase ASCII or lowercase ASCII or Unicode character) or an underscore (`_`).
- The rest of the identifier name can consist of letters (uppercase ASCII or lowercase ASCII or Unicode character), underscores (`_`) or digits ($0 - 9$).
- Identifier names are case-sensitive. For example, `myname` and `myName` are not the same.
- Examples of valid identifier names are `i`, `name_2_3`, `_007`, `jAy`. Examples of invalid identifier names are `2things`, `007bond`, `my-name`, `my name` and `>a1b2_c3`.

Data Types

Variables can hold values of different types called *data types*. The basic data types are numbers and strings.

Objects

Python refers anything used in a program as an *object*. Instead of saying "the something" we say "the object". For example, variables, constants, functions, etc. all are referred to as objects.

1.1.8 Logical and Physical Lines

A *physical line* is what we see (as a single line) when we write a program. A *logical line* is what Python sees as a single statement. Python implicitly assumes that each physical line corresponds to a logical line. An example of logical line is a statement like `print(“Hello world”)`. If this statement was on a line by itself (as seen in Python editor) then it also corresponds to a physical line. Implicitly Python encourages use of a single statement per line, which makes the code more readable.

If one wants to write more than one logical line on a single physical line, then this has to be explicitly specified using a semicolon(`;`) which indicates the end of a logical line. For example,

```
i = 5
print(i)
```

is effectively same as

```
i = 5; print(i);
```

or

```
i = 5; print(i)
```

or

```
i = 5;
print(i);
```

A special situation where this concept is very useful is the following. Suppose we have a long line of codes. Then we can break it into multiple physical lines by using the backslash (`\`). This is called *explicit line joining*. For example,

```
s = ‘This is a string. \
This continues the string.’
```

Output:

```
This is a sting. This continues the sting.
```

Similarly,

```
i = \
5
```

is same as

```
i = 5
```

Sometimes there is an implicit assumption where we do not need to use a backslash. This is the case where the logical lines has starting parentheses, starting square brackets or a starting curly braces but not an ending one. This is called *implicit line joining*.

1.1.9 Indentation

Whitespace at the beginning of a line in a Python program is called *indentation*.

Leading whitespaces (spaces or tabs) at teh beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements. This means that statements which go (or execute) together have the same indentation. Each such set of statements is called a *block*.

Note that wrong indentation can result into errors. For example, consider the following program with filename “whitespace.py”.

```
i = 5
 print(‘Value of i is ’, i)    # Note the single space at the beginning of line
print(‘I repeat, the value of i is’, i)
```

When we run this program, we get the following error:

```
File "whitespace.py", line 3
    print(‘Value is’, i)
    ^
IndentationError: unexpected indent
```

Note that there is a single whitespace at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid.

1.1.10 Operators and Expressions

A simple example of an expression is $2 + 3$. An expression can be broken down into operators and operands.

Operators are functionality that do something and can be represented by symbols such as $+$, $-$ or by special keywords. Operators require some data to operate on and such data is called *operands*. In the above example 2 and 3 are the operands and the symbol $+$ is the operator.

The following are some of the operators available in Python:

- $+$ (plus/addition)
  - Adds two objects (numbers or strings).
  - For example, $3 + 5$ gives 8, 'Jay' + 'Mehta' gives 'JayMehta'.
- $-$ (minus/subtraction)
  - Gives the subtraction of one number from the other. If the first operand is absent it is assumed to be zero.
  - For example, -5.2 or $0 - 5.2$ gives negative number -5.2, $50 - 26$ gives 24.
- $*$ (multiply)
  - Gives the multiplication of two numbers or returns the string repeated that many times.
  - For example, $2 * 3$ gives 6, 'la'$*3$ gives 'lalala'.
- $**$ (power)
  - Returns x to the power y, i.e. x^y.
  - For example, $2 * *3$ gives 8, $3 * *4$ gives 81.
- / (divide)
  - Divide x by y, i.e. $\frac{x}{y}$.
  - For example, $13/3$ gives 4.333333333333333, 'la'$*3$ gives 'lalala'.
- // (divide and floor)
  - Divide x by y are round the answer down to the nearest integer value. Note that if one of the values is a float, then it returns a float.
  - For example, $13//3$ gives 4, $-13//3$ gives -5, $9//1.81$ gives 4.0.
- $\%$ (modulo)
  - Returns the remainder of the division.
  - $13 \% 3$ gives 1, $-25.5 \% 2.25$ gives 1.5.

- `<<` (left shift)
  - Shifts the bits of the number to the left by the number of bits specified.
  - `2 << 2` gives `8`. 2 is expressed in binary as 10 which when shifted to left by 2 bits gives 1000 which is binary of 8.
- `>>` (right shift)
  - Shifts the bits of the number to the right by the number of bits specified.
  - `11 >> 1` gives `5`. 11 is expressed in binary as 1011 which when shifted to right by 1 bit gives 101 which is binary of 5.
- `&` (bit-wise AND)
  - Gives bit-wise AND of numbers.
  - `5 & 3` gives `1`. 5 in binary is 101 and 3 is binary is 011. Only the bit in the unit place is 1 for both representations and hence bitwise AND will give 001 which is binary of 1.
- `|` (bit-wise OR)
  - Gives bit-wise OR of numbers.
  - `5 | 3` gives `7`.
- `^` (bit-wise XOR)
  - Gives bit-wise XOR of numbers.
  - `5 ^ 3` gives `6`.
- `<` (less than)
  - Returns whether x is less than y.
  - All comparison operators returns `True` or `False`.
  - `5 < 3` gives `False`. `3 < 5` gives `True`.
  - Comparisons can be chained arbitrarily. `3 < 5 < 7` gives `True`. `5 < 3 < 7` gives `False`.
- `>` (greater than)
  - Returns whether x is greater than y.
  - `5 > 3` gives `True`. `3 > 5` gives `False`.
- `<=` (less than or equal to)
  - Returns whether x is less than or equal to y.
  - `x=3; y=5; x <=y` returns `True`.
- `>=` (greater than or equal to)
  - Returns whether x is greater than or equal to y.

  - x=5; y=3.0; x >=y returns True.

- $==$ (equal to)
  - Compares if two objects are equal.
  - $x = 2; y = 2.0; x == y$ returns True.
  - $x =$‘str’; $y =$‘Str’; $x == y$ returns False. $x =$‘str’; $y =$‘str’; $x == y$ returns True.

- $! =$ (not equal to)
  - Compares if two objects are not equal.
  - x=2; y=2.0; x !=y returns False. x=2; y=3; x !=y returns True.
  - x=‘str’; y=‘Str’; x !=y returns True. x = ‘str’; y=‘str’; x !=y returns False.

- not (boolean NOT)
  - If x is True, it returns False and if x if False, it returns True.
  - For example, x = True; not x returns False.

- and (boolean AND)
  - x and y returns False if x is False, else it returns the evaluation of y.
  - x = False; y = True; x and y returns False. In this case Python will not evaluate y as it knows that the left hand side of ‘and’ expression is False and hence the whole expression is False irrespective of the other values. This is called *short-circuit evaluation*.

- or (boolean OR)
  - If x is True, it returns True. If x is False, it returns the evaluation of y.
  - x = False; y = True; x or y returns True. Short-circuit evaluation applies here also.

Shortcut for math operation and assignment

It is common practice in programming to run a math operation on a variable and then assign the result of the operation back to the same variable. Hence, there is a shortcut for such expressions.

```
a = 2
a = a * 3
```

can be written as

```
a = 2
a *= 3
```

Note that `var = var operation expression` becomes `var operation= expression`.

Order of evaluation

If we have an expression like $2 + 3 * 4$, which operation is carried out at first? We know that in mathematics, multiplication operator has higher precedence (preference) than the addition operator.

The following table gives the precedence order for Python from the lowest precedence (i.e. least binding) to the highest precedence (i.e. most binding). This means that in a given expression, Python will first evaluate the operators and expression lower in the table given below before the ones listed higher in the table.

No.	Operators	Purpose/Functioning
1.	lambda	Lambda Expression
2.	if - else	Conditional expression
3.	or	Boolean OR
4.	and	Boolean AND
5.	not x	Boolean NOT
6.	in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
7.	|	Bitwise OR
8.	^	Bitwise XOR
9.	&	Bitwise AND
10.	<<, >>	Shifts
11.	+, -	Addition and subtraction
12.	*, /, //, %	Multiplication, Division, Floor Division and Remainder
13.	+x, -x, ~x	Positive, Negative, bitwise NOT
14.	**	Exponentiation
15.	x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
16.	(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or tuple display, list display, dictionary display, set display

Operators with the same precedence are listed in the same row in the above table. For example, `+` and `-` have the same precedence.

Changing the order of evaluation

To make an expression more readable we can use parentheses. For example, $2 + (3 * 4)$ is definitely easier to understand than $2 + 3 * 4$. There is an additional advantage of using parentheses, it helps to change the order of evaluation. For example, if we want the addition to be evaluated before multiplication, then we can write $(2+3)*4$.

Associativity

Operators are usually associated from left to right. This means that operators with the same precedence are evaluated in a left to right manner. For example, $2+3+4$ is evaluate as $(2+3)+4$.

1.1.11 Expressions

To understand the working of expressions in a Pythong program, consider the following program with file name `expressions.py`:

```
length = 5
breadth = 2
area = length * breadth
print('Area is ',area)
print('Perimeter is ',2 * (length + breadth))
```

Output:

```
Area is 10
Perimeter is 14
```

How it works:
The length and the breadth of the rectangle are stored in variables by the same name. We store the results of the expression of `length * breadth` in the variable `area` and then print it using the `print` function. In the second case, we directly use the value of the expression `2 * (length + breadth)` in the print function.

1.2 Control Flow

In the programs, a series of statements are faithfully executed by Python in exact top to down order. Suppose we want to change or control the flow of the execution of a Python program. For example, taking some decisions in the program or do different things depending on different situations. This can be done using *control flow*. There are three types of control flow statements in Python. They are `if`, `for` and `while`.

1.2.1 The `if` statement

The `if` statement is used to check a condition: *if* the condition is true, we run a block of statements called the *if-block*, else we process another block called the *else-block*. The *else* clause is optional. Consider the following example.

Example `if.py`

```
num = 23
guess = int(input('Enter an integer: '))

if guess == num:
    # New block starts here
    print('Congratulations, you guessed it.')
    # New block ends here

elif guess < num:
    # Another block
    print('No, it is little higher than that.')

else:
    print('No, it is little lower than that.')
    # One must have guessed a greater number to reach here

print('Done') # This last statement will always be executed after the if statement is executed
```

Output:

```
Enter an integer: 50
No, it is a little lower than that.
Done

Enter an integer: 22
No, it is a little higher than that.
Done

Enter an integer: 23
Congratulations, you guessed it.
Done
```

How it works:
In this program, we take guesses from the user and check if it is the number that we have. We set the variable `num` to any integer we want, say 23. Then we take the user's guess using the `input()` function. We supply a string to the built in `input` function which prints it to the screen and waits for input from the user. Whatever the user enters is returned by the `input()` function as a string. It is converted to integer using `int` and then stored in the variable `guess`.

Next we compare the guess of the user with the number we have chosen. If they are equal we print a success message. Note that we use indentation levels to tell Python which statements belong to which block. Also note that we use a colon at the end of the `if` statement indicating that a block of statement follows.

Then we check if the guess is less than number, and if so, we inform the user that they must guess a little higher. Here we use `elif` clause which combines two `if else-if else` statements into one command.

The `elif` and `else` statements must also have a colon at the end of the logical line followed by their corresponding block of statements. We can use another `if` statement inside the if-block of and `if` statement and so on. This is called a *nested* `if` statement. Note that the `elif` and `else` parts are optional.

After Python has finished executing the complete `if` statement with associated `elif` and `else` clauses, it moves to the next statement in the block containing the `if` statement. In this case, it is the main block and the next statement is `print('Done')`.

1.2.2 The `while` statement

The `while` statement allows us to repeatedly execute a block of statements as long as a condition is true. A `while` statement is an example of what is called *looping* statement. A `while` statement can also have an optional `else` clause.

Example `while.py`

```
num = 23
run = True

while run:
    # This is while block
    guess = int(input('Enter an integer: '))

    if guess == num:
        # This is if block
        print('Congratulations, you guessed it.')
        # if block ends here
        run = False # This stops the while loop
    elif guess < num:
        print('No, it is little higher than that.')
    else:
        print('No, it is little lower than that.')
else:
    print('The while loop is over.')
print('Done')
```

Output:

```
Enter an integer: 50
No, it is little lower than that.
Enter an integer: 22
No, it is little higher than that.
Enter an integer: 23
Congratulations, you guessed it.
The while loop is over.
Done
```

How it works:
In this program, the advantage is that the user is allowed to keep guessing until the guess is correct. Thus, there is no need to repeatedly run the program for each guess. This shows the significance of the `while` statement.

We write `input` and `if` statements inside the `while` loop and set the variable `run` to `True` before the `while` loop. First, the `while` statement checks if the variable `run` is `True` and then proceeds to execute the corresponding *while-block*. After the block is executed, the condition is again checked which in this case is the `run` variable. If it is true, we execute the while-block again, else we continue to execute the optional *else-block* and then proceed to the next statement.

The *else-block* is executed when the `while` loop condition becomes `False`. This is true even for the first time when the condition is checked. If there is an `else` clause for a `while` loop, it is always executed unless we break out of the loop with a `break` statement.

The `True` and `False` are called Boolean types and we can consider them to be equivalent to 1 and 0 respectively.

1.2.3 The `for` loop

The `for..in` statement is another example of a looping statement which iterates over a sequence of objects, i.e., it goes through each item in a sequence. Consider the the following program involving `for` loop.

Example `for.py`

```
for i in range(1,5):
    print(i)
else:
    print('The for loop is over.')
```

Output:

```
1
2
3
4
The for loop is over.
```

How it works:
In this program, we are printing a sequence of number. We generate this sequence of numbers using the built-in `range` function.

We supply two numbers to the `range( )` function and it returns a sequence of numbers starting from the first number up to the second number, i.e. it does not include the second number. For example, `range(1,5)` gives the sequence [1, 2, 3, 4]. The range functions take the step count of 1 by default. If we supply a third argument (number) to range, then it becomes the step count. For example, `range(1,5,2)` gives [1, 3]. Note that `range( )` generates only one number at a time. To get the full list of numbers, we can call `list( )` on `range( )`. For example, `list(range(5))` will give [0, 1, 2, 3, 4]. Note that here only one argument 5 is supplied to range. It is the ending value. By default, `range` starts from 0 (as we know that Python starts from 0).

The `for` loop iterates over this range. The statement `for i in range(1,5)` is equivalent to `for i in [1, 2, 3, 4]` which is like assigning each number (or object) in the sequence to the variable i, one at a time, and then executing the block of statements for each value of i. In this case, we print the value of i in the block of statements.

Note that the `else` part is optional. It is always executed once after the `for` loop is over unless we break out of the loop using a `break` statement.

The `for..in` loop works for any sequence. Here we have a list of numbers generated by built-in function. However, in general we can use any kind of sequence of any kind of objects.

1.2.4 The `break` statement

The `break` statement is used to break out of a loop statement, i.e. to stop the execution of a looping statement even if the loop condition has not become `False` or the sequence of items has not be completely iterated over.

Note that if we break out of a `for` or a `while` loop, any corresponding loop `else-block` is not executed.

Example `break.py`

```
while True:
    s = input('Enter something: ')
    if s == 'quit':
        break
    print('Length of the string is ', len(s))
print('Done')
```

Output:

```
Enter something: Programming is fun
Length of the string is 18
Enter something: Use Python
Length of the string is: 11
Enter something: quit
Done
```

How it works: In this program, we repeatedly take user's input and print the length of each input. Each time we are providing a special condition to stop the program by checking if the user input is `'quit'`. We stop the program by breaking out of the `while` loop and reach the end of the program.

The length of the input string can be found using the built-in `len()` function. Also note that, `break` statement can be used with the `for` loop also.

1.2.5 The `continue` statement

The `continue` statement is used to tell Python to skip the rest of the statements in the current loop block and to *continue* to the next iteration of the loop.

Example: `continue.py`

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
        continue
print('Input is of sufficient length')
```

Output:

```
Enter something : a
Too small
```

```
Enter something : 12
Too small
Enter something : abc
Input is of sufficient length
Enter something : quit
```

How it works:
In this program, we accept input from the user, but we process the input string only if it is at least 3 characters long. So, we use the built-in `len` function to get the length and if the length is less than 3, we skip the rest of the statements in the block by using the `continue` statement. Otherwise, the rest of the statements in the loop are executed.

Note that the `continue` statement works with the `for` loop as well.

CHAPTER 2

Functions and Modules

2.1 Functions

Functions are reusable pieces of programs. They allow us to give a name to a block of statements that allows us to run that block using the specified name anywhere in the program and any number of times. This is known as *calling the function*. There are some built-in functions such as `len` and `range`.

Functions are defined using the `def` keyword. After this keyword comes an identifier name for the function, followed by a pair of parentheses which may enclose some names of variables. Finally the line ends by a colon. Next follows the block of statements that are part of this function. Consider the following example.

Example: `function.py`

```
def say_hello():
    # block belonging to the function
    print('hello world')
# End of function

say_hello() # call the function
say_hello() # call the function again
```

Output:

```
hello world
hello world
```

How It works
We define a function called `say_hello` using the syntax as explained above. This function takes no parameters and hence no variables are appearing in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results. Notice that we can call the same function twice which means we do not have to write the same code again.

2.1.1 Function Parameters

A function can take parameters, which are values we supply to the function so that the function does something using these values. These parameters are just like variables except that the values of these variables are defined when we call the function and are already assigned values when the function runs. Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. The names given in the function definition are called *parameters* whereas the values we supply during the calling of the function are called *arguments*.

Example: `function_param.py`

```
def print_max(a, b):
    if a > b:
        print(a, ‘is maximum’)
    elif a == b:
        print(a, ‘is equal to’, b)
    else:
        print(b, ‘is maximum’)
# directly pass literal values
print_max(3, 4)
x = 5
y = 7
# pass variables as arguments
print_max(x, y)
```

Output:

```
4 is maximum
7 is maximum
```

How It works:
Here, we define a function called `print_max` that uses two parameters called `a` and `b`. We find out the greater number using a simple `if..else` statement and then print the bigger number.

The first time we call the function `print_max`, we directly supply the numbers as arguments. In the second case, we call the function with variables as arguments. `print_max(x, y)` causes the value of argument `x` to be assigned to parameter `a` and the value of argument `y` to be assigned to parameter `b`. The `print_max` function works the same way in both the cases.

2.1.2 Local variables

When variables are declared inside a function, they are not related to other variables with the same names outside the function, i.e. the variable names are *local* to the function. This is called the *scope of the variables*. All the variables have the scope of the block they are declared in starting from the point of definition of the name. For example,

Example: `function_local.py`

```
x = 50
def func(x):
    print('x is ', x)
    x = 2
    print('Changed local x to ',x)

func(x)
print('x is still ',x)
```

Output:

```
x is 50
Changed local x to 2
x is still 50
```

How it works:
The first time we print the value of the variable named `x` with the first line in the function's body, Python uses the value of the parameter `x` declared in the main block above the function definition, i.e. `x = 50`.

Next, we assign the value `2` to `x`. The name `x` is local to our function now. So when we change the value of x in the function, the x defined in the main block remains unaffected.

Finally, with the last print statement, we display the value of x in the main block, thereby confirming that it is actually unaffected by the local assignment within the function called above.

2.1.3 The `global` statement

If we want to assign or change a value to a variable name defined at the top level (beginning) of the program, then we have to tell Python that the name is not local, but it is *global*. This can be done using the `global` statement. It is impossible to assign or change a value of a variable defined outside a function without the `global` statement.

Use of the `global` statement indicates that the variable is defined in an outermost block, i.e. the main block.

Example: `function_global.py`

```
x = 50
def func():
    global x

    print('x is', x)
    x = 2
    print('Changed global x to', x)

func()
print('Value of x is', x)
```

Output:

```
x is 50
Changed global x to 2
Value of x is 2
```

How It works:
In the above program, the `global` statement is used to declare that `x` is a global variable. Hence, when we assign a value to `x` inside the function, that change is reflected when we use the value of `x` in the main block. One can specify more than one global variable using the same global statement e.g. `global x, y, z`.

2.1.4 Default Argument Values

For some functions, certain parameters are made optional and default values are used in case the user does not want to provide their values. This is done with the help of default argument values. We can specify the default arguments by appending the assignment operator (`=`) to the parameter name in the function definition followed by the default value. Note that the default argument value should be constant and immutable.

Example: `function_default.py`

```
def say(message,times=1):
    print(message * times)
say('Hello')
say('World',5)
```

Output:

```
Hello
World World World World World
```

How it works:
The function named `say` is used to print the string as many times as specified. If we do not supply the value, then the default value is `1`. We do this by specifying a default argument value `1` to the parameter `times`.

In the first usage of the function `say` in the above program, we supply only the string `‘Hello’` and it prints the string once (by default). During the second call of the function `say`, we supply both the arguments, i.e. `‘World’` to the string and argument `5` to the parameter `times` stating that we want to print the string `5` times.

2.1.5 Keyword Arguments

In a function with many parameters if we want to specify only some of them, then we can give values of such parameters by naming them. This is called *keyword arguments*. We use the name keyword instead of position to specify the arguments to the function. There are two advantages of this.

1. Using the function is easier since we do not need to worry about the order of the arguments.
2. We can give values to only those parameters which we want to, provided that the other parameters have default argument values.

Example: `function_keyword.py`

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

How it works:
A function named `func` has one parameter without default argument value followed by two parameters with default argument values.

In the first usage of the function, `func(3,7)`, the parameter `a` gets the value `3`, parameter `b` gets the value `7`, and parameter `c` gets the default value which is `10`.

During the second usage, i.e. `func(25, c=24)`, the parameter `a` get the value `25` due to the position of the argument, then the parameter `c` get the value `24` due to naming, i.e. keyword arguments. The variable `b` gets the default value of `5`.

In the third usage `func(c=50, a=100)` we use keyword arguments for all specified values. Note that here we are specifying the value for parameter `c` before `a` even though `a` is defined before `c` in the function definition.

2.1.6 VarArgs Parameters

Sometimes we may want to define a function that can take any number of parameter, i.e. variable number of arguments. This can be achieved by using the stars. For example, **Example:** `function_varargs.py`

```
def total(a=5, *numbers, **phonebook):
    print('a', a)

    # iterate through all the items in tuple
    for single_item in numbers:
        print('single_item', single_item)

    # iterate through all the items in dictionary
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))
```

Output:

```
a 10
single_item 1
single_item 2
single_item 3
Inge 1560
John 2231
Jack 1123
None
```

How it works: When we declare a starred parameter such as `*param`, then all the possible arguments from that point till end are collected as a tuple called 'param'.

Similarly, when we declare a double-starred parameter such as `**param`, then all the keyword arguments from that point till the end are collected as a dictionary called ‘param’.

2.1.7 The `return` statement

The `return` statement is used to return from a function, i.e. break out of the function. We can optionally return a value from the function also. For example

Example: `function_return.py`

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return ‘The numbers are equal’
    else:
        return y

print(maximum(2, 3))
```

Output

```
3
```

How it works:
The `maximum` function returns the maximum of the parameters supplied to the function. It uses a simple `if..else` statement to find a greater value and then returns that value.

A `return` statement without a value is equivalent to `return None`, where `None` in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of `None`.

Every function implicitly contains a `return None` statement at the end unless we have specified our own `return` statement. This can be seen by running `print(some_function())`, where the function `some_function` does not use the `return` statement:

```
def some_function():
    pass
```

The `pass` statement is used in Python to indicate an empty block of statements.

2.1.8 DocStrings

Documentation strings are usually referred in short by *docstrings*. DocStrings are an important tool that helps us to document the program better and makes it easier to understand. We can access the docstring from a function when the program is running.

Example (save as function_docstring.py):

```
def print_max(x, y):
    “‘Prints the maximum of two numbers.


    The two values must be integers.”’
    # convert to integers, if possible
    x = int(x)
    y = int(y)

    if x > y:
        print(x, ‘is maximum’)
    else:
        print(y, ‘is maximum’)

print_max(3, 5)
print(print_max.__doc__)
```

Output:

```
5 is maximum
Prints the maximum of two numbers.


    The two values must be integers.
```

How it works:
A string on the first logical line of a function is the *docstring* or *document string* for that function. DocStrings apply to modules and other classes also. The following convention is followed for a docstring, i.e. docstrings are written in the following way:

Docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. Then the second line is blank followed by any detailed explanation starting from the third line.

In the above program the docstring of the `print_max` function can be accessed by using the `__doc__` attribute (note the double underscores) of the function. It is same as accessing `help( )` in Python.

2.2 Modules

Suppose we want to reuse a some functions in other programs that we write. This can be done using modules. There are many ways of writing modules. The easiest way is to create a file with a .py extension that contains functions and variables. Another method is to write the modules in the native language (for example C programming) in which the Python interpreter itself was written.

The functionality of a module can be used by importing it through another program. This is how we can use the Python standard library also. First, we will describe how to use the standard library modules.

Example: module_using_sys.py

```
import sys

print('The command line arguments are:')
for i in sys.argv:
    print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

Output:

```
$ python    module_using_sys.py we are arguments    # each arg is separated by white space
The command line arguments are:
module_using_sys.py
we
are
arguments


The PYTHONPATH is ['/tmp/py',
# many entries here, not shown here
'/Library/Python/2.7/site-packages',
'/usr/local/lib/python2.7/site-packages']
```

How it works:
First, we import the sys module using the import statement which tells Python that we want to use this module. The sys module contains functionality related to the Python interpreter and its environment i.e. the system.

When Python executes the import sys statement, it looks for the sys module. Since it is a built-in module, Python knows where to find it. If its imported is not built-in, then, the Python interpreter will look for it in the directories listed in its sys.path variable. If the module is found, then the statements in the content of that module are run and the module is made available. Note that the initialization is done only the first time when we import any module.

The `argv` variable in the `sys` module is accessed using the dotted notation i.e. `sys.argv`. The `sys.argv` variable is a list of strings. It contains the list of command line arguments.

Here, when we execute `python module_using_sys.py we are arguments`. We run the module `module_using_sys.py` with the python command and the other things that follow are arguments passed to the program. Python stores the command line arguments in the `sys.argv` variable.

Note that the name of the script running is always the first element in the `sys.argv` list. So, in this case we will have `‘module_using_sys.py’` as `sys.argv[0]` , `‘we’` as `sys.argv[1]`, `‘are’` as `sys.argv[2]` and `‘arguments’` as `sys.argv[3]`.

2.2.1 The from..import statement

We can directly import the argv variable into the program, i.e. avoid writing `sys.` everytime by using the `from sys import argv` statement.

Example:

```
from math import sqrt
print("Square root of 16 is", sqrt(16))
```

2.2.2 A module’s `__name__`

Every module has a name. Statements in a module can find out the name of their module. This is useful to determine whether the module is being run standalone or being imported. A module can be used by itself or it can be imported from another module. This can be done using the `__name__` attribute of the module.

Example: `module_using_name.py`

```
if __name__ == ‘__main__’:
    print(‘This program is being run by itself’)
else:
    print(‘I am being imported from another module’)
```

Output:

```
$ python module_using_name.py
This program is being run by itself

$ python
>>> import module_using_name
```

```
I am being imported from another module
>>>
```

How it works:
Every Python module has its `__name__` defined. If this is `'__main__'`, that implies that the module is being run standalone by the user and we can take appropriate actions.

2.2.3 Creating custom modules

Creating our own modules is easy since every python program is a module. So we just have to create a file with `.py` extension. Consider the following example of a sample module.

Example: `mymodule.py`

```
def say_hi():
    print('Hi! This is mymodule.')

__version__ = '0.1'
```

Now we describe how this module can be used in other programs. Note that which we want to import should be in the same directory (folder) as our python program or it should be place in the `sys.path`. Now we write a program to exhibit the use of `mymodule` created above.

Example: `using_mymodule.py`

```
import mymodule

mymodule.say_hi()
print('Version is ',mymodule.__version__)
```

Output:

```
$ python using_mymodule.py
Hi! this is mymodule.
Version is 0.1
```

How it works:
The members of any module can be accessed by the dotted notation. Python. The following is a version of the above program using the `from..import` statement.

Example: `using_mymodule.py`

```
from mymodule import say_hi, __version__

say_hi()
print('Version', __version__)
```

The output of `using_mymodule.py` is same as the output of `using_mymodule.py`.

If we want to import all the components (i.e. members) of the module then we can also use the following method:

```
from mymodule import *
```

This will import all public names such as `say_hi` but would not import `__version__` because it starts with double underscores.

2.2.4 The `dir` function

The `dir()` function is a built-in function which returns list of names defined by an object. If the object is a module, this list includes functions, classes and variables, defined inside that module.

The `dir()` function can accept arguments. If the argument is the name of the module, then it returns the list of names from that specified module. If there is no argument, function returns list of names from the current module.

Example:

```
$ python
>>> import sys

# get names of attributes in sys module
>>> dir(sys)
['__displayhook__', '__doc__',
'argv', 'builtin_module_names',
'version', 'version_info']
# only few entries shown here

# get names of attributes for current module
>>> dir()
['__builtins__', '__doc__',
'__name__', '__package__', 'sys']

# create a new variable 'a'
>>> a = 5
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys', 'a']

# delete/remove a name
>>> del a

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

How it works:
First usage of the `dir` function is on the imported `sys` module. We can see the huge list of attributes contained in `sys` module.

Next, we use the `dir` function without passing parameters to it. By default, it returns the list of attributes for the current module. Notice that the list of imported modules is also part of this list.

Further, we define a new variable `a` and assign it the value `5`. Now we check `dir` and we can observe that there is an additional value in the list of the same name. We remove the variable/attribute of the current module using the `del` statement and the change is reflected again in the output of the `dir` function.

Note that `del` statement is used to delete a variable/name and after the statement has run, the variable/name does not exist at all. In this case we wrote `del a` after which we can see that we can no longer access the variable `a` as if it never existed before at all.