# CHAPTER 1

INTRODUCTION TO PYTHON PROGRMMING

* 1. Features of Python programming

Python's features include-

 Easy-to-learn: Python has few keywords, simple structure, and a clearly defined

syntax. This allows a student to pick up the language quickly.

 Easy-to-read: Python code is more clearly defined and visible to the eyes.

 Easy-to-maintain: Python's source code is fairly easy-to-maintain.

 A broad standard library: Python's bulk of the library is very portable and cross-

platform compatible on UNIX, Windows, and Macintosh.

 Interactive Mode: Python has support for an interactive mode, which allows

interactive testing and debugging of snippets of code.

 Portable: Python can run on a wide variety of hardware platforms and has the

same interface on all platforms.

 Extendable: You can add low-level modules to the Python interpreter. These

modules enable programmers to add to or customize their tools to be more

efficient.

 Databases: Python provides interfaces to all major commercial databases.

 GUI Programming: Python supports GUI applications that can be created and

ported to many system calls, libraries and windows systems, such as Windows MFC,

Macintosh, and the X Window system of Unix.

 Scalable: Python provides a better structure and support for large programs than

shell scripting.

Apart from the above-mentioned features, Python has a big list of good features. A few

are listed below-

 It supports functional and structured programming methods as well as OOP.

 It can be used as a scripting language or can be compiled to byte-code for building

large applications.

 It provides very high-level dynamic data types and supports dynamic type

checking.

 It supports automatic garbage collection.

 It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

* 1. **Applications of Python programming**
* Game Development.
* Scientific and Numeric Applications.
* Artificial Intelligence and Machine Learning.
* Software Development.
* Enterprise-level/Business Applications.
* Education programs and training courses.
* Language Development.
  1. **Identifiers and Reserved words**

A Python identifier is a name used to identify a variable, function, class, module or other

object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by

zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers.

Python is a case sensitive programming language. Thus, Manpower and manpower are

two different identifiers in Python.

Here are naming conventions for Python identifiers-

 Class names start with an uppercase letter. All other identifiers start with a

lowercase letter.

 Starting an identifier with a single leading underscore indicates that the identifier

is private.

 Starting an identifier with two leading underscores indicates a strong private

identifier.

 If the identifier also ends with two trailing underscores, the identifier is a language-

defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot

use them as constants or variables or any other identifier names. All the Python keywords

contain lowercase letters only.

and

exec

Not

as

finally

or

assert

for

pass

break

from

print

class

global

raise

continue

if

return

def

import

try

del

in

while

elif

is

with

else

lambda

yield

except

* 1. **Operators**

Operators are the constructs, which can manipulate the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called the operator.

## TypesofOperator

Python language supports the following types of operators-

1. ArithmeticOperators
2. Comparison (Relational)Operators
3. AssignmentOperators
4. LogicalOperators
5. BitwiseOperators
6. MembershipOperators
7. IdentityOperators

Let us have a look at all the operators one by one.

## Python Arithmetic Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then-

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + Addition | Adds values on either side of the operator. | a + b = 31 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -11 |
| \* Multiplication | Multiplies values on either side of the operator | a \* b = 210 |
| / Division | Divides left hand operand by right hand operand | b / a = 2.1 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 1 |
| \*\* Exponent | Performs exponential (power) calculation on operators | a\*\*b =10 tothe power 20 |

|  |  |  |
| --- | --- | --- |
| // | FloorDivision-Thedivisionofoperandswhere the result is the quotient in which the digits after the decimal point areremoved. | 9//2 = 4 and 9.0//2.0 = 4.0 |

##### Example

Assume variable a holds 10 and variable b holds 20, then-

#!/usr/bin/python3 a =21

b =10

c = 0

c = a + b

print("Line1-Valueofcis",c)

c = a - b

print("Line2-Valueofcis",c)

c = a \* b

print("Line3-Valueofcis",c)

c = a / b

print("Line4-Valueofcis",c)

c = a % b

print("Line5-Valueofcis",c)

a =2

b =3

c = a\*\*b

print("Line6-Valueofcis",c)

a = 10

b = 5

c=a//b

print("Line7-Valueofcis",c)

When you execute the above program, it produces the following result-

Line1-Valueofcis 31Line2-Valueofcis 11

Line3-Valueofcis 210Line4-Valueofcis 2.1Line5-Valueofcis 1 Line6-Valueofcis 8 Line7-Valueofcis 2

## Python ComparisonOperators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds the value 10 and variable b holds the value 20, then-

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | If the values of two operands are equal, then the condition becomes true. | (a == b) is not true. |
| != | If values of two operands are not equal, then condition becomes true. | (a!= b) is true. |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. | (a > b) is not true. |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. | (a < b) is true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | (a >= b) is not true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

##### Example

Assume variable a holds 10 and variable b holds 20, then-

#!/usr/bin/python3 a =21

b =10

if ( a == b ):

print("Line1-aisequaltob") else:

print("Line1-aisnotequaltob")

if(a!=b):

print("Line2-aisnotequaltob") else:

print("Line2-aisequaltob")

if ( a < b ):

print("Line3-aislessthanb") else:

print("Line3-aisnotlessthanb")

if ( a > b ):

print("Line4-aisgreaterthanb") else:

print("Line4-aisnotgreaterthanb")

a,b=b,a#valuesofaandbswapped.abecomes10,bbecomes21 if ( a <= b ):

print("Line5-aiseitherlessthanorequaltob")

else:

print("Line5-aisneitherlessthannorequaltob")

if ( b >= a ):

print("Line6-biseithergreaterthanorequaltob") else:

print("Line6-bisneithergreaterthannorequaltob")

When you execute the above program, it produces the following result-

Line1-aisnotequaltob Line2-aisnotequaltob Line3-aisnotlessthanb Line4-aisgreaterthanb

Line5-aiseitherlessthanorequalto b Line6-biseithergreaterthan orequaltob

## PythonAssignmentOperators

Assume variable a holds 10 and variable b holds 20, then-

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assignsvaluesfromrightsideoperandsto left sideoperand | c = a + b assigns value of a + b into c |
| += Add AND | It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |

|  |  |  |
| --- | --- | --- |
| -= Subtract AND | It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*= Multiply AND | It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /= Divide AND | It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / ac /= a is equivalent to c = c/ a |
| %= Modulus AND | It takes modulus using two operands and assign the result to left operand | c %= a is equivalent to c = c % a |
| \*\*= Exponent AND | Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a is equivalent to c =c  \*\* a |
| //= Floor Division | Itperformsfloordivisiononoperatorsand assign value to the leftoperand | c //= a is equivalent to c = c // a |

##### Example

Assume variable a holds 10 and variable b holds 20, then-

#!/usr/bin/python3

a =21

b =10

c = 0

c = a + b

print("Line1-Valueofcis",c)

c += a

print("Line2-Valueofcis",c)

c \*= a

print("Line3-Valueofcis",c)

c/=a

print("Line4-Valueofcis",c)

c = 2 c %=a

print("Line5-Valueofcis",c)

c \*\*= a

print("Line6-Valueofcis",c)

c//=a

print("Line7-Valueofcis",c)

When you execute the above program, it produces the following result-

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Line | 1 | -Value | of | c | is | 31 |
| Line | 2 | -Value | of | c | is | 52 |
| Line | 3 | -Value | of | c | is | 1092 |
| Line | 4 | -Value | of | c | is | 52.0 |
| Line | 5 | -Value | of | c | is | 2 |
| Line | 6 | -Value | of | c | is | 2097152 |
| Line | 7 | -Value | of | c | is | 99864 |

## Python BitwiseOperators

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows-

a = 00111100

b = 00001101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Pyhton's built-in function bin() can be used to obtain binary representation of an integer number.

The following Bitwise operators are supported by Python language-

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Operator** | | **Description** | | | | | | | | **Example** |
| & Binary AND | | Operator copies a bit to the result, if it exists in both operands | | | | | | | | (a & b) (means 0000 1100) |
| | Binary OR | | It copies operand. | a | bit, | if | it | exists | in | either | (a | b) = 61 (means 0011 1101) |
| ^ Binary XOR | | Itcopiesthebit,ifitissetinoneoperand but notboth. | | | | | | | | (a ^ b) = 49 (means 0011 0001) |
| ~ Binary Complement | Ones | It is unary and has the effect of 'flipping' bits. | | | | | | | | (~a ) = -61 (means1100 0011 in 2's  complement form due to a signed binarynumber. |
| << Binary Left Shift | | The left operand’s value is moved leftby the number of bits specified by the right operand. | | | | | | | | a << = 240 (means 1111 0000) |
| >> Binary RightShift | | The left operand’s value is moved right by the number of bits specified by the right operand. | | | | | | | | a >> = 15 (means 0000 1111) |

##### Example

#!/usr/bin/python3

a =60

b =13

# 60 = 00111100

# 13 = 00001101

print('a=',a,':',bin(a),'b=',b,':',bin(b))

c = 0

c = a &b;

# 12 = 0000 1100

print("resultofANDis",c,':',bin(c))

c = a | b;

# 61 = 0011 1101

print("resultofORis",c,':',bin(c))

c = a ^ b;

# 49 = 0011 0001

print("resultofEXORis",c,':',bin(c))

c = ~a;

# -61 = 1100 0011

print("resultofCOMPLEMENTis",c,':',bin(c))

c = a << 2;

# 240 = 1111 0000

print("resultofLEFTSHIFTis",c,':',bin(c))

c = a >> 2;

# 15 = 0000 1111

print("resultofRIGHTSHIFTis",c,':',bin(c))

When you execute the above program, it produces the following result-

a= 60 : 0b111100 b= 13 : 0b1101

resultofANDis 12:0b1100 resultofORis 61:0b111101 resultofEXORis 49:0b110001

resultofCOMPLEMENTis -61:-0b111101 resultofLEFTSHIFTis 240:0b11110000resultofRIGHTSHIFTis 15:0b111

## Python LogicalOperators

ThefollowinglogicaloperatorsaresupportedbyPythonlanguage.Assumevariableaholds True and variable b holds Falsethen-

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and Logical AND | If both the operands are true then condition becomes true. | (a and b) is False. |
| or Logical OR | If any of the two operands are non-zero then condition becomes true. | (a or b) is True. |
| not Logical NOT | Used to reverse the logical state of its operand. | Not(a and b) is True. |

## Python MembershipOperators

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below-

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Evaluates to true, if it finds a variable in the specified sequence and false otherwise. | x in y, here in results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true, if it does not find a variableinthespecifiedsequenceand falseotherwise. | x not in y, here not in results in a 1 if x is not a member of sequence y. |

##### Example

#!/usr/bin/python3

a =10

b =20

list=[1,2,3,4,5]

if(ainlist):

print("Line1-aisavailableinthegivenlist") else:

print("Line1-aisnotavailableinthegivenlist")

if(bnotinlist):

print("Line2-bisnotavailableinthegivenlist") else:

print("Line2-bisavailableinthegivenlist")

c=b/a

if(cinlist):

print("Line3-aisavailableinthegivenlist") else:

print("Line3-aisnotavailableinthegivenlist")

When you execute the above program, it produces the following result-

Line1-aisnotavailableinthegivenlist Line2-bisnotavailableinthegivenlist Line3-aisavailableinthegivenlist

## PythonIdentityOperators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | x is y, here is results in 1 if id(x) equals id(y). |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | x is not y, here is not results in 1 if id(x) is not equal toid(y). |

##### Example

#!/usr/bin/python3

a =20

b =20

print('Line1','a=',a,':',id(a),'b=',b,':',id(b))

if(aisb):

print("Line2-aandbhavesameidentity") else:

print("Line2-aandbdonothavesameidentity")

if(id(a)==id(b)):

print("Line3-aandbhavesameidentity") else:

print("Line3-aandbdonothavesameidentity")

b = 30

print('Line4','a=',a,':',id(a),'b=',b,':',id(b))

if(aisnotb):

print("Line5-aandbdonothavesameidentity") else:

print("Line5-aandbhavesameidentity")

When you execute the above program, it produces the following result-

Line1a=20:1594701888b=20:1594701888

Line2-aandbhavesameidentity Line3-aandbhavesameidentity

Line4a=20:1594701888b=30:1594702048

Line5-aandbdonothavesameidentity

## Python Operators Precedence

The following table lists all the operators from highest precedence to the lowest.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Ccomplement, unary plus and minus (method names for the last two are +@ and -@) |
| \* / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >><< | Right and left bitwise shift |
| & | Bitwise 'AND' |
| ^ | | Bitwise exclusive `OR' and regular `OR' |
| <= <>>= | Comparison operators |
| <> == != | Equality operators |

|  |  |
| --- | --- |
| = %= /= //= -= += \*=  \*\*= | Assignment operators |
| isis not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

Operator precedence affects the evaluation of an an expression.

For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because the operator \* has higher precedence than +, so it first multiplies 3\*2 and then is added to 7.

Here,theoperatorswiththehighestprecedenceappearatthetopofthetable,thosewith the lowest appear at thebottom.

##### Example

#!/usr/bin/python3

a =20

b =10

c =15

d = 5

print("a:%db:%dc:%dd:%d"%(a,b,c,d))

e = (a + b) \* c /d #( 30 \* 15 ) / 5 print("Valueof(a+b)\*c/dis",e)

e = ((a + b) \* c) / d

# (30 \* 15 ) / 5

print("Valueof((a+b)\*c)/dis",e)

e = (a + b) \* (c /d)

#(30)\*(15/5)

print("Valueof(a+b)\*(c/d)is",e)

e=a+(b\*c)/d # 20+(150/5)

print("Valueofa+(b\*c)/dis",e)

When you execute the above program, it produces the following result-

a:20b:10c:15d:5

Valueof(a+b)\*c/dis90.0

Valueof((a+b)\*c)/dis90.0 Valueof(a+b)\*(c/d)is90.0 Valueofa+(b\*c)/dis50.0

**1.5 Expressions**

In Python, operators are special symbols that designate that some sort of computation should be performed. The values that an operator acts on are called operands.

Here is an example:

>>>

>>> a = 10

>>> b = 20

>>> a + b

30

In this case, the + operator adds the operands a and b together. An operand can be either a literal value or a variable that references an object:

>>>

>>> a = 10

>>> b = 20

>>> a + b - 5

25

A sequence of operands and operators, like a + b - 5, is called an expression. Python supports many operators for combining data objects into expressions. These are explored below.

**1.6 Input and output statements**

## Reading Input From the Keyboard

Programs often need to obtain data from the user, usually by way of input from the keyboard. The simplest way to accomplish this in Python is with input().

input([<prompt>])

Reads a line of input from the keyboard.

input() pauses program execution to allow the user to type in a line of input from the keyboard. Once the user presses the Enter key, all characters typed are read and returned as a string:

>>>

>>>s=input()

foo bar baz

>>>s

'foo bar baz'

Note that the newline generated when the user presses the Enter key isn’t included as part of the return string.

If you include the optional <prompt> argument, input() displays it as a prompt to the user before pausing to read input:

>>>

>>>name=input('What is your name? ')

What is your name? Winston Smith

>>>name

'Winston Smith'

input() always returns a string. If you want a numeric type, then you need to [convert the string](https://realpython.com/courses/convert-python-string-int/) to the appropriate type with the int(), float(), or complex() built-in functions:

>>>

1>>>n=input('Enter a number: ')

2Enter a number: 50

3>>>print(n+100)

4Traceback (most recent call last):

5 File"<stdin>", line 1, in <module>

6TypeError: must be str, not int

7

8>>>n=int(input('Enter a number: '))

9Enter a number: 50

10>>>print(n+100)

11150

In the example above, the expression n + 100 on line 3 is invalid because n is a string and 100 is an integer. Line 8 converts n to an integer so the print() statement on line 10 succeeds.

Python Version Note: Should you find yourself working with Python 2.x code, you might bump into a slight difference in the input functions between Python versions 2 and 3.

raw\_input() in Python 2 reads input from the keyboard and returns it. raw\_input() in Python 2 behaves just like input() in Python 3, as described above.

But Python 2 also has a function called input(). In Python 2, input() reads input from the keyboard, parses and evaluates it as a Python expression, and then returns the resulting value.

Python 3 does not provide a single function that does exactly what Python 2’s input() does. The effect can be mimicked in Python 3 with the expression eval(input()). However, this is considered a security risk because it allows a user to run arbitrary, potentially malicious code.

For more information on eval() and its potential security risks, check out [Python eval(): Evaluate Expressions Dynamically](https://realpython.com/python-eval-function/).

## Writing Output to the Console

In addition to obtaining data from the user, a program will also usually need to present data back to the user. You can display program data to the console in Python with print().

### Unformatted Console Output

To display objects to the console, pass them as a comma-separated list of argument to print().

print(<obj>, ..., <obj>)

Displays a string representation of each <obj> to the console.

By default, print() separates each object by a single space and appends a newline to the end of the output:

>>>

>>>fname='Winston'

>>>lname='Smith'

>>>print('Name:',fname,lname)

Name: Winston Smith

Any type of object can be specified as an argument to print(). If an object isn’t a string, then print() converts it to an appropriate string representation displaying it:

>>>

>>>a=[1,2,3]

>>>type(a)

<class 'list'>

>>>b=-12

>>>type(b)

<class 'int'>

>>>d={'foo':1,'bar':2}

>>>type(d)

<class 'dict'>

>>>type(len)

<class 'builtin\_function\_or\_method'>

>>>print(a,b,d,len)

[1, 2, 3] -12 {'foo': 1, 'bar': 2} <built-in function len>

As you can see, even complex types like lists, dictionaries, and functions can be displayed to the console with print().

[Remove ads](https://realpython.com/account/join/)

### Keyword Arguments to print()

print() takes a few additional arguments that provide modest control over the format of the output. Each of these is a special type of argument called a keyword argument. This introductory series of tutorials will include a tutorial on functions and parameter passing so you can learn more about keyword arguments.

For now, here’s what you need to know:

* Keyword arguments have the form <keyword>=<value>.
* Any keyword arguments passed to print() must come at the end, after the list of objects to display.

In the following sections, you’ll see how these keyword arguments affect console output produced by print().

#### The sep= Keyword Argument

Adding the keyword argument sep=<str> causes objects to be separated by the string <str> instead of the default single space:

>>>

>>>print('foo',42,'bar')

foo 42 bar

>>>print('foo',42,'bar',sep='/')

foo/42/bar

>>>print('foo',42,'bar',sep='...')

foo...42...bar

>>>d={'foo':1,'bar':2,'baz':3}

>>>fork,vind.items():

... print(k,v,sep=' -> ')

...

foo -> 1

bar -> 2

baz -> 3

To squish objects together without any space between them, specify sep='':

>>>

>>>print('foo',42,'bar',sep='')

foo42bar

You can specify any arbitrary string as the separator with the sep= keyword.

#### The end= Keyword Argument

The keyword argument end=<str> causes output to be terminated by <str> instead of the default newline:

>>>

>>>ifTrue:

... print('foo',end='/')

... print(42,end='/')

... print('bar')

...

foo/42/bar

For example, if you are displaying values in a loop, you might use end= to cause the values to be displayed on one line, rather than on individual lines:

>>>

>>>forninrange(10):

... print(n)

...

0

1

2

3

4

5

6

7

8

9

>>>forninrange(10):

... print(n,end=(' 'ifn<9else'\n'))

...

0 1 2 3 4 5 6 7 8 9

Any string may be specified as the output terminator with the end= keyword.

#### Output Stream Keyword Arguments

print() accepts two additional keyword arguments, both of which affect handling of the output stream:

1. file=<stream>: By default, print() sends its output to a default stream called sys.stdout, which is usually equivalent to the console. The file=<stream> argument causes output to be sent to an alternate stream designated by <stream> instead.
2. flush=True: Ordinarily, print() buffers its output and only writes to the output stream intermittently. flush=True specifies that the output stream is forcibly flushed with each print().

These two keyword arguments are presented here for the sake of completeness. You probably don’t need to be too concerned about output streams at this point. They are discussed later in this series in the tutorial on File I/O.

**1.7 Flow control**

A program’s control flow is the order in which the program’s code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

## 1.7.1 Conditional Statments

## The if Statement

Often, you need to execute some statements only if some condition holds, or choose statements to execute depending on several mutually exclusive conditions. The Python compound statement if, which uses if, elif, and else clauses, lets you conditionally execute blocks of statements. Here’s the syntax for the if statement:

if expression:

statement(s)

elifexpression:

statement(s)

elifexpression:

statement(s)

...

else:

statement(s)

The elif and else clauses are optional. Note that unlike some languages, Python does not have a switch statement, so you must use if, elif, and else for all conditional processing.

Here’s a typical if statement:

if x < 0: print "x is negative"

elif x % 2: print "x is positive and odd"

else: print "x is even and non-negative"

When there are multiple statements in a clause (i.e., the clause controls a block of statements), the statements are placed on separate logical lines after the line containing the clause’s keyword (known as the header line of the clause) and indented rightward from the header line. The block terminates when the indentation returns to that of the clause header (or further left from there). When there is just a single simple statement, as here, it can follow the : on the same logical line as the header, but it can also be placed on a separate logical line, immediately after the header line and indented rightward from it. Many Python practitioners consider the separate-line style more readable:

if x < 0:

print "x is negative"

elif x % 2:

print "x is positive and odd"

else:

print "x is even and non-negative"

You can use any Python expression as the condition in an if or elif clause. When you use an expression this way, you are using it in a Boolean context. In a Boolean context, any value is taken as either true or false. As we discussed earlier, any non-zero number or non-empty string, tuple, list, or dictionary evaluates as true. Zero (of any numeric type), None, and empty strings, tuples, lists, and dictionaries evaluate as false. When you want to test a value x in a Boolean context, use the following coding style:

if x:

This is the clearest and most Pythonic form. Don’t use:

if x is True:

if x= = True:

if bool(x):

There is a crucial difference between saying that an expression “returns True" (meaning the expression returns the value 1 intended as a Boolean result) and saying that an expression “evaluates as true” (meaning the expression returns any result that is true in a Boolean context). When testing an expression, you care about the latter condition, not the former.

If the expression for the if clause evaluates as true, the statements following the if clause execute, and the entire if statement ends. Otherwise, the expressions for any elif clauses are evaluated in order. The statements following the first elif clause whose condition is true, if any, are executed, and the entire if statement ends. Otherwise, if an else clause exists, the statements following it are executed.

## The while Statement

The while statement in Python supports repeated execution of a statement or block of statements that is controlled by a conditional expression. Here’s the syntax for the while statement:

while expression:

statement(s)

A while statement can also include an else clause and break and continue statements, as we’ll discuss shortly.

Here’s a typical while statement:

count = 0

while x > 0:

x = x // 2 # truncating division

count += 1

print "The approximate log2 is", count

First, expression, which is known as the loop condition, is evaluated. If the condition is false, the while statement ends. If the loop condition is satisfied, the statement or statements that comprise the loop body are executed. When the loop body finishes executing, the loop condition is evaluated again, to see if another iteration should be performed. This process continues until the loop condition is false, at which point the while statement ends.

The loop body should contain code that eventually makes the loop condition false, or the loop will never end unless an exception is raised or the loop body executes a break statement. A loop that is in a function’s body also ends if a return statement executes in the loop body, as the whole function ends in this case.

**1.7.2 Transfer Statements**

## The for Statement

The for statement in Python supports repeated execution of a statement or block of statements that is controlled by an iterable expression. Here’s the syntax for the for statement:

for target in iterable:

statement(s)

Note that the in keyword is part of the syntax of the for statement and is functionally unrelated to the in operator used for membership testing. A for statement can also include an else clause and break and continue statements, as we’ll discuss shortly.

Here’s a typical for statement:

for letter in "ciao":

print "give me a", letter, "..."

iterable may be any Python expression suitable as an argument to built-in function iter, which returns an iterator object (explained in detail in the next section). target is normally an identifier that names the control variable of the loop; the for statement successively rebinds this variable to each item of the iterator, in order. The statement or statements that comprise the loop body execute once for each item in iterable (unless the loop ends because an exception is raised or a break or return statement is executed).

A target with multiple identifiers is also allowed, as with an unpacking assignment. In this case, the iterator’s items must then be sequences, each with the same length, equal to the number of identifiers in the target. For example, when d is a dictionary, this is a typical way to loop on the items in d:

for key, value in d.items( ):

if not key or not value: del d[key] # keep only true keys and values

The items method returns a list of key/value pairs, so we can use a for loop with two identifiers in the target to unpack each item into key and value.

If the iterator has a mutable underlying object, that object must not be altered while a for loop is in progress on it. For example, the previous example cannot use iteritems instead of items. iteritems returns an iterator whose underlying object is d, so therefore the loop body cannot mutate d (by del d[key]). items returns a list, though, so d is not the underlying object of the iterator and the loop body can mutate d.

The control variable may be rebound in the loop body, but is rebound again to the next item in the iterator at the next iteration of the loop. The loop body does not execute at all if the iterator yields no items. In this case, the control variable is not bound or rebound in any way by the for statement. If the iterator yields at least one item, however, when the loop statement terminates, the control variable remains bound to the last value to which the loop statement has bound it. The following code is thus correct, as long as someseq is not empty:

for x in someseq:

process(x)

print "Last item processed was", x

### **1.7.3 Iterative statements**

### Iterators

An iterator is any object i such that you can call i .next( ) without any arguments. i .next( ) returns the next item of iterator i, or, when iterator i has no more items, raises a StopIteration exception. When you write a class (see [Chapter 5](https://www.oreilly.com/library/view/python-in-a/0596001886/ch05.html)), you can allow instances of the class to be iterators by defining such a method next. Most iterators are built by implicit or explicit calls to built-in function iter, covered in [Chapter 8](https://www.oreilly.com/library/view/python-in-a/0596001886/ch08.html). Calling a generator also returns an iterator, as we’ll discuss later in this chapter.

The for statement implicitly calls iter to get an iterator. The following statement:

for x in c:

statement(s)

is equivalent to:

\_temporary\_iterator = iter(c)

while True:

try: x = \_temporary\_iterator.next( )

except StopIteration: break

statement(s)

Thus, if iter( c ) returns an iterator i such that i .next( ) never raises StopIteration (an infinite iterator), the loop for x in c: never terminates (unless the statements in the loop body contain suitable break or return statements or propagate exceptions). iter( c ), in turn, calls special method c .\_\_iter\_\_( ) to obtain and return an iterator on c. We’ll talk more about the special method \_\_iter\_\_ in [Chapter 5](https://www.oreilly.com/library/view/python-in-a/0596001886/ch05.html).

Iterators were first introduced in Python 2.2. In earlier versions, for x in S: required S to be a sequence that was indexable with progressively larger indices 0, 1, ..., and raised an IndexError when indexed with a too-large index. Thanks to iterators, the for statement can now be used on a container that is not a sequence, such as a dictionary, as long as the container is iterable (i.e., it defines an \_\_iter\_\_ special method so that function iter can accept the container as the argument and return an iterator on the container). Built-in functions that used to require a sequence argument now also accept any iterable.

### range and xrange

Looping over a sequence of integers is a common task, so Python provides built-in functions range and xrange to generate and return integer sequences. The simplest, most idiomatic way to loop n times in Python is:

for i in xrange(n):

statement(s)

range( x ) returns a list whose items are consecutive integers from 0 (included) up to x (excluded). range( x,y ) returns a list whose items are consecutive integers from x (included) up to y (excluded). The result is the empty list if x is greater than or equal to y. range( x,y,step ) returns a list of integers from x (included) up to y (excluded), such that the difference between each two adjacent items in the list is step. If step is less than 0, range counts down from x to y. range returns the empty list when x is greater than or equal to y and step is greater than 0, or when x is less than or equal to y and step is less than 0. If step equals 0, range raises an exception.

While range returns a normal list object, usable for all purposes, xrange returns a special-purpose object, specifically intended to be used in iterations like the for statement shown previously. xrange consumes less memory than range for this specific use. Leaving aside memory consumption, you can use range wherever you could use xrange.

### List comprehensions

A common use of a for loop is to inspect each item in a sequence and build a new list by appending the results of an expression computed on some or all of the items inspected. The expression form, called a list comprehension, lets you code this common idiom concisely and directly. Since a list comprehension is an expression (rather than a block of statements), you can use it directly wherever you need an expression (e.g., as an actual argument in a function call, in a return statement, or as a subexpression for some other expression).

A list comprehension has the following syntax:

[ expression for target in iterable

lc-clauses ]

target and iterable are the same as in a regular for statement. You must enclose the expression in parentheses if it indicates a tuple.

lc-clauses is a series of zero or more clauses, each with one of the following forms:

for target in iterable

if expression

target and iterable in each for clause of a list comprehension have the same syntax as those in a regular for statement, and the expression in each if clause of a list comprehension has the same syntax as the expression in a regular if statement.

A list comprehension is equivalent to a for loop that builds the same list by repeated calls to the resulting list’s append method. For example (assigning the list comprehension result to a variable for clarity):

result1 = [x+1 for x in some\_sequence]

is the same as the for loop:

result2 = [ ]

for x in some\_sequence:

result2.append(x+1)

Here’s a list comprehension that uses an if clause:

result3 = [x+1 for x in some\_sequence if x>23]

which is the same as a for loop that contains an if statement:

result4 = [ ]

for x in some\_sequence:

if x>23:

result4.append(x+1)

And here’s a list comprehension that uses a for clause:

result5 = [x+y for x in alist for y in another]

which is the same as a for loop with another for loop nested inside:

result6 = [ ]

for x in alist:

for y in another:

result6.append(x+y)

As these examples show, the order of for and if in a list comprehension is the same as in the equivalent loop, but in the list comprehension the nesting stays implicit.

## The break Statement

The break statement is allowed only inside a loop body. When break executes, the loop terminates. If a loop is nested inside other loops, break terminates only the innermost nested loop. In practical use, a break statement is usually inside some clause of an if statement in the loop body so that it executes conditionally.

One common use of break is in the implementation of a loop that decides if it should keep looping only in the middle of each loop iteration:

while True: # this loop can never terminate naturally

x = get\_next( )

y = preprocess(x)

if not keep\_looping(x, y): break

process(x, y)

**1.8 Functions**

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

## Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* 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. You can also define parameters inside 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.
* 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.

## Syntax

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

## Example

The following function takes a string as input parameter and prints it on standard screen.

defprintme( str):

"This prints a passed string into this function"

print str

return

## 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 −

#!/usr/bin/python

# Function definition is here

defprintme( str):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

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

I'm first call to user defined function!

Again second call to the same function

## Pass by reference vs 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 −

#!/usr/bin/python

# Function definition is here

defchangeme(mylist):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print"Values inside the function: ",mylist

return

# Now you can call changeme function

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.

#!/usr/bin/python

# Function definition is here

defchangeme(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

# Now you can call changeme function

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]

## Function Arguments

You can call a function by using the following types of formal 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. 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 −

#!/usr/bin/python

# Function definition is here

defprintme( 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)

## 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 −

#!/usr/bin/python

# Function definition is here

defprintme( 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.

#!/usr/bin/python

# Function definition is here

defprintinfo( 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

## 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 −

#!/usr/bin/python

# Function definition is here

defprintinfo( 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

## 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 −

#!/usr/bin/python

# Function definition is here

defprintinfo( arg1,\*vartuple):

"This prints a variable passed arguments"

print"Output is: "

print arg1

forvarinvartuple:

printvar

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

## The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You 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.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

## Syntax

The syntax of lambda functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how lambda form of function works −

#!/usr/bin/python

# Function definition is here

sum =lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

print"Value of total : ", sum(10,20)

print"Value of total : ", sum(20,20)

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

Value of total : 30

Value of total : 40

## 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.

All the above examples are not returning any value. You can return a value from a function as follows −

#!/usr/bin/python

# Function definition is here

defsum( 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

## Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

* Global variables
* Local variables

## Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

#!/usr/bin/python

total =0;# This is global variable.

# Function definition is here

defsum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2;# Here total is local variable.

print"Inside the function local total : ", total

return total;

# Now you can call sum function

sum(10,20);

print"Outside the function global total : ", total

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

Inside the function local total : 30

Outside the function global total : 0

**1.9 Modules**

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

# CHAPTER 2

# SYSTEM DESIGN AND REQUIREMENTS

# System Design

# “conflict and combat” is a mini game project where we derived the basic concept of two character fighting against each other from the very popular and simple Rock-paper-scissors game that is played by two people.

# Here, the players count to three in unison and simultaneously “throw” one of three hand signals that correspond to rock, paper or scissors. The winner is determined by the rules:

# • Rock smashes scissors

# • Scissors cuts paper

# • Paper covers rock

# The fact that a tie happens around 1/3 of the time, several variants of Rock-Paper-Scissors exist that include more choices to make ties more unlikely. Rock-paper-scissors-lizard-Spock (RPSLS) is a variant of Rock-paper-scissors that allows five choices. Each choice wins against two other choices, loses against two other choices and ties against itself.

# This possibility of extending the choices and developing a huge web pushed us towards developing this mini game by creating characters and providing their relationship.

# In our first mini-project, we will build a Python function that simulates playing a round of game aka “combat” by generating its own random choice from these alternatives character warriors we provided and then determining the winner using a simple rule that we will next describe

# While a simple Rock-paper-scissor has a set of three rules that logically determine who wins a round of coding up these rules would require of if/ elif/ else. We opted for simpler method for determining the winner is to assign each of the respective choices.

# For example: choice number -7

# number:

# 0 — character 1

# 1 — character 2

# 2 — character 3

# 3 — character 4

# 4 — character 5

# 5— character 6

# 6 — character 7

# In this expanded list, each choice wins against the preceding three choices and loses against the following three choices and there by developing a web of relationship of warriors/characters.

# 44d7d51b-571d-4850-b0ae-036df1d441d9.jpg

# We also developed a matrix based system where we can define each cases possible depending upon the number of the choices user wants to make. For instance:

# Number the choices of character -3

# .Now ,We will make a table of which moves beat which, with +1 in the (*i*, *k*) position if move *i* beats move *j* and -1 if *j* beats *i*. There will be zeros on the diagonal since it’s a tie if both players make the same move.(draw)

# In order for this game to not have a winning strategy, the table must be filled in as below, with the only option being to set *a* = 1 or *a* = -1.

|---+----+----+----|

| | 1 | 2 | 3 |

|---+----+----+----|

| 1 | 0 | a | -a |

|---+----+----+----|

| 2 | -a | 0 | a |

|---+----+----+----|

| 3 | a | -a | 0 |

|---+----+----+----|

# If 1, 2, and 3 correspond to Rock, Paper, and Scissors, then *a* = 1 according to the usual rules.

# Next we fill in the rest of the moves. The table must be skew-symmetric, i.e. the (*i*, *j*) element must be the negative of the (*j*, *i*) element, because if (*i*, *j*) is a winning move then (*j*, *i*) is a losing move and vice versa. Also, the rows and columns must sum to zero. Together these requirements greatly reduce the number of possibilities.

# System Requirements

## 2.2.1Software Requirements

* Operating System: Windows 7 above
* Python 3.9.0
* Text Editor: VS CODE, Anaconda

## 2.2.2Hardware Requirements

* + - * Processor: Intel core i3 8100 3.60Ghz
      * RAM: 8 GB
      * HDD: 500 GB
      * Keyboard
      * Mouse
      * Monitor

**CHAPTER 3**

**SYSTEM IMPLEMENTATION**

**3.1 CODE SNIPPETS**

# import system from os

from os import system

# import colorama for text colour

import colorama

# import sleep to show output for some time period

from time import sleep

# define our clear function

def clear():

\_ = system('cls')

#funtion for our introduction

name=''

def intro():

clear()

global name

print("\n\n\n\n\n")

print("\t\t\t \*------------------\*")

print("\t\t\t =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=")

print("\t\t\t\t\t = WELCOME =")

print("\t\t\t\t\t = TO =")

print("\t\t\t\t\t = COMBACT AND CONFLICT =")

print("\t\t\t .......Developed by....... ")

print("\t\t\t .......SidharthaParasramka.......")

print("\t\t\t ..........Soumya Agrawal..........")

print("\t\t\t ..........Swetalina Nayak.........")

print("\t\t\t =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=")

print("\t\t\t \*------------------\*")

print("\n\n\n\t\t\t Game Will Start In few seconds")

print("\n\n\n\t\t\tMeanwhile You can Enter Your Name:-",end='')

name=input()

sleep(2)

def heading():

with open('heading.txt',encoding="utf8") as f:

print(f.read())

#funtion to print the rules from the text file

def txt(a):

heading()

f=open(a,'r')

data=f.read()

print(data)

f.close()

intro()

clear()

txt('introduction.txt')

print("\n\n\n\tPRESS ENTER KEY TO CONTINUE:--",end='')

input()

clear()

heading()

class WrongSelectionException(Exception):

def \_init\_(self):

arg='Invalid Selection of Number Of warriors'

numberElements=[3,5,7,9]

print("\n\t\tYou can have either 3,5,7 or 9 Warriors in the pool")

Element=int(input("\n\t\tCaution!!!- Wrong input will result in Exception\n\t\tEnter The Number Of Warriors In The Game "))

# If using Windows, init() will cause anything sent to stdout or stderr

# will have ANSI color codes converted to the Windows versions. Hooray!

# If you are already using an ANSI compliant shell, it won't do anything

colorama.init()

# Now regular ANSI codes should work, even in Windows

RED = '\033[31m' # mode 31 = red forground

RESET = '\033[0m' # mode 0 = reset

print("\t\tselected input: "+RED + str(Element) + RESET)

if(Element not in numberElements):

raise WrongSelectionException

print("\n\t\t Game will Procede in 2 seconds")

sleep(2)

clear()

txt('gamedata.txt')

print("\n\tPRESS ANY KEY TO START PLAYING THE GAME:--",end='')

input()

clear()

Pwarriors=[]

Twarriors=("KORG The Fighter","Jack The Ripper","Goat Skinned"," Gula monster","BountyHunter","UndeadGhoul","Zeus'sSon","NightCrawler","Militaristic Fighting Machine")

print("\n",'\*'\*50)

print("\tEnter 1 for KORG The Fighter")

print("\tEnter 2 for Jack The Ripper")

print("\tEnter 3 for Goat Skinned")

print("\tEnter 4 for Gula monster")

print("\tEnter 5 for Bounty Hunter")

print("\tEnter 6 for Undead Ghoul")

print("\tEnter 7 for Zeus's Son")

print("\tEnter 8 for Night Crawler")

print("\tEnter 9 for Militaristic Fighting Machine")

print("\n",'\*'\*50)

for i in range(Element):

print("\t\tEnter your ",i+1,"th warrior:",end='')

a=int(input())

Pwarriors.append(Twarriors[a-1])

clear()

print("\n",'\*'\*50)

print('\t\tselcted warriors are:')

for i in range(Element):

print('\t\t',i+1,'th warrior: '+RED + Pwarriors[i]+RESET)

print("\n",'\*'\*50)

**CHAPTER 4**

**RESULTS AND DISCUSSION**

**4.1 Screen Shots**