NextGen Academy

Towards fulfilling a million dreams

Python Programming

Module I: Introduction to Python Programming

June 20, 2023

Contents

This unit covers

- Features/characteristics of Python
- Basic syntax: Writing and executing simple programs
- Basic Data Types and declaring variables
- Performing assignments and arithmetic operations
- Simple input-output operations
- Precedence of operators and type conversion
- Conditional Statements: if, if-else, nested if-else
- Looping: for, while, nested loops
- Terminating loops and skipping specific conditions

1 Introduction:

Python is a widely used high-level programming language created by Guido van Rossum in the late 1980s. The language places strong emphasis on code readability and simplicity, making it possible for programmers to develop applications rapidly.

Like all high level programming languages, Python code resembles the English language which computers are unable to understand. Codes that we write in Python have to be interpreted by a special program known as the Python interpreter, which we'll have to install before we can code, test and execute our Python programs.

There are also a number of third-party tools, such as Py2exe or Pyinstaller that allow us to package our Python code into stand-alone executable programs for some of the most popular operating systems like Windows and Mac OS. This allows us to distribute our Python programs without requiring the users to install the Python interpreter.

1.1 Features in Python

In this, we will see what are the features of Python programming language.

1. **Free and Open Source :** Python language is freely available at the official website and you can download it from online source. Since it is open-source, this means that source code is also available to the public. So you can download it, use it as well as share it.
2. **Easy to code :** Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in the Python language and anybody can learn Python basics in a few hours or days. It is also a developer-friendly language.
3. **Easy to Read :** As you will see, learning Python is quite simple. As was already established, Python's syntax is really straightforward. The code block is defined by the indentations rather than by semicolons or brackets.
4. **Object-Oriented Language :** One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.
5. **GUI Programming Support :** Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in Python. PyQt5 is the most popular option for creating graphical apps with Python.
6. **High-Level Language :** Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.
7. **Large Community Support :** Python has gained popularity over the years. Our questions are constantly answered by the enormous StackOverflow community. These websites have already provided answers to many questions about Python, so Python users can consult them as needed.
8. **Easy to Debug :** Excellent information for mistake tracing. You will be able to quickly identify and correct the majority of your program's issues once you understand how to interpret Python's error traces. Simply by glancing at the code, you can determine what it is designed to perform.
9. **Python is a Portable language :** Python language is also a portable language. For example, if we have Python code for Windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do

not need to change it, we can run this code on any platform.

10. **Python is an Integrated language :** Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.
11. **Interpreted Language :** Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code this makes it easier to debug our code. The source code of Python is converted into an immediate form called **bytecode**.
12. **Large Standard Library :** Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing. There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.
13. **Dynamically Typed Language :** Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.
14. **Frontend and backend development :** With a new project py script, you can run and write Python codes in HTML with the help of some simple tags <py-script>, <py-env>, etc. This will help you do frontend development work in Python like javascript. Backend is the strong forte of Python it's extensively used for this work cause of its frameworks like Django and Flask.
15. **Allocating Memory Dynamically :** In Python, the variable data type does not need to be specified. The memory is automatically allocated to a variable at runtime when it is given a value. Developers do not need to write int y = 18 if the integer value 15 is set to y. You may just type y=18.

1.2 Basic syntax Writing and executing simple program

There are three ways to invoke python, each with its' own uses. The first way is to type "python" at the shell command prompt. This brings up the python interpreter with a message similar to this one:

Python 2.2.1 (#2, Aug 27 2002, 09:01:47)

[GCC 2.95.4 20011002 (Debian prerelease)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

The three greater-than signs (>>>) represent python's prompt; you type your commands after the prompt, and hit return for python to execute them. If you've typed an executable statement, python will execute it immediately and display the results of the statement on the screen. For example, if I use python's print statement to print the famous "Hello, world" greeting, I'll immediately see a response:

>>> print 'hello,world' hello,world

The print statement automatically adds a newline at the end of the printed string. This is true regardless of how python is invoked. (You can suppress the newline by following the string to be printed with a comma.)

When using the python interpreter this way, it executes statements im- mediately, and, unless the value of an expression

is assigned to a variable (See Section 6.1), python will display the value of that expression as soon as it's typed. This makes python a very handy calculator:

>>> cost = 27.00

>>> taxrate = .075

*>>> cost * taxrate 2.025*

*>>> 16 + 25 + 92 * 3*

317

When you use python interactively and wish to use a loop, you must, as always, indent the body of the loop consistently when you type your statements. Python can't execute your statements until the completion of the loop, and as a reminder, it changes its prompt from greater-than signs to periods. Here's a trivial loop that prints each letter of a word on a separate line — notice the change in the prompt, and that python doesn't respond until you enter a completely blank line.

>>> word = 'python'

>>> for i in word:

... print i

...

p y t h o n

The need for a completely blank line is peculiar to the interactive use of python. In other settings, simply returning to the previous level of indenta- tion informs python that you're closing the loop.

You can terminate an interactive session by entering the end-of-file charac- ter appropriate to your system (control-Z for Windows, control-D for Unix), or by entering at the python prompt.

import sys sys.exit()

or

raise SystemExit

For longer programs, you can compose your python code in the editor of your choice, and execute the program by either typing "python", followed by the name of the file containing your program, or by clicking on the file's icon, if you've associated the suffix of your python file with the python in- terpreter. The file extension most commonly used for python files is ".py". Under UNIX systems, a standard technique for running programs written in languages like python is to include a specially formed comment as the first line of the file, informing the shell where to find the interpreter for your program. Suppose that python is installed as /usr/local/bin/python on your system. (The UNIX command "which python" should tell you where python is installed if it's not in /usr/local/bin.) Then the first line of your python program, starting in column 1, should look like this:

#!/usr/local/bin/python

After creating a file, say myprogram.py, which contains the special comment as its first line, you would make the file executable (through the UNIX com- mand "chmod +x myprogram.py"), and then you could execute your pro- gram by simply typing "myprogram.py" at the UNIX prompt.

When you're running python interactively, you can instruct python to ex- ecute files containing python programs with the execfile function. Suppose that you are using python interactively, and wish to run the program you've stored in the file myprog.py. You could enter the following statement:

execfile("myprog.py")

The file name, since it is not an internal python symbol (like a variable name or keyword), must be surrounded by quotes.

2 Basic Data Types

In computer programming, data types specify the type of data that can be stored inside a variable. For example,

num = 24

Here, **24** (an integer) is assigned to the num variable. So the data type of num is of the int class.

2.1 Python Data Types

Data Types	Classes	Description
Numeric	int, float, complex	holds numeric values
String	str	holds sequence of characters
Sequence	list, tuple, range	holds collection of items
Mapping	dict	holds data in key-value pair form
Boolean	bool	holds either True or False
Set	set, frozeenset	hold collection of unique items

Since everything is an object in Python programming, data types are actually classes and variables are instances(object) of these classes.

2.2 Python Numeric Data type

In Python, numeric data type is used to hold numeric values.

Integers, floating-point numbers and complex numbers fall under Python numbers category. They are defined as int, float and complex classes in Python.

- int - holds signed integers of non-limited length.

- float - holds floating decimal points and it's accurate up to **15** decimal places.
- complex - holds complex numbers.

We can use the type() function to know which class a variable or a value belongs to.

Let's see an example,

num1 = 5

print(num1, 'is of type', type(num1))

num2 = 2.0

print(num2, 'is of type', type(num2))

num3 = 1+2j

print(num3, 'is of type', type(num3))

Output

5 is of type <class 'int'>

2.0 is of type <class 'float'>

(1+2j) is of type <class 'complex'>

In the above example, we have created three variables named num1, num2 and num3 with values **5**, **5.0**, and 1+2j respectively.

We have also used the type() function to know which class a certain variable belongs to.

Since,

- **5** is an integer value, type() returns int as the class of num1 i.e <class 'int'>
- **2.0** is a floating value, type() returns float as the class of num2 i.e <class 'float'>
- 1 + 2j is a complex number, type() returns complex as the class of num3 i.e <class 'complex'>

2.3 Python List Data Type

List is an ordered collection of similar or different types of items separated by commas and enclosed within brackets [ ]. For example,

languages = ["Swift", "Java", "Python"]

Here, we have created a list named languages with **3** string values inside it.

2.3.1 Access List Items

To access items from a list, we use the index number **(0, 1, 2 ...)**. For example,

languages = ["Swift", "Java", "Python"]

 # access element at index 0

print(languages[0]) # Swift

 # access element at index 2

print(languages[2]) # Python

In the above example, we have used the index values to access items from the languages list.

- languages[0] - access first item from languages i.e. Swift
- languages[2] - access third item from languages i.e. Python

2.4 Python Tuple Data Type

Tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

In Python, we use the parentheses () to store items of a tuple. For example,

product = ('Xbox', 499.99)

Here, product is a tuple with a string value Xbox and integer value **499.99**.

2.4.1 Access Tuple Items

Similar to lists, we use the index number to access tuple items in Python . For example,

 # create a tuple

product = ('Microsoft', 'Xbox', 499.99)

 # access element at index 0

print(product[0]) # Microsoft

 # access element at index 1

print(product[1]) # Xbox

2.5 Python String Data Type

String is a sequence of characters represented by either single or double quotes. For example,

name = 'Python'

print(name)

message = 'Python for beginners'

print(message)

Output

Python

Python for beginners

In the above example, we have created string-type variables: name and message with values 'Python' and 'Python for beginners' respectively.

2.6 Python Set Data Type

Set is an unordered collection of unique items. Set is defined by values separated by commas inside braces { }. For example,

 # create a set named student_id

student_id = {112, 114, 116, 118, 115}

 # display student_id elements

print(student_id)

 # display type of student_id

print(type(student_id))

Output

{112, 114, 115, 116, 118}

<class 'set'>

Here, we have created a set named student_info with **5** integer values.

Since sets are unordered collections, indexing has no meaning. Hence, the slicing operator [] does not work.

2.7 Python Dictionary Data Type

Python dictionary is an ordered collection of items. It stores elements in key/value pairs.

Here, keys are unique identifiers that are associated with each value.

Let's see an example,

 # create a dictionary named capital_city

capital_city = {'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

print(capital_city)

Output

{'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

In the above example, we have created a dictionary named capital_city. Here,

1. **Keys** are 'Nepal', 'Italy', 'England'
2. **Values** are 'Kathmandu', 'Rome', 'London'

2.7.1 Access Dictionary Values Using Keys

We use keys to retrieve the respective value. But not the other way around. For example,

 # create a dictionary named capital_city

capital_city = {'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

print(capital_city['Nepal']) # prints Kathmandu

print(capital_city['Kathmandu']) # throws error message

Here, we have accessed values using keys from the capital_city dictionary.

Since 'Nepal' is key, capital_city['Nepal'] accesses its respective value i.e. Kathmandu

However, 'Kathmandu' is the value for the 'Nepal' key, so capital_city['Kathmandu'] throws an error message.

3 Declaring variables Performing assignments

Python Variable is containers that store values. Python is not "statically typed". We do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it. A Python variable is a name given to a memory location. It is the basic unit of storage in a program.

3.1 Example of Variable in Python

An Example of a Variable in Python is a representational name that serves as a pointer to an object. Once an object is assigned to a variable, it can be referred to by that name. In layman's terms, we can say that Variable in Python is containers that store values.

Notes:

- *The value stored in a variable can be changed during program execution.*
- *A Variables in Python is only a name given to a memory location, all the operations done on the variable effects that memory location.*

3.1.1 Rules for Python variables

- A Python variable name must start with a letter or the underscore character.
- A Python variable name cannot start with a number.
- A Python variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable in Python names are case-sensitive (name, Name, and NAME are three different variables).
- The reserved words(keywords) in Python cannot be used to name the variable in Python.

Variables Assignment in Python

Here, we will define a variable in python. Here, clearly we have assigned a number, a floating point number, and a string to a variable such as age, salary, and name.

```
# An integer assignment

age = 45

# A floating point

salary = 1456.8

# A string

name = "John"

print(age)

print(salary)

print(name)
```

Output:

45
1456.8
John

Declaration and Initialization of Variables

Let's see how to declare a variable and how to define a variable and print the variable.

```
# declaring the var

Number = 100

# display

print( Number)
```

Output:

100

Redeclaring variables in Python

We can re-declare the Python variable once we have declared the variable and define variable in python already.

```
# declaring the var

Number = 100

# display

print("Before declare: ", Number)

# re-declare the var

Number = 120.3

print("After re-declare:", Number)
```

Output:

Before declare: 100
After re-declare: 120.3

4 Arithmetic operations

Operators are special symbols that perform some operation on operands and returns the result. For example, 5 + 6 is an expression where + is an operator that performs arithmetic add operation on numeric left operand 5 and the right side operand 6 and returns a sum of two operands as a result.

Python includes the operator module that includes underlying methods for each operator. For example, the + operator calls the operator.add(a,b) method.

Example: Operator Methods

```
import operator

n=5+5

print(n)

n=operator.add(5, 10)

print(n)

n=operator.___add___(5, 20)

print(n)
```

Above, expression 5 + 6 is equivalent to the expression operator.add(5, 6) and operator.__add__(5, 6). Many function names are those used for special methods, without the double underscores (dunder methods). For backward compatibility, many of these have functions with the double underscores kept.

Python includes the following categories of operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Identity Operators
- Membership Test Operators
- Bitwise Operators

4.1 Arithmetic Operators

Arithmetic operators perform the common mathematical operation on the numeric operands.

The arithmetic operators return the type of result depends on the type of operands, as below.

1. If either operand is a complex number, the result is converted to complex;
2. If either operand is a floating point number, the result is converted to floating point;
3. If both operands are integers, then the result is an integer and no conversion is needed.

The following table lists all the arithmetic operators in Python:

Operation	Operator	Function	Example in Python Shell
Addition: Sum of two operands	+	operator.add(a,b)	x,y= 5,6 print(x + y) #**Output:** 11 import operator operator.add(5,6) #**Output:** 11
Subtraction: Left operand minus right operand	-	operator.sub(a,b)	x,y =5,6 print(x - y) #**Output:** -1 import operator operator.sub(10, 5) #**Output:** 5
Multiplication	*	operator.mul(a,b)	x,y =5,6 print(x * y) #**Output:** 30 import operator operator.mul(5,6) #**Output:** 30

Operation	Operator	Function	Example in Python Shell
Exponentiation: Left operand raised to the power of right	**	operator.pow(a,b)	x = 2; y = 3 print(x ** y) #**Output:** 8 import operator operator.pow(2, 3) #**Output:** 8
Division	/	operator.truediv(a,b)	x = 6; y = 3 print(x / y) #**Output:** 2 import operator operator.truediv(6, 3) #**Output:** 2

Floor division: equivilant to math.floor(a/b)	//	operator.floordiv(a,b)	x = 6; y = 5
			print(x // y) #**Output:** 1
			import operator
			operator.floordiv(6,5) #**Output:** 1

Modulus: Reminder of a/b	%	operator.mod(a, b)	x = 11; y = 3 print(x % y) #**Output:** 12 import operator operator.mod(11, 3) #**Output:** 2

4.2 Assignment Operators

The assignment operators are used to assign values to variables. The following table lists all the arithmetic operators in Python:

Operator	Function	Example in Python Shell
=		x = 5; x 5
+=	operator.iadd(a,b)	x = 5 print(x += 5) #**Output:** 10 import operator x = operator.iadd(5, 5) #**Output:** 10
-=	operator.isub(a,b)	x = 5 print(x -= 2) #**Output:** 3 import operator x = operator.isub(5,2)

Operator	Function	Example in Python Shell
*=	operator.imul(a,b)	x = 2 print(x *= 3) #**Output:** 6 import operator x = operator.imul(2, 3)
/=	operator.itruediv(a,b)	x = 6 print(x /= 3) #**Output:** 2 import operator x = operator.itruediv(6, 3)
//=	operator.ifloordiv(a,b)	x = 6 print(x //= 5) #**Output:** 1 import operator operator.ifloordiv(6,5)
%=	operator.imod(a, b)	x = 11 print(x %= 3) #**Output:** 2 import operator operator.imod(11, 3) #**Output:** 2
&=	operator.iand(a, b)	x = 11 print(x &= 3) #**Output:** 1 import operator operator.iand(11, 3) #**Output:** 1
|=	operator.ior(a, b)	x = 3 print(x |= 4) #**Output:** 7 import operator operator.mod(3, 4) #**Output:** 7
^=	operator.ixor(a, b)	x = 5 print(x ^= 2) #**Output:** 7 import operator operator.ixor(5, 2) #**Output:** 7
>>=	operator.irshift(a, b)	x = 5 print(x >>= 2) #**Output:** 1 import operator operator.irshift(5, 2) #**Output:** 1
«=	operator.ilshift(a, b)	x = 5 print(x «= 2) #**Output:** 20 import operator operator.ilshift(5, 2) #**Output:** 20

4.3 Comparison Operators

The comparison operators compare two operands and return a boolean either True or False. The following table lists comparison operators in Python.

Operator	Function	Description	Example in Python Shell
>	operator.gt(a,b)	True if the left operand is higher than the right one	x,y =5,6 print(x > y) #**Output:** False import operator operator.gt(5,6) #**Output:** False
<	operator.lt(a,b)	True if the left operand is lower than right one	x,y =5,6 print(x < y) #**Output:** True import operator operator.add(5,6) #**Output:** True
==	operator.eq(a,b)	True if the operands are equal	x,y =5,6 print(x == y) #**Output:** False import operator operator.eq(5,6) #**Output:** False
!=	operator.ne(a,b)	True if the operands are not equal	x,y =5,6 print(x != y) #**Output:** True import operator operator.ne(5,6) #**Output:** True
>=	operator.ge(a,b)	True if the left operand is higher than or equal to the right one	x,y =5,6 print(x >= y) #**Output:** False import operator operator.ge(5,6) #**Output:** False
<=	operator.le(a,b)	True if the left operand is lower than or equal to the right one	x,y =5,6 print(x <= y) #**Output:** True import operator operator.le(5,6) #**Output:** True

4.4 Logical Operators

The logical operators are used to combine two boolean expressions. The logical operations are generally applicable to all objects, and support truth tests, identity tests, and boolean operations.

Operator	Description	Example
and	True if both are true	x,y =5,6 print(x > 1 and y <10) #**Output:** True
or	True if at least one is true	x,y =5,6 print(x > 6 or y <10) #**Output:** True
not	Returns True if an expression evalutes to false and vice-versa	x = 5 print(not x > 1) #**Output:** False

4.5 Identity Operators

The identity operators check whether the two objects have the same id value e.i. both the objects point to the same memory location.

Operator	Function	Description	Example in Python Shell
is	operator.is_(a,b)	True if both are true	x,y =5,6 print(x is y) #**Output:** False import operator operator.is_(x,y) #**Output:** False
is not	operator.is_not(a,b)	True if at least one is true	x,y =5,6 print(x is not y) #**Output:** True import operator operator.is_not(x, y) #**Output:** True

4.6 Membership Test Operators

The membership test operators in and not in test whether the sequence has a given item or not. For the string and bytes types, x in y is True if and only if x is a substring of y.

Operator	Function	Description	Example in Python Shell
in	operator.contains(a,b)	Returns True if the sequence contains the specified item else returns False.	nums = [1,2,3,4,5] print(1 in nums) #**Output:** True print(10 in nums) #**Output:** False print('str' in 'string') #**Output:** True import operator operator.contains(nums, 2) #**Output:** True

Operator	Function	Description	Example in Python Shell
not in	not operator.contains(a,b)	Returns True if the sequence does not contains the specified item, else returns False.	nums = [1,2,3,4,5] print(1 not in nums) #**Output:** False print(10 not in nums) #**Output:** True print('str' not in 'string') #**Output:** False import operator not operator.contains(nums, 2) #**Output:** False

4.7 Bitwise Operators

Bitwise operators perform operations on binary operands.

Operator	Function	Description	Example in Python Shell
&	operator.and_(a,b)	Sets each bit to 1 if both bits are 1.	x=5; y=10 z=x & y print(z) #**Output:** 0 import operator operator.and_(x, y)
|	operator.or_(a,b)	Sets each bit to 1 if one of two bits is 1.	x=5; y=10 z=x | y print(z) #**Output:** 15 import operator operator.or_(x, y)
^	operator.xor(a,b)	Sets each bit to 1 if only one of two bits is 1.	x=5; y=10 z=x ^ y print(z) #**Output:** 15 import operator operator.xor(x, y)
~	operator.invert(a)	Inverts all the bits.	x=5 print(~x) #**Output:** -6 import operator operator.invert(x)
«	operator.lshift(a,b)	Shift left by pushing zeros in from the right and let the leftmost bits fall off.	x=5 print(x«2) #**Output:** 20 import operator operator.lshift(x,2)

Operator	Function	Description	Example in Python Shell
>>	operator.rshift(a,b)	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off.	x=5 print(x>>2) #**Output:** 1 import operator operator.rshift(x,2)

5 Simple input-output

Display Output

The print() funtion in Python displays an output to a console or to the text stream file. You can pass any type of data to the print() function to be displayed on the console.

Example: Display Output

print('Hello World!')

print(1234)

print(True)

In Python shell, it echoes the value of any Python expression.

Display Output

The print() function can also display the values of one or more variables separated by a comma.

Example: Display Variables

name="Ram"

print(name) # display single variable

age=21

print(name, age)# display multiple variables

print("Name:", name, ", Age:",age) # display formatted output

By default, a single space ' ' acts as a separator between values. However, any other character can be used by providing a sep parameter. Visit the print() funtion for more information.

5.1 Getting User's Input

The input() function is used to get the user's input. It reads the key strokes as a string object which can be referred to by a variable having a suitable name.

Taking User's Input

Note that the blinking cursor waits for the user's input. The user enters his input and then hits Enter. This will be captured as a string.

In the above example, the input() function takes the user's input from the next line, e.g. 'Steve' in this case. input() will capture it and assign it to a name variable. The name variable will display whatever the user has provided as the input.

The input() function has an optional string parameter that acts as a prompt for the user.

Taking User's Input

The input() function always reads the input as a string, even if comprises of digits. Visit input() function for more information.

5.2 Python Statements

Python statement ends with the token NEWLINE character (carriage return). It means each line in a Python script is a statement. The following Python script contains three statements in three separate lines.

Example: Python Statements

```
print('id: ', 1)

print('First Name: ', 'Steve')

print('Last Name: ', 'Jobs')
```

Use backslash character \ to join a statement span over multiple lines, as shown below.

Example: Python Statements

```
print('id: ',\

1)

print('First Name: ',\

'Ste\

ve')

print('Last \

Name: ',

'Jobs')
```

Expressions in parentheses (), square brackets [ ], or curly braces { } can be spread over multiple lines without using backslashes.

Example: Multi-line Expression

list = [1, 2, 3, 4

5, 6, 7, 8,

9, 10, 11, 12]

Please note that the backslash character spans a single line in one logical line and multiple physical lines, but not the two different statements in one logical line.

Use the semicolon ; to separate multiple statements in a single line.

Example: Multiple Statements in Single Line

print('id: ', 1);print('First Name: ', 'Steve');print('Last Name: ', 'Jobs')

5.3 Code Comments in Python

In a Python script, the symbol # indicates the start of a comment line. It is effective till the end of the line in the editor.

Example: Single Line Comment

 # this is a comment

print("Hello World")

print("Welcome to Python Tutorial") #comment after a statement.

In Python, there is no provision to write multi-line comments, or a block comment. For multi-line comments, each line should have the # symbol at the start.

A triple quoted multi-line string is also treated as a comment if it is not a docstring of the function or the class.

Example: Multi-line Comments

'''

comment1

comment2

comment3

'''

6 Precedence of operators

In Python, different operators have different **precedence levels**, which determine the order in which they are executed.

When an expression involves multiple operators, Python will evaluate them based on their precedence levels before producing the final result.

Arithmetic Operators

Arithmetic operators are used for mathematical operations like *addition*, *subtraction*, *multiplication*, and *division*.

Here's a list of arithmetic operators in Python, ordered from **highest** to **lowest** precedence:

1. Exponentiation (**)
2. Unary positive/negative (+x, -x)
3. Multiplication (*), Division (/), Floor Division (//), and Modulus (%)
4. Addition (+) and Subtraction (-)

Examples:

result1 = 8 + 2 * 3 - 4 / 2

2 * 3 = 6, 4 / 2 = 2, 8 + 6 = 14, 14 - 2 = 12

result2 = (3 + 5) * 2 - 6 / 3

3 + 5 = 8, 6 / 3 = 2, 8 * 2 = 16, 16 - 2 = 14

result3 = 4 ** 2 // 3 * 5 % 7

4 ** 2 = 16, 16 // 3 = 5, 5 * 5 = 25, 25 % 7 = 4

result4 = 2 ** 3 * 4 + 5

2 ** 3 = 8, 8 * 4 = 32, 32 + 5 = 37

result5 = 10 / 2 * 3 + 4

10 / 2 = 5.0, 5.0 * 3 = 15.0, 15.0 + 4 = 19.0

Comparison Operators

Comparison operators are used to compare values and produce a **boolean** result (*True* or *False*) based on the outcome of the comparison. These operators have lower precedence than arithmetic operators.

Here's a list of comparison operators in Python, ordered from **highest** to **lowest** precedence:

1. Less than (<)
2. Less than or equal to (<=)
3. Greater than (>)
4. Greater than or equal to (>=)
5. Equal to (==)
6. Not equal to (!=)

When multiple comparison operators are used in an expression, they are evaluated from **left** to **right**.

Examples:

result1 = 3 * 2 > 4 + 2 == 6

3 * 2 = 6, 4 + 2 = 6, so 6 > 6 is False,

6 == 6 is True, the result is False

result2 = 5 < 10 > 3

5 < 10 is True, 10 > 3 is True, so the result is True

result3 = 10 - 3 < 5 != 3 * 2

10 - 3 = 7, 3 * 2 = 6, so 7 < 5 is False,

5 != 6 is True, the result is False

result4 = 7 >= 5 == 2 + 3

7 >= 5 is True, 5 == 2 + 3 is True, so the result is True

result5 = 5 * 2 >= 10 == 2 ** 3

5 * 2 = 10, 2 ** 3 = 8, so 10 >= 10 is True,

10 == 8 is False, the result is False

Logical Operators

Logical operators are used to perform logical operations, such as *AND*, *OR*, and *NOT*, on **boolean** values. They have lower precedence than arithmetic and comparison operators.

Here's a list of logical operators in Python, ordered from **highest** to **lowest** precedence:

1. NOT (**not**)
2. AND (**and**)
3. OR (**or**)

Examples:

result1 = not 5 > 2 and 4 < 6

not (5 > 2) and (4 < 6),

not True and True, False and True, the result is False

result2 = 3 < 7 or 8 > 12 and not 10 == 5 * 2

(3 < 7) or (8 > 12) and not (10 == 5 * 2),

True or False and not True, True or False, the result is True

result3 = not 7 < 5 or 3 * 2 == 6 and 4 + 2 > 5

not (7 < 5) or (3 * 2 == 6) and (4 + 2 > 5),

not False or True and True, True or True, the result is True

result4 = not (10 > 8 and 6 / 3 == 2) or 12 - 3 < 9

not ((10 > 8) and (6 / 3 == 2)) or (12 - 3 < 9),

not (True and True) or False, not True or False, the result is False

result5 = 2 * 2 == 4 and not 6 > 10 or 8 < 5

(2 * 2 == 4) and not (6 > 10) or (8 < 5),

True and not False or False, True and True or False, the result is True

Bitwise Operators

Bitwise operators are used to perform operations on *binary numbers*, or the individual *bits* within integer values. They have higher precedence than logical operators but lower precedence than comparison operators.

Here's a list of bitwise operators in Python, ordered from **highest** to **lowest** precedence:

1. Bitwise NOT (~)
2. Bitwise AND (**&**)
3. Bitwise XOR (^)
4. Bitwise OR (|)

Examples:

result1 = 5 & 3 | 4 ^ 6

(5 & 3) | (4 ^ 6), 1 | 2, the result is 3

result2 = ~4 & 5 | 6 ^ 3

(~4) & 5 | (6 ^ 3), -5 & 5 | 5, 1 | 5, the result is 5

result3 = 7 & 3 ^ 5 | 2

(7 & 3) ^ (5 | 2), 3 ^ 7, the result is 4

result4 = 12 | 5 & 6 ^ 3

(12 | 5) & (6 ^ 3), 13 & 5, the result is 5

result5 = 4 ^ 2 & 6 | 1

(4 ^ 2) & (6 | 1), 6 & 7, the result is 6

Precedence Rules and Hierarchy

In Python, operators have a defined **order of precedence** that determines the evaluation order of expressions.

Here is a summary of the operator precedence hierarchy, ordered from **highest** to **lowest** precedence:

1. Parentheses ()
2. Exponentiation **
3. Bitwise NOT ~, Unary plus +, Unary minus -
4. Multiplication *, Division /, Floor division //, Modulus %
5. Addition +, Subtraction -
6. Bitwise shift left «, Bitwise shift right >>
7. Bitwise AND &
8. Bitwise XOR ^
9. Bitwise OR |
10. Comparison operators (<, <=, >, >=, !=, ==)
11. Identity operators (**is**, **is not**)
12. Membership operators (**in**, **not in**)
13. Logical NOT **not**
14. Logical AND **and**
15. Logical OR **or**

Precedence	Operator	Description
1	()	Parentheses
2	**	Exponentiation
3	~, +, - (unary)	Bitwise NOT, unary plus, and minus
4	*, /, //, %	Multiplication, division, floor division, and modulo
5	+, - (binary)	Addition and subtraction
6	«, >>	Bitwise shift left and right
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	<, <=, >, >=, !=, ==	Comparison operators
11	is, is not	Identity operators
12	in, not in	Membership operators
13	not	Logical NOT
14	and	Logical AND
15	or	Logical OR

Examples of Operator Precedence

Example 1:

*result = 2 + 3 * 4 ** 2 - 1 # 2 + 3 * (4 ** 2) - 1, 2 + 3 * 16 - 1, 2 + 48 - 1, # the result is 49* **Explanation:**

1. Exponentiation: **4 ** 2** evaluates to **16**
2. Multiplication: **3 * 16** evaluates to **48**
3. Addition: **2 + 48** evaluates to **50**
4. Subtraction: **50 - 1** evaluates to **49**

7 Type conversion

In Python, we have the flexibility to convert one data type to another using a specific function. On top of that, we do not need to explicitly define the data type while declaring variables unlike other programming languages like C++ and Java.

There are mainly two types of type conversion methods in Python: implicit type conversion and explicit type conversion.

7.1 Implicit Type Conversion in Python

In Python, when the data type conversion takes place during compilation or during the runtime, it's called an implicit data type conversion. Python handles the implicit data type conversion, so we don't have to explicitly convert the data type into another data type. We have seen various examples of this kind of data type conversion throughout the tutorial. Let's see another example.

Let's add two Python variables of two different data types and store the result in a variable and see if the Python compiler is able to convert the data type of the resultant variable or not.

a = 5

b = 5.5

sum = a + b

print (sum)

print (type (sum)) #type() is used to display the datatype of a variable

Output:

10.5

<class 'float'>

In the above example, we have taken two variables of integer, and float data types and added them. Further, we have declared another variable named 'sum' and stored the result of the addition in it. When we checked the data type of the sum variable, we could see that the data type of the sum variable had been automatically converted into the float data type by the Python compiler. This is called implicit type conversion.

The reason that the sum variable was converted into the float data type and not the integer data type is that if the compiler had converted it into the integer data type, then it would've had to remove the fractional part, which would have resulted in data loss. So, Python always converts smaller data types into larger data types to prevent the loss of data.

7.2 Explicit Type Conversion in Python

Explicit type conversion is also known as typecasting. Explicit type conversion takes place when the programmer clearly and explicitly defines the same in the program. For explicit type conversion, there are some in-built Python functions. **The following table contains some of the in-built functions for type conversion, along with their descriptions.**

Function	**Description**
int(y [base])	It converts y to an integer, and Base specifies the number base. For example, if you want to convert the string into decimal numbers then you'll use 10 as the base.
float(y)	It converts y to a floating-point number.
complex(real [imag])	It creates a complex number.
str(y)	It converts y to a string.
tuple(y)	It converts y to a tuple.
list(y)	It converts y to a list.
set(y)	It converts y to a set.
dict(y)	It creates a dictionary and y should be a sequence of (key, value) tuples.
ord(y)	It converts a character into an integer.
hex(y)	It converts an integer to a hexadecimal string.
oct(y)	It converts an integer to an octal string

Now that we know the in-built functions provided by Python that are used for explicit type conversion, let's see the syntax for explicit type conversion:

(required_data type)(expression)

Let's go over the following examples for explicit type conversion in Python.

```
# adding string and integer data types using explicit type conversion

a = 100

b = "200"

result1 = a + b

b = int(b)

result2 = a + b
```

print(result2)

Output:

Traceback (most recent call last):

File “ ”, line 1, in

TypeError: unsupported operand type(s) for +: ‘int’ and ‘str’

300

In the above example, variable a is of the number data type, and variable b is of the string data type. When we try to add these two integers and store the value in a variable named result1, a TypeError occurs, as shown in the output. So, in order to perform this operation, we have to use explicit type casting.
As we can see in the above code block, we have converted the variable b into an int type and then added variables a and b. The sum is stored in the variable named result2, and when printed, it displays 300 as output, as we can see in the output block.

8 Conditional Statements: if, if else, nested ifelse

In a Python program, the if statement is how you perform this sort of decision-making. It allows for **conditional** execution of a statement or group of statements based on the value of an expression.

8.1 The if statement

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the **if** statement, which has the genaral form:

if BOOLEAN EXPRESSION:

STATEMENTS

A few important things to note about if statements:

1. The colon (:) is significant and required. It separates the **header** of the **compound statement** from the **body**.
2. The line after the colon must be indented. It is standard in Python to use four spaces for indenting.
3. All lines indented the same amount after the colon will be executed whenever the BOOLEAN_EXPRESSION is true.

Here is an example:

food = 'spam'

if food == 'spam':

print('Ummmm, my favorite!')

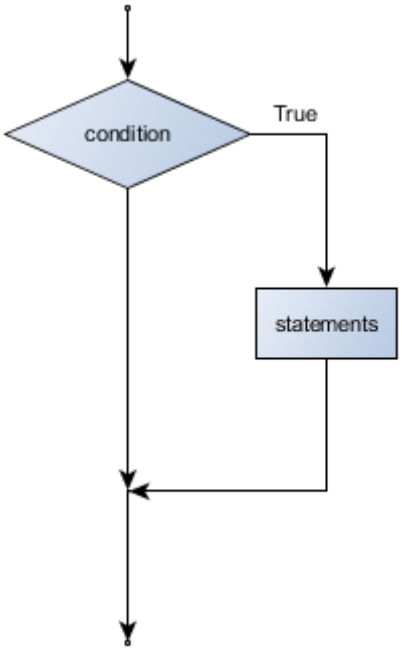
print('I feel like saying it 100 times...')

*print(100 * (food + '! '))*

The boolean expression after the if statement is called the **condition**. If it is true, then all the indented statements get executed. What happens if the condition is false, and food is not equal to 'spam'? In a simple if statement like this, nothing happens, and the program continues on to the next statement.

Run this example code and see what happens. Then change the value of food to something other than 'spam' and run it again, confirming that you don't get any output.

Flowchart of an if statement



As with the for statement from the last chapter, the if statement is a **compound statement**. Compound statements consist of a header line and a body. The header line of the if statement begins with the keyword if followed by a *boolean expression* and ends with a colon (:).

The indented statements that follow are called a **block**. The first unindented statement marks the end of the block. Each statement inside the block must have the same indentation.

Indentation and the PEP 8 Python Style Guide

The Python community has developed a Style Guide for Python Code, usually referred to simply as "PEP 8". The *Python Enhancement Proposals*, or PEPs, are part of the process the Python community uses to discuss and adopt changes to the language.

PEP 8 recommends the use of 4 spaces per indentation level. We will follow this (and the other PEP 8 recommendations) in this book.

To help us learn to write well styled Python code, there is a program called pep8 that works as an automatic style guide checker for Python source code. pep8 is installable as a package on Debian based GNU/Linux systems like Debian.

In the Vim section of the appendix, Configuring Debian for Python Web Development, there is instruction on configuring vim to run pep8 on your source code with the push of a button.

8.2 The if else statement

It is frequently the case that you want one thing to happen when a condition it true, and **something else** to happen when it is false. For that we have the if else statement.

if *food == 'spam':*

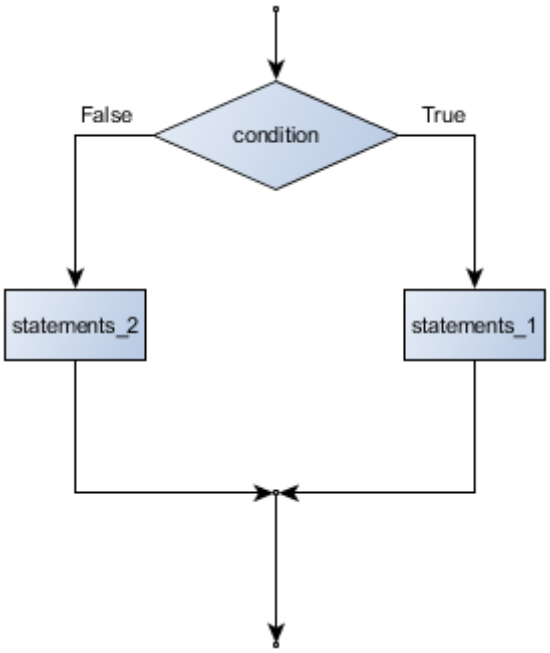
print('Ummmm, my favorite!')

else*:*

print("No, I won't have it. I want spam!")

Here, the first print statement will execute if food is equal to 'spam', and the print statement indented under the else clause will get executed when it is not.

Flowchart of a if else statement



The syntax for an if else statement looks like this:

if *BOOLEAN EXPRESSION:*

STATEMENTS_1 # executed if condition evaluates to True

else*:*

STATEMENTS_2 # executed if condition evaluates to False

Each statement inside the if block of an if else statement is executed in order if the boolean expression evaluates to True. The entire block of statements is skipped if the boolean expression evaluates to False, and instead all the statements under the else clause are executed.

There is no limit on the number of statements that can appear under the two clauses of an if else statement, but there has to be at least one statement in each block. Occasionally, it is useful to have a section with no statements (usually as a place keeper, or scaffolding, for code you haven't written yet). In that case, you can use the pass statement, which does nothing except act as a placeholder.

***if True**: # This is always true*

***pass** # so this is always executed, but it does nothing*

***else**:*

pass

Python terminology

Python documentation sometimes uses the term **suite** of statements to mean what we have called a *block* here. They mean the same thing, and since most other languages and computer scientists use the word *block*, we'll stick with that.

Notice too that else is not a statement. The if statement has two *clauses*, one of which is the (optional) else clause. The Python documentation calls both forms, together with the next form we are about to meet, the if statement.

8.3 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

***if** $x < y$:*

STATEMENTS_A

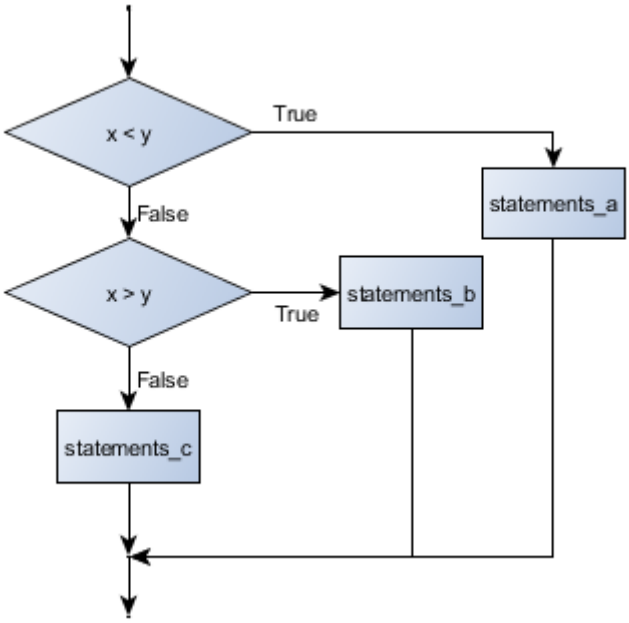
***elif** $x > y$:*

STATEMENTS_B

***else**:*

STATEMENTS_C

Flowchart of this chained conditional



elif is an abbreviation of else if. Again, exactly one branch will be executed. There is no limit of the number of elif statements but only a single (and optional) final else statement is allowed and it must be the last branch in the statement:

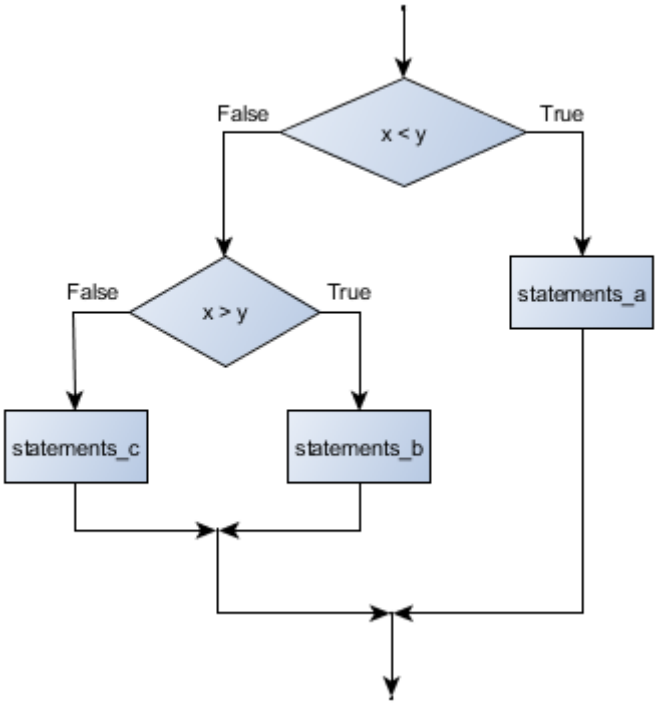
```
if choice == 'a':
    print("You chose 'a'.")
elif choice == 'b':
    print("You chose 'b'.")
elif choice == 'c':
    print("You chose 'c'.")
else:
    print("Invalid choice.")
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

8.4 Nested conditionals

One conditional can also be **nested** within another. (It is the same theme of composibility, again!) We could have written the previous example as follows:

Flowchart of this nested conditional



if *x < y:*

STATEMENTS_A

else*:*

if *x > y:*

STATEMENTS_B

else*:*

STATEMENTS_C

The outer conditional contains two branches. The second branch contains another if statement, which has two branches of its own. Those two branches could contain conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals very quickly become difficult to read. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

if 0 < x: # assume x is an int here

if x < 10:

print("x is a positive single digit.")

The print function is called only if we make it past both the conditionals, so we can use the and operator:

if *0 < x* ***and*** *x < 10:*

print("x is a positive single digit.")

Note

Python actually allows a short hand form for this, so the following will also work:

if *0 < x < 10:*

print("x is a positive single digit.")

9 Looping: for, while,

The for loop

The for loop processes each item in a sequence, so it is used with Python's sequence data types - strings, lists, and tuples.

Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed.

The general form of a for loop is:

for LOOP_VARIABLE **in** SEQUENCE:

STATEMENTS

This is another example of a compound statement in Python, and like the branching statements, it has a header terminated by a colon (:) and a body consisting of a sequence of one or more statements indented the same amount from the header.

The loop variable is created when the for statement runs, so you do not need to create the variable before then. Each iteration assigns the the loop variable to the next element in the sequence, and then executes the statements in the body. The statement finishes when the last element in the sequence is reached.

This type of flow is called a **loop** because it loops back around to the top after each iteration.

for friend **in** ['Margot', 'Kathryn', 'Prisila']:

invitation = "Hi " + friend + ". Please come to my party on Saturday!"

print(invitation)

Running through all the items in a sequence is called **traversing** the sequence, or **traversal**.

You should run this example to see what it does.

Tip

As with all the examples you see in this book, you should try this code out yourself and see what it does. You should also try to anticipate the results before you do, and create your own related examples and try them out as well.

If you get the results you expected, pat yourself on the back and move on. If you don't, try to figure out why. This is the essence of the scientific method, and is essential if you want to think like a computer programmer.

Often times you will want a loop that iterates a given number of times, or that iterates over a given sequence of numbers.

The range function come in handy for that.

```
>>> for i in range(5):
... print('i is now:', i)
...
i is now 0
i is now 1
i is now 2
i is now 3
i is now 4
>>>
```

The while statement

The general syntax for the while statement looks like this:

```
while BOOLEAN_EXPRESSION:
STATEMENTS
```

Like the branching statements and the for loop, the while statement is a compound statement consisting of a header and a body. A while loop executes an unknown number of times, as long at the BOOLEAN EXPRESSION is true.

Here is a simple example:

```
number = 0
prompt = "What is the meaning of life, the universe, and everything? "
while number != "42":
number = input(prompt)
```

Notice that if number is set to 42 on the first line, the body of the while statement will not execute at all.

Here is a more elaborate example program demonstrating the use of the while statement

```
name = 'Harrison'
guess = input("So I'm thinking of person's name. Try to guess it: ")
pos = 0
while guess != name and pos < len(name):
print("Nope, that's not it! Hint: letter ", end='')
```

```
print(pos + 1, "is", name[pos] + ". ", end='')

guess = input("Guess again: ")

pos = pos + 1

if pos == len(name) and name != guess:

print("Too bad, you couldn't get it. The name was", name + ".")

else:

print("\nGreat, you got it in", pos + 1, "guesses!")
```

The flow of execution for a while statement works like this:

1. Evaluate the condition (BOOLEAN EXPRESSION), yielding False or True.
2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute each of the STATEMENTS in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**.

An endless source of amusement for computer programmers is the observation that the directions on shampoo, lather, rinse, repeat, are an infinite loop.

In the case here, we can prove that the loop terminates because we know that the value of len(name) is finite, and we can see that the value of pos increments each time through the loop, so eventually it will have to equal len(name). In other cases, it is not so easy to tell.

What you will notice here is that the while loop is more work for you — the programmer — than the equivalent for loop. When using a while loop one has to control the loop variable yourself: give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.

Choosing between for and while

So why have two kinds of loop if for looks easier? This next example shows a case where we need the extra power that we get from the while loop.

Use a for loop if you know, before you start looping, the maximum number of times that you'll need to execute the body. For example, if you're traversing a list of elements, you know that the maximum number of loop iterations you can possibly need is "all the elements in the list". Or if you need to print the 12 times table, we know right away how many times the loop will need to run.

So any problem like "iterate this weather model for 1000 cycles", or "search this list of words", "find all prime numbers up to 10000" suggest that a for loop is best.

By contrast, if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when this will happen, as we did in the "greatest name" program, you'll need a while loop.

We call the first case **definite iteration** — we have some definite bounds for what is needed. The latter case is called **indefinite iteration** — we're not sure how many iterations we'll need — we cannot even establish an upper bound!

10 Terminating loops, skipping specific conditions

10.1 Break Statement in Python

The break statement is used inside the loop to exit out of the loop. In Python, when a break statement is encountered inside a loop, the loop is immediately terminated, and the program control transfer to the next statement following the loop.

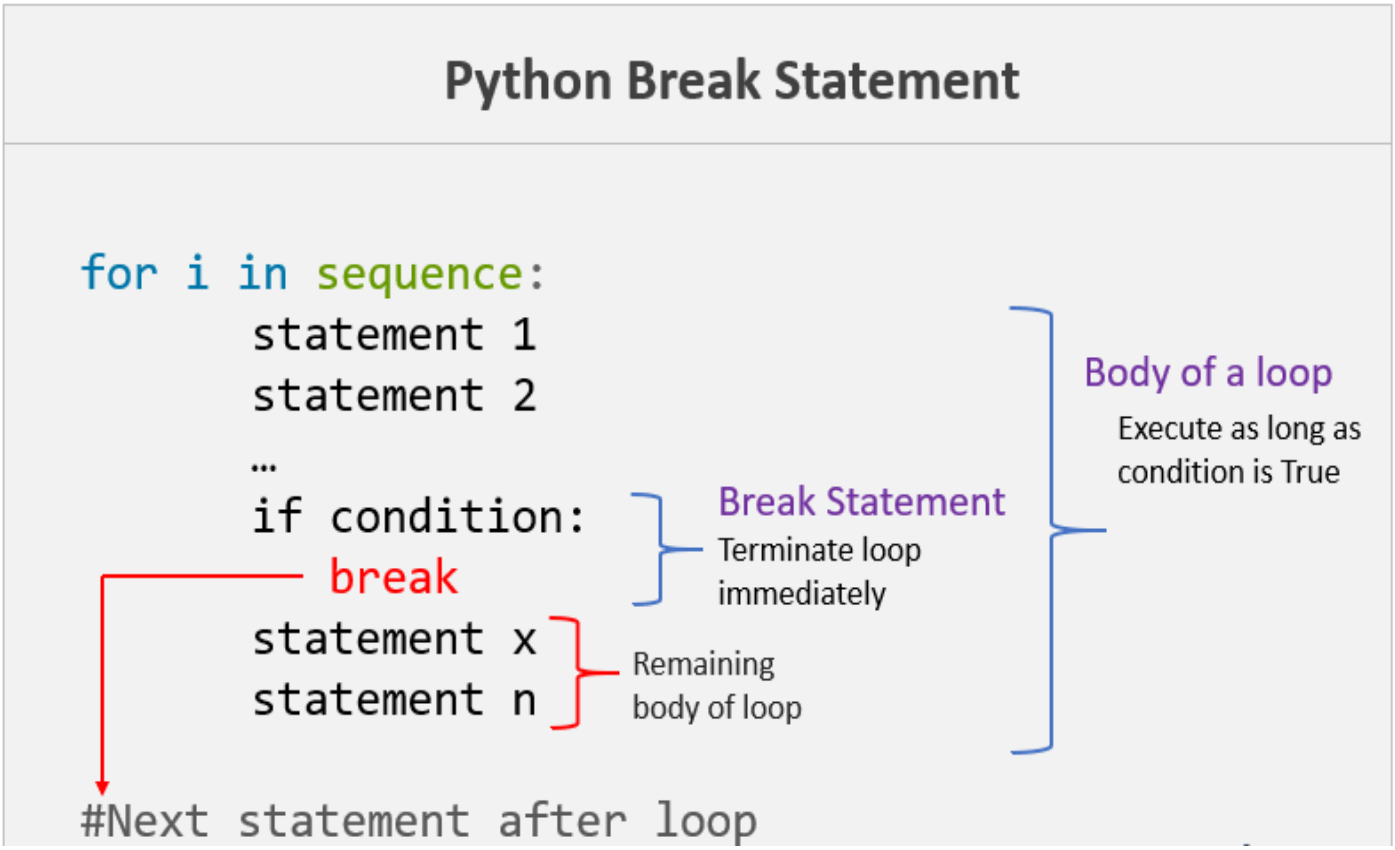
In simple words, A break keyword terminates the loop containing it. If the break statement is used inside a nested loop (loop inside another loop), it will terminate the innermost loop.

For example, you are searching a specific email inside a file. You started reading a file line by line using a loop. When you found an email, you can stop the loop using the break statement.

We can use Python break statement in both for loop and while loop. It is helpful to terminate the loop as soon as the condition is fulfilled instead of doing the remaining iterations. It reduces execution time.

Syntax of break:

break



Example: Break for loop in Python

In this example, we will iterate numbers from a list using a for loop, and if we found a number greater than 100, we will break the loop.

Use the if condition to terminate the loop. If the condition evaluates to true, then the loop will terminate. Else loop will continue to work until the main loop condition is true.

numbers = [10, 40, 120, 230]

for i **in** numbers:

if i > 100:

break

print('current number', i)

Output:

curret number 10

curret number 40

Note: As you can see in the output, we got numbers less than 100 because we used the break statement inside the if condition (the number is greater than 100) to terminate the loop

10.1.1 How break statement works

We used a break statement along with if statement. Whenever a specific condition occurs and a break statement is encountered inside a loop, the loop is immediately terminated, and the program control transfer to the next statement following the loop.

Let's understand the above example iteration by iteration.

- In the first iteration of the loop, 10 gets printed, and the condition i > 100 is checked. Since the value of variable i is 10, the condition becomes false.
- In the second iteration of the loop, 20 gets printed again, and the condition i > 100 is checked. Since the value of i is 40, the condition becomes false.
- In the third iteration of the loop, the condition i > 100 becomes true, and the break statement terminates the loop

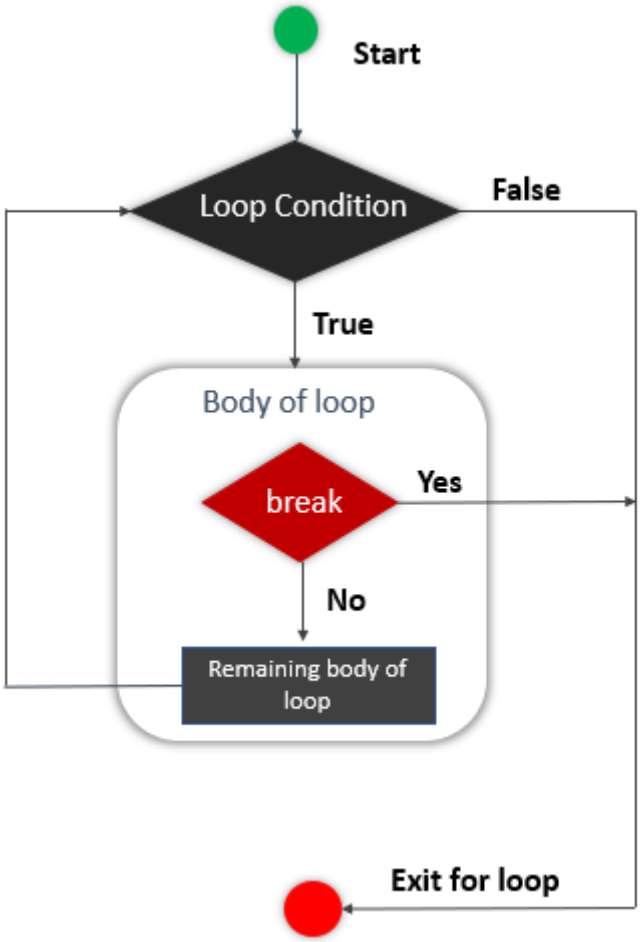
Example: Break while loop

We can use the break statement inside a while loop using the same approach.

Write a while loop to display each character from a string and if a character is a space then terminate the loop.

Use the if condition to stop the while loop. If the current character is space then the condition evaluates to true, then the break statement will execute and the loop will terminate. Else loop will continue to work until the main loop condition is true.

```
name = 'Jesaa29 Roy'

size = len(name)

i = 0

# iterate loop till the last character

while i < size:

# break loop if current character is space

if name[i].isspace():

break

# print current character

print(name[i], end=' ')

i = i + 1
```



Flow chart of a break statement

10.1.2 Break Nested Loop in Python

To terminate the nested loop, use a break statement inside the inner loop. Let's see the example.

In the following example, we have two loops, the outer loop, and the inner loop. The outer for loop iterates the first 10 numbers using the range() function, and the internal loop prints the multiplication table of each number.

But if the current number of both the outer loop and the inner loop is greater than 5 then terminate the inner loop using the break statement.

Example: Break nested loop

```
for i in range(1, 11):
print('Multiplication table of', i)
for j in range(1, 11):
# condition to break inner loop
if i > 5 and j > 5:
break
```

print(i * j, end=' ')

print('')

10.1.3 Break Outer loop in Python

To terminate the outside loop, use a break statement inside the outer loop. Let's see the example.

In the following example, we have two loops, the outer loop, and the inner loop. The outer loop iterates the first 10 numbers, and the internal loop prints the multiplication table of each number.

But if the current number of the outer loop is greater than 5 then terminate the outer loop using the break statement.

Example: Break outer loop

for i **in range**(1, 11):

condition to break outer loop

if i > 5:

break

print('Multiplication table of', i)

for j **in range**(1, 11):

print(i * j, end=' ')

print('')

10.2 Continue Statement in Python

The continue statement skip the current iteration and move to the next iteration. In Python, when the continue statement is encountered inside the loop, it skips all the statements below it and immediately jumps to the next iteration.

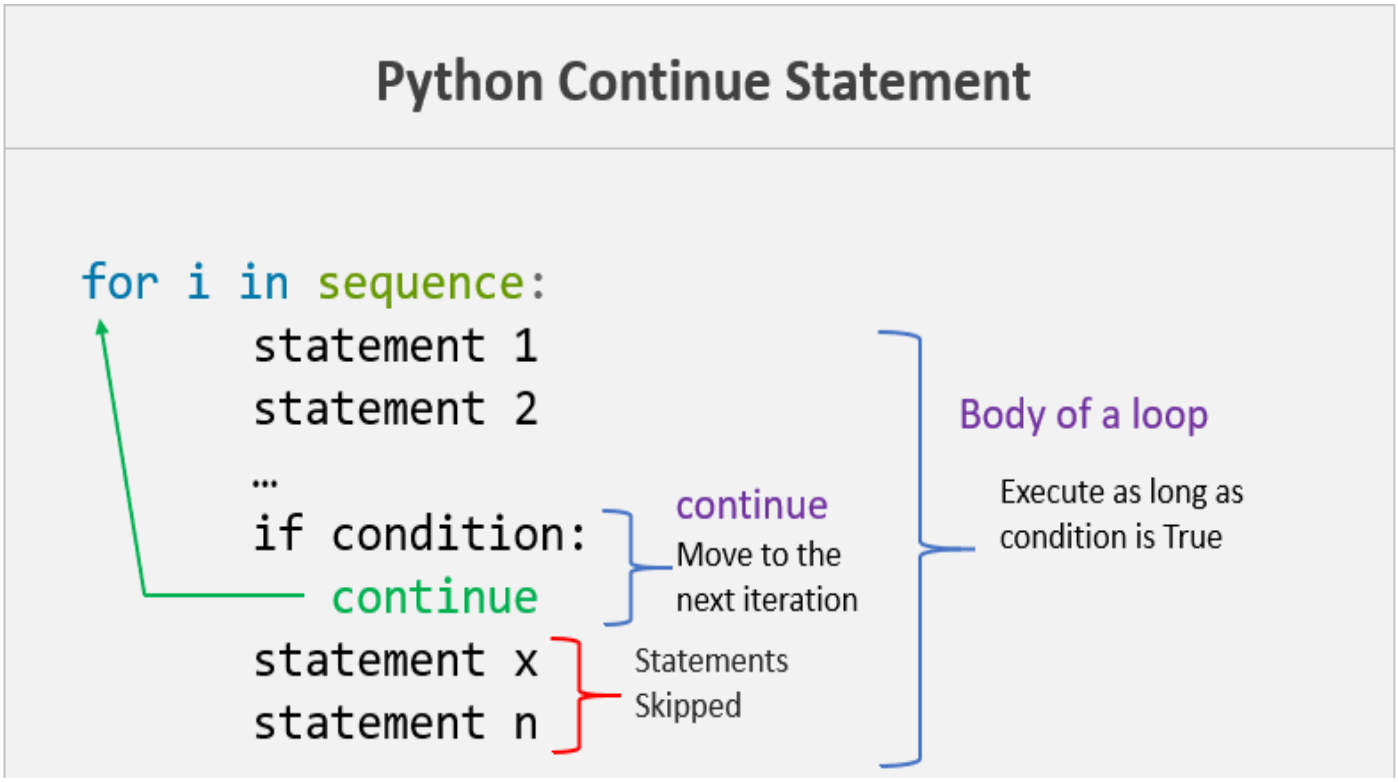
In simple words, the continue statement is used inside loops. Whenever the continue statement is encountered inside a loop, control directly jumps to the start of the loop for the next iteration, skipping the rest of the code present inside the loop's body for the current iteration.

In some situations, it is helpful to skip executing some statement inside a loop's body if a particular condition occurs and directly move to the next iteration.

Syntax of continue:

continue

Let us see the use of the continue statement with an example.



Example: continue statement in for loop

In this example, we will iterate numbers from a list using a for loop and calculate its square. If we found a number greater than 10, we will not calculate its square and directly jump to the next number.

Use the if condition with the continue statement. If the condition evaluates to true, then the loop will move to the next iteration.

```
numbers = [2, 3, 11, 7]

for i in numbers:

print('Current Number is', i)

# skip below statement if number is greater than 10

if i > 10:

continue

square = i * i

print('Square of a current number is', square)
```

Output:

```
Current Number is 2

Square of a current number is 4

Current Number is 3
```

Square of a current number is 9

Current Number is 11

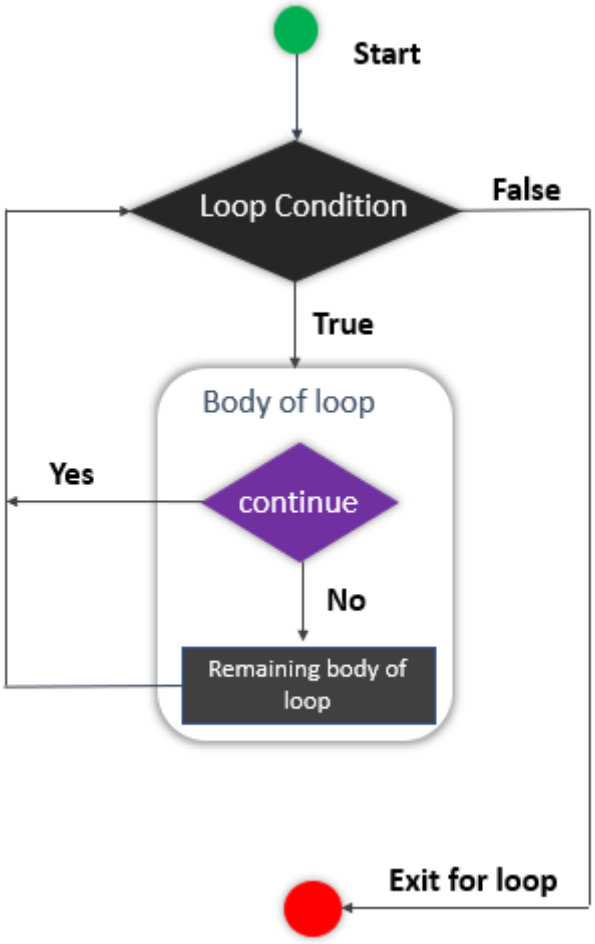
Current Number is 7

Square of a current number is 49

Note: As you can see in the output, we got square of 2, 3, and 7, but the loop ignored number 11 because we used the if condition to check if the number is greater than ten, and the condition evaluated to true, then loop skipped calculating the square of 11 and moved to the next number.

10.2.1 How continue statement works

We used the continue statement along with the if statement. Whenever a specific condition occurs and the continue statement is encountered inside a loop, the loop immediately skips the remeaning body and move to the next iteration.



Flow chart of a continue statement

Let's understand the above example iteration by iteration.

- In the first iteration of the loop, 4 gets printed, and the condition i > 10 is checked. Since the value of i is 2, the condition becomes false.
- In the second iteration of the loop, 9 gets printed, and the condition i > 10 is checked. Since the value of i is 9, the

condition becomes false.

- In the third iteration of the loop, the condition i > 10 becomes true, and the continue statement skips the remaining statements and moves to the next iteration of the loop
- In the second iteration of the loop, 49 gets printed, and the condition i > 10 is checked. Since the value of i is 7, the condition becomes false.

Example: continue statement in while loop

We can also use the continue statement inside a while loop. Let's understand this with an example.

Write a while loop to display each character from a string and if a character is a space, then don't display it and move to the next character.

Use the if condition with the continue statement to jump to the next iteration. If the current character is space, then the condition evaluates to true, then the continue statement will execute, and the loop will move to the next iteration by skipping the remeaning body.

```
name = 'Je sa a'

size = len(name)

i = -1

# iterate loop till the last character

while i < size - 1:

i = i + 1

# skip loop body if current character is space

if name[i].isspace():

continue

# print current character

print(name[i], end=' ')
```

Output:

```
J e s a a
```

10.2.2 Continue Statement in Nested Loop

To skip the current iteration of the nested loop, use the continue statement inside the body of the inner loop. Let's see the example.

In the following example, we have the outer loop and the inner loop. The outer loop iterates the first 10 numbers, and the internal loop prints the multiplication table of each number.

But if the current number of the inner loop is equal to 5, then skip the current iteration and move to the next iteration of the inner loop using the continue statement.

Example: continue statement in nested loop

```
for i in range(1, 11):
print('Multiplication table of', i)
for j in range(1, 11):
# condition to skip current iteration
if j == 5:
continue
print(i * j, end=' ')
print('')
```

10.2.3 Continue Statement in Outer loop

To skip the current iteration of an outside loop, use the continue statement inside the outer loop. Let's see the example

In the following example, The outer loop iterates the first 10 numbers, and the internal loop prints the multiplication table of each number.

But if the current number of the outer loop is even, then skip the current iteration of the outer loop and move to the next iteration.

Note: If we skip the current iteration of an outer loop, the inner loop will not be executed for that iteration because the inner loop is part of the body of an outer loop.

Example: continue statement in outer loop

```
for i in range(1, 11):
# condition to skip iteration
# Don't print multiplication table of even numbers
if i % 2 == 0:
continue
print('Multiplication table of', i)
for j in range(1, 11):
print(i * j, end=' ')
```

```
print('')
```