Python

Programming

For Absolute Beginners

Haider Ali

Dedicated to all those children who don't have access to quality education.

PREFACE

I am delighted to present my first book on Python Programming. The main idea behind writing this book is to address a serious issues regarding our Computer Science curriculum at matric and FSc level of KP Textbook board. Contents of the curriculum are far away from what is taught internationally, for example adding Microsoft Office in the course makes no sense.

In my opinion we need to start with programming courses and this book is my effort to address the issue.

The idea of writing this tiny book is taken back from my experience when Dr. Lutfullah Kakakhel wrote a tiny 40-pages book on C language in 1991 when I was MSc Computer Science student. That book, although, very tiny in size was of great help for us.

About the Author

Haider Ali is a developer with 32 years of development experience, ranging for desktop applications to web and mobile apps.

His experience in Python include developing streamlit and Django apps including AI applications using LLMs.

Introduction

Python is very powerful programming language. Python can be used as a tool to develop large range of application including desktop, web and mobile apps. An extensive library of tools makes Python very powerful.

Whom is this book for?

This book is for beginners who want to start writing computer programs. This book may also be used as a first book for programming courses taught at school at college level in Pakistan.

Conventions

The book is written as a tutorial for beginners starting from the very basics and then going into details of the topic. Programming examples are given with each topic along with the output of the program.

Course Material

Course material including programming examples, Book in PDF format and tutorial articles are available online.

Coding examples are available at GitHub at:

https://github.com/haideralikhalil/python_book

Book in PDF format is also available on GitHub at:

https://tinyurl.com/2sefjhd2

Tutorial Articles are available on Medium at:
https://medium.com/python-tutorial-beginner-to-advance

Table of Contents

1

Getting Started

In this chapter you will learn basic concepts of programming. You will learn which tools are required for writing and running a Python program and will be able to write your first Python program.

Introduction

Computers need instructions to operate. The process of creating and giving these instructions to a computer is called programming. List of instructions that perform a specific task is called a computer program, or simply a program.

We need a language to give instructions to computers. Just as we use languages like English or Arabic to talk to people, we use a programming language to communicate with computers. There are hundreds of programming languages, such as Java, C, C++, Python, and Rust.

Basic concepts of programming

Python is a computer programming language but before going into the details of Python programming language, it is important to know the basic concepts of related to programming languages.

It's important to remember that computers only understand machine language, which is made up of binary code. Binary code is a code written as combination of zeros and one (0s and 1s). A program written in binary code might look like:

```
10110000 00000101
00000100 00000011
```

Binary code is extremely difficult for humans to read/understand and write.

To make it easier, computer scientists first created Assembly language. It uses simple commands to do operations. A sample program written in assembly language is as follows:

```
mov rax, 5
add rax, 3
sub rax, 2
mov rbx, rax
```

Explanation of the code:

mov rax, 5: *Put the number 5 in our special box RAX*

add rax, 3: *Add 3 to whatever is in RAX (now it's 8!)*

sub rax, 2: *Subtract 2 from RAX (now it's 6!)*

mov rbx, rax: *Copy the number from RAX to another box RBX*

Note that the code written in Assembly Language is much simpler than collection of raw 0s and 1s, but it is still complex and requires knowledge of the computer's hardware. A special software translates Assembly code into machine language for the computer to run. This software is called Assembler.

To make programming easier for more people, high-level programming languages were developed. These languages, like Python, use English-like words and syntax, making them much easier to learn and use. They allow people to write programs without needing to know the internal details of computer hardware.

Here is an example of code written in a high-level language:

```
x = 10
y = 3
z = x + y
print("Sum is: ", z)
```

This English-like code is called source code.

Compilers and Interpreters

Since computers only understand machine language, the source code must be translated into machine language such that it can be executed by computer. This job of this translation is carried out by a special software program, which can be either a compiler or an interpreter.

There is a slight difference between compiler and interpreter. A compiler translates the entire program into machine language all at once. It creates a new file that the computer can run directly. An interpreter translates and executes the program one line at a time.

A programming language is usually designed to be either compiled or interpreted. Python is an interpreted language. The program that translates and runs Python code is called the Python interpreter.

Writing your first program in Python

In order to write a program in any programming language you need to:

Use a text editor to write source code of the program

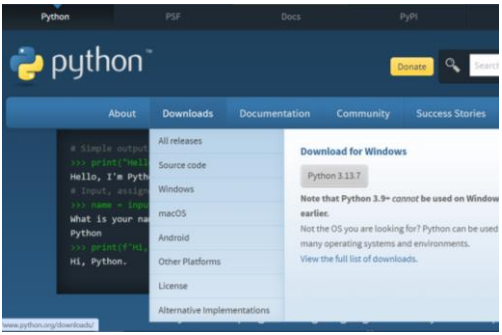
A compiler or interpreter to compile/interpret and run the program

You can write your python program in any text editor like Note Pad in Windows operating system.

In order to run the Python program, you need python interpreter. Follow the following steps to write and execute you first python program.

Step-1: Get Python interpreter

Python interpreter can be downloaded from http://python.org.



In the downloads options select the operating system for which you want to download, in my case it is Windows. Select the required version:

In my case it Windows Installer (64-bit). Download the .exe file (in my case it was *python-3.13.7-amd64.exe*) and execute it. Follow the installation steps.

Confirm python installation

In order to confirm whether Python interpreter has been installed, open the command prompt and give the command *python –version*

```
C:\Users\haider>python --version
Python 3.13.1
```

Step-2: Write Python Program

Python program can be written in any text editor. If you are using Windows, open Notepad and write the following line of code in it.

```
print ("Hello World!")
```

Save the file with the name app.py

Step-3: Run Python Program

To run the program, give the following command in the command prompt:

```
C:\Users\haider>python app.py
Hello World!
```

The python command invokes the python interpreter, that compiles and runs the give .py file, in this case app.py.

Working with Integrated Development Environment (IDE)

While a simple text editor and a Python interpreter are sufficient for writing code, specialized software called

Integrated Development Environments (IDEs) are designed to make the process much more efficient.

IDEs provide a comprehensive set of tools all within a single application, offering numerous features to assist programmers. These features often include code completion, syntax highlighting, debugging tools, and integrated terminal access, which significantly streamline the development process.

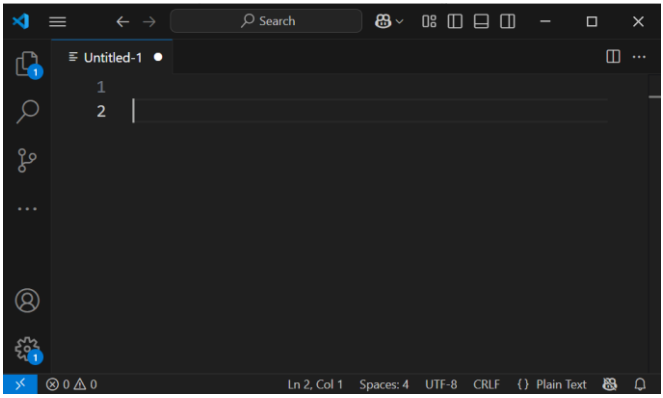
Many powerful IDEs are available for Python development. Popular choices include Visual Studio Code (VS Code), PyCharm, and several others.

I am using VS Code from several years and am much comfortable with it.

How to Install VS Code?

VS Code can be downloaded from
https://code.visualstudio.com/

When installed and run the interface looks like:

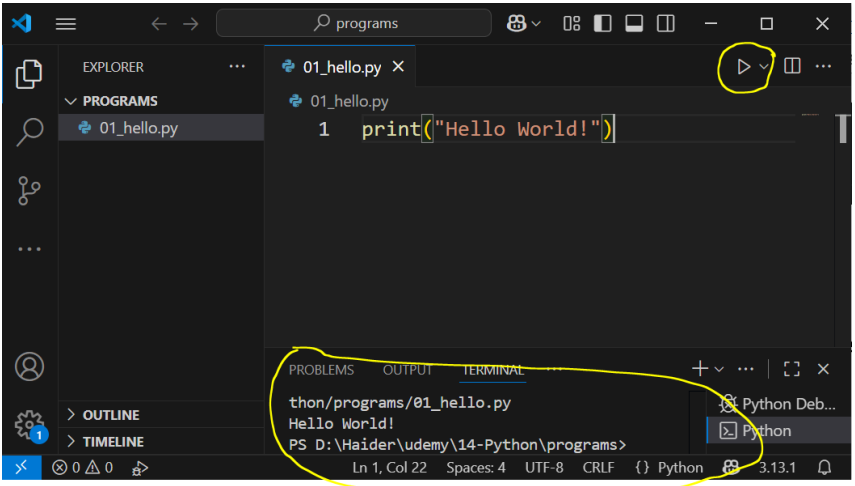


Create a folder (in my case programs) and open it to write your python programs in it.

Write and run Python program in VS Code

Let me write my first program (01_hello.py) in VS code that is stored in the "Programs" folder. The program has only one statement:

```
print("Hello World!")
```



To run Python program, click on the Run button at the top write of the screen. Output is shown in the terminal window of the VS Code.

You may be wondering why I named the file as 01_hello.py? The reason is that since I will be writing example programs for every topic and I want to keep in them in proper sequence when these programs are viewed or listed in a directory.

2

Basic Concepts

In this chapter you will learn basic concepts of Python programming like statement, variables, expressions and operators.

What is a statement?

In programming an instruction is called a statement. In the previous section we used **`print`** statement to print/display the *"Hello World!"* message on the screen.

Statements in a programming language can be categorized as:

Input/output statements: Statements that takes some input form the keyboard or mouse and gives output to an output device. Print in an output statement

Assignment statements: Used to store some values in a variable.

Control statements: Controls the execution of a program

Declarative statement: Used for declaration

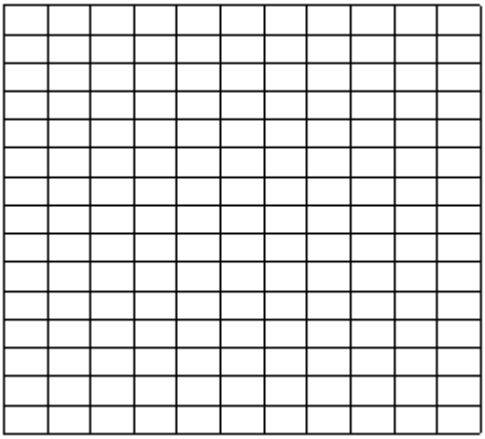
Comments: Used to document a program

Variables in Python

Every computer has a memory called Random Access Memory (RAM) in which data/information can be stored.

Computer Memory

(RAM)



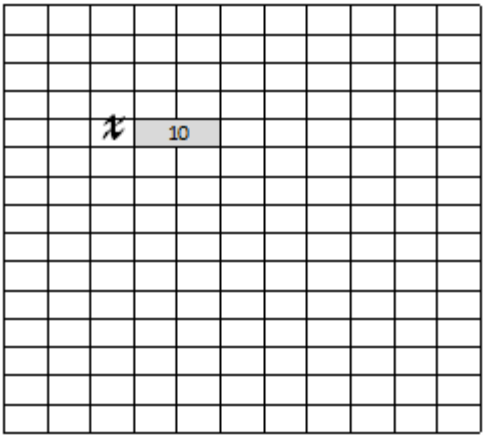
In order to store some data in computer memory (RAM), variables are used in programming languages. Variable is a named memory location in which data can be stored.

Let us write a python program that stores a number 10 in computer memory and then display it con screen.

```
x = 10
print (x)
```

When the above code is executed it displays the number 10 on screen. The statement x = 10 creates a variable named x in computer memory and stores 10 in it.

Variable x with value of 10 created in memory



Variable name conventions

In the previous example we used variable 'x' to store a value. x was a variable name. We can give any name to a variable but there are certain rules while giving names to variables. These rules are:

Variable names can only contain Letters (a-z, A-Z), Digits (0-9) and Underscores (_)

Variable names **Cannot Start with a Digit (0-9).**

Spaces are not allowed. Use underscores to separate words.

Cannot Use Keywords: You cannot use Python's reserved words (keywords) like if, for, while, class, def, etc., as variable names.

Some examples of variable names are *age, gender, value, abc, xyz, a23b,* etc.

Variable names are case-sensitive. Python treats uppercase and lowercase letters as different. age, Age, and AGE are three completely different variables.

Comments

In some cases, explanation is required for a code inside the program. This is done using comments. Comments makes the code more readable. Comments in Python starts with #. Anything written beyond # is not skipped by the Python Interpreter.

```
# This is a comment
x = 10  # this is another comment
```

Python don't allow multiline comments; however, it can be achieved my writing a multiline string by enclosing it in triple quotes """ without assigning it to a variable.

```
"""
This is a multiline comment
in Python.
It can span as many lines as you want
"""
```

Data Types

Each variable in Python has a data type, meaning what type of data is stored in the variable. In the given example i.e. x = 10, the data type of x is integer because 10 is an integer value that is stored in x.

Built-in python data types are:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict

Set Types: set, frozenset
Boolean Type: bool
Binary Types: bytes, bytearray, memoryview
None Type: None

Some examples are given below:

```
age = 62 # integer
pi = 3.27  # float
name = “Ali” # string
passed = True # boolean
```

We will explore most these data types later in the book.

Memory allocation on data types? Operations?

Input from keyboard using **`input`** statement

In Python, input functions is used take input from the keyboard and store it in a variable. In the previous example, we assigned a value of 10 to variable x, but in most of the cases we will need to input data from the keyboard. Python built-in function input is used to get user input using keyboard. The following code asks user to enter some value and then it is displayed on the screen.

```
x = input("Enter a value for x: ")
print(x)
```

```
Enter value for x: 34
34
```

Casting

Casting is used to give a data type to a variable. Sometimes it is important to specify a data type for a variable. Let me explain it with a simple example:

The following code seems OK:

```
x= input("Enter a value for x: ")
y = input("Enter a value for y: ")
z = x / y
print(z)
```

But when you run the code, you will get the errors:

```
Enter a value for x: 5
Enter a value for y: 2
Traceback (most recent call last):
  File "d:\Haider\udemy\14-
Python\programs\06_casting.py", line 3, in
<module>
    z = x / y
        ~~^~~
TypeError: unsupported operand type(s) for /:
'str' and 'str'
```

What is the problem?

When the user enters 5 and 2 from the input in line 1 and 2 respectively the values of x and y are taken as Strings and not numbers. Division cannot be performed on strings.

How to resolve the issue?

We resolve this issue by converting the entered value into a float. This is called casting. The program will be like:

```
x= float(input("Enter a value for x: "))
y = float(input("Enter a value for y: "))
z = x / y
print(z)
```

```
Enter a value for x: 5
Enter a value for y: 2
2.5
```

The **`print`** statement

The `print` statement is one of the most fundamental and frequently used functions in Python. It is used to display output to the standard output device, which is usually your screen.

The basic syntax is very simple: you write `print` and place what you want to display inside the parentheses.

```
print (“Hello World!”)
```

You can print multiple items (like text, numbers, variables) in a single `print` statement by separating them with commas. Python will print them with a single space between each item by default.

```
name = "Alice"
age = 25
print("Name:", name, "Age:", age)
```

Formatted Print

You can also use f for formatted string in the print to print values. The above example can be re-written using the f.

```
name = "Alice"
age = 25
print(f"Name: {name}, Age: {age}")
```

```
Name: Alice, Age: 25
```

Variables are enclosed in braces. When enclosed in braces, value of the variable will be displayed on screen.

3

Assignment Statement and Operators

In this chapter you will learn performing basic arithmetic operations in Python, understand arithmetic operators and work on lists.

The Assignment statement

Assignment statement in Python is used to assign a value to a variable. The general syntax of assignment statement is:

variable = expression

For example: x = 10

We defined variable in the previous chapter. Let us explore what is an expression? An expression is a combination of operators and operands that holds a value. There are different types of expressions in Python including:

- Arithmetic Expressions
- String Expressions
- Logical Expressions

An arithmetic expression is a combination of arithmetic operators and operands. It returns/holds an arithmetic value. Examples of arithmetic expressions:

```
x = 20
y = x + 2
z = x + y * 2 / 10
```

An operand can be a variable or literal (a value like 20, 2.5, "Alice").

Arithmetic Operators

Arithmetic operators are symbols used to perform common mathematical calculations on numerical values (like integers and floats). Arithmetic operators in Python are: + (addition),

- (subtraction), * (multiplication), / (division), // (floor division), % (modulus), and ** (exponentiation).

Example: Program that takes radius of circle as input and calculates and prints area and circumference.

```
# Get input from user and convert it to a float (decimal number)
radius = float (input ("Enter the radius of the circle: "))
# Calculate the area (π * r^2)
area = 3.14159 * (radius ** 2)
# Calculate the circumference (2 * π * r)
circumference = 2 * 3.14159 * radius
# Display the results
print(f"Area = {area:.2f}")
# :.2f means format to 2 decimal places
print(f"Circumference = {circumference:.2f}")
```

```
Enter the radius of the circle: 15
Area = 706.86
Circumference = 94.25
```

Operator Precedence

Operator precedence define the order in which operators operate in the expression. The question of whether 2 + 2 * 2 returns 6 or 8 is answered by operators' precedence.

Python follows the standard mathematical rules for the order of calculations: **P**arentheses > **E**xponents > **M**ultiplication/**D**ivision > **A**ddition/**S**ubtraction.

So 2 + 2 * 2 returns 6 according to the operator precedence rule of Python.

You can use parentheses () to force a different order.

Compound Assignment Operators

Arithmetic operator can be combined with assignment operator to make compound assignment operator. The compound assignment operators are +=, -=, *=, /=, //=, %=, and **=.

The statement num = num + 5 performs exactly the same operation as num += 5.

Some other examples are:

Assignment statement	Equivalent statement
x = x / 2	x /= 2
z = z * 2	z *= 2

Strings in Python

Text data is stored as string in Python. Strings are enclosed in single or double quotation marks.

```
name= 'Bacha Khan'
name = "Bacha Khan"
```

Strings can be concatenated using the + operator

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)
```

```
John Doe
```

Useful String functions

There are useful built-in functions available in Python for manipulating string like slicing string ([:]), converting to lower and upper case, trimming/stripping left and right spaces, and replacing string within the string.

Example:

```
name = "Bacha Khan "
print(name[6:10])
# Prints 'Khan'
print(name.lower())
# Prints 'bacha khan '
print(name.upper())
# Prints 'BACHA KHAN '
print(name.strip())
# Prints 'Bacha Khan'
print(name.replace("Bacha", "Abdul Ghaffar"))
# Prints 'Abdul Ghaffar Khan
```

Lists in Python

Lists are used to store multiple items in one variable. In Python lists are stored in 4 data types i.e. List, Tuple, Set and Dictionary.

Lists are created in Python using [ ] symbols. Some examples of lists in Python are:

```
list1 = ["apple", "banana", "cherry"]
list2 = [43, 35, 2, 19, 30]
list3 = ["Ali", 22, 3.5, "Umer"]
```

In the code above, we have created three lists with names list1, list2 and list3. List1 is a list of strings, list2 is a list of numbers and list3 consists of three different data types i.e. string, integer and float.

Accessing items in the list

Individual item in the list is referred to by its index. In the example above list1[0] refers to "apple" list2[4] refers to 30 and list3[0] refers to "Ali"

New value to any list item can be assigned using its index. For instance, in the above example if we want to replace the value of "apple" to "mango" we will use the following statement:

```
list1[0] = "mango"
```

Adding items in the list

New items can be added to the list using the `append()` method. The following statement add "orange" to the list1.

```
list1.append("orange")
```

Note that append method adds item to the end of the list. If an item has to be added at any given position in the list, `insert()` method is used.

```
players = ["Ali", "Nasir", "Ibrahim", "Umer"]
players.insert(1, "Tahir")
print(players)
```

```
['Ali', 'Tahir', 'Nasir', 'Ibrahim', 'Umer']
```

In this example "Tahir" is added at positon 1 in players list.

Removing item from the list

Item from the list can be removed using the del statement.

```
players = ["Ali", "Nasir", "Ibrahim", "Umer"]
del players[1]
print(players)
```

In this example items at position 1 i.e. “Nasir” is removed in players list.

```
['Ali', 'Ibrahim', 'Umer']
```

Removing all items from the list

`clear()` method is used to remove all items from the list.

```
players = ["Ali", "Nasir", "Ibrahim", "Umer"]
players.clear()
print(players)
```

```
[]
```

4

Control Structures

In this chapter you will learn if-else and match control structures. You will also learn how to run a set of statements multiple times using loops.

What are control structures?

Until now we wrote Python programs in which all the statements in the program are executed one-by-one in a sequential fashion. But we may come across programming problems where we would like to execute one set of statement or another set of statement based on a condition. In some cases, we would like to execute a set of statements more than one time. To address these two types of issues, Python uses control structures, also called control statements or flow control statements.

Flow control statements can be divided in two main categories i.e. conditional execution and loops. Conditional execution is performed using *if-else* and match structures and loops are carried out using *for* and *while* structures.

Conditional execution using if-else

By conditional execution we mean that a set of statements is based on a condition. If a condition becomes true, one set of statement is executed otherwise another set of statement is executed. Syntax of the if structure is:

if condition:

    Statements-1…

else:

    statements-2…

Condition is evaluated, if the condition is true then set of statements-1 are executed and if the condition is false, the set

of statements-2 are executed. Let us understand it with an example.

Example: Write a program that takes age as input from the keyboard. If age is 18 or above, it displays "You are eligible to vote" otherwise it displays "You are not eligible to vote".

```
age = int(input("Enter your age: "))
if age >= 18:
   print("You are eligible to vote.")
else:
   print("You are not eligible to vote.")
```

```
Enter your age: 52
You are eligible to vote.
```

Condition in the if structure is a Logical Expression. Logical Expression is an expression that returns a True to False value. If the value of the logical expression is true, the if: portion is executed; if expression is false, the else: portion is executed.

Logical expression is a combination of relational (comparison) operators and operands. An operand can be a variable or literal. Relational operators are:

> Greater than	<= Less than or equal to
< Less than	== Equal to
>= Greater than or equal to	!= Not Equal to

'if' structures can be nested, i.e. one '**if**' can be nested insider another **if** structure.

```
marks = int(input("Enter your marks: "))
if marks >= 50:
    print("You have passed.")
    if marks >= 90:
        print("You have achieved a distinction.")
    else:
        print("You have not achieved a distinction.")
else:
    print("You have failed.")
    if marks < 0:
        print("Invalid marks entered.")
    else:
        print("You need to score at least 50 to pass.")
```

```
Enter your marks: 80
You have passed.
You have not achieved a distinction.
```

Logical Operators

Two or more logical expressions can be combined using a logical operator. There are three logical operators used in Python i.e. and, or and not.

```
day = int(input("Enter day of week (1-7): "))
if day > 0 and day < 6:
    print("Working Day.")
else:
    if day == 6 or day == 7:
        print("Weekend.")
    else:
        print("Invalid day entered.")
```

```
Enter the day of the week (1-7): 4
Working Day.
```

The **`match`** statement

The **`match`** statement is a simpler way to check a value against many possible options.

Syntax of match statement is:

```
match expression:
  case pattern1:
    # Code block to execute if expression matches pattern1
  case pattern2:
    # Code block to execute if expression matches pattern2
  # ...
  case _:
    # Optional default case, executed if no other pattern matches
```

A simple example of match is given below:

```
day = int(input("Enter day of week (1-7): "))
match day:
 case 1:
  print("Monday")
 case 2:
  print("Tuesday")
 case 3:
  print("Wednesday")
 case 4:
  print("Thursday")
 case 5:
  print("Friday")
 case 6:
  print("Saturday")
 case 7:
  print("Sunday")
 case _:
  print("Invalid day entered.")
```

Multiple patterns/values can be combined in case clause using | (vertical bar symbol) operator.

Example:

```
day = int(input("Enter day of week (1-7): "))
match day:
 case 1 | 2 | 3 | 4 | 5:
  print("Weekday")
 case 6 | 7:
  print("Weekend")
 case _:
  print("Invalid day entered.")
```

Loops

Loop structures are used to execute a single statement multiple times. Loops are achieved with while and for structures.

`while` loop

`while` loop is used to execute a set of statements multiple times. Execution of set of statements is repeated as far as condition with the while loop is true.

Syntax of the while loop is:

`while` *condition:*

set of statements…

Example: Print numbers from 1 to 5

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

```
1
2
3
4
5
```

Using **`continue`** in loop

The **`continue`** statement is used to transfer control to the top of the loop.

Example: Print numbers from 1 to 10 and skip numbers that are divisible by 3.

```
i = 0
while i < 10:
    i += 1
    if i == 3:
        continue
    print(i)
```

```
1
2
4
5
6
7
8
9
10
```

Using **break** to break the loop

If at any point it is required to stop the loop from the loop body, it is achieved using break statement. The break statement immediately breaks the loop and transfer control to the next statement next to the while loop.

Example: Input side width of a square and calculate area of the square. Repeat the process until 0 is entered.

```
while True:
    x = float(input("Enter a number: "))
    if x == 0:
        break
    area = x * x
    print(f"Area of square: {area}")
print("Finished!")
```

```
Enter a number: 3
Area of square: 9.0
Enter a number: 7
Area of square: 49.0
Enter a number: 0
Finished!
```

for Loop

Python has a very powerful for loop. Syntax of the for loop is:

for item in list:

Set of statements

In its simplest form we will use for loop to print numbers from 1 to 5

```
for i in range(1, 6):
    print(i)
```

```
1
2
3
4
5
```

Example-2:

```
players = ['charles', 'martina', 'michael', 'florence', 'elisa']
for player in players:
    print(player)
```

```
charles
martina
michael
florence
elisa
```

`break` and `continue`, explained in previous sections, equally applies to `for` loop as well.

5

Functions

In this chapter you will learn functions in Python, passing arguments to functions and scope of variables.

Functions in Python

A block of code is combined to create a function in Python. Modularity (dividing a bigger problem into small pieces) and code re-usability are the two main advantages of using functions. You are already familiar with using functions. These are the built-in functions you used so far i.e. input(), print() and len() functions.

Functions are defined in Python using the def keyword. Syntax of function definition in Python is:

def function_name([arguments]):

function_body

The function_name is a user-defined name and rules of naming a variable applies to function name as well. Arguments to function are optional and will be explained later in this chapter. *function_body* consists of valid Python statements.

Let us define a simple function.

```
def my_func():
    print("Hello from a function")
```

We have defined a simple function that displays the message "Hello from a function" on the screen. Note that if we have a program that has only the function definition above and we run it, nothing will be displayed. The reason is that we have no called the function. In order to execute a function, we must call it. We can call the function using the function name like my_func(). Note that we may call the function as many times

as we want. In the following code the function is called 3 times and the output can be seen on the screen.

```
def my_func():
    print("Hello from a function")

my_func()
my_func()
my_func()
```

```
Hello from a function
Hello from a function
Hello from a function
```

Arguments to functions

Values can be passed to functions as arguments also called parameters. We will write a simple function greet with one argument.

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
greet("Bob")
greet("Charlie")
```

```
Hello, Alice!
Hello, Bob!
Hello, Charlie!
```

The return statement

`return` statement is used in function to return control to the calling function. When this statement is executed, the function is terminated and control is transferred to the calling program.

Syntax of return statement is:

`return` *[return_value]*

A function may or may not return a value i.e. the return value is optional. Both the following statements are true.

```
return
return 100
```

Functions that returns a value that can be used by the calling program.

Let us define a function that calculates area of a square and returns it to the calling program.

```
def square(x):
   return x * x
area1 = square(5)
area2 = square(10)
print(f"Area of square#1: {area1}")
print(f"Area of square#2: {area2}")
```

```
Area of square#1: 25
Area of square#2: 100
```

Default values to arguments (Optional arguments)

Default values can be given to argument that makes it an optional argument.

```
def square(x=1):
    return x * x
area1 = square(5)
area2 = square()
print(f"Area of square#1: {area1}")
print(f"Area of square#2: {area2}")
```

```
Area of square#1: 25
Area of square#2: 1
```

Note that if value for the optional argument is not provided, the default value is taken.

Scope of variables

Variable scope refers to the **parts of your code** where a variable is accessible and can be used. Not all variables can be accessed from everywhere in your program. Understanding scope prevents errors and helps you write better code.

Think of it like rooms in a house. A item (variable) in your bedroom (local scope) isn't automatically available in the kitchen (a different scope).

Let us explain with a simple example:

```
x = 100
def func():
    print("Inside func():", x)

func()
print("Outside:", x)
```

```
Inside func(): 100
Outside: 100
```

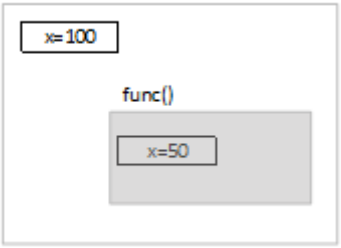
In the given code variable x defined outside function body can be access within the function. Let us change the value of x inside the function body and see what is the effect.

```
x = 100
def func():
   x = 50
   print("Inside func():", x)

func()
print("Outside:", x)
```

```
Inside func(): 50
Outside: 100
```

What happened? We changed the value of x inside the function to 50 but when we try to print it outside the function it is still 100. The reason is that when we assign value of x inside the function, it is not the variable x outside the function that is changed but another variable x created inside the function. So in this case we have now two different variables x, one defined outside the function and the other one defined inside it. In computer memory, it is like:



Global variable

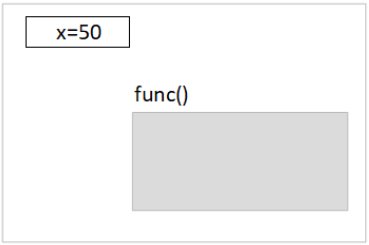
There may be a requirement where we need to have a single variable that can be accessed from any function inside the program scope. This is achieved by using global variable. Global variable is a variable created **in the main body of your code** (outside all functions). It can be accessed from anywhere in your code. The keyword global is added before the variable inside the function body to tell the Python interpreter that the given variable is a global variable and not to create a new local variable. The concept can be understood by modifying the code mentioned above to see the effect.

```
x = 100
def func():
   global x
   x = 50
   print("Inside func():", x)

func()
print("Outside:", x)
```

```
Inside func(): 50
Outside: 50
```

The statement global x means that any access to the variable x in the function referrers to the global variable x and no local variable should be created. The memory will be like:


x=50
func()

6

Object-Oriented Programming (OOP)

In this chapter you will learn the basic concepts of Object Orients Programming and its implementation in Python.

What is Object Oriented Programming?

Object-Oriented Programming (OOP) is a way of writing programs by creating "objects" that model real-world things or concepts. Instead of just writing a list of instructions (procedural programming), you build a set of interacting objects.

Here are some advantages of using OOP:

Reusability: You can write a class once and create many objects from it.

Modularity: Code is organized into clear, self-contained units (classes). It's easier to find and fix bugs.

Scalability: OOP programs are often easier to manage and expand as they become larger and more complex.

Models the Real World: It provides a clear structure for programs that need to model real-world systems with interacting entities.

Python is an object-oriented language. Classes and objects are the core concepts of object-oriented programming.

In OOP, a class is a definition of the object. A class defines the functions and properties of the objects. For example, we can have classes for Car, Student, Fruit etc. An object is an instance of the class, for example objects of the Car class may be Toyota, Honda, and Suzuki, and objects of the Student class may be Ali, Umer, Fatima etc.

Classes and objects in Python

Every language has its own syntax for the definition of classes and objects. Classes are defined in Python using the keyword class. A Student class with two properties i.e. name and age can be defined as follows:

```
class Student:
    name = ""
    age = 0
```

We have a Student class with two properties i.e. name and age. The next step is to instantiate objects of the Student class.

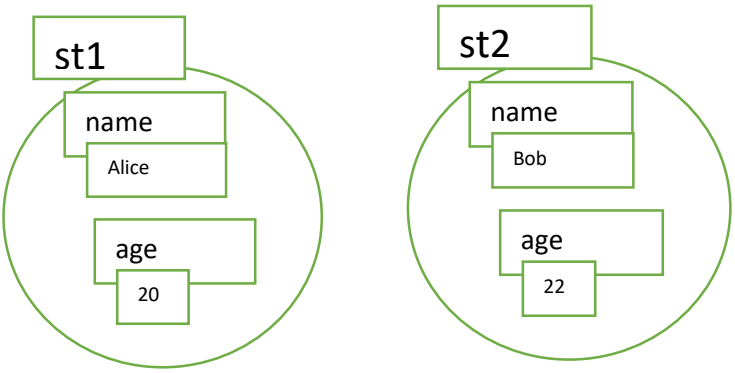
```
st1 = Student()
st2 = Student()
```

These two statements instantiate (creates) two objects of the student class with names st1 and st2.

Properties of the object (name and age) can be access using the dot (.) operator.

```
st1.name = "Alice"
st1.age = 20

st2.name = "Bob"
st2.age = 22
```



To display object properties on screen we use the following code:

```
print(f"Student#1: {st1.name} ({st1.age} years old)")
print(f"Student#2: {st2.name} ({st2.age} years old)")
```

The complete code is as follows:

```
class Student:
    name = ""
    age = 0

st1 = Student()
st2 = Student()

st1.name = "Alice"
st1.age = 20

st2.name = "Bob"
st2.age = 22

print(f"Student#1: {st1.name} ({st1.age} years old)")
print(f"Student#2: {st2.name} ({st2.age} years old)")
```

```
Student#1: Alice (20 years old)
Student#2: Bob (22 years old)
```

Let me give you another example in which we define functions in the class. Functions in the class are also called methods.

```
class Student:
    name = None
    age = None

    def set(self, st_name, st_age):
        self.name = st_name
        self.age = st_age

    def display(self):
        print(f"{self.name} is {self.age} years old.")

st1 = Student()
st2 = Student()

st1.set("Charlie", 23)
st2.set("Diana", 21)

st1.display()
st2.display()
```

In this Student class we have two functions i.e. set() to set values of name and age in the object. Note that the keyword self is used to refer to the object variable. Object variables are the variables that are defined in the class and assosciated with the object (in this case name and age). On the other hand there are local variables that are either defined within the function inside the object or variables passed as parameters/arguments. st_name and st_age are local variables. Object variables are accessed by the `self` keyword.

It may also be noted that the first argument to the functions is self.

Static Method

A **static method** is a method that belongs to a class but does **not** operate on an instance. The keyword self is not

mentioned as first argument to the method. It's like a regular function that you put inside a class because it's logically related to that class. An instance of the class i.e. object is not required to call a static method.

```
class Car:
   def display():
      print("This is a Car")

Car.display()
```

```
This is a Car
```

Constructors

A constructor is a special method that is automatically called when a new **object** (instance) of a **class** is created.

Name of the constructor method __init__.

Let us create a Student class with a constructor.

```
class Student:
   name = None
   age = None
   def __init__(self):
      print("I am constructor!")

st1 = Student()
st2 = Student()
```

```
I am constructor!
I am constructor!
```

Notes that when the objects are instantiated, the constructor method is executed.

Its primary job is to **initialize** the new object by setting its initial state (i.e., by giving its attributes their first values). Let me explain with another example:

```
class Student:
    name = None
    age = None
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

st1 = Student("Alice", 20 )
st2 = Student("Bob", 22 )

st1.display()
st2.display()
```

```
Name: Alice, Age: 20
Name: Bob, Age: 22
```

More advance topics like inheritance and polymorphism will explained in the coming edition of the book.

7

Modules

In this chapter you will learn what is a module? How can it be created and used in programs?

What is module?

Module is a file that contains a set of functions that can be included in a program. A module can be defined as normal python file with functions in it.

A simple example of a module is the following source code saved as my_module.py

```
def hello():
    print("Hello World!")
```

Functions in the module can be used in other python programs. import statement is used to use a module within another module.

If we have another file app.py we will use functions of my_module.py as follows.

```
import my_module
my_module.hello()
```

```
Hello World!
```

We can also rename a module inside our program. The above code can be re-written as follows:

```
import my_module as m
m.hello()
```

In the above code the module my_module has been renamed as m and can refer to it using m.

Classes in modules

Modules can have classes that can be used by programs. The following module module2.py has two classes Car and Bike, each have one method display.

```
class Car:
   def display():
      print("This is a car")
      print("The car has 4 wheels")
      print("The car is used for transportation")
class Bike:
   def display():
      print("This is a bike")
      print("The bike has 2 wheels")
      print("The bike is used for transportation")
```

Methods in the class can be called in the calling program.

```
import module2

module2.Car.display()
module2.Bike.display()
```

```
This is a car
The car has 4 wheels
The car is used for transportation
This is a bike
The bike has 2 wheels
The bike is used for transportation
```

Single class can also be imported from a module. The syntax is:

from module_name ***import*** class_name

Example:

```
from module2 import Car
Car.display()
```

```
This is a car
The car has 4 wheels
The car is used for transportation
```

Built-in Modules

Python comes with many useful built-in modules that can be used in the programs by programmers.

We will use some of these like platform, datetime, math

platform module

The platform module in Python is a built-in module that provides a straightforward way to **retrieve detailed information about the system (platform) on which your program is running**.

This is extremely useful for writing scripts that need to behave differently depending on the operating system, hardware, or Python interpreter.

Here are the key functions, categorized by what they tell you:

Operating System & Version Information

Function	Description	Example Output (on Windows)
platform.system()	Returns the OS name.	'Windows'

platform.release()	Returns the OS release version.	'10'
platform.version()	Returns the OS version string.	'10.0.22631'
platform.platform()	Returns a single string with a detailed description.	'Windows-10-10.0.22631-SP0'
platform.win32_ver()	Windows only. Gets detailed Windows version.	('10', '10.0.22631', 'SP0', 'Multiprocessor Free')
platform.mac_ver()	macOS only. Gets detailed macOS version.	() (empty tuple)

Machine Architecture Information

Function	Description	Example Output
platform.machine()	Returns the machine type (e.g., processor architecture).	'AMD64'
platform.processor()	Returns the real processor name (if possible).	'Intel64 Family 6 Model 142 Stepping 12, GenuineIntel'
platform.architecture()	Returns the bit architecture of the OS and Python executable.	('64bit', 'Wind

Python Interpreter Information

Function	Description	Example Output

platform.python_implementation()	Returns the Python implementation.	'CPython' (most common), 'PyPy', 'Jython'
platform.python_version()	Returns the Python version as a string.	'3.11.4'
platform.python_version_tuple()	Returns the Python version as a tuple.	('3', '11', '4')

Network & Host Information

Function	Description
platform.node()	Returns the computer's network name (hostname).
platform.uname()	Returns a named tuple with system, node, release, version, machine, and processor. This is a summary of many functions above.

Example:

```
import platform

print("=== System Information Report ===")
print(f"Operating System: {platform.system()} {platform.release()}")
print(f"OS Version: {platform.version()}")
print(f"Platform: {platform.platform()}")
print(f"Machine Architecture: {platform.machine()}")
print(f"Processor: {platform.processor()}")
print("---")
print(f"Hostname: {platform.node()}")
print("---")
print(f"Python Implementation:
{platform.python_implementation()}")
print(f"Python Version: {platform.python_version()}")
```

```
=== System Information Report ===
Operating System: Windows 10
OS Version: 10.0.19045
Platform: Windows-10-10.0.19045-SP0
Machine Architecture: AMD64
Processor: Intel64 Family 6 Model 142
Stepping 10, GenuineIntel
---
Hostname: DESKTOP-QI52MAH
---
Python Implementation: CPython
Python Version: 3.13.1
```

The datetime module

The `datetime` module is one of Python's most important built-in modules for working with dates and times. It provides classes for manipulating dates, times, and combinations of both.

The module contains several classes. The most important ones are date, time, datetime and timedelta. Usage of these classes are:

date	working with just dates (year, month, day)
Time	working with just times (hour, minute, second, microsecond), independent of any date
datetime	working with both dates and times. This is the most commonly used class
timedelta	representing a duration (the difference between two dates or times)

Functions of each class are as follows:

Function	Purpose	Example
datetime.now()		2023-11-07 14:30:45.123456
datetime.today()		2023-11-07
date()	Create date	date(1995, 8, 15)
Time()	Create time	time(14, 30, 0)

datetime()	Create date and time	datetime(2024, 1, 1, 0, 0, 0)
now()	Current date and time	
strptime()	format date/time objects into readable strings	
Timedelta()	represents a duration or difference between two dates/times	

A simple program that displays the current date and time is given below:

```
from datetime import datetime
print(datetime.now())
print(datetime.today())
```

```
2025-09-10 13:40:28.600312
2025-09-10 13:40:28.600508
```

Date and time in printed in the format:

year:month:date hour:minute:seconds.microsecods

Note first line in the code i.e. from datetime import datetime

that there is a datetime class in the datetime module.

Formatting Dates and Times (strftime)

The strftime() method allows you to format date/time objects into readable strings.

Mostly used syntax of the stftime functions is:

date_time.strftime(format)

Common format codes are:

- `%Y`: Year (4 digits)
- `%m`: Month (01-12)
- `%d`: Day (01-31)
- `%H`: Hour (00-23)
- `%M`: Minute (00-59)
- `%S`: Second (00-59)
- `%A`: Weekday full name
- `%B`: Month full name

Following examples explains different formats:

```
from datetime import datetime

current_date = datetime.now()
print(current_date.strftime("%Y-%m-%d %H:%M:%S"))
print(current_date.strftime("%d/%m/%Y"))

print(current_date.strftime("%B %dth, %Y"))
print(current_date.strftime("%A, %B %d, %Y"))
```

```
2025-09-10 14:08:06
10/09/2025
September 10th, 2025
Wednesday, September 10, 2025
```

Working with timedelta

'timedelta' represents a duration or difference between two dates/times.

```
from datetime import datetime, timedelta
```

```
today = datetime.now()
print("Today's date is:", today)
print(today + timedelta(days = 15))
print(today + timedelta(weeks = 5))
```

```
Today's date is: 2025-09-10 14:17:22.158972
2025-09-25 14:17:22.158972
2025-10-15 14:17:22.158972
```

`math` Module

`math` module contains a library of common mathematical operations—more advanced than the basic +, -, *, / operators. Functions in the math module can be grouped into several categories:

Number-Theoretic and Representation Functions

These functions work with numbers and their properties.

Function	Description
math.ceil(x)	Returns the smallest integer >= x (rounds up). math.ceil(4.2) returns 5
math.floor(x)	Returns the largest integer <= x (rounds down).

Function	Description
	`math.floor(4.9) returns 4`
`math.fabs(x)`	Returns the absolute value of x (as a float). `math.fabs(-5) returns 5.0`

Power and Logarithmic Functions

Function	Description
`math.sqrt(x)`	Returns the square root of x. `math.sqrt(16) returns 4.0`
`math.pow(x, y)`	Returns x raised to the power y (x^y). `math.pow(2, 3) returns 8.0`
`math.exp(x)`	Returns e raised to the power x (e^x). `math.exp(1) returns ~2.718`
`math.log(x)`	Returns the natural logarithm of x (base e). `math.log(10) returns ~2.302`
`math.log10(x)`	Returns the base-10 logarithm of x. `math.log10(100) returns 2.0`

Trigonometric Functions

Function	Description
math.sin(x)	Returns the sine of x (x in radians).
math.cos(x)	Returns the cosine of x (x in radians).
math.tan(x)	Returns the tangent of x (x in radians).
math.degrees(x)	Converts angle x from radians to degrees.
math.radians(x)	Converts angle x from degrees to radians.

Example

```
import math
print(math.sqrt(16))
print(math.ceil(3.7))
print(math.floor(3.7))
print(math.factorial(5))

print(math.gcd(48, 180))
print(math.lcm(4, 5))
print(math.pi)
print(math.e)
print(math.sin(math.pi / 2))
print(math.log(100, 10))
print(math.pow(2, 3))
print(math.radians(180))
```

```
4.0
4
3
120
12
20
3.141592653589793
2.718281828459045
1.0
2.0
8.0
3.141592653589793
```

8

Third-party modules, pip and virtual environment

In this chapter you will learn using third party modules in your programs. You will also learn using virtual environment in your Python projects.

Third-Party Modules

Python comes with some basic modules like math, datetime etc. Python is open source and allows anyone to write and share modules. The enormous number of modules added by large community of developers have made Python a very rich platform. New modules are added by developers and added to the main repository of modules at pypi.org. Developers can download and use these modules in their applications.

Some of popular modules are:

- numpy & pandas**:** For powerful data analysis and scientific computing.
- requests**:** For making HTTP requests to websites and APIs (much easier than the built-in urllib).
- django & flask**:** For building full-featured web applications.
- pygame**:** For creating games.
- matplotlib & seaborn**:** For creating graphs and visualizations.
- openai**:** For using the OpenAI API (like ChatGPT).

How to download and install 3rd party modules?

Third-party modules are installed using the pip command. Pip stands "Package Installer for Python". Syntax of the pip command is:

pip install package-name

For example, to install pandas library the following command is used:

```
pip install pandas
```

Multiple libraries can be installed using single pip command. Pip command to install numpy and pandas is:

```
pip install pandas numpy
```

The installed libraries can be used in the app using the import statement.

Import pandas

In order to check which libraries are used by the project, pip list command is used.

```
pip list
```

The above command displays the long list of libraries as follows:

```
Authlib                 1.6.1
beautifulsoup4          4.12.3
blinker                 1.9.0
…
numpy                   2.2.1
packaging               24.2
pandas                  2.2.3
pillow                  11.1.0
pip                     24.3.1
platformdirs            4.3.6
…
virtualenv              20.28.0
watchdog                6.0.0
```

A library can be uninstalled using the uninstall option in pip command. To uninstall pandas we use the following command:

```
pip uninstall pandas
```

The requirements.txt file

The **`requirements.txt`** file is a fundamental component of Python projects that helps manage dependencies. It ensures consistent environments across different setups. requirements.txt file is a text file that lists all the Python packages and their specific versions that your project depends on. This allows others to easily install all the necessary dependencies with a single command.

The basic structure of requirements.txt looks like:

```
package1==1.2.3
package2>=4.5.6
package3<7.8.9
package4
```

How to create requirements.txt file?

Requirements.txt file can be created manually or generated from current environment.

Entries are made manually after creating reqirements.txt file. A requirements.txt file might look like:

```
requests==2.28.1
numpy>=1.21.0
pandas
```

requirements.txt file can be created from the current environment using the following pip command:

```
pip freeze > requirements.txt
```

This command captures all installed packages and lists them in the requirements.txt file. Contents created by this command depends upon the installed modules in the current environment. On my system, this command resulted in a long list of contents in my requirements.txt file.

```
altair==5.5.0
attrs==24.3.0
Authlib==1.6.1
...
virtualenv==20.28.0
watchdog==6.0.0
```

The Virtual Environment

A virtual environment is an isolated Python environment that allows you to work on different projects with different dependencies without conflicts. It's one of the most important tools in Python development. It is a self-contained directory that contains:

- A specific Python interpreter version
- Its own set of installed packages
- Isolation from the system-wide Python installation

Why Use Virtual Environments?

Dependency Isolation: Different projects can use different package versions

No System Pollution: Avoid modifying the system Python installation

Reproducibility: Easily share and reproduce development environments

Conflict Prevention: Prevent version conflicts between projects

Clean Testing: Test packages without affecting other projects

Creating Virtual Environment

There are two methods for creating virtual environment i.e using Python built-in utility venv and third-party virtualenv.

Method 1: Using **venv** (Built-in Python 3.3+)

Step-1: Create the virtual environment

Virtual Environment can be created using Python built-in utility venv using the following command.

```
python –m venv environment_directory
```

To create a virtual environment with the name myenv the following command is used:

```
python –m venv myenv
```

This command creates a virtual environment with the name myenv. Subbfolders in the myenv folder are:

```
myenv
        Include
        Lib
        Scripts
        .gitignore
        pyenv.cfg
```

Step-2: Activating the virtual environment

In order to use Virtual environment, it needs to be activated. Virtual environment is activated using running the activate.bat file in the 'scripts' folder. The following command activates the virtual environment myenv:

```
myenv\Scripts\activate
```

Method 2: Using **virtualenv** (Third-Party)

Step-1: Install virtualenv module

The first step is to install the module virtualenv

```
pip install virtualenv
```

Step-2: Create the virtual environment

Virtual Environment can be created using Python built-in utility venv using the following command.

virtualenv environment_directory

To create a virtual environment with the name myenv the following command is used:

```
virtualenv myenv
```

This command creates a virtual environment with the name myenv

Step-3: Activating the virtual environment

In order to use Virtual environment, it needs to be activated. Virtual environment is activated using running the activate.bat file in the 'scripts' folder. The following command activates the virtual environment myenv:

```
myenv\Scripts\activate
```