Basic Computer Coding:
Python

2nd Edition

BIBLIOTEX
Digital Library

BASIC COMPUTER CODING: PYTHON

2nd Edition

BIBLIOTEX
Digital Library

www.bibliotex.com

BASIC COMPUTER CODING: PYTHON

2ND EDITION

www.bibliotex.com

email: info@bibliotex.com

e-book Edition 2022

ISBN: 978-1-98467-604-7 (e-book)

In Collaboration with 3G E-Learning LLC. Originally Published in printed book format by 3G E-Learning LLC with ISBN 978-1-98465-897-5

EDITORIAL BOARD

Aleksandar Mratinković was born on May 5, 1988 in Arandjelovac, Serbia. He has graduated on Economic high school (2007), The College of Tourism in Belgrade (2013), and also has a master degree of Psychology (Faculty of Philosophy, University of Novi Sad). He has been engaged in different fields of psychology (Developmental Psychology, Clinical Psychology, Educational Psychology and Industrial Psychology) and has published several scientific works.

Dan Piestun (PhD) is currently a startup entrepreneur in Israel working on the interface of Agriculture and Biomedical Sciences and was formerly president-CEO of the National Institute of Agricultural Research (INIA) in Uruguay. Dan is a widely published scientist who has received many honours during his career including being a two-time recipient of the Amit Golda Meir Prize from the Hebrew University of Jerusalem, his areas of expertise includes stem cell molecular biology, plant and animal genetics and bioinformatics. Dan's passion for applied science and technological solutions did not stop him from pursuing a deep connection to the farmer, his family and nature. Among some of his interest and practices counts enjoying working as a beekeeper and onboard fishing.

Hazem Shawky Fouda has a PhD. in Agriculture Sciences, obtained his PhD. From the Faculty of Agriculture, Alexandria University in 2008, He is working in Cotton Arbitration & Testing General Organization (CATGO).

Felecia Killings is the Founder and CEO of LiyahAmore Publishing, a publishing company committed to providing technical and educational services and products to Christian Authors. She operates as the Senior Editor and Writer, the Senior Writing Coach, the Content Marketing Specialist, Editor-in-Chief to the company's quarterly magazine, the Executive and Host of an international virtual network, and the Executive Director of the company's online school for Authors. She is a former high-school English instructor and professional development professor. She possesses a Master of Arts degree in Education and a Bachelor's degree in English and African American studies.

Dr. Sandra El Hajj, Ph.D. in Health Sciences from Nova Southeastern University, Florida, USA is a health professional specialized in Preventive and Global Health. With her 12 years of education obtained from one of the most prominent universities in Beirut, in addition to two leading universities in the State of Florida (USA), Dr. Sandra made sure to incorporate interdisciplinary and multicultural approaches in her work. Her long years of studies helped her create her own miniature world of knowledge linking together the healthcare field with Medical Research, Statistics, Food Technology, Environmental & Occupational Health, Preventive Health and most noteworthy her precious last degree of Global Health. Till today, she is the first and only doctor specialized in Global Health in the Middle East area.

Fozia Parveen has a Dphil in Sustainable Water Engineering from the University of Oxford. Prior to this she has received MS in Environmental Sciences from National University of Science and Technology (NUST), Islamabad Pakistan and BS in Environmental Sciences from Fatima Jinnah Women University (FJWU), Rawalpindi.

Igor Krunic 2003-2007 in the School of Economics. After graduating in 2007, he went on to study at The College of Tourism, at the University of Belgrade where he got his bachelor degree in 2010. He was active as a third-year student representative in the student parliament.Then he went on the Faculty of science, at the University of Novi Sad where he successfully defended his master's thesis in 2013. The crown of his study was the work titled Opportunities for development of cultural tourism in Cacak". Later on, he became part of a multinational company where he got promoted to a deputy director of logistic. Nowadays he is a consultant and writer of academic subjects in the field of tourism.

Dr. Jovan Pehcevski obtained his PhD in Computer Science from RMIT University in Melbourne, Australia in 2007. His research interests include big data, business intelligence and predictive analytics, data and information science, information retrieval, XML, web services and service-oriented architectures, and relational and NoSQL database systems. He has published over 30 journal and conference papers and he also serves as a journal and conference reviewer. He is currently working as a Dean and Associate Professor at European University in Skopje, Macedonia.

Dr. Tanjina Nur finished her PhD in Civil and Environmental Engineering in 2014 from University of Technology Sydney (UTS). Now she is working as Post-Doctoral Researcher in the Centre for Technology in Water and Wastewater (CTWW) and published about eight International journal papers with 80 citations. Her research interest is wastewater treatment technology using adsorption process.

Stephen obtained his PhD from the University of North Carolina at Charlotte in 2013 where his graduate research focused on cancer immunology and the tumor microenvironment. He received postdoctoral training in regenerative and translational medicine, specifically gastrointestinal tissue engineering, at the Wake Forest Institute of Regenerative Medicine. Currently, Stephen is an instructor for anatomy and physiology and biology at Forsyth Technical Community College.

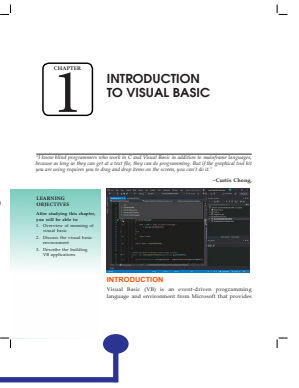
Michelle holds a Masters of Business Administration from the University of Phoenix, with a concentration in Human Resources Management. She is a professional author and has had numerous articles published in the Henry County Times and has written and revised several employee handbooks for various YMCA organizations throughout the United States.

HOW TO USE THE BOOK

This book has been divided into many chapters. Chapter gives the motivation for this book and the use of templates. The text is presented in the simplest language. Each paragraph has been arranged under a suitable heading for easy retention of concept. Keywords are the words that academics use to reveal the internal structure of an author's reasoning. Review questions at the end of each chapter ask students to review or explain the concepts. References provides the reader an additional source through which he/she can obtain more information regarding the topic.

LEARNING OBJECTIVES

See what you are going to cover and what you should already know at the start of each chapter


CHAPTER 1

INTRODUCTION TO VISUAL BASIC

"I know blind programmers who work in C and Visual Basic in addition to mainframe languages, because as long as they can get at a text file, they can do programming. But if the graphical tool kit you are using requires you to drag and drop items on the screen, you can't do it."

–Curtis Chong,

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Overview of meaning of visual basic
2. Discuss the visual basic programming
3. Describe the building VB applications

INTRODUCTION

Visual Basic (VB) is an event-driven programming language and environment from Microsoft that provides

enhancements, including the striking ability of creating web based applications. The extended support for Visual Basic 6.0 was ceased in the month of March in 2008. The basic parts of development environment of Visual Basic 6, however, still run in all the 32-bit Microsoft windows, including Windows 8.1.

After the cessation of mainstream and extended support for Visual Basic 6.0 caused a number of programs to show concern. The community members then created a lobby of users and a petition was signed by them. The basic aim of this petition was to ensure that the product remains alive. However, the petition did not attain its aim effectively.

1.1.2 The Importance of Visual Basic Programming Language

Visual Basic is regarded as the third generation event-drivon programming language. It was released in 1987. Being the first visual development tool from Microsoft, it is considered as one of the most powerful programming languages. As compared to other computer programming languages, such as, C, C++, it is easy to learn and understand, provided that one has determination and dedication to do so.

Visual basic programming language allows programmers to create software interface and codes in an easy to use graphical environment. VB is the combination of different components that are used on forms having specific attributes and actions with the help of those components. On the one hand it allows programmers to develop seldom based applications rapidly; on the other hand, it helps greatly in accessing data bases, using ADO while letting the programmers use ActiveX controls and various objects. While it is intended more to develop applications, it is also useful for games development for particular or limited purposes, unlike C++ that is more suitable for developing games.

As compared to other languages, Visual basic may be slower though, yet it is flexible and it can be rightly said that things that are difficult in other languages are comparatively easier in visual basic programming language. It may also be said that, since it is one of the most popular programming languages, lots of related books and material and other resources are available and can be accessed for developing programming skills at visual basic programming language conveniently.

One of the most important things to be considered with regard to programming in Visual basic is that the structure of VB is designed in a way that allows programmers to create executable code – Exe files. It enables programmers to develop programs that can be used as front end to databases. Besides, it's with the help of visual basic tools, one can change the abstract ideas into programs or into the whole software while it allows revising and modifying the programs fittingly.

ABOUT THIS CHAPTER

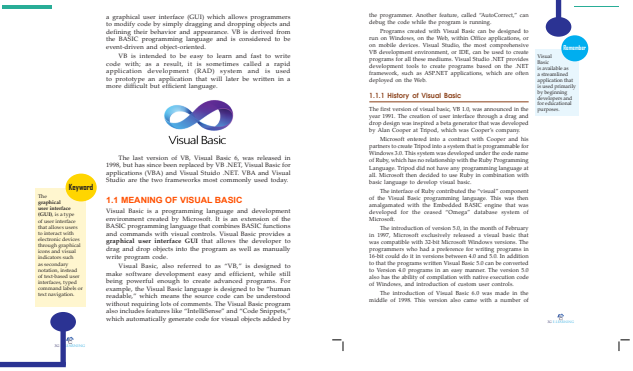
An introduction is a beginning of section which states the purpose and goals of the topics which are discussed in the chapter. It also starts the topics in brief.

REMEMBER

This revitalizes a must read information of the topic.

KEYWORDS

This section contains some important definitions that are discussed in the chapter. A keyword is an index entry that identifies a specific record or document. It also gives the extra information to the reader and an easy way to remember the word definition.


Keyword

The graphical user interface (GUI) is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, as opposed to of text-based user interfaces, typed command labels or text navigation.

a graphical user interface (GUI) which allows programmers to modify code by simply dragging and dropping objects and defining their behavior and appearance. VB is derived from the BASIC programming language and is considered to be event-driven and object-oriented.

VB is intended to be easy to learn and fast to write code with; as a result, it is sometimes called a rapid application development (RAD) system and is used to prototype an application that will later be written in a more difficult but efficient language.

Visual Basic

The last version of VB, Visual Basic 6, was released in 1998, but has since been replaced by VB .NET, Visual Basic for applications (VBA) and Visual Studio .NET. VBA and Visual Studio are the two frameworks most commonly used today.

1.1 MEANING OF VISUAL BASIC

Visual Basic is a programming language and development environment created by Microsoft. It is an extension of the BASIC programming language that combines BASIC functions and commands with visual controls. Visual Basic provides a **graphical user interface GUI** that allows the developer to drag and drop objects into the program as well as manually write program code.

Visual Basic, also referred to as "VB," is designed to make software development easy and efficient, while still being powerful enough to create advanced programs. For example, the Visual Basic language is designed to be "human readable," which means the source code can be understood without requiring lots of comments. The Visual Basic program also includes features like "IntelliSense" and "Code Snippets," which automatically generate code for visual objects added by

the programmer. Another feature, called "AutoCorrect," can debug the code while the program is running.

Programs created with Visual Basic can be designed to run on Windows, on the Web, within Office applications, or on mobile devices. Visual Studio, the most comprehensive VB development environment, or IDE, can be used to create programs for all these mediums. Visual Studio .NET provides development tools to create programs based on the .NET framework, such as ASP.NET applications, which are often deployed on the Web.

Remember

Visual Basic is available as a standalone application that is used primarily by beginning developers and for educational purposes.

1.1.1 History of Visual Basic

The first version of visual basic, VB 1.0, was announced in the year 1991. The creation of user interface through a drag and drop design was inspired a beta generator that was developed by Alan Cooper at Tripod, which was Cooper's company.

Microsoft entered into a contract with Cooper and his partners to create Tripod into a system that is programmable for Windows 3.0. This system was developed under the code name of Ruby, which has no relationship with the Ruby Programming Language. Tripod did not have any programming language at all. Microsoft then decided to use Ruby in combination with basic language to develop visual basic.

The interface of Ruby contributed the "visual" component of the Visual Basic programming language. This was then amalgamated with the Embedded BASIC engine that was developed for the ceased "Omega" database system of Microsoft.

The introduction of version 5.0, in the month of February in 1997, Microsoft exclusively released a visual basic that was compatible with 32-bit Microsoft Windows versions. The programmers who had a preference for writing programs in 16-bit could do it in versions between 4.0 and 5.0. In addition to that the programs written Visual Basic 5.0 can be converted to Version 4.0 programs in an easy manner. The version 5.0 also has the ability of compilation with native execution code of Windows, and introduction of custom user controls.

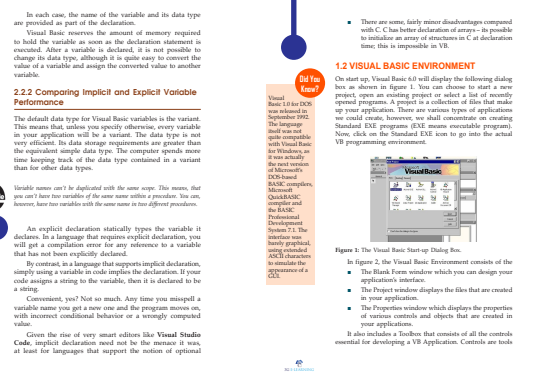
The introduction of Visual Basic 6.0 was made in the middle of 1998. This version also came with a number of

DID YOU KNOW?

This section equip readers the interesting facts and figures of the topic.

EXAMPLE

The book cabinets' examples to illustrate specific ideas in each chapter.



In each case, the name of the variable and its data type are provided as part of the declaration.

Visual Basic reserves the amount of memory required to hold the variable as soon as the declaration statement is executed. After a variable is declared, it is not possible to change its data type, although it is quite easy to convert the value of a variable and assign the converted value to another variable.

2.2.2 Comparing Implicit and Explicit Variable Performance

The default data type for Visual Basic variables is the variant. This means that, unless you specify otherwise, every variable in your application will be a variant. The data type is not very efficient. Its data storage requirements are greater than the equivalent simple data type. The computer spends more time keeping track of the data type contained in a variant than for other data types.

Variable names can't be duplicated with the same scope. This means, that you can't have two variables of the same name within a procedure. You can, however, have two variables with the same name in two different procedures.

An explicit declaration statically types the variable it declares. In a language that requires explicit declaration, you will get a compilation error for any reference to a variable that has not been explicitly declared.

By contrast, in a language that supports implicit declaration, simply using a variable in code implies the declaration. If your code assigns a string to the variable, then it is declared to be a string.

Convenient, yes? Not so much. Any time you misspell a variable name you get a new one and the program moves on, with incorrect conditional behavior or a wrongly computed value.

Given the rise of very smart editors like **Visual Studio Code**, implicit declaration need not be the menace it was, at least for languages that support the notion of optional

- There are some, fairly minor disadvantages compared with C. C has better declaration of arrays – its possible to initialize an array of structures in C at declaration time; this is impossible in VB.

1.2 VISUAL BASIC ENVIRONMENT

Did You Know?

Visual Basic 1.0 for DOS was released in September 1992. The language itself was not quite compatible with Visual Basic for Windows, as it was actually the next version of Microsoft's DOS-based BASIC compilers, Microsoft QuickBASIC compiler and the BASIC Professional Development System 7.1. The interface was barely graphical, using extended ASCII characters to simulate the appearance of a GUI.

On start up, Visual Basic 6.0 will display the following dialog box as shown in figure 1. You can choose to start a new project, open an existing project or select a list of recently opened programs. A project is a collection of files that make up your application. There are various types of applications we could create, however, we shall concentrate on creating Standard EXE programs (EXE means executable program). Now, click on the Standard EXE icon to go into the actual VB programming environment.

Figure 1: The Visual Basic Start-up Dialog Box.

In figure 2, the Visual Basic Environment consists of the

- The Blank Form window which you can design your application's interface.
- The Project window displays the files that are created in your application.
- The Properties window which displays the properties of various controls and objects that are created in your applications.

It also includes a Toolbox that consists of all the controls essential for developing a VB Application. Controls are tools

ROLE MODEL

A biography of someone who has/had acquired remarkable success in their respective field as Role Models are important because they give us the ability to imagine our future selves.

CASE STUDY

This reveals what students need to create and provide an opportunity for the development of key skills such as communication, group working and problem solving.

ROLE MODEL

ALAN COOPER: FATHER OF VISUAL BASIC

Born in San Francisco in 1952 and raised in Marin County, California, Alan Cooper has always taken the path less traveled. A rebellious teenager, he dropped out of high school, but eventually made his way to the College of Marin to pursue his interest in architecture. After an exploratory course in programming, it became clear that his future was in architecture—software architecture. After getting his associate degree and a COBOL programming job, he saw an advertisement for one of the first personal computers and .conceived an idea for a new business venture

In 1976, Cooper founded Structured Systems Group (SSG), a company Fire in the Valley authors Paul Freiberger and Michael Swaine said created "the first serious business software for microcomputers." In four years, Cooper wrote and shipped a dozen application programs. SSG became the archetype for many software startups in the early days of the personal computer revolution.

During the 1980s, after leaving SSG, Cooper invented, wrote, and sold three major software packages to prominent publishers. One of those was the visual programming front-end code named "Ruby," for what became Visual BASIC. Bill Gates purchased it from Cooper in 1988, noting that it would have significant impact across Microsoft's entire product line. Visual BASIC was deemed both a commercial and critical success, earning Cooper the moniker "Father of Visual BASIC." Visual BASIC has influenced integrated development languages ever since.

In 1990 Cooper became fascinated with the challenge of making software products that were easy to use and understand. He and his wife, Susan, founded Cooper Interaction Design (now "Cooper") to assist in what Cooper calls "interaction design." In the design field, Cooper's software development background was unique and, over the next few years, he invented many of the tools and techniques now standard in the user experience industry, including personas and scenarios,

CASE STUDY

FUJITSU FACILITATES SMOOTH MIGRATION TO VB.NET AT AN POST

Fujitsu has an excellent technical team, which works closely with our staff. We have had a good working relationship for many years and Fujitsu has an in-depth knowledge of our mission-critical application gained from several years' development and support work.

Challenge

A Post, one of Ireland's largest companies, is a major commercial organization providing a wide range of postal, communication, retail and financial services. With 9,600 employees throughout its national network of retail, processing and delivery points, the business also provides services to government departments, the National Treasury Management Agency and its own National Lottery Company. A decade ago, A Post implemented a new nationwide time and attendance system to calculate and record staff salary and wages functions. The Staff Remuneration and Administration Management System (STREAMS) is a bespoke, mission-critical application developed by Fujitsu as a reliable, scalable client server system using Microsoft technologies. The STREAMS front-end system gathers information and feeds the data to the company's HR, payroll and financial departments. It primarily creates more efficient processes for A Post to capture data for the weekly payroll run whilst simultaneously minimizing the number of payroll queries by employees. Following deployment, STREAMS improved cost center reporting, significantly lowered the time to record pay details and enhanced the processing of casual staff pay. During this period, Fujitsu provided quality support and maintenance services and application enhancements to increase functionality, ensuring the long-term reliability of STREAMS. For instance, as employee numbers steadily increased to exceed original expectations, Fujitsu boosted system performance by upgrading the infrastructure and optimizing the software. STREAMS originally employed Visual Basic (VB), a third-generation event-driven programming language and integrated development environment (IDE) from Microsoft. IDE provides programmers with comprehensive facilities for software development and comprises a source code editor, a compiler and/or an interpreter, build automation tools and a debugger. However, Microsoft no longer supports VB version 6.0, the edition employed by A Post. Syl Byrne, IT Manager Remuneration Services, A Post, explains: "To ensure that our business-critical application is future-proof, we needed to move to a platform that Microsoft will support for the foreseeable future." A Post therefore decided to migrate STREAMS to the VB.NET platform, an object-orientated programming language. This strategy would protect its investment for the next 10 years by creating a secure, scalable

KNOWLEDGE CHECK

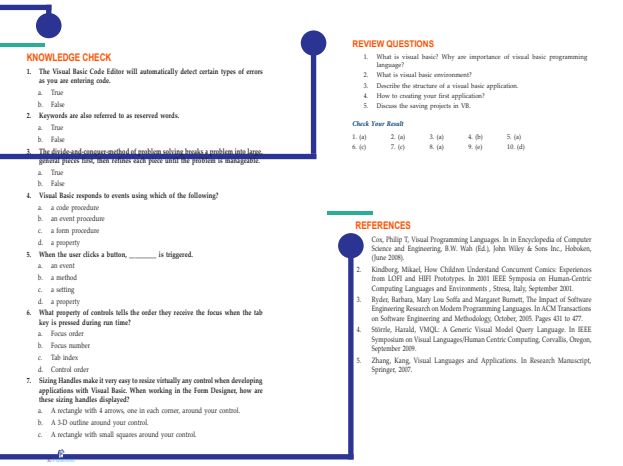
This is given to the students for progress check at the end of each chapter.

REVIEW QUESTIONS

This section is to analyze the knowledge and ability of the reader.

REFERENCES

References refer those books which discuss the topics given in the chapters in almost same manner.



KNOWLEDGE CHECK

1. **The Visual Basic Code Editor will automatically detect certain types of errors as you are entering code.**
   a. True
   b. False
2. **Keywords are also referred to as reserved words.**
   a. True
   b. False
3. **The divide-and-conquer method of problem solving breaks a problem into large general pieces first, then refines each piece until the problem is manageable.**
   a. True
   b. False
4. **Visual Basic responds to events using which of the following?**
   a. a code procedure
   b. an event procedure
   c. a form procedure
   d. a property
5. **When the user clicks a button, _____ is triggered.**
   a. an event
   b. a method
   c. a setting
   d. a property
6. **What property of controls tells the order they receive the focus when the tab key is pressed during run time?**
   a. Focus order
   b. Focus number
   c. Tab index
   d. Control order
7. **Sizing Handles make it very easy to resize virtually any control when developing applications with Visual Basic. When working in the Form Designer, how are these sizing handles displayed?**
   a. A rectangle with 4 arrows, one in each corner, around your control.
   b. A 3-D outline around your control.
   c. A rectangle with small squares around your control.

REVIEW QUESTIONS

1. What is visual basic? Why are importance of visual basic programming language?
2. What is visual basic environment?
3. Describe the structure of a visual basic application.
4. How to creating your first application?
5. Discuss the saving projects in VB.

Check Your Result

1. (a)	2. (a)	3. (a)	4. (b)	5. (a)
6. (c)	7. (c)	8. (a)	9. (e)	10. (d)

REFERENCES

1. Cox, Philip T, Visual Programming Languages. In in Encyclopedia of Computer Science and Engineering, B.W. Wah (Ed.), John Wiley & Sons Inc., Hoboken, (June 2008).
2. Kindborg, Mikael, How Children Understand Concurrent Comics: Experiences from LOFI and HIFI Prototypes. In 2001 IEEE Symposia on Human-Centric Computing Languages and Environments , Stresa, Italy, September 2001.
3. Ryder, Barbara, Mary Lou Soffa and Margaret Burnett, The Impact of Software Engineering Research on Modern Programming Languages. In ACM Transactions on Software Engineering and Methodology, October, 2005. Pages 431 to 477.
4. Störrle, Harald, VMQL: A Generic Visual Model Query Language. In IEEE Symposium on Visual Languages/Human Centric Computing, Corvallis, Oregon, September 2009.
5. Zhang, Kang, Visual Languages and Applications. In Research Manuscript, Springer, 2007.

TABLE OF CONTENTS

Preface xv

Chapter 1 Introduction to Python 1

Introduction 1

1.1 Overview of Python 2

1.1.1 History of Python 3

1.1.2 Python Features 3

1.2 Python Environment Setup 4

1.2.1 Getting Python 5

1.2.2 Installing Python 5

1.2.3 Setting up PATH 6

1.2.4 Python Environment Variables 7

1.2.5 Running Python 8

1.3 Basic Syntax of Python 10

1.3.1 First Python Program 10

1.3.2 Python Identifiers 11

1.3.3 Reserved Words 11

1.3.4 Lines and Indentation 12

1.3.5 Multi-Line Statements 13

1.3.6 Quotation in Python 14

1.3.7 Comments in Python 14

1.3.8 Using Blank Lines 15

1.3.9 Waiting for the User 15

1.3.10 Multiple Statements on a Single Line 15

1.3.11 Multiple Statement Groups as Suites 15

1.3.12 Command Line Arguments 16

1.4 Python Variables 16

1.4.1 Assigning Values to Variables 16

1.4.2 Multiple Assignment 17

1.4.3 Standard Data Types 17
1.4.4 Data Type Conversion 22
1.5 Python Basic Operators 24
1.5.1 Types of Operator 24
1.5.2 Python Operators Precedence 28
Summary 33
Knowledge Check 34
Review Questions 35
References 36

Chapter 2 Python Functions, Modules and Packages 37

Introduction 37
2.1 Function in Python 38
2.1.1 Syntax of Function 38
2.1.2 Docstring 39
2.1.3 The Return Statement 40
2.1.4 How Function works in Python? 41
2.1.5 Python Function Arguments 41
2.1.6 The Anonymous Functions 45
2.1.7 The Return Statement 46
2.2 Python Modules 46
2.2.1 More on Modules 48
2.2.2 Standard Modules 51
2.3 Python Packages 54
2.3.1 Importing* From a Package 56
2.3.2 Intra-package References 57
2.3.3 Packages in Multiple Directories 58
Summary 59
Knowledge Check 60
Review Questions 61
References 62

Chapter 3 Dictionaries, Sets, and Files 63

Introduction 63
3.1 Python Dictionaries 64
3.1.1 Accessing Dictionary Elements 65
3.1.2 Modifying Dictionaries 68

3.1.3 The dict() Constructor 73
3.1.4 Dictionary Methods 73
3.1.5 Aliasing and Copying 74
3.2 Python Sets 75
3.2.1 Defining a Set 75
3.2.2 Set Size and Membership 79
3.2.3 Methods for Sets 79
3.2.4 Creating a Set 80
3.2.5 Accessing Values in a Set 81
3.2.6 Adding Items to a Set 81
3.2.7 Removing Item from a Set 81
3.2.8 Union of Sets 82
3.2.9 Intersection of Sets 82
3.2.10 Difference of Sets 83
3.2.11 Compare Sets 83
3.3 Files 83
3.3.1 The open function 84
3.3.2 Opening a File that Doesn't Exist 86
3.3.3 Reading Data from Files 86
Summary 92
Knowledge Check 93
Review Questions 94
References 95

Chapter 4 Exceptions, Unit Testing and Comprehensions 97

Introduction 97
4.1 Exceptions 98
4.1.1 Handling Exceptions 99
4.1.2 Raising Exceptions 103
4.1.3 User-defined Exceptions 104
4.1.4 Defining Clean-up Actions 106
4.1.5 Predefined Clean-up Actions 107
4.2 Unit testing 108
4.2.1 Basic example 109
4.2.2 Command-Line Interface 111
4.2.3 Test Discovery 112
4.2.4 Organizing test code 113
4.2.5 Re-using old test code 117

4.2.6 Skipping tests and expected failures 118
4.3 Comprehensions 120
4.3.1 List Comprehensions 120
4.3.2 Dict Comprehensions 121
4.3.3 Set Comprehensions 121
4.3.4 Generator Comprehensions 121
Summary 129
Knowledge Check 130
Review Questions 131
References 132

Chapter 5 Object Oriented Programming 133

Introduction 133
5.1 Introduction of OOPS In Python 135
5.1.1 Classes in Python 135
5.1.2 Python Objects (Instances) 136
5.1.3 Instantiating Objects 138
5.1.4 Instance Methods 140
5.1.5 Python Object Inheritance 141
5.2 Methods of OOPS 147
5.2.1 Inheritance 148
5.2.2 Encapsulation 151
5.2.3 Polymorphism 155
5.2.4 Abstraction 159
Summary 168
Knowledge Check 169
Review Questions 170
References 171

Chapter 6 Python Regular Expression 173

Introduction 173
6.1 Regex Search and Match 174
6.1.1 The Match Function 176
6.1.2 The Search Function 177
6.1.3 Matching Versus Searching 179
6.1.4 Search and Replace 180
6.2 Regular Expression Modifiers: Option Flags 180

6.2.1 Regular Expression Patterns 181
6.2.2 Regular Expression Examples 184
Summary 197
Knowledge Check 198
Review Questions 199
References 200

Chapter 7 Python Multithreading 201

Introduction 201
7.1 Python Threading – Python Multithreading 202
7.1.1 Getting Started with Python Multithreading 203
7.1.2 Python Multithreading Modules for Thread Implementation 204
7.1.3 Difference between Multiprocessing and Multithreading 204
7.2 Functions in Python Multithreading 206
7.2.1 Thread-Local Data 208
7.2.2 Thread Objects 209
7.2.3 Lock Objects 212
7.2.4 RLock Objects 213
7.2.5 Condition Objects 214
7.2.6 Semaphore Objects 216
7.2.7 Event Objects 218
7.2.8 Timer Objects 219
7.2.9 Barrier Objects 220
7.2.10 Using locks, Conditions, and Semaphores in the with-statement 221
Summary 224
Knowledge Check 225
Review Questions 226
References 227

Chapter 8 Operations in Python 229

Introduction 229
8.1 Python - Decision Making 230
8.1.1 Python if Statement 231
8.1.2 Python if-else Statement 233
8.1.3 Python if-elif ladder 235
8.1.4 Python Nested if statement 237
8.2 Python - Loops 239
8.2.1 The range() function 241

8.2.2 for loop with else 243
8.2.3 Loop Control Statements 244
8.3 Python - Numbers 244
8.3.1 Number Type Conversion 246
8.3.2 Mathematical Functions 246
8.3.3 Random Number Functions 248
8.3.4 Trigonometric Functions 248
8.3.5 Mathematical Constants 249
8.4 Python - Strings 249
8.4.1 Accessing Values in Strings 250
8.4.2 Updating Strings 250
8.4.3 Escape Characters 250
8.4.4 String Special Operators 251
8.4.5 String Formatting Operator 252
8.4.6 Triple Quotes 253
8.4.7 Unicode String 255
8.4.8 Built-in String Methods 255
8.5 Python - Lists 259
8.5.1 Accessing Values in Lists 259
8.5.2 Updating Lists 260
8.5.3 Delete List Elements 261
8.5.4 Basic List Operations 261
8.5.5 Indexing, Slicing, and Matrixes 262
8.5.6 Built-in List Functions & Methods 262
8.6 Python - Tuples 263
8.6.1 Accessing Values in Tuples 264
8.6.2 Updating Tuples 264
8.6.3 Delete Tuple Elements 265
8.6.4 Basic Tuples Operations 266
8.6.5 Indexing, Slicing, and Matrixes 266
8.6.6 No Enclosing Delimiters 266
8.6.7 Built-in Tuple Functions 267
8.7 Python - Date & Time 267
8.7.1 Getting Current Time 269
8.7.2 Getting formatted time 270
8.7.3 Getting calendar for a month 270
8.7.4 The time Module 271
8.7.5 The calendar Module 272
Summary 275

Knowledge Check 276
Review Questions 277
References 278

Chapter 9 Python Database Programming 279

Introduction 279
9.1 DB-API (SQL-API) for Python 281
9.1.1 Connection Objects 281
9.1.2 Cursor objects 282
9.1.3 Error and Exception Handling in DB-API 283
9.1.4 Python and MySQL 285
9.1.5 More SQL operations 286
9.1.6 Python MySQL – Create Database 289
9.2 MySQL with Python 290
9.2.1 Comparing MySQL to Other SQL Databases 291
9.2.2 Installing MySQL Server and MySQL Connector/Python 293
9.2.3 Establishing a Connection with MySQL Server 295
9.3 Creating, Altering, and Dropping a Table 300
9.3.1 Defining the Database Schema 301
9.3.2 Creating Tables Using the CREATE TABLE Statement 302
9.3.3 Showing a Table Schema Using the DESCRIBE Statement 305
9.3.4 Modifying a Table Schema Using the ALTER Statement 306
9.3.5 Deleting Tables Using the DROP Statement 308
9.4 Inserting Records in Tables 308
9.4.1 Using .execute() 308
9.4.2 Using .executemany() 310
9.4.3 Reading Records from the Database 313
9.4.4 Handling Multiple Tables Using the JOIN Statement 318
9.5 Updating and Deleting Records from the Database 320
9.5.1 Update Command 320
9.5.2 Delete Command 328
9.5.3 Other Ways to Connect Python and MySQL 329
Summary 332
Knowledge Check 333
Review Questions 334
References 335

Index 337

PREFACE

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Its high-level built in data structures, combined with dynamic typing and dynamic binding; make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python has become one of the most popular programming languages in the world in recent years. It's used in everything from machine learning to building websites and software testing. It can be used by developers and non-developers alike.

Organization of the Book

This edition is organized into nine chapters. This is a comprehensive guide on how to get started in Python, why you should learn it and how you can learn it. This hands-on guide takes you through the language a step at a time, beginning with basic programming concepts including with functions, recursion, data structures, and object-oriented design.

Chapter 1 presents an introduction to Python. You will learn the Python environment setup, syntax of Python and Python variables. Basic operators of Python are also discussed.

Chapter 2 aims to focus on Python functions, modules and packages. In Python, function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular parts. It also describes the python modules and python packages.

Chapter 3 begins with python dictionaries. It also explains Python sets and files used in Python. They can be used to read and write text memos, audio clips, Excel documents, saved email messages, and whatever else you happen to have stored on your machine.

Chapter 4 gives an overview of how to use the exceptions? Further, it explains the unit testing used to validate that each unit of the software performs as designed. In last, the chapter focuses on understanding the comprehensions that allow sequences to be built from other sequences.

Chapter 5 is aimed to discuss the use of OOPS in python, including various types of methods of OOPS. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them.

Chapter 6 focuses on Python regular expression that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

Chapter 7 discusses about Python Multithreading used to implement multithreading in python programs and also used to run multiple threads (tasks, function calls) at the same time.

Chapter 8 focuses on operations in Python. Python is a powerful general-purpose programming language. It is used in web development, data science, creating software prototypes, and so on. Fortunately for beginners, Python has simple easy-to-use syntax. This makes Python an excellent language to learn to program for beginners.

Chapter 9 sheds light on Python database programming. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions.

CHAPTER

1

INTRODUCTION TO PYTHON

"Now, it's my belief that Python is a lot easier than to teach to students programming and teach them C or C++ or Java at the same time because all the details of the languages are so much harder. Other scripting languages really don't work very well there either."

–Guido van Rossum

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Overview the Python
2. Learn about Python environment setup
3. Describe the basic syntax of Python
4. Understand Python variables
5. Discuss about basic operators of Python

INTRODUCTION

Python is a general-purpose, versatile, and powerful programming language. It's a great first language because it's concise and easy to read. Whatever you want to do,

Python can do it. From web development to machine learning to data science, Python is the language for you.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

Python is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. It will list down some of the key advantages of learning Python:

- Python is Interpreted – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Python is Interactive – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- Python is Object-Oriented – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- Python is a Beginner's Language – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

1.1 OVERVIEW OF PYTHON

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- ***Python is Interpreted*** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- ***Python is Interactive*** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- ***Python is Object-Oriented*** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- ***Python is a Beginner's Language*** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.
-

1.1.1 History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute .for Mathematics and Computer Science in the Netherlands

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Did You Know?

Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales. In July 2018, Van Rossum stepped down as the leader in the language community after 30 years.

1.1.2 Python Features

Python's features include –

- **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
- **A broad standard library** – Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** – Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** – Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** – You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** – Python provides interfaces to all major commercial databases.

- **GUI Programming** – Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** – Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below –

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and **Java**.

Keyword

Java is a programming language that produces software for multiple platforms.

1.2 PYTHON ENVIRONMENT SETUP

Python is available on a wide variety of platforms including Linux and Mac OS X. Let's understand how to set up our Python environment.

Open a terminal window and type "python" to find out if it is already installed and which version is installed.

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga

- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python has also been ported to the Java and .NET virtual machines

1.2.1 Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python https://www.python.org/

You can download Python documentation from https://www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats.

1.2.2 Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on various platforms –

Unix and Linux Installation

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the *Modules/Setup* file if you want to customize some options.
- run ./configure script
- make
- make install

This installs Python at standard location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX* where XX is the version of Python.

Windows Installation

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you need to install.
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Keyword

Unix is a multi-user operating system designed for flexibility and adaptability.

Macintosh Installation

Recent Macs come with Python installed, but it may be several years out of date. See http://www.python.org/download/mac/ for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available.

Jack Jansen maintains it and you can have full access to the entire documentation at his website – http://www.cwi.nl/~jack/macpython.html. You can find complete installation details for Mac OS installation.

1.2.3 Setting up PATH

Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

The **path** variable is named as PATH in **Unix** or Path in Windows (Unix is case sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke

the Python interpreter from any particular directory, you must .add the Python directory to your path

Setting path at Unix/Linux

To add the Python directory to the path for a particular session in Unix –

- **In the csh shell** – type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
- **In the bash shell (Linux)** – type export ATH="$PATH:/usr/local/bin/python" and press Enter.
- **In the sh or ksh shell** – type PATH="$PATH:/usr/local/bin/python" and press Enter.
- **Note** – /usr/local/bin/python is the path of the Python directory

Remember

C:\
Python is
the path of the
Python directory

Setting path at Windows

To add the Python directory to the path for a particular session – in Windows

At the command prompt – type path %path%;C:\Python .and press Enter

1.2.4 Python Environment Variables

Here are important environment variables, which can be recognized by Python –

Sr.No.	Variable & Description
1	**PYTHONPATH** It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer.
2	**PYTHONSTARTUP** It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.

3	**PYTHONCASEOK** It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.
4	**PYTHONHOME** It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

1.2.5 Running Python

There are three different ways to start Python −

Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

Start coding right away in the interactive interpreter.

$python # Unix/Linux

or

python% # Unix/Linux

or

C:> python # Windows/DOS

Here is the list of all the available command line options −

Sr.No.	Option & Description
1	**-d** It provides debug output.
2	**-O** It generates optimized bytecode (resulting in .pyo files).
3	**-S** Do not run import site to look for Python paths on startup.
4	**-v** verbose output (detailed trace on import statements).

5	**-X** disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6.
6	**-c cmd** run Python script sent in as cmd string
7	**file** run Python script from given file

Script from the Command-line

A Python script can be executed at command line by invoking the interpreter on your application, as in the following −

$python script.py # Unix/Linux

or

python% script.py # Unix/Linux

or

C: >python script.py # Windows/DOS

Note − Be sure the file permission mode allows execution.

Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- **Unix** − IDLE is the very first Unix IDE for Python.
- **Windows** − PythonWin is the first Windows interface for Python and is an IDE with a GUI.
- **Macintosh** − The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.
- If you are not able to set up the environment properly, then you can take help from your system admin. Make sure the Python environment is properly set up and working perfectly fine.

Note − All the examples given in subsequent chapters are executed with Python 2.4.3 version available on CentOS flavor of Linux.

Keyword

Perl is a family of two high-level, general-purpose, interpreted, dynamic programming languages, Perl 5 and Perl 6.

We already have set up Python Programming environment online, so that you can execute all the available examples online at the same time when you are learning theory. Feel free to modify any example and execute it online.

1.3 BASIC SYNTAX OF PYTHON

The Python language has many similarities to **Perl**, C, and Java. However, there are some definite differences between the languages.

1.3.1 First Python Program

Let us execute programs in different modes of programming.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt –

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Type the following text at the Python prompt and press the Enter –

```
>>> print "Hello, Python!"
```

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result –

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py file –

```
print "Hello, Python!"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows –

$ python test.py

This produces the following result –

Hello, Python!

Let us try another way to execute a Python script. Here is the modified test.py file –

#!/usr/bin/python

print "Hello, Python!"

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows –

$ chmod +x test.py # This is to make file executable

$./test.py

This produces the following result –

Hello, Python!

1.3.2 Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers –

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

1.3.3 Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

1.3.4 Lines and Indentation

Remember

Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example –

```
if True:
   print "True"
else:
   print "False"
```

However, the following block generates an error –

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks –

```
#!/usr/bin/python
import sys
```

```
try:
    # open file stream
    file = open(file_name, "w")
except IOError:
    print "There was an error writing to", file_name
    sys.exit()
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
    file_text = raw_input("Enter text: ")
    if file_text == file_finish:
        # close the file
        file.close
        break
    file.write(file_text)
    file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
    print "Next time please enter something"
    sys.exit()
try:
    file = open(file_name, "r")
except IOError:
    print "There was an error reading file"
    sys.exit()
file_text = file.read()
file.close()
print file_text
```

1.3.5 Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For

example –

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example –

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

1.3.6 Quotation in Python

Python accepts single (‘), double (“) and triple (‘’’ or “””) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal –

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

1.3.7 Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python
# First comment
print "Hello, Python!" # second comment
```

This produces the following result –

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression –

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows –

```
# This is a comment.
# This is a comment, too.
```

```
# This is a comment, too.
# I said that already.
```

1.3.8 Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

1.3.9 Waiting for the User

The following line of the program displays the prompt, the statement saying "Press the enter key to exit", and waits for the user to take action –

```
#!/usr/bin/python
raw_input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Keyword

Code block is a lexical structure of source code which is grouped together.

1.3.10 Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new **code block**. Here is a sample snip using the semicolon –

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

1.3.11 Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example –

```
if expression :
```

```
    suite
elif expression :
    suite
else :
    suite
```

1.3.12 Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h –

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit

[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts.

1.4 PYTHON VARIABLES

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

1.4.1 Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. *For example* –

```
#!/usr/bin/python

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result –

```
100
1000.0
John
```

1.4.2 Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example –

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example –

```
a,b,c = 1,2,"john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

1.4.3 Standard Data Types

The data stored in memory can be of many types. *For example*, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example –

var1 = 1

var2 = 10

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is –

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement. For example –

del var

del var_a, var_b

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J

-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

- A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example −

```
#!/usr/bin/python

str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result −

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

Keyword

Data type is a classification of data which tells the compiler or interpreter how the programmer intends to use the data.

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different **data type**.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example –

usr/bin/python/!#

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

This produce the following result –

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example −

usr/bin/python/!#

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple               # Prints complete list
print tuple[0]            # Prints first element of the list
print tuple[1:3]          # Prints elements starting from 2nd till 3rd
print tuple[2:]           # Prints elements starting from 3rd element
print tinytuple * 2       # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

This produce the following result −

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000     # Invalid syntax with tuple
list[2] = 1000      # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost

any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

1.4.4 Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.

15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

1.5 PYTHON BASIC OPERATORS

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

1.5.1 Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Python Arithmetic Operators

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

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0

** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

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

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

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

Python Assignment Operators

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

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

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

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Remember

Relational operators are used for comparing the values. It either returns True or False according to the condition. These operators are also known as Comparison Operators.

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Used to reverse the logical state of its operand.

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

Python Membership Operators

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

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

Python Identity Operators

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

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

1.5.2 Python Operators Precedence

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

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift

&	Bitwise 'AND'td>
^ |	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operator precedence affects how an expression is evaluated.

*x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies 3*2 and then adds into 7.*

Here, operators with the highest precedence appear at the .top of the table, those with the lowest appear at the bottom

Example

```
#!/usr/bin/python

a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d        #( 30 * 15 ) / 5
print "Value of (a + b) * c / d is ",  e

e = ((a + b) * c) / d      # (30 * 15 ) / 5
print "Value of ((a + b) * c) / d is ",  e

e = (a + b) * (c / d);     # (30) * (15/5)
print "Value of (a + b) * (c / d) is ",  e
```

```
e = a + (b * c) / d;        #  20 + (150/5)
print "Value of a + (b * c) / d is ",  e
```

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

```
Value of (a + b) * c / d is 90
Value of ((a + b) * c) / d is 90
Value of (a + b) * (c / d) is 90
Value of a + (b * c) / d is 50
```

ROLE MODEL

GUIDO VAN ROSSUM

Guido van Rossum; born 31 January 1956) is a Dutch programmer best known as the author of the Python programming language, for which he was the "Benevolent Dictator For Life" (BDFL) .until he stepped down from the position in July 2018

Education and Life

Van Rossum was born and raised in the Netherlands, where he received a master's degree in mathematics and computer science from the University of Amsterdam in 1982. He has a brother, Just van Rossum, who is a type designer and programmer who designed the typeface used in the "Python Powered" logo.

Guido lives in Belmont, California, with his wife, Kim Knapp, and their son. According to his home page and Dutch naming conventions, the "van" in his name is capitalized when he is referred to by surname alone, but not when using his first and last name together.

Work

While working at the Centrum Wiskunde & Informatica (CWI), Van Rossum wrote and contributed a glob()routine to BSD Unix in 1986 and helped develop the ABC programming language. He once stated, "I try to mention ABC's influence because I'm indebted to everything I learned during that project and to the people who worked on it." He also created Grail, an early web browser written in Python, and engaged in discussions about the HTML standard.

He has worked for various research institutes, including the Centrum Wiskunde & Informatica (CWI) in the Netherlands, the U.S. National Institute of Standards and Technology (NIST), and the Corporation for National Research Initiatives (CNRI). From 2000 until 2003 he worked for Zope corporation. In 2003 van Rossum left Zope for Elemental Security. While there he worked on a custom programming language for the organization. From 2005 to December 2012, he worked at

Google, where he spent half of his time developing the Python language. In January 2013, he started working for Dropbox.

Python

In December 1989, Van Rossum had been looking for a "'hobby' programming project that would keep [him] occupied during the week around Christmas" as his office was closed when he decided to write an interpreter for a "new scripting language [he] had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers". He attributes choosing the name "Python" to "being in a slightly irreverent mood (and [being] a big fan of *Monty Python's Flying Circus*)". He has explained that Python's predecessor, ABC, was inspired by SETL, noting that ABC co-developer Lambert Meertens had "spent a year with the SETL group at NYU before coming up with the final ABC design". In July 2018, Van Rossum announced that he would be stepping down from the position of BDFL of the Python programming language.

Computer Programming for Everybody

In 1999, Van Rossum submitted a funding proposal to DARPA called "Computer Programming for Everybody," in which he further defined his goals for Python:

- An easy and intuitive language just as powerful as major competitors
- Open source, so anyone can contribute to its development
- Code that is as understandable as plain English
- Suitability for everyday tasks, allowing for short development times

Python has grown to become a popular programming language. As of October 2017, it was the second most popular language on GitHub, a social coding website, behind Javascript and ahead of Java. According to a programming language popularity survey it is consistently amongst the top 10 most mentioned languages in job postings. Furthermore, Python is consistently in the top 10 most popular languages according to the TIOBE Programming Community Index.

Mondrian

At Google, Van Rossum developed Mondrian, a web-based code review system written in Python and used within the company. He named the software after the Dutch painter Piet Mondriaan. He named another related software projectafter Gerrit Rietveld, a Dutch designer.

Dropbox

In 2013, Van Rossum started working at the cloud file storage company Dropbox.

SUMMARY

- Python is a general-purpose, versatile, and powerful programming language. It's a great first language because it's concise and easy to read. Whatever you want to do, Python can do it.
- Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable.
- Python is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain
- Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.
- A Python identifier is a name used to identify a variable, function, class, module or other object.
- Python provides no braces to indicate blocks of code for class and function definitions or flow control.
- Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue.
- A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

KNOWLEDGE CHECK

1. **Which of the following is correct about Python?**
   a. Python is a high-level, interpreted, interactive and object-oriented scripting language.
   b. Python is designed to be highly readable.
   c. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.
   d. All of the above.
2. **Which of the following is correct about Python?**
   a. It supports functional and structured programming methods as well as OOP.
   b. It can be used as a scripting language or can be compiled to byte-code for building large applications.
   c. It provides very high-level dynamic data types and supports dynamic type checking.
   d. All of the above.
3. **Which of the following environment variable for Python tells the Python interpreter where to locate the module files imported into a program?**
   a. Pythonpath
   b. Pythonstartup
   c. Pythoncaseok
   d. Pythonhome
4. **Which of the following data types is not supported in python?**
   a. List
   b. Slice
   c. String
   d. Numbers
5. **Which of the following is correct about tuples in python?**
   a. A tuple is another sequence data type that is similar to the list.
   b. A tuple consists of a number of values separated by commas.
   c. Unlike lists, however, tuples are enclosed within parentheses.
   d. All of the above.
6. **What is the maximum possible length of an identifier?**
   a. 16
   b. 32

c. 64

d. None of these above

7. Who developed the Python language?

a. Zim Den

b. Guido van Rossum

c. Niene Stom

d. Wick van Rossum

8. In which year was the Python language developed?

a. 1995

b. 1972

c. 1981

d. 1989

REVIEW QUESTIONS

1. What is Python? Name some of the features of python.
2. What are the purpose of pythonpath, pythonstartup, Pythoncaseok, and pythonhome environment variable?
3. What are the supported data types in python?
4. What are python's dictionaries?
5. What is the difference between tuples and lists in python?

Check Your Result

1. (d) 2. (d) 3. (a) 4. (b) 5. (d)
6. (d) 7. (b) 8. (d)

REFERENCES

1. Deily, Ned (28 March 2018). "Python 3.7.0 is now available". Python Insider. The Python Core Developers. Retrieved 29 March 2018.
2. Downey, Allen B. (May 2012). Think Python: How to Think Like a Computer Scientist (Version 1.6.6 ed.).
3. Guttag, John V. (2016-08-12). Introduction to Computation and Programming Using Python: With Application to Understanding Data. MIT Press.
4. Hamilton, Naomi (5 August 2008). "The A-Z of Programming Languages: Python". Computerworld. Archived from the original on 29 December 2008. Retrieved 31 March 2010.
5. Peterson, Benjamin (1 May 2018). "Python 2.7.15 released". Python Insider. The Python Core Developers. Retrieved 1 May 2018.
6. Summerfield, Mark (2009). Programming in Python 3 (2nd ed.). Addison-Wesley Professional.

CHAPTER

2

PYTHON FUNCTIONS, MODULES AND PACKAGES

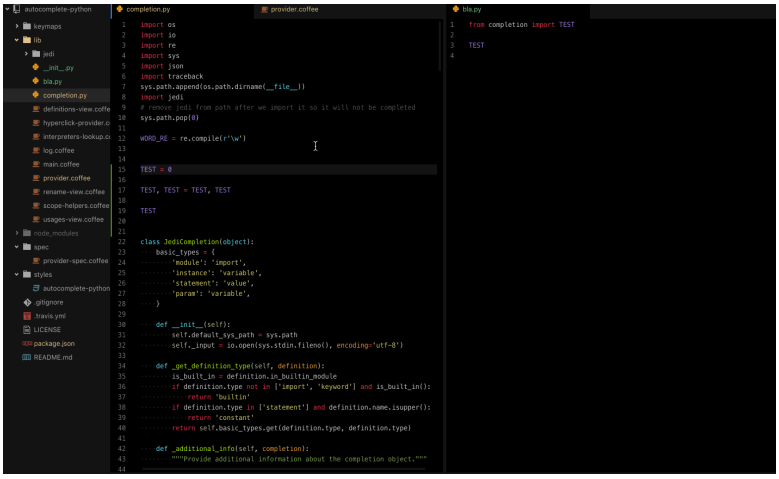
"Everyone knows that any scripting language shootout that doesn't show Python as the best language is faulty by design."

–Max M

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Discuss the function in python
2. Describe the python modules and python packages



INTRODUCTION

Python Functions is a block of related statements designed to perform a computational, logical, or evaluative task. The idea is to put some commonly or repeatedly done

tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the Python program.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.

Python provide us various inbuilt functions like range() or print(). Although, the user can create its functions, which can be called user-defined functions.

2.1 FUNCTION IN PYTHON

In Python, function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes code reusable.

2.1.1 Syntax of Function

```
def function_name(parameters):
 """docstring"""
 statement(s)
```

Above shown is a function definition which consists of following components.

- Keyword def marks the start of function header.
- A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python **statements** that make up the function body. Statements must have same indentation level (usually 4 spaces).

- An optional return statement to return a value from the function.

Example of a function

:(def greet(name
This function greets to“””
the person passed in as
“””parameter
(“!print(“Hello, “ + name + “. Good morning

Keyword

A **statement** is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far.

?How to call a function in python

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
    Hello, Paul. Good morning!
```

2.1.2 Docstring

The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as __doc__ attribute of the function.

For example:

Try running the following into the Python shell to see the output.

```
>>> print(greet.__doc__)
    This function greets to
        the person passed into the
        name parameter
```

2.1.3 The Return Statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of Return

return [expression_list]

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

Remember

A function that does not have a return statement returns by default something called .""None

For example:

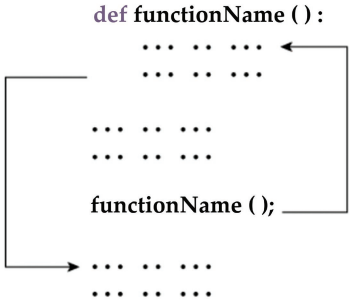
```
>>> print(greet("May"))
    Hello, May. Good morning!
    None
```

Here, None is the returned value.

Example of Return

```
def absolute_value(num):
        """This function returns the absolute
        value of the entered number"""

        if num >= 0:
                return num
        else:
                return -num

# Output: 2
print(absolute_value(2))

# Output: 4
print(absolute_value(-4))
```

2.1.4 How Function works in Python?



Did You Know?

Google began a project named Unladen Swallow in 2009 with the aim of speeding up the Python interpreter fivefold by using the LLVM, and of improving its multithreading ability to scale to thousands of cores.

2.1.5 Python Function Arguments

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

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows –

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;
```

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

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

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

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

Keyword

Parameter are commonly used, and are referred to as parameters and arguments—or more formally as a formal parameter and an actual parameter.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways –

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme( str = "My string")
```

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

```
My string
```

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

```
#!/usr/bin/python
# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;
# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

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

```
Name:  miki
Age  50
```

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
#!/usr/bin/python
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;
# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

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

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
#!/usr/bin/python
# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

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

```
Output is:
10
Output is:
70
60
50
```

2.1.6 The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

An anonymous function cannot be a direct call to print because lambda requires an expression

Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Anonymous functions are often arguments being passed to higher-order functions, or used for constructing the result of a higher-order function that needs to return a function.

Syntax

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

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

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

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

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

Value of total : 30
Value of total : 40

2.1.7 The Return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

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

```
#!/usr/bin/python

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

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

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

Inside the function : 30
Outside the function : 30

2.2 PYTHON MODULES

Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The **file name** is the module name with the suffix .py appended. Within a module, the module's name

(as a string) is available as the value of the global variable __name__. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module


def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b


def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

Keyword

A **file name** is a name used to uniquely identify a computer file stored in a file system.

```
>>> fib = fibo.fib
>>> fib(500)
377 233 144 89 55 34 21 13 8 5 3 2 1 1
```

2.2.1 More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement. (They are also run if the file (.is executed as a script

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, modname.itemname.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_).

Note that in general the practice of importing * from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by as, then the name following as is bound directly to the imported module.

```
>>> import fibo as fib
```

```
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that import fibo will do, with the only difference of it being available as fib.

It can also be used when utilising from with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Executing Modules as Scripts

When you run a Python module with

<python fibo.py <arguments

the code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__". That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

The Module Search Path

When a module named spam is imported, the interpreter first searches for a built-in module with that name. If not found, it

Remember

Domestic robot is a type of service robot, an autonomous robot that is primarily used for household chores, but may also be used for education, entertainment or therapy.

then searches for a file named spam.py in a list of directories given by the variable sys.path. sys.path is initialized from these locations:

the directory containing the input script (or the current directory).

PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).

the installation-dependent default.

After initialization, Python programs can modify sys.path. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended.

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called spam.pyc exists in the directory where spam.py is found, this is assumed to contain an already-"byte-compiled" version of the module spam. The modification time of the version of spam.py used to create spam.pyc is recorded in spam.pyc, and the .pyc file is ignored if these don't match.

Did You Know?

Python convention encourages the use of named functions defined in the same scope as one might typically use an anonymous functions in other languages.

Normally, you don't need to do anything to create the spam.pyc file. Whenever spam.py is successfully compiled, an attempt is made to write the compiled version to spam.pyc. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting spam.pyc file will be recognized as invalid and thus ignored later. The contents of the spam.pyc file are platform independent, so a Python module directory can be shared by machines of different architectures.

Some tips for experts:

- When the Python interpreter is invoked with the **-O** flag, optimized code is generated and stored in .pyo files. The optimizer currently doesn't help much; it only removes **assert** statements. When **-O** is used, *all* bytecode is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode.
- Passing two **-O** flags to the Python interpreter (**-OO**) will cause the bytecode compiler to perform optimizations that could in some rare cases result in

malfunctioning programs. Currently only __doc__ strings are removed from the bytecode, resulting in more compact .pyo files. Since some programs may rely on having these available, you should only use this option if you know what you're doing.

- A program doesn't run any faster when it is read from a .pyc or .pyo file than when it is read from a .py file; the only thing that's faster about .pyc or .pyo files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a .pyc or .pyo file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a .pyc or .pyo file directly on the command line.
- It is possible to have a file called spam.pyc (or spam.pyo when **-O** is used) without a file spam.py for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
- The module **compileall** can create .pyc files (or .pyo files when **-O** is used) for all modules in a directory.

Keyword

The **compileall** module finds Python source files and compiles them to the byte-code representation, saving the results in .pyc or .pyo files.

2.2.2 Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the winreg module is only provided on Windows systems. One particular module deserves some attention: sys, which is built into every Python interpreter. The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:

```
>>> import sys
```

```
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable sys.path is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

The dir() Function¶

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
 '_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
 'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
```

'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
'version', 'version_info', 'warnoptions']

Without arguments, dir() lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', '__package__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

dir() does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module __builtin__:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
```

'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']

2.3 PYTHON PACKAGES

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name A.B designates a submodule named B in a package named A. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
```

```
        auread.py
        auwrite.py
        ...
effects/                Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
filters/                Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.

The __init__.py files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, __init__.py can just be an empty file, but it can also execute initialization code for the package or set the __all__ variable, described later.

Users of the package can import individual modules from the package, for example:

import sound.effects.echo

This loads the submodule sound.effects.echo. It must be referenced with its full name.

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

An alternative way of importing the submodule is:

from sound.effects import echo

This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

echo.echofilter(input, output, delay=0.7, atten=4)

Yet another variation is to import the desired function or variable directly:

from sound.effects.echo import echofilter

Again, this loads the submodule echo, but this makes its function echofilter() directly available:

echofilter(input, output, delay=0.7, atten=4)

Note that when using from package import item, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ImportError exception is raised.

Contrarily, when using syntax like import item.subitem. subsubitem, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

Remember

A python package is a collection of modules. Modules that are related to each other are mainly put in the same package. When a module from an external package is required in a program, that package can be imported and its modules can be put to use.

2.3.1 Importing* From a Package

Now what happens when the user writes from sound.effects import *? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The import statement uses the following convention: if a package's __init__.py code defines a list named __all__, it is taken to be the list of module names that should be imported when from package import * is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file sound/effects/__init__.py could contain the following code:

__all__ = ["echo", "surround", "reverse"]

This would mean that from sound.effects import * would import the three named submodules of the sound package.

If __all__ is not defined, the statement from sound.effects import * does *not* import all submodules from the package sound.effects into the current namespace; it only ensures that

the package sound.effects has been imported (possibly running any initialization code in __init__.py) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by __init__.py. It also includes any submodules of the package that were explicitly loaded by previous import statements. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the echo and surround modules are imported in the current namespace because they are defined in the sound.effects package when the from... import statement is executed. (This also works when __all__ is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use import *, it is still considered bad practice in production code.

Remember, there is nothing wrong with using from Package import specific_ submodule! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

2.3.2 Intra-package References

The submodules often need to refer to each other. For example, the surround module might use the echo module. In fact, such references are so common that the import statement first looks in the containing package before looking in the standard module search path. Thus, the surround module can simply use import echo or from echo import echofilter. If the imported module is not found in the current package (the package of which the current module is a submodule), the import statement looks for a top-level module with the given name.

When packages are structured into subpackages (as with the sound package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module sound.filters.vocoder needs to use the echo module in the sound.effects package, it can use fromsound.effects import echo.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the from module import nameform of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the surround module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always "__main__", modules intended for use as the main module of a Python application should always use absolute imports.

2.3.3 Packages in Multiple Directories

Packages support one more special attribute, __path__. This is initialized to be a list containing the name of the directory holding the package's __init__.py before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

SUMMARY

- Python Functions is a block of related statements designed to perform a computational, logical, or evaluative task.
- Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.
- The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.
- The return statement is used to exit a function and go back to the place from where it was called.
- Required arguments are the arguments passed to a function in correct positional order.
- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.

KNOWLEDGE CHECK

1. **Which of these definitions correctly describes a module?**
   a. Denoted by triple quotes for providing the specification of certain program elements
   b. Design and implementation of specific functionality to be incorporated into a program
   c. Defines the specification of how it is to be used
   d. Any program that reuses code
2. **Which of the following is the use of function in python?**
   a. Functions are reusable pieces of programs
   b. Functions don't provide better modularity for your application
   c. you can't also create your own functions
   d. All of the mentioned
3. **Which keyword is use for function?**
   a. Fun
   b. Define
   c. Def
   d. Function
4. **Which of the following is not an advantage of using modules?**
   a. Provides a means of reuse of program code
   b. Provides a means of dividing up tasks
   c. Provides a means of reducing the size of the program
   d. Provides a means of testing individual parts of the program
5. **Program code making use of a given module is called a ..... of the module.**
   a. Client
   b. Docstring
   c. Interface
   d. Modularity
6. **In which language is Python written?**
   a. English
   c. PHP
   d. C
   e. All of the above

7. Which one of the following is the correct extension of the Python file?

a. .py
b. .python
c. .p
d. None of these

8. What is the maximum possible length of an identifier?

a. 31 characters
b. 63 characters
c. 79 characters
d. Identifiers can be of any length.

REVIEW QUESTIONS

1. How Function works in Python?
2. Discuss the python function arguments.
3. How to create a Python module.
4. How to create a module that is executable as a standalone script.
5. What is the difference between a python module and a python package?

Check Your Result

1. (a) 2. (a) 3. (c) 4. (c) 5. (a)
6. (b) 7. (a) 8. (d)

REFERENCES

1. http://codefruxtechnology.com/pdf/PythonSyllabus.pdf
2. https://docs.python.org/3/tutorial/modules.html
3. https://github.com/PyGithub/PyGithub
4. https://realpython.com/python-modules-packages/
5. https://www.codesdope.com/python-boolean/
6. https://www.geeksforgeeks.org/bool-in-python/
7. https://www.programiz.com/python-programming/function-argument
8. https://www.programiz.com/python-programming/methods/built-in/bool
9. https://www.programiz.com/python-programming/modules
10. https://www.tutorialspoint.com/python/python_functions.htm
11. https://www.w3schools.com/python/python_functions.asp
12. Kuchling, A. M. "Functional Programming HOWTO". Python v2.7.2 documentation. Python Software Foundation. Retrieved 9 February 2012.
13. The Python Tutorial. Python Software Foundation. Retrieved 20 February 2012. It is a mixture of the class mechanisms found in C++ and Modula-3
14. Kuchling, A. M. "Functional Programming HOWTO". Python v2.7.2 documentation. Python Software Foundation. Retrieved 9 February 2012.
15. Steven Lott. Copyright © 2005. Steven F. Lott. https://homepage.mac.com/s_lott/books/oodesign/oodesign.pdf. Building Skills in Object-Oriented Design. Step-by-Step Construction of A Complete Application.
16. The Python Tutorial. Python Software Foundation. Retrieved 20 February 2012. It is a mixture of the class mechanisms found in C++ and Modula-3

CHAPTER

3

DICTIONARIES, SETS, AND FILES

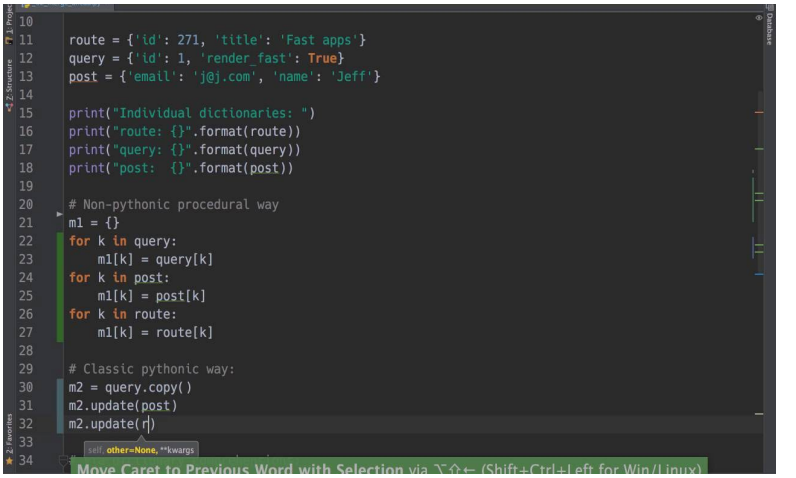
"Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered."

–Guido van Rossum

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Understand python dictionaries
2. Explain python sets
3. Discuss about files



INTRODUCTION

Python dictionaries are something completely different—they are not sequences at all, but are instead known as *mappings*. Mappings are also collections of other objects,

but they store objects by *key* instead of by relative position. In fact, mappings don't maintain any reliable left-to-right order; they simply map keys to associated values. Dictionaries, the only mapping type in Python's core objects set, are also *mutable*: like lists, they may be changed in place and can grow and shrink on demand. Also like lists, they are a flexible tool for representing collections, but their more *mnemonic* keys are better suited when a collection's items are named or labeled—fields of a database record, for example.

Sets are constructed from a sequence (or some other iterable object). Since sets cannot have duplicated, there are usually used to build sequence of unique items (e.g., set of identifiers).

File objects are Python code's main interface to external files on your computer. They can be used to read and write text memos, audio clips, Excel documents, saved email messages, and whatever else you happen to have stored on your machine. Files are a core type, but they're something of an oddball—there is no specific literal syntax for creating them. Rather, to create a file object, you call the built-in open function, passing in an external filename and an optional processing mode as strings.

3.1 PYTHON DICTIONARIES

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Example:

Create and print a dictionary:

thisdict = {

```
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

3.1.1 Accessing Dictionary Elements

We can call the values of a dictionary by referencing the related keys.

Accessing Data Items with Keys

Because dictionaries offer key-value pairs for storing data, they can be important elements in your Python program.

If we want to isolate Sammy's username, we can do so by calling sammy['username']. Let's print that out:

```
print(sammy['username'])
```

Output

```
sammy-shark
```

Dictionaries behave like a database in that instead of calling an integer to get a particular index value as you would with a list, you assign a value to a key and can call that key to get its related value.

By invoking the key 'username' we receive the value of that key, which is 'sammy-shark'.

The remaining values in the sammy dictionary can similarly be called using the same format:

```
sammy['followers']
# Returns 987


sammy['online']
# Returns True
```

By making use of dictionaries' key-value pairs, we can reference keys to retrieve values.

Using Methods to Access Elements

In addition to using keys to access values, we can also work with some built-in methods:

- dict.keys() isolates keys
- dict.values() isolates values
- dict.items() returns items in a list format of (key, value) tuple pairs

To return the keys, we would use the dict.keys() method. In our example, that would use the variable name and be sammy.keys(). Let's pass that to a print() method and look :at the output

Keyword

Data structure is a data organization, management and storage format that enables efficient access and modification.

(()print(sammy.keys

Output

dict_keys(['followers', 'username', 'online'])

We receive output that places the keys within an iterable view object of the dict_keys class. The keys are then printed within a list format.

This method can be used to query across dictionaries. For example, we can take a look at the common keys shared between two dictionary **data structures**:

```
sammy = {'username': 'sammy-shark', 'online': True, 'followers': 987}
jesse = {'username': 'JOctopus', 'online': False, 'points': 723}
for common_key in sammy.keys() & jesse.keys():
    print(sammy[common_key], jesse[common_key])
```

The dictionary sammy and the dictionary jesse are each a user profile dictionary.

Their profiles have different keys, however, because Sammy has a social profile with associated followers, and Jesse has a gaming profile with associated points. The 2 keys they have in common are usernameand online status, which we can find when we run this small program:

Output

sammy-shark JOctopus

True False

We could certainly improve on the program to make the output more user-readable, but this illustrates that dict.keys() can be used to check across various dictionaries to see what they share in common or not. This is especially useful for large dictionaries.

Similarly, we can use the dict.values() method to query the values in the sammy dictionary, which would be constructed as sammy.values(). Let's print those out:

sammy = {'username': 'sammy-shark', 'online': **True**, 'followers': 987}

print(sammy.values())

Output

dict_values([True, 'sammy-shark', 987])

Both the methods keys() and values() return unsorted lists of the keys and values present in the sammy dictionary with the view objects of dict_keys and dict_values respectively.

If we are interested in all of the items in a dictionary, we can access them with the items() method:

print(sammy.items())

Output

dict_items([('online', True), ('username', 'sammy-shark'), ('followers', 987)])

The returned format of this is a list made up of (key, value) tuple pairs with the dict_items view object.

We can iterate over the returned list format with a for loop. For example, we can print out each of the keys and values of a given dictionary, and then make it more human-readable by adding a string:

for key, value **in** sammy.items():

    print(key, 'is the key for the value', value)

Output

online is the key for the value True

followers is the key for the value 987

username is the key for the value sammy-shark

The for loop above iterated over the items within the sammy dictionary and printed out the keys and values line by line, with information to make it easier to understand by humans.

We can use built-in methods to access items, values, and keys from dictionary data structures.

3.1.2 Modifying Dictionaries

Dictionaries are a mutable data structure, so you are able to modify them. In this section, we will go over adding and deleting dictionary elements.

Adding and Changing Dictionary Elements

Without using a method or function, you can add key-value pairs to dictionaries by using the following syntax:

dict[key] = value

We'll look at how this works in practice by adding a key-value pair to a dictionary called usernames:

usernames = {'Sammy': 'sammy-shark', 'Jamie': 'mantisshrimp54'}

usernames['Drew'] = 'squidly'

print(usernames)

Output

{'Drew': 'squidly', 'Sammy': 'sammy-shark', 'Jamie': 'mantisshrimp54'}

We see now that the dictionary has been updated with the 'Drew': 'squidly' key-value pair. Because dictionaries may be unordered, this pair may occur anywhere in the dictionary output. If we use the usernames dictionary later in our program file, it will include the additional key-value pair.

Additionally, this syntax can be used for modifying the value assigned to a key. In this case, we will reference an existing key and pass a different value to it.

Let's consider a dictionary drew that is one of the users on a given network. We will say that this user got a bump in followers today, so we need to update the **integer** value passed to the 'followers' key. We'll use the print() function to check that the dictionary was modified.

drew = {'username': 'squidly', 'online': **True**, 'followers': 305}

drew['followers'] = 342

print(drew)

Output

{'username': 'squidly', 'followers': 342, 'online': True}

In the output, we see that the number of followers jumped from the integer value of 305 to 342.

We can use this method for adding key-value pairs to dictionaries with user-input. Let's write a quick program, usernames.py that runs on the command line and allows input from the user to add more names and associated usernames:

usernames.py

```
# Define original dictionary
usernames = {'Sammy': 'sammy-shark', 'Jamie': 'mantisshrimp54'}
# Set up while loop to iterate
while True:

    # Request user to enter a name
    print('Enter a name:')

    # Assign to name variable
    name = input()

    # Check whether name is in the dictionary and print feedback
    if name in usernames:
        print(usernames[name] + ' is the username of ' + name)

    # If the name is not in the dictionary...
    else:

        # Provide feedback
        print('I don\'t have ' + name + '\'s username, what is it?')

        # Take in a new username for the associated name
        username = input()

        # Assign username value to name key
        usernames[name] = username

        # Print feedback that the data was updated
        print('Data updated.')
```

Let's run the program on the command line:

```
python usernames.py
```

When we run the program we'll get something like the following output:

Output

Enter a name:

Sammy

sammy-shark is the username of Sammy

Enter a name:

Jesse

I don't have Jesse's username, what is it?

JOctopus

Data updated.

Enter a name:

When we are done testing the program, we can press CTRL + C to escape the program. You can set up a trigger to quit the program (such as typing the letter q) with a **conditional statement** to improve the code.

Keyword

Conditional statements are those statements where a hypothesis is followed by a conclusion. It is also known as an " If-then" statement. If the hypothesis is true and the conclusion is false, then the conditional statement is false.

This shows how you can modify dictionaries interactively. With this particular program, as soon as you exit the program with CTRL + C you'll lose all your data unless you implement a way to handle reading and writing files.

We can also add and modify dictionaries by using the dict.update() method. This varies from the append() method available in lists.

In the jesse dictionary below, let's add the key 'followers' and give it an integer value with jesse.update(). Following that, let's print() the updated dictionary.

```
jesse = {'username': 'JOctopus', 'online': False, 'points': 723}
jesse.update({'followers': 481})
print(jesse)
```

Output

{'followers': 481, 'username': 'JOctopus', 'points': 723, 'online': False}

From the output, we can see that we successfully added the 'followers': 481 key-value pair to the dictionary jesse.

We can also use the dict.update() method to modify an existing key-value pair by replacing a given value for a specific key.

Let's change the online status of Sammy from True to False in the sammy dictionary:

sammy = {'username': 'sammy-shark', 'online': **True**, 'followers': 987}

sammy.update({'online': **False**})

print(sammy)

Output

{'username': 'sammy-shark', 'followers': 987, 'online': False}

The line sammy.update({'online': False}) references the existing key 'online' and modifies its Boolean value from True to False. When we call to print() the dictionary, we see the update take place in the output.

To add items to dictionaries or modify values, we can use wither the dict[key] = value syntax or the method dict.update().

Keyword

Syntax is the grammatical structure of sentences. The format in which words and phrases are arranged to create sentences is called syntax.

Deleting Dictionary Elements

Just as you can add key-value pairs and change values within the dictionary data type, you can also delete items within a dictionary.

To remove a key-value pair from a dictionary, we will use the following **syntax**:

del dict[key]

Let's take the jesse dictionary that represents one of the users. We'll say that Jesse is no longer using the online platform for playing games, so we'll remove the item associated with the 'points' key. Then, we'll print the dictionary out to confirm

that the item was deleted:

```
jesse = {'username': 'JOctopus', 'online': False, 'points': 723, 'followers': 481}

del jesse['points']
print(jesse)
```

Output

```
{'online': False, 'username': 'JOctopus', 'followers': 481}
```

The line del jesse['points'] removes the key-value pair 'points': 723 from the jesse dictionary.

If we would like to clear a dictionary of all of its values, we can do so with the dict.clear() method. This will keep a given dictionary in case we need to use it later in the program, but it will no longer contain any items.

Let's remove all the items within the jesse dictionary:

```
jesse = {'username': 'JOctopus', 'online': False, 'points': 723, 'followers': 481}
jesse.clear()

print(jesse)
```

Output

```
{}
```

The output shows that we now have an empty dictionary devoid of key-value pairs.

If we no longer need a specific dictionary, we can use del to get rid of it entirely:

```
del jesse
print(jesse)
```

When we run a call to print() after deleting the jesse dictionary, we'll receive the following error:

Output

```
...
NameError: name 'jesse' is not defined
```

Because dictionaries are mutable data types, they can be added to, modified, and have items removed and cleared.

The del keyword removes the item with the specified key name:


```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
del thisdict["model"]
print(thisdict)
```

3.1.3 The dict() Constructor

It is also possible to use the dict() constructor to make a dictionary:

```
Example
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

3.1.4 Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair
keys()	Returns a list contianing the dictionary's keys
pop()	Removes the element with the specified key

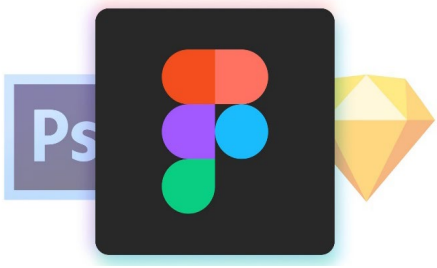
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Remember

Although we just used a dictionary to link names to phone numbers in this extended example, we can use a dictionary to link any one type of object to another type of object.

3.1.5 Aliasing and Copying

Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.



If you want to modify a dictionary and keep a copy of the original, use the copy method. For example, opposites is a dictionary that contains pairs of opposites:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> an_alias = opposites
>>> a_copy = opposites.copy()
```

an_alias and opposites refer to the same object; a_copy refers to a fresh copy of the same dictionary. If we modify alias, opposites is also changed:

```
>>> an_alias['right'] = 'left'
>>> opposites['right']
'left'
```

If we modify a_copy, opposites is unchanged:

```
>>> a_copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

3.2 PYTHON SETS

Sets are a collection of distinct (unique) objects. These are useful to create lists that only hold unique values in the dataset. It is an unordered collection but a mutable one, this is very helpful when going through a huge dataset.

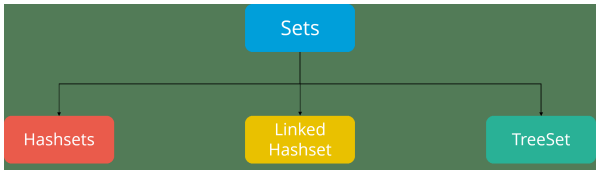
```
x_set = set('CAKE&COKE')
y_set = set('COOKIE')
print(x_set)
{'A', '&', 'O', 'E', 'C', 'K'}
print(y_set) # Single unique 'o'
{'I', 'O', 'E', 'C', 'K'}
print(x - y) # All the elements in x_set but not in y_set
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)
<ipython-input-3-31abf5d98454> in <module>()
----> 1 print(x - y) # All the elements in x_set but not in y_set

NameError: name 'x' is not defined
print(x_set|y_set) # Unique elements in x_set or y_set or both
{'C', '&', 'E', 'A', 'O', 'K', 'I'}
print(x_set & y_set) # Elements in both x_set and y_set
{'O', 'E', 'K', 'C'}
```

Remember

Sets can store anything, not just strings. It's just easiest to illustrate the set methods using sets of strings.

3.2.1 Defining a Set

You can define a set as simple as by naming all of its elements in brackets. The only exception is *empty set*, which can be created using the function set(). If set(..) has a list, a string or a tuple as a parameter, it will return a set composed of its elements.



Let's see what all that means, and how you can work with sets in Python.

A set can be created in two ways. First, you can define a set with the built-in set() function:

x = set(<iter>)

In this case, the argument <iter> is an iterable—again, for the moment, think list or tuple—that generates the list of objects to be included in the set. This is analogous to the <iter>argument given to the .extend() list method:

Keyword

String is traditionally a sequence of characters, either as a literal constant or as some kind of variable.

```
>>> x = set(['foo', 'bar', 'baz', 'foo', 'qux'])
>>> x
{'qux', 'foo', 'bar', 'baz'}

>>> x = set(('foo', 'bar', 'baz', 'foo', 'qux'))
>>> x
{'qux', 'foo', 'bar', 'baz'}
```

Strings are also iterable, so a string can be passed to set() as well. You have already seen that list(s) generates a list of the characters in the **string** s. Similarly, set(s) generates a set of the characters in s:

```
>>> s = 'quux'

>>> list(s)
['q', 'u', 'u', 'x']
>>> set(s)
{'x', 'u', 'q'}
```

You can see that the resulting sets are unordered: the original order, as specified in the definition, is not necessarily preserved. Additionally, duplicate values are only represented in the set once, as with the string 'foo' in the first two examples and the letter 'u' in the third.

Alternately, a set can be defined with curly braces ({}):

x = {<obj>, <obj>, ..., <obj>}

When a set is defined this way, each <obj> becomes a distinct element of the set, even if it is an iterable. This behavior is similar to that of the .append() list method.

Thus, the sets shown above can also be defined like this:

```
>>> x = {'foo', 'bar', 'baz', 'foo', 'qux'}
>>> x
{'qux', 'foo', 'bar', 'baz'}

>>> x = {'q', 'u', 'u', 'x'}
>>> x
{'x', 'q', 'u'}
```

To recap:

- The argument to set() is an iterable. It generates a list of elements to be placed into the set.
- The objects in curly braces are placed into the set intact, even if they are iterable.

Observe the difference between these two set definitions:

```
>>> {'foo'}
{'foo'}

>>> set('foo')
{'o', 'f'}
```

A set can be empty. However, recall that Python interprets empty curly braces ({}) as an empty dictionary, so the only way to define an empty set is with the set() function:

```
>>> x = set()
>>> type(x)
<class 'set'>
>>> x
set()

>>> x = {}
>>> type(x)
```

```
<class 'dict'>
```

An empty set is falsy in Boolean context:

```
>>> x = set()
>>> bool(x)
False
>>> x or 1
1
>>> x and 1
set()
```

You might think the most intuitive sets would contain similar objects—for example, even numbers or surnames:

```
>>> s1 = {2, 4, 6, 8, 10}
>>> s2 = {'Smith', 'McArthur', 'Wilson', 'Johansson'}
```

Python does not require this, though. The elements in a set can be objects of different types:

```
>>> x = {42, 'foo', 3.14159, None}
>>> x
{None, 'foo', 42, 3.14159}
```

Don't forget that set elements must be immutable. For example, a tuple may be included in a set:

```
>>> x = {42, 'foo', (1, 2, 3), 3.14159}
>>> x
{42, 'foo', 3.14159, (1, 2, 3)}
```

But lists and dictionaries are mutable, so they can't be set elements:

```
>>> a = [1, 2, 3]
>>> {a}
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    {a}
TypeError: unhashable type: 'list'

>>> d = {'a': 1, 'b': 2}
>>> {d}
```

Did You Know?

Earlier versions of Python used a cryptic way to format strings. It is considered deprecated and will eventually disappear from the language.

```
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    {d}
TypeError: unhashable type: 'dict'
```

3.2.2 Set Size and Membership

The len() function returns the number of elements in a set, and the in and not inoperators can be used to test for membership:

```
>>> x = {'foo', 'bar', 'baz'}


>>> len(x)
3


>>> 'bar' in x
True
>>> 'qux' in x
False
```

3.2.3 Methods for Sets

add(x) Method

Adds the item x to set if it is not already present in the set.

people = {"Jay", "Idrish", "Archil"}

people.add("Daxit")

-> This will add Daxit in people set.

union(s) Method

Returns a union of two set.Using the '|' operator between 2 sets is the same as writing set1.union(set2)

people = {"Jay", "Idrish", "Archil"}

vampires = {"Karan", "Arjun"}

population = people.union(vampires)

OR

population = people|vampires

-> Set population set will have components of both people and vampire

intersect(s) Method

Returns an intersection of two sets.The '&' operator comes can also be used in this case.

victims = people.intersection(vampires)

-> Set victims will contain the common element of people and vampire

difference(s) Method

Returns a set containing all the elements of invoking set but not of the second set. We can use '-' operator here.

safe = people.difference(vampires)

OR

safe = people – vampires

-> Set safe will have all the elements that are in people but not vampire

clear() Method

Empties the whole set.

victims.clear()

-> Clears victim set

However there are two major pitfalls in Python sets:

- The set doesn't maintain elements in any particular order.
- Only instances of immutable types can be added to a Python set.

3.2.4 Creating a Set

A set is created by using the set() function or placing all the elements within a pair of curly braces.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

3.2.5 Accessing Values in a Set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])

for d in Days:
        print(d)
```

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

```
Wed
Sun
Fri
Tue
Mon
Thu
Sat
```

3.2.6 Adding Items to a Set

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])

Days.add("Sun")
print(Days)
```

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

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

3.2.7 Removing Item from a Set

We can remove elements from a set by using discard() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])

Days.discard("Sun")
print(Days)
```

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

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

3.2.8 Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element "Wed" is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one "wed".

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Did You Know?

Since 2003, Python has consistently ranked in the top ten most popular programming languages in the TIOBE Programming Community Index where, as of February 2021, it is the third most popular language (behind Java, and C).

3.2.9 Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element "Wed" is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one "wed".

```
set(['Wed'])
```

3.2.10 Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element "Wed" is present in both the sets so it will not be found in the result set.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one "wed".

```
set(['Mon', 'Tue'])
```

3.2.11 Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
SubsetRes = DaysA <= DaysB
SupersetRes = DaysB >= DaysA
print(SubsetRes)
print(SupersetRes)
```

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

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

```
True
True
```

3.3 FILES

Files are traditionally a part of data structures. And although big data is commonplace in the data science industry, a programming language without the capability to store and retrieve previously stored information would hardly be useful. You still have to make use of the all the data sitting in files across databases and you will learn how to do this.

The syntax to read and write files in Python is similar to other **programming languages** but a lot easier to handle. Here are some of the basic functions that will :help you to work with files using Python

- open() to open files in your system, the filename is the name of the file to be opened;
- read() to read entire files;
- readline() to read one line at a time;
- write() to write a string to a file, and return the number of characters written; And
- close() to close the file.

Keyword

Programming language is a formal language, which comprises a set of instructions used to produce various kinds of output.

```
# File modes (2nd argument): 'r'(read), 'w'(write), 'a'(appending), 'r+'(both reading and writing)
f = open('file_name', 'w')


# Reads entire file
f.read()
# Reads one line at a time
f.readline()
# Writes the string to the file, returning the number of char written
f.write('Add this line.')
f.close()
```

The second argument in the open() function is the file mode. It allows you to specify whether you want to read (r), write (w), append (a) or both read and write (r+).

3.3.1 The open function

The open function takes two arguments. The first is the name of the file, and the second is the **mode**. Mode 'w' means that we are opening the file for writing. Mode 'r' means reading, and mode 'a' means appending.

Let's begin with an example that shows these three modes in operation:

```
>>> myfile = open('test.txt', 'w')
>>> myfile.write('My first file written from Python\n')
34
>>> myfile.write('---------------------------------\n')
34
>>> myfile.write('Hello, world!')
13
>>> myfile.close()
>>> myfile = open('test.txt', 'r')
>>> contents = myfile.read()
>>> myfile.close()
>>> print(contents)
My first file written from Python
---------------------------------
Hello, world!
```

Opening a file creates what we call a file descriptor. In this example, the variable myfile refers to the new descriptor object. Our program calls methods on the descriptor, and this makes changes to the actual file which is located in non-volatile storage.

The first line opens the test.txt for writing. If there is no file named test.txt on the disk, it will be created. If there already is one, it will be replaced by the file we are writing and any previous data in it will be lost.

To put data in the file we invoke the write method on the file descriptor. We do this three times in the example above, but in bigger programs, the three separate calls to write will usually be replaced by a loop that writes many more lines into the file. The write method returns the number of bytes (characters) written to the file.

Closing the file handle tells the system that we are done writing and makes the disk file available for reading by other programs (or by our own program).

We finish this example by openning test.txt for reading. We then call the read method, assigning the contents of the file, which is a string, to a variable named contents, and finally print contents to see that it is indeed what we wrote to the file previously.

If we want to add to an already existing file, use the *append* mode.

Remember

Many common files you may use, such as Word documents or Excel spreadsheets are NOT text documents. They have their own complicated file formats. Text files are ones you can create in simple text editors, by simply typing regular keys without any special features, such as bolding text, different fonts, etc.

```
>>> myfile = open('test.txt', 'a')
>>> myfile.write('\nOoops, I forgot to add this line ;-)')
37
>>> myfile.close()
>>> myfile = open('test.txt', 'r')
>>> print(myfile.read())
My first file written from Python
--------------------------------
Hello, world!
Ooops, I forgot to add this line ;-)
>>>
```

3.3.2 Opening a File that Doesn't Exist

If we try to open a file that doesn't exist, we get an error:

```
>>> f = open('wharrah.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'wharrah.txt'
>>>
```

Keyword

Error is something you have done which is considered to be incorrect or wrong, or which should not have been done.

There is nothing wrong with the syntax of the line that resulted in the error. The error occurred because the file did not exist. **Errors** like these are called **exceptions**. Most modern programming languages provide support for dealing with situations like this. The process is called exception handling.

In Python, exceptions are handled with the try ... except statement.

```
try:
    f = open('thefile.txt', 'r')
    mydata = f.read()
    f.close()
except IOError:
    mydata = ''
```

In this example we *try* to open the data file for reading. If it succeeds, we use the read() method to read the file contents as a string into the variable mydata and close the file. If an IOError exception occurs, we still create mydata as an empty string and continue on with the program.

3.3.3 Reading Data from Files

Python file descriptors have three methods for reading in data from a file. We've already seen the read() method, which returns the entire contents of the file as a single string. For really big files this may not be what you want.

The readline() method returns one line of the file at a time. Each time you call it readline() returns the next line. Calls made to readline() after reaching the end of the file return an empty string (‘’).

```
f = open('test.txt', 'r')
while True:                             # keep reading forever
    theline = f.readline()              # try to read next line
    if len(theline) == 0:               # if there are no more lines
        break                           # leave the loop

    # Now process the line we've just read
    print(theline, end='')

f.close()
```

This is a handy pattern for our toolbox. In bigger programs, we'd squeeze more extensive logic into the body of the loop at line 8 — for example, if each line of the file contained the name and email address of one of our friends, perhaps we'd split the line into some pieces and call a function to send the friend a party invitation.

On line 8 we suppress the newline character that print usually appends to our strings. Why? This is because the string already has its own newline: the readline method in line 3 returns everything up to *and including* the newline character. This also explains the end-of-file detection logic: when there are no more lines to be read from the file, readline returns an empty string — one that does not even have a newline at the end, hence it's length is 0.

Turning a File into a List of Lines

It is often useful to fetch data from a disk file and turn it into a list of lines. Suppose we have a file containing our friends and their email addresses, one per line in the file. But we'd like the lines sorted into alphabetical order. A good plan is to read everything into a list of lines, then sort the list, and then write the sorted list back to another file:

```
f1 = open('friends.txt', 'r')
friends_list = f.readlines()
f1.close()

friends_list.sort()

f2 = open('sortedfriends.txt', 'w')
for friend in friends_list:
    f2.write(v)
f2.close()
```

The readlines method in line 2 reads all the lines and returns a list of the strings.

We could have used the template from the previous section to read each line one-at-a-time, and to build up the list ourselves, but it is a lot easier to use the method that the Python implementors gave us!

An Example

Many useful line-processing programs will read a text file line-at-a-time and do some minor processing as they write the lines to an output file. They might number the lines in the output file, or insert extra blank lines after every 60 lines to make it convenient for printing on sheets of paper, or extract some specific columns only from each line in the source file, or only print lines that contain a specific substring. We call this kind of program a **filter**.

Here is a filter that copies one file to another, omitting any lines that begin with #:

```
infile = open(oldfile, 'r')
outfile = open(newfile, 'w')
while True:
    text = infile.readline()
    if len(text) == 0:
        break
    if text[0] == '#':
        continue

    # put any more processing logic here
    outfile.write(text)

infile.close()
outfile.close()
```

The continue statement at line 9 skips over the remaining lines in the current iteration of the loop, but the loop will still iterate. This style looks a bit contrived here, but it is often useful to say *"get the lines we're not concerned with out of the way early, so that we have cleaner more focussed logic in the meaty part of the loop that might be written around line 11."*

Thus, if text is the empty string, the loop exits. If the first character of text is a hash mark, the flow of execution goes to the top of the loop, ready to start processing the next line. Only if both conditions fail do we fall through to do the processing at line 11, in this example, writing the line into the new file.

Let's consider one more case: suppose your original file contained empty lines. At line 6 above, would this program find the first empty line in the file, and terminate immediately? No! Recall that readline always includes the newline character in the string it returns. It is only when we try to read *beyond* the end of the file that we get back the empty string of length 0.

CASE STUDY

A SET PARTITIONING PROBLEM

A set partitioning problem determines how the items in one set (S) can be partitioned into smaller subsets. All items in S must be contained in one and only one partition. Related problems are:

- set packing - all items must be contained in zero or one partitions;
- set covering - all items must be contained in at least one partition.

In this case study a wedding planner must determine guest seating allocations for a wedding. To model this problem the tables are modelled as the partitions and the guests invited to the wedding are modelled as the elements of S. The wedding planner wishes to maximize the total happiness of all of the tables.

A set partitioning problem may be modelled by explicitly enumerating each possible subset. Though this approach does become intractable for large numbers of items (without using column generation) it does have the advantage that the objective function co-efficients for the partitions can be non-linear expressions (like happiness) and still allow this problem to be solved using Linear Programming.

First we use **allcombinations()** to generate a list of all possible table seatings.

```
#create list of all possible tables
possible_tables = [tuple(c) for c in pulp.allcombinations(guests,
                                                  max_table_size)]
```

Then we create a binary variable that will be 1 if the table will be in the solution, or zero otherwise.

```
#create a binary variable to state that a table setting is used
x = pulp.LpVariable.dicts('table', possible_tables,
                          lowBound = 0,
                          upBound = 1,
                          cat = pulp.LpInteger)
```

We create the **LpProblem** and then make the objective function. Note that happiness function used in this script would be difficult to model in any other way.

```
seating_model = pulp.LpProblem("Wedding Seating Model", pulp.LpMinimize)
seating_model += sum([happiness(table) * x[table] for table in possible_tables])
```

We specify the total number of tables allowed in the solution.

```
#specify the maximum number of tables
seating_model += sum([x[table] for table in possible_tables]) <= max_tables, \
```

This set of constraints defines the set partitioning problem by guaranteeing that a guest is allocated to exactly one table.

```
#A guest must seated at one and only one table
for guest in guests:
    seating_model += sum([x[table] for table in possible_tables
                          if guest in table]) == 1, "Must_seat_%s"%guest
```

The full file can be found here wedding.py

```
"""
A set partitioning model of a wedding seating problem

Authors: Stuart Mitchell 2009
"""

import pulp

max_tables = 5
max_table_size = 4
guests = 'A B C D E F G I J K L M N O P Q R'.split()

def happiness(table):
    """
    Find the happiness of the table
```

```
    - by calculating the maximum distance between the letters
    """
    return abs(ord(table[0]) - ord(table[-1]))

#create list of all possible tables
possible_tables = [tuple(c) for c in pulp.allcombinations(guests,
                                        max_table_size)]

#create a binary variable to state that a table setting is used
x = pulp.LpVariable.dicts('table', possible_tables,
                          lowBound = 0,
                          upBound = 1,
                          cat = pulp.LpInteger)

seating_model = pulp.LpProblem("Wedding Seating Model", pulp.LpMinimize)

seating_model += sum([happiness(table) * x[table] for table in possible_tables])

#specify the maximum number of tables
seating_model += sum([x[table] for table in possible_tables]) <= max_tables, \
                          "Maximum_number_of_tables"

#A guest must seated at one and only one table
for guest in guests:
    seating_model += sum([x[table] for table in possible_tables
                          if guest in table]) == 1, "Must_seat_%s"%guest

seating_model.solve()

print("The choosen tables are out of a total of %s:"%len(possible_tables))
for table in possible_tables:
    if x[table].value() == 1.0:
        print(table)
```

SUMMARY

- Dictionary in Python is an unordered collection of data values, used to store data values like a map, which, unlike other Data Types that hold only a single value as an element, Dictionary holds key: value pair. Key-value is provided in the dictionary to make it more optimized.
- Sets are used to store multiple items in a single variable. Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.
- If you want to modify a dictionary and keep a copy of the original, use the copy method.
- Sets are a collection of distinct (unique) objects. These are useful to create lists that only hold unique values in the dataset.
- A set is created by using the set() function or placing all the elements within a pair of curly braces.
- The union operation on two sets produces a new set containing all the distinct elements from both the sets.
- The intersection operation on two sets produces a new set containing only the common elements from both the sets.

KNOWLEDGE CHECK

1. **Data dictionary is a special file that contains**
   a. the names of all fields in all files
   b. the data types of all fields of all files
   c. Both of above
   d. None of above
2. **Physical location of a record in database is determined with the help of**
   a. B tree file
   b. Indexed file
   c. Hashed file
   d. sequential file
3. **Which of the following statements create a dictionary?**
   a. d = {}
   b. d = {“john”:40, “peter”:45}
   c. d = {40:”john”, 45:”peter”}
   d. All of the mentioned
4. **Read the code shown below carefully and pick out the keys?**
   d = {“john”:40, “peter”:45}
   a. “john”, 40, 45, and “peter”
   b. “john” and “peter”
   c. 40 and 45
   d. d = (40:”john”, 45:”peter”)
5. **Which of the following isn’t true about dictionary keys?**
   a. More than one key isn’t allowed
   b. Keys must be immutable
   c. Keys must be integers
   d. When duplicate keys encountered, the last assignment wins
6. **Given a function that does not return any value, What value is thrown by default when executed in shell.**
   a. Int
   b. bool
   c. void
   d. none

7. **In python we do not specify types, it is directly interpreted by the compiler, so consider the following operation to be performed.**
   a. x = 13 // 2
   b. x = int(13 / 2)
   c. x = 13 % 2
   d. All of the mentioned
8. **What is the value of the following expression?**
   a. (1.0, 4.0)
   b. (1.0, 1.0)
   c. (4.0. 1.0)
   d. (4.0, 4.0)

REVIEW QUESTIONS

1. What is the output of the following code?

   a={1:"A",2:"B",3:"C"}

   a.setdefault(4,"D")

   print(a)
2. Write a program that reads the words in words.txt and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.
3. Write a program that asks the user to enter 10 words and prints out the word that comes first alphabetically.
4. Generate a set containing each positive integer less than 1000 divisible by 15 and a second set containing each positive integer less than 1000 divisible by 21. Create a set of integers that is divisible by either value, both values and exactly one value. Print out the contents of each of these resultant sets.
5. What will be the output?
   a. d = {"john":40, "peter":45}
   b. "john" in d

Check Your Result

1. (b) 2. (c) 3. (d) 4. (b) 5. (c)
6. (d) 7. (d) 8. (a)

REFERENCES

1. http://www.openbookproject.net/books/bpp4awd/ch06.html
2. http://www.u.arizona.edu/~erdmann/mse350/topics/list_comprehensions.html
3. https://docs.python.org/2/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops
4. https://www.digitalocean.com/community/tutorials/understanding-dictionaries-in-python-3
5. https://www.tutorialspoint.com/python/python_sets.htm
6. https://www.w3schools.com/python/python_dictionaries.asp
7. Holth, Moore (30 March 2014). "PEP 0441 -- Improving Python ZIP Application Support". Archived from the original on 26 December 2018. Retrieved 12 November 2015.
8. Rossum, Guido Van (20 January 2009). "The History of Python: A Brief Timeline of Python". The History of Python. Archived from the original on 5 June 2020. Retrieved 5 March 2021.
9. Schemenauer, Neil; Peters, Tim; Hetland, Magnus Lie (18 May 2001). "PEP 255 – Simple Generators". Python Enhancement Proposals. Python Software Foundation. Archived from the original on 5 June 2020. Retrieved 9 February 2012.

CHAPTER

4

EXCEPTIONS, UNIT TESTING AND COMPREHENSIONS

"Abstraction is one of those notions that Python tosses out the window, yet expresses very well"

–Gordon McMillan

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. How to use the exceptions?
2. Explain the unit testing
3. Understanding the comprehensions

```
// ItemServiceTest.java

@Test
public void findItemBasic() throws NotFoundException {
    // PREPARATION
    Mockito.when(itemRepository.findById(1)).thenReturn(
            Optional.of(new Item(1, "Ball 1", 10, 1000))
    );

    // ACTION
    Item item = itemService.findOneItem(1);

    // ASSERTION
    assertEquals(10000, item.getValue());
}

@Test(expected = NotFoundException.class)
public void findItemNotFound() throws NotFoundException {
    // PREPARATION
    Mockito.when(itemRepository.findById(1)).thenReturn(
            Optional.empty()
    );

    // ACTION
    Item item = itemService.findOneItem(1);

    // ASSERTION never happen because exception thrown

}
```

INTRODUCTION

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters

a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. Here, you will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.

Unit testing is a software testing method by which individual units of source code are put under various tests to determine whether they are fit for use (Source). It determines and ascertains the quality of your code.

Generally, when the development process is complete, the developer codes criteria, or the results that are known to be potentially practical and useful, into the test script to verify a particular unit's correctness. During test case execution, various frameworks log tests that fail any criterion and report them in a summary.

The developers are expected to write automated test scripts, which ensures that each and every section or a unit meets its design and behaves as expected.

Though writing manual tests for your code is definitely a tedious and time-consuming task, Python's built-in unit testing framework has made life a lot easier.

4.1 EXCEPTIONS

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>>
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
```

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

TypeError: Can't convert 'int' object to str implicitly

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack trace back. In general it contains a stack trace back listing source lines; however, it will not display lines read from standard input.

Keyword

Stack trace is a report of the active stack frames at a certain point in time during the execution of a program.

4.1.1 Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-Cor whatever the operating system supports); note that a user-generated interruption is signalled by raising the KeyboardInterrupt exception.

```
>>>
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...       print("Oops!  That was no valid number.  Try again...")
...
```

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown.

Remember

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when the some internal events occur which changes the normal flow of the program.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass
```

```
for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
import sys


try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

The try … except statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
```

```
except OSError:
    print('cannot open', arg)
else:
    print(arg, 'has', len(f.readlines()), 'lines')
    f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines__str__() so the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>>
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
>>>
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

4.1.2 Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
>>>
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
>>>
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
```

```
...     raise
...
An exception flew by!
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
NameError: HiThere
```

4.1.3 User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the Exception class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass
class InputError(Error):
    """Exception raised for errors in the input.
Live Demo
#!/usr/bin/python
```


```
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

```
Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
   """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
       message -- explanation of why the specific transition is not
allowed
    """

    def __init__(self, previous, next, message):
```

```
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end in "Error," similar to the naming of the standard exceptions.

4.1.4 Defining Clean-up Actions

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>>
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A *finally clause* is always executed before leaving the try statement, whether an exception has occurred or not. When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in an except or else clause), it is re-raised after the finally clause has been executed. The finallyclause is also executed "on the way out" when any other clause of the try statement is left via a break, continue or **return statement.** A more complicated example:

Keyword

Return Statement causes execution to leave the current subroutine and resume at the point in the code immediately after where the subroutine was called, known as its return address.

```
>>>
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
```

```
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
    TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the finally clause is executed in any event. The TypeError raised by dividing two **strings** is not handled by the except clause and therefore re-raised after the finally clause has been executed.

In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

Keyword

String
a long flexible structure made from threads twisted together, which is used to tie, bind, or hang other objects

4.1.5 Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the

code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The with statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

4.2 UNIT TESTING

The Python unit testing framework, sometimes referred to as "PyUnit," is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent's Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language.

Keyword

Test Automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes.

unittest supports **test automation**, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The unittest module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, unittest supports some important concepts:

test fixture

A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case

A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite

A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner

A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

The test case and **test fixture** concepts are supported through the TestCase and FunctionTestCase classes; the former should be used when creating new tests, and the latter can be used when integrating existing test code with a unittest-driven framework. When building test fixtures using TestCase, the setUp() and tearDown() methods can be overridden to provide initialization and cleanup for the fixture. With FunctionTestCase, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the TestCase will only be used to run a single test method, so a new fixture is created for each test.

Keyword

Test fixture is an environment used to consistently test some item, device, or piece of software.

Test suites are implemented by the TestSuite class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in "child" test suites are run.

A test runner is an object that provides a single method, run(), which accepts a TestCase or TestSuite object as a parameter, and returns a result object. The class TestResult is provided for use as the result object. unittest provides the TextTestRunner as an example test runner which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.

4.2.1 Basic example

The unittest module provides a rich set of tools for constructing and running tests. This demonstrates that a small subset of the tools suffices to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest
class TestStringMethods(unittest.TestCase):
```

```
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())
    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to assertEqual() to check for an expected result; assertTrue() or assertFalse() to verify a condition; or assertRaises() to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

The setUp() and tearDown() methods allow you to define instructions that will be executed before and after each test method.

The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script. When run from the command line, the script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Instead of unittest.main(), there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with:

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

Running the revised script from the interpreter or another script produces the following output:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s
OK
```

The examples show the most commonly used unittest features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

4.2.2 Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v test_module
```

For a list of all the command-line options:

```
python -m unittest -h
```

Changed in version 2.7: In earlier versions it was only possible to run individual test methods and not modules or classes.

Command-line options

unittest supports these command-line options:

-b, --buffer

The standard output and standard error streams are buffered during the test run.

Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure messages.

-c, --catch

Control-C during the test run waits for the current test to end and then reports all the results so far. A second Control-C raises the normal Keyboard Interrupt exception.

See Signal Handling for the functions that provide this functionality.

-f, --failfast

Stop the test run on the first error or failure.

New in version 2.7: The command-line options -b, -c and -f were added.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

4.2.3 Test Discovery

Remember

Test discovering are the steps that are taken to find the tests in your code-base. This means you don't have to specify where your tests are but if the files contains the tests follow a certain location (filenames, directories, etc) then the testing framework can find them automatically.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in TestLoader.discover(), but can also be used from the command line. The basic command-line usage is:

cd project_directory

python -m unittest discover

The discover sub-command has the following options:

-v, --verbose

Verbose output

-s, --start-directory directory

Directory to start discovery (. default)

-p, --pattern pattern

Pattern to match test files (test*.py default)

-t, --top-level-directory directory

Top level directory of project (defaults to start directory)

The -s, -p, and -t options can be passed in as positional arguments in that order. The following two command lines are equivalent:

python -m unittest discover -s project_directory -p "*_test.py"

python -m unittest discover project_directory "*_test.py"

As well as being a path it is possible to pass a package name, for example myproject.subpackage.test, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

Caution: Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example foo/bar/baz.py will be imported as foo.bar.baz.

If you have a package installed globally and attempt test discovery on a different copy of the package, then the import could happen from the wrong place. If this happens test discovery will warn you and exit.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the load_tests protocol.

4.2.4 Organizing test code

The basic building blocks of unit testing are test cases — single scenarios that must be set up and checked for correctness. In unittest, test cases are represented by instances of unittest's TestCase class. To make your own test cases you must write subclasses of TestCase, or use FunctionTestCase.

An instance of a TestCase-derived class is an object that can completely run a single test method, together with optional set-up and tidy-up code.

The testing code of a TestCase instance should be entirely self-contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest TestCase subclass will simply override the runTest() method in order to perform specific testing code:

```
import unittest
class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
```

Note that in order to test something, we use one of the assert*() methods provided by the TestCase base class. If the test fails, an exception will be raised, and unittest will identify the test case as a failure. Any other exceptions will be treated as errors. This helps you identify where the problem is: failures are caused by incorrect results - a 5 where you expected a 6. Errors are caused by incorrect code - e.g., a TypeError caused by an incorrect function call.

The way to run a test case will be described later. For now, note that to construct an instance of such a test case, we call its constructor without arguments:

testCase = DefaultWidgetSizeTestCase()

Now, such test cases can be numerous, and their set-up can be repetitive. In the case, constructing a Widget in each of 100 Widget test case **subclasses** would mean unsightly duplication.

Keyword

Subclass
"derived class", heir class, or child class is a modular, derivative class that inherits one or more language entities from one or more other classes (called superclass, base classes, or parent classes).

Luckily, we can factor out such set-up code by implementing a method called setUp(), which the testing framework will automatically call for us when we run the test:

```
import unittest
class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
class Default WidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')
class Widget Resize Test Case (Simple WidgetTest Case):
    def runTest(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

If the setUp() method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the runTest() method will not be executed.

Similarly, we can provide a tearDown() method that tidies up after the runTest() method has been run:

```
import unittest
class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
    def tearDown(self):
        self.widget.dispose()
        self.widget = None
```

If setUp() succeeded, the tearDown() method will be run whether runTest() succeeded or not.

Such a working environment for the testing code is called a fixture.

Often, many small test cases will use the same fixture. In this case, we would end up subclassing SimpleWidgetTestCase into many small one-method classes such as DefaultWidgetSizeTestCase. This is time-consuming and discouraging, so in the same vein as JUnit, unittest provides a simpler mechanism:

```
import unittest
class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
    def tearDown(self):
        self.widget.dispose()
        self.widget = None
    def test_default_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')
    def test_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Here we have not provided a runTest() method, but have instead provided two different test methods. Class instances will now each run one of the test_*() methods, with self.widget created and destroyed separately for each instance. When creating an instance we must specify the test method it is to run. We do this by passing the method name in the constructor:

```
defaultSizeTestCase = WidgetTestCase('test_default_size')
resizeTestCase = WidgetTestCase('test_resize')
```

Test case instances are grouped together according to the features they test. unittest provides a mechanism for this: the test suite, represented by unittest's TestSuite class:

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase('test_default_size'))
widgetTestSuite.addTest(WidgetTestCase('test_resize'))
```

For the ease of running tests, as we will see later, it is a good idea to provide in each test module a callable object that returns a pre-built test suite:

```
def suite ():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

or even:

```
def suite ():
    tests = ['test_default_size', 'test_resize']
    return unittest.TestSuite(map(WidgetTestCase, tests))
```

Since it is a common pattern to create a TestCase subclass with many similarly named test functions, unittest provides a TestLoader class that can be used to automate the process of creating a test suite and populating it with individual tests. For example,

```
suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)
```

will create a test suite that will run WidgetTestCase.test_default_size() and WidgetTestCase.test_resize. TestLoader uses the 'test' method name prefix to identify test methods automatically.

Often it is desirable to group suites of test cases together, so as to run tests for the whole system at once. This is easy, since TestSuite instances can be added to a TestSuite just as TestCase instances can be added to a TestSuite:

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite([suite1, suite2])
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as widget.py), but there are several advantages to placing the test code in a separate module, such as test_widget.py:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without

a good reason.

- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

4.2.5 Re-using old test code

Some users will find that they have existing test code that they would like to run from unittest, without converting every old test function to a TestCase subclass.

For this reason, unittest provides a FunctionTestCase class. This subclass of TestCase can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows:

```
testcase = unittest.FunctionTestCase(testSomething)
```

If there are additional set-up and tear-down methods that should be called as part of the test case's operation, they can also be provided like so:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

To make migrating existing test suites easier, unittest supports tests raising AssertionError to indicate test failure. However, it is recommended that you use the explicit TestCase.fail*() and TestCase.assert*() methods instead, as future versions of unittest may treat AssertionError differently.

Note: Even though FunctionTestCase can be used to quickly convert an existing test base over to a unittest-based system, this approach is not recommended. Taking the time to set up proper TestCase subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the doctest module. If so, doctest provides a DocTestSuite class that can automatically build unittest. TestSuite instances from the existing doctest-based tests.

4.2.6 Skipping tests and expected failures

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an "expected failure," a test that is broken and will fail, but shouldn't be counted as a failure on a TestResult.

Skipping a test is simply a matter of using the skip() decorator or one of its conditional variants.

Basic skipping looks like this:

```
class MyTestCase(unittest.TestCase):
    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")
    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
      # Tests that work for only a certain version of the library.
        pass
      @unittest.skipUnless(sys.platform.startswith("win"),
"requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

This is the output of running the example in verbose mode:

```
test_format (__main__.MyTestCase) ... skipped 'not supported
in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating
skipping'
test_windows_support (__main__.MyTestCase) ... skipped
'requires Windows'
----------------------------------------------------------------------
Ran 3 tests in 0.005s

    OK (skipped=3)
    Classes can be skipped just like methods:
    @unittest.skip("showing class skipping")
```

Did You Know?

The computer algebra system AXIOM (1973) has a similar construct that processes streams, but the first use of the term "comprehension" for such constructs was in Rod Burstall and John Darlington's description of their functional programming language NPL from 1977.

```
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the expectedFailure() decorator.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls skip() on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

The following decorators implement test skipping and expected failures:

unittest.skip(reason)

Unconditionally skip the decorated test. reason should describe why the test is being skipped.

unittest.skipIf(condition, reason)

Skip the decorated test if condition is true.

unittest.skipUnless(condition, reason)

Skip the decorated test unless condition is true.

unittest.expectedFailure()

Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

exception unittest.SkipTest(reason)

This exception is raised to skip a test.

Usually you can use TestCase.skipTest() or one of the skipping decorators instead of raising this directly.

Skipped tests will not have setUp() or tearDown() run around them. Skipped classes will not have setUpClass() or tearDownClass() run.

4.3 COMPREHENSIONS

Comprehensions are constructs that allow sequences to be built from other sequences. Types of comprehensions are supported in both Python 2 and Python 3:

- list comprehensions
- dictionary comprehensions
- set comprehensions
- generator comprehensions

We will discuss them one by one. Once you get the hang of using list comprehensions then you can use any of them easily.

4.3.1 List Comprehensions

List comprehensions provide a short and concise way to create lists. It consists of square brackets containing an expression followed by a for clause, then zero or more for or if clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists. The result would be a new list made after the evaluation of the expression in context of the if and for clauses.

Blueprint

variable = [out_exp for out_exp in input_list if out_exp == 2]

Here is a short example:

```
multiples = [i for i in range(30) if i % 3 == 0]
print(multiples)
# Output: [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

This can be really useful to make lists quickly. It is even preferred by some instead of the filter function. List comprehensions really shine when you want to supply a list to a method or function to make a new list by appending to it in each iteration of the for loop.

For instance you would usually do something like this:

```
squared = []
for x in range(10):
    squared.append(x**2)
```

You can simplify it using list comprehensions. For example:

```
squared = [x**2 for x in range(10)]
```

4.3.2 Dict Comprehensions

They are used in a similar way. Here is an example which I found recently:

```
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}
mcase_frequency = {
    k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
    for k in mcase.keys()
}
# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```

In the example we are combining the values of keys which are same but in different typecase. You can also quickly switch keys and values of a dictionary:

```
{v: k for k, v in some_dict.items()}
```

4.3.3 Set Comprehensions

They are also similar to list comprehensions. The only difference is that they use braces {}. Here is an example:

```
squared = {x**2 for x in [1, 1, 2]}
print(squared)
# Output: {1, 4}
```

4.3.4 Generator Comprehensions

They are also similar to list comprehensions. The only difference is that they don't allocate memory for the whole list but generate one item at a time, thus more memory effecient.

```
multiples_gen = (i for i in range(30) if i % 3 == 0)
print(multiples_gen)
# Output: <generator object <genexpr> at 0x7fdaa8e407d8>
for x in multiples_gen:
  print(x)
  # Outputs numbers
```

CASE STUDY

REAL-WORLD PYTHON USE CASES AND APPLICATIONS

Everyone in the web development community knows that Python applications are now becoming mainstream. The programming language is now one of the most popular ever and continues to mark its presence in different industries.

The plethora of Python programming uses are proof why it is now a dominating programming language amongst developers. From web development to machine learning, the applications of Python are increasing every day.

What makes Python applications so amazing?

Python is a really fascinating programming language. Developers can think that they have done everything they want, but then it offers more. Businesses now realize how much they would profit if they build their application in Python. Here's what makes Python so amazing:-

Clear syntax

Python has a clear and clean syntax which is easily readable. It allows even beginners to work with complex software development projects as the team can coordinate easily on the coding front.

The simple coding syntax facilitates test-driven development for all applications of Python.

Scalable

Companies love Python for its scalability. Some of the companies implementing the uses of Python language include Google, Spotify, Netflix, Instagram, and many more that want scalable applications.

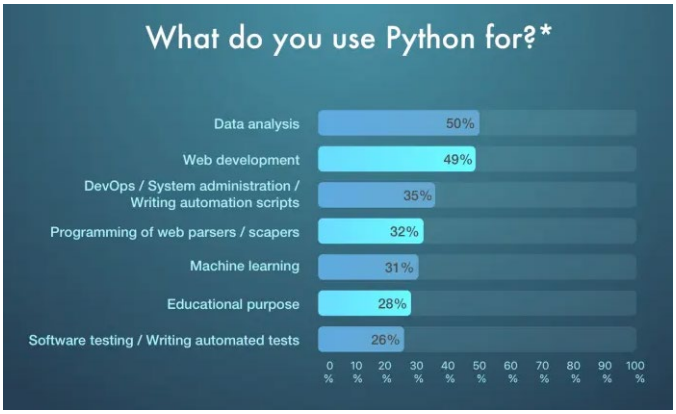
It allows handling a massive amount of traffic with ease.

Versatile

Unlike most programming languages, the practical uses for Python are not limited to just web or mobile development.

It is a popular choice for building web apps, gaming applications, enterprise-grade apps, e-commerce applications, ML and AI applications, and much more.

Why businesses should build applications in Python



If you want to scale your application and expand its customer base, Python programming is an excellent choice for you. It comes with a vast collection of libraries, which allow companies to add a lot of features without reducing the load time.

Python programming uses have made their way in every business. The programming language has a massive community, enabling developers to get all the help they need.

Most of the businesses are hiring Python developers because of the dynamic applications they can design. Python is now the most preferred programming language for developers to learn.

Top 10 uses of Python in the real-world

Python is an excellent tool for businesses for web development. But there's more to Python than meets the eye. It is a powerful programming language for applications of the future.

Here are the top 10 uses of Python in the real world:

Web application development

Unarguably, one of the top practical uses for Python is web application development. Python is now easily the go-to programming language for web applications.

Web development has several uses of Python in the real world. It provides security, convenience, and scalability to applications.

Python has a lot of web development frameworks like Django and Flask, which enable rapid app development. Django's dynamic development capabilities have made Python a useful tool for web applications. The framework is packed with standard

libraries, reducing the development time and providing more time-to-market for the web application.

Data Science

As a highly-demanded skill, Data science is now reaching the top. It is becoming one of the most important areas with applications of Python programming.

Python libraries like Pandas, NumPy, SciPy, and several others help you to work with data and extract valuable information and insights.

Data scientists have to know the uses of Python for extracting and processing data. It allows them to visualize the data through graphs. Matplotlib and Seaborn, both are used for data visualization.

With increasing popularity, Python is the first thing that data scientists have to learn. It is preliminary to working with research and data-based companies.

Artificial Intelligence

Probably the most interesting practical uses for Python is in Artificial Intelligence and Machine Learning. Python is a stable and secure language that can handle the computations required for developing Machine Learning models.

Machine Learning algorithms are one of the important real life uses of Python. Developers can write algorithms easily using the programming language.

Python has an extensive collection of libraries for Machine Learning applications. These include SciPy, Pandas, Keras, TensorFlow, NumPy and many more.

The uses of Python language in AI solutions include advanced computing, data analytics, image recognition, text & data processing and much more that businesses can profit from. If you want to learn more about AI and Python, click here.

Game development

Gaming app development is now a prominent industry, and it has many applications of Python programming. There are libraries which are widely used for interactive game development.

Some of the real world Python projects in the gaming industry include Battlefield 2, Frets on Fire, World of Tanks, etc. These games use Python libraries like PySoy and PyGame for development.

Python allows game developers to build tree-based algorithms which are useful in designing different levels in a game. Games require handling multiple requests at once, and Python is extremely fantastic at that.

Python game app development is one of the top 10 uses of Python in the real world. It offers developers the opportunity to install a 3D game engine that helps in building powerful games and interfaces.

Internet of Things

Another one of the real life uses of Python is in the internet of things. Python programming language enables developers to turn any object into an electronic gadget with the help of Raspberry Pi.

Python is used to create embedded software, allowing high-performance application of Python on smaller objects which can work with the programming language.

With the help of Raspberry Pi, developers can do high-level computations using Python applications. By embedding it, developers can turn normal objects into smart electronics.

In large scale industries, IoT is widely used to track inventory, move machines, and track order processing along with the status of shipment.

Web Scraping

Web scraping of massive amounts of data is becoming useful for companies for extraction valuable customer information and making smart decisions.

This real life application of Python includes scraping large amounts of websites and webpages to extract data for a particular purpose. It could be job listing, price comparison, detailed information and much more.

Selenium, PythonRequest, MechanicalSoup are some of the tools which are used to build web scraping applications of Python programming.

Python has simple code, so it doesn't involve any complexity in writing software that can provide large amounts of data.

Desktop GUI

Python programming language can work with multiple operating systems and has a powerful architecture for building applications.

It has rich text processing tools and a clear syntax, allowing developers to code Desktop GUI applications without any hassle.

PyQT, Kivy, PyGUI are a few toolkits and frameworks offered to get you started with the practical uses of Python for GUI development.

Developers can create highly functional GUIs with Python and reduce the turnaround time for development.

Enterprise applications

Enterprise applications are highly different from regular web applications. They are designed to serve the needs of an organization rather than individual users.

The applications of Python programming in building enterprise-grade applications vary from enterprise to enterprise. It is used mostly for scalability, readability, and its powerful functionality.

Enterprise applications can be complicated as they require a lot of security and database handling capabilities. Python is a robust language that can handle multiple database requests at once.

Odoo and Tryton are some of the enterprise application development tools that enable building apps with Python. Enterprise apps are one of the most significant uses of Python language.

Improve your efficiency with an enterprise application built with Python at an affordable price. **Get Free Estimate Here.**

Image recognition and text processing

Applications built with Python can also enable companies to identify images from a database of images and also helps in text processing.

With its unique image processing and graphic ensign capabilities, Python allows developers to design 2D and 3D images through different tools.

Inkscape, GIMP, Paint Shop are a few examples that showcase the real life applications of Python for designing graphics and images.

Some of the top 3D animation packages use Python in their programming stack, which includes Blender, Houdini, 3ds Max, Lightwave, and many more.

Education programs

One of the popular Python programming uses is in the development of education programs and online courses. Python is a really beginner-friendly programming language with a simple learning curve and a wide variety of resources.

The syntax of Python is similar to English, which makes it the preferred programming language for beginners. Because of this, education program development at the basic and advanced level is done using Python.

Professionals all around the world use Python for building education programs and training courses based on levels. That is why it is one of the best use cases of Python Development.

Practical uses of Python

Python can handle almost all types of requests, which makes it highly useful for all kinds of development activities. From enterprise apps to gaming, the application of Python now ranges to a wide variety of applications.

Python is becoming a popular tool for building all kinds of applications. At BoTree Technologies, we have an expert's team of Python developers who are there to help you build a Python app.

SUMMARY

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception.
- Unit testing is a software testing method by which individual units of source code are put under various tests to determine whether they are fit for use (Source). It determines and ascertains the quality of your code.
- The developers are expected to write automated test scripts, which ensures that each and every section or a unit meets its design and behaves as expected.
- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs.
- Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause.

KNOWLEDGE CHECK

1. How many except statements can a try-except block have?

a. zero
b. one
c. more than one
d. more than zero

2. When will the else part of try-except-else be executed?

a. always
b. when an exception occurs
c. when no exception occurs
d. when an exception occurs in to except block

3. Can one block of except statements handle multiple exception?

a. yes, like except TypeError, SyntaxError [,...].
b. yes, like except [TypeError, SyntaxError].
c. no
d. none of the mentioned

4. What is the output of the code shown?

```
l=[1,2,3,4,5]
[x&1 for x in l]
```

a. [1, 1, 1, 1, 1]
b. [1, 0, 1, 0, 1]
c. [1, 0, 0, 0, 0]
d. [0, 1, 0, 1, 0]

5. What is the output of the code shown below?

```
l1=[1,2,3]
l2=[4,5,6]
[x*y for x in l1 for y in l2]
```

a. [4, 8, 12, 5, 10, 15, 6, 12, 18]
b. [4, 10, 18]
c. [4, 5, 6, 8, 10, 12, 12, 15, 18]
d. [18, 12, 6, 15, 10, 5, 12, 8, 4]

6. What will be the output of the following Python code?

```
max("what are you")
```

a. error
b. u
c. t
d. y

7. What will be the output of the following Python list comprehension?

[j for i in range(2,8) for j in range(i*2, 50, i)]

a. A list of prime numbers up to 50
b. A list of numbers divisible by 2, up to 50
c. A list of non-prime numbers, up to 50
d. Error

8. What will be the output of the following Python code?

l=[2, 3, [4, 5]] l2=l.copy() l2[0]=88 l l2

a. [88, 2, 3, [4, 5]] [88, 2, 3, [4, 5]]
b. [2, 3, [4, 5]] [88, 2, 3, [4, 5]
c. [88, 2, 3, [4, 5]] [2, 3, [4, 5]]
d. [2, 3, [4, 5]] [2, 3, [4, 5]]

REVIEW QUESTIONS

1. Determine the user defined exception in python.
2. What are handling exceptions in Python?
3. What is Unit Testing?
4. Designing a test case for Python Testing using PyUnit.
5. The correct expansion of list_1 = [expr(i) for i in list_0 if func(i)]?

Check Your Result

1. (d) 2. (c) 3. (a) 4. (b) 5. (c)
6. (d) 7. (c) 8. (b)

REFERENCES

1. Batista, Facundo (17 October 2003). "PEP 327 – Decimal Data Type". Python Enhancement Proposals. Python Software Foundation. Archived from the original on 4 June 2020. Retrieved 24 November 2008.
2. Borderies, Olivier (24 January 2019). "Pythran: Python at C++ speed !". Medium. Archived from the original on 25 March 2020. Retrieved 25 March 2020.
3. Francisco, Thomas Claburn in San. "Google's Grumpy code makes Python Go". www.theregister.com. Archived from the original on 7 March 2021. Retrieved 20 January 2021.
4. Murri, Riccardo (2013). Performance of Python runtimes on a non-numeric scientific code. European Conference on Python in Science (EuroSciPy). arXiv:1404.6388.
5. Yegulalp, Serdar (29 October 2020). "Pyston returns from the dead to speed Python". InfoWorld. Archived from the original on 27 January 2021. Retrieved 26 January 2021.

CHAPTER

5

OBJECT ORIENTED PROGRAMMING

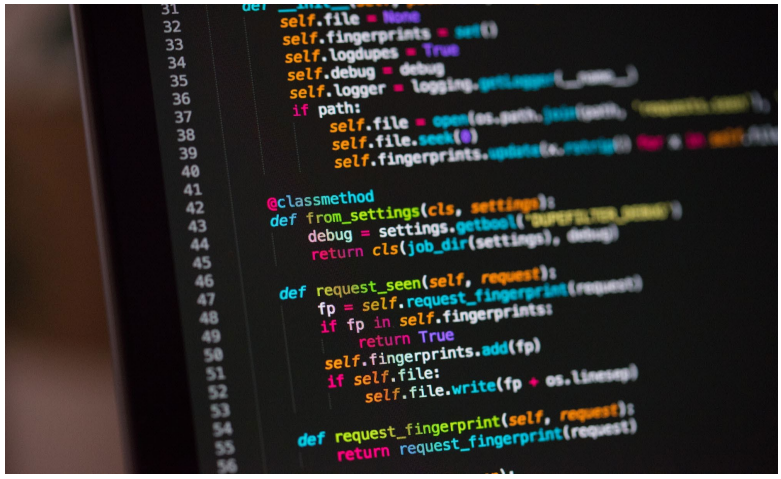
"Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches."

–Paul Graham

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Examine the introduction of OOPS in python.
2. Define the various types of methods of OOPS



INTRODUCTION

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects. Conceptually, objects are

like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line a system component processes some material, ultimately transforming raw material into a finished product.

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed −

- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like

inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

5.1 INTRODUCTION OF OOPS IN PYTHON

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Another common programming paradigm is procedural programming which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, which flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of the object-oriented programming paradigm, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

5.1.1 Classes in Python

Focusing first on the data, each thing or object is an instance of some class. The primitive data structures available in Python, like numbers, strings, and lists are designed to represent simple things like the cost of something, the name of a poem, and your favorite colors, respectively. What if you wanted to represent something much more complicated?

For example, let's say you wanted to track a number of different animals. If you used a list, the first element could be the animal's name while the second element could represent its age. How would you know which element is supposed to be which? What if you had 100 different animals? Are you certain each animal has both a name and an age, and so forth? What if you wanted to add other properties to these animals? This lacks organization, and it's the exact need for classes. Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an Animal() class to track properties about the Animal like the name and age. It's important to note that a class

just provides structure—it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. The Animal() class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal's name or age is. It may help to think of a class as an idea for how something should be defined.

5.1.2 Python Objects (Instances)

While the class is the blueprint, an *instance* is a copy of the class with *actual* values, literally an object belonging to a specific class. It's not an idea anymore; it's an actual animal, like a dog named Roger who's eight years old.

Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains actual information relevant to you.

You can fill out multiple copies to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, we must first specify what is needed by defining a class.

How to Define a Class in Python

Defining a class is simple in Python:

```
class Dog:
    pass
```

You start with the class keyword to indicate that you are creating a class, then you add the name of the class (using CamelCase notation, starting with a capital letter.)

Also, we used the Python keyword pass here. This is very often used as a place holder where code will eventually go. It allows us to run this code without throwing an error.

The above code is correct on Python 3. On Python 2.x ("legacy Python") you'd use a slightly different class definition:

```
# Python 2.x Class Definition:
class Dog(object):
    pass
```

The (object) part in parentheses specifies the parent class that you are inheriting from. In Python 3 this is no longer necessary because it is the implicit default.

Instance Attributes

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the __init__() method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). This method must have at least one argument as well as the self variable, which refers to the object itself (e.g., Dog).

:class Dog

Initializer / Instance Attributes #

:(def __init__(self, name, age

self.name = name

self.age = age

In the case of our Dog() class, each dog has a specific name and age, which is obviously important to know for when you start actually creating different dogs. Remember: the class is just for defining the Dog, not actually creating *instances* of .individual dogs with specific names and ages; we'll get to that shortly

Similarly, the self variable is also an instance of the class. Since instances of a class have varying values we could state Dog.name = name rather than self.name = name. But since not all dogs share the same name, we need to be able to assign different values to different instances. Hence the need for the special self variable, which will .help to keep track of individual instances of each class

You will never have to call the __init__() method; it gets called automatically when .you create a new 'Dog' instance

Class Attributes

While instance attributes are specific to each object, class attributes are the same for all instances—which in this case is *all* dogs.

class Dog:

```
    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

So while each dog has a unique name and age, every dog will be a mammal.

Let's create some dogs…

5.1.3 Instantiating Objects

Instantiating is a fancy term for creating a new, unique instance of a class.

For example:

```
>>> class Dog:
...     pass
...
>>> Dog()
<__main__.Dog object at 0x1004ccc50>
>>> Dog()
<__main__.Dog object at 0x1004ccc90>
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

We started by defining a new Dog() class, then created two new dogs, each assigned to different objects. So, to create an instance of a class, you use the class name, followed by parentheses. Then to demonstrate that each instance is actually different, we instantiated two more dogs, assigning each to a variable, then tested if those variables are equal.

What do you think the type of a class instance is?

```
>>> class Dog:
...     pass
...
>>> a = Dog()
>>> type(a)
<class '__main__.Dog'>
```

Let's look at a slightly more complex example…

```
class Dog:

    # Class Attribute
    species = 'mammal'
```

```
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age


# Instantiate the Dog object
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 6)

# Access the instance attributes
print("{} is {} and {} is {}.".format(
    philo.name, philo.age, mikey.name, mikey.age))

# Is Philo a mammal?
if philo.species == "mammal":
    print("{0} is a {1}!".format(philo.name, philo.species))
```

Save this as *dog_class.py*, then run the program. You should see:

```
Philo is 5 and Mikey is 6.
Philo is a mammal!
```

What's Going On?

We created a new instance of the Dog() class and assigned it to the variable philo. We then passed it two arguments, "Philo" and 5, which represent that dog's name and age, respectively.

These attributes are passed to the __init__ method, which gets called any time you create a new instance, attaching the name and age to the object. You might be wondering why we didn't have to pass in the self argument.

This is Python magic; when you create a new instance of the class, Python automatically determines what self is (a Dog in this case) and passes it to the __init__ method.

5.1.4 Instance Methods

Instance methods are defined inside a class and are used to get the contents of an **instance**. They can also be used to perform operations with the attributes of our objects. Like the __init__ method, the first argument is always self:

Keyword

Instance is a concrete occurrence of any object, existing usually during the runtime of a computer program. Formally, it is synonymous with "object" as they are each a particular value (realization), and these may be called an instance object; "instance" emphasizes the distinct identity of the object.

```
    class Dog:

Class Attribute #
'species = 'mammal

Initializer / Instance Attributes #
:(def __init__(self, name, age
self.name = name
self.age = age

instance method #
:(def description(self
(return "{} is {} years old".format(self.name, self.age

instance method #
:(def speak(self, sound
(return "{} says {}".format(self.name, sound

Instantiate the Dog object #
        mikey = Dog("Mikey", 6)


        # call our instance methods

        print(mikey.description())

        print(mikey.speak("Gruff Gruff"))
```

Save this as *dog_instance_methods.py*, then run it:

```
    Mikey is 6 years old
```

```
Mikey says Gruff Gruff
```

In the latter method, speak(), we are defining behavior. What other behaviors could you assign to a dog? Look back to the beginning paragraph to see some example behaviors for other objects.

Modifying Attributes

You can change the value of attributes based on some behavior:

```
>>> class Email:
...     def __init__(self):
...         self.is_sent = False
...     def send_email(self):
...         self.is_sent = True
...
>>> my_email = Email()
>>> my_email.is_sent
False
>>> my_email.send_email()
>>> my_email.is_sent
True
```

Here, we added a method to send an email, which updates the is_sent variable to True.

5.1.5 Python Object Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called *child classes*, and the classes that child classes are derived from are called *parent classes*.

It's important to note that child classes override *or* extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an object, which generally all other classes inherit as their parent.

When you define a new class, Python 3 it implicitly uses object as the parent class. So the following two definitions are equivalent:

```
class Dog(object):
    pass
```

```
# In Python 3, this is the same as:


class Dog:
    pass
```

In Python 2.x there's a distinction between *new-style* and *old-style* classes. I won't go into detail here, but you'll generally want to specify object as the parent class to ensure you're definint a new-style class if you're writing Python 2 OOP code.

Dog Park Example

Let's pretend that we're at a dog park. There are multiple Dog objects engaging in Dog behaviors, each with different attributes. In regular-speak that means some dogs are running, while some are stretching and some are just watching other dogs. Furthermore, each dog has been named by its owner and, since each dog is living and breathing, each ages.

What's another way to differentiate one dog from another? How about the dog's breed:

```
>>> class Dog:
...     def __init__(self, breed):
...         self.breed = breed
...
>>> spencer = Dog("German Shepard")
>>> spencer.breed
'German Shepard'
>>> sara = Dog("Boston Terrier")
>>> sara.breed
'Boston Terrier'
```

Each breed of dog has slightly different behaviors. To take these into account, let's create separate classes for each breed. These are child classes of the parent Dog class.

Extending the Functionality of a Parent Class

Create a new file called *dog_inheritance.py*:

```
# Parent class
class Dog:
```

```
    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class
jim = Bulldog("Jim", 12)
print(jim.description())
```

```
# Child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))
```

Read the comments aloud as you work through this program to help you understand what happening, then before you run the program, sees if you can predict the expected output.

You should see:

```
Jim is 12 years old
Jim runs slowly
```

We haven't added any special attributes or methods to differentiate a RussellTerrier from a Bulldog, but since they're now two different classes, we could for instance give them different class attributes defining their respective speeds.

Parent vs. Child Classes

The isinstance() function is used to determine if an instance is also an instance of a certain parent class.

Save this as *dog_isinstance.py*:

```
# Parent class
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
```

```
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog() class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog() class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class
jim = Bulldog("Jim", 12)
print(jim.description())

# Child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))

# Is jim an instance of Dog()?
print(isinstance(jim, Dog))

# Is julie an instance of Dog()?
julie = Dog("Julie", 100)
print(isinstance(julie, Dog))

# Is johnny walker an instance of Bulldog()
```

```
johnnywalker = RussellTerrier("Johnny Walker", 4)
print(isinstance(johnnywalker, Bulldog))

# Is julie and instance of jim?
print(isinstance(julie, jim))
```

Output:

```
('Jim', 12)
Jim runs slowly
True
True
False
Traceback (most recent call last):
  File "dog_isinstance.py", line 50, in <module>
    print(isinstance(julie, jim))
TypeError: isinstance() arg 2 must be a class, type, or tuple of classes and types
```

Make sense? Both jim and julie are instances of the Dog() class, while johnnywalker is not an instance of the Bulldog() class. Then as a sanity check, we tested if julie is an instance of jim, which is impossible since jim is an instance of a class rather than a class itself—hence the reason for the **TypeError**.

Keyword

TypeError is an unintended condition which might manifest in multiple stages of a program's development. Thus a facility for detection of the error is needed in the type system.

Overriding the Functionality of a Parent Class

Remember that child classes can also override attributes and behaviors from the parent class. For examples:>>> class Dog:

```
...     species = 'mammal'
...
>>> class SomeBreed(Dog):
...     pass
...
>>> class SomeOtherBreed(Dog):
...     species = 'reptile'
...
>>> frank = SomeBreed()
```

```
>>> frank.species
'mammal'
>>> beans = SomeOtherBreed()
>>> beans.species
'reptile'
```

The SomeBreed() class inherits the species from the parent class, while the SomeOtherBreed() class overrides the species, setting it to reptile.

5.2 METHODS OF OOPS

Python has been an object-oriented language from day one. Because of this, creating and using classes and objects are downright easy. If you don't have any experience with object-oriented (OO) programming.

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Creating Methods in Python

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)
    def dance(self):
        return "{} is now dancing".format(self.name)
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

When we run program, the output will be:

Blu sings 'Happy'

Blu is now dancing

In the above program, we define two methods i.e sing() and dance(). These are called instance method because they are called on an instance object i.e blu.

5.2.1 Inheritance

Keyword

Base class is the parent class of a derived class. Classes may be used to create other classes. A class that is used to create (or derive) another class is called the base class.

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a **base class** (or parent class).

Python Inheritance Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):
```

```
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

When we run this program, the output will be:

- Bird is ready
- Penguin is ready
- Penguin
- Swim faster
- Run faster

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim()method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method.

Additionally, we use super() function before __init__() method. This is because we want to pull the content of __init__() method from the parent class into the child class.

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
```

```
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides, *n* and magnitude of each side as a list, *sides*.

Method inputSides() takes in magnitude of each side and similarly, dispSides() will display these properly.

A triangle is a polygon with 3 sides. So, we can created a class called Triangle which inherits from Polygon. This makes all the attributes available in class Polygon readily available in Triangle. We don't need to define them again (code re-usability). Triangle is defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class Triangle has a new method findArea() to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
```

```
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

We can see that, even though we did not define methods like inputSides() or dispSides() for class Triangle, we were able to use them.

If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

5.2.2 Encapsulation

Encapsulation is the packing of *data* and *functions operating on that data* into a single component and restricting the access to some of the object's components. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables etc.

Difference between Abstraction and Encapsulation: Abstraction is a mechanism which represent the essential features without including implementation details.

Encapsulation: — Information hiding.

Abstraction: — Implementation hiding.

Python follows the philosophy of we're all adults here with respect to hiding attributes and methods; i.e. you should trust the other programmers who will use your classes. Use plain attributes whenever possible.

You might be tempted to use getter and setter methods instead of attributes, but the only reason to use getters and setters is so you can change the implementation later if you need to. However, Python 2.2 and later allows you to do this with properties:

Protected members

Protected member is accessible only from within the class and it's subclasses. How to accomplish this in Python? The answer is — by convention. By prefixing the name of

your member with a single underscore, you're telling others "don't touch this, unless you're a subclass".

Private members

But there is a method in Python to define Private: Add "__" (double underscore) in front of the variable and function .name can hide them when accessing them from out of class

Python doesn't have real private methods, so one underline in the beginning of a method or attribute means you shouldn't access this method. But this is just convention. And can still .access the variables with single underscore

Also when using double underscore (__).we can still access the private variables

An example of accessing private member data.(Using name mangling)

```
class Person:

def __init__(self):

self.name = 'Manjula'

self.__lastname = 'Dube'

def PrintName(self):

return self.name +' ' + self.__lastname


#Outside class

P = Person()

print(P.name)

print(P.PrintName())

print(P.__lastname)

#AttributeError: 'Person' object has no attribute '__lastname'
```

ccess public variable out of class, succeed

Access private variable our of class, fail

Access public function but this function access **Private variable** __B successfully since they are in the same class.

An example of accessing private member data.(Using name mangling technique)

```
class SeeMee:
  def youcanseeme(self):
    return 'you can see me'

  def __youcannotseeme(self):
    return 'you cannot see me'

#Outside class
Check = SeeMee()
print(Check.youcanseeme())
# you can see me
print(Check.__youcannotseeme())
```

#AttributeError: 'SeeMee' object has no attribute '__youcannotseeme'

If you need to access the private member function

```
class SeeMee:
  def youcanseeme(self):
    return 'you can see me'
  def __youcannotseeme(self):
    return 'you cannot see me'
#Outside class
Check = SeeMee()
print(Check.youcanseeme())
print(Check._SeeMee__youcannotseeme())
#Changing the name causes it to access the function
```

You can still call the method using its mangled name, so this feature doesn't provide much protection.

Remember

The __init__ method is a constructor and runs as soon as an object of a class is instantiated. Its aim is to initialize the object

Keyword

Private variables, are variables that are visible only to the class to which they belong.

You should know the following for accessing private members and private functions:

- When you write to an attribute of an object, that does not exist, the python system will normally not complain, but just create a new attribute.
- Private attributes are not protected by the Python system. That is by design decision.
- Private attributes will be masked. The reason is, that there should be no clashes in the inheritance chain. The masking is done by some implicit renaming. Private attributes will have the *real* name

"__<className>_<attributeName>"

With that name, it can be accessed from outside. When accessed from inside the class, the name will be automatically changed correctly.

Data Encapsulation in Python

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single " _ " or double " __".

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
```

```
c.setMaxPrice(1000)
c.sell()
```

When we run this program, the output will be:

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a class Computer. We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

5.2.3 Polymorphism

Polymorphism means that different types respond to the same function. Polymorphism is very useful as it makes programming more intuitive and therefore easier. Polymorphism is a fancy word that just means the same function is defined on objects of different types. Python provides protocols which is polymorphism under the hood. These implement consistent behavior for built in objects of different type.

Protocols

When we introspect an object we have a lot of attributes that take this format: *__names__*. This section will make many of those clear.

Everything is an object and all actions ultimately mean calling functions defined on objects.

Protocols are polymorphic functions that are embbedded into python. Most importantly the interpreter is aware of them.

Protocols enable:

- consistency - programmers can rely on intuition
- special syntax - interpreter translates nice syntax to functions on objects.
- We will look at two protocols: *__contains__* and *__iter__*

__add__

x + y resolves to *x.__add__(y)*

```
>>> 1 + 2
3
>>> one = 1
>>> one.__add__(2)
```

```
3
>>> '1' + '2'
'12'
>>> '1'.__add__('2')
'12'
```

Any object that implements the *__add__* function will work with the *<object> + x* syntax.

Keyword

Parameter is any characteristic that can help in defining or classifying a particular system. That is, a parameter is an element of a system that is useful, or critical, when identifying the system, or when evaluating its performance, status, condition, etc.

__contains__

__contains__ is the built in protocol for membership.

x in y resolves to *y.__contains__(x)*

When the interpreter encounters *'b' in ['a', 'b']* it knows to look for the *__contains__* function on the object to the right of *in* and pass it the object to the left of *in* as **parameter**.

A list object has that function defined and the interpreter then executes the corresponding code block.

All data structures have the concept of membership defined:

```
>>> 'b' in ['a', 'b']
True
>>> 'b' in ('a', 'b')
True
>>> 'b' in {'a': 1, 'b': 2}
True
>>> 'b' in {'a', 'b'}
True
```

Demonstrating *__contains__*:

```
>>> ['a', 'b'].__contains__('b')
True
>>> ('a', 'b').__contains__('b')
True
>>> {'a': 1, 'b': 2}.__contains__('b')
True
>>> {'a', 'b'}.__contains__('b')
True
```

Any object that implements the *__contains__* function will work with the *x in <object>* syntax.

__iter__

__iter__ is how iteration is implemented in Python. This protocol is a bit more involved than the protocols.

Taking this code:

```
>>> number = [1, 2]
>>> for i in [1, 2]:
...     print(i)
...
1
2
```

Roughly here is the sequence of events: * interpreter calls *__iter__* on the list object, * an object of type iterator is returned. * interpreter then calls *__next__* repeatedly on the iterator * interpreter actions the code in the for loop * interpreter interrupts the loop if a *StopIteration* Exception occurs.

To illustrate:

```
>>> itr_obj = [1, 2].__iter__()
>>> type(itr_obj)
<class 'list_iterator'>
>>> itr_obj.__next__()
1
>>> itr_obj.__next__()
2
>>> itr_obj.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Any object that implements the *__iter__* function will work with the *for x in <object>: ...* syntax.

Exercise

Boolean Operators

Using introspection functions, which protocol functions do the following syntax resolve to:

- 3 > 2
- 3 < 2
- 3 <= 2
- 3 >= 2

String representations

What function gets called when we get results in the interpreter? Is it the same that gets called when we type *print(x)*?

len() implementation

len() works on many object types:

```
>>> len('hi')
2
>>> len([1, 2])
2
```

Which protocol function is called by the function *len* on the object it is passed?

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types). Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

Using Polymorphism in Python

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")
```

```
class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

When we run above program, the output will be:

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes Parrot and Penguin. Each of them have common method fly() method. However, their functions are different. To allow polymorphism, we created common interface i.e flying_test() function that can take any object. Then, we passed the objects blu and peggy in the flying_test() function, it ran effectively.

5.2.4 Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things, so that the name captures the core of what

Keyword

Modularity is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use.

a function or a whole program does. As we consider the wide set of things in the world that we would like to represent in our programs, we find that most of them have compound structure. A date has a year, a month, and a day; a geographic position has a latitude and a longitude. To represent positions, we would like our programming language to have the capacity to "glue together" a latitude and longitude to form a pair --- a *compound data* value --- that our programs could manipulate in a way that would be consistent with the fact that we regard a position as a single conceptual unit, which has two parts.

The use of compound data also enables us to increase the **modularity** of the programs. If we can manipulate geographic positions directly as objects in their own right, then we can separate the part of our program that deals with values per se from the details of how those values may be represented. The general technique of isolating the parts of a program that deal with how data are represented from the parts of a program that deal with how those data are manipulated is a powerful design methodology called *data abstraction*. Data abstraction makes programs much easier to design, maintain, and modify.

Data abstraction is similar in character to functional abstraction. When we create a functional abstraction, the details of how a function is implemented can be suppressed, and the particular function itself can be replaced by any other function with the same overall behavior. In other words, we can make an abstraction that separates the way the function is used from the details of how the function is implemented. Analogously, data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed.

The basic idea of data abstraction is to structure programs so that they operate on abstract data. That is, our programs should use data in such a way as to make as few assumptions about the data as possible. At the same time, a concrete data representation is defined, independently of the programs that use the data. The interface between these two parts of our system will be a set of functions, called selectors and constructors, that implement the abstract data in terms of the concrete representation. To illustrate this technique, we will consider how to design a set of functions for manipulating

rational numbers. As you read the next few sections, keep in mind that most Python code written today uses very high-level abstract data types that are built into the language, like classes, dictionaries, and lists. Since we're building up an understanding of how these abstractions work, we can't use them yet ourselves. As a consequence, we will write some code that isn't Pythonic --- it's not necessarily the typical way to implement our ideas in the language. What we write is instructive, however, because it demonstrates how these abstractions can be constructed! Remember that computer science isn't just about learning to use programming languages, but also learning how they work.

Example: Arithmetic on Rational Numbers

Recall that a rational number is a ratio of integers, and rational numbers constitute an important sub-class of real numbers. A rational number like 1/3 or 17/29 is typically written as:

<numerator>/<denominator>

where both the <numerator> and <denominator> are placeholders for integer values. Both parts are needed to exactly characterize the value of the rational number.

Rational numbers are important in computer science because they, like integers, can be represented exactly. Irrational numbers (like pi or e or sqrt(2)) are instead approximated using a finite binary expansion. Thus, working with rational numbers should, in principle, allow us to avoid approximation errors in our arithmetic.

However, as soon as we actually divide the numerator by the denominator, we can be left with a truncated decimal approximation (a float).

```
>>> 1/3
0.3333333333333333
```

and the problems with this approximation appear when we start to conduct tests:

```
>>> 1/3 == 0.33333333333333300000  # Beware of approximations
True
```

How computers approximate real numbers with finite-length decimal expansions is a topic for another class. The important idea here is that by representing rational numbers as ratios of integers, we avoid the approximation problem entirely. Hence, we would like to keep the numerator and denominator separate for the sake of precision, but treat them as a single unit.

We know from using functional abstractions that we can start programming productively before we have an implementation of some parts of our program. Let us begin by assuming that we already have a way of constructing a rational number from a numerator and a denominator. We also assume that, given a rational number,

we have a way of extracting (or selecting) its numerator and its denominator. Let us further assume that the constructor and selectors are available as the following three functions:

- make_rat(n, d) returns the rational number with numerator n and denominator d.
- numer(x) returns the numerator of the rational number x.
- denom(x) returns the denominator of the rational number x.

We are using here a powerful strategy of synthesis: *wishful thinking*. We haven't yet said how a rational number is represented, or how the functions numer, denom, and make_rat should be implemented. Even so, if we did have these three functions, we could then add, multiply, and test equality of rational numbers by calling them:

```
>>> def add_rat(x, y):
        nx, dx = numer(x), denom(x)
        ny, dy = numer(y), denom(y)
        return make_rat(nx * dy + ny * dx, dx * dy)
>>> def mul_rat(x, y):
        return make_rat(numer(x) * numer(y), denom(x) * denom(y))
>>> def eq_rat(x, y):
        return numer(x) * denom(y) == numer(y) * denom(x)
```

Now we have the operations on rational numbers defined in terms of the selector functions numer and denom, and the constructor function make_rat, but we haven't yet defined these functions. What we need is some way to glue together a numerator and a denominator into a unit.

Tuples

To enable us to implement the concrete level of our data abstraction, Python provides a compound structure called a tuple, which can be constructed by separating values by commas. Although not strictly required, parentheses almost always surround tuples.

```
>>> (1, 2)
(1, 2)
```

The elements of a tuple can be unpacked in two ways. The first way is via our familiar method of multiple assignment.

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
```

```
>>> x
1
>>> y
2
```

In fact, multiple assignment has been creating and unpacking tuples all along.

A second method for accessing the elements in a tuple is by the indexing operator, written as square brackets.

```
>>> pair[0]
1
>>> pair[1]
2
```

Tuples in Python (and sequences in most other programming languages) are 0-indexed, meaning that the index 0 picks out the first element, index 1 picks out the second, and so on. One intuition that underlies this indexing convention is that the index represents how far an element is offset from the beginning of the tuple.

The equivalent function for the element selection operator is called getitem, and it also uses 0-indexed positions to select elements from a tuple.

```
>>> from operator import getitem
>>> getitem(pair, 0)
1
```

Tuples are native types, which means that there are built-in Python **operators** to manipulate them. We'll return to the full properties of tuples shortly. At present, we are only interested in how tuples can serve as the glue that implements abstract data types.

Representing Rational Numbers. Tuples offer a natural way to implement rational numbers as a pair of two integers: a numerator and a denominator. We can implement our constructor and selector functions for rational numbers by manipulating 2-element tuples.

```
>>> def make_rat(n, d):
        return (n, d)
>>> def numer(x):
```

Keyword

Operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

```
        return getitem(x, 0)
>>> def denom(x):
        return getitem(x, 1)
```

A function for printing rational numbers completes our implementation of this abstract data type.

```
>>> def str_rat(x):
        """Return a string 'n/d' for numerator n and denominator d."""
        return '{0}/{1}'.format(numer(x), denom(x))
```

Together with the arithmetic operations we defined earlier, we can manipulate rational numbers with the functions we have defined.

```
>>> half = make_rat(1, 2)
>>> str_rat(half)
'1/2'
>>> third = make_rat(1, 3)
>>> str_rat(mul_rat(half, third))
'1/6'
>>> str_rat(add_rat(third, third))
'6/9'
```

As the final example shows, our rational-number implementation does not reduce rational numbers to lowest terms. We can remedy this by changing make_rat. If we have a function for computing the greatest common denominator of two integers, we can use it to reduce the numerator and the denominator to lowest terms before constructing the pair. As with many useful tools, such a function already exists in the Python Library.

```
>>> from fractions import gcd
>>> def make_rat(n, d):
        g = gcd(n, d)
        return (n//g, d//g)
```

The double slash operator, //, expresses integer division, which rounds down the fractional part of the result of division. Since we know that g divides both n and d evenly, integer division is exact in this case. Now we have

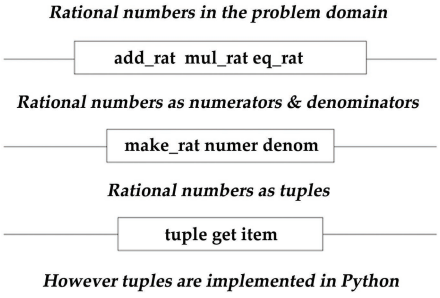
```
>>> str_rat(add_rat(third, third))
'2/3'
```

as desired. This modification was accomplished by changing the constructor without changing any of the functions that implement the actual arithmetic operations.

Abstraction Barriers

Before continuing with more examples of compound data and data abstraction, let us consider some of the issues raised by the rational number example. We defined operations in terms of a constructor make_rat and selectors numer and denom. In general, the underlying idea of data abstraction is to identify for each type of value a basic set of operations in terms of which all manipulations of values of that type will be expressed, and then to use only those operations in manipulating the data.

We can envision the structure of the rational number system as a series of layers.



The horizontal lines represent abstraction barriers that isolate different levels of the system. At each level, the barrier separates the functions (above) that use the data abstraction from the functions (below) that implement the data abstraction. Programs that use rational numbers manipulate them solely in terms of the their arithmetic functions: add_rat, mul_rat, and eq_rat. These, in turn, are implemented solely in terms of the constructor and selectors make_rat, numer, and denom, which themselves are implemented in terms of tuples. The details of how tuples are implemented are irrelevant to the rest of the layers as long as tuples enable the implementation of the selectors and constructor.

At each layer, the functions within the box enforce the abstraction boundary because they are the only functions that depend upon both the representation above them (by their use) and the implementation below them (by their definitions). In this way, abstraction barriers are expressed as sets of functions. Abstraction barriers provide many advantages. One advantage

Remember

The str_rat implementation above uses format strings, which contain placeholders for values. The details of how to use format strings and the format method appear in the formatting strings section of Dive Into Python 3.

is that they makes programs much easier to maintain and to modify. The fewer functions that depend on a particular representation, the fewer changes are required when one wants to change that representation.

The Properties of Data

We began the rational-number implementation by implementing arithmetic operations in terms of three unspecified functions: make_rat, numer, and denom. At that point, we could think of the operations as being defined in terms of data objects --- numerators, denominators, and rational numbers --- whose behavior was specified by the latter three functions.

But what exactly is meant by data? It is not enough to say "whatever is implemented by the given selectors and constructors." We need to guarantee that these functions together specify the right behavior. That is, if we construct a rational number x from integers n and d, then it should be the case that numer(x)/denom(x) is equal to n/d.

In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

This point of view can be applied to other data types as well, such as the two-element tuple that we used in order to implement rational numbers. We never actually said much about what a tuple was, only that the language supplied operators to create and manipulate tuples. We can now describe the behavior conditions of two-element tuples, also called pairs, that are relevant to the problem of representing rational numbers.

In order to implement rational numbers, we needed a form of glue for two integers, which had the following behavior:

- If a pair p was constructed from values x and y, then getitem_pair(p, 0) returns x, and getitem_pair(p, 1) returns y.

We can implement functions make_pair and getitem_pair that fulfill this description just as well as a tuple.

```
>>> def make_pair(x, y):
        """Return a function that behaves like a pair."""
        def dispatch(m):
            if m == 0:
                return x
            elif m == 1:
                return y
```

```
        return dispatch
>>> def getitem_pair(p, i):
        """Return the element at index i of pair p."""
        return p(i)
```

With this implementation, we can create and manipulate pairs.

```
>>> p = make_pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
```

This use of functions corresponds to nothing like our intuitive notion of what data should be. Nevertheless, these functions suffice to represent compound data in our programs.

The subtle point to notice is that the value returned by make_pair is a function called dispatch, which takes an argument m and returns either x or y. Then, getitem_pair calls this function to retrieve the appropriate value.

The point of exhibiting the functional representation of a pair is not that Python actually works this way (tuples are implemented more directly, for efficiency reasons) but that it could work this way. The functional representation, although obscure, is a perfectly adequate way to represent pairs, since it fulfills the only conditions that pairs need to fulfill. This example also demonstrates that the ability to manipulate functions as values automatically provides us the ability to represent compound data.

Did You Know?

Terminology invoking "objects" and "oriented" in the modern sense of object-oriented programming made its first appearance at MIT in the late 1950s and early 1960s. In the environment of the artificial intelligence group, as early as 1960, "object" could refer to identified items (LISP atoms) with properties (attributes) Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in 1966.

SUMMARY

- Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects. Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts.
- Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
- The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.
- Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.
- Instance methods are defined inside a class and are used to get the contents of an instance.
- Inheritance is the process by which one class takes on the attributes and methods of another.
- Python has been an object-oriented language from day one. Because of this, creating and using classes and objects are downright easy. If you don't have any experience with object-oriented (OO) programming.
- Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class).
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables etc.
- Python follows the philosophy of we're all adults here with respect to hiding attributes and methods; i.e. you should trust the other programmers who will use your classes.
- Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

KNOWLEDGE CHECK

1. **Which is private member functions access scope?**
   a. Member functions which can only be used within the class
   b. Member functions which can used outside the class
   c. Member functions which are accessible in derived class
   d. Member functions which can't be accessed inside the class
2. **Which among the following is true?**
   a. The private members can't be accessed by public members of the class
   b. The private members can be accessed by public members of the class
   c. The private members can be accessed only by the private members of the class
   d. The private members can't be accessed by the protected members of the class
3. **Which member can never be accessed by inherited classes?**
   a. Private member function
   b. Public member function
   c. Protected member function
   d. All can be accessed
4. **Which syntax among the following shows that a member is private in a class?**
   a. private: functionName(parameters)
   b. private(functionName(parameters))
   c. private functionName(parameters)
   d. private::functionName(parameters)
5. **How many private member functions are allowed in a class?**
   a. Only 1
   b. Only 7
   c. Only 255
   d. As many as required
6. **Which of the following language was developed as the first purely object programming language?**
   a. SmallTalk
   b. C++
   c. Kotlin
   d. Java

7. Who developed object-oriented programming?

a. Adele Goldberg
b. Dennis Ritchie
c. Alan Kay
d. Andrea Ferro

8. Which of the following is not an OOPS concept?

a. Encapsulation
b. Polymorphism
c. Exception
d. Abstraction

REVIEW QUESTIONS

1. What are the classes and objects (instances) in python?
2. How to instantiating the objects? Explain with suitable example.
3. Discuss on the concept of inheritance.
4. What is the mechanism of encapsulation in OOPS?
5. Write a program to using polymorphism in python.

Check Your Result

1. (a) 2. (b) 3. (a) 4. (c) 5. (d)
6. (a) 7. (c) 8. (c)

REFERENCES

1. Beal, V. (2016). What is Polymorphism? Webopedia Definition. [online] Webopedia. com. Available at: http:// www.webopedia.com/TERM/P/polymorphism.html
2. Derek Coleman, et. al. Object-Oriented Development - The Fusion Method. Prentice-Hall Object-Oriented Series.
3. E. Colbert. The Object-Oriented Software Development Method: a practical approach to object-oriented development. Tri-Ada Proc., New York.
4. Lambert, S. (2012). Quick Tip: The OOP Principle of Encapsulation. [online] Game Development Envato Tuts+. Available at: http://gamedevelopment.tutsplus.com/ tutorials/quick-tip-the-oop-principle-of-encapsulation-- gamedev-2187
5. Lau, Yun-Tung, Ph.D. The Art of Objects: Object-Oriented Design and Architecture. Addison-Wesley, 2001.
6. Obbayi, R. (2016). Compare Structured and Object-Oriented Programming: What Are the Real Differences?. [online] Bright Hub. Available at: http://www.brighthub. com/ internet/web-development/articles/82024.aspx

CHAPTER

6

PYTHON REGULAR EXPRESSION

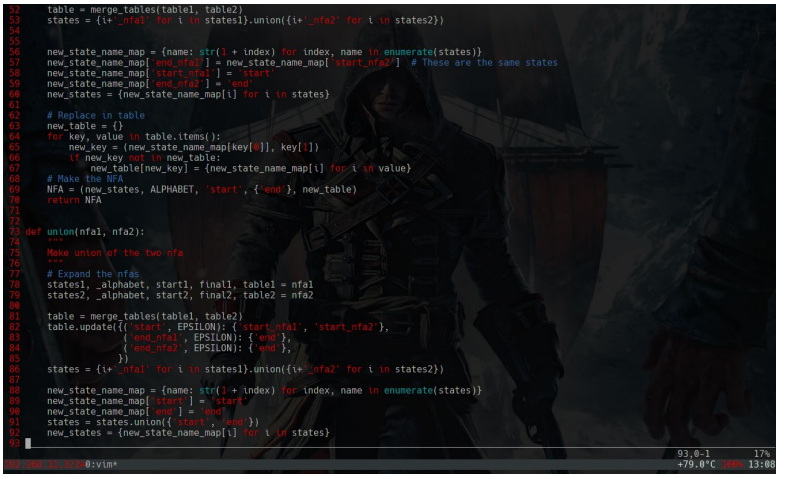
"In this beginner-friendly book, called 'Learn to Program with Minecraft,' you will learn how to do cool things in Minecraft using the Python programming language. No prior programming experience is needed."

–Mark Frauenfelder

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Understand the regex search and match
2. Learn about regular expression modifiers in case of option flags



INTRODUCTION

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made

available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as "Does this string match the pattern?", or "Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn't covered in this document, because it requires that you have a good understanding of the matching engine's internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

Keyword

Constant is a value that cannot be altered by the program during normal execution, i.e., the value is constant.

6.1 REGEX SEARCH AND MATCH

Python is a high level open source scripting language. Python's built-in "re" module provides excellent support for regular expressions, with a modern and complete regex flavor. The only significant features missing from Python's regex syntax are atomic grouping, possessive quantifiers, and Unicode properties.

The first thing to do is to import the regexp module into your script with import re.

Call re.**search**(regex, subject) to apply a regex pattern to a subject string. The function returns None if the matching attempt fails, and a Match object otherwise. Since None evaluates to False, you can easily use re.search() in an if statement. The Match object stores details about the part of

the string matched by the regular expression pattern. You can set regex matching modes by specifying a special **constant** as a third parameter to re.search(). re.I or re.IGNORECASE applies the pattern case insensitively. re.S or re.DOTALL makes the dot match newlines. re.Mor re.MULTILINE makes the caret and dollar match after and before line breaks in the subject string. There is no difference between the single-letter and descriptive options, except for the number of characters you have to type in. To specify more than one option, "or" them together with the | operator: re.search("^a", "abc", re.I | re.M).

By default, Python's regex engine only considers the letters A through Z, the digits 0 through 9, and the underscore as "word characters". Specify the flag re.L or re.LOCALE to make \w match all characters that are considered letters given the current locale settings. Alternatively, you can specify re.U or re.UNICODE to treat all letters from all scripts as word characters. The setting also affects word boundaries.

Do not confuse re.search() with re.**match**(). Both functions do exactly the same, with the important distinction that re.search() will attempt the pattern throughout the string, until it finds a match. re.match() on the other hand, only attempts the pattern at the very start of the string. Basically, re.match("regex", subject) is the same as re.search("\Aregex", subject).

Python 3.4 adds a new re.**fullmatch**() function. This function only returns a Match object if the regex matches the string entirely. Otherwise it returns None. re.fullmatch("regex", subject) is the same as re.search("\Aregex\Z", subject). This is useful for validating user input. If subject is an empty string then fullmatch() evaluates to True for any regex that can find a zero-length match.

To get all matches from a string, call re.**findall**(regex, subject). This will return an array of all non-overlapping regex matches in the string. "Non-overlapping" means that the string is searched through from left to right, and the next match attempt starts beyond the previous match. If the regex contains one or more capturing groups, re.findall() returns an array of tuples, with each tuple containing text matched by all the capturing groups. The overall regex match is *not* included in the tuple, unless you place the entire regex inside a capturing group.

re.match() does not require the regex to match the entire string. re.match("a", "ab") will succeed.

More efficient than re.findall() is re.**finditer**(regex, subject). It returns an iterator that enables you to loop over the regex matches in the subject string: for m in re.finditer(regex, subject). The for-loop variable m is a Match object with the details of the current match.

Unlike re.search() and re.match(), re.findall() and re.finditer() do not support an optional third parameter with regex matching flags. Instead, you can use global mode modifiers at the start of the regex. E.g. "(?i)regex" matches regex case insensitively.

6.1.1 The Match Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function −

re.match(pattern, string, flags=0)

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.
2	**string** This is the string, which would be searched to match the pattern at the beginning of string.
3	**flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Method & Description
1	**group(num=0)** This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)


```
#!/usr/bin/python

import re


line = "Cats are smarter than dogs"


matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)


if matchObj:

print "matchObj.group() : ", matchObj.group()

print "matchObj.group(1) : ", matchObj.group(1)

print "matchObj.group(2) : ", matchObj.group(2)

else:

print "No match!!"
```

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

```
matchObj.group() :  Cats are smarter than dogs

matchObj.group(1) :  Cats

matchObj.group(2) :  smarter
```

If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.

6.1.2 The Search Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function –

re.search(pattern, string, flags=0)

Here is the description of the parameters –

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.

2	**string** This is the string, which would be searched to match the pattern anywhere in the string.
3	**flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Methods & Description
1	**group(num=0)** This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
usr/bin/python/!#
import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
   print "searchObj.group() : ", searchObj.group()
   print "searchObj.group(1) : ", searchObj.group(1)
   print "searchObj.group(2) : ", searchObj.group(2)
else:
   print "Nothing found!!"
```

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

searchObj.group() : Cats are smarter than dogs
searchObj.group(1) : Cats
searchObj.group(2) : smarter

6.1.3 Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the **string** (this is what Perl does by default).

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"

searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print "search --> searchObj.group() : ", searchObj.group()
else:
   print "Nothing found!!"
```

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

```
No match!!
search --> matchObj.group() : dogs
```

Keyword

String
is traditionally a sequence of characters, either as a literal constant or as some kind of variable.

Did You Know?

Regular expressions originated in 1951, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called regular sets. These arose in theoretical computer science, in the subfields of automata theory (models of computation) and the description and classification of formal languages.

6.1.4 Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

(re.sub(pattern, repl, string, max=0

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* .provided. This method returns modified string

Example

```
usr/bin/python/!#
import re
"phone = "2004-959-559 # This is Phone Number

Delete Python-style comments #
(num = re.sub(r'#.*$', "", phone
print "Phone Num : ", num

Remove anything other than digits #
(num = re.sub(r'\D', "", phone
print "Phone Num : ", num
```

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

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Keyword

Regular expression is a sequence of characters that define a search pattern.

6.2 REGULAR EXPRESSION MODIFIERS: OPTION FLAGS

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these –

Sr.No.	Modifier & Description
1	**re.I** Performs case-insensitive matching.
2	**re.L** Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).
3	**re.M** Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
4	**re.S** Makes a period (dot) match any character, including a newline.
5	**re.U** Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
6	**re.X** Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

6.2.1 Regular Expression Patterns

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

Sr.No.	Pattern & Description
1	**^** Matches beginning of line.
2	**$** Matches end of line.
3	**.** Matches any single character except newline. Using m option allows it to match newline as well.

4	**[...]** Matches any single character in brackets.
5	**[^...]** Matches any single character not in brackets
6	**re*** Matches 0 or more occurrences of preceding expression.
7	**re+** Matches 1 or more occurrence of preceding expression.
8	**re?** Matches 0 or 1 occurrence of preceding expression.
9	**re{ n}** Matches exactly n number of occurrences of preceding expression.
10	**re{ n,}** Matches n or more occurrences of preceding expression.
11	**re{ n, m}** Matches at least n and at most m occurrences of preceding expression.
12	**a| b** Matches either a or b.
13	**(re)** Groups regular expressions and remembers matched text.
14	**(?imx)** Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
15	**(?-imx)** Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.

16	**(?: re)** Groups regular expressions without remembering matched text.
17	**(?imx: re)** Temporarily toggles on i, m, or x options within parentheses.
18	**(?-imx: re)** Temporarily toggles off i, m, or x options within parentheses.
19	**(?#...)** Comment.
20	**(?= re)** Specifies position using a pattern. Doesn't have a range.
21	**(?! re)** Specifies position using pattern negation. Doesn't have a range.
22	**(?> re)** Matches independent pattern without backtracking.
23	**\w** Matches word characters.
24	**\W** Matches nonword characters.
25	**\s** Matches whitespace. Equivalent to [\t\n\r\f].
26	**\S** Matches nonwhitespace.
27	**\d** Matches digits. Equivalent to [0-9].
28	**\D** Matches nondigits.
29	**\A** Matches beginning of string.

30	**\Z** Matches end of string. If a newline exists, it matches just before newline.
31	**\z** Matches end of string.
32	**\G** Matches point where last match finished.
33	**\b** Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
34	**\B** Matches nonword boundaries.
35	**\n, \t, etc.** Matches newlines, carriage returns, tabs, etc.
36	**\1...\9** Matches nth grouped subexpression.
37	**\10** Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a **character code**.

Remember

The Python module re provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

6.2.2 Regular Expression Examples

Literal characters

Sr.No.	Example & Description
1	**python** Match "python".

Character classes

Sr.No.	Example & Description
1	**[Pp]ython** Match "Python" or "python"
2	**rub[ye]** Match "ruby" or "rube"
3	**[aeiou]** Match any one lowercase vowel
4	**[0-9]** Match any digit; same as [0123456789]
5	**[a-z]** Match any lowercase ASCII letter
6	**[A-Z]** Match any uppercase ASCII letter
7	**[a-zA-Z0-9]** Match any of the above
8	**[^aeiou]** Match anything other than a lowercase vowel
9	**[^0-9]** Match anything other than a **digit**

Keyword

Digit is a single character in a numeric system. *For example*, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are all digits.

Special Character Classes

Sr.No.	Example & Description
1	. Match any character except newline
2	**\d** Match a digit: [0-9]

3	**\D** Match a nondigit: [^0-9]
4	**\s** Match a whitespace character: [\t\r\n\f]
5	**\S** Match nonwhitespace: [^ \t\r\n\f]
6	**\w** Match a single word character: [A-Za-z0-9_]
7	**\W** Match a nonword character: [^A-Za-z0-9_]

Repetition Cases

Sr.No.	Example & Description
1	**ruby?** Match "rub" or "ruby": the y is optional
2	**ruby*** Match "rub" plus 0 or more ys
3	**ruby+** Match "rub" plus 1 or more ys
4	**\d{3}** Match exactly 3 digits
5	**\d{3,}** Match 3 or more digits
6	**\d{3,5}** Match 3, 4, or 5 digits

Nongreedy repetition

This matches the smallest number of repetitions −

Sr.No.	Example & Description
1	**<.*>** Greedy repetition: matches "<python>perl>"
2	**<.*?>** Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Sr.No.	Example & Description
1	**\D\d+** No group: + repeats \d
2	**(\D\d)+** Grouped: + repeats \D\d pair
3	**([Pp]ython(,)?)+** Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again −

Sr.No.	Example & Description
1	**([Pp])ython&\1ails** Match python&pails or Python&Pails
2	**(['"])[^\1]*\1** Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Sr.No.	Example & Description
1	**python|perl** Match "python" or "perl"
2	**rub(y|le))** Match "ruby" or "ruble"
3	**Python(!+|\?)** "Python" followed by one or more ! or one ?

Anchors

This needs to specify match position.

Sr.No.	Example & Description
1	**^Python** Match "Python" at the start of a string or internal line
2	**Python$** Match "Python" at the end of a string or line
3	**\APython** Match "Python" at the start of a string
4	**Python\Z** Match "Python" at the end of a string
5	**\bPython\b** Match "Python" at a word boundary
6	**\brub\B** \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
7	**Python(?=!)** Match "Python", if followed by an exclamation point.
8	**Python(?!!)** Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Sr.No.	Example & Description
1	**R(?#comment)** Matches "R". All the rest is a comment
2	**R(?i)uby** Case-insensitive while matching "uby"
3	**R(?i:uby)** Same as above
4	**rub(?:y|le))** Group only without creating \1 backreference

CASE STUDY

PYTHON

Python is an interpreted, dynamically-typed, object-oriented scripting language with a host of built-in data types. It is implemented in C, but in a very object-oriented fashion. The design is a good model for language implementation.

The Python interpreter works by loading a source file or reading a line typed at the keyboard, parsing it into an abstract syntax tree, compiling the tree into bytecode, and executing the bytecode. We will concentrate mainly on how the bytecode is executed, such as how inheritance and environments are implemented.

Parsing

The parsing process is fairly standard. The basic idea is to first convert the input characters into a more abstract representation like name: x, integer: 7, string: "hello", less-than-or-equal, etc. The abstracted characters are called *tokens*. (There are utilities, such as lex, for automatically generating tokenizers, but Python does not use one.) The tokenization process is fully described in the language reference.

For example, the statement

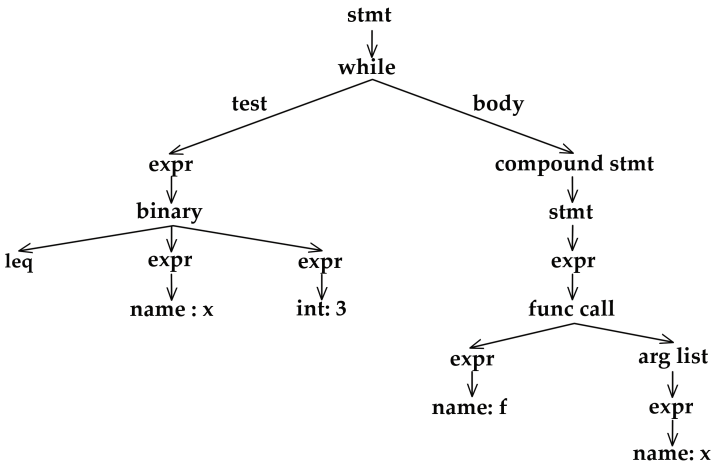
```
while(x <= 3):
    f(x)
```

might be tokenized as

keyword: while
left-paren
name: x
leq
int: 3
right-paren
colon
indent
name: f
left-paren
name: x
right-paren

Then the tokens are assembled into expressions, statements, function definitions, class definitions, etc. Since function definitions contain statements which contain

expressions which may contain nested expressions, and so on, the resulting data structure of tokens is a tree, called a *parse tree* or *abstract syntax tree*. The tokens above might be parsed into this tree:



There are standard utilities, such as yacc, which aid in generating parsers, but Python does not use one. Instead, it uses nested DFAs, which is a recursive tokenizer that fills the tree out downward. A similar technique was used in the book Essentials of Programming Languages. You can learn more about tokenizing, parsing, and compilation in MIT course 6.035 or from various books such as *Crafting a Compiler in C* or *Compilers: Principles, Techniques, and Tools*.

Once you have a parse tree, you can do type checking, type reconstruction, constant folding, liveness analysis, and many other kinds of optimizations and analysis. Python only uses it for compilation.

Compilation

The parse tree is then compiled into bytecode by a recursive walk. For example, a while syntax node contains an expression node for the test and a compound statement node for the body. A while node is compiled into:

loop:
(code for test)
jump_if_false done
(code for body)
jump loop
done:

where the test and body nodes are compiled recursively. Virtually all of the compilation rules can be described as rewrites like this. Compilation has the opposite structure of parsing: it flattens the tree, converting it back to bytes.

Bytecode is virtual machine instructions are packed into an array of bytes. The instructions are based on stack manipulation. Some instructions have no arguments and take up one byte, e.g. BINARY_ADD (pop two values from the stack and push their sum), while other instructions have an additional two-byte integer argument, e.g. LOAD_NAME i (push the value of the *i*-th variable name). The full list of bytecodes is here. Bytecode is paired with a symbol table and constant pool so that names and literals can be referenced by number in the instructions.

The bytecode for the above parse tree is something like:

```
loop:

load-name 1       (x)
load-const 1      (3)
compare-op le

jump_if_false done

load-name 2       (f)
load-name 1       (x)
call-function 1

jump loop
done:
```

Python does not perform much optimization on the bytecode, except for speeding-up local variable access. The reasons for this are tied to how the interpreter works and will be discussed later.

This conversion pattern, where the input is abstracted, processed, then specialized, is a common one in many varieties of programs and is also presented in Abstraction and Specification in Program Development.

Execution

At face value, the execution algorithm is straightforward: fetch the next instruction, perform the required stack manipulation or, in the case of a jump, reposition the instruction pointer. However, much of the real functionality is hidden in the value

objects. *For example*, there is only one BINARY_ADD instruction, yet Python must do very different things when adding integers, floats, strings, or user-defined objects.

Value objects

The trick is to decouple the kinds of values in the language from the interpreter core. This is allows Python to have many built-in data types without a complexity explosion. Each value supports the same interface, some of which is listed here:

add(v)

Add yourself to *v* and return the result as a new object. It corresponds to x + v.

cmp(v)

Compare yourself to *v* and return -1, 0, or 1 (a la strcmp). It corresponds to x == v.

repr()

Return a string representation of yourself.

getattr(name), getitem(v)

Subscript yourself by *name* or arbitrary value *v*. getattr corresponds to x.name and getitem corresponds to x[v].

call(args, keywords)

Call yourself with positional arguments *args* and keyword arguments *keywords*. It corresponds to x(1, 2, 3, foo = 4, bar = 5).

Each method in this interface has a bytecode counterpart which invokes it, e.g. BINARY_ADD. The stack is simply an array of value objects, which each know how to add themselves. Numerical Python takes advantage of this open-ended design to add multi-dimensional arrays, e.g. matrices, to the language.

New kinds of values can even be defined from within Python. An object with a method called __cmp__ will use that method for comparison instead of the default object comparison method. All methods in the value interface can be overrided in this way (including the getattr method which is used to look up object methods in the first place).

The built-in value objects are:

Primitives

Integer/Long

Float

String

File

Function

Composites

Tuple

List

Dictionary

Class

Class instance

Module

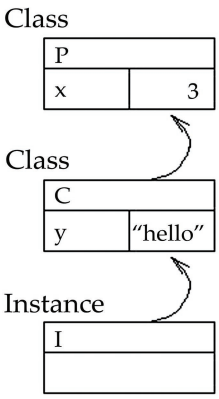
Note that functions are simply values that implement the call method. Thus classes and their instances can just associate names with arbitrary values, i.e. act like dictionaries.

Inheritance

Classes and instances differ slightly from dictionaries, however, in that instances inherit from classes and classes can inherit from other classes. The inheritance is such that changes to classes are immediately visible in their descendants and instances, as illustrated in this example:

```
class p:
    x = 3


class c(p):
    y = "hello"


i = c()
i.y      (prints "hello")
i.x      (prints 3)
p.x = 4
c.x      (prints 4)
i.x      (prints 4)
```

Thus classes and instances differ from dictionaries in that if a read cannot be resolved, the request is passed to the parent class. Since classes can change at run-time, this makes the inheritance process highly dynamic. This is a recurring pattern called a Chain of Responsibility. The object diagram for the above example is thus:



Note that if Python hadn't provided inheritance, we could recreate it by providing a __getattr__ method to do the forwarding. The parent class would be stored explicitly in a variable. This would have the interesting side-effect that inheritance links could be changed at run-time by modifying this variable. The Chain of Responsibility pattern fully supports such changes.

What happens on a write? If requests were forwarded up the Chain, as with a read, there would be no way to override a slot in a parent class. In the above example, c.x = 5 would be the same as p.x = 5. (If the variable turns out to be unbound, then the object written to would gain a new slot.) If requests were not forwarded, then the object written to would always gain a new slot, shadowing the ancestor slot. Python chooses the latter, so that parent methods can be overrided in children. Thus we get:

```
c.x = 5
p.x      (prints 4)
i.x      (prints 5)

i.x = i.x
c.x = 6
i.x      (prints 5)
```

Note that i.x = i.x performs useful work under this design choice.

Does Python need to distinguish between classes and instances? Both are essentially dictionaries and have the same read/write and inheritance mechanism. Perhaps it's for efficiency reasons: there are usually far fewer classes than instances, so certain optimizations can be applied to classes but not instances.

Variable scope

Python variable environments, aka "stack frames", have many of the qualities of objects. They are dictionaries of names with the same kind of inheritance semantics. That is, variables used in a function by default refer to global names, but if a variable is assigned to in the function, it becomes local. The same applies to class definitions.

One can imagine also designing variable scope with a Chain of Responsibility. In fact, some languages do this, even ones which don't have the corresponding notion of inheritance. *Structure and Interpretation* of Computer Programs uses it to describe Scheme's method of variable scope. The Self language actually treats environments as objects and uses inheritance to implement lexical scope!

However, Python does not take this approach, probably for efficiency reasons. Python does not allow more than two simultaneous environments (the global and the local), as noted in the PLE exercise on nested scopes. This allows certain optimizations, like the LOAD_FAST bytecode, but can confuse programmers used to lexical scoping. The following C++ fragment has no equivalent in Python:

```
int x = 1;
if(x > y) {
  int x = 2;
  cout << x;     // prints 2
}
cout << x;       // prints 1
```

Scheme programmers may wonder: then how does Python implement lambda? The escape was to abandon lexical scoping for lambda. The body of a lambda only has the global and local environments.

SUMMARY

- Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module.
- Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster.
- The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated.
- Python is a high level open source scripting language. Python's built-in "re" module provides excellent support for regular expressions, with a modern and complete regex flavor.
- Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string (this is what Perl does by default).
- Except for control characters, (+ ? . * ^ $ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

KNOWLEDGE CHECK

1. **The character Dot (that is, '.') in the default mode, matches any character other than .....**
   a. caret
   b. ampersand
   c. percentage symbol
   d. newline
2. **The expression a{5} will match ..... characters with the previous regular expression.**
   a. 5 or less
   b. exactly 5
   c. 5 or more
   d. exactly 4
3. **Choose the function whose output can be: <_sre.SRE_Match object; span=(4, 8), match='aaaa'>.**
   a. >>> re.search('aaaa', "alohaaaa", 0)
   b. >>> re.match('aaaa', "alohaaaa", 0)
   c. >>> re.match('aaa', "alohaaa", 0)
   d. >>> re.search('aaa', "alohaaa", 0)
4. **Which of the following functions clears the regular expression cache?**
   a. re.sub()
   b. re.pos()
   c. re.purge()
   d. re.subn()
5. **Which of the following functions results in case insensitive matching?**
   a. re.A
   b. re.U
   c. re.I
   d. re.X
6. **Which of the following creates a pattern object?**
   a. re.create(str)
   b. re.regex(str)

c. re.compile(str)
d. re.assemble(str)

7. What does the function re.match do?

a. matches a pattern at the start of the string
b. matches a pattern at any position in the string
c. such a function does not exist
d. none of the mentioned

8. What does the function re.search do?

a. matches a pattern at the start of the string
b. matches a pattern at any position in the string
c. such a function does not exist
d. none of the mentioned

REVIEW QUESTIONS

1. What is the match function?
2. What is the search function?
3. Discuss about search and replace.
4. Discuss about regular expression modifiers: option flags.
5. Describe the regular expression patterns.

Check Your Result

1. (d) 2. (b) 3. (a) 4. (c) 5. (d)
6. (c) 7. (a) 8. (b)

REFERENCES

1. A.M. Kuchling (2001-12-21). "PEP 255: Simple Generators". What's New in Python 2.2. Python Foundation. Retrieved 2008-09-05.
2. Barry Warsaw (2011-11-09). "PEP 404 -- Python 2.8 Un-release Schedule". Retrieved 2012-10-07.
3. Guido van Rossum (January 20, 2009). "The History of Python". Retrieved March 3, 2018.
4. Neal Norwitz; Barry Warsaw (2006-06-29). "PEP 361 -- Python 2.6 and 3.0 Release Schedule". Retrieved 2012- 10-07.
5. Rossum, Guido van van. "Python 3000 FAQ". artima. com. Retrieved December 27, 2016.

CHAPTER

7

PYTHON MULTITHREADING

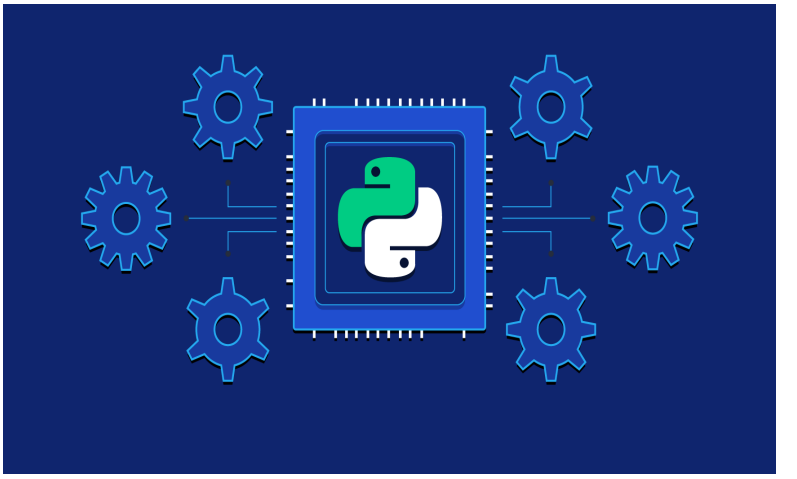
"The main languages out of which web applications are built - whether it›s Perl or Python or PHP or any of the other languages - those are all open source languages. So the infrastructure of the web is open source the web as we know it is completely dependent on open source. "

–Mitch Kapor

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Discuss on python threading and python multithreading
2. Determine the useful functions in python multithreading



INTRODUCTION

Multithreading is a threading technique in Python programming to run multiple threads concurrently by rapidly switching between threads with a CPU help (called

context switching). Besides, it allows sharing of its data space with the main threads inside a process that share information and communication with other threads easier than individual processes. Multithreading aims to perform multiple tasks simultaneously, which increases performance, speed and improves the rendering of the application.

Following are the benefits to create a multithreaded application in Python, as follows:

- It ensures effective utilization of computer system resources.
- Multithreaded applications are more responsive.
- It shares resources and its state with sub-threads (child) which makes it more economical.
- It makes the multiprocessor architecture more effective due to similarity.
- It saves time by executing multiple threads at the same time.
- The system does not require too much memory to store multiple threads.

It is a very useful technique for time-saving and improving the performance of an application. Multithreading allows the programmer to divide application tasks into sub-tasks and simultaneously run them in a program. It allows threads to communicate and share resources such as files, data, and memory to the same processor. Furthermore, it increases the user's responsiveness to continue running a program even if a part of the application is the length or blocked.

7.1 PYTHON THREADING – PYTHON MULTITHREADING

Python threading module is used to implement multithreading in python programs. Threading in python is used to run multiple threads (tasks, function calls) at the same time. Note that this does not mean that they are executed on different CPUs. Python threads will not make your program faster if it already uses 100 % CPU time. In that case, you probably want to look into parallel programming. If you are interested in parallel programming with python. Python multiprocessing is one of the similar module that we looked into sometime back.

The threading module builds on the low-level features of thread to make working with threads even easier and more pythonic. Using threads allows a program to run multiple operations concurrently in the same process space. In Computer Science, threads are defined as the smallest unit of work which is scheduled to be done by an Operating System. Some points to consider about Threads are:

- Threads exists inside a process.
- Multiple threads can exist in a single process.
- Threads in same process share the state and memory of the parent process.

This was just a quick overview of Threads in general. This post will mainly focus on the threading module in Python.

Did You Know?

Python 2.0 was released on 16 October 2000 and had many major new features, including a cycle-detecting garbage collector and support for Unicode. With this release, the development process became more transparent and community-backed.

7.1.1 Getting Started with Python Multithreading

Running several threads is similar to running several different programs concurrently, but with the following benefits –

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Let us start by creating a Python module, named download.
py. This file will contain all the functions necessary to fetch the list of images and download them. We will split these :functionalities into three separate functions

- get_links

- download_link
- setup_download_dir

The third function, setup_download_dir, will be used to create a download destination directory if it does not already exist.

Imgur's API requires **HTTP** (Hypertext Transfer Protocol) requests to bear the Authorization header with the client ID. You can find this client ID from the dashboard of the application that you have registered on Imgur, and the response will be JSON encoded. We can use Python's standard JSON library to decode it. Downloading the image is an even simpler task, as all you have to do is fetch the image by its URL and write it to a file.

Keyword

HTTP is an application protocol for distributed, collaborative, and hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

7.1.2 Python Multithreading Modules for Thread Implementation

Python offers two modules to implement threads in programs.

- ***<thread>*** module and
- ***<threading>*** module.

For your information, <*thread*> module is deprecated in Python 3 and renamed to <*_thread*> module for backward compatibility. But we will explain both methods because many of the users still use legacy Python versions.

The key difference between the two modules is that the module <*thread*> implements a thread as a function. On the other hand, the module <*threading*> offers an object-oriented approach to enable thread creation.

IronPython, a Python implementation using the .NET framework, does not have a GIL, and neither does Jython, the Java-based implementation. You can find a list .of working Python implementations

7.1.3 Difference between Multiprocessing and Multithreading

Multiprocessing and Multithreading both adds performance to the system. Multiprocessing is adding more number of or

CPUs/processors to the system which increases the computing speed of the system. Multithreading is allowing a process to create more threads which increase the responsiveness of the system.

Comparison Chart

BASIS FOR COMPARISON	MULTIPROCESSING	MULTITHREADING
Basic	Multiprocessing adds CPUs to increase computing power.	Multithreading creates multiple threads of a single process to increase computing power.
Execution	Multiple processes are executed concurrently.	Multiple threads of a single process are executed concurrently.
Creation	Creation of a process is time-consuming and resource intensive.	Creation of a thread is economical in both sense time and resource.
Classification	Multiprocessing can be symmetric or asymmetric.	Multithreading is not classified.

Key Differences between Multiprocessing and Multithreading

- The key difference between multiprocessing and multithreading is that multiprocessing allows a system to have more than two CPUs added to the system whereas multithreading lets a process generate multiple threads to increase the computing speed of a system.
- Multiprocessing system executes **multiple processes** simultaneously whereas, the multithreading system let execute multiple threads of a process simultaneously.
- Creating a process can consume time and even exhaust the system resources. However creating threads is economical as threads belonging to the same process share the belongings of that process.
- Multiprocessing can be classified into symmetric multiprocessing and asymmetric multiprocessing whereas, multithreading is not classified further.

The benefits of multithreading can be gradually increased in multiprocessing environment as multithreading on a multiprocessing system increases parallelism.

7.2 FUNCTIONS IN PYTHON MULTITHREADING

The module 'threading', for Python, helps us with thread-based parallelism. It constructs higher-level threading interfaces on top of the lower level _thread module. Where _thread is missing, we cannot use threading. For such situations, we have dummy_threading.

Remember

The fork function creates a copy of the process, all memory pages are copied, open file descriptors are copied etc. All this stuff is intuitive for a UNIX programmer. One important thing that differs the child process from the parent is that the child has only one thread. Cloning the whole process with all threads would be problematic and in most cases not what the programmer wants.

We have the following functions in the Python Multithreading module:

a. active_count()

This returns the number of alive(currently) Thread objects. This is equal to the length the of the list that enumerate() returns.

```
>>> threading.active_count()
2
```

b. current_thread()

Based on the caller's thread of control, this returns the current Thread object. If this thread of control is not through 'threading', it returns a dummy thread object with limited functionality.

```
>>> threading.current_thread()


<_MainThread(MainThread, started 14352)>
```

c. get_ident()

get_ident() returns the current thread's identifier, which is a non-zero integer. We can use this to index a dictionary of thread-specific data. Apart from that, it has no special meaning.

When one thread exits and another creates, Python recycles such an identifier.

```
>>> threading.get_ident()
14352
```

d. enumerate()

This returns a list of all alive(currently) Thread objects. This includes the main thread, daemonic threads, and dummy thread objects created by current_thread(). This obviously does not include terminated threads as well as those that have not begun yet.

```
>>> threading.enumerate()
[<_MainThread(MainThread, started 14352)>, <Thread(SockThread, started daemon 9864)>
```

e. main_thread()

This method returns the main Thread object. Normally, it is that thread which started the interpreter.

```
>>> threading.main_thread()
<_MainThread(MainThread, started 14352)>
```

f. settrace(func)

settrace() traces a function for all threads we started using 'threading'. The argument func passes to sys.settrace() for each thread before it calls its run() method.

```
>>> def sayhi():
print("Hi")
>>> threading.settrace(sayhi)
>>>
```

g. setprofile(func)

This method sets a profile function for all threads we started from 'threading'. It passes *func* to sys.setprofile() for each thread before it calls its run() method.

```
>>> threading.setprofile(sayhi)
   >>>      1.
```

h. stack_size([size])

stack_size() returns the stack size of a thread when creating new threads. *size* is the stack size we want to use for subsequently created threads. This must be equal to 0 or a positive integer of value at least 32,768 (32KiB). When not specified, it uses 0. And if it does not support changing thread stack size, it raises a RuntimeError.

When we pass an invalid stack size, it raises a ValueError, and does not modify it. The minimum stack size it currently supports to guarantee enough stack space for the interpreter itself is 32KiB. Some platforms may need a minimum stack size of

greater than 32KiB. Others may need to allocate in multiples of system memory page size.

```
>>> threading.stack_size()
0
```

Apart from functions, 'threading' also defines a constant.

h. TIMEOUT_MAX

This holds the maximum allowed value for this constant, the timeout parameter for blocking functions like Lock.acquire(), Condition.wait(), RLock.acquire(), and others. If we denote a timeout greater than this, it raises an OverflowError.

```
>>> threading.TIMEOUT_MAX
4294967.0
```

In Java, locks and condition variables are the basic behavior of every object. Whereas in Python, they are individual objects. Here, the class Thread supports some of the functionality of class **Thread** in Java. However, currently, we have no thread groups, priorities, and we cannot destroy, stop, suspend, resume, or interrupt threads. When we implement the static methods from Java's Thread, they map to module-level functions. This way, 'threading' is much like Java's threading model in design.

Keyword

Thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

7.2.1 Thread-Local Data

That data for which the values are thread-specific, is thread-local. To manage such data, we can create an instance of local/a subclass, and then store attributes on it.

```
>>> mydata=threading.local()
>>> mydata.x=7
>>>
```

These instance values differ for each thread. We have the following class denoting thread-local data:

class threading.local

A class that represents thread-local data.

7.2.2 Thread Objects

The Thread class that we mentioned earlier in this blog denotes an activity running in a separate thread of control. We can represent this activity either by passing a callable object to the constructor, ot by overriding the method run() in a subclass. You must make sure to not override other methods in a subclass, except for the constructor. In short, only override a .class' __init__() and run() methods

Once the interpreter creates a thread object, we must start its activity by calling its start() method. This will invoke its run() method in a separate thread of control. Once this happens, we consider the thread to be 'alive'. When run() terminates normally or raising an exception we did not handle, it is no longer alive. To test whether a thread is alive, we may use the method is_alive().

A thread may call another's join() method. This will block the calling thread until the other terminates.

Threads have names, and we can pass these names to the constructor, and even read or modify them.

We can flag a thread as a 'daemon thread'. This means that the whole program exits when only the daemon threads remain. This initial value comes from the creating thread. We can set this flag via the property 'daemon', or via the **constructor** argument 'daemon'. Daemons abruptly stop at shutdown, and they may not properly release all the resources held. These resources may include open files, database transactions, and others. To stop our threads gracefully, we must make them non-daemonic. It is also preferable to use a suitable signaling process, like an Event.

The 'main thread' object pertains to the initial thread of control in our program; it is not a daemon.

Finally, it is possible that the interpreter creates 'dummy thread objects'. These are 'alien threads' (threads of control started outside 'threading', for ex, directly from C code). Such objects have limited functionality, and are always live and daemonic. We cannot join() them. We can also never delete them since it is impossible to detect when they terminate.

Keyword

Constructor is a block of codes similar to the method. It is called when an instance of the class is created.

This is the class:

class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)

Take note:

- Always call the constructor with keyword arguments. It has the following arguments:
- *group* must be None. Python reserves this for future extension when we implement a ThreadGroup class.
- *target* is a callable object that run() will invoke. The default for this is None, which means it calls nothing.
- *name* is the name of the thread. The default for this is "Thread-N". Here, N is a small decimal number.
- *args* is an argument tuple. It helps invoke the target. The default for this is ().
- *Kwargs* is a dictionary holding keyword arguments. Even this helps invoke the target. The default for this is {}.
- *daemon* decides whether the thread is daemonic. When None, it inherits the daemonic property from the current thread. The default for this is None.
- Ensure that you invoke the base class constructor(Thread.__init__()) first if the subclass overrides the constructor.

Thread has the following methods:

a. start()

This starts thread activity. For a thread object, we can call it maximum once; if we call it again, it raises a RuntimeError. This lets run() for the object invoke in a separate thread of control.

- >>> threading.Thread.start(threading.current_thread())
- Traceback (most recent call last):
- File "<pyshell#135>", line 1, in <module>
- threading.Thread.start(threading.current_thread())

RuntimeError: threads can only be started once

b. run()

This method explains the thread's activity. It invokes the callable object we passed to the object's constructor as the target argument, if it exists. This is with keyword and sequential arguments from *kwargs* and *args*.

We can override run() in a subclass.

c. join(timeout=None)

For join() to work, we must wait until the thread terminates. Because when that happens, it blocks the calling thread until the one on which we call join() terminates normally or via an exception we did not handle, or until *timeout* occurs.

When you do provide a *timeout* (other than None), make sure it is a floating point number. This is so you can pass a timeout in seconds or fractions.

So, what is the return value? Well, join() always returns None. Hence, you will need to call is_alive() after calling join() to determine if a timeout happened. If we find out that it is indeed still alive, then we infer that the join() call timed out.

However, if *timeout* is None, or if we did not pass it, this blocks the operation until the thread terminates. We can join() a thread many times.

Finally, join() will raise a RuntimeError if we try to join the current thread, because that causes a deadlock. To join() a thread before we start it also causes an error.

- >>> threading.Thread.join(threading.current_thread())
- Traceback (most recent call last):
- File "<pyshell#138>", line 1, in <module>
- threading.Thread.join(threading.current_thread())

RuntimeError: cannot join current thread

d. name

This is a string we use for identification; it has no meaning. We can also give the .same meaning to multiple threads. The constructor sets the initial name

- >>> threading.Thread.name='First'

>>>

e. getName() and setName()

.These are old getter and setter APIs for *name*. We use them directly as properties

f. ident

If we started the thread, this returns its identifier. Otherwise, it returns None. Note that it is a non-zero integer, like in the get_ident() function. Python may recycle identifiers when one thread exits and another creates. Such identifiers exist even after a thread exits.

g. is_alive()

This returns whether the thread is alive. is_alive() returns true from just before run() starts until just after it terminates.

>>> threading.Thread.is_alive(threading.current_thread())

True

h. daemon

daemon is a Boolean value that tells us whether the thread is a daemon. If it is, it returns True. We must set it before we call start(). Otherwise, it raises a RuntimeError. Its initial value comes from the creating thread. The main thread is not a daemon; hence, all threads in the main thread have a default of False for *daemon*.

When only daemon threads remain, the whole program exits.

i. isDaemon() and setDaemon()

These are old getter and setter APIs for *daemon*. You can use them directly as properties.

Remember

Whenever a function wants to modify a variable, it locks that variable. When another function wants to use a variable, it must wait until that variable is unlocked.

7.2.3 Lock Objects

A synchronization primitive, a primitive lock does not belong to a certain thread when locked. This is the lowest-level synchronization primitive we currently have in Python, and we implement it using the extension module _thread.

Such a lock can be in one of two states: 'locked' and 'unlocked'. When we create a lock, it is in the 'unlocked' state. It also has two methods- acquire() and release(). When we want to lock it, acquire() changes its state to 'locked', and immediately returns it. If it was 'locked' instead, then acquire() blocks until another thread calls release(). This changes the state to 'unlocked'. Finally, acquire() resets it to 'locked', and then returns immediately.

If you try to release a lock that is already unlocked, it raises a RuntimeError.

These locks also support the CMP(Context Management Protocol).

When acquire() blocks more than one thread, only one thread continues when release() resets the state to 'unlocked'. Which one, you ask? Well, we cannot say.

Also, all methods execute atomically.

This is the class:

class threading.Lock

This call implements primitive lock objects. Once a thread acquires a lock, the interpreter blocks further attempts to

acquire it. Only after it releases, does any other thread have a chance in acquiring it. Any thread may release a lock.

a. acquire(blocking=True, timeout=-1)

This method acquires a blocking or non-blocking lock. When *blocking*=True, it blocks until the lock unlocks. Then, it changes its state to 'locked', and returns True. And when it is False, it does not block. A call with *blocking*=True that blocks, immediately returns False. Otherwise, it sets the lock to 'locked' and returns True.

timeout is a floating-point argument. When it has a positive value, it blocks for a maximum of *timeout* number of seconds; as long as the lock is not acquirable. When it is -1, it denotes an unbounded wait.

When *blocking* is False, we cannot specify *timeout*.

Also, if the lock acquires successfully, it returns True; otherwise, False, like when *timeout* expires.

b. release()

This method releases a lock. You can call it from any thread. This means that any thread can release a lock, no matter which thread has acquired it.

When 'locked', release() resets it to 'unlocked', and returns. If other threads wait for it to unlock, only one gets to continue once it unlocks.

When we call release() on an 'unlocked' lock, it raises a RuntimeError.

release() does not return any value.

7.2.4 RLock Objects

RLock is very important topic when you learn Python Multithreading. An RLock is a reentrant lock. It is a synchronization primitive that a certain thread can acquire again and again. It does so using concepts like 'owning thread' and 'recursion level', and locked/unlocked states. When locked, an RLock belongs to a certain thread; but when unlocked, no thread owns it.

Now, how does this work? To lock, a thread calls acquire(). Now that this thread owns the lock, it returns. To unlock it, a thread calls release(). It is also possible to nest acquire()/release() pairs. The outermost release() resets the lock to the 'unlocked' state. It also lets another blocked thread to continue.

Reentrant locks also support CMP(Context Management Protocol).

This is the class:

class threading.RLock

RLock implements reentrant lock objects. Such a lock only release by the thread holding it. A thread can acquire it again without blocking. However, it must release it once each time it acquires it.

It has two methods:

a. acquire(blocking=True, timeout=-1)

acquire() lets us acquire a blocking or non-blocking lock. Without arguments, if the thread already owns the lock, this method ups the recursion level by one, and then returns. If it does not already own it, and another thread owns it, it blocks until the lock 'unlocks'. And once unlocked, and if it does not belong to any other thread, acquire() declares ownership and sets recursion level to 1, and then returns. If more than one thread waits blocked, at once, only one will get ownership.

Remember

Context management is a dynamic computer process that uses 'subjects' of data in one application, to point to data resident in a separate application also containing the same subject

This method returns no value. Finally, when we set *blocking* to True, it does the same things we discussed, and then returns True.

When *blocking* is False, however, it doe not block. When a call without arguments blocks, it returns False. Otherwise, it does what it does for a call without arguments, and then returns True. And when we call acquire() with *timeout*, which is a floating-point number, with a positive value, this blocks for a maximum of *timeout* number of seconds, as long as we cannot acquire the lock. If a thread has acquired it, it returns True; if *timeout* has elapsed, it returns False.

b. release()

This method releases a lock and decrements the recursion level. Once the decrement is 0, it resets the lock to the 'unlocked' state. This means no thread owns it. If other threads are blocked, only one of them may continue. If the decrement is non-zero, the lock stays in the 'locked' state, and belongs to the calling thread.

You should only call release() when the calling thread actually owns the lock. If it is already 'unlocked', this raises a RuntimeError.

release() returns no value.

Now lets come to Condition Objects in Python Multithreading.

7.2.5 Condition Objects

A condition variable always pertains to a lock, and we can pass it in, or it creates by default. When several such condition

variables must share a lock, we can pass it in. But we do not need to exclusively track a lock; it is a part of the condition object. A condition variable follows CMP (Context Management Protocol) in that it uses the with-statement to acquire the associated lock as long as the enclosed block is alive. acquire() and release() call the lock's methods.

For other methods, we must call them with the associated lock the thread holds. Once wait() releases the lock, it blocks until another thread wakes it up with a call to notify() or notify_all(). After this, wait() acquires the lock again, and then returns. We can also specify a *timeout*.

While notify() awakens one waiting thread, if any, notify_all() awakens all threads waiting for the condition variable. Note that these two methods do not release the lock. So, the threads awakened do not return from wait() immediately. They return only when the calling thread for notify() or notify_all() gives up ownership for the lock.

This is the class:

class threading.Condition(lock=None)

Condition implements condition variable objects. A condition variable lets any number of threads wait until another thread notifies them.

If *lock* is not None, and we do pass it, make sure it is a Lock or RLock object. This should also serve as the underlying lock, otherwise this creates a new RLock object.

It has the following methods:

a. acquire(*args)

This acquires the underlying lock. It calls the corresponding method on it, and returns what that method returns.

b. release()

This releases the underlying lock. It calls the corresponding method on it, and returns nothing.

c. wait(timeout=None)

This method waits until a timeout happens or until someone notifies it. If at the time of calling wait(), the calling thread does not own the lock, this raises a RuntimeError.

wait() releases the underlying lock, then blocks until a notify()/notify_all() call for the same condition variable in another thread wakes it up, or until *timeout* happens. And once this happens, it acquires the lock again, and then returns.

When we do pass *timeout*, and that is not None, make sure it is a floating point number denoting a timeout for the operation in seconds or fractions.

If the underlying lock is an RLock, its release() method does not release it, because this does not necessarily unlock it if it was acquired multiple times recursively. So, what do we do? We use an internal interface of the RLock class. This unlocks it even when it was recursively acquired many times. Then, we use another internal interface to restore the **recursion** level when the thread acquires the lock again.

wait() returns False if *timeout* expires. Otherwise, it returns True.

d. wait_for(predicate, timeout=None)

This method waits until a condition becomes True. The *predicate* is a callable with a Boolean result. We may provide a *timeout* to specify a maximum time to wait.

wait_for() is a utility method, and it can repeatedly make a call to wait() until the predicate satisfies, or until a timeout happens. It returns the predicate's last return value, and returns False if the method times out.

Keyword

Recursion occurs when a thing is defined in terms of itself or of its type. Recursion is used in a variety of disciplines ranging from linguistics to logic.

With this method, the same rules apply as do to wait(). When we call it, the lock must be held, and acquires again on return. This evaluates the predicate with the lock held.

(e. notify(n=1

notify() wakes up a thread waiting on this condition, if there is any. When we call it, if the calling thread does not own the lock, this raises a RuntimeError. It wakes up a maximum of n threads that wait for the condition variable. If no threads wait, then it is a no-operation(NOP).

If at least n threads wait, this implementation will wake exactly n threads up. But we cannot rely on this behavior. An optimized implementation can occasionally wake more than n threads up.

f. notify_all()

This wakes up all threads that wait on this condition. So, this is like notify(), except that it wakes all waiting threads instead of exactly one. If at the time of calling it, if the calling thread does not own the lock, this raises a RuntimeError.

7.2.6 Semaphore Objects

Early Dutch computer scientist Edsger W. Dijkstra invented one of the oldest synchronization primitives. Instead of acquire() and release(), he used P() and V().

What is a semaphore? It is a primitive that lets us manage an internal counter. Each call to acquire() decrements, and each call to release() decrements it. But let us tell you, the counter never goes below zero. When it is 0, acquire() blocks, and waits until a thread makes a call to release().

Semaphores in Python Multithreading support CMP(Context Management Protocol).

This is the class we have:

class threading.Semaphore(value=1)

It implements semaphore objects. A semaphore holds an atomic counter denoting the count of release() calls minus the count of acquire() calls, added to an initial value. acquire() blocks if needed until it can leave the counter non-negative and still return. The default value for the counter is 1.

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of release() calls minus the number of acquire() calls, plus an initial value. The acquire() method blocks if necessary until it can return without making the counter negative. If not given, value defaults to 1.

value can serve as an initial value for the internal counter. The default for this is 1. If we pass a value less than 0, this raises a ValueError.

It has the following methods:

(a. acquire(blocking=True, timeout=None

This acquires a semaphore. When we pass a *timeout* value other than None, it blocks for a maximum of *timeout* seconds. If in that interval, acquire() does not complete successfully, it returns False. Otherwise, it returns True.

When we call it without arguments, the following cases may be:

- If, on entry, the internal counter is greater than zero, it decrements it by one, and then returns.
- If, on entry, the internal counter is zero, it blocks until a call to release() wakes it up. Now that the counter is greater than 0, it decrements it by 1, and then returns True. Each call to release() wakes exactly one thread. We cannot say what order this happens in.
- When we call it with *blocking* with a value of False, it does not block. And if a call without arguments blocks, then it returns False. Otherwise, it does the same as when called without arguments, and then returns True.

b. release()

This method releases a semaphore, and increments the internal counter by 1. When, on entry, it is 0, and another thread waits for it to grow again, it wakes that thread up.

We also have bounded semaphores:

class threading.BoundedSemaphore(value=1)

This class implements bounded semaphore objects. Such objects ensure that their current values do not exceed their initial values. It this happens, this raises a ValueError. Mostly, semaphores guard resources with limited capacity, for ex., a database server. Where the resource size is fixed, use a bounded semaphore. But if it releases the

semaphore way too many times, then you may have a bug in your code. The default for this is 1.

Let's take an example. The main thread initializes the semaphore before spawning any worker threads:

>>> maxconnections=5

>>> pool_sema=threading.Bounded Semaphore (value = maxconnections)

Now that it is spawned, the worker threads call acquire() and release() when they must connect to the server:

- >>>with pool_sema:
- conn=connectdb()
- try:
- #use connection
- finally:
- conn.close()
- Using a bounded semaphore lessens the chances of programming errors.

7.2.7 Event Objects

An extremely simple tool in Python Multithreading to communicate, it lets one thread play an event, and the other must wait for it. An event object deals an internal flag. The methods set() and clear() allow us to set and reset it to True and False, respectively. Until *flag* is True, wait() blocks.

:This is the class

class threading.Event

This class implements event objects. An event handles a flag, and we can use the methods set() and clear() to set and reset it to True and False, respectively. Initially, the flag is False. wait() blocks it until it becomes True.

It has the following methods:

a. is_set()

If the internal flag is True, it returns True.

b. set()

This method sets the internal flag to True, and wakes all threads waiting for it to become True. Once it is True, waiting threads do not block at all.

c. clear()

This resets the internal flag to False. Eventually, waiting threads block until somebody calls set() to set the internal flag to True yet again.

d. wait(timeout=None)

Until the internal flag is True, this method blocks. On entry, if it is True, it returns immediately. Otherwise, it blocks until another thread makes a call to set() to set the flag to True, or until *timeout* happens.

When *timeout* exists, and is not Now, make sure it is a floating-point number denoting a *timeout* for the operation in seconds or fractions.

It returns True only if the internal flag is True- either before the call to wait(), or after. This way, wait() always returns True. However, if *timeout* exists and the operation times out, it returns False.

7.2.8 Timer Objects

Timer denotes an action that should run only after a given amount of time; it is a timer in Python Multithreading. This is a subclass of Thread, and we can also use it to learn how to create our own threads.

When we call start() on a thread, a timer start with it. We can stop it before it begins, if we call cancel() on it. Before executing, a timer waits for some interval; this may differ from the interval we specify.

Take an example:

```
>>> def hello():
print("Hello")
>>> t=threading.Timer(30.0,hello)
>>> t.start()
```

This is the class:

class threading.Timer(interval, function, args=None, (kwargs=None

It creates a timer that runs a function(with arguments args and kwargs) after *interval*seconds. When args is None, it uses an empty list. This is the default. And when kwargs is None, it uses an empty dict. This is a default too.

It has one method:

a. cancel()

Keyword

Synchronization refers to one of two distinct but related concepts: synchronization of processes, and synchronization of Data.

This stops the timer, and then cancels its action. This only works if the timer is waiting.

Now the last in Python Multithreading is Barrier Objects.

7.2.9 Barrier Objects

Barrier is a simple **synchronization** primitive to a fixed number of threads that must wait for each other. Each thread tries to pass the barrier by making a call to wait; it blocks until all threads have done this. Then, the threads release simultaneously.

You can reuse a barrier any number of times for the same number of threads. .Let's take an example

:A way to synchronize a client and server thread is

```
>>> b=threading.Barrier(2,timeout=5)
>>> def server():
start_server()
b.wait()
while True:
connection = accept_connection()
process_server_connection(connection)
>>> def client():
b.wait()
while True:
connection = make_connection()
process_client_connection(connection)
```

This is the class:

class threading.Barrier(parties, action=None, timeout=None)

Barrier creates a barrier object for *parties* number of threads. When we do pass *action*, it is a callable that a thread will call when we release it. Finally, *timeout* is a default value for timeout if we do not specify the same for wait().

It has the following methods:

a. wait(timeout=None)

wait() passes the barrier. Once all thread parties have called wait(), they all release together. If we do pass a value for *timeout*, it uses this one, no matter whether we provided a value for the same to the class constructor.

It returns any integer value from 0 to parties-1. This is different for each thread. You can use this to choose a thread to do special housekeeping. Take an example:

```
>>> i=barrier.wait()
>>> if i==0:
#Only one thread must print this
print("passed the barrier")
```

If we provided an action to the constructor, one thread calls it before releasing. If this raises an error, the barrier sinks into a 'broken' state. The same happens if the call times out.

Finally, wait() may raise a BrokenBarrierError if the barrier breaks or resets as a thread waits.

b. reset()

This function resets the barrier to its default, empty state. Any waiting threads receive a BrokenBarrierError.

reset() may need external synchronization if other threads with unknown states exist. If it breaks a barrier, just create a new one.

c. abort()

abort() puts a barrier into a 'broken' state. Consequently, active/future calls to wait() fail with a BrokenBarrierError. To avoid deadlocking an application, we may need to abort it. This is one use-case.

Try to create the barrier with a sensible value for *timeout* so it automatically guards against a thread going haywire.

d. parties

This returns the number of threads we need to pass the barrier.

e. n_waiting

This returnd the number of threads that currently wait in the barrier.

f. broken

This is a Boolean value that is True if the barrier is in a 'broken' state.

Check out this exception that Barrier may raise:

exception threading.BrokenBarrierError

This is a subclass of RuntimeError, and it raises if a Barrier object resets or breaks.

7.2.10 Using locks, Conditions, and Semaphores in the with-statement

If an object in this module has acquire() and release(), we can use it as a context-manager for a with-statement. When it enters the block, it calls acquire(), and when it exits it, it calls release().

This is the syntax for the same:

```
with some_lock:
#do something
```

This is the same as:

```
>>> some_lock.acquire()
>>> try:
#do something
finally:
some_lock.release()
```

We can currently use Lock, RLock, Condition, Semaphore, and BoundedSemaphore objects as context-managers for with-statements.

ROLE MODEL

EDSGER DIJKSTRA: DUTCH COMPUTER SCIENTIST

Edsger Dijkstra, in full Edsger Wybe Dijkstra, (born May 11, 1930, Rotterdam, Neth.—died Aug. 6, 2002, Nuenen), Dutch computer scientist. He received a Ph.D. from the University of Amsterdam while working at Amsterdam's Mathematical Center (1952–62). He taught at the Technical University of Eindhoven from 1963 to 1973 and at the University of Texas from 1984. He was widely known for his 1959 solution to the shortest-path problem; his algorithm is still used to determine the fastest way between two points, as in the routing of communication networks and in flight planning. His research on the idea of mutual exclusion in communications led him to suggest in 1968 the concept of computer semaphores, which are used in virtually every modern operating system. A letter he wrote in 1968 was extremely influential in the development of structured programming. He received the Turing Award in .1972

SUMMARY

- Multithreading is a threading technique in Python programming to run multiple threads concurrently by rapidly switching between threads with a CPU help (called context switching). Besides, it allows sharing of its data space with the main threads inside a process that share information and communication with other threads easier than individual processes.
- Multithreading allows the programmer to divide application tasks into sub-tasks and simultaneously run them in a program. It allows threads to communicate and share resources such as files, data, and memory to the same processor.
- Python threading module is used to implement multithreading in python programs. Threading in python is used to run multiple threads (tasks, function calls) at the same time.
- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Multiprocessing and Multithreading both adds performance to the system. Multiprocessing is adding more number of or CPUs/processors to the system which increases the computing speed of the system.
- The Thread class that we mentioned earlier in this blog denotes an activity running in a separate thread of control. We can represent this activity either by passing a callable object to the constructor, ot by overriding the method run() in a subclass.
- A synchronization primitive, a primitive lock does not belong to a certain thread when locked. This is the lowest-level synchronization primitive we currently have in Python, and we implement it using the extension module _thread.

KNOWLEDGE CHECK

1. **A thread can be created by using …….. class.**
   a. MultiThread
   b. Thread
   c. Threading
   d. SuperThread
2. **Which Java feature enables to handle multiple tasks simultaneously?**
   a. Class and Object
   b. Platform Independent
   c. Dynamic Object Initialization
   d. Multi Threading
3. **Which method is used to schedule a thread for execution?**
   a. start()
   b. init()
   c. run()
   d. resume()
4. **What is called when a function is defined inside a class?**
   a. Module
   b. Class
   c. Another Function
   d. Method
5. **Which of the following is the use of id() function in python?**
   a. Id returns the identity of the object
   b. Every object does not have a unique id
   c. All of the mentioned
   d. None of the mentioned
6. **_____ makes it possible for two or more activities to execute in parallel on a single processor.**
   a. Multithreading
   b. Threading
   c. SingleThreading
   d. Both Multithreading and SingleThreading

7. In _____ an object of type Thread in the namespace System.Threading represents and controls one thread.

a. . PY
b. .SAP
c. .NET
d. .EXE

8. The method will be executed once the thread's _____ method is called.

a. EventBegin
b. EventStart
c. Begin
d. Start

REVIEW QUESTIONS

1. What do you understand by multithreading?
2. How to use the thread module to create threads.
3. How to use Python Multi-threading Modules for Thread Implementation?
4. Differentiate between multiprocessing and multithreading.
5. Explain the functions in python multithreading.
6. Write the difference between Lock objects and RLock objects.

Check Your Result

1. (b) 2. (d) 3. (d) 4. (d) 5. (a)
6. (a) 7. (c) 8. (d)

REFERENCES

1. Bini, Ola (2007). Practical JRuby on Rails Web 2.0 Projects: bringing Ruby on Rails to the Java platform. Berkeley: APress. p. 3. ISBN 978-1-59059-881-8.
2. Holth, Moore (30 March 2014). "PEP 0441 -- Improving Python ZIP Application Support". Retrieved 12 November 2015.
3. Rauschmayer, Axel. "Chapter 3: The Nature of JavaScript; Influences". O'Reilly, Speaking JavaScript. Retrieved 15 May 2015.
4. Smith, Kevin D.; Jewett, Jim J.; Montanaro, Skip; Baxter, Anthony (2 September 2004). "PEP 318 – Decorators for Functions and Methods". Python Enhancement Proposals. Python Software Foundation. Retrieved 24 February 2012.

CHAPTER

8

OPERATIONS IN PYTHON

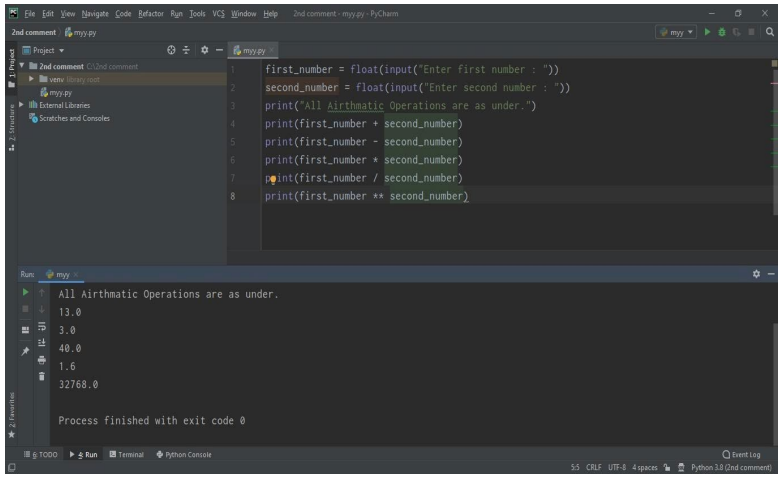
"Python is a truly wonderful language. When somebody comes up with a good idea it takes about 1 minute and five lines to program something that almost does what you want. Then it takes only an hour to extend the script to 300 lines, after which it still does almost what you want."

–Jack Jansen

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Discuss on python - decision making
2. Give an overview on loops, numbers, strings and lists in python
3. Learn about the python tuples
4. Define python - date & time



INTRODUCTION

Python is a powerful general-purpose programming language. It is used in web development, data science, creating software prototypes, and so on. Fortunately for

beginners, Python has simple easy-to-use syntax. This makes Python an excellent language to learn to program for beginners.

Python is an interpreted, object-oriented programming language similar to PERL that has gained popularity because of its clear syntax and readability. Python is said to be relatively easy to learn and portable, meaning its statements can be interpreted in a number of operating systems, including UNIX-based systems, Mac OS, MS-DOS, OS/2, and various versions of Microsoft Windows 98. Python was created by Guido van Rossum, a former resident of the Netherlands, whose favorite comedy group at the time was Monty Python's Flying Circus. The source code is freely available and open for modification and reuse. Python has a significant number of users.

Below are some facts about Python Programming Language:

- Python is currently the most widely used multi-purpose, high-level programming language.
- Python allows programming in Object-Oriented and Procedural paradigms.
- Python programs generally are smaller than other programming languages like Java. Programmers have to type relatively less and indentation requirement of the language, makes them readable all the time.
- Python language is being used by almost all tech-giant companies like – Google, Amazon, Facebook, Instagram, Dropbox, Uber… etc.
- The biggest strength of Python is huge collection of standard library which can be used for the following:
  - Machine Learning
  - GUI Applications (like Kivy, Tkinter, PyQt etc.)
  - Web frameworks like Django (used by YouTube, Instagram, Dropbox)
  - Image processing (like OpenCV, Pillow)
  - Web scraping (like Scrapy, BeautifulSoup, Selenium)
  - Test frameworks
  - Multimedia
  - Scientific computing
  - Text processing and many more.

8.1 PYTHON - DECISION MAKING

Decisions in a program are used when the program has conditional choices to execute a code block. Let's take an example of traffic lights, where different colors of lights lit up in different situations based on the conditions of the road or any specific rule.

It is the prediction of conditions that occur while executing a program to specify actions. Multiple expressions get evaluated with an outcome of either TRUE or FALSE. These are logical decisions, and Python also provides decision-making statements that to make decisions within a program for an application based on the user requirement.

There comes a point in your life where you need to decide what steps should be taken and based on that you decide your next decisions.

In programming, we often have this similar situation where we need to decide which block of code should be executed based on a condition.

Let's take a simple example.

Suppose you are writing a program for a game. So at every step, we need to take decisions like:

- If the user presses 'w' key, then the character will move forward.
- If the user presses 'spacebar', then the character will jump.
- If the character runs into an obstacle, then the game is over otherwise we continue playing.

Decisions like these are required everywhere in programming and they decide the direction of flow of program execution.

Python has the following decision-making statements:

- if statements
- if-else statements
- if-elif ladder
- Nested statements

Let's discuss these decision making statements in Python in detail.

Keyword

Decision making statements allow you to decide the order of execution of specific statements in your program. You can set up a condition and tell the compiler to take a particular action if the condition is met.

8.1.1 Python if Statement

if statement is the most simple form of **decision-making statement**. It takes an expression and checks if the expression evaluates to True then the block of code in if statement will be executed.

If the expression evaluates to False, then the block of code is skipped.

Syntax:

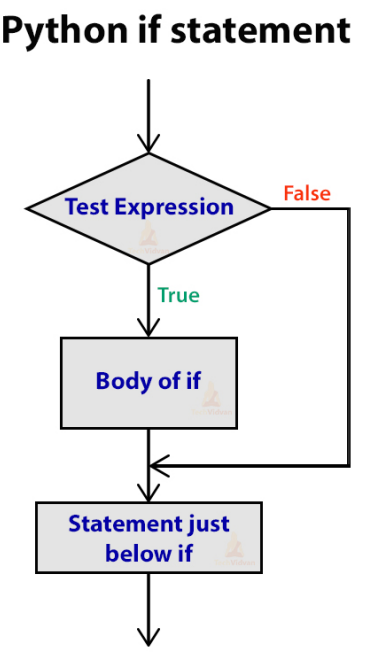
if (expression):

Statement 1

Statement 2

.

Statement n



Example 1:

a = 20 ; b = 20

if (a == b):

print("a and b are equal")

print("**If** block ended")

Output:

a and b are equal

If block ended

Example 2:

```
num = 5
if ( num >= 10):
print("num is greater than 10")
print("if block ended")
```

Output:

If block ended

In example 1, we see that the condition a==b evaluates to True. Therefore, the block of code inside if statement is executed.

In example 2, the condition evaluates to False, therefore, the print statement was not executed and the only statement that got executed was because it was outside the if block.

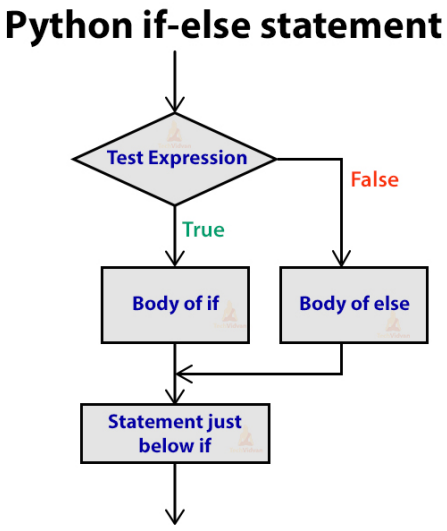
Note: Don't forget to add a colon(:) after if statement and indent the statements properly that are executed when a condition is True.

8.1.2 Python if-else Statement

From the name itself, we get the clue that the if-else statement checks the expression and executes the if block when the expression is True otherwise it will execute the else block of code. The else block should be right after if block and it is executed when the expression is False.

Syntax:

```
if( expression ):
Statement
else:
Statement
```



Example:

```
number1 = 20 ; number2 = 30
if(number1 >= number2 ):
print("number 1 is greater than number 2")
else:
print("number 2 is greater than number 1")
```

Output:

```
number 2 is greater than number 1
```

Note: Only one else statement is followed by an if statement. If you use two else statements after an if statement, then you get the following error.

Example:

```
if (5>10):
print(5)
else:
print(10)
```

```
else:

print("End")
```

Output:

```
SyntaxError: invalid syntax
```

8.1.3 Python if-elif ladder

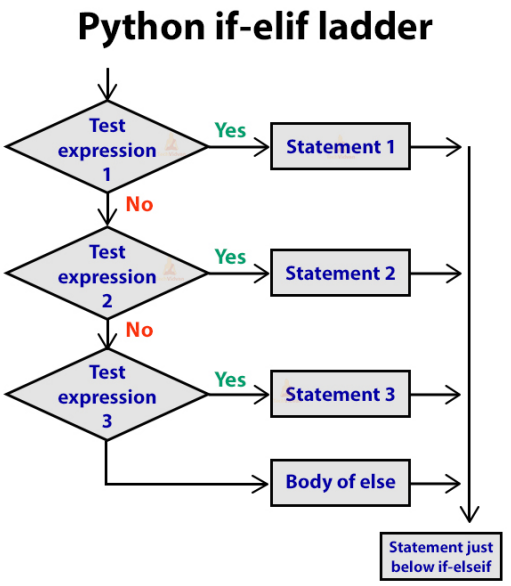
You might have heard of the else-if statements in other languages like C/C++ or Java.

In Python, we have an elif keyword to chain multiple conditions one after another. With elif ladder, we can make complex decision-making statements.

The elif statement helps you to check multiple expressions and it executes the code as soon as one of the conditions evaluates to True.

Syntax:

```
if( expression1 ):

statement

elif (expression2 ) :

statement

elif(expression3 ):

statement

.

.

else:

statement
```



Example:

```
print("Select your ride:")
print("1. Bike")
print("2. Car")
print("3. SUV")
choice = int( input() )
if( choice == 1 ):
print( "You have selected Bike" )
elif( choice == 2 ):
print( "You have selected Car" )
elif( choice == 3 ):
print( "You have selected SUV" )
else:
print("Wrong choice!")
```

Output:

Select your ride:

1. Bike

2. Car

3. SUV

3

You have selected SUV

Output:

Select your ride:

1. Bike

2. Car

3. SUV

10

Wrong choice!

Note: Core Python doesn't support switch-case statements that are available in other **programming languages** but we can use the elif ladder instead of switch cases.

8.1.4 Python Nested if statement

In very simple words, Nested if statements is an if statement inside another if statement. Python allows us to stack any number of if statements inside the block of another if statements. They are useful when we need to make a series of decisions.

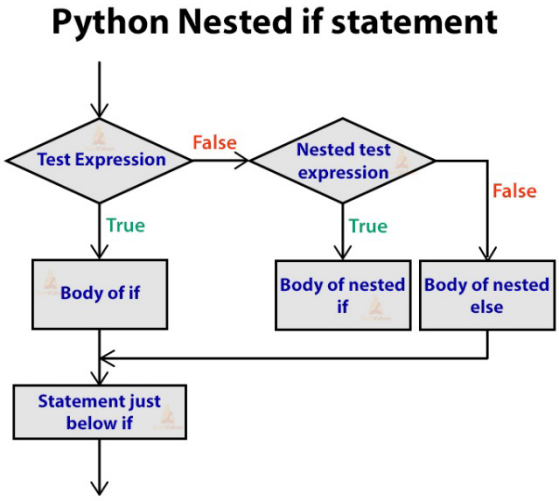
Syntax:

```
if (expression):

if(expression):

Statement of nested if
```

Keyword

Programming languages are one kind of computer language, and are used in computer programming to implement algorithms. Most programming languages consist of instructions for computers.

```
else:
Statement of nested if else
Statement of outer if
Statement outside if block
```



Example:

```
num1 = int( input())
num2 = int( input())
if( num1>= num2):
if(num1 == num2):
print(f'{num1} and {num2} are equal')
else:
print(f'{num1} is greater than {num2}')
else:
print(f'{num1} is smaller than {num2}')
```

Output 1:

10

20

10 is smaller than 20

Output 2:

5

5

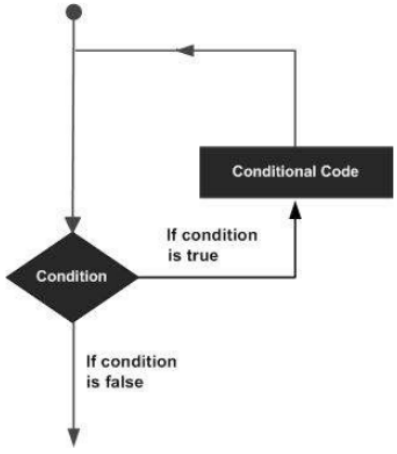
5 and 5 are equal

8.2 PYTHON - LOOPS

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	nested loops You can use one or more loop inside any another while, for or do..while loop.

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

for val in sequence:

loop body

Here, val is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

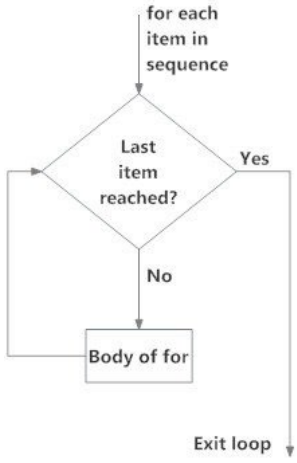


Figure. Flowchart of for Loop in Python.

Example: Python for Loop

```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

When you run the program, the output will be:

The sum is 48

Remember

If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

8.2.1 The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start, stop,step_size). step_size defaults to 1 if not provided.

The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and __getitem__ operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

```
print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))
```

Output

```
range(0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[2, 3, 4, 5, 6, 7]

[2, 5, 8, 11, 14, 17]
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index

for i in range(len(genre)):

   print("I like", genre[i])
```

Output

```
I like pop

I like rock

I like jazz
```

8.2.2 for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

The break keyword can be used to stop a for loop. In such cases, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

```
digits = [0, 1, 5]


for i in digits:
   print(i)
else:
   print("No items left.")
```

When you run the program, the output will be:

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.

This for...else statement can be used with the break keyword to run the else block only when the break keyword was not executed. Let's take an example:

```
# program to display student's marks from record
student_name = 'Soyuj'


marks = {'James': 90, 'Jules': 55, 'Arthur': 77}
```

```
for student in marks:
    if student == student_name:
        print(marks[student])
        break
else:
    print('No entry with that name found.')
```

Output

No entry with that name found.

8.2.3 Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

8.3 PYTHON - NUMBERS

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example −

var1 = 1

var2 = 10

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is –

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the **del** statement. For example –

del var

del var_a, var_b

Python supports four different numerical types –

- **int (signed integers)** – They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
- **long (long integers)** – Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
- **float (floating point real values)** – Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).
- **complex (complex numbers)** – are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. **Complex numbers** are not used much in Python programming.

Keyword

Complex number is a number that can be expressed in the form a + bi, where a and b are real numbers, and i is a symbol called the imaginary unit, and satisfying the equation $i^2 = -1$.

Examples

Here are some examples of numbers

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEL	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

- A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

8.3.1 Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type **int(x)** to convert x to a plain integer.
- Type **long(x)** to convert x to a long integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are **numeric expressions**

Keyword

Numeric expression is a combination of numeric elements (such as numbers, variables, and functions) and operators that evaluates to a numeric value. You can perform additional mathematical operations using math functions.

8.3.2 Mathematical Functions

Python includes following functions that perform mathematical calculations.

Sr.No.	Function & Returns (description)
1	abs(x) The absolute value of x: the (positive) distance between x and zero.
2	ceil(x) The ceiling of x: the smallest integer not less than x

3	cmp(x, y) -1 if x < y, 0 if x == y, or 1 if x > y
4	exp(x) The exponential of x: e^x
5	fabs(x) The absolute value of x.
6	floor(x) The floor of x: the largest integer not greater than x
7	log(x) The natural logarithm of x, for x> 0
8	log10(x) The base-10 logarithm of x for x> 0.
9	max(x1, x2,...) The largest of its arguments: the value closest to positive infinity
10	min(x1, x2,...) The smallest of its arguments: the value closest to negative infinity
11	modf(x) The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
12	pow(x, y) The value of x**y.
13	round(x [,n]) **x** rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
14	sqrt(x) The square root of x for x > 0

8.3.3 Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

Sr.No.	Function & Description
1	choice(seq) A random item from a list, tuple, or string.
2	randrange ([start,] stop [,step]) A randomly selected element from range(start, stop, step)
3	random() A random float r, such that 0 is less than or equal to r and r is less than 1
4	seed([x]) Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
5	shuffle(lst) Randomizes the items of a list in place. Returns None.
6	uniform(x, y) A random float r, such that x is less than or equal to r and r is less than y

8.3.4 Trigonometric Functions

Python includes following functions that perform trigonometric calculations.

Sr.No.	Function & Description
1	acos(x) Return the arc cosine of x, in radians.
2	asin(x) Return the arc sine of x, in radians.
3	atan(x) Return the arc tangent of x, in radians.

4	atan2(y, x) Return atan(y / x), in radians.
5	cos(x) Return the cosine of x radians.
6	hypot(x, y) Return the Euclidean norm, sqrt(x*x + y*y).
7	sin(x) Return the sine of x radians.
8	tan(x) Return the tangent of x radians.
9	degrees(x) Converts angle x from radians to degrees.
10	radians(x) Converts angle x from degrees to radians.

8.3.5 Mathematical Constants

The module also defines two mathematical constants −

Sr.No.	Constants & Description
1	**pi** The mathematical constant pi.
2	**e** The mathematical constant e.

8.4 PYTHON - STRINGS

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

var1 = 'Hello World!'

var2 = "Python Programming"

8.4.1 Accessing Values in Strings

Did You Know?

A string datatype is a datatype modeled on the idea of a formal string. Strings are such an important and useful datatype that they are implemented in nearly every programming language.

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example –

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

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

```
var1[0]:  H
var2[1:5]:  ytho
```

8.4.2 Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example –

```
#!/usr/bin/python

var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

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

```
Updated String :-  Hello Python
```

8.4.3 Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

8.4.4 String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell

in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

8.4.5 String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

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

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')

%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table –

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

8.4.6 Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive single or double quotes.

```
#!/usr/bin/python

para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""
print para_str
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n) –

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB (	) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
 ], or just a NEWLINE within
the variable assignment will also show up.
```

Keyword

Special character is one that is not considered a number or letter. Symbols, accent marks, and punctuation marks are considered special characters.

Raw strings do not treat the backslash as a **special character** at all. Every character you put into a raw string stays the way you wrote it –

```
#!/usr/bin/python

print 'C:\\nowhere'
```

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

```
C:\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'** as follows –

```
#!/usr/bin/python

print r'C:\\nowhere'
```

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

```
C:\\nowhere
```

8.4.7 Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following −

```
#!/usr/bin/python


print u'Hello, world!'
```

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

```
Hello, world!
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

8.4.8 Built-in String Methods

Python includes the following built-in methods to manipulate strings −

Sr.No.	Methods with Description
1	capitalize() Capitalizes first letter of string
2	center(width, fillchar) Returns a space-padded string with the original string centered to a total of width columns.
3	count(str, beg= 0,end=len(string)) Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	decode(encoding='UTF-8',errors='strict') Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	encode(encoding='UTF-8',errors='strict') Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.

6	endswith(suffix, beg=0, end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	expandtabs(tabsize=8) Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	find(str, beg=0 end=len(string)) Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	index(str, beg=0, end=len(string)) Same as find(), but raises an exception if str not found.
10	isalnum() Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	isalpha() Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	isdigit() Returns true if string contains only digits and false otherwise.
13	islower() Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	isnumeric() Returns true if a unicode string contains only numeric characters and false otherwise.
15	isspace() Returns true if string contains only whitespace characters and false otherwise.
16	istitle() Returns true if string is properly “titlecased” and false otherwise.

17	isupper() Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	join(seq) Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	len(string) Returns the length of the string
20	ljust(width[, fillchar]) Returns a space-padded string with the original string left-justified to a total of width columns.
21	lower() Converts all uppercase letters in string to lowercase.
22	lstrip() Removes all leading whitespace in string.
23	maketrans() Returns a translation table to be used in translate function.
24	max(str) Returns the max alphabetical character from the string str.
25	min(str) Returns the min alphabetical character from the string str.
26	replace(old, new [, max]) Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	rfind(str, beg=0,end=len(string)) Same as find(), but search backwards in string.
28	rindex(str, beg=0, end=len(string)) Same as index(), but search backwards in string.
29	rjust(width,[, fillchar]) Returns a space-padded string with the original string right-justified to a total of width columns.

30	rstrip() Removes all trailing whitespace of string.
31	split(str="", num=string.count(str)) Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	splitlines(num=string.count('\n')) Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	startswith(str, beg=0,end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	strip([chars]) Performs both lstrip() and rstrip() on string.
35	swapcase() Inverts case for all letters in string.
36	title() Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	translate(table, deletechars="") Translates string according to translation table str(256 chars), removing those in the del string.
38	upper() Converts lowercase letters in string to uppercase.
39	zfill (width) Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	isdecimal() Returns true if a unicode string contains only decimal characters and false otherwise.

8.5 PYTHON - LISTS

Lists are used to store multiple items in a single variable. Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5 ];

list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Remember

Data type is an attribute associated with a piece of data that tells a computer system how to interpret its value. Understanding data types ensures that data is collected in the preferred format and the value of each property is as expected.

8.5.1 Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
#!/usr/bin/python


list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5, 6, 7 ];
```

```
print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]
```

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

```
list1[0]:  physics

list2[1:5]:  [2, 3, 4, 5]
```

8.5.2 Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
#!/usr/bin/python


list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "

print list[2]

list[2] = 2001;

print "New value available at index 2 : "

print list[2]
```

Note − append() method is discussed in subsequent section.

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

```
Value available at index 2 :

1997

New value available at index 2 :

2001
```

8.5.3 Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
#!/usr/bin/python


list1 = ['physics', 'chemistry', 1997, 2000];

print list1

del list1[2];

print "After deleting value at index 2 : "

print list1
```

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

```
['physics', 'chemistry', 1997, 2000]

After deleting value at index 2 :

['physics', 'chemistry', 2000]
```

Note − remove() method is discussed in subsequent section.

8.5.4 Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

8.5.5 Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

L = [‘spam’, ‘Spam’, ‘SPAM!’]

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	[‘Spam’, ‘SPAM!’]	Slicing fetches sections

8.5.6 Built-in List Functions & Methods

Python includes the following list functions –

Sr.No.	Function with Description
1	cmp(list1, list2) Compares elements of both lists.
2	len(list) Gives the total length of the list.
3	max(list) Returns item from the list with max value.
4	min(list) Returns item from the list with min value.
5	list(seq) Converts a tuple into list.

Python includes following list methods

Sr.No.	Methods with Description
1	list.append(obj) Appends object obj to list

2	list.count(obj) Returns count of how many times obj occurs in list
3	list.extend(seq) Appends the contents of seq to list
4	list.index(obj) Returns the lowest index in list that obj appears
5	list.insert(index, obj) Inserts object obj into list at offset index
6	list.pop(obj=list[-1]) Removes and returns last object or obj from list
7	list.remove(obj) Removes object obj from list
8	list.reverse() Reverses objects of list in place
9	list.sort([func]) Sorts objects of list, use compare func if given

8.6 PYTHON - TUPLES

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value −

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

8.6.1 Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0];

print "tup2[1:5]: ", tup2[1:5];
```

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

```
tup1[0]:  physics

tup2[1:5]:  [2, 3, 4, 5]
```

8.6.2 Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

```
#!/usr/bin/python


tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');
```

```
# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print tup3;
```

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

```
(12, 34.56, 'abc', 'xyz')
```

8.6.3 Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the del statement. For example −

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup;

del tup;

print "After deleting tup : ";

print tup;
```

This produces the following result. Note an exception raised, this is because after del tup tuple does not exist any more −

```
('physics', 'chemistry', 1997, 2000)

After deleting tup :

Traceback (most recent call last):
  File "test.py", line 9, in <module>

    print tup;

NameError: name 'tup' is not defined
```

8.6.4 Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter –

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Remember

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

8.6.5 Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input –

L = ('spam', 'Spam', 'SPAM!')

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

8.6.6 No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for

tuples, etc., default to tuples, as indicated in these short examples −

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';

x, y = 1, 2;

print "Value of x , y : ", x,y;
```

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

```
abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2
```

8.6.7 Built-in Tuple Functions

Python includes the following tuple functions −

Sr.No.	Function with Description
1	cmp(tuple1, tuple2) Compares elements of both tuples.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

8.7 PYTHON - DATE & TIME

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

What is Tick?

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 00:00:00 hrs January 1, 1970(epoch).

There is a popular **time** module available in Python which provides functions for working with times, and for converting between representations. The function *time.time()* returns the current system time in ticks since 00:00:00 hrs January 1, 1970(epoch).

Example

```
#!/usr/bin/python

import time;  # This is required to include time module.


ticks = time.time()

print "Number of ticks since 12:00am, January 1, 1970:", ticks
```

This would produce a result something as follows −

Number of ticks since 12:00am, January 1, 1970: 7186862.73399

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

What is TimeTuple?

Many of Python's time functions handle time as a tuple of 9 numbers, as shown below −

Index	Field	Values
0	4-digit year	2008
1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 61 (60 or 61 are leap-seconds)
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)
8	Daylight savings	-1, 0, 1, -1 means library determines DST

The above tuple is equivalent to **struct_time** structure. This structure has following attributes −

Index	Attributes	Values
0	tm_year	2008
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6	tm_wday	0 to 6 (0 is Monday)
7	tm_yday	1 to 366 (Julian day)
8	tm_isdst	-1, 0, 1, -1 means library determines DST

8.7.1 Getting Current Time

To translate a time instant from a *seconds since the epoch* floating-point value into a time-tuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all nine items valid.

```
#!/usr/bin/python

import time;


localtime = time.localtime(time.time())

print "Local current time :", localtime
```

This would produce the following result, which could be formatted in any other presentable form −

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,

tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198, tm_
isdst=0)
```

8.7.2 Getting formatted time

You can format any time as per your requirement, but simple method to get time in readable format is asctime() −

```
#!/usr/bin/python

import time;


localtime = time.asctime( time.localtime(time.time()) )

print "Local current time :", localtime
```

This would produce the following result −

```
Local current time : Tue Jan 13 10:17:09 2009
```

8.7.3 Getting calendar for a month

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008) −

```
#!/usr/bin/python

import calendar


cal = calendar.month(2008, 1)

print "Here is the calendar:"

print cal
```

This would produce the following result −

Here is the calendar:

```
    January 2008

Mo Tu We Th Fr Sa Su
```

```
   1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

8.7.4 The time Module

There is a popular time module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods −

Sr.No.	Function with Description
1	time.altzone The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.
2	time.asctime([tupletime]) Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
3	time.clock( ) Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().
4	time.ctime([secs]) Like asctime(localtime(secs)) and without arguments is like asctime( )
5	time.gmtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0
6	time.localtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).

7	time.mktime(tupletime) Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.
8	time.sleep(secs) Suspends the calling thread for secs seconds.
9	time.strftime(fmt[,tupletime]) Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.
10	time.strptime(str,fmt='%a %b %d %H:%M:%S %Y') Parses str according to format string fmt and returns the instant in time-tuple format.
11	time.time( ) Returns the current time instant, a floating-point number of seconds since the epoch.
12	time.tzset() Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

Let us go through the functions briefly –

There are following two important attributes available with time module –

Sr.No.	Attribute with Description
1	**time.timezone** Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).
2	**time.tzname** Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

8.7.5 The calendar Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function.

Here is a list of functions available with the *calendar* module –

Sr.No.	Function with Description
1	**calendar.calendar(year,w=2,l=1,c=6)** Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.
2	**calendar.firstweekday( )** Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.
3	**calendar.isleap(year)** Returns True if year is a leap year; otherwise, False.
4	**calendar.leapdays(y1,y2)** Returns the total number of leap days in the years within range(y1,y2).
5	**calendar.month(year,month,w=2,l=1)** Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.
6	**calendar.monthcalendar(year,month)** Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.
7	**calendar.monthrange(year,month)** Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.
8	**calendar.prcal(year,w=2,l=1,c=6)** Like print calendar.calendar(year,w,l,c).
9	**calendar.prmonth(year,month,w=2,l=1)** Like print calendar.month(year,month,w,l).
10	**calendar.setfirstweekday(weekday)** Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).

11	**calendar.timegm(tupletime)** The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.
12	**calendar.weekday(year,month,day)** Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).

SUMMARY

- Python is a powerful general-purpose programming language. It is used in web development, data science, creating software prototypes, and so on.
- Python is an interpreted, object-oriented programming language similar to PERL that has gained popularity because of its clear syntax and readability.
- Decisions in a program are used when the program has conditional choices to execute a code block. Let's take an example of traffic lights, where different colors of lights lit up in different situations based on the conditions of the road or any specific rule.
- if statement is the most simple form of decision-making statement. It takes an expression and checks if the expression evaluates to True then the block of code in if statement will be executed.
- In very simple words, Nested if statements is an if statement inside another if statement. Python allows us to stack any number of if statements inside the block of another if statements. They are useful when we need to make a series of decisions.
- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

KNOWLEDGE CHECK

1. **How to output the string "May the odds favor you" in Python?**
   a. print("May the odds favor you")
   b. echo("May the odds favor you")
   c. System.out("May the odds favor you")
   d. printf("May the odds favor you")
2. **In which year was the Python 3.0 version developed?**
   a. 2005
   b. 2000
   c. 2010
   d. 2008
3. **Which character is used in Python to make a single line comment?**
   a. /
   b. //
   c. #
   d. ?
4. **Python is often described as a:**
   a. Batteries excluded language
   b. Gear included language
   c. Batteries included language
   d. Gear excluded language
5. **What do we use to define a block of code in Python language?**
   a. Indentation
   b. Key
   c. Brackets
   d. None of these
6. **Mathematical operations can be performed on a string in Python? State whether true or false:**
   a. False
   b. True
7. **Which one of the following is not a python's predefined data type?**
   a. List
   b. Dictionary

c. Tuple

d. Class

8. Which of the following has more precedence?

a. +

b. ()

c. /

d. –

REVIEW QUESTIONS

1. Discuss on python if statement.
2. Define the loop control statements.
3. What do you understand by the random number functions?
4. How to accessing values in strings?
5. What is unicode string?

Check Your Result

1. (a)	2. (d)	3. (c)	4. (c)
5. (a)	6. (a)	7. (d)	8. (b)

REFERENCES

1. Christopher Ramsay Holdgraf, Wendy de Heer, Brian N. Pasley, Jochem W. Rieger, Nathan Crone, Jack J. Lin, Robert T. Knight, and Frédéric E. Theunissen. Rapid tuning shifts in human auditory cortex enhance speech intelligibility. Nature Communications, 7(May):13654, 2016. URL: http://www.nature.com/doifinder/10.1038/ncomms13654, doi:10.1038/ncomms13654.
2. Fernando Perez, Brian E Granger, and John D Hunter. Python: an ecosystem for scientific computing. Computing in Science \\& Engineering, 13(2):13–21, 2011.
3. Hammond, Mark, and Andy Robinson. Python: Programming on Win32. Sebastopol, CA: O'Reilly, 2000.
4. J Gregory Caporaso, Justin Kuczynski, Jesse Stombaugh, Kyle Bittinger, Frederic D Bushman, Elizabeth K Costello, Noah Fierer, Antonio Gonzalez Pena, Julia K Goodrich, Jeffrey I Gordon, and others. Qiime allows analysis of high-throughput community sequencing data. Nature methods, 7(5):335–336, 2010.
5. John Stachurski and Takashi Kamihigashi. Stochastic stability in monotone economies. Theoretical Economics, 2014.

CHAPTER

9

PYTHON DATABASE PROGRAMMING

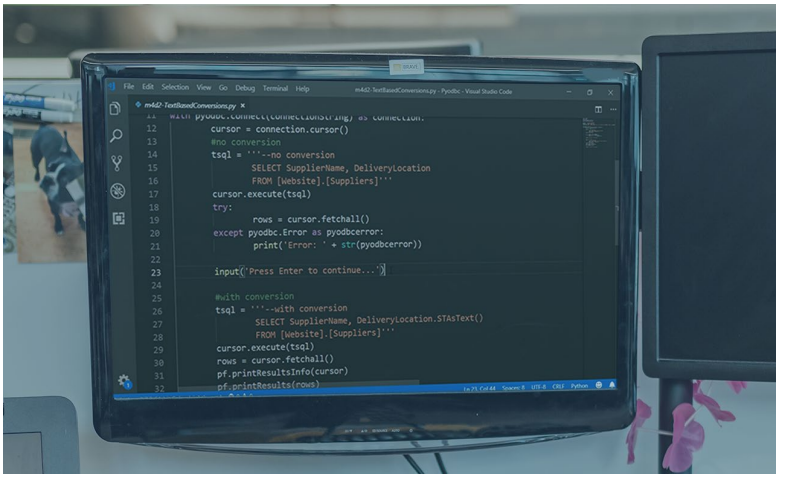
"Computing should be taught as a rigorous - but fun - discipline covering topics like programming, database structures, and algorithms. That doesn't have to be boring."

–Geoff Mulgan

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Learn about the DB-API (SQL-API) for python
2. Discuss on MySQL with Python
3. Creating, altering, and dropping a table
4. Inserting Records in Tables
5. Updating and Deleting Records from the Database

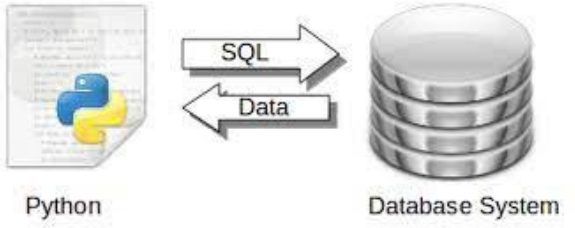


INTRODUCTION

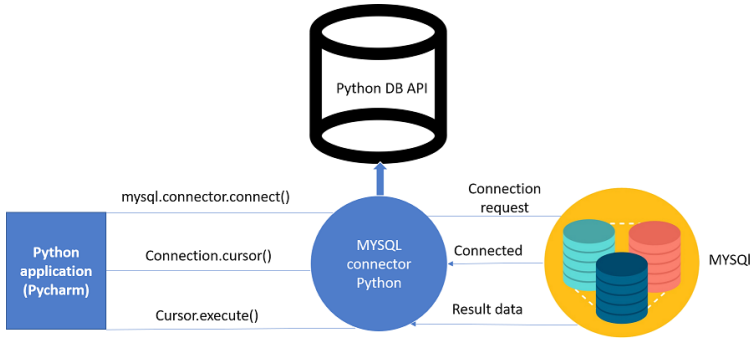
A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. The traditional term for database

software is "database management system". Database programs let users create and edit single files interactively at the keyboard. However, as soon as they want data in one file to automatically update another, programming has to be done. That is where the faint of heart take their leave, and the techies take over.

The database is a collection of organized information that can easily be used, managed, update, and they are classified according to their organizational approach.



From a construction firm to a stock exchange, every organization depends on large databases. These are essentially collections of tables, and' connected with each other through columns. These database systems support SQL, the Structured Query Language, which is used to create, access and manipulate the data. SQL is used to access data, and also to create and exploit the relationships between the stored data. Additionally, these databases support database normalization rules for avoiding redundancy of data. The Python programming language has powerful features for database programming. Python supports various databases like MySQL, Oracle, Sybase, PostgreSQL, etc. Python also supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements. For database programming, the Python DB API is a widely used module that provides a database application programming interface.



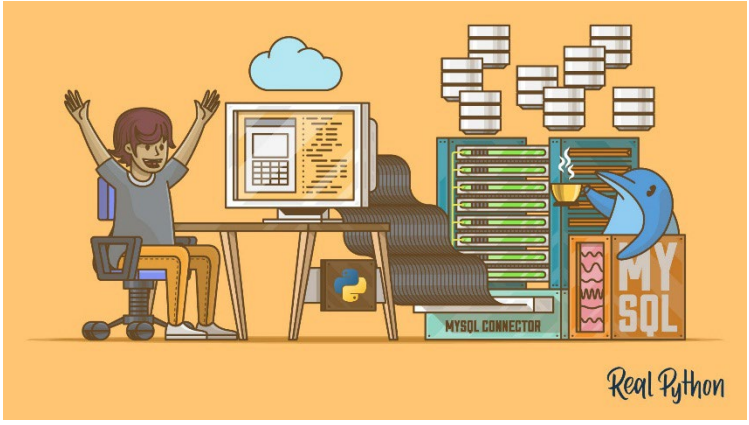
There are many good reasons to use Python for programming database applications:

- Programming in Python is arguably more efficient and faster compared to other languages.

- Python is famous for its portability.
- It is platform independent.
- Python supports SQL cursors.
- In many programming languages, the application developer needs to take care of the open and closed connections of the database, to avoid further exceptions and errors. In Python, these connections are taken care of.
- Python supports relational database systems.
- Python database APIs are compatible with various databases, so it is very easy to migrate and port database application interfaces.

9.1 DB-API (SQL-API) FOR PYTHON

Python DB-API is independent of any database engine, which enables you to write Python scripts to access any database engine. The Python DB API implementation for MySQL is MySQLdb. For PostgreSQL, it supports psycopg, PyGresQL and pyPgSQL modules. DB-API implementations for Oracle are dc_oracle2 and cx_oracle. Pydb2 is the DB-API implementation for DB2. Python's DB-API consists of connection objects, cursor objects, standard exceptions and some other module contents, all of which we will discuss.



9.1.1 Connection Objects

Connection objects create a connection with the database and these are further used for different transactions. These connection objects are also used as representatives of the database session.

A connection is created as follows:

>>>conn = MySQLdb.connect('library', user='suhas', password='python')

You can use a connection object for calling methods like commit(), rollback() and close() as shown below:

>>>cur = conn.cursor() //creates new cursor object for executing SQL statements

>>>conn.commit() //Commits the transactions

>>>conn.rollback() //Roll back the transactions

>>>conn.close() //closes the connection

>>>conn.callproc(proc,param) //call stored procedure for execution

>>>conn.getsource(proc) //fetches stored procedure code

9.1.2 Cursor objects

Keyword

Database server is a server which uses a database application that provides database services to other computer programs or to computers, as defined by the client–server model.

Cursor is one of the powerful features of SQL. These are objects that are responsible for submitting various SQL statements to a **database server**. There are several cursor classes in MySQLdb. cursors:

- BaseCursor is the base class for Cursor objects.
- Cursor is the default cursor class. CursorWarningMixIn, CursorStoreResultMixIn, CursorTupleRowsMixIn, and BaseCursor are some components of the cursor class.
- CursorStoreResultMixIn uses the mysql_store_result() function to retrieve result sets from the executed query. These result sets are stored at the client side.
- CursorUseResultMixIn uses the mysql_use_result() function to retrieve result sets from the executed query. These result sets are stored at the server side.

The following example illustrates the execution of SQL commands using cursor objects. You can use execute to execute SQL commands like SELECT. To commit all SQL operations you need to close the cursor as cursor.close().

```
>>>cursor.execute('SELECT * FROM books')
>>>cursor.execute('''SELECT * FROM books WHERE book_name = 'python' AND book_author = 'Mark Lutz' )
>>>cursor.close()
```

9.1.3 Error and Exception Handling in DB-API

Exception handling is very easy in the Python DB-API module. We can place warnings and error handling messages in the programs. Python DB-API has various options to handle this, like Warning, InterfaceError, DatabaseError, IntegrityError, InternalError, NotSupportedError, OperationalError and ProgrammingError. Let's take a look at them one by one:

- IntegrityError: Let's look at integrity error in detail. In the following example, we will try to enter duplicate records in the database. It will show an integrity error, _mysql_exceptions.IntegrityError, as shown below:

  ```
  >>> cursor.execute('insert books values (%s,%s,%s,%s)',('Py9098','Programming With Perl',120,100))
  Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 95, in execute
  return self._execute(query, args)
  File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 114, in _execute
  ```

self.errorhandler(self, exc, value)

raise errorclass, errorvalue

_mysql_exceptions.IntegrityError: (1062, "Duplicate entry 'Py9098' for key 1")

- OperationalError: If there are any operation errors like no databases selected, Python DB-API will handle this error as OperationalError, shown below:

>>> cursor.execute('Create database Library')

>>> q='select name from books where cost>=%s order by name'

>>>cursor.execute(q,[50])

Traceback (most recent call last):

File "<stdin>", line 1, in ?

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 95, in execute

return self._execute(query, args)

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 114, in _execute

self.errorhandler(self, exc, value)

File "/usr/lib/python2.3/site-packages/MySQLdb/connections.py", line 33, in defaulterrorhandler

raise errorclass, errorvalue

_mysql_exceptions.OperationalError: (1046, 'No Database Selected')

- ProgrammingError: If there are any programming errors like duplicate database creations, Python DB-API will handle this error as ProgrammingError, shown below:

>>> cursor.execute('Create database Library')

Traceback (most recent call last):>>> cursor.execute('Create database Library')

Traceback (most recent call last):

File "<stdin>", line 1, in ?

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 95, in execute

return self._execute(query, args)

File "/usr/lib/python2.3/site-packages/MySQLdb/cursors.py", line 114, in _execute

self.errorhandler(self, exc, value)

File "/usr/lib/python2.3/site-packages/MySQLdb/connections.py", line 33, in defaulterrorhandler

raise errorclass, errorvalue

_mysql_exceptions.ProgrammingError: (1007, "Can't create database 'Library'. Database exists")

9.1.4 Python and MySQL

Python and MySQL are a good combination to develop database applications. After starting the MySQL service on Linux, you need to acquire MySQLdb, a Python DB-API for MySQL to perform database operations. You can check whether the MySQLdb module is installed in your system with the following command:

>>>import MySQLdb

If this command runs successfully, you can now start writing scripts for your database.

To write database applications in Python, there are five steps to follow:

- Import the SQL interface with the following command:
  >>> import MySQLdb
- Establish a connection with the database with the following command:
  >>> conn=MySQLdb.connect(host='localhost',user='root',passwd='')
  …where host is the name of your host machine, followed by the username

and password. In case of the root, there is no need to provide a password.

- Create a cursor for the connection with the following command:

  >>>cursor = conn.cursor()

- Execute any SQL query using this cursor as shown below—here the outputs in terms of 1L or 2L show a number of rows affected by this query:

  >>> cursor.execute('Create database Library')

  1L // 1L Indicates how many rows affected

  >>> cursor.execute('use Library')

  >>>table='create table books(book_accno char(30) primary key, book_name char(50),no_of_copies int(5),price int(5))'

  >>> cursor.execute(table)

  0L

- Finally, fetch the result set and iterate over this result set. In this step, the user can fetch the result sets as shown below:

  >>> cursor.execute('select * from books')

  2L

  >>> cursor.fetchall()

  (('Py9098', 'Programming With Python', 100L, 50L), ('Py9099', 'Programming With Python', 100L, 50L))

In this example, the fetchall() function is used to fetch the result sets.

9.1.5 More SQL operations

We can perform all SQL operations with Python DB-API. Insert, delete, aggregate and update queries can be illustrated as follows.

- Insert SQL Query

  >>>cursor.execute('insert books values (%s,%s,%s,%s)',('Py9098','Programming With Python',100,50))

lL // Rows affected.

>>> cursor.execute('insert books values (%s,%s,%s,%s)',('Py9099','Programming With Python',100,50))

1L //Rows affected.

If the user wants to insert duplicate entries for a book's **accession number**, the Python DB-API will show an error as it is the primary key. The following example illustrates this:

```
>>> cursor.execute('insert books values (%s,%s,%s,%s)',('
Py9099','Programming With Python',100,50))

>>>cursor.execute('insert books values (%s,%s,%s,%s)',('
Py9098','Programming With Perl',120,100))

Traceback (most recent call last):

File "<stdin>", line 1, in ?

File "/usr/lib/python2.3/site-packages/MySQLdb/
cursors.py", line 95, in execute

return self._execute(query, args)

File "/usr/lib/python2.3/site-packages/MySQLdb/
cursors.py", line 114, in _execute

self.errorhandler(self, exc, value)

File "/usr/lib/python2.3/site-packages/MySQLdb/
connections.py", line 33, in defaulterrorhandler

raise errorclass, errorvalue

_mysql_exceptions.IntegrityError: (1062, "Duplicate
entry 'Py9098' for key 1")
```

- The Update SQL query can be used to update existing records in the database as shown below:

```
>>> cursor.execute('update books set price=%s where
no_of_copies<=%s',[60,101])

2L

>>> cursor.execute('select * from books')

2L

>>> cursor.fetchall()

(('Py9098', 'Programming With Python', 100L, 60L),
```

Keyword

Accession number is a sequential number assigned to each record or item as it is added to a to a library collection or database and which indicates the chronological order of its acquisition.

('Py9099', 'Programming With Python', 100L, 60L))

1. The Delete SQL query can be used to delete existing records in the database as shown below:

```
>>> cursor.execute('delete from books where no_of_copies<=%s',[101])

2L

>>> cursor.execute('select * from books')

0L

>>> cursor.fetchall()

()


>>> cursor.execute('select * from books')

3L

>>> cursor.fetchall() (('Py9099', 'Python-Cookbook', 200L, 90L), ('Py9098', 'Programming With Python', 100L, 50L), ('Py9097', 'Python-Nut shell', 300L, 80L))
```

- Aggregate functions can be used with Python DB-API in the database as shown below:

```
>>> cursor.execute('select * from books')

4L

>>> cursor.fetchall()

(('Py9099', 'Python-Cookbook', 200L, 90L), ('Py9098', 'Programming With Python', 100L, 50L), ('Py9097', 'Python-Nut shell', 300L, 80L), ('Py9096', 'Python-Nut shell', 400L, 90L))

>>> cursor.execute("select sum(price),avg(price) from books where book_name='Python-Nut shell'")

1L

>>> cursor.fetchall()

((170.0, 85.0),)
```

9.1.6 Python MySQL – Create Database

Python Database API (Application Program Interface) is the Database interface for the standard Python. This standard is adhered to by most Python Database interfaces. There are various Database servers supported by Python Database such as MySQL, GadFly, mSQL, PostgreSQL, Microsoft SQL Server 2000, Informix, Interbase, Oracle, Sybase etc. To connect with MySQL database server from Python, we need to import the mysql.connector interface.

Syntax:

CREATE DATABASE DATABASE_NAME

Example:

```
# importing required libraries
import mysql.connector


dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
  passwd ="gfg"
)
 # preparing a cursor object
cursorObject = dataBase.cursor()
 # creating database
cursorObject.execute("CREATE DATABASE geeks4geeks")
```

Output:

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| College            |
| geeks4geeks        |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.01 sec)
```

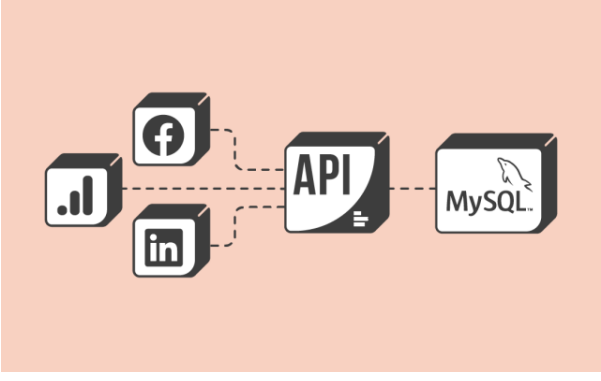
The above program illustrates the creation of MySQL database geeks4geeks in which host-name is localhost, the username is user and password is gfg.

Let's suppose we want to create a table in the database, then we need to connect to a database. Below is a program to create a table in the geeks4geeks database which was created in the above program.

9.2 MYSQL WITH PYTHON

MySQL is one of the most popular database management systems (DBMSs) on the market today. It ranked second only to the Oracle DBMS in this year's DB-Engines Ranking. As most software applications need to interact with data in some form, programming languages like Python provide tools for storing and accessing these data sources.

You'll be able to efficiently integrate a MySQL database with a Python application. You'll develop a small MySQL database for a movie rating system and learn how to query it directly from your Python code.



Structured Query Language is a standard Database language which is used to create, maintain and retrieve the relational database. Following are some interesting facts about SQL.

- SQL is case insensitive. But it is a recommended practice to use keywords (like SELECT, UPDATE, CREATE, etc) in capital letters and use user defined things (liked table name, column name, etc) in small letters.
- We can write comments in SQL using "–" (double hyphen) at the beginning of any line.
- SQL is the programming language for relational databases (explained below) like MySQL, Oracle, Sybase, SQL Server, Postgre, etc. Other non-relational databases (also called NoSQL) databases like MongoDB, DynamoDB, etc do not use SQL
- Although there is an ISO standard for SQL, most of the implementations slightly vary in syntax. So we may encounter queries that work in SQL Server but do not work in MySQL.

9.2.1 Comparing MySQL to Other SQL Databases

SQL stands for Structured Query Language and is a widely used programming language for managing relational databases. You may have heard of the different flavors of SQL-based DBMSs. The most popular ones include MySQL, PostgreSQL, SQLite, and SQL Server. All of these databases are compliant with the SQL standards but with varying degrees of compliance.

Being open source since its inception in 1995, MySQL quickly became a market leader among SQL solutions. MySQL is also a part of the Oracle ecosystem. While its core functionality is completely free, there are some paid add-ons as well. Currently, MySQL is used by all major tech firms, including Google, LinkedIn, Uber, Netflix, Twitter, and others.

Apart from a large open source community for support, there are many other reasons for MySQL's success:

- **Ease of installation:** MySQL was designed to be user-friendly. It's quite straightforward to set up a MySQL

MySQL is free and open-source software under the terms of the GNU General Public License, and is also available under a variety of proprietary licenses. MySQL was owned and sponsored by the Swedish company MySQL AB, which was bought by Sun Microsystems (now Oracle Corporation).

database, and several widely available third-party tools, like phpMyAdmin, further streamline the setup process. MySQL is available for all major operating systems, including Windows, macOS, Linux, and Solaris.

- **Speed:** MySQL holds a reputation for being an exceedingly fast database solution. It has a relatively smaller footprint and is extremely scalable in the long run.
- **User privileges and security:** MySQL comes with a script that allows you to set the password security level, assign admin passwords, and add and remove user account privileges. This script uncomplicates the admin process for a web hosting user management portal. Other DBMSs, like PostgreSQL, use config files that are more complicated to use.

While MySQL is famous for its speed and ease of use, you can get more advanced features with PostgreSQL. Also, MySQL isn't fully SQL compliant and has certain functional limitations, like no support for FULL JOIN clauses.

You might also face some issues with concurrent reading and writing in MySQL. If your software has many users writing data to it at once, then PostgreSQL might be a more suitable choice.

Note: For a more in-depth comparison of MySQL and PostgreSQL in a real-world context, check out Why Uber Engineering Switched from Postgres to MySQL.

SQL Server is also a very popular DBMS and is known for its reliability, efficiency, and security. It's preferred by companies, especially in the banking domain, who regularly deal with large traffic workloads. It's a commercial solution and is one of the systems that are most compatible with Windows services.

In 2010, when Oracle acquired Sun Microsystems and MySQL, many were worried about MySQL's future. At the time, Oracle was MySQL's biggest competitor. Developers feared that this was a hostile takeover from Oracle with the aim of destroying MySQL.

Several developers led by Michael Widenius, the original author of MySQL, created a fork of the MySQL code base and laid the foundation of MariaDB. The aim was to secure access to MySQL and keep it free forever.

To date, MariaDB remains fully GPL licensed, keeping it completely in the public domain. Some features of MySQL, on the other hand, are available only with paid licenses. Also, MariaDB provides several extremely useful features that aren't supported by MySQL server, like distributed SQL and columnar storage. You can find more differences between MySQL and MariaDB listed on MariaDB's website.

MySQL uses a very similar syntax to the Standard SQL. There are, however, some notable differences mentioned in the official documentation.

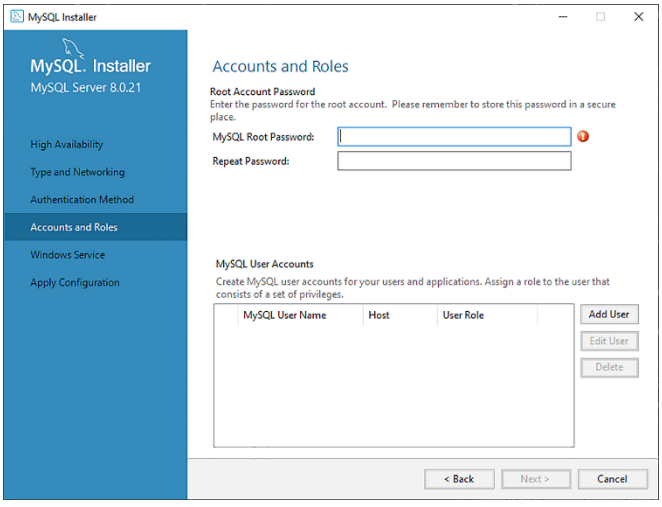
9.2.2 Installing MySQL Server and MySQL Connector/Python

Now, to start working through this tutorial, you need to set up two things: a MySQL server and a MySQL connector. MySQL server will provide all the services required for handling your database. Once the server is up and running, you can connect your Python application with it using MySQL Connector/Python.

Installing MySQL Server

The official documentation details the recommended way to download and install MySQL server. You'll find instructions for all popular operating systems, including Windows, macOS, Solaris, Linux, and many more.

For Windows, the best way is to download MySQL Installer and let it take care of the entire process. The installation manager also helps you configure the security settings of the MySQL server. On the Accounts and Roles page, you need to enter a password for the **root** (admin) account and also optionally add other users with varying privileges:



While you must specify credentials for the root account during setup, you can modify these settings later on. Remember the hostname, username, and password as these will be required to establish a connection with the MySQL server later on.

Although you only need the MySQL server for this tutorial, you can also set up other helpful tools like MySQL Workbench using these installers. If you don't want to install MySQL directly in your operating system, then deploying MySQL on Linux with Docker is a convenient alternative.

Installing MySQL Connector/Python

A database driver is a piece of software that allows an application to connect and interact with a database system. Programming languages like Python need a special driver before they can speak to a database from a specific vendor.

These drivers are typically obtained as third-party modules. The Python Database API (DB-API) defines the standard interface with which all Python database drivers must comply. These details are documented in PEP 249. All Python database drivers, such as sqlite3 for SQLite, psycopg for PostgreSQL, and MySQL Connector/Python for MySQL, follow these implementation rules.

Note: MySQL's official documentation uses the term connector instead of driver. Technically, connectors are associated only with connecting to a database, not interacting with it. However, the term is often used for the entire database access module comprising the connector *and* the driver.

To maintain consistency with the documentation, you'll see the term connector whenever MySQL is mentioned.

Many popular programming languages have their own database API. For example, Java has the Java Database Connectivity (JDBC) API. If you need to connect a Java application to a MySQL database, then you need to use the MySQL JDBC connector, which follows the JDBC API.

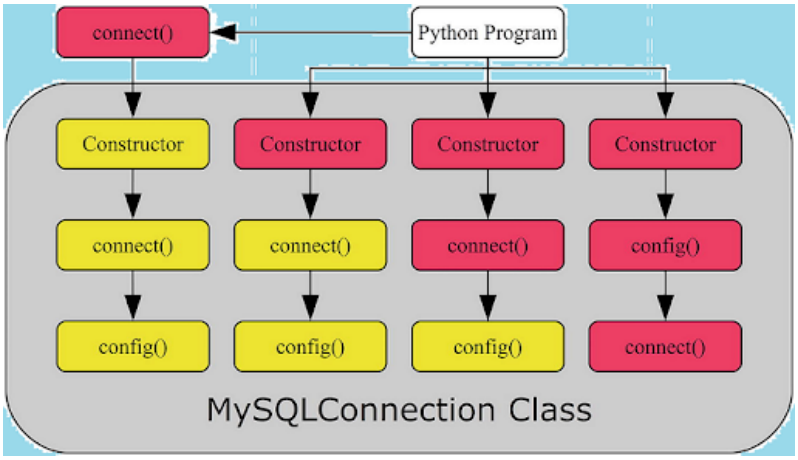
Similarly, in Python you need to install a Python MySQL connector to interact with a MySQL database. Many packages follow the DB-API standards, but the most popular among them is MySQL Connector/Python. You can get it with pip:

- $ pip install mysql-connector-python

pip installs the connector as a third-party module in the currently active virtual environment. It's recommended that you set up an isolated virtual environment for the project along with all the dependencies.

To test if the installation was successful, type the following command on your Python terminal:

```
>>>

>>> import mysql.connector
```



If the above code executes with no errors, then mysql.connector is installed and ready to use. If you encounter any errors, then make sure you're in the correct virtual environment and you're using the right Python interpreter.

Make sure that you're installing the correct mysql-connector-python package, which is a pure-Python implementation. Beware of similarly named but now depreciated connectors like mysql-connector.

9.2.3 Establishing a Connection with MySQL Server

MySQL is a server-based database management system. One server might contain multiple databases. To interact with a database, you must first establish a connection with the server. The general workflow of a Python program that interacts with a MySQL-based database is as follows:

- Connect to the MySQL server.
- Create a new database.
- Connect to the newly created or an existing database.
- Execute a SQL query and fetch results.
- Inform the database if any changes are made to a table.
- Close the connection to the MySQL server.

Did You Know?

Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is a Java-based data access technology used for Java database connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation.

This is a generic workflow that might vary depending on the individual application. But whatever the application might be, the first step is to connect your database with your application.

Establishing a Connection

The first step in interacting with a MySQL server is to establish a connection. To do this, you need connect() from the mysql.connector module. This function takes in parameters like host, user, and password and returns a MySQLConnection object. You can receive these credentials as input from the user and pass them to connect():

```
from getpass import getpass

from mysql.connector import connect, Error


try:
   with connect(
      host="localhost",
      user=input("Enter username: "),
      password=getpass("Enter password: "),
   ) as connection:
      print(connection)
except Error as e:
   print(e)
```

The code above uses the entered login credentials to establish a connection with your MySQL server. In return, you get a MySQLConnection object, which is stored in the connection variable. From now on, you'll use this variable to access your MySQL server.

There are several important things to notice in the code above:

- You should always deal with the exceptions that might be raised while establishing a connection to the MySQL server. This is why you use a try … except block to catch and print any exceptions that you might encounter.
- You should always close the connection after you're done accessing the database. Leaving unused open connections can lead to several unexpected

errors and performance issues. The above code takes advantage of a context manager using with, which abstracts away the connection cleanup process.

- You should *never hard-code your login credentials*, that is, your username and password, directly in a Python script. This is a bad practice for deployment and poses a serious security threat. The code above prompts the user for login credentials. It uses the built-in getpass module to hide the password. While this is better than hard-coding, there are other, more secure ways to store sensitive information, like using environment variables.

You've now established a connection between your program and your MySQL server, but you still need to either create a new database or connect to an existing database inside the server.

Creating a New Database

Here you established a connection with your MySQL server. To create a new database, you need to execute a SQL statement:

```
CREATE DATABASE books_db;
```

The above statement will create a new database with the name books_db.

In MySQL, it's mandatory to put a semicolon (;) at the end of a statement, which denotes the termination of a query. However, MySQL Connector/Python automatically appends a semicolon at the end of your queries, so there's no need to use it in your Python code.

To execute a SQL query in Python, you'll need to use a cursor, which abstracts away the access to database records. MySQL Connector/Python provides you with

the MySQLCursor class, which instantiates objects that can execute MySQL queries in Python. An instance of the MySQLCursor class is also called a cursor.

cursor objects make use of a MySQLConnection object to interact with your MySQL server. To create a cursor, use the .cursor() method of your connection variable:

```
cursor = connection.cursor()
```

The above code gives you an instance of the MySQLCursor class.

A query that needs to be executed is sent to cursor.execute() in string format. In this particular occasion, you'll send the CREATE DATABASE query to cursor.execute():

```
from getpass import getpass
from mysql.connector import connect, Error


try:
   with connect(
      host="localhost",
      user=input("Enter username: "),
      password=getpass("Enter password: "),
   ) as connection:
      create_db_query = "CREATE DATABASE online_movie_rating"
      with connection.cursor() as cursor:
         cursor.execute(create_db_query)
except Error as e:
   print(e)
```

After executing of the code above, you'll have a new database called online_movie_rating in your MySQL server.

The CREATE DATABASE query is stored as a string in the create_db_query variable and then passed to cursor.execute() for execution. The code uses a context manager with the **cursor object** to handle the cleanup process.

You might receive an error here if a database with the same name already exists in your server. To confirm this, you can display the name of all databases in your server. Using the same MySQLConnection object from earlier, execute the SHOW DATABASES statement:

```
>>>

>>> show_db_query = "SHOW DATABASES"

>>> with connection.cursor() as cursor:

...     cursor.execute(show_db_query)

...     for db in cursor:

...         print(db)

...

('information_schema',)

('mysql',)

('online_movie_rating',)

('performance_schema',)

('sys',)
```

The above code prints the names of all the databases currently in your MySQL server. The SHOW DATABASES command also outputs some databases that you didn't create in your server, like information_schema, performance_schema, and so on. These databases are generated automatically by the MySQL server and provide access to a variety of database metadata and MySQL server settings.

You created a new database in this section by executing the CREATE DATABASE statement. In the next section, you'll see how to connect to a database that already exists.

Keyword

Cursor object is an object that is used to make the connection for executing SQL queries. It acts as middleware between SQLite database connection and SQL query.

Connecting to an Existing Database

In the last section, you created a new database called online_movie_rating. However, you still haven't connected to it. In

many situations, you'll already have a MySQL database that you want to connect with your Python application.

You can do this using the same connect() function that you used earlier by sending an additional parameter called database:

```
from getpass import getpass

from mysql.connector import connect, Error


try:
   with connect(
      host="localhost",
      user=input("Enter username: "),
      password=getpass("Enter password: "),
      database="online_movie_rating",
   ) as connection:
      print(connection)
except Error as e:
   print(e)
```

The above code is very similar to the connection script that you used earlier. The only change here is an additional database parameter, where the name of your database is passed to connect(). Once you execute this script, you'll be connected to the online_movie_rating database.

9.3 CREATING, ALTERING, AND DROPPING A TABLE

In this section, you'll learn how to perform some basic DDL queries like CREATE, DROP, and ALTER with Python. You'll get a quick look at the MySQL database that you'll use in the rest of this tutorial. You'll also create all the tables required for the database and learn how to perform modifications on these tables later on.

9.3.1 Defining the Database Schema

You can start by creating a database schema for an online movie rating system. The database will consist of three tables:

- **movies** contains general information about movies and has the following attributes:
  - id
  - title
  - release_year
  - genre
  - collection_in_mil
- **reviewers** contains information about people who posted reviews or ratings and has the following attributes:
  - id
  - first_name
  - last_name
- **ratings** contains information about ratings that have been posted and has the following attributes:
  - movie_id (foreign key)
  - reviewer_id (foreign key)
  - rating

A real-world movie rating system, like IMDb, would need to store a bunch of other attributes, like emails, movie cast lists, and so on. If you want, you can add more tables and attributes to this database. But these three tables will suffice for the purpose of this tutorial.

The image below depicts the database schema:

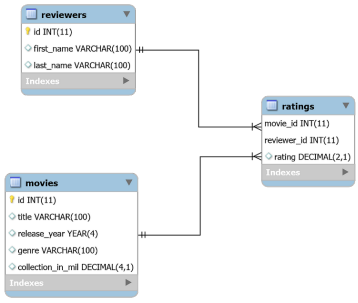


Figure 1. Schema Diagram for an Online Movie Rating System.

The tables in this database are related to each other, movies and reviewers will have a many-to-many relationship since one movie can be reviewed by multiple reviewers and one reviewer can review multiple movies. The ratings table connects the movies table with the reviewers table.

9.3.2 Creating Tables Using the CREATE TABLE Statement

Now, to create a new table in MySQL, you need to use the CREATE TABLE statement. The following MySQL query will create the movies table for your online_movie_rating database:

```
CREATE TABLE movies(
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100),
    release_year YEAR(4),
    genre VARCHAR(100),
    collection_in_mil INT
);
```

If you've looked at SQL statements before, then most of the above query might make sense. But there are some differences in the MySQL syntax that you should be aware of.

MySQL has a wide variety of data types for your perusal, including YEAR, INT, BIGINT, and so on. Also, MySQL uses the AUTO_INCREMENT keyword when a column value has to be incremented automatically on the insertion of new records.

To create a new table, you need to pass this query to cursor.execute(), which accepts a MySQL query and executes the query on the connected MySQL database:

```
create_movies_table_query = """
CREATE TABLE movies(
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    title VARCHAR(100),
    release_year YEAR(4),
    genre VARCHAR(100),
    collection_in_mil INT
)
"""
with connection.cursor() as cursor:
    cursor.execute(create_movies_table_query)
    connection.commit()
```

Now you have the movies table in your database. You pass create_movies_table_query to cursor.execute(), which performs the required execution.

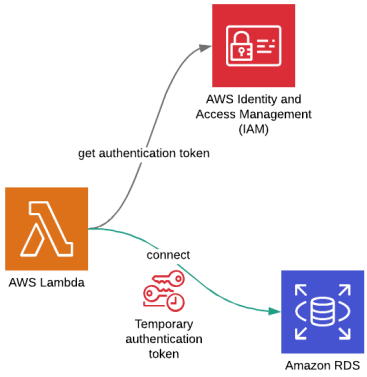
In MySQL, modifications mentioned in a transaction occur only when you use a COMMIT command in the end. Always call this method after every transaction to perform changes in the actual table.

As you did with the movies table, execute the following script to create the reviewers table:

```
create_reviewers_table_query = """
CREATE TABLE reviewers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(100),
    last_name VARCHAR(100)
)
"""
with connection.cursor() as cursor:
    cursor.execute(create_reviewers_table_query)
    connection.commit()
```

Remember

The connection variable refers to the MySQL Connection object that was returned when you connected to your database. Also, notice the connection.commit() statement at the end of the code. By default, your MySQL connector doesn't autocommit transactions.



If required, you could add more information about a reviewer, such as their email ID or demographic information. But first_name and last_name will serve your purpose for now.

Finally, you can create the ratings table using the following script:

```
create_ratings_table_query = """
CREATE TABLE ratings (
   movie_id INT,
   reviewer_id INT,
   rating DECIMAL(2,1),
   FOREIGN KEY(movie_id) REFERENCES movies(id),
   FOREIGN KEY(reviewer_id) REFERENCES reviewers(id),
   PRIMARY KEY(movie_id, reviewer_id)
)
"""
with connection.cursor() as cursor:
   cursor.execute(create_ratings_table_query)
   connection.commit()
```

The implementation of foreign key relationships in MySQL is slightly different and limited as compared to the standard SQL. In MySQL, both the parent and the child in the foreign key constraint must use the same storage engine.

A storage engine is the underlying software component that a database management system uses for performing SQL operations. In MySQL, storage engines come in two different flavors:

- Transactional storage engines are transaction safe and allow you to roll back transactions using simple commands like rollback. Many popular MySQL engines, including InnoDB and NDB, belong to this category.
- Nontransactional storage engines depend on elaborate manual code to undo statements committed on a database. MyISAM, MEMORY, and many other MySQL engines are nontransactional.

InnoDB is the default and most popular storage engine. It helps maintain data integrity by supporting foreign key constraints. This means that any CRUD operation on a foreign key is checked to ensure that it doesn't lead to inconsistencies across different tables.

Also, note that the ratings table uses the columns movie_id and reviewer_id, both foreign keys, jointly as the primary key. This step ensures that a reviewer can't rate the same movie twice.

You may choose to reuse the same cursor for multiple executions. In that case, all executions would become one atomic transaction rather than multiple separate transactions. For example, you can execute all CREATE TABLE statements with one cursor and then commit your transaction only once:

```
with connection.cursor() as cursor:

    cursor.execute(create_movies_table_query)

    cursor.execute(create_reviewers_table_query)

    cursor.execute(create_ratings_table_query)

    connection.commit()
```

The above code will first execute all three CREATE statements. Then it will send a COMMIT command to the MySQL server that commits your transaction. You can also use .rollback() to send a ROLLBACK command to the MySQL server and remove all data changes from the transaction.

9.3.3 Showing a Table Schema Using the DESCRIBE Statement

Now, that you've created all three tables, you can look at their schema using the following SQL statement:

DESCRIBE <table_name>;

To get some results back from the cursor object, you need to use cursor.fetchall(). This method fetches all rows from the last executed statement. Assuming you already have the MySQLConnection object in the connection variable, you can print out all the results fetched by cursor.fetchall():

```
>>>
>>> show_table_query = "DESCRIBE movies"
>>> with connection.cursor() as cursor:
...     cursor.execute(show_table_query)
...     # Fetch rows from last executed query
...     result = cursor.fetchall()
...     for row in result:
...         print(row)
...
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('title', 'varchar(100)', 'YES', '', None, '')
('release_year', 'year(4)', 'YES', '', None, '')
('genre', 'varchar(100)', 'YES', '', None, '')
('collection_in_mil', 'int(11)', 'YES', '', None, '')
```

Once you execute the above code, you should receive a table containing information about all the columns in movies table. For each column, you'll receive details like the column's data type, whether the column is a primary key, and so on.

9.3.4 Modifying a Table Schema Using the ALTER Statement

In the movies table, you have a column called collection_in_mil, which contains a movie's box office collection in millions of dollars. You can write the following MySQL statement to modify the data type of collection_in_mil attribute from INT to DECIMAL:

```
ALTER TABLE movies MODIFY COLUMN collection_in_mil DECIMAL(4,1);
```

DECIMAL(4,1) means a decimal number that can have a maximum of 4 digits, of which 1 is decimal, such as 120.1, 3.4, 38.0, and so on. After executing the ALTER TABLE statement, you can show the updated table schema using DESCRIBE:

```
>>>
>>> alter_table_query = """
... ALTER TABLE movies
... MODIFY COLUMN collection_in_mil DECIMAL(4,1)
... """
>>> show_table_query = "DESCRIBE movies"
>>> with connection.cursor() as cursor:
...     cursor.execute(alter_table_query)
...     cursor.execute(show_table_query)
...     # Fetch rows from last executed query
...     result = cursor.fetchall()
...     print("Movie Table Schema after alteration:")
...     for row in result:
...         print(row)
...
Movie Table Schema after alteration
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('title', 'varchar(100)', 'YES', '', None, '')
('release_year', 'year(4)', 'YES', '', None, '')
('genre', 'varchar(100)', 'YES', '', None, '')
('collection_in_mil', 'decimal(4,1)', 'YES', '', None, '')
```

As shown in the output, the collection_in_mil attribute is now of type DECIMAL(4,1). Also note that in the code above, you call cursor.execute() twice. But cursor.fetchall()

fetches rows from only the last executed query, which is the show_table_query.

9.3.5 Deleting Tables Using the DROP Statement

To delete a table, you need to execute the DROP TABLE statement in MySQL. Deleting a table is an *irreversible* process. If you execute the code below, then you'll need to call the CREATE TABLE query again to use the ratings table in the upcoming sections.

To delete the ratings table, send drop_table_query to cursor.execute():

```
drop_table_query = "DROP TABLE ratings"

with connection.cursor() as cursor:

  cursor.execute(drop_table_query)
```

If you execute the above code, you will have successfully deleted the ratings table.

9.4 INSERTING RECORDS IN TABLES

Earlier you created three tables in your database: movies, reviewers, and ratings. Now you need to populate these tables with data. This section will cover two different ways to insert records in the MySQL Connector for Python.

The first method, .execute(), works well when the number of records is small and the records can be hard-coded. The second method, .executemany(), is more popular and is better suited for real-world scenarios.

9.4.1 Using .execute()

The first approach uses the same cursor.execute() method that you've been using until now. You write the INSERT INTO query in a string and pass it to cursor.execute(). You can use this method to insert data into the movies table.

For reference, the movies table has five attributes:

- id
- title
- release_year
- genre
- collection_in_mil

You don't need to add data for id as the AUTO_INCREMENT automatically calculates id for you. The following script inserts records into the movies table:

```
insert_movies_query = """
```

```
INSERT INTO movies (title, release_year, genre, collection_in_mil)
VALUES
    ("Forrest Gump", 1994, "Drama", 330.2),
    ("3 Idiots", 2009, "Drama", 2.4),
    ("Eternal Sunshine of the Spotless Mind", 2004, "Drama", 34.5),
    ("Good Will Hunting", 1997, "Drama", 138.1),
    ("Skyfall", 2012, "Action", 304.6),
    ("Gladiator", 2000, "Action", 188.7),
    ("Black", 2005, "Drama", 3.0),
    ("Titanic", 1997, "Romance", 659.2),
    ("The Shawshank Redemption", 1994, "Drama",28.4),
    ("Udaan", 2010, "Drama", 1.5),
    ("Home Alone", 1990, "Comedy", 286.9),
    ("Casablanca", 1942, "Romance", 1.0),
    ("Avengers: Endgame", 2019, "Action", 858.8),
    ("Night of the Living Dead", 1968, "Horror", 2.5),
    ("The Godfather", 1972, "Crime", 135.6),
    ("Haider", 2014, "Action", 4.2),
    ("Inception", 2010, "Adventure", 293.7),
    ("Evil", 2003, "Horror", 1.3),
    ("Toy Story 4", 2019, "Animation", 434.9),
    ("Air Force One", 1997, "Drama", 138.1),
    ("The Dark Knight", 2008, "Action",535.4),
    ("Bhaag Milkha Bhaag", 2013, "Sport", 4.1),
```

```
    ("The Lion King", 1994, "Animation", 423.6),
    ("Pulp Fiction", 1994, "Crime", 108.8),
    ("Kai Po Che", 2013, "Sport", 6.0),
    ("Beasts of No Nation", 2015, "War", 1.4),
    ("Andadhun", 2018, "Thriller", 2.9),
    ("The Silence of the Lambs", 1991, "Crime", 68.2),
    ("Deadpool", 2016, "Action", 363.6),
    ("Drishyam", 2015, "Mystery", 3.0)
"""
with connection.cursor() as cursor:
    cursor.execute(insert_movies_query)
    connection.commit()
```

The movies table is now loaded with thirty records. The code calls connection.commit() at the end. It's crucial to call .commit() after preforming any modifications to a table.

9.4.2 Using .executemany()

The previous approach is more suitable when the number of records is fairly small and you can write these records directly into the code. But this is rarely true. You'll often have this data stored in some other file, or the data will be generated by a different script and will need to be added to the MySQL database.

This is where .executemany() comes in handy. It accepts two parameters:

- A **query** that contains placeholders for the records that need to be inserted
- A **list** that contains all records that you wish to insert

The following example inserts records for the reviewers table:

```
insert_reviewers_query = """
INSERT INTO reviewers
(first_name, last_name)
VALUES ( %s, %s )
```

```
"""
reviewers_records = [
    ("Chaitanya", "Baweja"),
    ("Mary", "Cooper"),
    ("John", "Wayne"),
    ("Thomas", "Stoneman"),
    ("Penny", "Hofstadter"),
    ("Mitchell", "Marsh"),
    ("Wyatt", "Skaggs"),
    ("Andre", "Veiga"),
    ("Sheldon", "Cooper"),
    ("Kimbra", "Masters"),
    ("Kat", "Dennings"),
    ("Bruce", "Wayne"),
    ("Domingo", "Cortes"),
    ("Rajesh", "Koothrappali"),
    ("Ben", "Glocker"),
    ("Mahinder", "Dhoni"),
    ("Akbar", "Khan"),
    ("Howard", "Wolowitz"),
    ("Pinkie", "Petit"),
    ("Gurkaran", "Singh"),
    ("Amy", "Farah Fowler"),
    ("Marlon", "Crafford"),
```

```
]

with connection.cursor() as cursor:

    cursor.executemany(insert_reviewers_query, reviewers_records)

    connection.commit()
```

In the script above, you pass both the query and the list of records as arguments to .executemany(). These records could have been fetched from a file or from the user and stored in the reviewers_records list.

The code uses %s as a placeholder for the two strings that had to be inserted in the insert_reviewers_query. Placeholders act as format specifiers and help reserve a spot for a variable inside a string. The specified variable is then added to this spot during execution.

You can similarly use .executemany() to insert records in the ratings table:

```
insert_ratings_query = """

INSERT INTO ratings

(rating, movie_id, reviewer_id)

VALUES ( %s, %s, %s)

"""

ratings_records = [

    (6.4, 17, 5), (5.6, 19, 1), (6.3, 22, 14), (5.1, 21, 17),

    (5.0, 5, 5), (6.5, 21, 5), (8.5, 30, 13), (9.7, 6, 4),

    (8.5, 24, 12), (9.9, 14, 9), (8.7, 26, 14), (9.9, 6, 10),

    (5.1, 30, 6), (5.4, 18, 16), (6.2, 6, 20), (7.3, 21, 19),

    (8.1, 17, 18), (5.0, 7, 2), (9.8, 23, 3), (8.0, 22, 9),

    (8.5, 11, 13), (5.0, 5, 11), (5.7, 8, 2), (7.6, 25, 19),

    (5.2, 18, 15), (9.7, 13, 3), (5.8, 18, 8), (5.8, 30, 15),

    (8.4, 21, 18), (6.2, 23, 16), (7.0, 10, 18), (9.5, 30, 20),

    (8.9, 3, 19), (6.4, 12, 2), (7.8, 12, 22), (9.9, 15, 13),
```

```
    (7.5, 20, 17), (9.0, 25, 6), (8.5, 23, 2), (5.3, 30, 17),
    (6.4, 5, 10), (8.1, 5, 21), (5.7, 22, 1), (6.3, 28, 4),
    (9.8, 13, 1)
]
with connection.cursor() as cursor:
    cursor.executemany(insert_ratings_query, ratings_records)
    connection.commit()
```

All three tables are now populated with data. You now have a fully functional online movie rating database. The next step is to understand how to interact with this database.

9.4.3 Reading Records from the Database

Until now, you've been building your database. Now it's time to perform some queries on it and find some interesting properties from this dataset. In this section, you'll learn how to read records from database tables using the SELECT statement.

Reading Records Using the SELECT Statement

To retrieve records, you need to send a SELECT query to cursor.execute(). Then you use cursor.fetchall() to extract the retrieved table in the form of a list of rows or records.

Try writing a MySQL query to select all records from the movies table and send it to .execute():

```
>>>
>>> select_movies_query = "SELECT * FROM movies LIMIT 5"
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     result = cursor.fetchall()
...     for row in result:
...         print(row)
...
```

(1, 'Forrest Gump', 1994, 'Drama', Decimal('330.2'))

(2, '3 Idiots', 2009, 'Drama', Decimal('2.4'))

(3, 'Eternal Sunshine of the Spotless Mind', 2004, 'Drama', Decimal('34.5'))

(4, 'Good Will Hunting', 1997, 'Drama', Decimal('138.1'))

(5, 'Skyfall', 2012, 'Action', Decimal('304.6'))

The result variable holds the records returned from using .fetchall(). It's a list of tuples representing individual records from the table.

In the query above, you use the LIMIT clause to constrain the number of rows that are received from the SELECT statement. Developers often use LIMIT to perform **pagination** when handling large volumes of data.

Keyword

Pagination is the process of dividing a document into discrete pages, either electronic pages or printed pages.

In MySQL, the LIMIT clause takes one or two nonnegative numeric arguments. When using one argument, you specify the maximum number of rows to return. Since your query includes LIMIT 5, only the first 5 records are fetched. When using both arguments, you can also specify the offset of the first row to return:

SELECT * FROM movies LIMIT 2,5;

The first argument specifies an offset of 2, and the second argument constrains the number of returned rows to 5. The above query will return rows 3 to 7.

You can also query for selected columns:

```
>>>

>>> select_movies_query = "SELECT title, release_year
FROM movies LIMIT 5"

>>> with connection.cursor() as cursor:

...     cursor.execute(select_movies_query)

...     for row in cursor.fetchall():

...         print(row)

...
```

('Forrest Gump', 1994)

('3 Idiots', 2009)

('Eternal Sunshine of the Spotless Mind', 2004)

('Good Will Hunting', 1997)

('Skyfall', 2012)

Now, the code outputs values only from the two specified columns: title and release_year.

Filtering Results Using the WHERE Clause

You can filter table records by specific criteria using the WHERE clause. For example, to retrieve all movies with a box office collection greater than $300 million, you could run the following query:

```
SELECT title, collection_in_mil

FROM movies

WHERE collection_in_mil > 300;
```

You can also use ORDER BY clause in the last query to sort the results from the highest to the lowest earner:

```
>>>

>>> select_movies_query = """

... SELECT title, collection_in_mil

... FROM movies

... WHERE collection_in_mil > 300

... ORDER BY collection_in_mil DESC

... """

>>> with connection.cursor() as cursor:

...     cursor.execute(select_movies_query)

...     for movie in cursor.fetchall():
```

```
...         print(movie)
...
('Avengers: Endgame', Decimal('858.8'))
('Titanic', Decimal('659.2'))
('The Dark Knight', Decimal('535.4'))
('Toy Story 4', Decimal('434.9'))
('The Lion King', Decimal('423.6'))
('Deadpool', Decimal('363.6'))
('Forrest Gump', Decimal('330.2'))
('Skyfall', Decimal('304.6'))
```

MySQL offers a plethora of string formatting operations like CONCAT for concatenating strings. Often, websites will show the movie title along with its release year to avoid confusion. To retrieve the titles of the top five grossing movies, concatenated with their release years, you can write the following query:

```
>>>
>>> select_movies_query = """
... SELECT CONCAT(title, " (", release_year, ")"),
...       collection_in_mil
... FROM movies
... ORDER BY collection_in_mil DESC
... LIMIT 5
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
```

```
...
('Avengers: Endgame (2019)', Decimal('858.8'))
('Titanic (1997)', Decimal('659.2'))
('The Dark Knight (2008)', Decimal('535.4'))
('Toy Story 4 (2019)', Decimal('434.9'))
('The Lion King (1994)', Decimal('423.6'))
```

If you don't want to use the LIMIT clause and you don't need to fetch all the records, then the cursor object has .fetchone() and .fetchmany() methods as well:

- **.fetchone()** retrieves either the next row of the result, as a tuple, or None if no more rows are available.
- **.fetchmany()** retrieves the next set of rows from the result as a list of tuples. It has a size argument, which defaults to 1, that you can use to specify the number of rows you need to fetch. If no more rows are available, then the method returns an empty list.

Try retrieving the titles of the five highest-grossing movies concatenated with their release years again, but this time use .fetchmany():

```
>>>
>>> select_movies_query = """
... SELECT CONCAT(title, " (", release_year, ")"),
...       collection_in_mil
... FROM movies
... ORDER BY collection_in_mil DESC
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchmany(size=5):
...         print(movie)
...     cursor.fetchall()
```

```
...

('Avengers: Endgame (2019)', Decimal('858.8'))

('Titanic (1997)', Decimal('659.2'))

('The Dark Knight (2008)', Decimal('535.4'))

('Toy Story 4 (2019)', Decimal('434.9'))

('The Lion King (1994)', Decimal('423.6'))
```

The output with .fetchmany() is similar to what you received when you used the LIMIT clause. You might have noticed the additional cursor.fetchall() call at the end. You do this to clean all the remaining results that weren't read by .fetchmany().

It's necessary to clean all unread results before executing any other statements on the same connection. Otherwise, an InternalError: Unread result found exception will be raised.

9.4.4 Handling Multiple Tables Using the JOIN Statement

If you found the queries in the last section to be quite straightforward, don't worry. You can make your SELECT queries as complex as you want using the same methods from the last section.

Let's look at some slightly more complex JOIN queries. If you want to find out the name of the top five highest-rated movies in your database, then you can run the following query:

```
>>>

>>> select_movies_query = """

... SELECT title, AVG(rating) as average_rating

... FROM ratings

... INNER JOIN movies

...     ON movies.id = ratings.movie_id

... GROUP BY movie_id

... ORDER BY average_rating DESC

... LIMIT 5
```

```
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
('Night of the Living Dead', Decimal('9.90000'))
('The Godfather', Decimal('9.90000'))
('Avengers: Endgame', Decimal('9.75000'))
('Eternal Sunshine of the Spotless Mind', Decimal('8.90000'))
('Beasts of No Nation', Decimal('8.70000'))
```

As shown above, Night of the Living Dead and The Godfather are tied as the highest-rated movies in your online_movie_rating database.

To find the name of the reviewer who gave the most ratings, write the following query:

```
>>>
>>> select_movies_query = """
... SELECT CONCAT(first_name, " ", last_name), COUNT(*) as num
... FROM reviewers
... INNER JOIN ratings
...     ON reviewers.id = ratings.reviewer_id
... GROUP BY reviewer_id
... ORDER BY num DESC
... LIMIT 1
... """
>>> with connection.cursor() as cursor:
```

```
... cursor.execute(select_movies_query)
... for movie in cursor.fetchall():
...     print(movie)
...
('Mary Cooper', 4)
```

Mary Cooper is the most frequent reviewer in this database. As seen above, it doesn't matter how complicated the query is because it's ultimately handled by the MySQL server. Your process for executing a query will always remain the same: pass the query to cursor.execute() and fetch the results using .fetchall().

9.5 UPDATING AND DELETING RECORDS FROM THE DATABASE

In this section, you'll be updating and deleting records from the database. Both of these operations can be performed on either a single record or multiple records in the table. You'll select the rows that need to be modified using the WHERE clause.

9.5.1 UPDATE Command

One of the reviewers in your database, Amy Farah Fowler, is now married to Sheldon Cooper. Her last name has now changed to Cooper, so you need to update your database accordingly. For updating records, MySQL uses the UPDATE statement:

```
update_query = """
UPDATE
  reviewers
SET
  last_name = "Cooper"
WHERE
  first_name = "Amy"
"""
with connection.cursor() as cursor:
```

```
cursor.execute(update_query)

connection.commit()
```

The code passes the update query to cursor.execute(), and .commit() brings the required changes to the reviewers table.

Note: In the UPDATE query, the WHERE clause helps specify the records that need to be updated. If you don't use WHERE, then all records will be updated!

Suppose you need to provide an option that allows reviewers to modify ratings. A reviewer will provide three values, movie_id, reviewer_id, and the new rating. The code will display the record after performing the specified modification.

Assuming that movie_id = 18, reviewer_id = 15, and the new rating = 5.0, you can use the following MySQL queries to perform the required modification:

```
UPDATE

   ratings

SET

   rating = 5.0

WHERE

   movie_id = 18 AND reviewer_id = 15;


SELECT *

FROM ratings

WHERE

   movie_id = 18 AND reviewer_id = 15;
```

Keyword

Python script is basically a file containing code written in Python. The file containing python script has the extension '.py' or can also have the extension '.pyw' if it is being run on a windows machine.

The above queries first update the rating and then display it. You can create a complete **Python script** that establishes a connection with the database and allows the reviewer to modify a rating:

```
from getpass import getpass
from mysql.connector import connect, Error

movie_id = input("Enter movie id: ")
reviewer_id = input("Enter reviewer id: ")
new_rating = input("Enter new rating: ")
update_query = """
UPDATE
   ratings
SET
   rating = "%s"
WHERE
   movie_id = "%s" AND reviewer_id = "%s";

SELECT *
FROM ratings
WHERE
   movie_id = "%s" AND reviewer_id = "%s"
""" % (
   new_rating,
   movie_id,
   reviewer_id,
   movie_id,
   reviewer_id,
```

```
)


try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
        database="online_movie_rating",
    ) as connection:
        with connection.cursor() as cursor:
            for result in cursor.execute(update_query, multi=True):
                if result.with_rows:
                    print(result.fetchall())
            connection.commit()
except Error as e:
    print(e)
```

Save this code to a file named modify_ratings.py. The above code uses %s placeholders to insert the received input in the update_query string. For the first time in this tutorial, you have multiple queries inside a single string. To pass multiple queries to a single cursor.execute(), you need to set the method's multi argument to True.

If multi is True, then cursor.execute() returns an iterator. Each item in the iterator corresponds to a cursor object that executes a statement passed in the query. The above code runs a for loop on this iterator and then calls .fetchall() on each cursor object.

Note: Running .fetchall() on all cursor objects is important. To execute a new statement on the same connection, you must ensure that there are no unread results from previous executions. If there are unread results, then you'll receive an exception.

If no result set is fetched on an operation, then .fetchall() raises an exception. To avoid this error, in the code above you use the cursor.with_rows property, which indicates whether the most recently executed operation produced rows.

While this code should solve your purpose, the WHERE clause is a prime target for web hackers in its current state. It's vulnerable to what is called a SQL injection attack, which can allow malicious actors to either corrupt or misuse your database.

Warning: Don't try the below inputs on your database! They will corrupt your table and you'll need to recreate it.

For example, if a user sends movie_id=18, reviewer_id=15, and the new rating=5.0 as input, then the output looks like this:

```
$ python modify_ratings.py

Enter movie id: 18

Enter reviewer id: 15

Enter new rating: 5.0

Enter username: <user_name>

Enter password:

[(18, 15, Decimal('5.0'))]
```

The rating with movie_id=18 and reviewer_id=15 has been changed to 5.0. But if you were hacker, then you might send a hidden command in your input:

```
$ python modify_ratings.py

Enter movie id: 18

Enter reviewer id: 15"; UPDATE reviewers SET last_name = "A

Enter new rating: 5.0

Enter username: <user_name>

Enter password:

[(18, 15, Decimal('5.0'))]
```

Again, the output shows that the specified rating has been changed to 5.0. What's changed?

The hacker sneaked in an update query while entering the reviewer_id. The update query, update reviewers set last_name = "A, changes the last_name of all records in the reviewers table to "A". You can see this change if you print out the reviewers table:

```
>>>
```

```
>>> select_query = """
... SELECT first_name, last_name
... FROM reviewers
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_query)
...     for reviewer in cursor.fetchall():
...         print(reviewer)
...
('Chaitanya', 'A')
('Mary', 'A')
('John', 'A')
('Thomas', 'A')
('Penny', 'A')
('Mitchell', 'A')
('Wyatt', 'A')
('Andre', 'A')
('Sheldon', 'A')
('Kimbra', 'A')
('Kat', 'A')
('Bruce', 'A')
('Domingo', 'A')
('Rajesh', 'A')
('Ben', 'A')
```

('Mahinder', 'A')

('Akbar', 'A')

('Howard', 'A')

('Pinkie', 'A')

('Gurkaran', 'A')

('Amy', 'A')

('Marlon', 'A')

The above code displays the first_name and last_name for all records in the reviewers table. The SQL injection attack corrupted this table by changing the last_name of all records to "A".

There's a quick fix to prevent such attacks. Don't add the query values provided by the user directly to your query string. Instead, update the modify_ratings.py script to send these query values as arguments to .execute():

```
from getpass import getpass

from mysql.connector import connect, Error


movie_id = input("Enter movie id: ")

reviewer_id = input("Enter reviewer id: ")

new_rating = input("Enter new rating: ")

update_query = """

UPDATE

   ratings

SET

   rating = %s

WHERE

   movie_id = %s AND reviewer_id = %s;
```

```
SELECT *
FROM ratings
WHERE
   movie_id = %s AND reviewer_id = %s
"""
val_tuple = (
   new_rating,
   movie_id,
   reviewer_id,
   movie_id,
   reviewer_id,
)

try:
   with connect(
      host="localhost",
      user=input("Enter username: "),
      password=getpass("Enter password: "),
      database="online_movie_rating",
   ) as connection:
      with connection.cursor() as cursor:
         for result in cursor.execute(update_query, val_tuple, multi=True):
            if result.with_rows:
```

```
            print(result.fetchall())

        connection.commit()

    except Error as e:

      print(e)
```

Notice that the %s placeholders are no longer in string quotes. Strings passed to the placeholders might contain some special characters. If necessary, these can be correctly escaped by the underlying library.

cursor.execute() makes sure that the values in the tuple received as argument are of the required data type. If a user tries to sneak in some problematic characters, then the code will raise an exception:

```
$ python modify_ratings.py

Enter movie id: 18

Enter reviewer id: 15"; UPDATE reviewers SET last_name = "A

Enter new rating: 5.0

Enter username: <user_name>

Enter password:

1292 (22007): Truncated incorrect DOUBLE value: '15";

UPDATE reviewers SET last_name = "A'
```

cursor.execute() will raise an exception if it finds any unwanted characters in the user input. You should use this approach whenever you incorporate user input in a query. There are other ways of preventing SQL injection attacks as well.

9.5.2 DELETE Command

Deleting records works very similarly to updating records. You use the DELETE statement to remove selected records.

Note: Deleting is an *irreversible* process. If you don't use the WHERE clause, then all records from the specified table will be deleted. You'll need to run the INSERT INTO query again to get back the deleted records.

It's recommended that you first run a SELECT query with the same filter to make sure that you're deleting the right records. For example, to remove all ratings given by reviewer_id = 2, you should first run the corresponding SELECT query:

```
>>>
>>> select_movies_query = """
... SELECT reviewer_id, movie_id FROM ratings
... WHERE reviewer_id = 2
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
(2, 7)
(2, 8)
(2, 12)
(2, 23)
```

The above code snippet outputs the reviewer_id and movie_id for records in the ratings table where reviewer_id = 2. Once you've confirmed that these are the records that you need to delete, you can run a DELETE query with the same filter:

```
delete_query = "DELETE FROM ratings WHERE reviewer_id = 2"

with connection.cursor() as cursor:

    cursor.execute(delete_query)

    connection.commit()
```

With this query, you remove all ratings given by the reviewer with reviewer_id = 2 from the ratings table.

9.5.3 Other Ways to Connect Python and MySQL

In this tutorial, you saw MySQL Connector/Python, which is the officially recommended means of interacting with a MySQL database from a Python application. There are two other popular connectors:

- mysqlclient is a library that is a close competitor to the official connector and is actively updated with new features. Because its core is written in C, it has better performance than the pure-Python official connector. A big drawback is that it's fairly difficult to set up and install, especially on Windows.
- MySQLdb is a legacy software that's still used in commercial applications. It's

written in C and is faster than MySQL Connector/Python but is available only for Python 2.

These connectors act as interfaces between your program and a MySQL database, and you send your SQL queries through them. But many developers prefer using an object-oriented paradigm rather than SQL queries to manipulate data.

Object-relational mapping (ORM) is a technique that allows you to query and manipulate data from a database directly using an object-oriented language. An ORM library encapsulates the code needed to manipulate data, which eliminates the need to use even a tiny bit of SQL.

Here are the most popular Python ORMs for SQL-based databases:

- SQLAlchemy is an ORM that facilitates communication between Python and other SQL databases. You can create different engines for different databases like MySQL, PostgreSQL, SQLite, and so on. SQLAlchemy is commonly used alongside the pandas library to provide complete data-handling functionality.
- peewee is a lightweight and fast ORM that's quick to set up. This is quite useful when your interaction with the database is limited to extracting a few records. For example, if you need to copy selected records from a MySQL database into a CSV file, then peewee might be your best choice.
- Django ORM is one of the most powerful features of Django and is supplied alongside the Django web framework. It can interact with a variety of databases such as SQLite, PostgreSQL, and MySQL. Many Django-based applications use the Django ORM for data modeling and basic queries but often switch to SQLAlchemy for more complex requirements.

Keyword

Object–relational mapping in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages.

ROLE MODEL

DAVID AXMARK: *ONE OF THE FOUNDERS OF MYSQL AB AND A DEVELOPER OF THE FREE DATABASE SERVER, MYSQL.*

Biography

David Axmark is one of the founders of MySQL AB and a developer of the free database server, MySQL. He has been involved with MySQL development from its beginning along with the fellow co-founder Michael Widenius. He studied at Uppsala University between 1980 and 1984

Scroll Down and find everything about the David Axmark you need to know, latest relationships update, Family and how qualified he is. David Axmark's Estimated Net Worth, Age, Biography, Career, Social media accounts i.e. Instagram, Facebook, Twitter, Family, Wiki. Also, learn details Info regarding the Current Net worth of David Axmark as well as David Axmark 's earnings, Worth, Salary, Property, and Income.

David Axmark, better known by the Family name David Axmark, is a popular Engineer. He was born on 28 May 1962, in Sweden.

David's estimated net worth, monthly and yearly salary, primary source of income, cars, lifestyle, and much more information have been updated below.

David who brought in $3 million and $5 million Networth David collected most of his earnings from his Yeezy sneakers While he had exaggerated over the years about the size of his business, the money he pulled in from his profession real–enough to rank as one of the biggest celebrity cashouts of all time. His Basic income source is mostly from being a successful Engineer.

SUMMARY

- A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions.
- The database is a collection of organized information that can easily be used, managed, update, and they are classified according to their organizational approach.
- Python DB-API is independent of any database engine, which enables you to write Python scripts to access any database engine.
- Connection objects create a connection with the database and these are further used for different transactions. These connection objects are also used as representatives of the database session.
- Cursor is one of the powerful features of SQL. These are objects that are responsible for submitting various SQL statements to a database server.
- Exception handling is very easy in the Python DB-API module. We can place warnings and error handling messages in the programs.
- Python and MySQL are a good combination to develop database applications. After starting the MySQL service on Linux, you need to acquire MySQLdb, a Python DB-API for MySQL to perform database operations.
- Python Database API (Application Program Interface) is the Database interface for the standard Python. This standard is adhered to by most Python Database interfaces.
- MySQL is one of the most popular database management systems (DBMSs) on the market today. It ranked second only to the Oracle DBMS in this year's DB-Engines Ranking. As most software applications need to interact with data in some form, programming languages like Python provide tools for storing and accessing these data sources.
- SQL Server is also a very popular DBMS and is known for its reliability, efficiency, and security. It's preferred by companies, especially in the banking domain, who regularly deal with large traffic workloads.

KNOWLEDGE CHECK

1. **What is the output of this code?**

   a,b=1,0

   a. a=a^b

   b. b=a^b

   c. a=a^b

   d. print(a)

2. **What is the value of this expression?**

   22**3**1**

   a. 12

   b. 64

   c. 128

   d. 256

   e. This code will raise an exception

3. **Which of the following is generally used for performing tasks like creating the structure of the relations, deleting relation?**

   a. DML(Data Manipulation Language)

   b. Query

   c. Relational Schema

   d. DDL(Data Definition Language)

4. **Which of the following provides the ability to query information from the database and insert tuples into, delete tuples from, and modify tuples in the database?**

   a. DML(Data Manipulation Language)

   b. DDL(Data Definition Language)

   c. Query

   d. Relational Schema

5. **Which one of the following given statements possibly contains the error?**

   a. select * from emp where empid = 10003;

   b. select empid from emp where empid = 10006;

   c. select empid from emp;

   d. select empid where empid = 1009 and Lastname = 'GELLER';

6. **What do you mean by one to many relationships?**
   a. One class may have many teachers
   b. One teacher can have many classes
   c. Many classes may have many teachers
   d. Many teachers may have many classes
7. **A Database Management System is a type of _____software.**
   a. It is a type of system software
   b. It is a kind of application software
   c. It is a kind of general software
   d. Both A and C
8. **The term "FAT" is stands for_____**
   a. File Allocation Tree
   b. File Allocation Table
   c. File Allocation Graph
   d. All of the above

REVIEW QUESTIONS

1. What is cursor objects?
2. How to create database in python MySQL?
3. How to installing MySQL server and MySQL connector/python?
4. Access and establishing a connection with MySQL server.
5. How to modifying a table schema using the alter statement?

Check Your Result

1. (a)	2. (d)	3. (d)	4. (a)
5. (d)	6. (b)	7. (a)	8. (b)

REFERENCES

1. Abraham Silberschatz, Henry F. Korth and S. Sudarshan, Database System Concepts, McGraw-Hill Education (Asia), Fifth Edition, 2006.
2. C. J. Date, A. Kannan and S. Swamynathan, An Introduction to Database Systems, Pearson Education, Eighth Edition, 2009.
3. Patrick O'Neil and Elizabeth O'Neil, Database Principles, Programming and Performance, Harcourt Asia Pte. Ltd., First Edition, 2001.
4. Peter Norton, Alex Samuel, David Aitel, Eric Foster-Johnson, Leonard Richardson, Jason Diamond, Aleatha Parker, Michael Roberts, Begining Python, 2005.
5. Peter Rob and Carlos Coronel, Database Systems Design, Implementation and Management, Thomson Learning-Course Technology, Seventh Edition, 2007.
6. Shio Kumar Singh, Database Systems Concepts, Designs and Application, Pearson Education, Second Edition, 2011.

Index

A

Accession number 287
aliasing 74
Anonymous functions 45
Application programming interface 280, 295
atomic grouping 174

B

Backreferences 187
Blueprint 136
Business information system 279, 332

C

C 1, 3, 4, 5, 7, 8, 9, 10, 20, 32
Character classes 185
characters 175, 179, 180, 181, 183, 190, 198
Chronological order 287
CMP(Context Management Protocol) 212, 213, 217
Computer Science 203
Condition object 215
Connection object 281, 332

D

Database engine 281, 332
Database management system 280, 295, 305
Database server 282, 289, 331, 332
Data Definition Language (DDL) 280
Data Manipulation Language (DML) 280
Data Query Statement 280
data structures 66, 67, 83
data type 16, 20, 22, 34
Decision-making statement 231, 235
Default arguments 41

E

Encapsulation 151, 154
Exception handling 283, 332
Execution 203, 208, 225

F

File objects 64
Files 64, 83, 86

G

General Public License (GPL) 3
Graphical User Interface (GUI) 9

H

High-level programming language 230
HTTP (Hypertext Transfer Protocol) 204

I

Image processing 230
Indexing 242, 259, 262, 266
Inheritance 141, 148
Instance attribute 137, 139, 147
Instance method 140
intersection operation 82

J

Java 1, 4, 5, 10, 32

K

keys 64, 65, 66, 67, 73, 85, 93, 94
Keyword arguments 41, 42

L

Library Reference 51
Literal characters 184
Loop exhaust 243
Loop statement 239, 244

M

Machine Learning 230
Mac OS 4, 6
Mappings 63
Match Function 176
Multimedia 230
Multiprocessing 204, 205
mutable data structure 68
MySQL 279, 280, 281, 285, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 302, 303, 305, 306, 308, 310, 313, 314, 316, 320, 321, 329, 330, 331, 332, 334

N

Nongreedy repetition 187
No-operation(NOP) 216
Normalization 280

O

Object-oriented programming language 230, 275
Object-Oriented style 2
open function 64, 84
Operating System 203
optional flags 176, 177

P

Parallel programming 202
Parent class 136, 141, 142, 143, 144, 145, 146, 147, 148, 149
Parentheses 136, 138, 162
pattern 174, 175, 176, 177, 178, 180, 183, 192, 194, 195
PERL 2
possessive quantifiers 174
programming languages 83, 86
Programming paradigm 135
Python 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 24, 25, 26, 27, 28, 31, 32, 34, 35, 63, 64, 65, 73, 76, 77, 78, 80, 83, 86, 87, 173, 174, 175, 179, 180, 181, 185, 187, 188, 190, 191, 192, 193, 195, 196
Python code 51, 290, 297
Python Database interface 289, 332
Python directory 7
Python implementation 204
Python interpreter 39, 41, 42, 47, 50, 51
Python magic 139
Python module 203
Python programming language 239
Python's module 54

Python's regex syntax 174
Python support 280, 281
Python threading 202

R

Rating system 290, 301
Recursion 213, 214, 216
regex 173, 174, 175, 176
Regular expressions 180
Return statement 39, 40, 46
re.UNICODE 175

S

Search Function 177
Self variable 137
Sets 63, 64, 75, 79, 82, 83
Software prototype 229, 275
Stock exchange 280
string 67, 73, 75, 76, 84, 85, 86, 87, 88, 94, 174, 175, 176, 177, 178, 179, 180, 181, 183, 184, 187, 188, 190, 193
Structured Query Language 280, 291
Synchronization 212, 213, 216, 220, 221
syntax 64, 68, 71, 83, 86

T

Thread 204, 206, 207, 208, 209, 210, 211, 219, 225
TypeError 146

U

Unicode properties 174

V

Variables 7, 16

Level: Beginner to Advanced
Subject: Computer and Information Science

Basic Computer Coding: Python

2nd Edition

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Its high-level built in data structures, combined with dynamic typing and dynamic binding; make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python has become one of the most popular programming languages in the world in recent years. It's used in everything from machine learning to building websites and software testing. It can be used by developers and non-developers alike.

This edition is organized into nine chapters. This is a comprehensive guide on how to get started in Python, why you should learn it and how you can learn it. This hands-on guide takes you through the language a step at a time, beginning with basic programming concepts including with functions, recursion, data structures, and object-oriented design.

ISBN 978-1-98467-604-7

BIBLIOTEX
Digital Library

www.3ge-learning.com
email: info@3ge-learning.com