Learning Python

B. Nagesh Rao

SECOND INTERNATIONAL EDITION

Published by:

CyberPlus Infotech Pvt. Ltd.
32/5, 8th Main, 11th Cross,
Malleswaram, Bengaluru – 560 003.
INDIA.

www.cyberplusindia.com

*For my **parents**, who always allowed me to follow my own path...*

*For the person I consider my programming guru: **Peter Norton**!*
His books and software were incredibly rich and useful...

*For my **teachers**, who put in their best to make an impact in my life,*
especially my first Computer Science teacher, Late Ms. P. S. Maithily,
whose encouragement and support allowed me
to excel in programming...

*For my **students**, who gave me the joy of knowing*
that I had made an impact in their lives...

About the Author

B. Nagesh Rao has more than 20 years of professional experience as a software developer, trainer, mentor and entrepreneur. He is a Principal Engineer and part of the Architecture Team in **Nokia**, where his job responsibilities include architecting and developing software solutions, training, coaching and mentoring. Not only does he design and develop software there, but also reviews the code written by others and leads the Clean Code campaign.

He is the founder of **CyberPlus Infotech Pvt. Ltd.**, a company specialising in open source technologies since the year 2000. He has developed more than 150 software applications, utilities, libraries and games. He has guided more than 1000 IT professionals and students in the development of various software projects. He has conducted more than 200 batches of corporate training and trained over 6500 candidates in more than 25 courses! He has authored several course materials and now plans to write several books.

He says he admires the simplicity and omnipotence of C, the support for OOP in C++, the ease of coding in and maintainability of Java, and the brevity of Perl! He adds that what makes Python a very appropriate language for him is that Python can be extended using C/C++, can run within the JVM and communicate with Java, and provides the brevity of Perl without it's obscurity! As a big fan of Object Oriented Programming paradigm, he loves the fact that Python is completely object-oriented!

The author can be reached at **nagesh.rao@cyberplusit.com**.

About the Publisher

CyberPlus Infotech Pvt. Ltd. is an open-source company founded in the year 2000, headquartered in Bengaluru, the silicon valley of India! The main activities of the company are software development, corporate training, and professional mentoring.

The team in CyberPlus has developed close to 200 software applications, utilities and libraries. They excel at developing object-oriented wrappers for various open-source libraries, like Textuality (a wrapper on ncurses for Textual User Interface development), GUICPP (a wrapper on GTK+ for GUI development) and GLOO (a wrapper on OpenGL for 3D graphics programming and rendering).

The trainers in CyberPlus have conducted more than 200 batches of corporate training for various clients in India like Nokia (formerly Nokia Siemens Networks, and prior to that, Siemens Communications Systems), IBAB (Institute for Bioinformatics and Applied Biotechnology), Delphi Technical Center, OpenStream Technologies, Alcatel Lucent (now acquired by Nokia), Altiostar (formerly Radio Mobile Access), Wipro Technologies, Comptel (now acquited by Nokia), Samsung India, Indecomm, Unisys and Stryker India.

They have conducted training programs on C, C++, STL, Java, UNIX/Linux, Advanced Linux, Advanced C, Advanced C++, Design Patterns, Perl, CGI, HTML, CSS, Javascript, XML, Python, Data Structures, Analysis and Design of Algorithms, Finite Automata and Formal Languages, Lex and Yacc, Microprocessors (8085 and 8086), Advanced Microprocessors (80286 to Pentium), Computer Graphics, OpenGL, Network Programming on Linux, SQL and MySQL, J2EE, etc.

They have mentored professionals and students to develop software at various client locations including Wipro Technologies, ISRO, Nokia Siemens Networks and Siemens Information Systems Limited.

CyberPlus Infotech Pvt. Ltd.
32/5, 8th Main, 11th Cross,
Malleswaram, Bengaluru – 560 003.
INDIA.

Email: info@cyberplusit.com

www.cyberplusindia.com

Preface to the First Edition

It was love at first sight and I was only 14!

I fell in love with computers the moment I saw them work! I remember, it all started when our seniors were demonstrating how to boot MS-DOS using a 5¼-inch bootable floppy disk and the first thing DOS would do upon booting was ask for the current date and time. I was stunned by the validation it would perform. I had a lot of questions:

How does it know when I type in a particular number?

How does it know that 1 and 2, when typed in succession makes 12?

How does it know what is the meaning of day, month and year?

How does it know what is a valid date and what isn't?

How does it know what is a leap year? And that a leap year has 29 days in February?

How can something that runs on electricity have intelligence?

One question led to another, and the answer to that led to yet another. And that's how my journey in programming began!

I first learnt BASIC, and when I felt there was something it could not do, I used to load binary instructions in my BASIC program using pre-assembled assembly code! There was no expert around to tell me that assembly language programming was not for kids!

Over time, I reluctantly learnt many more programming languages – some out of curiosity and some out of compulsion. I loved most of them, liked the rest of them and hated COBOL! I realised that learning programming languages was secondary; learning *programming* was primary! My strong programming base only made it a child's task to learn more programming languages.

I enjoyed programming so much that I could code, code and code for days together at a stretch, stopping only for eating, sleeping and other really important activities! I remember asking myself many a times, "*Why do I enjoy programming so much?*" I did find the answer, if I'm not mistaken!

Programming allowed me to play God!

Through programming, I could bring to life a machine that would faithfully, obediently, reliably, selflessly, correctly, accurately, quickly, tirelessly do as I commanded like a servant who never questions!

I was not even 16 years old when I designed my first programming language! After that I have invented and implemented so many custom languages that frankly I can't

even recollect all of them! This book's existence is actually related to the fact that I used to invent languages!

My fascination with gaming in particular led me to guide students to develop games in Java using a game development library we had invented in CyberPlus. I found the Java code too verbose and decided to invent a language specifically for writing and expressing game logic. I developed a C++ game engine and a game interpreter using C++ with Lex and Yacc, but what I realised later was that my game scripts looked shockingly close to Python scripts!

Later when we were developing a game development framework in Java, I realised that Python could be directly used for writing game scripts that could run within the JVM!

Right from my teenage I had always wanted to share my knowledge. While I did a lot of stuff – from teaching friends and classmates, writing articles, course materials and lab manuals – I still had one dream to be realised: writing books! This is my first book to be published, though I had made attempts in the past to write a book on C and data structures!

Let me be honest here: I have spent far more time on other programming languages like C, C++, Java and Perl than on Python. It is probably preordained that my first book to be published will be on Python rather than on those other languages. But as I pointed out earlier, programming is different from programming languages and I was able to learn and understand the nuances of Python quickly because I knew and understood programming in these other languages very well!

The proof of my deep understanding of programming concepts and languages was the feedback the participants would give after my training programs – even those who knew the language(s) beforehand felt they learnt many new things in my training! Many novices also appreciated the way concepts were taught to them. They felt I could explain even highly technical concepts in a simple way! It is that belief that I can contribute that led me to write this book, though there are many books and materials out there teaching Python in a dozen different ways!

I sincerely hope this book delivers what I expect it to: novices get to learn Python *well* and experienced programmers get to learn Python *fast*!

While there are many people I am thankful to in my life, and many such contributions from others have indirectly made this book possible, I will only name those who have been directly responsible in making this book what it is!

I thank Sumesh Kumar for being the first reviewer. I appreciate the fact that he has always been eager and more than willing to help, in any and every way he could.

I thank Santosh Manoharan for all his inputs and critical analysis. I admire his attention to details in the review.

Preface to the Second Edition

It has been an amazing journey since I wrote the first edition of Learning Python! A lot of things have happened, completely unexpected, related to the book! Let me share a few such experiences:

It was planned that we would be releasing the International Edition first, followed by the Indian Edition if needed. But we ended up releasing the Indian Edition first and the International Edition later! There were reasons for this, but most of these events happened spontaneously, forcing us to change our plans.

I had primarily targeted the programming community and my vast network of students whom I had trained. The people I had trained on various technologies were working in different places in various capacities. I thought they would love the opportunity to learn this “new” programming language in my style that they love! Of course, I wrote the book in such a simple manner that even school kids could read, understand and apply. But what I was unprepared for was the mass marketing to engineering colleges where Python was just being introduced in the syllabus!

In the process of conducting Python workshops for working professionals, I accidentally got in touch with a lecturer of mine, **Mr. Purushotham B. V.** He had taught me various subjects during engineering and we both have high regards for each other. With his help and support, I was able to reach out to about 50 engineering colleges across Karnataka. His business partner, **Prof. K. B. Shadaksharappa**, was earlier the HoD of CSE and ISE branches in PESIT, Bangalore, apart from having worked in the industry as well. He also actively assisted me in the endeavour and accompanied me to various colleges. I thank them both for all the help, and the time and efforts they have spent for popularising this book in engineering colleges.

This connection with colleges resulted in me getting involved in some kind of “Python awareness seminars” in various colleges, and that in turn ended up with me being added as a Member of Board of Studies for the Information Technology Engineering Course in Alliance University! I have always maintained that C is definitely not the right language for beginners, and Python should be introduced as the common programming language in the first year of engineering!

Special thanks are due to **Ms. Chethana R. Prasad**, who has always helped me in a lot many ways for more than a decade! As far as this book is concerned, she has helped reach out to colleges, conduct various programs there and of course, supply books there. She can’t be thanked enough in this page! Writing a book like this is no small task and her encouragement and support ensured that I consistently put in the effort that was necessary to bring the book out to the world! In the same context, I also thank **Sumesh Kumar** for all his critical reviews and suggestions. **Sumesh Kumar**, **Ganapati Hegde** and **Roopa Rao** have helped improve the previous edition to the

current form.

Many thanks to **Prof. Nisha Choudhary**, Asst. Prof., CSE, MS Engineering College, Bangalore for framing the model VTU question papers on Python, added at the end of this book. This will help students get an idea on what to expect in the examination and help score better marks.

Some colleges have already made the first edition of this book the recommended textbook for Python and we are thankful to those colleges! Not only have they made the right decision by introducing Python in their respective syllabii, but also evaluated our book and decided that this should be the book their students need to use!

Since we somehow had got associated with engineering colleges and their students were becoming our customers, we decided to expand the book and add additional chapters so that it completely covers the VTU syllabus. These additional chapters are extensions that help students apply Python to solve common real world problems. There are chapters added on Regular Expressions, Database Access and Parsing HTML, XML and JSON. I hope that readers find the second edition even more practical than the first and are able to accomplish more with Python!

As with the first edition of this book, my expectation remains the same: **novices get to learn Python *well* and experienced programmers get to learn Python *fast*!**

Reviews for the First Edition

Its a brilliant book, I'm loving it!

The author goes an extra mile to cover each concept and introduces it with reasoning followed by a sample program ending with its output.

Its a great coursework material that will not only get you started with confidence if a beginner, but also lets the concepts sink-in broadening the knowledge of an experienced programmer!

I've seen some other quick reads for python that assumes certain concepts, but the author makes sure he doesn't skip anything! The contents are very clear for each chapter if you wanted to go back and have a read again for any section.

Lastly but primarily, it also deals with installation on various platforms and setups which I found it extremely helpful. I felt like I was in a practical session being assisted by a tutor on the spot! :)

-Bhavana Ananda
Software Developer, University of Oxford

Content in the book is very good. Quality is excellent.

-Manjunath N.,
HoD, Computer Science & Engineering,
R. L. Jalappa Institute of Technology

Congratulations to the author and team of members who have put in all the efforts to bring the book on "Learning Python" a successful one.

Python programming book is found to be very simple and user friendly for the students as well as for the faculty members. It is contented with basic introductory chapters and an aim to show the differences between python and other languages. It also goes over how to use IDLE and run "Hello World" type program. The author explains how Python 'Compiles' the source code before running it. Explains about the syntax and runtime errors.

The early chapters gives an introduction to coding in python. Learn about comments, built-in functions, data types, arithmetic expressions, strings, control statements and few more topics. Also gives an idea about functions and modules, data structures like lists, Tuples, dictionaries, files and other topics.

The book is actually quite good. I feel like it covers all the topics well.

-Dr. Manjula Sanjay Koti
Prof & HoD, Dept. of MCA,
Member, BoE-VTU,
Dayananda Sagar Academy of Technology & Management

The book Learning Python by Nagesh Rao is a well structured book for beginners. The example programs at the end of the chapter exposes learners to practical programming. The chapter 10 practical python is highlight of the book, which shows various data structures implementation. Overall the book written in simple language and is great for Python learners.

-Vindhya N. S.
Asst. Prof. (Computer Science & Engg.), Dayananda Sagar Academy of Technology & Management

"Learning Python" has allowed me to delve into the world of Python with such ease that i just didn't realize the amount of knowledge that I have been able to garner in a span of a week. The book is just a master-class where the author's inquisitive mind has worked phenomenally well in expressing complex and the most intricate details in the most lucid way. It has done complete justice to all those individuals who intend to make Python their primary skill. I am humbly and genuinely indebted to the author in releasing this first edition and wish him all the very best in his professional career.

-Sumesh Kumar
Oracle

The book "Learning Python" by Mr. B. Nagesh Rao is a well written, well organised, cohesive book on the subject. It can be referred by the beginner as well as an intermediate programmer in Python. The flow of subject matter through the book is logical and elaborate. From basics of python, to its features, installation, data-types, variables, input output, control structures,derived data types like lists, tuples, dictionaries, sets, strings have all been explained in detail with a good number of examples. Functions, data structures like Stacks, Queues, Matrices have also been discussed appropriately. The Object Oriented Programming principles have been covered well with examples. The book provides nice coverage to Exception Handling and File Handling in Python along with an introductory chapter on Modules as well.

-Prof. Sunilkumar S. Manvi
Principal,
Reva Institute of Technology and Mgmt
& Director,
School of Computing and Information Technology, Reva University

Impeccable!! Difficult things made easy to learn and almost impossible to forget! Very well organised and structured. A great way of learning Python and programming as such. Typical of B. Nagesh Rao – needs no more to say about it. He himself is the best adjective to describe the book.

-Roopa Deepak
Freelance Trainer

This is the best book for one who wants to learn Python. The chapters are structured in a way that any one can learn. Nagesh sir has made sure that everything is covered in a proper way and his trademark teaching style is evident. I truly enjoyed the book.

-Naveen Kumar H.
Technical Test Lead, Infosys

The book was very good, easy to understand and you have covered almost all the topics from basics.

-Nisha Choudhary
Asst. Prof. (Computer Science & Engg.),
M. S. Engineering College

This book is very useful for the beginners to learn Python and for the experienced to know more. Also you can learn to use the resources available in Python for developing applications quickly. I have attended a 4 day training program based on this book and was able to develop small to medium size programs very efficiently. I appreciate the knowledge of the author in Python and the way he has presented it.

-Prof. K. B. Shadaksharappa
Former HoD (Dept. of CS & IS)
PES Institute of Technology
(now PES University)

I had the privilege of knowing Nagesh Rao for over 15 years, right from my college days through work. I have become a better software engineer learning from his programming and Linux course books and still use them to this day as a reference material at work. I was delighted to see his published book on Python.

This book, “Learning Python”, is wonderfully written to satisfy a new programmer and an experienced one at the same time. I had done scripting in Python a few years earlier but did not have the foundations as I had just picked it up for the brief work. But now after reading this book, I feel confident about the fundamentals and have started using more pythonic stuff like list comprehensions. I love the way the entire book is wound around great short examples, making it a very interesting and fun read.

I had started preparing for interviews in Python, but coming from the system domain, I was writing programs like a C programmer in Python. I did not have the time to read up a Python book just for the sake of interviews. But thanks to this book I could quickly read up and start programming in a more pythonic way and got more fluent for my interviews.

Overall I enjoyed this book thoroughly and look forward to reading more editions and books from this great and dear teacher.

- Santosh Manoharan
Software Engineer

Check These Out Too:

Book Website: www.learningpython.in

Our Website: www.cyberplusindia.com

Our Blog: www.cyberplusindia.com/blog

1 Introduction

In this chapter you will be able to:

- ☑ Figure out how this is the best book for you to learn Python
- ☑ Install Python on your computer, regardless of whether you use Windows or Linux
- ☑ Compare text editors and IDEs and select the right development environment for you
- ☑ Be able to launch the Python interpreter and quit it

INTRODUCTION

1.1 About This Book

Whether you are a novice to programming or a seasoned professional who is new to Python, this book has been designed for you!

Here are the top 10 salient features of this book that makes this book stand apart and makes it special for you:

1. **This book starts slowly!** The fundamental concepts are taught gently to ensure that you have a good foundation of basic concepts!
2. **This book is well organised!** The chapters, headings, subheadings and content within the chapters have been planned very carefully so that you conquer Python – chapter by chapter!
3. **This book teaches interactively!** The moment a new concept is taught, there is code that immediately follows so that you understand how it looks to the Python interpreter!
4. **This book teaches even when you can't practice!** Not only do we show you code immediately after teaching a concept, we also provide output from a real Python session so that you can imagine how Python reacts when you type in a piece of code!
5. **This book teaches good programming practices!** It is not only important to learn Python, but to also code like a professional. While it will definitely take a little bit of time to metamorphose from novice to professional, we show you best practices and pitfalls that will accelerate your journey!
6. **This book presents programs that solve real problems!** When it is time to apply Python, we show you constructive programs that demonstrate how to apply Python concepts!
7. **We analyse every bit of code!** Everything there is to analyse is analysed. Code snippets and programs are followed by output, which is then followed by analysis!
8. **We compare Python with other programming languages!** For the benefit of those readers who already know other programming languages like C/C++, Java or Perl, we provide tips that help them migrate to Python faster. These tips are in separate boxes to ensure that they don't disturb those readers who are not savvy with these languages!
9. **We have filtered out content!** While this might appear disadvantageous or counter-intuitive, we believe in presenting the most important concepts in detail and probably even skipping some concepts that you can live without! Call it the 80/20 rule if you will – we have decided to present in great detail those 20% of the features that you will use 80% of the time!

10. **Each heading and subheading is need based!** Our style of explaining a new concept is by first establishing a need. We believe this makes it easier for learners to understand not only what they are learning but also why!

Happy learning!

1.2 About Python

Python is a widely used high-level, general-purpose, interpreted, dynamic, object-oriented programming language. Its programming style emphasizes code readability and its syntax allows programmers to code their logic in much fewer lines than in languages like C++ and Java!

It is a fairly old language created by Guido Van Rossum in the late 1980s, when he was working on the Amoeba distributed operating system group and wanted to use an interpreted language like ABC that could access the Amoeba system calls. He decided to create a language that was extensible and that led to the design of a new programming language which was later named Python.

While most people at first might assume that the name 'Python' was derived from the reptile, it was actually adopted from a comedy series of the late seventies called “Monty Python's Flying Circus”.

The design began in the late 1980s and was first released in February 1991. Python 2.0 was released on 16 October 2000 and had many major new features like a cycle-detecting garbage collector and support for Unicode. With this release the development mode got changed and it became more transparent and community-backed.

Python 3.0 (which early in its development was commonly referred to as Python 3000 or py3k), a major, backwards-incompatible release, was released on 3 December 2008 after a long period of testing. Many of its major features have been back-ported to the backwards-compatible Python 2.6.x and 2.7.x version series

1.3 Installing Python

When downloading from Python's website, currently both the latest 3.x series of Python as well as 2.x series will be available, as older versions of Python are still in use. Unless you have strong reasons to act on the contrary, always download the latest version of Python listed there. **This book concentrates on Python 3.x, which in many ways will be different from Python 2.x**.

1.3.1 Installing Python on Windows

Download the latest version of Python from the official website: https://www.python.org/downloads/windows/.

The Windows version is provided as an MSI package. To install it manually, just double-click the file.

By design, Python installs to a directory with the version number embedded, e.g. Python version 2.7 will install at `C:\Python27\`, so that you can have multiple versions of Python on the same system without conflicts. Of course, only one interpreter can be the default application for Python file types. It also does not automatically modify the `PATH` environment variable, so that you always have control over which copy of Python is run.

Typing the full path name for a Python interpreter each time quickly gets tedious, so add the directories for your default Python version to the `PATH`. Assuming that your Python installation is in `C:\Python36\`, add this to your `PATH`:

```
C:\Python36\;C:\Python36\Scripts\
```

You do not need to install or configure anything else to use Python.

1.3.2 Installing Python on Linux

Python comes pre-installed on most Linux distributions, and is available as a package on all others. However there are certain features you might want to use that are not available in the version bundled in your your distribution. To do that, you can compile the latest version of Python from source. Follow the steps below :

Download the required python source from www.python.org/downloads/release as a “Gzipped source tarball”. You can either do that using a web browser or by using the in-built `wget` utility. If you plan to use the `wget` utility, you could use the browser to visit the above-mentioned website to determine the latest version available. We assume version 3.5.2 below:

```
$ wget https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
```

Untar the file downloaded and switch to the created directory:

```
$ tar -zxvf Python-3.5.2.tgz
$ cd Python-3.5.2
```

The next step is to configure the installation. Depending on your requirement, there are a couple of options. If you want this version of Python to be installed for all users of the system (and you have root privileges via `su` or `sudo`), run configure:

```
$ ./configure
```

The above command will configure Python to be installed in /usr/local. If you do not have root privileges or wish to make this a local installation for the current user, run configure in this manner:

```
$ ./configure –prefix=/some/other/directory
```

Typically in such cases, the specified directory would be a subdirectory within the current user's home directory. Here is an example of how the above command would actually be used:

```
$ ./configure –prefix=$HOME/py-352
```

The above program will configure Python to be installed in the directory `py-352` under the current user's home directory.

After the configuration is performed, run the make command:

```
$ make
```

The last step is to copy the final files to the destination location. If you had opted for a global installation, run `make install` as root:

```
$ sudo make install
```

The Python interpreter will now be available to all users.

If you had opted to install in a local directory under the current user's home directory, run `make install` without root privileges:

```
$ make install
```

The Python interpreter would now be available inside the bin subdirectory within the installation directory. Considering our example of local directory "`$HOME/py-352`", the interpreter will now be available at '`$HOME/py-352/bin/python3`'. You could add the directory to the `PATH` variable so that you could just type in `python3` henceforth. To do so, add this line to your `~/.bashrc` file:

```
PATH="$PATH:~/py-352/bin"
```

1.4 The Python Interpreter

Assuming you have taken care of changing the `PATH` environment variable as suggested in section 1.3, you can launch the Python 3.x interpreter by typing:

```
python3
```

You will be greeted by the following interpreter screen:

```
Python 3.5.2 (default, Sep 14 2016, 11:28:32)
[GCC 6.2.1 20160901 (Red Hat 6.2.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

The "`>>>`" is the **Python prompt** (technically the Python primary prompt) – an indication that the Python interpreter is ready to accept commands from you. You can type in any valid Python statement or expression here and the result is immediately displayed. We will use this feature to quickly test stuff when introducing a new concept. Of course, for bigger pieces of code, we will prefer a full-fledged Python script.

Let us first try out giving expressions at the Python prompt:

```
>>> 2+3
5
>>>
```

Observation:

1. `2+3` is a valid expression in Python. If it was an invalid expression, we would have got an error message.
2. The Python interpreter responds by printing the value of the expression, `5` in this case.
3. After finishing with the expression, the Python interpreter shows the Python prompt again, waiting for the next command from us.

Let us also type in a valid Python statement and see what happens:

```
>>> import math
>>>
```

Observation:

1. The statement "`import math`" is a valid statement in Python and means that we wish to import the `math` module. Had the statement been invalid, we would have received an error message.
2. Valid Python statements are executed by the Python interpreter immediately, though it's effect might not always be visible. The Python interpreter prints the result of the statement only if either the statement was actually an expression (in which case, the value of the expression is printed), or we had explicitly issued a Python statement to print something (using the `print()` function, for instance, which we will see later on).

Now that we know how to work on the Python interpreter, let us also learn how to quit the interpreter when we are done. Typing an end-of-file character (Control-D on Unix/Linux, Control-Z on Windows) at the Python primary prompt causes the interpreter to exit. You can also exit the interpreter by typing the `quit()` command:

```
>>> quit()
```

The interpreter's line-editing features include the following (provided the system supports them):

1. **Interactive editing** – the ability to edit the current line contents before submission by using cursor keys, backspace and delete keys.
2. **History substitution** – the ability to reproduce previously entered lines by using the up and down cursor keys.
3. **Code completion** – the ability to complete the current word when the user

presses tab (or produces a list of options when there are multiple possibilities).

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file.

A second way of starting the interpreter is:

```
python -c command [arg] ...
```

This launches the Python interpreter which executes the statement(s) in command, analogous to the Unix shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote command with quotes (preferably single quotes).

Here's an example:

```
$ python3 -c 'a=2+3; print(a)'
5
```

Observation:

1. We have given 2 statements: `a=2+3` and `print(a)`. Python understands they are separate statements because of the semi-colon (`;`) between them. The space given between them is only for readability.
2. Since the shell might have a problem with some special characters within the Python statements (like semi-colon and space), we prefer to enclose the set of statements within single quotes, the contents of which the shell will not process.
3. After executing the set of statements, the Python interpreter automatically terminates.

1.5 Python Editors and IDEs

1.5.1 Editors

There are many editors available for users to code in Python. The most popular ones which have been designed especially to suit the needs of Python programmers are:

1. **Sublime Text** – Supported on Linux, Windows, Mac OS X. (Website: https://www.sublimetext.com/3)
2. **SPE (Stani's Python Editor)** – Supported on Linux, Windows, Mac OS X (Website: https://sourceforge.net/projects/spe/)
3. **SciTE** - Supported on Windows & Linux (Website: http://www.scintilla.org/SciTEDownload.html)
4. **Geany** – Supported on Windows, Linux, MAC OS X (Website: https://www.geany.org/Download/Releases)

Besides the above, vim and emacs are extensively used for Python programming on Linux.

1.5.2 IDEs

The IDEs most suitable for Python are:

1. **PyCharm** – Supported on Windows, Linux, MAC OS X (Website: https://www.jetbrains.com/pycharm/download/)
2. **Wing IDE** - Supported on Windows, Linux, MAC OS X (Website: http://www.wingware.com/downloads)
3. **Pyzo** – Supported on Windows, Linux, MAC OS X (Website: http://test.pyzo.org/downloads.html)
4. **PyScripter** – Supported only on windows (Website: https://sourceforge.net/projects/pyscripter/)

Eclipse users who do not wish to switch to other IDEs, can make of PyDEV plugin for Python support.

1.6 Running Python Scripts

Section 1.4 introduced the fact that you can type in Python code and get it executed a line at a time directly from the interpreter. For bigger pieces of code, however, it is preferable to save the instructions in the form of a Python script (a file with a `.py` extension) and get it executed as a unit by the interpreter. Yet, there are 2 ways of doing so:

1. **Feeding the script to the Python interpreter.** In this method, you invoke the python interpreter and pass the filename (pathname if the script is not in the current working directory) of the script as an argument. For example, to run the script `test.py`, the command will probably be `python test.py`.
2. **Executing the script directly.** This technique is especially true in the UNIX/Linux world and is considered to be a better technique! In this method, we grant execute permission to the Python script, ensure that the first line of the Python script selects the Python interpreter (more on this soon), and execute the script directly by providing it's pathname. The first line of the

script should start with a "`#`" followed immediately by the pathname of the Python interpreter. For example, the first line of the script could be "`#/usr/bin/python`".

NOTE:

Even if we use the first method of execution, there is no harm in retaining the first line shown in method 2 as it would be considered to be merely a comment!

1.7 Questions

1. List the salient features of Python.
2. How did Python evolve? Give a brief description of its development from its early years to what it is now.
3. How would you install Python on the following platforms?
   - Windows
   - Linux
4. What alternative do you have to install Python locally assuming that you do not have super-user permission and do not intend to install it globally?
5. Assuming that you have a default Python interpreter installed (e.g. `/usr/bin/python`) and you wish to upgrade Python to a newer version, how would you go about setting it up? Is it advisable to overwrite the existing Python or have it as a separate entity?
6. How would you determine the version of Python that you are currently using?
7. How do you quit from a Python terminal in Windows & Linux? Is there a common-way of quitting on both platforms? If yes, what would that be?
8. What options do you find most useful when you look at the output of '`python --help`'?
9. At what stage would you prefer switching from a Python interpreter to writing a full-fledged progam?
10. When would you want to use the '`-c`' option with a Python interpreter?

SUMMARY

- Python is portable and Python scripts will work equally well both in Windows as well as Linux.
- The Python interpreter session is the best way to quickly type in code, check the results and learn!
- Bigger programs require to be written as separate Python scripts and executed. This will allow us to reuse the program whenever required.

2 Python Basics

In this chapter you will be able to:

- ☑ Understand how statements make up a Python script.
- ☑ Work with the basic data types of Python and operate upon them.
- ☑ Understand how to use variables in Python and differentiate between references and objects.
- ☑ Take input from the user and generate formatted output.

PYTHON BASICS

2.1 Our First Python Script

The first program in any language is typically a “Hello World” program, and let's be no different! Here is our first program that prints `Hello World` when executed:

`HelloWorld.py`

```
1.   print("Hello World")
```

Output:

```
Hello World
```

Type the above program in your favourite text editor or IDE (See section 1.5 for help in selecting) and execute it.

> **NOTE:**
>
> Of course, don't type in the line number (“`1.`”) in your programs! They have been given only for your reference, and will help when we discuss larger programs.

If you're done executing the script, let's discuss some points:

2.1.1 Statements and Lines

A **statement** is a unit logical instruction to the interpreter. Our `HelloWorld.py` program was made up of just 1 statement. A program, therefore, is an integral collection of statements.

A **line** is a sequence of characters terminated by the newline character. Our `Helloworld.py` program was a 1-line program. Thus, our program was both a 1-line program as well as contained just a single statement.

Statements in most popular programming languages end with a semi-colon, but this is not the case in Python. What it means to a simple programmer who does not want to think any deeper is that life has just become more convenient for you! You can simply type in statements and not worry about terminating them! But if you are the more inquisitive type who questions why this is so in Python, or maybe why this is not so in other languages, then let's proceed to understand.

Statements and Lines are connected in Python.

2.1.1.1 One Statement Per Line

In Python, a line is typically considered to host 1 statement. Thus, as long as there is 1 statement per line, there is no need of using semi-colons at the end of the line.

> **For C/C++/Java Programmers:**
>
> Programming languages like C, C++ and Java use semi-colons (`;`) as the **statement terminator**, which means that a semi-colon marks the end of a statement. This means that a statement is not considered complete unless the terminating semi-colon is seen, and thus allows a statement to span multiple lines as required.

Python relies on indentation rules to decide which statements belong to which block (will be covered in section 3.2.1) and uses the newline character as the statement terminator. In other words, a statement in Python is understood to be terminated in the same line.

Let us revisit `Helloworld.py` and write multiple statements – of course, one per line.

`HelloWorld2.py`

```
1.   print("Hello World")
2.   print("Hi from India!")
```

Output:

```
Hello World
Hi from India!
```

2.1.1.2 Multiple Statements Per Line

Python uses semi-colons as **statement separators** and this behaviour is elaborated below:

As a statement separator, the role of a semi-colon is to ensure that multiple statements can be given in one line and the translator is able to differentiate them as separate statements because of the semi-colon separating them.

> **For Perl/Shell Scripters:**
>
> Perl and the shell in Linux use semi-colons the same way, forcing programmers to use them as separators if they want to issue multiple statements/commands in the same line!

Let us revisit `HelloWorld2.py` and write multiple statements in a single line.

HelloWorld3.py

```
1.   print("Hello World"); print("Hi from India!")
```

Output:

```
Hello World
Hi from India!
```

NOTE:

There is no harm in ending a statement with a semi-colon, though it is neither required nor preferred.

2.1.1.3 One Statement Spanning Multiple Lines

What if a particular statement is long and we prefer to break it into multiple lines?

A single statement can span multiple lines if required, but this information needs to be conveyed to the Python interpreter. Any line that is going to be continued in the next line needs to end with a backslash (`\`). Thus, if a single statement has to span 4 lines for example, the first 3 lines will end with backslashes, each indicating that the statement continues further into the following line, but the 4th line will not end with a backslash and thus terminates the statement.

Let us intentionally rewrite `HelloWorld3.py` using multiple lines.

HelloWorld4.py

```
1.   print(\
2.   "Hello\
3.    World")
4.   print("Hi fr\
5.   om India!")
```

Output:

```
Hello World
Hi from India!
```

2.1.2 Quotation Marks

Strings in Python are enclosed in quotations, but unlike most other programming languages, Python does not differentiate between single quotes and double quotes.

2.1.2.1 Single Quotes

A string can well be enclosed in single quotes. Such a string can easily contain double quotes without any issue, but any single quote character inside the string needs to be escaped by prefixing it with backslash.

Here is another version of `HelloWorld.py`:

HelloWorld5.py

```
1.   print('Hello World')
2.   print('"Hi" from \'India\'!')
```

Output:

```
Hello World
"Hi" from 'India'!
```

2.1.2.2 Double Quotes

Just like single quotes, a string can well be enclosed in double quotes too. Such a string can easily contain single quotes inside, but any double quote character inside the string needs to be escaped by prefixing it with backslash.

Here is another version of `HelloWorld5.py`:

HelloWorld6.py

```
1.   print("Hello World")
2.   print("'Hi' from \"India\"!")
```

Output:

```
Hello World
'Hi' from "India"!
```

2.1.2.3 Triple Quotes

Python also supports a special kind of quotation mark made up 3 single-quotes (''') or 3 double-quotes (""""). Triple quotes allow a string to span multiple lines.

Furthermore, single quotes and double quotes can be freely used inside triple quotes – of course you can't directly use the 3 quotes used to start the string in succession or else it would be considered to be triple quotes and will terminate the string. Escaping of single quotes and double quotes is possible, if at all needed.

Here is the improved program:

HelloWorld7.py

```
1.   print("""Hello World
2.   'Hi' from "India"!""")
```

Output:

```
Hello World
'Hi' from "India"!
```

Triple quotes differs from single quotes and double quotes in the following ways:

1. They allow strings to span multiple lines
2. They do not require a terminating backslash at the end of every line to indicate continuation
3. They allow the embedding of both single quotes and double quotes without the need for escaping (except when there are 3 quotes in succession in which case an escaping might be required)
4. They preserve the newline character (that exists at the end of every line) within the string
5. They can have a special purpose – that of providing documentation (covered in section 10.15)

2.2 Comments

Comments are pieces of text added within programs to improve the readability. Comments are non-executable and the interpreter will ignore it's contents.

A comment in Python is indicated by the hash (`#`) symbol – everything from the hash till the end of that line is considered to be a comment.

Here is another version of `Helloworld.py` using comments:

HelloWorld8.py

```
1.   #Script to print "Hello World"
2.   #This two lines are comments
3.   print("Hello World") #One more comment
4.   print("Hi from India!")
```

Output:

```
Hello World
Hi from India!
```

A good programmer always uses comments to document why the code is doing what it is doing, or document facts that are either difficult to make out from the code or is not present in the code. Of course, the code should always be written in a manner that it clearly specifies what it is doing. Sometimes comments are also added to summarise what a block is doing – with the definition of a block also extending to functions, classes and modules!

Sometimes, the comments require multiple lines, and we end up using many hashes. There is a simpler way to implement such a comment using the concept of triple quotes. Triple quotes are not comments as such; they are technically strings – but strings do not do anything on their own, and can end up behaving like comments!

Section 10.15 will introduce an interesting application of comments used for documenting program elements, implemented using triple-quoted strings!

Here is a version that demonstrates both approaches:

HelloWorld9.py

```
1.   # Hi!
2.   # This program prints Hello World
3.   # I hope you like it!
4.   # ==============================
5.   #
6.   print("Hello World")
7.   """
8.   ==================================
9.   Hi!
10.  This program printed Hello World
11.  Did you run it?
12.  """
```

Output:

```
Hello World
```

2.3 Basic Data Types

One of the first things to learn in a programming language is what kinds of data types it provides. This will then help us know what can be represented and what operations are permitted. Python being a dynamic scripting language does not require (and does not allow) a variable to be declared to be of a specific type before use. Being a powerful object-oriented language, all data is considered to be objects and therefore all data types are classes (more details on objects, classes and object-oriented programming is covered in section 12).

When a variable stores a value, it is actually referring to an object of a specific type. The type of any expression can be found out using the built-in function `type()`. This will be a useful learning tool now and a useful debugging tool later on.

We are going to cover the following data types in the following sections:

1. Integers (`int`)
2. Real numbers (`float`)
3. Complex numbers (`complex`)
4. Strings (`str`)
5. Boolean (`bool`)

A more detailed coverage of these data types is available in section 20.1.

2.3.1 The `int` type

The simplest numeric data type to deal with is `int`. Objects of this type represent integers (both positive and negative) and cannot store a fractional part. The range is not limited in Python and we can store very large integers without worrying about whether Python will be able to handle it!

2.3.1.1 Literals of Type `int`

- Any literal constant made up of only digits (with an optional leading sign) is considered as an `int` constant.
- Any literal constant made up of only digits and with a leading `0o` (with an optional leading sign before that) is considered to be an octal constant of type `int`. Such constants cannot contain the digits `8` and `9` (as they are illegal in octal).

- Any literal constant made up of only digits and alphabets `a-f` in upper-case or lower-case and with a leading `0x` or `0X` (with an optional leading sign before that) is considered to be a hexadecimal constant of type `int`.

Example:

```
>>> 15
15
>>> -25 #Negative integer
-25
>>> 0o15 #Octal integer
13
>>> -0o15 #Negative octal integer
-13
>>> 0o08 #Illegal octal integer
  File "<stdin>", line 1
    0o08
       ^
SyntaxError: invalid syntax
>>> 0x12abc4 #Hexadecimal integer
1223620
>>> -0X12aBc4 #Negative hexadecimal integer with mixed case
-1223620
>>>
```

NOTE:

Any output that contains the Python prompt `>>>` (as in the case above) implies that this is a snapshot of an interactive python session and not the execution of a Python script. An interactive Python session helps us get output instantaneously as each statement is immediately processed and it's result is displayed. This can help as a learning tool. In such examples, whatever we need to type is shown in bold and rest is printed by the Python interpreter. When we want to combine multiple statements together for a more complex job, we will shift to writing Python scripts instead. See section 1.4 for information on how to launch the Python interpreter.

Objects of type `int` can also be created using the explicit constructor of class `int`. This of course also resembles a function call for type conversion. Thus, the following two statements are identical in function:

Example:

```
x = 2
x = int(2)
```

This built-in function is also helpful for converting other types like `float` and `string` to the type `int`:

```
>>> int(23.99)
23
>>> int("25")
25
```

The default constructor of `int` class will create an integer with value `0`, and a 2-argument constructor has been provided to convert a string representation of an integer in a particular base into a normal integer. The constructors have the following declaration:

Syntax:

```
int(x=0)
int(str_x, base=10)
```

Some examples follow:

```
>>> int()
0
>>> int(25)
25
>>> int("25")
25
>>> int("25",8)
21
```

2.3.1.2 Operations on Type `int`

The following table lists out the arithmetic operators that can be used with data items of type `int`:

Expression	*Result*	*Meaning*
`2 ** 3`	`8`	`2` to the power `3`
`2 * 3`	`6`	Multiplication
`10 / 3`	`3.3333333333333335`	Division
`10 // 3`	`3`	Integer division
`10 % 3`	`1`	Modulus/Remainder
`2 + 3`	`5`	Addition
`2 - 3`	`-1`	Subtraction

Table 1: Arithmetic operations on `int`

NOTE:

If the example "`10/3`" above yields "`3`" as the answer, you are using an old version of Python (less than 3.x). In such a case, while you can continue and learn Python through this book, it is recommended that you upgrade Python to the latest version to derive maximum benefits.

The following mathematical functions are available on type `int`:

Expression	*Result*	*Meaning*
`pow(2,3)`	`8`	`2` to the power `3`
`divmod(10,3)`	`(3,1)`	A tuple containing the quotient and remainder after integer division. Tuples will be covered in section 5
`abs(-10)`	`10`	Absolute value
`math.factorial(5)`	`120`	Factorial

Table 2: Mathematical functions that operate on int

NOTE:

When you encounter functions like "`math.factorial()`", it indicates that we are referring to the `factorial()` function of `math` module. For this to work, it is necessary to execute the statement `import math` once before invoking any function of that module in the session/script.

Objects of type `int` can be converted to other bases using these built-in functions:

Expression	*Result*	*Meaning*
`bin(12)`	`0b1100`	Converts the integer to binary
`oct(12)`	`0o14`	Converts the integer to octal
`hex(12)`	`0xc`	Converts the integer to hexadecimal

Table 3: Base conversion functions that operate on int

The following bitwise operators are available on type `int` (have a look at section 20.2 if you want to learn bit manipulation):

Expression	*Result*	*Meaning*
`2 & 3`	`2`	Bitwise AND
`2 | 3`	`3`	Bitwise OR
`2 ^ 3`	`1`	Bitwise XOR
`~2`	`-3`	1's complement
`2 << 3`	`16`	`2` left-shifted `3` bits
`2 >> 3`	`0`	`2` right-shifted `3` bits

Table 4: Bitwise operations on int

The following comparisons are possible on type `int`:

Expression	*Result*	*Meaning*
`2 < 3`	`True`	`2` is less than `3`?
`2 <= 3`	`True`	`2` is less than or equal to `3`?
`2 > 3`	`False`	`2` is greater than `3`?
`2 >= 3`	`False`	`2` is greater than or equal to `3`?
`2 == 3`	`False`	`2` is equal to `3`?
`2 != 3`	`True`	`2` is not equal to 3?

Table 5: Comparison operations on int

NOTE:

These comparisons are available for all numeric types. It is possible to compare two objects of different types using these operators. If the comparison does not makes sense and is not implemented, a `TypeError` exception is raised instead. Since these are available for all numeric types, it will not be repeated in the following sections.

NOTE:

The type `int` also supports Boolean operations, covered in section 2.3.5.2.

2.3.2 The `float` type

The `float` data type of Python allows us to store numbers that have a decimal part too.

For C/C++/Java programmers:

The `float` data type of Python can be considered to be equivalent to the `double` data types of languages like `C`, `C++` and `Java`, rather than their `float` data type.

2.3.2.1 Literals of Type `float`

Any literal constant made up of only digits (with an optional leading sign) with either a decimal point (`.`) or an exponentiation (`e`/`E`) is considered as a `float` constant.

Examples:

```
>>> 123.5
123.5
>>> 123.0
123.0
>>> 12e2
1200.0
>>> 123.5e2
12350.0
>>> 123.45678e2
12345.678
```

Objects of type `float` can also be created using the explicit constructor of class float. This of course also resembles a function call for type conversion. Thus, the following two statements are identical in function:

Example:

```
x = 2.5
x = float(2.5)
```

This built-in function is also helpful for converting other types like `int` and `string` to the type `float`. Also, the default constructor of `float` class will create a float with value `0.0`.

```
>>> float(23)
23.0
>>> float("23")
23.0
>>> float()
0.0
```

2.3.2.2 Operations on Type `float`

The following table lists out the arithmetic operators that can be used with data items of type `float`:

Expression	Result	Meaning
`1.2 ** 0.1`	`1.0183993761470242`	`1.2` to the power `0.1`
`2.5 * 3`	`7.5`	Multiplication
`7.5 / 3`	`2.5`	Division
`7.5 // 3`	`2.0`	Integer Division
`4.5 % 1.2`	`0.9000000000000001`	Modulus/Remainder
`2.1 + 3.3`	`5.4`	Addition
`2.1 - 3.3`	`-1.1999999999999997`	Subtraction

Table 6: Arithmetic operations on float

You might have observed that some results are not mathematically correct – there is a very small deviation, but obvious in these examples. Python provides the `Decimal` class (not covered in this book) that works better as we'd expect it to, but it comes with it's own cost in terms of efficiency.

The following mathematical functions are available on type `float`:

Expression	Result	Meaning
`pow(1.2,0.1)`	`1.0183993761470242`	`1.2` to the power `0.1`
`divmod(7.5,3)`	`(2.0, 1.5)`	A tuple containing the quotient and remainder after division. Tuples will be covered in section 5
`abs(-1.2)`	`1.2`	Absolute value
`math.floor(2.3)`	`2.0`	Rounds down (towards -∞) to the nearest integral float
`math.ceil(2.3)`	`3.0`	Rounds up (towards +∞) to the nearest integral float
`round(2.3)`	`2.0`	Rounds to the nearest integral float

Expression	Result	Meaning
`math.trunc(2.3)`	`2`	Truncates the float (towards 0) and gives an integer
`math.exp(2.3)`	`9.974182454814718`	Exponentiation (e^x)
`math.log(9.974182454814718)`	`2.3`	Natural logarithm
`math.log10(100)`	`2.0`	Common logarithm
`math.sqrt(2.3)`	`1.51657508881031`	Square root
`math.hypot(3,4)`	`5.0`	Hypotenuse
`math.degrees(math.pi)`	`180.0`	Radians to Degrees
`math.radians(180)`	`3.141592653589793`	Degrees to Radians
`math.sin(0)`	`0.0`	Sin(x), x in radians
`math.cos(0)`	`1.0`	Cos(x), x in radians
`math.tan(0)`	`0.0`	Tan(x), x in radians
`math.asin(0)`	`0.0`	$\sin^{-1}$(x) in radians
`math.acos(0)`	`1.5707963267948966`	$\cos^{-1}$(x) in radians
`math.atan(0)`	`0.0`	$\tan^{-1}$(x) in radians
`math.atan2(3,0)`	`1.5707963267948966`	$\tan^{-1}$(`3/0`) in radians

Table 7: Mathematical functions that operate on float

NOTE:

The functions that start with `math.` are functions of the `math` module, which needs to be imported prior to function call using the statement: `import math`.

NOTE:

The type `float` also supports comparison, like the ones covered in section 2.3.1.2 and Boolean operations, like the ones covered in section 2.3.5.2.

2.3.3 The `complex` type

The `complex` type allows us to represent complex numbers. Python stores complex numbers in rectangular coordinates, but conversion to polar coordinates is possible. Similarly, it is possible to create a complex number from polar coordinates. Given a complex object, the `real` field tells us the real part and the `imag` field tells us the imaginary part.

2.3.3.1 Literals of Type `complex`

The notation Python uses to represent a complex number made up of real part `x` and imaginary part `y` is `x+yj` or `x+yJ`.

Examples:

```
>>> (2+3j)
(2+3j)
>>> 2+3J
(2+3j)
>>> (2+3j).real
2.0
>>> (2+3j).imag
3.0
```

Objects of type `complex` can also be created using the explicit constructor of class `complex`. This of course also resembles a function call for type conversion. Thus, the following two statements are identical in function:

Example:

```
x = 2+3j
x = complex(2,3)
```

This built-in function is also helpful for converting other types like `int` and `string` to the type `complex`. Also, the default constructor of `complex` class will create a `complex` with value `0j`, a 1-argument constructor that receives an `int x` will create a complex with value `x+0j` and a 1-argument constructor that receives a `string` in the format `x` or `x+yj` or `x+yJ` will create a complex object suitably.

Examples:

```
>>> complex()
0j
>>> complex(2)
(2+0j)
>>> complex("2+3j")
(2+3j)
>>> complex("2+3J")
(2+3j)
>>> complex("2")
(2+0j)
```

2.3.3.2 Operations on Type `complex`

The following table lists out the arithmetic operators that can be used with data items of type `complex`:

Expression	Result	Meaning
`1j ** 2`	`(-1+0j)`	i^2
`(2+3j) * (5+6j)`	`(-8+27j)`	Multiplication
`(2+3j) / (2+3j)`	`(1+0j)`	Division
`(2+3j) + (5+6j)`	`(7+9j)`	Addition
`(2+3j) - (5+6j)`	`(-3-3j)`	Subtraction

Table 8: Arithmetic operations on type complex

Just as how the `math` module provides mathematical functions for dealing with real numbers, the `cmath` module provides mathematical functions for dealing with complex numbers. The module needs to be imported prior to use using the statement: `import cmath`. We can convert any complex object into polar coordinates using the `polar()` function, and convert polar coordinates into a complex object using the `rect()` function as shown in the examples in the table below. We can also extract the magnitude using `abs()` and argument using `phase()` as shown in the examples in the table below:

Examples:

Expression	*Result*	*Meaning*
`cmath.polar(2+3j)`	`(3.605551275463989, 0.982793723247329)`	Conversion from rectangular coordinates (x+yJ) to polar coordinates (r,θ)
`cmath.rect(3.605551275463989, 0.982793723247329)`	`(2+3j)`	Conversion from polar coordinates (r,θ) to rectangular coordinates (x+yJ)
`abs(3+4j)`	`5.0`	Magnitude (r)
`cmath.phase(3+4j)`	`0.9272952180016122`	Angle (θ)

Table 9: Mathematical functions that operate on complex

NOTE:

The type `complex` also supports `==` and `!=` comparisons, covered in section 2.3.1.2 and Boolean operations, covered in section 2.3.5.2.

2.3.4 The `str` type

The string type (`str`) allows us to store non-numeric values in addition to numeric values.

2.3.4.1 Literals of Type `str`

We have seen in section 2.1.2 that string literals can be enclosed in single, double or triple `quotes`. Here are a few examples to refresh your memory.

Examples:

```
>>> 'Hello "World" from \'India\''
'Hello "World" from \'India\''
>>> "Hello \"World\" from 'India'"
'Hello "World" from \'India\''
>>> """Hello "World" from 'India'"""
'Hello "World" from \'India\''
>>> '''Hello "World" from 'India\''''
'Hello "World" from \'India\''
```

TIP

If a string has many single-quotes, enclose it in double-quotes so that the quotes need not be escaped. Similarly, if a string has many double-quotes, enclose it in single-quotes. If a string has both quotes, enclose it in triple-quotes!

Objects of type `str` can also be created using the explicit constructor of class `str`. This of course also resembles a function call for type conversion. Thus, the following two statements are identical in function:

Example:

```
x = "abc"
x = str("abc")
```

This built-in function is also helpful for converting other types like `int` and `float` to the type `str`. Also, the default constructor of `str` class will create a null string (`''`).

```
>>> str("10")
'10'
>>> str()
''
```

NOTE:

Python strings support Unicode. If strings made up of only ASCII characters are desired, Python provides a separate `bytes` class (section 20.3) for that, though Python strings can also store ASCII characters.

The following table shows the escape characters that can be used within strings:

Escape Sequence	Meaning
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	Alert (beep)
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\ooo	Character with ASCII code ooo in octal
\xhh	Character with ASCII code hh in hexadecimal
\uhhhh	Unicode character with code hhhh
\Uhhhhhhhh	Unicode character with code hhhhhhhh
\N{name}	Unicode character with name name

Table 10: Escape Sequences

If a string contains backslashes, but the intention is not to denote escape sequences, the backslashes need to be escaped as shown in the table above. If however, the string contains many such backslashes, the entire string can be passed to Python as a raw string using the prefix r or R. In such cases, all characters are retained within the string without any special meaning and therefore no escape sequences are processed. Examples of escape sequences and raw strings follow.

Examples:

```
>>> "\tHello\nWorld\b\n\101\x41"
'\tHello\nWorld\x08\nAA'
>>> r"\tHello\nWorld\b\n\101\x41"
'\\tHello\\nWorld\\b\\n\\101\\x41'
>>> R"\tHello\nWorld\b\n\101\x41"
'\\tHello\\nWorld\\b\\n\\101\\x41'
```

Observation:

1. The character `A` has an ASCII code of `65`, which is `101` in octal and `41` in hexadecimal. These codes have been used in the examples above.
2. Python processes certain escape sequences and represents them in a different form (`\b` became `\x08` and `\x41` became `A`).
3. The `r` or `R` prefix will escape any backslashes so that they represent the literal backslash character rather than indicating an escape sequence.

2.3.4.2 Operations on Type `string`

Many useful functions are present in the string class to process strings and while some of them are documented in the tables below, formatting will be covered in section 8.7 and more formatting related functions will be examined in section 8.6:

Expression	*Result*	*Meaning*
`'aBc'.lower()`	`'abc'`	Converts all characters to lowercase
`'aBc'.upper()`	`'ABC'`	Converts all characters to uppercase
`'hello woRLD'.capitalize()`	`'Hello world'`	Converts all characters to lowercase and the first character to uppercase
`'hello woRLD'.title()`	`'Hello World'`	Converts all characters to lowercase and the first character of every word to uppercase
`'hello woRLD'.swapcase()`	`'HELLO WOrld'`	Converts all lowercase characters to uppercase and vice-versa

Table 11: Case conversion functions of string

Expression	Result	Meaning
`'abc'.isalpha()`	`True`	Returns true if all characters of the given string are alphabetic and the string is non-null; returns false otherwise
`'abc'.isupper()`	`False`	Returns true if all alphabetic characters in the string are in uppercase and there is at least one alphabet; returns false otherwise
`'abc'.islower()`	`True`	Returns true if all alphabetic characters in the string are in lowercase and there is at least one alphabet; returns false otherwise
`'abc'.isnumeric()`	`False`	Returns true if all characters in the string are numeric and the string is non-null; returns false otherwise
`'abc'.isalnum()`	`True`	Returns true if all characters in the string are alphanumeric (alphabetic or numeric) and the string is non-null; returns false otherwise
`'abc'.isidentifier()`	`True`	Returns true if the string contents can be a valid identifier; returns false otherwise
`'abc'.istitle()`	`False`	Returns true if the string contents are in title case; returns false otherwise
`'abc'.isprintable()`	`True`	Returns true if all characters of the string are printable; returns false otherwise
`'abc'.isspace()`	`False`	Returns true if all characters of the string are whitespace characters and the string is non-null; returns false otherwise

Table 12: Content testing functions of string

NOTE:

The type `string` also supports comparisons, covered in section 2.3.1.2 and Boolean operations, covered in section 2.3.5.2.

String concatenation can be achieved by using the "+" operator. Furthermore, string literals separated by only whitespaces are implicitly concatenated.

Examples:

```
>>> "Hello" + 'world'
'Helloworld'
>>> "Hello"   'world'
'Helloworld'
```

NOTE:

Do not conclude from the above examples that the "+" operator is not required in Python – remember that "+" can concatenate the contents of two strings referred to by variables at runtime! Check out the code snippet below:

```
>>> x="abc"
>>> y="xyz"
>>> x+y
'abcxyz'
>>> x y
  File "<stdin>", line 1
    x y
      ^
SyntaxError: invalid syntax
```

2.3.5 The `bool` type

The `bool` type allows us to represent Boolean values.

2.3.5.1 Literals of Type `bool`

The two possible Boolean values are `True` and `False`, which are keywords in Python, and objects of class `bool` at the same time.

Objects of type `bool` can also be created using the explicit constructor of class `bool`. This of course also resembles a function call for type conversion. Thus, the following two statements are identical in function:

Example:

```
x = True
x = bool(True)
```

This built-in function is also helpful for converting other types like `int` and `string` to the type `bool`. Also, the default constructor of `bool` class will create a `bool` with value `False`.

```
>>> bool(0)
False
>>> bool(10)
True
>>> bool(0.0)
False
>>> bool(-12.5)
True
>>> bool("hello")
True
>>> bool("")
False
>>> bool()
False
```

NOTE:

Section 20.1.1.4 discusses conversion to `bool` in detail and can throw more light on why we are getting the above output.

2.3.5.2 Operations on Type `bool`

The most common and practical operations on Boolean types are Boolean operations of `and`, `or` and `not`.

The `and` operator performs a logical AND operation. It is a short-circuiting operator, implying that if the result can be obtained after evaluation of the first operand itself, it does not bother evaluating the second operand.

Examples:

```
>>> True and False
False
>>> False and True
False
>>> True and True
True
```

The `and` operator can work on other types too! To generalise it's working, consider the example `x and y`.

- If `x` evaluates to `False`, the result is `x`
- If `x` evaluates to `True`, the result is `y`

```
>>> 2 and 5
5
>>> 5 and 2
2
>>> 0 and 5
0
```

The `or` operator performs a logical OR operation. It is a short-circuiting operator, implying that if the result can be obtained after evaluation of the first operand itself, it does not bother evaluating the second operand.

Examples:

```
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

The `or` operator can work on other types too! To generalise it's working, consider the example `x or y`.

- If `x` evaluates to `False`, the result is `y`
- If `x` evaluates to `True`, the result is `x`

```
>>> 2 or 5
2
>>> 5 or 2
5
>>> 0 or 5
5
```

The `not` operator performs a logical NOT operation.

Examples:

```
>>> not False
True
>>> not True
False
```

The `not` operator can work on other types too! To generalise it's working, consider the example `not x`.

- If `x` evaluates to `False`, the result is `True`
- If `x` evaluates to `True`, the result is `False`

```
>>> not 2
False
>>> not 5
False
>>> not 0
True
```

NOTE:

When other data types are used here instead of `bool`, a Boolean conversion takes place as documented in section 20.1.1.4.

The type `bool` is considered to be a subtype of `Integral` (just like the type `int`), and hence all operators that work on `int` also work on `bool`. In such cases, it will help to remember to substitute `True=1` and `False=0`.

2.4 Identifiers

An identifier is a name given to a program element. Variable names, class names and function names are some examples of identifiers.

The rules for naming identifiers in Python are similar to those of C but also permit Unicode characters:

1. The permissible characters are alphabets, digits and underscores
2. The first character cannot be a digit. Underscore as the first character can have special meanings that will be discussed in later sections
3. The length is not limited (but is subject to practical limits however!)
4. An identifier cannot have the same name as a keyword (discussed in section 2.5)

Good programming practices dictate that identifier names be relevant and readable so that a reader can figure out what the identifier represents. The coding convention used by Python professionals are:

1. Variable names and function names should be in lowercase with underscores separating words. E.g.: `variable_name`, `function_name()`
2. Class names (and exception names as they are classes) should be in lowercase with the first letter of every word capitalized. E.g.: `ClassName`, `ExceptionName`
3. Constant names should be in uppercase with underscores separating words. E.g.: `CONSTANT_NAME`
4. All other identifiers (package name, module name, method name, instance variable name, local variable name, function parameter name) should follow the convention of variable names

2.5 Keywords

Keywords are special words that are understood by Python. These cannot be used as identifiers. We will learn keywords on a need basis as and when it is time to learn them. We have already seen some keywords like `True`, `False`, `and`, `or` and `not`. Below is the list of all keywords in Python in alphabetical order:

`and`	`else`	`in`	`return`
`as`	`except`	`is`	`True`
`assert`	`False`	`lambda`	`try`
`break`	`finally`	`None`	`while`
`class`	`for`	`nonlocal`	`with`
`continue`	`from`	`not`	`yield`
`def`	`global`	`or`	
`del`	`if`	`pass`	
`elif`	`import`	`raise`	

2.6 Variables

Let us now discuss a few points about variables, their data types, their values and operations on them before proceeding to input and output. We have already seen the data types available in Python. These data types come into use when we have certain values of that type and the entity that holds this value is a variable. Here are a few important points about variables in Python:

#1 Variables can be created when required and where required

Unlike some of the more restrictive languages, we can start using a variable whenever required without having to declare it at the beginning of a block. This gives us flexibility and allows us to maintain focus on solving the problem at hand. A variable springs into existence the moment a value to assigned to it. There is no concept of specifying it's data type.

```
x = 10 #x is created here
x = 25 * x
y = x #y is created here
```

#2 The value of a variable can change at any time

Like other programming languages, the values of variables can be changed whenever required. However, what makes Python different here is the fact that there is no direct way in Python to implement constants. A good programmer recognises a constant from it's name and treats it like a constant though the interpreter may not!

```
x = 10 #x is created here
x = 25 * x #The value of x changes here
MAX = 100 #MAX is created here
MAX = 200 #The value of MAX changes here
```

For C/C++/Java programmers:

Python does not have the equivalent of keywords like `const` (C/C++) and `final` (Java).

#3 The type of a variable can change at any time

Unlike statically typed languages, the data type in Python is associated with a value and not with a variable! Since the value of a variable can change at any point of time, and the data type is an associated property of the value, the data type referred to by a variable could also change! The data type of any data item (possibly referred to by a variable) could be found out using the `type()` function as shown in the examples below.

```
>>> x=2
>>> type(x)
<class 'int'>
>>> x=2.5
>>> type(x)
<class 'float'>
>>> type(int)
<class 'type'>
>>> type(type)
<class 'type'>
```

#4 We can programmatically find out the data type of the value associated with a variable, and can also determine whether the type is "compatible" with another

While the `type()` function can give us the data type, what is more practical most of the times is simply verifying whether the data type of a particular value is a particular one or not (or whether it is a subtype of a particular one or not). For the built-in numeric types, the hierarchy is covered in detail in section 20.1 and can be verified from the examples below:

```
>>> import numbers
>>> x=2.5
>>> isinstance(x,int)
False
>>> isinstance(x,float)
True
>>> isinstance(x,numbers.Integral)
False
>>> isinstance(x,numbers.Real)
True
>>> isinstance(x,numbers.Complex)
True
>>> isinstance(x,numbers.Number)
True
>>> isinstance(x,object)
True
```

Observation:

1. The value `2.5` is of type `float`, which is a subtype of `numbers.Real`, which is a subtype of `numbers.Complex`, which is a subtype of `numbers.Number`, which is a subtype of `object`. The entire type hierarchy can be verified from section 20.1.
2. At the end of the day, all values are stored within objects, making `object` the root of the entire hierarchy.

#5 The operations permissible on a variable depend on the data type of the value the variable is currently referring to and thus can change during the execution of the script

Since we have established that the data type of a variable can change depending on the data type of the value it is referring to, and the operations that can be performed depends on the data type, what can and cannot be done with a variable depends on the data type of the value the variable currently refers to. As the data type of the value changes, so do the permissible operations! This is illustrated in the examples below:

```
>>> x=25
>>> x%7
4
>>> len(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
>>> x='Hello'
>>> x%7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
>>> len(x)
5
```

#6 Values are objects and variables are merely references to these objects

Any data item (value) is an *object*, and when they are assigned to a variable, what the variable is said to contain is a *reference* to that object. Thus, values and variables are very different! A single value (object) can have multiple references to it. Put another way, multiple variables can claim to contain the same identical value. Destroying a variable decrements the number of references that particular value has. When this reference count reaches `0` (an indication that the value does not have any variable referring to it) it is considered to be garbage and is a candidate for garbage collection. Destroying a variable is immediate and results in the variable disappearing from the program (till it is recreated if required). This however need not imply a destruction of the value.

A variable that is no longer needed can be destroyed using the `del` built-in. Keep in mind the following:

1. There is no compulsion for using `del` to delete variables that are no longer needed
2. These destroy only the references; there is no guarantee that any object will be freed from memory
3. Even if an object is to be freed, there is no saying whether and when it will be freed

```
>>> x=5
>>> x
5
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

2.7 Basic Input and Output

2.7.1 Printing Using the print() Function

There are multiple ways of generating output from a Python script – the simplest method being using the `print()` built-in function. The `print()` function can print multiple objects one after another, with a user-defined separator string and a user-defined terminator string and with defaults in place. Let's get started!

2.7.1.1 Printing a Single Item

To print just a single item, we pass that item to the `print()` function as shown in the examples below:

```
>>> print('hello')
hello
>>> print(12)
12
>>> print(2.5)
2.5
>>> print(True)
True
>>> print("2.5")
2.5
>>> print(1.5 * 3)
4.5
```

2.7.1.2 Printing Multiple Items

To print multiple items, we pass all the items in sequence separated by commas as shown in the examples below. Do note that they are automatically separated by spaces.

```
>>> print("hello","world")
hello world
>>> print("2+3=",2+3)
2+3= 5
```

2.7.1.3 Separating Items

If we want to print multiple items but do not want space to be the separator, we can pass the separator string as an argument to `print()` as shown in the examples below. The separator string can be a single character, or multiple characters or even a null string to indicate that we don't want any separator between the items. Arguments like `sep` in the example below are called *keyword arguments* and are covered in detail in section 10.6.

```
>>> print("hello","world",sep=',')
hello,world
>>> print("hello","world",sep=" <-> ")
hello <-> world
>>> print("Welcome","to","Python",sep=" <-> ")
Welcome <-> to <-> Python
>>> print("2+3=",2+3,sep="")
2+3=5
```

2.7.1.4 Terminating Items

By default, the `print()` function prints a newline character (`\n`) after printing all the items. If we choose, we can control this too using the keyword argument `end` as shown in the examples below. Either we can end up printing any suffix string of our choice, or more practically, remove the newline so that multiple `print()` functions end up producing their output on the same line.

```
>>> print("Python", "is", "a", "dynamic", "language", end="\n\t\t--
Python\n")
Python is a dynamic language
            --Python
>>> print("Hello","world",end="\n\n\n\n")
Hello world



>>> print("2+3=",2+3,end="")
2+3= 5>>>
```

2.7.1.5 Separating Items and Terminating Items

We can combine the separator string and terminator string as shown in the examples below:

```
>>> print("Welcome","to","Python",sep="...",end="\n---===---\n")
Welcome...to...Python
---===---
>>> print("Welcome","to","Python",end="\n---===---\n",sep="...")
Welcome...to...Python
---===---
```

Observation:

1. The order in which the keyword arguments `sep` and `end` are given do not matter
2. Keyword arguments must be at the end of the argument list. They cannot precede any non-keyword argument.

2.7.1.6 Printing Blank Lines

As a special case, when no argument is passed to the `print()` function, it ends up printing only the terminator string, which is newline by default, and thus ends up printing a blank line:

```
>>> print()

>>>
```

2.7.2 Formatting Strings Using the format() Function

While we were able to print using the `print()` function, we were unable to format the output in any way (apart from adding separator and terminator strings). For more advanced formatting, we can rely on the `format()` function of string class, which can format a string and give us a resultant string that can then be printed.

The `format()` function searches the string for placeholders. These placeholders are indicated by braces(`{}`) and indicate that some value needs to be substituted there (and probably formatted too). As a special case, if the string has no placeholders in it, the `format()` function does nothing with the string and returns it as it is, as shown in the example below:

```
>>> 'Hello world'.format()
'Hello world'
```

NOTE:

The `format()` function of string class does not print! It returns the formatted string, which can then be printed using `print()` if required.

2.7.2.1 Substituting Arguments

Let's now make use of the placeholders for substituting values. The `format()` function can receive multiple arguments which are implicitly numbered from 0. These argument values can be accessed and substituted in the string wherever placeholders are present. The placeholder can specify which argument is to be substituted by identifying the argument number as shown in the example below:

```
>>> '{0} {1}'.format('Hello','world')
'Hello world'
```

In the above example, argument #0 is `Hello` and is substituted within the string wherever `{0}` is present, and similarly argument #1 (`world`) is substituted wherever `{1}` is present in the string.

2.7.2.2 Substituting Arguments Out of Order

Since the argument number is present in the placeholder, we can even print argument values out of sequence as shown in the example below:

```
>>> '{1} {0}'.format('Hello','world')
'world Hello'
```

2.7.2.3 Substituting Arguments Multiple Times

Now that we've got a hang of how placeholders work, nothing prevents us from referring to the same argument multiple times within the format string:

```
>>> '{0} {1}, {0} reader. Python is my {1}'.format('Hello','world')
'Hello world, Hello reader. Python is my world'
```

2.7.2.4 Substituting Arguments in Order

As a shortcut, if all placeholders reference the arguments in order, the placeholders need not identify the argument at all as shown in the below example:

```
>>> '{} {}'.format('Hello','world')
'Hello world'
```

2.7.2.5 Substituting Arguments by Name

If you find argument numbering cumbersome, you can even name the arguments using the concept of *keyword arguments* and *named placeholders* as shown in the examples below:

```
>>> '{p} is {a}, {p} is {b}! {p} is {c}!'.format\
... (p='Python',a='easy',b='fun',c='brilliant')
'Python is easy, Python is fun! Python is brilliant!'
>>> 'Hello {name}, welcome to {language}!'.format\
... (name='Ram',language='Python')
'Hello Ram, welcome to Python!'
```

2.7.2.6 Basic Formatting of Arguments

We have only visited the tip of the `format()` iceberg so far! Each placeholder supports additional formatting information that can control how something is to be printed. A more detailed coverage of this is provided in section 8.7, but for now let us learn just a few steps.

To specify formatting for a placeholder, a '`:`' is used after the argument specifier. After the '`:`', the formatting information follows. Perhaps the simplest formatting is selecting the output base (`b` for binary, `o` for octal, `d` for decimal and `x` for hexadecimal). Check out this example:

```
>>> 'bin={0:b} oct={0:o} dec={0:d} hex={0:x}'.format(10)
'bin=1010 oct=12 dec=10 hex=a'
```

Considering that we now know the various ways of specifying the argument and also some simple formatting, let us see various combinations that we can use:

Combination	Meaning
`{0}`	Display argument #0
`{age}`	Display argument named `age`
`{}`	Display next argument
`{0:d}`	Display argument #0 in decimal
`{age:d}`	Display argument named `age` in decimal
`{:d}`	Display next argument in decimal

Table 13: Examples of types of formatting

2.7.3 Taking Input Using the input() Function

We know how to print output. Let us now learn how to receive input from the user. This knowledge will help us write interactive Python scripts that ask the user to provide input for the script.

2.7.3.1 Taking an input

To receive any input from the user, the `input()` function can be used. This function, in it's simplest form, waits for the user to enter a line of text and returns the input entered after chopping off the trailing newline character.

Example:

```
>>> name=input()
Ram
>>> language=input()
Python
>>> print('{} is learning {}'.format(name,language))
Ram is learning Python
```

2.7.3.2 Taking an input After Prompting

As is evident from the output above, when the `input()` function is executed, it merely waits for the user to enter a line of input. The user might have no clue that the script is waiting for input. It is therefore a good idea to print some message informing the user that some input is required. This message can be printed using a `print()` call before invoking `input()`. There is an easier method though! The `input()` function, helpful as it is, permits us to pass a prompt string to be displayed just prior to accepting input from the user. In the previous example, no prompt string was given and hence nothing was displayed. But if a prompt string is given, it will be displayed and then the user is given a chance to enter input.

Example:

```
>>> name=input("Enter your name:")
Enter your name:Ram
>>> language=input("Enter language:")
Enter language:Python
>>> print('{} is learning {}'.format(name,language))
Ram is learning Python
```

2.8 Getting Help

During the learning process, it is natural to get stuck and the need arises to get timely help to overcome the temporary block in progress. There are a couple of ways to get help.

2.8.1 Getting Help in the Book

As far as this book is concerned, it has been carefully crafted to show you just enough steps at a time for you to understand, experiment and build confidence. But there are numerous tables and appendices that provide a standard reference. These might help.

2.8.2 Getting Help in Python

While working on the Python interpreter, you can get help on any function, class, keyword or module by typing `help(topic)`. You can scroll through the help document (just like how you would use the `less` command in Linux) and finally type in '`q`' to quit help.

2.8.3 Getting Help Online

If you are looking for online help, there are a couple of websites that can help:

- The official Python tutorial can be found at https://docs.python.org/3/tutorial/index.html.
- The standard library of Python is discussed at https://docs.python.org/3/library/index.html.
- Frequently Asked Questions are available at https://docs.python.org/3/faq/.
- The complete Python language reference is available at https://docs.python.org/3/reference/index.html.

2.8.4 Getting Help From People

If you want to interact with people to get help, here are some more “official” resources, but be sure to read “The Guide to Python Mailing Lists” (available at https://www.python.org/community/lists/#comp-lang-python) before proceeding with any of these options below:

- Post to the Python newsgroup (news:comp.lang.python)
- Post to the Python Tutor Mailing List (http://mail.python.org/mailman/listinfo/tutor)
- Send a mail to help@python.org

2.8.5 Other Ways of Getting Help

Lastly, do not forget Google (https://www.google.com) and Stack Overflow (https://www.stackoverflow.com), the most precious resources for programmers when they get stuck!

2.9 Questions

1. Define a 'statement' in Python.
2. Which ASCII character signifies the end of a statement in Python?
3. Which ASCII character is used to separate statements in a single line in Python?
4. How does triple quotes(`"""`) help to form a comment in Python?
5. List the basic data types of Python.
6. How do we know the data type of a data item referred to by a variable in Python?
7. Which operator is used for integer division in Python?
8. Which expression in Python is equivalent to a mathematical function `pow()`?
9. When does Python consider a given literal as a '`float`' type?
10. How can a null string be represented in Python?
11. List any 3 operations that can be performed on the type '`str`' in Python.
12. List any 2 operations that can be performed on the type '`bool`' in Python.
13. List the rules for naming identifiers in Python.
14. Write a short note on variables and references in Python.
15. Write a short note on the `complex` type of Python.
16. How is the `help()` function of the Python interpreter different from the `--help` option?
17. Write a short note on formatting strings in Python.

2.10 Exercises

1. Write a program to find if a given number is a perfect square or not.
2. Write Python statements to calculate and print the simple interest using variables `p`, `t` and `r` for principal, rate of interest and time duration respectively.
3. Write Python statements to create 2 `complex` objects and print their sum,

difference and product.

4. Write Python statements to convert a `complex` object from polar to rectangular coordinates and vice-versa.
5. Write Python statements to accept a string from the user and print it in title-case.

SUMMARY

- Generally, Python concludes that 1 statement spans 1 line. A single statement can span multiple lines using backslashes at the end of each line that continues. Multiple statements can be present in the same line if separated by semicolons.
- Python does not differentiate between single quotes, double quotes, triple single-quotes and triple double-quotes!
- The basic data types of Python are int, float, complex, str and bool.
- Variables hold references to objects. Multiple variables can hold references to the same object too!
- The print() function prints a list of items whereas the input() function prints an optional prompt message and reads in a single line of input.
- Output formatting can be done using the format() method of string (str) class. This method returns the formatted string, which can be printed using the print() function.

3 Python Control Structures

In this chapter you will be able to:

- ☑ Write Python scripts to solve problems and manage the flow of control through it.
- ☑ Frame decision constructs to conditionally execute statements.
- ☑ Frame loop constructs to repeat instructions as many times as required.
- ☑ Learn how to terminate control from a loop, a function, a block and a script.

Python Control Structures

3.1 Getting Started with Programs

We did get started with writing Python scripts that print "`Hello world`" and also learnt about various numeric data types that Python supports. Let us put all this into practice now!

A program is a **sequence** of statements that are executed in a sequential fashion in order to take in some input, process it as required and produce desired output. Sometimes, we would need to alter the flow of control through the program. We have **decision** constructs that help evaluate a condition and decide the path through which control should flow in the program. Sometimes we might need to repeat a set of statements multiple times, and **loops** can help implement this flow of control. We will see all these kinds of constructs that are available in Python.

This section ends with a description of how we can remove control away from a control structure.

Let us start with a program that asks the user to enter his/her name and age and prints it back in the form of a greeting.

name_age.py

```
1.   #!/usr/bin/python3
2.
3.   # Program that asks for the user's name and age
4.   # and prints it back as a greeting!
5.
6.   name = input("Enter your name: ")
7.   age = int(input("Enter your age: "))
8.
9.   print("Hi {}, you are {:d} years old!".format(name,age))
```

Output:

```
Enter your name: Ram
Enter your age: 25
Hi Ram, you are 25 years old!
```

Observation:

1. The first line of the script is "`#!/usr/bin/python`". Recall from section 1.6 that this helps in executing the script implicitly. Regardless of whether we intend to execute the script implicitly or pass it to the interpreter explicitly, we will follow this style from now on.
2. In line 9, the "`:d`" format is being applied on age to print it in decimal (though that is the default for an `int` anyway and would have printed the same as a string too). This format requires age to be an `int` rather than a string that `input()` returns. Therefore, this conversion is performed in line 7.
3. The `input()` function always returns a string. If we were expecting a different type (like an `int`), it is a good idea to perform a conversion immediately after obtaining the result from `input()` (as shown in line 7) .

Let us now write a program that performs some basic operations on integers like:

- Finding it's previous and next integer
- Finding it's square and cube
- Finding it's square root and factorial

`int_demo.py`

```
1.   #!/usr/bin/python3
2.
3.   # Program that performs basic operations on int
4.
5.   import math
6.
7.   x = int(input("Enter an integer: "))
8.
9.   print("x={}".format(x));
10.  print("x lies between {} and {}".format(x-1,x+1))
11.  print("square(x)={} cube(x)={}".format(x**2,pow(x,3)))
12.  print("sqrt(x)={}".format(math.sqrt(x)))
13.  print("factorial(x)={}".format(math.factorial(x)))
```

Output:

```
Enter an integer: 5
x=5
x lies between 4 and 6
square(x)=25 cube(x)=125
sqrt(x)=2.23606797749979
factorial(x)=120
```

Observation:

1. Line 5 imports the `math` module that is required for functions like `sqrt()` and `factorial()`
2. The square of an integer can be found out using `x**2` or `pow(x,2)`

The next program will deal with logarithms and anti-logarithms

log.py

```
1.   #!/usr/bin/python
2.
3.   # Logarithms and anti-logarithms
4.
5.   import math
6.
7.   x = float(input("Enter a real number: "))
8.
9.   print("Natural logarithm and anti-logarithm")
10.  print("ln(x)={} exp(ln(x))={}".format(math.log(x),
math.exp(math.log(x))))
11.
12.  print("\nCommon logarithm and anti-logarithm")
13.  print("log(x)={} antilog(log(x))={}".format(math.log10(x),
pow(10,math.log10(x))))
14.
15.  print("\nBase 2 logarithm and anti-logarithm")
16.  print("log2(x)={}
antilog2(log2(x))={}".format(math.log(x,2),
pow(2,math.log(x,2))))
```

Output:

```
Enter a real number: 1.2
Natural logarithm and anti-logarithm
ln(x)=0.1823215567939546 exp(ln(x))=1.2

Common logarithm and anti-logarithm
log(x)=0.07918124604762482 antilog(log(x))=1.2

Base 2 logarithm and anti-logarithm
log2(x)=0.2630344058337938 antilog2(log2(x))=1.2
```

Observation:

1. Line 5 includes the `math` module required for the functions used here
2. The `log()` function gives the natural logarithm by default, and `exp()` is the natural anti-logarithm
3. The `log10()` function gives the common logarithm (base 10) and common anti-logarithm of `x` is `10` to the power `x`
4. For logarithm of any other base, the `log()` function can be used with the desired base as the second argument as shown in line 16. The anti-logarithm of `x` to base `b` is `b` to the power `x`

The next program deals with trigonometry:

`trigonometry.py`

```
1.   #!/usr/bin/python
2.
3.   # Trigonometry demo
4.
5.   import math
6.
7.   angle = float(input("Enter an angle in degrees: "))
8.
9.   angle = math.radians(angle) # Conversion from degrees to
radians
10.
11.  print("sin(x) =",math.sin(angle))
12.  print("cos(x) =",math.cos(angle))
13.  print("tan(x) =",math.tan(angle))
14.  print("cosec(x) =",1/math.sin(angle))
15.  print("sec(x) =",1/math.cos(angle))
16.  print("cot(x) =",1/math.tan(angle))
```

Output:

```
Enter an angle in degrees: 90
sin(x) = 1.0
cos(x) = 6.123233995736766e-17
tan(x) = 1.633123935319537e+16
cosec(x) = 1.0
sec(x) = 1.633123935319537e+16
cot(x) = 6.123233995736766e-17
```

Observation:

1. The trigonometric functions work with radians and hence we first convert the given angle in degrees to radians in line 9
2. There are no separate functions for `sec()`, `cosec()` and `cot()`, which are the reciprocals of `cos()`, `sin()` and `tan()` respectively, as implemented in lines 14-16
3. The value of cos(90°) is mathematically 0, but here we get `6.123233995736766e-17` which is 0.00000000000000006123233995736766 (nearly 0). This is because of the precision limits as well as approximation of π used while converting to radians.
4. Similarly, the value of tan(90°) is infinity, which Python can represent as `float("inf")`. However, again due to precision issues and approximation of π, the value obtained is `1.633123935319537e+16` , which is 16331239353195370, a value supposedly "close" to infinity. However, it is also true that `math.degrees(math.atan(1.633123935319537e+16 ))` will yield `90.0`!

Let us now work with complex numbers.

complex1.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of operations on a complex number
4.   import math
5.   import cmath
6.
7.   r = float(input("Enter the real part of a complex number:
"))
8.   i = float(input("Enter the imaginary part of a complex
number: "))
9.
10.  c = complex(r,i) # Construct the complex number
11.
12.  print("The complex number is",c)
13.  print("\tIt's real part is",c.real)
14.  print("\tIt's imaginary part is",c.imag)
15.  print("It's amplitude is",abs(c))
16.  print("It's angle is",math.degrees(cmath.phase(c)))
```

Output:

```
Enter the real part of a complex number: 2
Enter the imaginary part of a complex number: 3
The complex number is (2+3j)
      It's real part is 2.0
      It's imaginary part is 3.0
It's amplitude is 3.605551275463989
It's angle is 56.309932474020215
```

Observation:

1. We have imported the `math` module in line 4 because of the call to `degrees()` function in line 16.
2. We have imported the `cmath` module (complex mathematics) in line 5 because of the call to `phase()` function in line 17.
3. Refer to section 2.3.3.2 for a help on the mathematical functions used here.

Let us now switch to solving real world problems!

Problem: Write a program to print the simple interest and amount, given the principal, the rate of interest (in percentage per annum) and the time (in years).

simple_interest.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the simple interest and amount
4.
5.   p = float(input("Enter the principal: "))
6.   r = float(input("Enter the rate of interest (%pa): "))
7.   t = float(input("Enter the duration (years): "))
8.
9.   si = (p*t*r)/100
10.  amount = p + si
11.
12.  print("Simple interest={}",si)
13.  print("Amount={}",amount)
```

Output:

```
Enter the principal: 1000
Enter the rate of interest (%pa): 9
Enter the duration (years): 2
Simple interest = 180.0
Amount = 1180.0
```

3.2 Decisions

One of the fundamental control flow statements any programming language provides is the support for decisions – to conditionally execute a piece of code. Python provides the `if` Statement to implement decisions. Like most other programming languages, this has many forms – each of which will be discussed in the following sections.

3.2.1 The if Statement

The simple `if` Statement conditionally executes a statement or a block of statements. It's syntax is shown below:

```
if condition: statement

if condition:
      statements
      ...
```

In the first form shown above, if the condition evaluates to `True`, the following statement is executed. If the condition evaluates to `False`, that statement is skipped and control moves on to the next line.

In the second form shown above, if the condition evaluates to `True`, all the statements within the block are executed in sequence. If the condition evaluates to `False`, all the statements within the block are skipped and control moves to the first statement outside the block.

The statements inside the `if` block are identified by their **indentation**! To keep things simple, for now we'll use a single tab character per level of indentation.

Let us write a program that illustrates this. Here is a program that accepts the user's name and age and prints a message saying whether the user is eligible to vote or not, using the premise that a person needs to be at least 18 years of age in order to be able to vote.

vote1.py

```
1.   #!/usr/bin/python3
2.
3.   # Program that asks for the user's name and age
4.   # and prints whether the user is eligible to vote or not!
5.
6.   name = input("Enter your name: ")
7.   age = int(input("Enter your age: "))
8.
9.   if age >= 18: print("Hi {}! You can vote!".format(name))
10.  if age < 18: print("Sorry {}! You can't
vote!".format(name))
```

Output:

```
Enter your name: Ram
Enter your age: 25
Hi Ram! You can vote!
```

```
Enter your name: Sham
Enter your age: 15
Sorry Sham! You can't vote!
```

Observation:

1. The condition for a person to be able to vote is `age >= 18`. Similarly, the condition for a person to not be able to vote is `age < 18`.
2. Only one of the two conditions above can be true for any given value of age

We have used the first syntax of the `if` statement for implementing this since we have only a single statement to be executed if the condition is true. However, nothing prevents us from using the second syntax and have just a single statement within the statement block. This is shown below:

vote2.py

```
1.   #!/usr/bin/python3
2.
3.   # Program that asks for the user's name and age
4.   # and prints whether the user is eligible to vote or not!
5.
6.   name = input("Enter your name: ")
7.   age = int(input("Enter your age: "))
8.
9.   if age >= 18:
10.      print("Hi {}! You can vote!".format(name))
11.
12.  if age < 18:
13.      print("Sorry {}! You can't vote!".format(name))
```

Output:

```
Enter your name: Ram
Enter your age: 25
Hi Ram! You can vote!
```

```
Enter your name: Sham
Enter your age: 15
Sorry Sham! You can't vote!
```

3.2.2 The if-else Statement

In the previous example, we had 2 conditions:

1. `age >= 18`
2. `age < 18`

We see that the 2 conditions are mutually exclusive – if one is true, it implies that the other is false. In such situations, we prefer using the `if-else` statement instead of 2 different `if` statements. Using `if-else` is more advantageous because only 1 condition drives the flow of control, thereby being more simple, more manageable and more efficient. The syntax of the `if-else` statement is given below:

```
if condition: statement1
else: statement2

if condition:
      statement_block1
      ...
else: statement2

if condition: statement1
else:
      statement_block2
      ...

if condition:
      statement_block1
      ...
else:
      statement_block2
      ...
```

These are 4 permutations of simple statements (single statements) and compound statements (statement blocks), but the working is similar:

- if the condition evaluates to `True`, `statement1` or `statement_block1` is executed sequentially and `statement2` or `statement_block2` is omitted, with control resuming finally after the `if-else` statement.
- If the condition evaluates to `False`, `statement1` or `statement_block1` is omitted and `statement2` or `statement_block2` is executed sequentially, with control resuming finally after the `if-else` statement.

Let us rewrite the previous program using `if-else` statement:

vote3.py

```
1.   #!/usr/bin/python3
2.
3.   # Program that asks for the user's name and age
4.   # and prints whether the user is eligible to vote or not!
5.
6.   name = input("Enter your name: ")
7.   age = int(input("Enter your age: "))
8.
9.   if age >= 18:
10.      print("Hi {}! You can vote!".format(name))
11.  else:
12.      print("Sorry {}! You can't vote!".format(name))
```

The output is the same as the previous program and hence not duplicated here. Isn't this program more readable and more simpler than the previous one? It also turns out to be more efficient as well!

3.2.3 The if-elif-else Statement

Let us take the mutual exclusion concept a little further. We have so far concluded that if 2 conditions are mutually exclusive, we can use a single `if-else` statement instead of 2 independent `if` statements. Similarly, if we have more than 2 mutually exclusive conditions, we can use the `if-elif-else` statement. It's syntax is given below:

```
if condition1:
       statement_block1
       ...
elif condition2:
       statement_block2
       ...
elif condition3:
       statement_block3
       ...
...
else:
       statement_blockN
```

Observation:

1. We have used the statement block syntax above, but each condition (and the `else` clause as well) could well be followed by single statements instead.
2. The complete syntax of the `if` statement is therefore an `if` block followed by 0 or more `elif` blocks followed by an optional `else` block.
3. The conditions are tested sequentially and the moment one of them evaluates to `True`, the corresponding block is executed sequentially and all the other blocks are skipped with control resuming after the `if-else` block.
4. If none of the conditions evaluate to `True`, the `else` block is executed if the `else` clause is present. Otherwise, the entire `if-else` block is skipped.

Let's write a program to use this. Given an integer, let us classify it as being a positive integer, negative integer or zero.

int_sign.py

```
1.   #!/usr/bin/python3
2.
3.   # Program that classifies an integer as being
4.   # 1. A positive integer
5.   # 2. A negative integer
6.   # 3. Zero
7.
8.   x = int(input("Enter an integer: "))
9.
10.  if x > 0:
11.      print("{} is positive".format(x))
12.  elif x < 0:
13.      print("{} is negative".format(x))
14.  else:
15.      print("{} is zero".format(x))
```

There is probably nothing more to explain about the program – it should be pretty self-explanatory! Do note however, that due to mutual exclusion of the conditions, we don't have to explicitly check if the number is zero when it is neither positive nor negative. In this program, the order in which we give the conditions does not matter, nor does it matter which 2 of the 3 cases we frame conditions for. This is not always the case, however, as will be demonstrated in the next program.

Let us write a program that classifies a given pair of coordinates. It has to tell us one of the following:

1. The point is on the origin
2. The point is on the x-axis
3. The point is on the y-axis
4. The quadrant the point belongs to, otherwise

quadrant.py

```
1.   #!/usr/bin/python
2.
3.   # A script to tell whether a given point is on the origin,
4.   # on the x- or y-axis, or in a particular quadrant.
5.
6.   x = int(input("Enter the x-coordinate of the point: "))
7.   y = int(input("Enter the y-coordinate of the point: "))
8.
9.   if x == 0 and y == 0:
10.      print("The point lies on the origin")
11.  elif y == 0:
12.      print("The point lies on the x-axis")
13.  elif x == 0:
14.      print("The point lies on the y-axis")
15.  elif x > 0 and y > 0:
16.      print("The point lies in the first quadrant")
17.  elif x < 0 and y > 0:
18.      print("The point lies in the second quadrant")
19.  elif x < 0 and y < 0:
20.      print("The point lies in the third quadrant")
21.  else:
22.      print("The point lies in the fourth quadrant")
```

Output:

```
Enter the x-coordinate of the point: -3
Enter the y-coordinate of the point: 0
The point lies on the x-axis
```

```
Enter the x-coordinate of the point: -3
Enter the y-coordinate of the point: 2
The point lies in the second quadrant
```

Observation:

1. We classify the quadrant to which the given points belongs to only when the point does not lie on either axes.
2. The given point lies on the y-axis if it's x-coordinate is 0, and on the x-axis if it's y-coordinate is 0. However, if the point lies on the origin (with both coordinates being 0), we neither say it lies on the x-axis nor on the y-axis.
3. The order of the conditions therefore is important here. If for instance we

decided to first frame a condition for the point lying on the y-axis, this would be the condition: `x==0 and y!=0`.

4. The order in which we check for the quadrants still does not matter.

Let us write a program that finds the roots of a quadratic equation $ax^2+bx+c=0$ using the formula:

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

We will also classify the roots of the equation based on the discriminant (b^2-4ac) as follows:

1. Real and equal, if discriminant is 0
2. Real and distinct, if discriminant > 0
3. Imaginary, if discriminant < 0

quadratic_roots.py

```
1.   #!/usr/bin/python
2.
3.   # Script to classify and determine
4.   # the roots of a quadratic equation
5.
6.   import math
7.
8.   a = int(input("Enter the value of a: "))
9.   b = int(input("Enter the value of b: "))
10.  c = int(input("Enter the value of c: "))
11.
12.  discriminant = b**2 - 4*a*c
13.
14.  if discriminant == 0:
15.      print("The roots are real and equal")
16.      x = -b/(2*a)
17.      print("The root is",x)
18.  elif discriminant > 0:
19.      print("The roots are real and distinct")
20.      part1 = -b/(2*a)
21.      part2 = math.sqrt(discriminant)/(2*a)
22.      x1 = part1 + part2
23.      x2 = part1 - part2
24.      print("The roots are",  x1, "and", x2)
25.  else:
```

```
26.      print("The roots are imaginary")
27.      part1 = -b/(2*a)
28.      part2 = math.sqrt(-discriminant)/(2*a)
29.      x1 = complex(part1, part2)
30.      x2 = complex(part1, -part2)
31.      print("The roots are",  x1, "and", x2)
```

Output:

```
Enter the value of a: 1
Enter the value of b: 4
Enter the value of c: 4
The roots are real and equal
The root is -2.0
```

```
Enter the value of a: 1
Enter the value of b: 5
Enter the value of c: 6
The roots are real and distinct
The roots are -2.0 and -3.0
```

```
Enter the value of a: 2
Enter the value of b: 4
Enter the value of c: 4
The roots are imaginary
The roots are (-1+1j) and (-1-1j)
```

Observation:

1. We take the discriminant as:

   $$discriminant = b^2 - 4\text{ac}$$

   This changes the formula to the following:

   $$x = \frac{-b \pm \sqrt{discriminant}}{2\text{a}}$$

2. If the discriminant is 0, the formula will reduce to:

$$x = \frac{-b}{2a}$$

3. If the discriminant is positive, the formula can be split into 2 terms as:

$$x = \frac{-b}{2a} \pm \frac{\sqrt{discriminant}}{2a}$$

4. If the discriminant is negative, the formula can be split into 2 terms as:

$$x = \frac{-b}{2a} \pm i\frac{\sqrt{-discriminant}}{2a}$$

3.2.4 Empty Blocks and the pass Keyword

There are situations when we need a block to be empty – something that is not permitted in Python. This could be because out of the many conditions that we would like to handle in particular ways, there might be certain exceptions that we don't want to process at all. Or less genuinely, there might be cases wherein we want to fill in the block contents at a later time when we are ready and right now want to proceed with an empty block nevertheless. Whatever be the reason, the fact remains that a single statement is a single statement and a block of statements is a block of 1 or more statements – Python does not support zero statements when a statement is expected!

In such situations where a statement is required, but we don't have anything to provide, we can use the `pass` keyword to indicate a null statement – a statement that does nothing on execution!

To demonstrate the usage of pass, consider the objective of printing the value of the variable x if it is not divisible by 3. Here are 2 ways of doing it:

Method #1:

```
>>> x=5
>>> if x%3 != 0: print(x)
...
5
```

Method #2:

```
>>> x=5
>>> if x%3==0: pass
... else: print(x)
...
5
```

Observation:

1. Both the methods are identical in functionality – different programmers prefer different styles!
2. The first method is preferred by programmers who want their code to be compact.
3. The second method is preferred by programmers who like “positive logic” more than “negative logic” - programmers who find it easier to deal with equality and true rather than inequality and false!

3.2.5 Nested if Statements

An `if` statement within an `if` statement is called a nested `if` statement. Of course, the inner `if` statement can be within either the `if` block or an `elif` block or the `else` block of the outer `if` statement. The indentation level helps Python realise to which block a statement belongs to. Statements belonging to the inner `if` will be indented one level deeper than statements belonging to the outer `if`.

Let us write a program that accepts a word from the user and tells us which of the following attributes hold good on the complete word:

- Alphanumeric characters
- Alphabetic characters
- Uppercase characters
- Lowercase characters
- Titlecase characters
- Numeric characters
- Whitespace characters

classify_word.py

```
1.   #!/usr/bin/python
2.
3.   # Program that accepts a word and classifies it
4.
5.   word = input("Enter a word: ")
6.
7.   if word.isalnum():
8.       print("Alphanumeric")
9.       if word.isalpha():
10.          print("Alphabetic")
11.          if word.isupper(): print("Uppercase")
12.          elif word.islower(): print("Lowercase")
13.          elif word.istitle(): print("Titlecase")
14.      elif word.isnumeric(): print("Numeric")
15.  elif word.isspace(): print ("Whitespace")
```

Output:

```
Enter a word: Hello
Alphanumeric
Alphabetic
Titlecase
```

```
Enter a word: world
Alphanumeric
Alphabetic
Lowercase
```

```
Enter a word: hello123
Alphanumeric
```

```
Enter a word: 123
Alphanumeric
Numeric
```

Observation:

1. Refer to section 2.3.4.2 for a description of the functions used in this program.
2. Observe that we have an `if` statement within the body of an `if` statement. The `if` statement testing whether the characters are uppercase is within the body of the `if` statement testing whether the characters are alphabetic, which in turn is within the body of the `if` statement testing whether the characters are alphanumeric.
3. In those cases where the body of an `if` statement comprised of just a single `print()` call, we have used the simple statement syntax of the `if` statement. In all other cases, we have used the compound statement form.
4. Just in case you're wondering, there are words that are alphabetic but neither uppercase nor lowercase nor titlecase. Therefore we cannot use an `else` clause instead of the `elif` in line 13. An example of a word that is neither of these three is "`heLLo`"
5. Do observe the mutual exclusion of the conditions involved and why we have decided to build our conditions in this manner.

3.3 Loops

It is now time to explore the final basic control structure – loops. A loop is defined as a set of statements executed over and over again.

- A *finite loop* is one that runs a finite number of times and then passes control to the statement following it
- An *infinite loop* is one that never terminates (and therefore might require user intervention to terminate it along with termination of the script, like pressing control+C).
- A *definite loop* is one that runs a fixed number of times (and the number of times it runs is evident in the script)
- An *indefinite loop* is one that runs as long as a condition is satisfied and without knowing the actual data, is impossible to predict exactly how many times.

We will eventually see examples of all these types.

3.3.1 The while Loop

The simplest looping control structure in Python is the `while` loop. It's syntax is very similar to that of the `if` statement. While in the case of the `if` statement the condition is tested only once, in the case of the `while` statement the condition is tested till it evaluates to `False`, executing the body of the loop each time the condition evaluated `True`. The body of the loop can either be a simple statement or a compound

statement, just like the `if` statement. Each time the loops runs, it is counted as an iteration. Thus, a loop typically is used when we want multiple iterations, controlled using a condition.

The syntax of the `while` loop is shown below:

```
while condition: statement

while condition:
    statements
    ...
```

Let us write a program to print all integers from `1` to `n`, where `n` is given by the user.

while_demo.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of while loop
4.
5.   n = int(input("Enter a positive integer: "))
6.
7.   i=1
8.   while i<=n:
9.       print(i)
10.      i=i+1
```

Output:

```
Enter a positive integer: 7
1
2
3
4
5
6
7
```

Observation:

1. The loop runs as long as the condition `i<=n` is `True` and executes the body each time the condition evaluated to `True`.
2. The loop terminates the moment the condition evaluates to `False`. In this example, that happens when `i>n`.
3. In each iteration, we increment the variable `i`, as shown in line 10.

Note for C/C++/Java/Perl Programmers:

There is no increment operator (++) - we need to use the ordinary addition operator (+) in order to increment, as shown in line 10.

Let us write a program that tells whether a given number is prime or composite.

prime.py

```
1.   #!/usr/bin/python
2.
3.   # Program to classify a given number
4.   # as prime or composite
5.
6.   n = int(input("Enter a positive integer: "))
7.
8.   isPrime=True
9.   i=2
10.  while i<=n/2:
11.      if n%i==0: isPrime=False
12.      i=i+1
13.
14.  if isPrime: print("Prime")
15.  else: print("Composite")
```

Output:

```
Enter a positive integer: 7
Prime
```

```
Enter a positive integer: 8
Composite
```

Observation:

1. A number is prime if it has no other factors apart from `1` and itself. Thus, the first factor we need to check is `2` and the last factor we need to check is `n/2`, since the only factor after that will be `n`.
2. The number `1` is neither prime nor composite and as such is not expected as an input for our program.
3. Mathematically, we need not search for factors till `n/2`, we can search only till `√n`, which could significantly bring down the number of iterations required when a number is prime, but the savings might not be justifiable as finding the

square root is a relatively time consuming operation for the system.

4. We will nevertheless improvise this program as we learn more concepts.
5. We use a Boolean variable `isPrime` to keep track of the kind of number we have concluded. We initially assume the number is prime and conclude otherwise if and when we encounter a factor of the given number (indicated by `n%i==0`).

3.3.2 Using break in while

The `break` statement results in an immediate exit from a loop. This could be useful when another condition has been found in the body of a loop that should result in the immediate termination of the loop. On execution of the `break` statement, control flows to the first statement outside the loop.

A loop can have any number of `break` statements, but the execution of any one of them will prevent the execution of any other statement within the loop.

While the above program works correctly, it can be improvised. The first improvement suggested is based on the fact that we currently do not terminate the loop the moment the conclusion is ready. Line 11 can determine if the given number is not prime, but the loop continues nevertheless. We can terminate the loop when we have concluded that the number is not prime using the `break` statement as shown in the improvised program below:

prime2.py

```
1.   #!/usr/bin/python
2.
3.   # Program to classify a given number
4.   # as prime or composite
5.
6.   n = int(input("Enter a positive integer: "))
7.
8.   isPrime=True
9.   i=2
10.  while i<=n/2:
11.      if n%i==0:
12.          isPrime=False
13.          break
14.      i=i+1
15.
16.  if isPrime: print("Prime")
17.  else: print("Composite")
```

This improved program works just as the previous one and hence it's output is not shown here.

Observation:

1. The only change in the program is the addition of line 13, thereby changing the `if` statement in line 11 from the simple statement syntax to the compound statement syntax
2. The number of iterations can drop significantly if the number is composite. For example, if we consider the input to be `100`, while `prime.py` would have made 49 iterations (from 2 to 50), `prime2.py` will use only 1 iteration (with i=2)!
3. If the number is prime, however, the number of iterations will not change! For example, the prime number `101` will require 49 iterations (2 to 50) in both `prime.py` and `prime2.py`

3.3.3 The while Loop with else Clause

The `while` loop also supports an optional `else` clause that can enclose a body which will be executed if control came out of the loop because of the loop condition evaluating to `False`. Note that this `else` clause will not be executed if the control came out of the loop because of any other reason, including execution of a `break` statement within the body of the loop. It's syntax is shown below:

```
while condition:
    statements
    ...
else:
    statements
```

We could use the `else` clause to perform any action in response to the loop condition evaluating to `False`. Here is another improvisation over `prime.py`:

prime3.py

```
1.   #!/usr/bin/python
2.
3.   # Program to classify a given number
4.   # as prime or composite
5.
6.   n = int(input("Enter a positive integer: "))
7.
8.   i=2
9.   while i<=n/2:
10.      if n%i==0:
11.          isPrime=False
12.          break
13.      i=i+1
14.  else:
15.      isPrime=True
16.
17.  if isPrime: print("Prime")
18.  else: print("Composite")
```

Observation:

1. Observe that line 8 of `prime.py` has been removed! We are no longer starting with an assumption that the number is prime!
2. Line 14 introduces an else clause which will be executed only if and when the condition `i<=n/2` evaluates to `False` – and this condition implies that the number is prime as none of the numbers generated were factors. This conclusion is recorded in line 15.

3.3.4 The for Loop

The `for` loop is another looping control structure provided by Python and can be very powerful and convenient at the same time. The `for` loop basically iterates over a *sequence* of items, and each time an item is found in the sequence, it will store it in the *loop control variable* and executes the body of the loop. It's syntax is given below:

```
for var in sequence: statement

for var in sequence:
    statements
    ...
```

In the above syntax, `var` is the *loop control variable* (also called *loop index variable*). For each item in the sequence, the loop body will execute with `var` containing the

value of that item. The number of iterations of the loop will be equal to the number of items in the sequence. Sequences will be covered later in various sections of the book (since there are different types of sequences), but we will need one right now to show the working of the for loop. Therefore, before we go on to an example, let us cover the `range()` function that generates a sequence of numbers using an arithmetic progression.

3.3.5 Using range() in for Loops

Syntax:

```
range(end)
range(start,end)
range(start,end,step)
```

Form #1: `range(end)`

In the simplest form, the `range()` function generates a sequence of all numbers from `0` up to and excluding the number specified as shown in the example below:

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

Form #2: `range(start, end)`

The `range()` function can also receive the starting and ending numbers and generate all the numbers in between as shown below:

```
>>> for i in range(5,10): print(i)
...
5
6
7
8
9
```

NOTE:

When the syntax `range(start, end)` is used, the `range()` function generates all numbers from `start` (including `start`) till `end-1` (excludes `end`).

Form #3: `range(start, end, step)`

Finally, a third argument can be provided to the `range()` function to act as a step size or increment, as is evident from this example:

```
>>> for i in range (10,20,3): print(i)
...
10
13
16
19
```

As always, the ending number is excluded (if at all present in the sequence).

The step size can be negative in order to generate decreasing numbers as shown in the example below:

```
>>> for i in range (20,10,-2): print(i)
...
20
18
16
14
12
```

The following calls to `range()` will result in an empty sequence:

- `range(10,10)`
- `range(20,10)`
- `range(10,20,-2)`
- `range(20,10,2)`

3.3.6 Nested Loops

A loop within a loop is called a *nested loop*. It is possible to nest both `while` and `for` loops within both `for` and `while` loops and any number of levels of such nesting is possible.

Let us combine our knowledge of the `range()` function with the code in `prime3.py` to create a program that will display all prime numbers till a given number, using the concept of nested loops.

prime4.py

```
1.   #!/usr/bin/python
2.
3.   # Program to generate all prime numbers till a given
number
4.
5.   lim = int(input("Enter the limit: "))
6.
7.   for n in range(2,lim+1):
8.       i=2
9.       while i<=n/2:
10.          if n%i==0:
11.              break
12.          i=i+1
13.      else:
14.          print(n)
```

Output:

```
Enter the limit: 30
2
3
5
7
11
13
17
19
23
29
```

Observation:

1. Line 7 generates all numbers from `2` till the given number, `lim`. Note that we have used `lim+1` in the `range()` function because the `range()` function excludes the last value. The body of the loop is more or less the code from `prime3.py`.
2. We do not find the need to use the variable `isPrime` that we had used in `prime3.py` as is evident from the code above! This simplifies our code and eliminates a variable.
3. If we encounter a prime number, we print it as shown in line 14. If we encounter a composite number, we do nothing but simply continue on with the next number.

3.3.7 The for Loop With else Clause

The `else` clause that was available for `while` loops is also available on `for` loops in a similar fashion. Control will enter the `else` clause of the `for` loop only if and when the loop has exhausted all values from the sequence and reaches a natural termination. Control will not enter the `else` clause if the loop terminated prematurely because of execution of the `break` statement within the loop. The syntax is given below:

```
for var in sequence:
    statements
    ...
else:
    statements
    ...
```

To demonstrate the `else` clause, we will rewrite `prime4.py` using only `for` loops:

prime5.py

```
1.   #!/usr/bin/python
2.
3.   # Program to generate all prime numbers till a given
number
4.
5.   lim = int(input("Enter the limit: "))
6.
7.   for n in range(2,lim+1):
8.       for i in range(2,int(n/2)+1):
9.           if n%i==0:
10.              break
11.      else:
12.          print(n)
```

Observation:

1. The program works just like `prime4.py` and produces identical output. We have replaced the inner `while` loop with an inner `for` loop.
2. The `int()` function in line 8 is required as the `range()` function works only on integers. If `n` is odd, `n/2` will give rise to a fractional (`float`) value, which `range()` does not support. Of course, we can also elect to use the `//` operator to guarantee that we get an integer after division.
3. As usual, the "`+1`" in line 8 is because the `range()` function excludes the end value.

3.3.8 The continue Statement

Just as how the `break` statement can be used within the body of a loop in order to immediately terminate the loop, the `continue` statement can be used within the body of a loop to immediately continue with the next iteration of the loop, discarding the current iteration.

Let us modify `prime5.py` to print only those primes that do not end with the digit `3`:

prime6.py

```
1.   #!/usr/bin/python
2.
3.   # Program to generate all prime numbers till a given
number,
4.   # excluding those primes that end with the digit 3.
5.
6.   lim = int(input("Enter the limit: "))
7.
8.   for n in range(2,lim+1):
9.       if n%10 == 3: continue
10.      for i in range(2,int(n/2)+1):
11.          if n%i==0:
12.              break
13.      else:
14.          print(n)
```

Output:

```
Enter the limit: 30
2
5
7
11
17
19
29
```

Observation:

1. Observe that this program does not print those primes that end with 3 – in this example, we observe that 3 and 13 are not present in the output despite being prime because they end with the digit 3.
2. The only change introduced from `prime5.py` to `prime6.py` is in line 9.
3. A number that ends with the digit `3` gives a remainder of `3` when divided by `10` – that is the condition used in line 9

4. For all those numbers that end with the digit `3`, we continue immediately with the next iteration. We do not bother to even find out whether the number was prime or not.
5. Of course, we can also write the program in such a way that we first find out if the number is prime, and if so then check in line 13 using an `elif` instead of the `else` whether the number ends with the digit `3` or not, and then decide whether to print it.

3.3.9 More Programs on Loops

Let's write a Python program that accepts a number and prints the sum of it's digits. Thus, given a number `789`, the output should be `24` (`7+8+9`). While there are many different ways of achieving this in Python, let's do it this way: in a loop we keep extracting the unit's place of the number `n` (by using the value of the expression `n%10`) while reducing the number to eliminate the unit's place (by using the expression `n//10`). The loop will run as many times as the number of digits in it (till `n==0`). Add each digit to a variable `sum` and finally print `sum` outside the loop.

sum_digits.py

```
1.   #!/usr/bin/python
2.
3.   # Program to print the sum of digits of a given number
4.
5.   n=int(input("Enter an integer: "))
6.   sum=0
7.
8.   while n>0:
9.       digit=n%10
10.      sum+=digit
11.      n//=10
12.
13.  print("The sum is",sum)
```

Output:

```
Enter an integer: 789
The sum is 24
```

Observation:

1. In line 11, we use the `//` operator to perform integer division. Furthermore, `//=` will integer divide the LHS by the RHS and store the result in the LHS.
2. The program works even for the special case where `0` is the input. The loop body will not get executed and `sum` will be printed directly (and `sum` is initialized to `0`).

We can use the digit splitting logic to reverse a number!

num_reverse.py

```
1.   #!/usr/bin/python
2.
3.   # Program to print the reverse of a given number
4.
5.   n=int(input("Enter an integer: "))
6.   reverse=0
7.
8.   while n>0:
9.       digit = n%10
10.      reverse = reverse*10 + digit
11.      n//=10
12.
13.  print("The reverse is",reverse)
```

Output:

```
Enter an integer: 789
The reverse is 987
```

Observation:

1. This program is based on `sum_digits.py`, with `sum` replaced by `reverse` and a change in line 10.
2. As an example, the reverse of the number `789` can be mathematically obtained as `((9*10)+8)*10+7`. That is the logic used in line 10.

How about using the previous program to determine whether a number is a palindrome or not? A palindrome number will be equal to it's reverse!

palindrome.py

```
1.   #!/usr/bin/python
2.
3.   # Program to determine if a given number
4.   # is a palindrome or not.
5.
6.   n=int(input("Enter an integer: "))
7.   num=n
8.   reverse=0
9.
10.  while n>0:
11.      digit = n%10
12.      reverse = reverse*10 + digit
13.      n//=10
14.
15.  if num==reverse: print("The number is a palindrome")
16.  else: print("The number is not a palindrome")
```

Output:

```
Enter an integer: 789
The number is not a palindrome
```

```
Enter an integer: 1221
The number is a palindrome
```

As a final example, let's write a program that will print all 3-digit Armstrong numbers – numbers which are equal to the sum of the cubes of it's digits. For example, 153=1+125+27.

armstrong.py

```
1.   #!/usr/bin/python
2.
3.   # Program to print all 3-digit Armstrong numbers
4.
5.   for num in range(100,1000):
6.       sum=0
7.       n=num
8.       while n>0:
9.           sum += (n%10)**3
10.          n //= 10
11.
12.      if num==sum: print(num)
```

Output:

```
153
370
371
407
```

Observation:

1. We use the `range()` function to generate all 3-digit integers – from `100` (inclusive) to `1000` (exclusive) in line 5.
2. We use the same logic that we had used in `sum_digits.py` to extract the digits from the number and find the sum – except that the sum here is the sum of the cubes of the digits rather than the sum of the digits (line 9).
3. We could have directly used a formula like `(n//100)**3 + (n//10%10)**3 + (n%10)**3` to find the sum of the cubes of the digits of a given number `n`. A loop not only increases the readability, but is also general enough to work with any number of digits, if required.

3.4 Terminating Control

While all the previous sections concentrated on controlling the flow of control *down* in the program, this section emphasises on removing control *away* from a control structure!

We can remove control out of:

1. A loop
2. A function
3. A block
4. A script

3.4.1 Terminating a Loop

Section 3.3.2 already introduced the `break` statement for terminating a loop.

3.4.2 Terminating a Function

The `return` statement is used to terminate a function and is covered in section 10.8.

3.4.3 Terminating a Block

The `raise` statement is used to raise an exception that has the potential of terminating the following:

1. a block
2. a function
3. a sequence of function calls
4. a script

The `raise` statement is covered in detail under exception handling (section 13). A form of `raise` will nevertheless be covered in the next section, though.

3.4.4 Terminating a Script

To terminate an *interactive Python session*, the following functions can be called:

1. quit()
2. exit()

To terminate a *script* however, it is recommended to use `sys.exit()` as shown below:

```
>>> import sys
>>> sys.exit()
```

Observation:

1. We need to import the `sys` module that contains the `exit()` function!
2. Ideally, we should pass an *exit code* as an argument to `sys.exit()` that can tell others the reason for our script's termination! An exit code of `0` is assumed to mean a normal, successful termination whereas a non-zero exit code is assumed to mean an error or unsuccessful termination! When an argument is not passed to `sys.exit()`, it is assumed to mean a normal, successful termination, equivalent to `sys.exit(0)`!

All the approaches above end up terminating the script by raising the `SystemExit` exception, technically. This means that as a programmer, you could as well terminate your script anytime by executing the following statement:

```
>>> raise SystemExit
```

Note:

1. The statement `raise SystemExit` does not require any modules to be imported and is technically the way a Python script or interpreter terminates normally anyway!
2. The most professional way of terminating a Python script is by invoking `sys.exit()`, passing an exit code if non-zero.

3.5 Questions

1. Write a short note on all forms of the `if` statement of Python along with syntax and examples.
2. Write a short note on all forms of the `while` statement of Python along with syntax and examples.
3. Explain the `break` and `continue` statements with examples.
4. Write a short note on terminating control that includes the following:
    1. Terminating a Loop
    2. Terminating a Function
    3. Terminating a Block of Code
    4. Terminating a Script
5. Write a short note on the `range()` function with examples.

3.6 Exercises

1. Write a program to tell if a given number is negative, zero or positive.
2. Write a program to print whether a given number is even or odd.
3. Write a program to print all the even and odd numbers (two separate sequences) up to given a number.
4. Write a program to check if the given character is an alphabet, digit or a punctuation character.
5. Write a program to check if the given alphabet is a consonant or a vowel.
6. Write a program to count the number vowels and consonants in the given input string.
7. Write a program to check if the given number is prime or not.
8. Write a program to print all the prime numbers up to a given number.
9. Write a program to check if the given year is a leap year or not.
10. Write a program to check if the given string is a palindrome.
11. Write a program to print factorial of a given number.
12. Write a program to generate the first `n` terms of the Fibonacci series.
13. Write a program that finds the factors for numbers between `2` and `10`. Also print the numbers that are prime.
14. Write a program to print the grade for a given percentage of marks. The grade should be A for marks>=80, B for marks>=60, C for marks>=40 and D otherwise.
15. Write a program to find if the given point lies on the x-axis, y-axis or on the origin, given it's coordinates.
16. Write a program to find the distance between the two points. User should enter the coordinates of the two points.
17. Write a program to print if the given triangle is an equilateral triangle, isosceles triangle or scalene triangle. User should enter the values for the sides of the triangle.

SUMMARY

- Python relies on indentation rules to figure out the body of any construct. Generally, programmers use a single tab or 4 spaces to identify a single level of indentation.
- The if-elif-else construct permits testing of various conditions and execution of a single block of code based on the truth value of the condition(s).
- Python does not permit empty blocks - such blocks should have at least the pass statement.
- The while-else construct permits the execution of a block of code as long as a given condition evaluates to True.
- The for-else construct permits the execution of a block of code for each element of a sequence.
- The break statement can be used to prematurely terminate a loop and the continue statement can be used to prematurely terminate the current iteration of a loop.
- The range() function can be used in conjunction with the for construct to loop through a range of values.
- A script can be terminated anytime using sys.exit().

4 Lists

In this chapter you will be able to:

- ☑ Create lists and access their elements.
- ☑ Iterate through lists and search for specific elements.
- ☑ Extract sub-lists from lists using the concept of list slices.
- ☑ Add, Delete and Modify elements in a list.
- ☑ Deal with nested lists and write programs to solve problems using lists.

Lists

A `list` in Python is defined as an ordered sequence of elements that can be dynamically altered.

- *Ordered* means that each item in a list has an *index* based on it's position in the list.
- *Sequence* means that the elements are arranged in order, based on their indices, and a sequential traversal through the list will give us the elements in the order of their indices.
- The phrase "*dynamically altered*" means that each item in the list can be changed and the list itself can be changed – more items can be added, and existing items can be replaced or removed.

For C/C++/Java programmers:

A list is perhaps the closest alternative to what C/C++/Java programmers call arrays.

Lists are dynamically growable and shrinkable – elements can be added and removed on the fly.

4.1 Creating Lists

Just as how an integer object can be created and assigned to a variable using the `int()` function, a list object can be created and assigned to a variable using the `list()` function:

```
>>> x=int(5)
>>> x
5
>>> L=list([5,2,3])
>>> L
[5, 2, 3]
```

Since a `list` is a collection of values, the square brackets (`[]`) are used to enclose the collection, and are always used in Python to denote a list. In fact, just as how we have integer literals whose data type is understood to be `int`, we have list literals that use `[]`, whose data type is understood to be `list`:

```
>>> x=5
>>> x
5
>>> L=[5,2,3]
>>> L
[5, 2, 3]
```

From now on, we will stick to the `[]` syntax for convenience and brevity, unless forced to use the `list()` function.

NOTE:

While `list` is a data type, the contents of a list are objects. Thus, there is no distinction like `list` of `int` vs. `list` of `float`.

4.2 Accessing List Elements

Each element in a list has an index which can be used to identify the element. This index increases from left to right, starting from `0`. Negative indices are relative to the end of the list, and increase in absolute magnitude from right to left, starting with `-1` from the end of the list. Thus, each element in a list has 2 indices associated with it – a non-negative one that is counted from the beginning of the list and a negative one that is counted from the end of the list. Together with the `[]` operator, we can identify any element in the list as shown in the examples below:

```
>>> L=[5,2,3]
>>> L[0]
5
>>> L[1]
2
>>> L[2]
3
>>> L[-1]
3
>>> L[-2]
2
>>> L[-3]
5
```

Elements of a list can be freely modified:

```
>>> L=[5,2,3]
>>> L
[5, 2, 3]
>>> L[0]=0
>>> L
```

```
[0, 2, 3]
>>> L[0]=L[1]+L[2]
>>> L
[5, 2, 3]
```

4.3 Counting List Elements

The `len()` function is available to count the number of elements in a list. The minimum size of any list is 0, corresponding to an empty list (denoted by `[]`).

```
>>> L=[5,2,3]
>>> L
[5, 2, 3]
>>> len(L)
3
>>> L=[5]
>>> L
[5]
>>> len(L)
1
>>> L=[]
>>> L
[]
>>> len(L)
0
```

4.4 Iterating Through List Elements

One of the most practical operations we end up performing on lists is iterating through them and processing each element in sequence. Since lists are sequences, the `for` loop is directly compatible for iterating as shown in the example below:

```
>>> L=[5,2,3]
>>> for i in L: print(i)
...
5
2
3
```

4.5 Searching Elements within Lists

Given that a list can contain multiple items, we may need to know if a particular element is present in a list or not, and if present, how many times and where all in the list. There are functions for all of these in Python.

4.5.1 Checking for Existence

The `in` and `not in` operators help us to verify whether a particular element is present in a list or not.

```
>>> L=[5,2,3]
>>> 3 in L
True
>>> 4 in L
False
>>> 4 not in L
True
```

4.5.2 Counting Occurrences

The `count()` member function of list tells us how many instances of the specified object is present in the list.

```
>>> L=[5,2,3,2]
>>> L.count(3)
1
>>> L.count(2)
2
>>> L.count(4)
0
```

4.5.3 Locating Elements

The `index()` member function of list searches for the first occurrence of an element within a list and returns the index where it was found, and throws a `ValueError` if not found. It's complete syntax is:

```
list.index(x[,i[,j]])
```

Form #1: `list.index(x)`

This searches for the first occurrence of the element `x` in the list `list` and returns the index where it was found or raises a `ValueError` if the element was not found, as shown in the example below:

```
>>> L=[5,2,3,2]
>>> L.index(2)
1
>>> L.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 4 is not in list
```

Form #2: `list.index(x,i)`

If the second argument (`i`) is given, the search starts from index `i` instead of `0`. This allows us to search for other occurrences of an element or to skip some initial elements in the search. In all other respects, this form is similar to form #1.

```
>>> L=[5,2,3,2]
>>> L.index(2,2)
3
```

In the above example, the first occurrence of `2` is in index `1`. If we want to search for the next occurrence of `2`, we need to start searching from index `2` onwards, which is what the example does.

Form #3: `list.index(x,i,j)`

If the third argument (`j`) is given, the search will start at index `i` but will stop at index `j` (excluding index `j`). Remember that if an element is not found when it stops, it will throw `ValueError`.

```
>>> L=[5,2,3,2]
>>> L.index(3,0,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 3 is not in list
```

In the above example, we are searching for the element `3` in the list `L` starting from index `0` and restricting our search to index `2` (`2` excluded). Thus, our search is restricted to the indices `0` and `1`, and the element `3` is not found in these indices.

4.6 List Slices

A list slice is a sub-list extracted from a list – it is a list comprising of a part of an existing list. A list slice uses syntax `list[start:end]`, where `list` is the parent list from which to extract elements, `start` is the starting index from where to start extracting elements and `end` is the ending index (excluding `end`) till where to extract the elements. The parent list does not undergo any change unless an assignment is made to the list slice. Both `start` and `end` are optional, with `start` defaulting to `0` (index of the first element) and `end` defaulting to `len(list)`, i.e. the end of the list. Some examples follow:

```
>>> L=[5,2,3,2]
>>> L[1:3] # Slice of L from 1 to 3 (exclusive)
[2, 3]
>>> L[1:] # Slice of L from 1 till the end of the list
[2, 3, 2]
>>> L[:3] # Slice of L from the beginning of the list till 3
[5, 2, 3]
>>> L[:] # Slice spanning the entire list
[5, 2, 3, 2]
```

In case you are of the opinion that `L[:]` is the same as `L`, know that they are not and will be dealt with in detail in section 4.8.3.

Let us now see examples of using list slices to modify the contents of a list (with the changes underlined for better recognition):

The first example shows how to replace a few elements of a list with a different set of elements without changing the list size:

```
>>> L=[5,2,3,2]
>>> L[1:3]=[3,2]
>>> L
[5, 3, 2, 2]
```

The number of elements assigned to a list slice need not be of the same size! Here is an example of putting in more number of elements than the number of elements deleted:

```
>>> L=[5,2,3,2]
>>> L[1:3]=[3,0,0,2]
>>> L
[5, 3, 0, 0, 2, 2]
```

Similarly, here is an example of putting in fewer number of elements than the number of elements deleted:

```
>>> L=[5,2,3,2]
>>> L[1:3]=[9]
>>> L
[5, 9, 2]
```

Let's discuss some special cases now!

```
>>> L=[5,2,3,2]
>>> L[1:2]=[9]
>>> L
```

```
[5, 9, 3, 2]
```

The above example, though valid, is an overly complicated way of doing a simple operation of `L[1]=9`!

```
>>> L=[5,2,3,2]
>>> L[1:1]=[9,8,7]
>>> L
[5, 9, 8, 7, 2, 3, 2]
```

The above example shows that it is not necessary to really replace elements – elements can be merely inserted.

```
>>> L=[5,2,3,2]
>>> L[1:2]=[]
>>> L
[5, 3, 2]
```

Similarly, the above example shows that it is not necessary to insert anything – elements can be simply deleted.

```
>>> L=[5,2,3,2]
>>> L[2:]=[]
>>> L
[5, 2]
```

The above example shows how to truncate a list.

```
>>> L=[5,2,3,2]
>>> L[:2]=[]
>>> L
[3, 2]
```

The above example similarly removes the initial indices that are less than `2`.

And our last example in this section:

```
>>> L=[5,2,3,2]
>>> L[:]=[]
>>> L
[]
```

The above example clears the list (deletes all elements from the list). We will cover more techniques for doing this later on.

4.7 Adding and Deleting Elements

We have already seen how we can add and delete elements from a list using assignment to list slices. There are better, more readable alternatives to do that, which will be explored now.

4.7.1 Appending Elements

To add one or more elements to the end of a list, the following functions can be used:

```
list.append(x)
```

The `list.append(x)` function appends the element `x` to the list `list`:

```
>>> L=[5,2,3]
>>> L.append(9)
>>> L
[5, 2, 3, 9]
```

NOTE:

`list.append(x)` is similar to `list[len(list):]=[x]`

```
list.extend(L)
```

The `list.extend(L)` function appends all the elements of the list `L` to the list `list`:

```
>>> L1=[5,2,3]
>>> L2=[7,8,9]
>>> L1.extend(L2)
>>> L1
[5, 2, 3, 7, 8, 9]
```

NOTE:

`list.extend(L)` is similar to `list[len(list):]=L`

4.7.2 Inserting Elements

```
list.insert(i,x)
```

To add elements at any desired location within the list, the `list.insert(i,x)` function can be used, which inserts the element `x` at index `i` within the list `list`:

```
>>> L=[5,2,3]
>>> L.insert(2,9)
>>> L
[5, 2, 9, 3]
```

NOTE:

`list.insert(i,x)` is similar to `list[i:i]=[x]`

`list.append(x)` is similar to `list.insert(len(list),x)`

Note that inserting at an index beyond the end of the list will end up in an insertion at the end of the list (an append operation):

```
>>> L=[5,2,3,2]
>>> L.insert(50,9)
>>> L
[5, 2, 3, 2, 9]
```

4.7.3 Deleting Elements

There are many different ways of deleting elements from a list. Before we explore these, keep in mind that deleting elements from a list is different from replacing elements in a list, something that we have already covered in section 4.2.

4.7.3.1 Using del to Delete an Element

The first way to delete an element at a specific position in the list is by using the `del` statement, similar to the way we had used it for deleting variables in section 2.6:

```
>>> L=[5,2,3]
>>> del L[1]
>>> L
[5, 3]
```

The `del` statement can also be used to delete a range of elements from a list:

```
>>> L=[5,2,3,7,8,9]
>>> del L[1:4]
>>> L
[5, 8, 9]
>>> L=[5,2,3,7,8,9]
>>> del L[1:5:2]
>>> L
[5, 3, 8, 9]
```

In the last example where we have used `del L[1:5:2]`, it will delete elements from index `1` onwards till index `5` (excluding `5`) in steps of `2`.

As a special case of the above, `del` can be used to remove all elements of a list, retaining the list variable and making the list empty:

```
>>> L=[5,2,3,2]
>>> del L[:]
>>> L
[]
```

Finally, the `del` statement can be used to delete the list variable itself:

```
>>> L=[5,2,3]
>>> del L
>>> L
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'L' is not defined
```

4.7.3.2 Using remove() to Delete an Element

```
list.remove(x)
```

If we wish to delete a specific element regardless of it's position, we can use the `list.remove(x)` which searches and removes the first occurrence of `x` in the list `list`. If the element `x` is not present in the list, it will throw a `ValueError`:

```
>>> L=[5,2,3,2]
>>> L.remove(2)
>>> L
[5, 3, 2]
>>> L.remove(2)
>>> L
[5, 3]
```

```
>>> L.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

4.7.3.3 Using pop() to Delete Elements

Another method to delete an element at a specific position or at the end of a list and return the element that was deleted is by using the `list.pop()` function, whose syntax is shown below:

```
list.pop([i])
```

If an index (`i`) is provided, it will delete the element at index `i` in list `list` and returns the element that was deleted. If no index is provided, it will delete the last element in the list `list` and returns the element that was deleted:

```
>>> L=[5,2,3,2]
>>> L.pop(1)
2
>>> L
[5, 3, 2]
>>> L.pop()
2
>>> L
[5, 3]
>>> L.pop(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

As can be seen from the last example above, giving an invalid index to `pop()` results in an `IndexError`!

4.7.3.4 Using clear() To Delete All Elements of a List

```
list.clear()
```

While most of the functions discussed earlier delete one or more elements from a list, the `list.clear()` function deletes all elements from the list `list` and makes it empty:

```
>>> L=[5,2,3,2]
>>> L.clear()
>>> L
[]
```

NOTE:

`L.clear()` is similar to `del L[:]`

4.8 Adding, Multiplying and Copying Lists

The `+` (addition), `+=` (addition and assignment), `*` (multiplication), `*=` (multiplication and assignment) and `=` (assignment) operators are available to operate on lists.

4.8.1 Adding Lists

Two lists can be concatenated to form a third list using the `+` operator:

```
>>> L1=[1,3]
>>> L2=[7,8]
>>> L3=L1+L2
>>> L3
[1, 3, 7, 8]
```

The `+=` operator will concatenate the list on the RHS to the list on the LHS:

```
>>> L=[1,3]
>>> L+=[7,8]
>>> L
[1, 3, 7, 8]
```

NOTE:

`L1 += L2` is similar to `L1.extend(L2)`

4.8.2 Multiplying Lists

A list `L` can be multiplied by an integer `n` to denote the list `L` repeated `n` times:

```
>>> L=[5,2,3]*3
>>> L
[5, 2, 3, 5, 2, 3, 5, 2, 3]
```

NOTE:

`list * n` is the same as `n * list`!

Similarly, the `*=` operator repeats the list on LHS, RHS number of times, and assigns the resultant list to the LHS variable:

```
>>> L=[5,2,3]
>>> L*=3
>>> L
[5, 2, 3, 5, 2, 3, 5, 2, 3]
```

4.8.3 Assigning and Copying Lists

The `=` operator allows us to assign a list to a list variable. However, this is not the same as copying elements of a list to another and is demonstrated in the example below. Recall from section 2.6 that variables merely hold references to objects – lists are objects and list variables are references to such objects, and the `=` operator only copies the reference, not the objects!

```
>>> L1=[5,2,3]
>>> L2=L1
>>> L2
[5, 2, 3]
>>> L1[0]=9
>>> L1
[9, 2, 3]
>>> L2
[9, 2, 3]
```

In order to copy all the elements of a list to another (copying the list object instead of the list reference), the `list.copy()` function can be used as follows:

```
>>> L1=[5,2,3]
>>> L2=L1.copy()
>>> L2
[5, 2, 3]
>>> L1[0]=9
>>> L1
```

```
[9, 2, 3]
>>> L2
[5, 2, 3]
```

NOTE:

`L2=L1.copy()` is similar to `L2=L1[:]`

Another form of copying is copy list elements to list elements. Here is an example:

```
>>> [x,y]=[7,8]
>>> x
7
>>> y
8
```

Each element of the RHS list is copied to the corresponding variable in the LHS list. A simpler way to achieve this will be covered in section 5.7.3. But do note that the sizes of the two lists have to be the same for such an assignment to work.

4.9 Miscellaneous Operations on Lists

We have examined some core functionality of lists. This part will introduce some more additional functionality, and more advanced operations will be introduced in section 11.

4.9.1 Finding the Smallest Element in a List

The `min()` function returns the smallest element in the list:

```
>>> L=[5,2,3]
>>> min(L)
2
```

4.9.2 Finding the Largest Element in a List

The `max()` function returns the largest element in the list:

```
>>> L=[5,2,3]
>>> max(L)
5
```

4.9.3 Reversing a List

The `list.reverse()` function reverses the order of elements in the list `list`:

```
>>> L=[5,2,3]
>>> L.reverse()
>>> L
[3, 2, 5]
```

4.9.4 Sorting a List

The list can be sorted using the `list.sort()` function as shown in the example below. Note that all elements will be reordered to lie in ascending order in the list.

```
>>> L=[5,2,3]
>>> L.sort()
>>> L
[2, 3, 5]
```

The list elements can be sorted in descending order as follows:

```
>>> L=[5,2,3]
>>> L.sort(reverse=True)
>>> L
[5, 3, 2]
```

Observation:

1. `L.sort()` is the same as `L.sort(reverse=False)`
2. The `list.sort()` function changes the order of elements in the original list. In order to retrieve a sorted list without affecting the original list, the `sorted(list)` function can be used, which sorts the elements of the list `list` in ascending order and returns the new list without affecting the list `list`. In case we wish to sort the elements in descending order, we can use `sorted(list,reverse=True)`.
3. Sorting a list in reverse order is functionally identical to sorting it and then reversing it. This, `list.sort(reverse=True)` is functionally identical to the sequence of statements `list.sort()` followed by `list.reverse()`.

4.10 Simple Programs Based on Lists

One of the most basic uses of lists is to store “items” and then process them sequentially. So let us start with a program that stores the names of a few countries and prints the name of those countries whose length is greater than 5:

countries.py

```
1.   #!/usr/bin/python
2.
3.   # Program to store country names and print the names
4.   # of those countries whose length is greater than 5.
5.
6.   countries=["India","Pakistan","Sri Lanka","China"]
7.
8.   for country in countries:
9.       if len(country)>5: print(country)
```

Output:

```
Pakistan
Sri Lanka
```

Let us improve the above program and let the user type in the names of as many countries as required – entering them one by one and typing in "`end`" when done. We will then print the names of those countries whose length is greater than 5.

countries2.py

```
1.   #!/usr/bin/python
2.
3.   # Program to accept country names and print the names
4.   # of those countries whose length is greater than 5.
5.
6.   countries=[]
7.
8.   while True:
9.       country=input("Enter a country name (or 'end' to
terminate): ")
10.      if country=="end": break
11.      countries.append(country)
12.
13.  for country in countries:
14.      if len(country)>5: print(country)
```

Output:

```
Enter a country name (or 'end' to terminate): India
Enter a country name (or 'end' to terminate): Pakistan
Enter a country name (or 'end' to terminate): Sri Lanka
```

```
Enter a country name (or 'end' to terminate): China
Enter a country name (or 'end' to terminate): end
Pakistan
Sri Lanka
```

Observation:

1. The program is similar to `countries.py`, except for the addition of lines 8-11 and initializing the list `countries` to an empty list in line 6.
2. We use the `list.append()` function in line 11 to add a new country to the list `countries`.
3. We break out of the loop in line 10 if and when we encounter "`end`" from the user.

Let us make the program more productive: we will now receive the names of countries from the user and print the names of their capitals! To do this, we need to have a "database" of country capitals with us. We can store the names of capitals is a list, say called `capitals`. We will also store the names of the corresponding countries in a list, say `countries`, in the same order as the capitals stored in capitals. Thus, for a given index `i`, `countries[i]` will tell us the country name while `capitals[i]` will tell us the corresponding capital. We will then accept a country name from the user, search for that country in our list countries using the index function, use the retrieved index to locate the capital in the capitals list and print out the information. We will repeat this in a loop till the user enters "`end`" as in the previous program.

countries3.py

```
1.   #!/usr/bin/python
2.
3.   # Program to accept country names and print their capitals
4.
5.
6.   countries=["India","Pakistan","Sri Lanka","China"]
7.   capitals=["New Delhi","Islamabad","Sri Jayawardenepura
Kotte","Beijing"]
8.
9.   while True:
10.      country=input("Enter a country name (or 'end' to
terminate): ")
11.      if country=="end": break
12.      index=countries.index(country)
13.      capital=capitals[index]
14.      print("The capital of {} is
{}".format(country,capital))
```

Output:

```
Enter a country name (or 'end' to terminate): India
The capital of India is New Delhi
Enter a country name (or 'end' to terminate): China
The capital of China is Beijing
Enter a country name (or 'end' to terminate): end
```

Observation:

1. The above program will report a `ValueError` and terminate if a country name that is not present in our list `countries` is entered. Section 7.7 will show how we can respond to cases where the country is not listed with us.
2. We will see a better way of implementing this program in section 5.10.

Similarly, we'll now write a program that accepts a single-digit number from the user and prints it in words. We will store these words in a list and use the digit as an index.

digit2word.py

```
1.   #!/usr/bin/python
2.
3.   # Program to accept a single digit and print it in words
4.
5.   digit=int(input("Enter a digit: "))
6.
7.   words=["zero","one","two","three","four",\
8.       "five","six","seven","eight","nine"]
9.
10.  print("{} is {}".format(digit,words[digit]))
```

Output:

```
Enter a digit: 5
5 is five
```

Observation:

1. The list defined in line 7 has been split into two lines since it was long, to aid readability.
2. Any input other than a digit in the range of `0` to `9` will result in an error.

Let us build on the previous example to write a program that converts a number into words! Thus, `1234` will be printed as `one thousand two hundred and thirty`

`four`. For simplicity, we'll limit the number to `9999`.

num2words.py

```
1.   #!/usr/bin/python
2.
3.   # Program to print a number in words
4.
5.   onesWords=('zero','one','two','three','four',\
6.       'five','six','seven','eight','nine')
7.   tensWords=('','ten','twenty','thirty','forty',\
8.       'fifty','sixty','seventy','eighty','ninety')
9.   teensWords=('','eleven','twelve','thirteen','fourteen',\
10.      'fifteen','sixteen','seventeen','eighteen','nineteen')
11.
12.  n=int(input("Enter an integer: "))
13.
14.  if n==0: print("Zero")
15.  else:
16.      thousands=n//1000
17.      hundreds=n//100%10
18.      tens=n//10%10
19.      ones=n%10
20.
21.      if thousands>0: print("{} thousand
".format(onesWords[thousands]),end='')
22.      if hundreds>0: print("{} hundred
".format(onesWords[hundreds]),end='')
23.      if (thousands>0 or hundreds>0) and (tens>0 or ones>0):
print("and ",end='')
24.      if tens==1 and ones>0: print(teensWords[ones],end='')
25.      else:
26.          if tens>0: print(tensWords[tens],'',end='')
27.          if ones>0: print(onesWords[ones],end='')
28.      print()
```

Observation:

1. the tuple `onesWords` (line 5) keeps track of the words for single digit numbers. The tuple `tensWords` (line 7) keeps track of the words for describing tens. The tuple `teensWords` (line 9) keeps track of the words for 11 to 19. These are the words that will be used for output. Additional words that can be included are "`thousand`", "`hundred`" and "`and`", apart from spaces to separate the words. In these tuples, note that the index 0 is generally unused, but present nevertheless to ensure that the indices work correctly.
2. The thousands part, hundreds part, tens part and units part are extracted in

lines 16-19.

3. We need to print an "`and`" between the hundreds and tens part if either a thousands or hundreds part exists and a tens or units part exists. This is done in line 23.
4. Since we are printing in stages using multiple `print()` calls, and the `print()` function prints a newline at the end by default which we do not want, we have used `end=''` to prevent anything from being printed after every `print()` call, and have finally called `print()` in line 28 to print a newline.
5. The input `0` is a special case and hence is handled in line 14.

More powerful and practical examples will be seen in section 11.

4.11 Nested Lists

A list contains elements which are references to objects, and these objects could be of any type, including lists. A list within a list is called a *nested list*. Thus, a nested list comprises of elements, some (or possibly all) of which are lists themselves.

In the case of a nested list, the list that contains other lists can be designated to be the *outer list* whereas each list inside this outer list can be designated to be an *inner list*. Each inner list behaves both as an element of the outer list as well as a list by itself. This is demonstrated below:

```
>>> L1=[1,2,3]
>>> L2=[4,5]
>>> L3=[6,7,8,9]
>>> L=[0,L1,L2,L3,100]
>>> L
[0, [1, 2, 3], [4, 5], [6, 7, 8, 9], 100]
```

Observation:

1. `L1`, `L2` and `L3` are lists
2. `L` is a list containing 5 elements, including `L1`, `L2` and `L3`. This makes `L` an outer list with `L1`, `L2` and `L3` being inner lists.

Let us continue with the above illustration and prove that indeed the outer list is considered to have 5 elements, 3 of which are lists:

```
>>> L1=[1,2,3]
>>> L2=[4,5]
>>> L3=[6,7,8,9]
>>> L=[0,L1,L2,L3,100]
```

```
>>> L
[0, [1, 2, 3], [4, 5], [6, 7, 8, 9], 100]
>>> len(L)
5
>>> type(L)
<class 'list'>
>>> type(L[0])
<class 'int'>
>>> type(L[1])
<class 'list'>
```

Observation:

1. We can verify that `len(L)` gives us 5 – all elements are counted equal irrespective of their type.
2. We can verify that the type of `L[0]`, the first element of the outer list, which is an integer, is `class 'int'`. However, the type of `L[1]`, the second element of the outer list, which is a list itself, is `class 'list'`.

Just as how we can index elements of the outer list, we can also index elements of the inner list as well, as shown in the example below:

```
>>> L1=[1,2,3]
>>> L2=[4,5]
>>> L3=[6,7,8,9]
>>> L=[0,L1,L2,L3,100]
>>> L[3]
[6, 7, 8, 9]
>>> L[3][1]
7
```

Observation:

1. `L[3]` refers to the index `3` (fourth element) of the outer list, which is an inner list with contents `[6,7,8,9]`.
2. Index `1` (second element) of this inner list is `7`.
3. Thus `L[3]` is our list `L3` and `L[3][1]` is the same as `L3[1]`, which is `7`.

We can nest lists within lists to any depth depending on our requirement! This is illustrated below:

```
>>> L=[4,[6,3,[7,2,5,[7,7,6,[4,1,0]],8,4],9],2]
>>> L
[4, [6, 3, [7, 2, 5, [7, 7, 6, [4, 1, 0]], 8, 4], 9], 2]
>>> L[1]
[6, 3, [7, 2, 5, [7, 7, 6, [4, 1, 0]], 8, 4], 9]
>>> L[1][2]
[7, 2, 5, [7, 7, 6, [4, 1, 0]], 8, 4]
>>> L[1][2][3]
[7, 7, 6, [4, 1, 0]]
>>> L[1][2][3][3]
[4, 1, 0]
>>> L[1][2][3][3][0]
4
```

4.12 Questions

1. Define a list in Python.
2. Explain the different ways of searching an element within a list.
3. Write a short note on list slices.
4. Differentiate between the `append()` and `extend()` methods of list with examples.
5. Differentiate between `del` and `pop()` on a list with examples.
6. Differentiate between the `remove()` and `pop()` methods on lists with examples.
7. Write a short note on operators that can be used on lists.
8. How is list assignment different from the list `copy()` method? Explain with examples.
9. Write a short note on sorting of lists.

4.13 Exercises

1. Write a program to add all the items in a list.
2. Write a program to get the largest and smallest number in a list.
3. Write a program to remove duplicates from a list.
4. Write a program that prints whether at least one member of the given two lists are same.
5. Write a program to convert a list of characters into a string.
6. Write a program to check whether two lists are circularly identical.

7. Write a program to print unique values from a list.
8. Write a program to print the number of occurrences of each element in a list.
9. Write a program to find the sub-list in a list of lists whose sum of elements is the highest.
10. Write a program to insert a given string at the beginning of all the strings in a list.
11. Write a program to convert a string to a list of characters.
12. Write a program to concatenate elements of a list with corresponding elements of another given list.
13. Write a program to split a list every n-th element to obtain a list of lists.
14. Write a program to convert a list of multiple integers into one single integer. Thus, the list [25,12,2] should get converted to the number 25122.

SUMMARY

- A list can be simply created using []. Elements of a list can be accessed using [].
- The len() function gives the number of elements in a list and the count() method returns the number of occurrences of a particular element in the list.
- The for loop can be naturally used to iterate through the elements of a list.
- The in and not in operators check for the existence of an element in a list. The index() method can return the position of an element in a list.
- A list slice is an independent list made up of elements taken from a list using the [:] syntax.
- The append() method adds a single element to a list while the extend() method adds all elements from one list to another. The insert() method inserts an element at a given position in a list.
- The del keyword can be used to delete an element within a list, to delete all the elements of a list or to delete the variables that holds a reference to a list.
- The remove() method can be used to search for and remove a particular element from a list, the pop() method can be used to delete an element at a specific index within the list or at the end of a list and the clear() method can be used to delete all the elements of a list.
- Two lists can be concatenated using the + and += operators.

SUMMARY

- A list can be quickly built as duplicates of elements using the * and *= operators.
- The = operator only copies references! In order to create an independent copy of all elements of a list, the copy() method can be used.
- The min() and max() functions return the smallest and largest elements in a list, respectively.
- The sort() function sorts the elements of a list. The sorted() function returns a sorted copy of a list without affecting the original list.
- The reverse() function reverses the order of elements in a list.
- A list can contain any objects within it - including other lists!

5 Tuples

In this chapter you will be able to:

- ☑ Create tuples and access their elements.
- ☑ Iterate through tuples and search for specific elements.
- ☑ Extract sub-tuples from tuples using the concept of tuple slices.
- ☑ Concatenate tuples and populate tuples instantly using tuple multiplication.
- ☑ Learn to differentiate between lists and tuples.

TUPLES

A `tuple` in Python is defined as an immutable ordered sequence of elements.

- *Immutable* means that the contents of a tuple cannot be changed once created.
- *Ordered* means that each item in a tuple has an *index* based on it's position in the tuple.
- *Sequence* means that the elements are arranged in order, based on their indices, and a sequential traversal through the tuple will give us the elements in the order of their indices.

A tuple is very closely related to a list, and hence it might be helpful to compare them. We will do this in section 5.9 after having covered various properties of tuples, but for now, do remember a very important distinguishing property of tuples: they are immutable!

5.1 Creating Tuples

5.1.1 Creating Tuples From Lists Using tuple()

Just as how a list object can be created and assigned to a variable using the `list()` function, a tuple object can be created from a list and assigned to a variable using the `tuple()` function:

```
>>> T=tuple([5,2,3])
>>> T
(5, 2, 3)
```

NOTE:

The `tuple()` function accepts an iterable object – an object over which it can iterate and fetch values one by one. Lists are examples of iterables, and there are many other iterable types.

5.1.2 Creating Empty Tuples Using tuple()

If a list is not passed as an argument, an empty tuple is created:

```
>>> T=tuple()
>>> T
()
```

5.1.3 Creating Tuples Using (,)

Just as how `[]` indicate `list` literals, `()` indicate `tuple` literals provided there is at least one comma within the parentheses:

```
>>> T=(5,2,3)
>>> T
(5, 2, 3)
```

From now on, we will stick to the `()` syntax for convenience and brevity.

The parentheses are used primarily for readability, and it is the presence of the comma that actually drives Python into concluding that we are dealing with a tuple! It is also possible to leave out the parentheses if required:

```
>>> T=5,2,3
>>> T
(5, 2, 3)
```

NOTE:

While lists are *generally* homogeneous, tuples are *generally* heterogeneous. Of course, Python does not prohibit us from having heterogeneous lists and homogeneous tuples!

5.1.4 Creating Empty Tuples Using ()

Just as how `[]` represents an empty list, `()` represents an empty tuple:

```
>>> T=()
>>> T
()
```

5.1.5 Creating Singleton Tuples

We already know that empty tuples use parentheses, and further know that more than the parentheses, it is the comma that is important. Creating a singleton tuple – a tuple that has only 1 element – might look a little weird. We cannot use this syntax: `T=(5)`. The reason why we can't use this syntax is that the interpreter has no way of knowing whether it has to treat `(5)` as an arithmetic expression that is using parentheses or as a tuple. The interpreter assumes that we are dealing with an arithmetic expression. Therefore, to clearly indicate that we are dealing with a tuple, we need to use a comma, with or without the parentheses:

```
>>> T=5,
>>> T
(5,)
>>> T=(5,)
>>> T
(5,)
```

5.2 Accessing Tuple Elements

We can access individual elements of a tuple in exactly the same way as we do for lists:

```
>>> T=(5,2,3)
>>> T[0]
5
>>> T[1]
2
>>> T[2]
3
>>> T[-1]
3
>>> T[-2]
2
>>> T[-3]
5
```

Elements of a tuple cannot be modified:

```
>>> T=(5,2,3)
>>> T
(5, 2, 3)
>>> T[0]=9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

5.3 Counting Tuple Elements

The `len()` function is available to count the number of elements in a tuple. The minimum size of any tuple is 0, corresponding to an empty tuple (denoted by `()`).

```
>>> T=(5,2,3)
>>> T
(5, 2, 3)
>>> len(T)
3
```

```
>>> T=(5,)
>>> len(T)
1
>>> T=()
>>> len(T)
0
```

5.4 Iterating Through Tuple Elements

One of the most practical operations we end up performing on tuples is iterating through them and processing each element in sequence. Since tuples are sequences, the `for` loop is directly compatible for iterating as shown in the example below:

```
>>> T=(5,2,3)
>>> for i in T: print(i)
...
5
2
3
```

5.5 Searching Elements Within Tuples

We can search for elements within tuples the same way we search in lists.

5.5.1 Checking for Existence

The `in` and `not in` operators help us to verify whether a particular element is present in a tuple or not.

```
>>> T=(5,2,3)
>>> 3 in T
True
>>> 4 in T
False
>>> 4 not in T
True
```

5.5.2 Counting Occurrences

The `count()` member function of tuple tells us how many instances of the specified object is present in the tuple.

```
>>> T=(5,2,3,2)
>>> T.count(3)
1
>>> T.count(2)
```

```
2
>>> T.count(4)
0
```

5.5.3 Locating Elements

The `index()` member function of tuple searches for the first occurrence of an element within a tuple and returns the index where it was found, and throws a `ValueError` if not found. It's complete syntax is:

```
tuple.index(x[,i[,j]])
```

Form #1: `tuple.index(x)`

This searches for the first occurrence of the element `x` in the tuple `tuple`, as shown in the example below:

```
>>> T=(5,2,3,2)
>>> T.index(2)
1
>>> T.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
```

Form #2: `tuple.index(x,i)`

If the second argument (`i`) is given, the search starts from index `i` instead of `0`. This allows us to search for other occurrences of an element or to skip some initial elements in the search.

```
>>> T=(5,2,3,2)
>>> T.index(2,2)
3
```

In the above example, the first occurrence of `2` is in index `1`. If we want to search for the next occurrence of `2`, we need to start searching from index `2` onwards, which is what the example does.

Form #3: `tuple.index(x,i,j)`

If the third argument (`j`) is given, the search will start at index `i` and stop at index `j` (excluding index `j`). Remember that if an element is not found when it stops, it will throw `ValueError`.

```
>>> T=(5,2,3,2)
>>> T.index(3,0,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
```

In the above example, we are searching for the element `3` in the tuple `T` starting from index `0` and restricting our search to index `2` (`2` excluded). Thus, our search is restricted to the indices `0` and `1`, and the element `3` is not found in these indices.

5.6 Tuple Slices

Just like a list slice, a tuple slice is a sub-tuble extracted from a tuple – it is a tuple comprising of a part of an existing tuple. A tuple slice uses syntax `tuple[start:end]`, where `tuple` is the parent tuple from which to extract elements, `start` is the starting index from where to start extracting elements and `end` is the ending index (excluding `end`) till where to extract the elements. The parent tuple does not undergo any change. Both `start` and `end` are optional, with `start` defaulting to `0` (index of the first element) and `end` defaulting to `len(tuple)`, i.e. the end of the tuple. Some examples follow:

```
>>> T=(5,2,3,2)
>>> T[1:3] # Slice of T from 1 to 3 (exclusive)
(2, 3)
>>> T[1:] # Slice of T from 1 till the end of the tuple
(2, 3, 2)
>>> T[:3] # Slice of T from the beginning of the tuple till 3
(5, 2, 3)
>>> T[:] # Slice spanning the entire tuple
(5, 2, 3, 2)
```

NOTE:

Since tuples are immutable, unlike lists, `T[:]` is the same as `T`!

5.7 Adding, Multiplying and Copying Tuples

The `+` (addition), `+=` (addition and assignment), `*` (multiplication), `*=` (multiplication and assignment) and `=` (assignment) operators are available to operate on tuples, the same way they are available to operate on lists.

5.7.1 Adding Tuples

Two tuples can be concatenated to form a third tuple using the `+` operator:

```
>>> T1=(1,3)
>>> T2=(7,8)
>>> T3=T1+T2
>>> T3
(1, 3, 7, 8)
```

The += operator will concatenate the tuple on the RHS to the list on the LHS:

```
>>> T=(1,3)
>>> T+=(7,8)
>>> T
(1, 3, 7, 8)
```

NOTE:

`T1 += T2` creates a new tuple and stores `T1+T2` result in that and assigns the new tuple to `T1`. Note that tuples are immutable and new elements cannot be added to an existing tuple.

5.7.2 Multiplying Tuples

A tuple `T` can be multiplied by an integer `n` to denote the tuple `T` repeated `n` times:

```
>>> T=(5,2,3)*3
>>> T
(5, 2, 3, 5, 2, 3, 5, 2, 3)
```

NOTE:

`tuple * n` is the same as `n * tuple`!

Similarly, the `*=` operator repeats the tuple on LHS, RHS number of times, and assigns the resultant tuple to the LHS variable:

```
>>> T=(5,2,3)
>>> T*=3
>>> T
(5, 2, 3, 5, 2, 3, 5, 2, 3)
```

5.7.3 Assigning and Copying Tuples

Before we proceed with assigning and copying tuples, we need to understand a basic point: even though a tuple is immutable, the elements within a tuple need not be immutable!

1. A tuple is immutable and that means that we cannot add elements to a tuple and cannot remove elements from a tuple. We also cannot change the elements of a tuple.
2. An element in a tuple can be a reference to some object. We cannot change this reference, as suggested in the the point above, but can change the contents of the object! Thus, if a tuple contains a list, we can add/remove/change elements in the list for instance.

The = operator allows us to assign a tuple to a tuple variable. Just like in the case of lists, this is not the same as copying elements of a tuple to another and is demonstrated in the example below.

```
>>> T1=(5,[],3)
>>> T2=T1
>>> T2
(5, [], 3)
>>> T1[1].append(9)
>>> T1
(5, [9], 3)
>>> T2
(5, [9], 3)
```

Since tuples are immutable, we had to insert a list as a member in the tuple and change the contents of the list instead. Since tuples are immutable, most methods to copy tuples end up copying only it's reference!

Tuples can be assigned to tuples – a tuple of values can be assigned to a tuple of variables. What makes the syntax even more simpler here is the fact that it is the comma that denotes we are dealing with a tuple, not the parentheses, and hence the parentheses can be omitted. Have a look at these example of multiple assignment:

```
>>> x = y = 5
>>> x
5
>>> y
5
>>> x,y = 2,3
>>> x
2
>>> y
3
```

The statement `x = y = 5` is an example of multiple assignment that does not involve tuples – the value `5` gets stored in both `x` and `y`. But what if we wish to assign different values to `x` and `y`? Can this be done in a single statement? Yes! That's what the statement `x,y = 2,3` achieves! In case the syntax is not clear to you, maybe parentheses can help: `(x,y) = (2,3)`. This is a tuple to tuple assignment wherein each value of the RHS tuple is assigned to the corresponding variable in the LHS tuple. Of course, the size of both the tuples must be the same!

Have a look at the classic way of swapping the values of 2 variables in Python:

```
>>> x,y = 2,3
>>> x
2
>>> y
3
>>> x,y = y,x
>>> x
3
>>> y
2
```

5.8 Miscellaneous Operations on Tuples

Just like lists, there are additional miscellaneous operations possible on tuples – but only those operations that honour the fact that tuples are immutable! Thus, operations like sorting in place or reversing in place are not supported.

5.8.1 Finding the Smallest Element in a Tuple

The `min()` function returns the smallest element in the tuple:

```
>>> T=(5,2,3)
>>> min(T)
2
```

5.8.2 Finding the Largest Element in a Tuple

The `max()` function returns the largest element in the tuple:

```
>>> T=(5,2,3)
>>> max(T)
5
```

5.8.3 Sorting a Tuple

Since in-place sorting of a tuple is not possible, owing to the fact that they are immutable, the `sorted()` function can be used to obtain a copy of the sorted elements of a tuple as shown in the example below:

```
>>> T=(5,2,3)
>>> sorted(T)
[2, 3, 5]
```

The tuple elements can be sorted in descending order as follows:

```
>>> T=(5,2,3)
>>> sorted(T,reverse=True)
[5, 3, 2]
```

5.9 Comparison of Lists and Tuples

Since lists and tuples do have a lot of similarities, as a programmer it becomes necessary to know how they are similar, how they differ, what are their unique capabilities and which one of them to use when. This section addresses these questions.

Both lists and tuples are *ordered sequences* – elements within them are organised based on indices and can be identified by their indices.

Lists and tuples are freely *inter-convertible*:

```
>>> L=[5,2,3]
>>> T=tuple(L)
>>> T
(5, 2, 3)
>>> L=list(T)
>>> L
[5, 2, 3]
```

Both of them support a very similar set of operations as far as *searching* for an element within them are concerned.

The biggest difference between them is the fact that tuples are *immutable* and lists are *dynamically alterable*! This single difference perhaps makes the strongest distinction between lists and tuples and can help you decide what to use when!

If you want a list whose contents are “frozen” (known at the time of creation and do not change after that), switch over to a tuple. If you want a tuple whose contents are manipulated later in the code, switch over to a list.

Also, it might be helpful to keep in mind that lists are generally *homogeneous* whereas tuples are generally *heterogeneous*. If you find yourself dealing with a heterogeneous list, maybe you wanted a tuple there instead. Do note however that Python does not enforce these and has no problems dealing with a heterogeneous list or homogeneous tuple, per se.

For C/C++ Programmers:

C/C++ arrays usually boil down to Python lists, but those arrays that are statically initialised and never changed after that (for example, lookup tables) usually boil down to Python tuples.

C/C++ structures usually boil down to either Python tuples (if you'd like to use indices to access the fields) or Python dictionaries (if you'd like to use field names to access the fields, covered later in section 7).

5.10 Simple Programs Based on Tuples

Applying our understanding of the differences between lists and tuples and re-examining the program `countries3.py`, we observe that both the lists – `countries` and `capitals` – do not change once created. Therefore, we conclude that tuples are a better choice there than lists. Let us rewrite `countries3.py` using tuples:

countries4.py

```
1.   #!/usr/bin/python
2.
3.   # Program to accept country names and print their capitals
4.
5.
6.   countries=("India","Pakistan","Sri Lanka","China")
7.   capitals=("New Delhi","Islamabad","Sri Jayawardenepura
Kotte","Beijing")
8.
9.   while True:
10.      country=input("Enter a country name (or 'end' to
terminate): ")
11.      if country=="end": break
12.      index=countries.index(country)
13.      capital=capitals[index]
14.      print("The capital of {} is
{}".format(country,capital))
```

Observation:

1. The program is identical to `countries3.py`, except that in lines 6 and 7, we have used `()` instead of `[]`. In other words, we are dealing with tuples instead of lists.
2. Tuples have better efficiency than lists. Therefore, if you find a list that could well exist as a tuple instead, switch over to tuple.

Next, we'll write a program to generate the first '`n`' terms of the Fibonacci sequence – a sequence of numbers where each number is the sum of the previous two numbers in the series.

`fib.py`

```
1.   #!/usr/bin/python
2.
3.   # Program to generate the first 'n' Fibonacci terms
4.
5.   t1 = t2 = 1
6.
7.   n = int(input("How many terms? "))
8.
9.   for i in range(n):
10.      print(t1)
11.      t1,t2 = t2,t1+t2
```

Observation:

1. The first 2 terms are `1` and `1` (some people consider the first 2 terms to be `0` and `1`). This is achieved with a multiple assignment in line 5. (In case you wish to start the series from `0`, then line 5 can be replaced by a tuple assignment: `t1,t2=0,1`.
2. Within the loop that runs `n` times (line 9), we print one term per iteration (line 10). The series moves on because of the tuple assignment in line 11, where `t1=t2` and `t2=t1+t2`.

5.11 Questions

1. How are tuples different from lists? When do we prefer a list over a tuple and vice-versa?
2. Write a short note on singleton tuples.
3. Write a short note on searching within tuples.
4. Write a short note on tuple slices.

5. Write a short note on the operators that can be used with tuples.

5.12 Exercises

1. Write a program to convert a tuple of values into a tuple of singleton tuples of values. Thus, `(1,2,3)` should get converted to `((1,)(2,),(3,))`.
2. Write a program to count the number of elements in a list until it finds a tuple in the list.
3. Write a program to add corresponding elements of two given tuples, giving rise to a new tuple.
4. Write a program to multiply each element of a given tuple with a given integer, giving rise to a new tuple.
5. Write a program that classifies a triangle as being equilateral, isosceles or scalene, given it's 3 vertices as a tuple of tuples.

SUMMARY

- A tuple is an immutable sequence of elements - elements cannot be added to or deleted from a tuple.
- A tuple can be simply created using (). Elements of a tuple can be accessed using [].
- Parentheses is not required for a tuple to be identified! Just the comma between elements is enough!
- The len() function gives the number of elements in a tuple and the count() method returns the number of occurrences of a particular element in the tuple.
- The for loop can be naturally used to iterate through the elements of a tuple.
- The in and not in operators check for the existence of an element in a tuple. The index() method can return the position of an element in a tuple.
- A tuple slice is an independent tuple made up of elements taken from a tuple using the [:] syntax.

6 Sets

In this chapter you will be able to:

- ☑ Create a set and access its elements.
- ☑ Iterate through sets and search for specific elements.
- ☑ Add and Delete elements in a set.
- ☑ Perform set operations like union, intersection, difference and symmetric difference.
- ☑ Check for disjoint sets, subsets and supersets.

Sets

A set is an unordered collection of unique elements.

- *Unordered* means that elements in a set do not have an index by which they can be addressed.
- *Unique* means that duplicates are not tolerated (attempt to add an element into a set when the element already exists will be ignored).

NOTE:

A set can contain only *hashable* elements – those elements that have a hash code. Though hashing is outside the scope of this book, do note that all immutable collections (like tuples) are hashable in Python whereas mutable collections (like lists) are not. Thus, for example, a set can contain a tuple, but can't contain a list!

6.1 Creating Sets

Just as how the `list()` function is used to create a list, the `set()` function is used to create a set:

```
>>> S=set([1,2,3,4,5])
>>> S
{1, 2, 3, 4, 5}
>>> S=set(range(1,6))
>>> S
{1, 2, 3, 4, 5}
```

NOTE:

The `set` constructor receives a sequence (like `list` and `tuple`) and extracts their elements to create a set. Do not mistake the above example to mean construction of a set that contains a list! Remember from the previous note that sets cannot contain lists as they are mutable!

Just as how `[]` is used to implicitly define lists, `{}` is used to implicitly define sets:

```
>>> S={2,4,6,8}
>>> S
{8, 2, 4, 6}
```

As can be seen from the example above, the order of the elements within a set is not under our control as sets are unordered collections. Sets support the mathematical set operations of finding the union, intersection and difference. We will explore these operations soon.

6.2 Accessing Set Elements

Being unordered collections, it is not possible to identify an element within a set using an index or any other mechanism. Therefore, the operations normally performed on a set is checking for the existence of an element within a set rather than accessing a particular element in the set. Of course, iterating through a set is still possible and meaningful.

6.3 Counting Set Elements

The `len()` function is available to count the number of elements in a set. The minimum size of any set is 0, corresponding to an empty set (denoted by `set()` and not `{}`).

```
>>> S=set(range(1,6))
>>> S
{1, 2, 3, 4, 5}
>>> len(S)
5
>>> S=set()
>>> S
set()
>>> len(S)
0
```

6.4 Iterating Through Set Elements

Since accessing particular set elements is not possible, iterating through a set is the only option to determine the contents of a set. The normal `for` loop can be used for this purpose:

```
>>> S=set(range(1,6))
>>> for i in S: print(i)
...
1
2
3
4
5
```

6.5 Searching Elements within Sets

Checking for the existence of an element within a set is a very efficient operation (compared to a similar search within a list or a tuple). The `in` and `not in` operators can be used for this purpose:

```
>>> S=set(range(1,6))
>>> 2 in S
True
>>> 9 in S
False
>>> 9 not in S
True
```

6.6 Adding and Deleting Elements

It is possible to add elements to a set and delete elements from a set. While adding, keep in mind that addition of duplicates are ignored. While deleting, we need to consider the situation what happens if we attempt to delete something that doesn't exist. We will explore various methods to add and delete elements.

6.6.1 Adding Elements

6.6.1.1 Adding Individual Elements to a Set

To add an element to a set, the `set.add(x)` function can be used which adds the element `x` to the set `set`:

```
>>> S=set(range(1,6))
>>> S
{1, 2, 3, 4, 5}
>>> S.add(9)
>>> S
{1, 2, 3, 4, 5, 9}
```

6.6.1.2 Adding a Set of Elements to a Set

To add a set of elements to another, the `set.update(s)` function can be used, which updates the set `set` to contain the union of the set `set` and the set `s` – in simple terms, it adds all the elements of the set `s` to the set `set`:

```
>>> S1={2,4,6,8}
>>> S1
{8, 2, 4, 6}
>>> S2={1,3,5,7}
>>> S2
{1, 3, 5, 7}
>>> S1.update(S2)
>>> S1
{1, 2, 3, 4, 5, 6, 7, 8}
```

It is also possible to find the union of a set with another and then assign that set to the original variable. We will examine how to find the union soon.

6.6.2 Deleting Elements

There are many ways of deleting elements from a set. These not only differ in how they allow the user to choose the element to be deleted, but also differ in what happens if they are unable to delete.

6.6.2.1 Using pop() to Delete an Element

The `set.pop()` function deletes an element from the set `set` and returns the element deleted. Since sets are unordered, we cannot predict which element will actually get deleted in the set. As a special case, if the set is empty, this function generates a `KeyError`.

```
>>> S=set(range(1,6))
>>> S
{1, 2, 3, 4, 5}
>>> S.pop()
1
>>> S
{2, 3, 4, 5}
>>> S.pop()
2
>>> S
{3, 4, 5}
```

6.6.2.2 Using remove() to Delete an Element

The `set.remove(x)` function removes the element `x` from the set `set`. If `x` was not found, it will produce a `KeyError`.

```
>>> S=set(range(1,6))
>>> S
{1, 2, 3, 4, 5}
>>> S.remove(3)
>>> S
{1, 2, 4, 5}
>>> S.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
```

6.6.2.3 Using discard() to Delete an Element

The `set.discard(x)` function deletes the element `x` from the set `set`. Where it differs from the `remove()` function is that if the element `x` was not found, it does nothing instead of generating an error.

```
>>> S=set(range(1,6))
>>> S
{1, 2, 3, 4, 5}
>>> S.discard(3)
>>> S
{1, 2, 4, 5}
>>> S.discard(3)
>>> S
{1, 2, 4, 5}
```

6.6.2.4 Using clear() to Delete All Elements

The `set.clear()` function deletes all elements in the set `set`.

```
>>> S=set(range(1,6))
>>> S
{1, 2, 3, 4, 5}
>>> S.clear()
>>> S
set()
```

6.6.3 Assigning and Copying Sets

The = operator can assign a set to a variable. However, this only ends up copying a reference to a set as demonstrated below:

```
>>> S1={1,2}
>>> S2=S1
>>> S2
{1, 2}
>>> S1.remove(1)
>>> S1
{2}
>>> S2
{2}
```

In order to copy the elements of a set to another, the `set.copy()` function can be used:

```
>>> S1={1,2}
>>> S2=S1.copy()
>>> S2
{1, 2}
>>> S1.remove(1)
>>> S1
{2}
>>> S2
{1, 2}
```

6.7 Set Operations

Mathematically, a variety of operations can be performed on sets:

- The *union* of set A and set B is a set comprising of all elements found in either set A or set B. The union of {1,2} and {2,3} is {1,2,3}.
- The *intersection* of set A and set B is a set comprising of all elements found in both set A and set B. The intersection of {1,2} and {2,3} is {2}.
- The *difference* of set A and set B (set A – set B) is a set comprising of all elements found in set A with all elements of set B removed from it. The difference of {1,2} and {2,3} is {1}.
- The *symmetric difference* of set A and set B is a set comprising of all elements found in either set A or set B, but not in both. The symmetric difference of {1,2} and {2,3} is {1,3}.

In addition:

- Two sets are said to be *disjoint* if there are no elements in common within them. The sets {1,2} and {2,3} are not disjoint because of the common element 2.
- A set A is said to be a *subset* of set B if all elements of set A are present in set B. Similarly, in such a case, set B is said to be a superset of set A. Neither is {1,2} a subset of {2,3}, nor the other way around.

6.7.1 Set Union

The `set.union(s)` function returns a new set that represents the union of the set `set` and the set `s`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.union(S2)
{1, 2, 3}
>>> S1
{1,2}
```

The `|` operator plays the same role as `set.union(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 | S2
{1, 2, 3}
>>> S1
{1, 2}
```

We can directly find the union of set literals as follows:

```
>>> {1,2}.union({2,3})
{1, 2, 3}
>>> {1,2} | {2,3}
{1, 2, 3}
```

It is also possible to find the union of multiple sets by passing multiple arguments to `union()` or using the `|` operator multiple times as follows:

```
>>> {1}.union({2},{3},{4})
{1, 2, 3, 4}
>>> {1} | {2} | {3} | {4}
{1, 2, 3, 4}
```

The `set.update(s)` function also finds the union, but stores it back into the original set `set` (this was introduced in section 6.6.1.2):

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.update(S2)
>>> S1
{1, 2, 3}
```

The `|=` operator plays the same role as `set.update(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 |= S2
>>> S1
{1, 2, 3}
```

It is also possible to find the union of multiple sets and store them in a set by passing multiple arguments to `update()` or using the `|` operator multiple times after using the `|=` operator as follows:

```
>>> S={1}
>>> S.update({2},{3},{4})
>>> S
{1, 2, 3, 4}
>>> S={1}
>>> S |= {2} | {3} | {4}
>>> S
{1, 2, 3, 4}
```

6.7.2 Set Intersection

The `set.intersection(s)` function returns a new set that represents the intersection of the set `set` and the set `s`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.intersection(S2)
{2}
>>> S1
{1, 2}
```

The `&` operator plays the same role as `set.intersection(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 & S2
{2}
>>> S1
{1, 2}
```

We can directly find the intersection of set literals as follows:

```
>>> {1,2}.intersection({2,3})
{2}
>>> {1,2} & {2,3}
{2}
```

It is also possible to find the intersection of multiple sets by passing multiple arguments to `intersection()` or using the `&` operator multiple times as follows:

```
>>>> {1,2,3}.intersection({2,3,4},{1,3,5})
{3}
>>> {1,2,3} & {2,3,4} & {1,3,5}
{3}
```

The `set.intersection_update(s)` function also finds the intersection, but stores it back into the original set `set`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.intersection_update(S2)
>>> S1
{2}
```

The `&=` operator plays the same role as `set.intersection_update(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 &= S2
>>> S1
{2}
```

It is also possible to find the intersection of multiple sets and store them in a set by passing multiple arguments to `intersection_update()` or using the `&` operator multiple times after using the `&=` operator as follows:

```
>>> S={1,2,3}
>>> S.intersection_update({2,3,4},{1,3,5})
>>> S
{3}
>>> S={1,2,3}
>>> S &= {2,3,4} & {1,3,5}
>>> S
{3}
```

6.7.3 Set Difference

The `set.difference(s)` function returns a new set that represents the difference of the set `set` and the set `s`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.difference(S2)
{1}
>>> S1
{1, 2}
```

The `-` operator plays the same role as `set.difference(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 - S2
{1}
>>> S1
{1, 2}
```

We can directly find the difference of set literals as follows:

```
>>> {1,2}.difference({2,3})
{1}
>>> {1,2} - {2,3}
{1}
```

It is also possible to find the difference of multiple sets by passing multiple arguments to `difference()` or using the `-` operator multiple times as follows:

```
>>> {1,2,3}.difference({2,4,6},{3,6,9})
{1}
>>> {1,2,3} - {2,4,6} - {3,6,9}
{1}
```

The `set.difference_update(s)` function also finds the difference, but stores it back into the original set `set`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.difference_update(S2)
>>> S1
{1}
```

The `-=` operator plays the same role as `set.difference_update(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 -= S2
>>> S1
{1}
```

It is also possible to find the difference of a set and the union of multiple sets and store them in a set by passing multiple arguments to `difference_update()` or using the `|` operator multiple times after using the `-=` operator as follows:

```
>>> S={1,2,3}
>>> S.difference_update({2,4,6},{3,6,9})
>>> S
{1}
>>> S={1,2,3}
>>> S -= {2,4,6} | {3,6,9}
>>> S
{1}
```

6.7.4 Set Symmetric Difference

The `set.symmetric_difference(s)` function returns a new set that represents the symmetric difference of the set `set` and the set `s`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.symmetric_difference(S2)
{1, 3}
>>> S1
{1, 2}
```

The `^` operator plays the same role as `set.symmetric_difference(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 ^ S2
{1, 3}
>>> S1
{1, 2}
```

We can directly find the symmetric difference of set literals as follows:

```
>>> {1,2}.symmetric_difference({2,3})
{1, 3}
>>> {1,2} ^ {2,3}
{1, 3}
```

The `set.symmetric_difference_update(s)` function also finds the symmetric difference, but stores it back into the original set `set`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.symmetric_difference_update(S2)
>>> S1
{1, 3}
```

The `^=` operator plays the same role as `set.symmetric_difference_update(s)`:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1 ^= S2
>>> S1
{1, 3}
```

6.7.5 Summary of Fundamental Set Operations

The below table summarises the basic set operations that were covered in the preceding sections:

Operation	*Operator*	*Function*
Union	`|`	`union()`
Union (with assignment)	`|=`	`update()`
Intersection	`&`	`intersection()`
Intersection (with assignment)	`&=`	`intersection_update()`
Difference	`-`	`difference()`
Difference (with assignment)	`-=`	`difference_update()`
Symmetric Difference	`^`	`symmetric_difference()`
Symmetric Difference (with assignment)	`^=`	`symmetric_difference_update()`

Table 14: Summary of Fundamental Set Operations

6.7.6 Disjoint Sets

The `set.isdisjoint(s)` function tells whether the set `set` and the set `s` are disjoint or not:

```
>>> S1={1,2}
>>> S2={2,3}
>>> S1.isdisjoint(S2)
False
>>> S1.isdisjoint({3,4})
True
```

6.7.7 Subsets and Supersets

The `set.issubset(s)` function tells whether the set `set` is a subset of the set `s` or not. The `<=` operator plays the same role.

```
>>> {1,2}.issubset({2,3})
False
>>> {1,2}.issubset({1,2,3})
True
>>> {1,2} <= {2,3}
False
>>> {1,2} <= {1,2,3}
True
```

A set A is a proper subset of a set B if A is a subset of B and A is not equal to B. While there is no function to do this check in Python, we can use the `<` operator for this:

```
>>> {1,2} <= {1,2}
True
>>> {1,2} < {1,2}
False
```

The `set.issuperset(s)` function tells whether the set `set` is a superset of the set `s` or not. The `>=` operator plays the same role.

```
>>> {1,2}.issuperset({2,3})
False
>>> {1,2,3}.issuperset({2,3})
True
>>> {1,2} >= {2,3}
False
>>> {1,2,3} >= {2,3}
True
```

A set A is a proper superset of a set B if A is a superset of B and A is not equal to B. While there is no function to do this check in Python, we can use the `>` operator for this:

```
>>> {1,2} >= {1,2}
True
>>> {1,2} > {1,2}
False
```

6.8 Programs Based on Sets

Practically, a set should be used when a collection is needed and at least one of the following are required:

- Duplicates don't matter – we only need to store unique elements
- We want to compare collections to find out what is common, unique, etc.

Let's rig up a demo program to keep track of user-defined items of two people and then compare them:

set_demo.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of sets
4.
5.   item_count_A = int(input("How many items does A have? "))
6.   items_A = set()
7.   for i in range(item_count_A):
8.       item = input("Enter item #{} of A: ".format(i+1))
9.       items_A.add(item)
10.
11.  item_count_B = int(input("How many items does B have? "))
12.  items_B = set()
13.  for i in range(item_count_B):
14.      item = input("Enter item #{} of B: ".format(i+1))
15.      items_B.add(item)
16.
17.  print("Number of items A has:",len(items_A))
18.  print("Number of items B has:",len(items_B))
19.
20.  print("All items with A and B:")
21.  for item in items_A | items_B: print(item)
22.
23.  print("Items common to A and B:")
24.  for item in items_A & items_B: print(item)
25.
26.  print("Items unique to A:")
27.  for item in items_A - items_B: print(item)
28.
29.  print("Items unique to B:")
30.  for item in items_B - items_A: print(item)
31.
32.  print("Items with only A and only B:")
33.  for item in items_A ^ items_B: print(item)
```

Output:

```
How many items does A have? 3
Enter item #1 of A: Bat
Enter item #2 of A: Ball
Enter item #3 of A: Shoes
How many items does B have? 4
Enter item #1 of B: Cap
Enter item #2 of B: Ball
Enter item #3 of B: Socks
Enter item #4 of B: Shoes
Number of items A has: 3
Number of items B has: 4
All items with A and B:
Socks
Shoes
Cap
Bat
Ball
Items common to A and B:
Ball
Shoes
Items unique to A:
Bat
Items unique to B:
Cap
Socks
Items with only A and only B:
Cap
Socks
Bat
```

6.9 Questions

1. What is a Set?
2. Define the following with respect to given 2 sets with examples:
    1. Set Union
    2. Set Intersection
    3. Set Difference
    4. Set Symmetric Difference
    5. Disjoint Sets
    6. Subset

    7. Superset
3. In which all ways can we create a set? Illustrate with simple examples.
4. Is it possible to access a specific element in a set using its index? Explain.
5. What do you observe when you declare a set with some elements and try displaying its contents? What do you infer from this?
6. Can a set contain a list? What kind of collection is a set capable of storing?
7. How do we determine the size of a set?
8. How can we add multiple elements from a list to a set? Demonstrate with an example.
9. Illustrate with examples how to remove elements from a set using the below:
    1. `pop()`
    2. `remove()`
    3. `discard()`
10. What do you observe when you perform the below operations on an empty set?
    1. `pop()`
    2. `remove()`
    3. `discard()`
11. How can we empty a set?
12. List all possible ways of finding:
    1. Union of 2 sets with examples.
    2. Intersection of 2 sets with examples.
    3. Difference of 2 sets with examples
13. Illustrate with a simple example how to evaluate the symmetric difference of 2 sets using '`symmetric_difference`' and '`^`' operator?
14. In what way does the '`symmetric_difference_update`' function differ from '`symmetric_difference`'? Illustrate with an example.
15. In what way does the '`difference`' function differ from '`symmetric_difference`'?
16. Given 2 sets A & B, how do we confirm if the below are true:
    1. A and B are disjoint sets

2. A is a subset of B
3. A is a superset of B

6.10 Exercises

1. Add a considerable huge no. of unique numbers (e.g. `10000`) in a set and use the '`in`' operator to search for an element and note the time it takes to fetch. Repeat the above task and use a `for` loop to iterate through the set in search of the element and then compare the time taken in both scenarios.
2. Write a program to convert a list of integers into a list of unique integers using sets.
3. Assume that we have 2 lists, each having only IP addresses as its contents. Write a program to determine if the 2 lists have the same IP addresses (albeit in different positions) in the most optimal manner.

SUMMARY

- A set is an unordered collection of unique elements - elements cannot be added to or deleted from a tuple.
- A tuple can be simply created using {}. Elements of a set cannot be directly accessed as they are unordered collections.
- The len() function gives the number of elements in a set.
- The for loop can be naturally used to iterate through the elements of a set.
- The in and not in operators check for the existence of an element in a set.
- The add() method allows an element to be added to a set and the update() method allows a set of elements to be added to a set.
- The pop() method deletes an element from a set and the remove() and discard() methods remove a specified element from a set. The clear() method removes all elements from a set.
- The union(), intersection(), difference() and symmetric_difference() perform basic set operations.
- The isdisjoint(), issubset() and issuperset() methods are useful Boolean methods.

7 Dictionaries

In this chapter you will be able to:

- ☑ Create a dictionary and access its elements.
- ☑ Iterate through dictionary pairs, keys and values.
- ☑ Check for the existence of a particular key in a dictionary and fetch the corresponding value.
- ☑ Add and Delete elements in a dictionary.

DICTIONARIES

A *dictionary* is a collection of key-value pairs subject to the constraint that all the keys should be unique.

The *keys* of a dictionary can be considered to be members of a set (and are thus unique), with each key keeping track of a value.

7.1 Creating Dictionaries

Just as how a list object can be created and assigned to a variable using the `list()` function, a dictionary object can be created and assigned to a variable using the `dict()` function:

```
>>> D=dict([('apple','red'),('grapes','green')])
>>> D
{'grapes': 'green', 'apple': 'red'}
```

Observation:

1. A dictionary can be created in a variety of ways – the above technique is one way by which an *interable* (where each element of the iterable has 2 elements – a key and a value) can be converted into a dictionary.
2. In the above example, the iterable is a *list* and each element of the list is a *tuple* with 2 elements – a key and a value.
3. Since the keys of a dictionary can be considered to be stored in a set and sets are unordered collections, even the keys of a dictionary are unordered. As can be observed from the output above, the order of the keys is not directly under our control.

Just as how `[]` is used to indicate list literals, `{}` are used to denote dictionary literals. Recall from section 6.1 that `{}` are also used to enclose members of a set. How does Python differentiate between sets and dictionaries then? Using the fact that a set comprises of multiple elements within `{}` separated by commas whereas a dictionary comprises of multiple elements within `{}` separated by colons (to separate the key from the value) and commas (to separate the pairs).

```
>>> D={'apple':'red','grapes':'green'}
>>> D
{'grapes': 'green', 'apple': 'red'}
```

From now on, we will stick to the `{}` syntax for convenience and brevity.

NOTE:

Since braces (`{}`) with commas (`,`) denote sets and braces with colons (`:`) denote dictionaries, the question might arise what does empty braces mean? The answer is that empty braces always indicate empty dictionaries and therefore the empty sets have to be created only by using `set()`.

7.2 Accessing Dictionary Elements

A dictionary can be viewed as a special kind of array whose subscripts are strings instead of integers. Such arrays are called *associative arrays*. The same `[]` operator that is used to access an element of a list given it's index can be used to access a value in a dictionary given it's key:

```
>>> D={'apple':'red','grapes':'green'}
>>> D['apple']
'red'
>>> D['grapes']
'green'
```

An attempt to access a value using a key that does not exist in the dictionary results in a `KeyError`:

```
>>> D={'apple':'red','grapes':'green'}
>>> D['mango']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'mango'
```

New values can be assigned to existing keys using an assignment as shown below. Also, if at the time of assignment the referenced key does not exist, it will be created:

```
>>> D={'apple':'red','grapes':'green'}
>>> D['grapes']='purple'
>>> D
{'grapes': 'purple', 'apple': 'red'}
>>> D['mango']='yellow'
>>> D
{'mango': 'yellow', 'grapes': 'purple', 'apple': 'red'}
```

7.3 Counting Dictionary Elements

The `len()` function can be used to count the number of key-value pairs in a dictionary:

```
>>> D={'apple':'red','grapes':'green'}
>>> len(D)
2
>>> D={'apple':'red'}
>>> len(D)
1
>>> D={}
>>> len(D)
0
```

7.4 Iterating through Dictionary Elements

Since each element in a dictionary is a key-value pair, iterating through a dictionary can be of multiple forms: iterating through the keys in a dictionary or iterating through the values or iterating through the elements (key-value pairs). We will see examples of each.

7.4.1 Iterating Through the Keys of a Dictionary

The normal `for` loop can be used to iterate through the keys of a dictionary:

```
>>> D={'apple':'red','grapes':'green'}
>>> for K in D: print(K)
...
grapes
apple
```

The keys of a dictionary can be explicitly retrieved using the `dict.keys()` function, which returns a new *view* of the keys within the dictionary `dict` – which means that if and when the keys of the dictionary `dict` changes, the view will also change automatically:

```
>>> D={'apple':'red','grapes':'green'}
>>> K=D.keys()
>>> for i in K: print(i)
...
grapes
apple
```

7.4.2 Iterating Through the Values of a Dictionary

Just as how `dict.keys()` returns a view of all the keys in the dictionary `dict`, `dict.values()` returns a view of all the values in the dictionary `dict`:

```
>>> D={'apple':'red','grapes':'green'}
>>> V=D.values()
>>> for i in V: print(i)
...
green
red
```

NOTE:

The order in which the values are returned is the same as the order in which their keys are returned.

Of course, it is also possible to iterate through the keys and print out the corresponding values:

```
>>> D={'apple':'red','grapes':'green'}
>>> for K in D: print(D[K])
...
green
red
```

Observation:

1. In the above example, we are iterating through the keys of the dictionary `D`.
2. We are printing `D[K]`, which is the value associated with the key `K` in the dictionary `D`.

7.4.3 Iterating Through the Key-Value Pairs of a Dictionary

One way to iterate through the dictionary is to iterate through the keys of the dictionary and fetch the corresponding values using the keys. Another method, which is more direct, is to iterate through each key-value pair of the dictionary using the `dict.items()` function that returns the key-value pairs (each pair as a tuple), as shown in the example below:

```
>>> D={'apple':'red','grapes':'green'}
>>> for K,V in D.items():
...     print(K,V)
...
apple red
grapes green
```

7.5 Searching Elements within Dictionaries

7.5.1 Checking for the Existence of a Key in a Dictionary

One of the most efficient operations on dictionaries is checking whether a key exists, and extracting the corresponding value if the key exists.

The `in` and `not in` operators can be used to check if a key exists in a dictionary:

```
>>> D={'apple':'red','grapes':'green'}
>>> 'apple' in D
True
>>> 'mango' in D
False
>>> 'mango' not in D
True
```

7.5.2 Extracting the Value of a Key Using []

Given a key that exists in a dictionary, extracting the corresponding value is a trivial operation, and has already been discussed in section 7.2, but is nevertheless presented below:

```
>>> D={'apple':'red','grapes':'green'}
>>> K='apple'
>>> V=D[K]
>>> print(V)
red
```

7.5.3 Extracting the Value of a Key Using dict.get()

Another method of extracting the value associated with a key is by using the `dict.get(key)` function, which returns the value corresponding to the key `key` in the dictionary `dict`, if the key exists, and returns `None` if the key was not found. Unlike in the previous method of using `[]`, this function does not generate a `KeyError` if the key was not found.

```
>>> D={'apple':'red','grapes':'green'}
>>> D.get('apple')
'red'
>>> D.get('mango')
>>>
```

Observation:

1. `D.get('mango')` is supposed to return the value of the key `mango` in the dictionary `D`, but since it does not exist it returns `None` instead. Therefore, there is no output at all.

The complete version is `dict.get(key,default)` function, which returns the value corresponding to the key `key` in the dictionary `dict` if the key exists, and returns the value `default` if the key was not found.

```
>>> D={'apple':'red','grapes':'green'}
>>> D.get('apple','blue')
'red'
>>> D.get('mango','blue')
'blue'
>>> D.get('mango')
>>>
```

Observation:

1. `D.get('apple','blue')` returns the value corresponding to the key `apple` in the dictionary `D`. The corresponding value is `red` and is hence returned.
2. `D.get('mango','blue')` tries to return the value of the key `mango` in the dictionary `D`, but since that key does not exist it returns the value `blue` instead.
3. `D.get('mango')` is supposed to return the value of the key `mango` in the dictionary `D`, but since it does not exist it returns `None` instead. Therefore, there is no output at all.

7.5.4 Extracting a Key Given it's Associated Value

While it is a trivial operation to extract the value associated with a key, extracting a key from it's value is slightly more complicated due to the following reasons:

1. There is no direct mechanism to extract a key from it's value
2. The values need not be unique. When there are multiple identical values, the question of which key to fetch comes into the picture. The only possibilities are to fetch any one of the candidate keys or fetch all the candidate keys.

Given a value, the following example shows how to fetch any one key with the given value:

```
>>>
D={'apple':'red','grapes':'green','banana':'yellow','mango':'green'}
>>> V='green'
>>> for K in D:
...     if D[K]==V:
...             print(K)
...             break
...
mango
```

Observation:

1. The `break` statement is just an optimization mechanism to ensure that once we find any suitable key, we abort the search process
2. If the `break` statement is removed and no other changes are made to the code, the code ends up listing all the keys that have the given value.

7.5.5 Extracting All Keys With a Particular Value

Based on the previous piece of code, here is a code snippet to print all keys that have a given value:

```
>>>
D={'apple':'red','grapes':'green','banana':'yellow','mango':'green'}
>>> V='green'
>>> for K in D:
...     if D[K]==V: print(K)
...
mango
grapes
```

7.6 Adding and Deleting Elements

7.6.1 Adding Elements

7.6.1.1 Adding Elements Using []

As discussed in section 7.2, if an assignment is made to an element of a dictionary using a key that does not exist, the key is created with the given associated value:

```
>>> D={'apple':'red','grapes':'green'}
>>> 'mango' in D
False
>>> D['mango']='yellow'
>>> 'mango' in D
True
```

7.6.1.2 Adding Elements Using setdefault()

The function `dict.setdefault(key)` returns the value associated with the key `key` in the dictionary `dict` if the key exists, and creates it with a value of `None` if the key does not exist (returning the value `None` as well):

```
>>> D={'apple':'red','grapes':'green'}
>>> D.setdefault('apple')
'red'
>>> D.setdefault('pear')
>>> D
{'mango': 'blue', 'apple': 'red', 'pear': None, 'grapes': 'green'}
```

In the more complete form, the `dict.setdefault(key,default)` function returns the value associated with the key `key` in the dictionary `dict` if the key exists, and if the key does not exist, creates it with a corresponding value of `default` and returns the same value.

```
>>> D={'apple':'red','grapes':'green'}
>>> D.setdefault('apple','blue')
'red'
>>> D
{'apple': 'red', 'grapes': 'green'}
>>> D.setdefault('mango','blue')
'blue'
>>> D
{'mango': 'blue', 'apple': 'red', 'grapes': 'green'}
```

7.6.2 Deleting Elements

An element (key-value pair) can be deleted from a dictionary using a variety of ways. Note that deleting an element is different from deleting the value associated with a key in the dictionary (which is nothing but setting the value as `None` for a given key in a dictionary).

7.6.2.1 Using del to Delete an Element

The `del` statement can be used to delete an element from a dictionary, identifying the element using it's key. The specified key needs to exist, otherwise a `KeyError` will be generated:

```
>>> D={'apple':'red','grapes':'green'}
>>> del D['apple']
>>> D
{'grapes': 'green'}
>>> del D['mango']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'mango'
```

7.6.2.2 Using popitem() to Delete Elements

The `dict.popitem()` function removes an element (key-value pair) from the dictionary `dict` and returns the pair in the form of a tuple. Each time this function is invoked, the size of the dictionary `dict` reduces by one. It generates `KeyError` however if invoked on an empty dictionary:

```
>>> D={'apple':'red','grapes':'green'}
>>> D.popitem()
('apple', 'red')
>>> D
{'grapes': 'green'}
>>> D.popitem()
('grapes', 'green')
>>> D
{}
>>> D.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

7.6.2.3 Using pop() to Delete Elements

The `dict.pop(key)` function deletes the element (key-value pair) with the key `key` from the dictionary `dict` and returns the corresponding value of the deleted key. If the key does not exist, it generates a `KeyError`:

```
>>> D={'apple':'red','grapes':'green'}
>>> D.pop('apple')
'red'
>>> D
{'grapes': 'green'}
>>> D.pop('grapes')
'green'
>>> D
{}
>>> D.pop('mango')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'mango'
```

Another version of the function is `dict.pop(key,default)`, which deletes the element with key `key` from the dictionary `dict` and returns it's value if the key was found. If the key was not found, it returns the value `default` instead without deleting anything:

```
>>> D={'apple':'red','grapes':'green'}
>>> D.pop('apple','blue')
'red'
>>> D
{'grapes': 'green'}
>>> D.pop('apple','blue')
'blue'
>>> D
{'grapes': 'green'}
```

7.6.2.4 Using clear() To Delete All Elements of a Dictionary

The `dict.clear()` function clears out the dictionary `dict` – it removes all elements from the dictionary `dict`:

```
>>> D={'apple':'red','grapes':'green'}
>>> D.clear()
>>> D
{}
```

7.7 Simple Programs Based on Dictionaries

If we re-examine the program `countries4.py`, we see that the capitals can actually be stored as a single dictionary with the country name as the key and it's capital as the corresponding value, instead of 2 different arrays/lists/tuples as in the program there. Let us rewrite the program using dictionaries:

countries5.py

```
1.   #!/usr/bin/python
2.
3.   # Program to accept country names and print their capitals
4.
5.
6.   capitals={"India":"New Delhi","Pakistan":"Islamabad","Sri
Lanka":"Sri Jayawardenepura Kotte","China":"Beijing"}
7.
8.   while True:
9.       country=input("Enter a country name (or 'end' to
terminate): ")
10.      if country=="end": break
11.      print("The capital of {} is {}" .format(country,
capitals[country]))
```

Observation:

1. This is the most efficient implementation so far of this program – fetching a value from a dictionary using it's key is a very fast operation.
2. The program will still report a `KeyError` if an unknown country is entered. This is because a `KeyError` is generated if we attempt to access the value of a key that does not exist in a dictionary. We can either use the `dict.get()` and check if the returned value was `None` or can use exception handling (discussed in section 13).

Let us improve the previous program by ensuring that the program does not crash even if an unknown country name is given by the user. The program should merely say that the country is not found in it's database!

countries6.py

```
1.   #!/usr/bin/python
2.
3.   # Program to accept country names and print their capitals
4.
5.
6.   capitals={"India":"New Delhi","Pakistan":"Islamabad","Sri
Lanka":"Sri Jayawardenepura Kotte","China":"Beijing"}
7.
8.   while True:
9.       country=input("Enter a country name (or 'end' to
terminate): ")
10.      if country=="end": break
11.      capital=capitals.get(country)
12.      if capital is None: print("Sorry, country not found!")
13.      else: print("The capital of {} is
{}".format(country,capital))
```

Output:

```
Enter a country name (or 'end' to terminate): India
The capital of India is New Delhi
Enter a country name (or 'end' to terminate): China
The capital of China is Beijing
Enter a country name (or 'end' to terminate): Russia
Sorry, country not found!
Enter a country name (or 'end' to terminate): end
```

Let's apply our understanding of dictionaries to write a small program to convert a Roman numeral to decimal (and to words reusing the code from the program `num2words.py`). In order to keep the program simple, the following constraints will be introduced:

1. The Roman numeral will be restricted to a single letter only (more letters require iterating through characters of a string, covered later in section 8)
2. Even though we don't require the complete functionality of `num2words.py`, we will try to retain the code as much as possible for convenience.
3. No error checking will be done – we will assume that the user gives us correct input.

roman2dec.py

```
1.   #!/usr/bin/python
2.
3.   # Program to convert a single Roman numeral to decimal and
words
4.
5.   roman={'I':1,'V':5,'X':10,'L':50,'C':100,'D':500,'M':1000}
6.
7.   n=input("Enter a single Roman uppercase numeral:
").upper()
8.   n=roman[n]
9.   print("Decimal:",n)
10.
11.  onesWords=('zero','one','two','three','four',\
12.       'five','six','seven','eight','nine')
13.  tensWords=('','ten','twenty','thirty','forty',\
14.       'fifty','sixty','seventy','eighty','ninety')
15.  teensWords=('','eleven','twelve','thirteen','fourteen',\
16.       'fifteen','sixteen','seventeen','eighteen','nineteen')
17.
18.  thousands=n//1000
19.  hundreds=n//100%10
20.  tens=n//10%10
21.  ones=n%10
22.
23.  if thousands>0: print("{} thousand
".format(onesWords[thousands]),end='')
24.  if hundreds>0: print("{} hundred
".format(onesWords[hundreds]),end='')
25.  if (thousands>0 or hundreds>0) and (tens>0 or ones>0):
print("and ",end='')
26.  if tens==1 and ones>0: print(teensWords[ones],end='')
27.  else:
28.       if tens>0: print(tensWords[tens],'',end='')
29.       if ones>0: print(onesWords[ones],end='')
30.  print()
```

Observation:

1. The code is mostly taken from `num2words.py`, but lines 5-9 are different.
2. Line 5 introduces a dictionary to keep track of the decimal equivalent of Roman numerals. The Roman numerals have been given in uppercase. We need to keep this in mind as the user might give lowercase input which cannot be directly used as a key in this dictionary.
3. Line 7 does not use the `int()` function that we used in `num2words.py` as the input here is not an integer but a string. Furthermore, since the dictionary keys are in uppercase, we have used the `upper()` function (introduced in section 2.3.4.2) to convert the user input to uppercase.
4. Line 8 extracts the corresponding value from the dictionary of Roman numerals – basically converting the Roman numeral to it's decimal equivalent using the dictionary. Note that no checks have been made for simplicity.
5. Line 9 prints the resultant decimal number and the rest of the program then proceeds to convert it to words.

7.8 Questions

1. What is a dictionary?
2. Explain all the ways by which we can access elements of a dictionary with examples.
3. Is there a way by means of which you can ensure that dictionary elements are maintained in the same order in which they were added?
4. Explain all possible ways of deleting elements from a dictionary with examples.
5. Illustrate the usage of the function `setdefault()` used on a dictionary when passed with arguments in the below scenarios:
    1. An element pair
    2. A single element which is already present as a key
    3. A single element which does not exist as a key in the dictionary
6. Illustrate with an example how to extract a key of a dictionary given its value .
7. Given 2 dictionaries `D1={'One':1, 'Two':2}` and `D2={'Three':3}`, state how can we append `D1` with `D2` in a single step so that resultant dictionary has 3 element pairs.
8. Illustrate how to swap keys with values in a dictionary.

7.9 Exercises

1. Prepare a dictionary that has odd numbers from `1` to `100` as keys with values as their cubes
2. Write a program that accepts only numbers from a user and at the time of exit, display the number of times the user had entered each number.
3. A shop has following commodities with its inventory maintained in a dictionary:

```
comm = {
  'chair' : 10,
  'table' : 22,
  'sofa-set': 2,
  'tv-unit': 0,
  'fan': 10,
  'table-lamp': 0,
  'iron-box': 0,
  'bed': 30
 }
```

and its price in another dictionary

```
price = {
  'chair': 2365,
  'table': 37752,
  'sofa-set': 23299,
  'tv-unit': 120344
  'bed': 40226
 }
```

Accept the number of items for each commodity sold from standard input and at the end do the following:

1. Evaluate the income made by the shop owner.
2. If user enters a commodity that is not available in the shop, display appropriate message.
3. If user enters a quantity that is beyond the available limit, display the size of the inventory.
4. Display the inventory available.

SUMMARY

- A dictionary is a collection of key-value pairs subject to the constraint that all the keys should be unique.
- A dictionary can be simply created using {} with colons separating the key from the value. Empty braces denote an empty dictionary.
- Elements of a dictionary can be accessed using [] or the get() method with the key being used as the index.
- The len() function gives the number of pairs in a dictionary.
- The for loop can be naturally used to iterate through the key-value pairs of a dictionary. It is also possible to iterate through the keys using the keys() method and iterate through the values using the values() method.
- The in and not in operators check for the existence of a key in a dictionary.
- New elements can be added to a dictionary using the setdefault() method or by assigning using the [] operator.
- The pop() method deletes a key-value pair from a dictionary while the popitem() method and del function remove a specified key-value pair identified by the key from a dictionary. The clear() method removes all key-value pairs from a dictionary.

8 Strings

In this chapter you will be able to:

- ☑ Represent strings and convert other objects to strings.
- ☑ Search for substrings in specific regions of a string.
- ☑ Split and join strings.
- ☑ Format strings based on your requirement.

Strings

A string is an immutable sequence of characters. They are one of most heavily used data types in programming. In Python, strings are instances of the class `str`.

Being a sequence, it is possible to iterate through the characters of a string using a simple for loop as shown below:

```
>>> for c in "Hello":
...     print(c)
...
H
e
l
l
o
```

8.1 Conversion From and To Strings

Objects of other types can be converted to strings in a couple of ways. Primitives (like integers and real numbers) can be easily converted to strings using the `str()` function, discussed in section 2.3.4. General conversions between primitives is discussed in section 20.1.1.3. Conversion of objects of other classes to strings is also discussed in section 20.1.1.3.

Bear in mind that values like `123` are integers whereas values like `'123'` and `"123"` are strings. More types of quoting for strings are discussed in section 2.1.2. Integers and strings are not implicitly convertible. Here are some examples of conversion:

```
>>> type(123)
<class 'int'>
>>> type('123')
<class 'str'>
>>> str(123)
'123'
>>> type(str(123))
<class 'str'>
>>> int('123')
123
>>> type(int('123'))
<class 'int'>
```

8.2 Searching In Strings

One of the basic operations we generally perform on strings is searching for a substring within it. Depending on what kind of search it is, suitable Python functions can be used:

1. If we want to check if a string contains a particular substring at a particular position, we can use `startswith()` or `endswith()`
2. If we want to check where a substring exists within the string, we can use `find()`, `rfind()`, `index()` or `rindex()`
3. If we want to count the number of occurrences of a substring within a string, we can use the `count()` function

8.2.1 Checking For the Existence of a Substring in a String Using startswith() and endswith()

8.2.1.1 The startswith() Function

To check if a string starts with a particular substring or not, we can use the `startswith()` function:

```
str.startswith(substring[,start=0[,end=len(str)]])
```

Forms:

```
1.      str.startswith(substring)
2.      str.startswith(substring,start)
3.      str.startswith(substring,start,end)
```

Form #1: `str.startswith(substring)`

The function returns `True` if the string `str` starts with the string `substring` and returns `False` otherwise:

```
>>> "This is a demo".startswith("This")
True
>>> "This is a demo".startswith("this")
False
```

We can also provide a tuple of strings instead of a single string for `substring`:

```
>>> "This is a demo".startswith(("This","this"))
True
>>> "This is a demo".startswith(("That","that"))
False
```

Thus, we can check if a string starts with one of a set of substrings or not.

Form #2: `str.startswith(substring,start)`

The function returns `True` if the string `str` starts with the string `substring`, with the check starting from the index `start` within the string `str`. In other words, we are checking for the existence of the substring `substring` at the index `start` in the string `str`:

```
>>> "This is a demo".startswith("is")
False
>>> "This is a demo".startswith("is",2)
True
>>> "This is a demo".startswith("is",5)
True
>>> "This is a demo".startswith("is",6)
False
```

Again, we can give a tuple of substrings instead of a single substring:

```
>>> "This is a demo".startswith(("this","is","that"),5)
True
```

Form #3: `str.startswith(substring,start,end)`

The function returns `True` if the substring `substring` is found starting from index `start` in the string `str`, but restricting the search till index `end` only. If the characters in `str` from index `start` till index `end` (excluding `end`) did not match the characters in `substring`, this function returns `False`. If the range of indices from `start` to `end` are not enough to compare against the entire length of `substring`, the function returns `False`:

```
>>> "This is a demo".startswith("is",5,10)
True
>>> "This is a demo".startswith("is",5,7)
True
>>> "This is a demo".startswith("is",5,6)
False
```

This form is the least useful form and is almost useless when a single substring is provided. It can be of use when a tuple of substrings are provided.

8.2.1.2 The endswith() Function

To check if a string ends with a particular substring or not, we can use the `endswith()` function:

```
str.endswith(substring[,start=0[,end=len(str)]])
```

Forms:

```
1.      str.endswith(substring)
2.      str.endswith(substring,start)
3.      str.endswith(substring,start,end)
```

Form #1: `str.endswith(substring)`

The function returns `True` if the string `str` ends with the string `substring` and returns `False` otherwise:

```
>>> "This is a demo".endswith("demo")
True
>>> "This is a demo".endswith("is")
False
```

We can also provide a tuple of strings instead of a single string for `substring`:

```
>>> "This is a demo".endswith(("is","demo"))
True
>>> "This is a demo".endswith(("is","was"))
False
```

Thus, we can check if a string ends with one of a set of substrings or not.

Form #2: `str.endswith(substring,start)`

The function returns `True` if the string `str` ends with the string `substring`, with the check starting from the index `start` within the string `str`. In other words, we are checking for the existence of the substring `substring` at the index `start` in the string `str` with nothing else following it:

```
>>> "This is a demo".endswith("is",2)
False
>>> "This is a demo".endswith("demo",10)
True
```

Again, we can give a tuple of substrings instead of a single substring:

```
>>> "This is a demo".endswith(("is","demo"),10)
True
```

Form #3: `str.endswith(substring,start,end)`

The function returns `True` if the substring `substring` is found starting from index `start` in the string `str`, but restricting the search till index `end` only and if there are no other characters following it in `str`. If the characters in `str` from index `start` till index `end` (excluding `end`) did not match the characters in `substring`, or if there are more characters after the match in the string `str`, this function returns `False`. If the range of indices from `start` to `end` are not enough to compare against the entire length of `substring`, the function returns `False`:

```
>>> "This is a demo".endswith("demo",10,14)
True
>>> "This is a demo".endswith("demo",10,13)
False
```

This form is the least useful form and is almost useless when a single substring is provided. It can be of use when a tuple of substrings are provided.

8.2.2 Checking For the Existence of a Substring in a String Using find(), rfind(), index() and rindex()

The previous section showed how we can check for the existence of a substring in a string at a specific position. That will be helpful if we know where to search. But what if we want to see whether a substring exists within a string, and if so, where within the string? This section concentrates on that.

8.2.2.1 Searching Using find()

The `find()` function searches for the first occurrence of a substring within a string and returns the index where it was found on success and returns `-1` if the substring was not found.

Complete syntax:

```
str.find(substring[,start=0[,end=len(str)]])
```

Forms:

```
1.      str.find(substring)
2.      str.find(substring,start)
3.      str.find(substring,start,end)
```

Form #1: `str.find(substring)`

The function searches for the first occurrence of the substring `substring` within a string `str` and returns the index where it was found on success and returns `-1` if the substring was not found:

```
>>> "This is a demo".find("is")
2
```

Form #2: `str.find(substring,start)`

The function searches for the first occurrence of the substring `substring` within a string `str`, starting from index `start` and returns the index where it was found on success and returns `-1` if the substring was not found:

```
>>> "This is a demo".find("is",3)
5
```

This form is useful if we want to search for a specific occurrence when there are multiple occurrences of a substring in a string. In fact, we can write a program that uses this form to show us all occurrences of a substring in a string:

finddemo.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of searching for all occurrences
4.   # of a substring in a string
5.
6.   string = input("Enter the string to search in:")
7.   substring = input("Enter the substring to search for:")
8.
9.   index=-1
10.  indices=[]
11.  while True:
12.      index = string.find(substring,index+1)
13.      if index==-1: break
14.      indices.append(index)
15.
16.  print(string)
17.  col=0
18.  for i in indices:
19.      print(' '*(i-col), end='')
20.      print('^',end='')
21.      col=i+1
22.  print()
```

Output:

```
Enter the string to search in:This is a demo
Enter the substring to search for:is
This is a demo
  ^  ^
```

Observation:

1. We accept the main string from the user in line 6 and store it in the variable `string`. We accept the substring to be searched for from the user in line 7 and store it in the variable `substring`.
2. We have an infinite loop in line 11 to search for all occurrences of the substring within the main string. We find the index of the next match in line 12 and break out of the loop in line 13 if there are no more matches.
3. To ensure that each time we search we don't end up searching the same position in the string, we use a starting position of `index+1` in line 12.
4. To further ensure that this code works correctly and we start searching from index `0`, we initialise `index` to `-1` in line 9 so that the expression `index+1` in

line 12 yields the value `0` initially.

5. Each time we obtain the index of a match, we add it to a list in line 14, with the list being created in line 10.
6. Lines 16-22 then display the main string with carats (`^`) below each occurrence of the substring in the string, depending on the indices stored in the list.

Form #3: `str.find(substring,start,end)`

This form is similar to the previous form, except that the search ends at index `end` (excluding `end`) instead of ending at the end of the string. If the substring `substring` was not found between indices `start` and `end-1`, this function fails and returns `-1`.

8.2.2.2 Searching Using rfind()

The `rfind()` function searches for the last (rightmost) occurrence of a substring within a string and returns the index where it was found on success and returns `-1` if the substring was not found.

Complete syntax:

```
str.rfind(substring[,start=0[,end=len(str)]])
```

Forms:

```
1.      str.rfind(substring)
2.      str.rfind(substring,start)
3.      str.rfind(substring,start,end)
```

Form #1: `str.rfind(substring)`

The function searches for the last occurrence of the substring `substring` within a string `str` and returns the index where it was found on success and returns `-1` if the substring was not found:

```
>>> "This is a demo".find("is")
5
```

Form #2: `str.rfind(substring,start)`

The function searches for the last occurrence of the substring `substring` within a string `str`, starting from index `start` and returns the index where it was found on success and returns `-1` if the substring was not found:

```
>>> "This is a demo".rfind("is",9)
-1
```

Practically, this form is not very useful and has few applications. The third form is more useful though.

Form #3: `str.rfind(substring,start,end)`

This form is similar to the previous form, except that the search ends at index `end` (excluding `end`) instead of ending at the end of the string. If the substring `substring` was not found between indices `start` and `end-1`, this function fails and returns `-1`.

This form is useful if we want to search for a specific occurrence when there are multiple occurrences of a substring in a string. In fact, we can write a program that uses this form to show us all occurrences of a substring in a string:

rfinddemo.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of searching for all occurrences
4.   # of a substring in a string in reverse order
5.
6.   string = input("Enter the string to search in:")
7.   substring = input("Enter the substring to search for:")
8.
9.   index=len(string)-len(substring)+1
10.  indices=[]
11.  while True:
12.      index = string.rfind(substring,0,index+len(substring)-
1)
13.      if index==-1: break
14.      indices.append(index)
15.
16.  print(string)
17.  col=0
18.  for i in reversed(indices):
19.      print(' '*(i-col), end='')
20.      print('^',end='')
21.      col=i+1
22.  print()
```

Output:

```
Enter the string to search in:This is a demo
Enter the substring to search for:is
This is a demo
  ^  ^
```

Observation:

1. This program is similar to the program `finddemo.py` covered in section 8.2.2.1. The changes are highlighted in bold.
2. Since we get the indices from right to left, each time the starting index should be `0` but the ending index should decrease based on each successful search.
3. Also, since we obtain indices from right to left, we finally reverse this order in line 18.

8.2.2.3 Searching Using index()

The `index()` function is just like the `find()` function, but generates a `ValueError` if the substring is not found, instead of simply returning `-1`. The `index()` function is preferred over the `find()` function in these cases:

1. We expect to find the substring, the only question being where is it found. It would be an unexpected case to not find the substring in the string at all.
2. We wish to use exception handling (covered in section 13) to process cases where the substring was not found in the string.

Conversely, we would prefer the `find()` function over the `index()` function in these cases:

1. We are unsure of the existence of the substring in the string.
2. We want a simple mechanism to deal with cases where the substring does not exist in the string.

Complete syntax:

```
str.index(substring[,start[,end]])
```

Forms:

```
1.      str.index(substring)
2.      str.index(substring,start)
3.      str.index(substring,start,end)
```

8.2.2.4 Searching Using rindex()

The `rindex()` function is just like the `rfind()` function, but generates a `ValueError` if the substring is not found, instead of simply returning `-1`. The `rindex()` function is preferred over the `rfind()` function in these cases:

1. We expect to find the substring, the only question being where is it found. It would be an unexpected case to not find the substring in the string at all.
2. We wish to use exception handling (covered in section 13) to process cases where the substring was not found in the string.

Conversely, we would prefer the `rfind()` function over the `rindex()` function in these cases:

1. We are unsure of the existence of the substring in the string.
2. We want a simple mechanism to deal with cases where the substring does not exist in the string.

Complete syntax:

```
str.rindex(substring[,start[,end]])
```

Forms:

```
1.      str.rindex(substring)
2.      str.rindex(substring,start)
3.      str.rindex(substring,start,end)
```

8.2.3 Counting Occurrences with count()

In the previous sections we have seen a variety of functions for searching substrings within strings. While the `startswith()` and `endswith()` functions return a Boolean value, the `find()`, `rfind()`, `index()` and `rindex()` functions return integers identifying indices.

In this section, we will examine the `count()` function that counts the number of occurrences of a substring within a string and returns the count.

Complete syntax:

```
str.count(substring[,start[,end]])
```

Forms:

```
1.      str.count(substring)
2.      str.count(substring,start)
3.      str.count(substring,start,end)
```

Form #1: `str.count(substring)`

This function returns the total number of times the substring `substring` is present in the string `str`:

```
>>> "abcdabadac".count("ab")
2
```

Form #2: `str.count(substring,start)`

This function returns the total number of times the substring `substring` is present in the string `str` starting from index `start`:

```
>>> "abcdabadac".count("ab",1)
1
```

Form #3: `str.count(substring,start,end)`

This function returns the total number of times the substring `substring` is present in the string `str` starting from index `start` and ending at index `end` (excluding `end`):

```
>>> "abcdabadac".count("ab",0,3)
1
```

8.3 Splitting Strings

Since strings can contain a large amount of data, data is generally stored as strings with separators. This makes it a common requirement to split the string at the separator to extract the pieces of data. There are also applications where we need to extract the part before and after a separator in a string. One more common application in this context is storage of multi-line data and extraction of lines - since lines are separated by the newline character, multiple lines of data can be stored in a single string, separated by the newline character. If need be, we should be able to extract the individual lines from such a string. This section concentrates on these kinds of requirements.

We will specifically see the following:

1. How to extract the part before and after a separator using `partition()` and `rpartition()`
2. How to split a string at a separator and extract all split parts of data using `split()` and `rsplit()`
3. How to extract the individual lines from a multi-line string using `splitlines()`

8.3.1 Splitting Strings Using partition() and rpartition()

The `partition()` and `rpartition()` functions allow extraction of the part before a separator, the separator itself and the part after a separator in a string. The difference is that `partition()` searches for the first (leftmost) occurrence of the separator while `rpartition()` searches for the last (rightmost) occurrence of the separator in the string.

8.3.1.1 The partition() Function

Syntax:

```
str.partition(separator)
```

The `partition()` function searches for the first (leftmost) occurrence of a separator within a string and returns a tuple containing 3 parts: the part before the separator, the separator itself and the part after the separator. In case the separator was not found in the string, it will still return a tuple containing 3 parts: the entire string followed by 2 null strings.

```
>>> "India:New Delhi".partition(":")
('India', ':', 'New Delhi')
```

In the above example, the string is partitioned at "`:`", giving rise to "`India`" as the part before the "`:`", the "`:`" separator itself and "`New Delhi`" as the part after the "`:`".

```
>>> "India:New Delhi China:Beijing".partition(":")
('India', ':', 'New Delhi China:Beijing')
```

In the above example, the string is partitioned at the first "`:`" with "`New Delhi China:Beijing`" being the part after the first "`:`".

```
>>> "India:New Delhi".partition("-")
('India:New Delhi', '', '')
```

In this example, since the separator "-" was not found in the string, the entire string is the first element of the tuple and the remainder elements are null strings.

8.3.1.2 The rpartition() Function

Syntax:

```
str.rpartition(separator)
```

The `rpartition()` function searches for the last (rightmost) occurrence of a separator within a string and returns a tuple containing 3 parts: the part before the separator, the separator itself and the part after the separator. In case the separator was not found in the string, it will still return a tuple containing 3 parts: 2 null strings followed by the entire string.

Here are the same examples we had used for `partition()`:

```
>>> "India:New Delhi".rpartition(":")
('India', ':', 'New Delhi')
>>> "India:New Delhi China:Beijing".rpartition(":")
('India:New Delhi China', ':', 'Beijing')
>>> "India:New Delhi".rpartition("-")
('', '', 'India:New Delhi')
```

8.3.2 Splitting Strings Using split() and rsplit()

The `partition()` and `rpartition()` functions make sense when we expect a single delimiter, or want to process the delimiters one at a time. The `split()` and `rsplit()` functions can work with multiple occurrences of delimiters and can give all the split parts between the delimiters.

8.3.2.1 The split() Function

Complete syntax:

```
str.split([,separator[,maxsplits]])
```

Forms:

```
1.      str.split()
2.      str.split(separator)
3.      str.split(separator,maxsplits)
```

Form #1: `str.split()`

This function splits the string `str` into pieces using the following rules:

1. Leading whitespaces are truncated
2. Trailing whitespaces are truncated
3. Any other sequence of whitespace within the string is considered as a delimiter

Thus, this function returns a list containing the parts of the string that are separated by one or more whitespaces.

As a special case, if the string `str` is empty or comprising of only whitespace characters, an empty list is returned.

Examples:

```
>>> "India:New Delhi China:Beijing".split()
['India:New', 'Delhi', 'China:Beijing']
>>> "  \t\nIndia:New    \n\r   Delhi\t\t\tChina:Beijing\n\n".split()
['India:New', 'Delhi', 'China:Beijing']
>>> "\t\t\n\r\n".split()
[]
>>> "".split()
[]
```

Form #2: `str.split(separator)`

Given a separator string, this function splits the string `str` into pieces wherever it finds the separator and returns a list of the split parts.

Examples:

```
>>> "125,342,264".split(",")
['125', '342', '264']
>>> "125,,342,264".split(",")
['125', '', '342', '264']
>>> "53269985".split("99")
['5326', '85']
```

Note:

1. The separator is a string and can be made up of multiple characters
2. If the separator occurs in consecutive succession, a null string is assumed to be present between them and is counted as one of the split pieces

Form #3: `str.split(separator,maxsplits)`

This function splits the string `str` into pieces whenever it finds the given separator. However, it ensures no more than `maxsplits` splits. Since each split gives one more piece, the total number of pieces this function returns is ideally `maxsplits+1`.

Note:

1. If `maxsplits` splits are not possible (because the `separator` is not present those many times), this function anyway returns the list of split parts disregarding `maxsplits`.
2. If `maxsplits` is less than the number of splits possible, the last string returned in the list is the remainder un-splitted part of `str`
3. If `separator` is `None`, the splitting is done as per form #1
4. If `maxsplits` is `-1` and `separator` is not `None`, the splitting is done as per form #2

Examples:

```
>>> "125,342,264".split(",",1) # 1 split, 2 pieces
['125', '342,264']
>>> "125,342,264".split(",",2) # 2 splits, 3 pieces
['125', '342', '264']
>>> "125,342,264".split(",",3) # 3 splits, only 2 possible
['125', '342', '264']
```

Special cases:

```
>>> "a b  c    d e  f".split(None,3)
['a', 'b', 'c', 'd e  f']
```

Since the `separator` is `None`, splitting is similar to form #1, but with a maximum of `3` splits only.

```
>>> "125,342,264".split(",",-1)
['125', '342', '264']
```

Since `maxsplits` is `-1` and `separator` is not `None`, splitting is similar to form #2.

8.3.3 Splitting Lines Using splitlines()

Given a multi-line string (a string containing newline characters), the `splitlines()` function can break them down into separate lines and return them as a list.

Complete syntax:

```
str.splitlines([keepends])
```

Forms:

```
1.      str.splitlines()
2.      str.splitlines(keepends)
```

Form #1: `str.splitlines()`

This function breaks down a multi-line string into a list of strings corresponding to each line. The splitting is done when it finds a suitable newline character (for simplicity, can be considered to be `\n`, though there are other characters including `\r` and Unicode newline characters). The newline character that was found is not present in the list of strings it returns.

Example:

```
>>> "Hello\nWorld\nHow are you?".splitlines()
['Hello', 'World', 'How are you?']
```

Form #2: `str.splitlines(keepends)`

This function works just like form #1, but if `keepends` is `True` then this retains the newline character at the end of each line in the list. If `keepends` is `False`, this works just like form #1.

Examples:

```
>>> "Hello\nWorld\nHow are you?".splitlines(True)
['Hello\n', 'World\n', 'How are you?']
>>> "Hello\nWorld\nHow are you?".splitlines(False)
['Hello', 'World', 'How are you?']
```

8.4 Joining Strings

The opposite of splitting is joining. We have learnt how to split a string into pieces using various functions. In this section, we will learn how to join various pieces together to form a new string.

8.4.1 Joining Strings Using join()

Syntax:

```
str.join(iterable)
```

The `join()` function concatenates all the strings in the `iterable` sequence (list, tuple, etc.) using the string `str` as the separator between them and returns the new string. As a special case, if `str` is a null string, there won't be any separator between the strings.

Examples:

```
>>> ",".join(("12","24","36"))
'12,24,36'
>>> " and ".join(("12","24","36"))
'12 and 24 and 36'
>>> "".join(("12","24","36"))
'122436'
```

8.5 Modifying Strings

The `split` and `join` operations covered in previous sections help break down a string into pieces and assemble them back into a string again. Strings being immutable cannot be modified. However, in this section, we will examine a few functions that derive a new string out of an existing one by performing some operation on it.

8.5.1 Stripping Strings Using lstrip(), rstrip() and strip()

Stripping refers to removal of certain characters from the ends of a string.

8.5.1.1 Left Stripping Using lstrip()

The `lstrip()` function strips leading characters (leftmost characters) from a string and returns the new string.

Complete syntax:

```
str.lstrip([chars])
```

Forms:

```
1.      str.lstrip()
2.      str.lstrip(chars)
```

Form #1: `str.lstrip()`

This function left-strips the string `str`. It ends up removing the leading whitespaces from the string `str` and returns the new string:

```
>>> "  \t\r\nHello \t\r\n".lstrip()
'Hello \t\r\n'
```

Note:

1. If the string `str` does not start with a whitespace, this function simply returns the original string without any changes

Form #2: `str.lstrip(chars)`

If we wish to strip other characters apart from whitespaces, this form of the function can be used. We can specify all the characters (as a single string) that we wish to strip, if found at the beginning of the string `str`.

The following example removes all digits from the beginning of a string:

```
>>> "233Hello466World599".lstrip("0123456789")
'Hello466World599'
```

8.5.1.2 Right Stripping Using rstrip()

The `rstrip()` function strips trailing characters (rightmost characters) from a string and returns the new string.

Complete syntax:

```
str.rstrip([chars])
```

Forms:

```
1.      str.rstrip()
2.      str.rstrip(chars)
```

Form #1: `str.rstrip()`

This function right-strips the string `str`. It ends up removing the trailing whitespaces from the string `str` and returns the new string:

```
>>> "  \t\r\nHello \t\r\n".rstrip()
'  \t\r\nHello'
```

Note:

1. If the string `str` does not end with a whitespace, this function simply returns the original string without any changes

Form #2: `str.rstrip(chars)`

If we wish to strip other characters apart from whitespaces, this form of the function can be used. We can specify all the characters (as a single string) that we wish to strip, if found at the end of the string `str`.

The following example removes all digits from the end of a string:

```
>>> "233Hello466World599".rstrip("0123456789")
'233Hello466World'
```

8.5.1.3 Stripping Using strip()

The `strip()` function strips leading and trailing characters (leftmost and rightmost characters) from a string and returns the new string.

Complete syntax:

```
str.strip([chars])
```

Forms:

```
1.      str.strip()
2.      str.strip(chars)
```

Form #1: `str.strip()`

This function left-strips and right-strips the string `str`. It ends up removing the leading and trailing whitespaces from the string `str` and returns the new string:

```
>>> "  \t\r\nHello \t\r\n".strip()
'Hello'
```

Note:

1. If the string `str` neither starts nor ends with a whitespace, this function simply returns the original string without any changes

Form #2: `str.strip(chars)`

If we wish to strip other characters apart from whitespaces, this form of the function can be used. We can specify all the characters (as a single string) that we wish to strip, if found at the beginning or end of the string `str`.

The following example removes all digits from the beginning and end of a string:

```
>>> "233Hello466World599".strip("0123456789")
'Hello466World'
```

8.5.2 Replacing Substrings Using replace()

Replacing a substring by another within a string is a common requirement – but strings being immutable in Python means that any such modification would require the creation of a new string. The `replace()` function helps replace one or more occurrences of a substring by another, returning a new string with the replacements performed.

Complete syntax:

```
str.replace(old,new[,count])
```

Forms:

```
1.      str.replace(old,new)
2.      str.replace(old,new,count)
```

Form #1: `str.replace(old,new)`

This function replaces all occurrences of the substring `old` in the string `str` by the substring `new` and returns the new string:

```
>>> "Delhi-Bombay-Bangalore-Bombay".replace("Bombay","Mumbai")
'Delhi-Mumbai-Bangalore-Mumbai'
```

Form #2: `str.replace(old,new,count)`

In this form, the function performs at most `count` replacements from the beginning of the string `str`:

```
>>> "Delhi-Bombay-Bangalore-Bombay".replace("Bombay","Mumbai",1)
'Delhi-Mumbai-Bangalore-Bombay'
```

8.5.3 Substituting Substrings Using Template Strings

An advanced form of string substitution is by making use of the `Template` class in the `string` module.

A template is a string that contains placeholders for values to be substituted. A single template can be used multiple times to generate multiple strings, given values to be substituted. Thus, a template generates a string whenever values are substituted.

8.5.3.1 Creating a Template

Templates are objects of the class `Template` and use a string to represent the template itself internally. This template string can contain placeholders for values and these are indicated by the '`$`' symbol. Whenever a '`$`' is found in the template string, it's interpretation is performed as follows:

1. If the '`$`' is followed by a valid identifier, the identifier is considered to be the name of the placeholder.
2. If the '`$`' is followed by '`{`', then the identifier found till '`}`' is considered to be the name of the placeholder.
3. If the '`$`' is followed by another '`$`', it is considered as a single '`$`' character

We will examine substitution later – first we'll concentrate on the template string.

Form #1: `$identifier`

This is the simplest form wherein the identifier starts immediately after the '`$`' and goes on till the first character that cannot be used in an identifier. Here are some examples with identifiers (placeholders) highlighted:

```
$capital is the capital of $country
The sum of $x and $y is $z
$abc123xyz789!@#
```

Form #2: `${identifier}`

A problem with form #1 is what if we want an identifier to be immediately followed by alphanumeric characters? How will it understand where the identifier ends and text begins? In such cases, we can use this form where the identifier is enclosed in braces. Here are some examples with the identifiers highlighted:

```
${thousand}000
${a}_${b}
```

Form #3: `$$`

Since the '`$`' symbol has a special meaning as an indication of a placeholder, what if we want a simple '`$`' character within the template string? The solution is to use two '`$`' characters instead as shown in the examples below:

```
$$1000
$$x=10
```

Now that we know how to frame template strings, let us see the code behind that in Python. Here is an example:

```
>>> from string import Template
>>> t=Template("The sum of $x and $y is $z")
```

Observation:

1. We need the `Template` class which is present in the `string` module. The first line takes care of that.
2. We create a `Template` object using a template string and store a reference to that object in the variable `t`.
3. Using this template, we'll be able to create multiple strings by substituting values, which we will see soon.

8.5.3.2 Substituting Values in a Template Using substitute()

The `substitute()` function of `Template` allows substitution of values within a template and returns a string after substitution.

There are multiple forms of `substitute()`:

```
1.      template.substitute(dict)
2.      template.substitute(keyargs)
3.      template.substitute(dict,keyargs)
```

Form #1: `template.substitute(dict)`

This function substitutes values from the dictionary `dict` into the placeholders in the template `template` and returns the resultant string after substitution. The dictionary is supposed to contain all the placeholder identifiers as keys with corresponding suitable values for substitution.

In case a placeholder identifier exists in the template `template` but not as a key in the dictionary `dict`, this will generate a `KeyError`.

In case a key exists in the dictionary `dict` but not in the template `template`, the key is ignored.

In case a '`$`' in the template `template` is not followed by an identifier, this will generate a `ValueError`.

Example:

```
>>> from string import Template
>>> t=Template("The sum of $x and $y is $z")
>>> d=dict(x=10,y=20,z=30)
>>> t.substitute(d)
'The sum of 10 and 20 is 30'
>>> t.substitute(dict(x=2,y=12,z=14))
'The sum of 2 and 12 is 14'
>>> t.substitute(dict(x=2,y=12))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.3/string.py", line 121, in substitute
    return self.pattern.sub(convert, self.template)
  File "/usr/lib64/python3.3/string.py", line 111, in convert
    val = mapping[named]
KeyError: 'z'
>>> t.substitute(dict(x=2,y=12,z=14,a=100))
'The sum of 2 and 12 is 14'
```

NOTE:

The statement "`from string import Template`" needs to be done only once per Python session. Repetitions of this statement are tolerated, however.

Form #2: `template.substitute(keyargs)`

It is possible to directly pass keyword arguments (covered in section 10.6 in detail) instead of passing a dictionary. This can be convenient when there are few placeholders in the template.

This function substitutes values from the keyword arguments `keyargs` into the placeholders in the template `template` and returns the resultant string after substitution. The keyword arguments are supposed to contain all the placeholder identifiers with corresponding suitable values for substitution.

In case a placeholder identifier exists in the template `template` but not in the keyword arguments `keyargs`, this will generate a `KeyError`.

In case a keyword argument exists in `keyargs` but not in the template `template`, the keyword argument is ignored.

In case a '`$`' in the template `template` is not followed by an identifier, this will generate a `ValueError`.

Example:

```
>>> from string import Template
>>> t=Template("The sum of $x and $y is $z")
>>> t.substitute(x=2,y=12,z=14)
'The sum of 2 and 12 is 14'
>>> t.substitute(x=2,y=12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.3/string.py", line 121, in substitute
    return self.pattern.sub(convert, self.template)
  File "/usr/lib64/python3.3/string.py", line 111, in convert
    val = mapping[named]
KeyError: 'z'
>>> t.substitute(x=2,y=12,z=14,a=100)
'The sum of 2 and 12 is 14'
```

NOTE:

The order of the keys in the dictionary in form #1 and the order of the keyword arguments in form #2 does not matter!

Form #3: `template.substitute(dict, keyargs)`

This form is a combination of the previous two forms: it uses both the dictionary `dict` and the keyword arguments `keyargs` to perform substitution in the template `template` and returns the string after substitution.

If a placeholder is found both as a key in the dictionary `dict` as well as a keyword argument in `keyargs`, preference is given to the keyword argument in `keyargs`.

Examples:

```
>>> from string import Template
>>> t=Template("The sum of $x and $y is $z")
>>> t.substitute(dict(x=2,y=12,z=14),z=100)
'The sum of 2 and 12 is 100'
```

Observation:

1. The value of the template identifier `z` is taken from the keyword argument (`100`), even though the dictionary contained a different value (`14`) since preference is given to keyword arguments.
2. The values of the template identifiers `x` and `y` are taken from the dictionary since no keyword arguments were provided for them.

8.5.3.3 Substituting Values in a Template Using safe_substitute()

The `safe_substitute()` function is very similar to the `substitute()` function covered in the previous section. Here are the differences:

1. If a placeholder in the template is not found in the dictionary and keyword arguments, the placeholder is retained as it is without generating any `KeyError`.
2. Any '`$`' that is not followed by an identifier is retained as it is without generating any `ValueError`.

Example:

```
>>> from string import Template
>>> t=Template("The sum of $x and $y is $z")
>>> t.safe_substitute(dict(x=2),z=100)
'The sum of 2 and $y is 100'
```

8.5.4 String Translation Using maketrans() and translate()

String translation is the mapping of each character of a string to zero or more replacement characters. One of the ways of doing this in Python is by making use of the `translate()` function for translation using a translation table by creating the translation table using the `maketrans()` function. This section concentrates on how to use these functions to perform string translation.

While the `translate()` function performs the actual string translation, we will first see how to build translation tables using `maketrans()`.

Complete syntax:

```
str.maketrans(x[,y[,z]])
```

Note:

1. This is a static method (section 12.4.4) of `str` class and hence we will literally use `str` to invoke it, not any string instance.
2. The meanings of the parameters `x`, `y` and `z` depend on how many parameters have been given. We will explore each case separately.

Forms:

```
1.      str.maketrans(dict)
2.      str.maketrans(lookup,replacement)
3.      str.maketrans(lookup,replacement,discard)
```

8.5.4.1 Building Translation Tables to Translate Based on Ordinals

Ordinals are the codes (integers) that represent characters. ASCII and Unicode are two of the most popular systems to represent characters and therefore strings. ASCII codes and Unicode code values are ordinals.

The first approach to building a translation table is by mapping an ordinal to it's replacement using a dictionary. The keys of the dictionary represent what has to be translated while the values of the dictionary represent the replacement.

Each key of the dictionary can be:

1. An ordinal (ASCII or Unicode)
2. Actual character itself

Even if an actual character is given, it is converted to it's ordinal. Thus, the keys of the dictionary are ordinals.

Each value in the dictionary can be:

1. A Unicode ordinal
2. A string of any length
3. The special value `None`

Syntax:

```
str.maketrans(dict)
```

The actual translation will be done by the `translate()` function, but the translation mechanism is finalised here. When the `translate()` function performs the translation of a string using this dictionary, it performs the following steps for each character found in the string to be translated:

1. If the character's ordinal is found in the keys of the dictionary and it's corresponding value is not `None`, the value is substituted instead.
2. In the above case, if the value found is `None`, the original character is silently deleted in the translated string.
3. If the character's ordinal was not found in the keys of the dictionary, the character is retained as it is without any changes in the translated string.

Here is an example of building a translation table that translates octal numbers to lowercase alphabets:

```
>>>
d={'1':'a','2':'b','3':'c','4':'d','5':'e','6':'f','7':'g','8':None,'9
':None}
>>> t=str.maketrans(d)
>>> "hello 0468 world".translate(t)
'hello 0df world'
```

Observation:

1. The keys of the dictionary are characters from `0` to `9` (in quotes), not integers! Note that the keys can be either ordinals (integers) or characters. What we require here are the characters `0` to `9` which have ordinals `48`-`57`.
2. The Octal system only supports digits from `0` to `7`. Hence, we decide to eliminate `8` and `9` using the value `None` for them.
3. We decide to retain the character `0` as it is and hence it is not present in the dictionary.
4. Thus, `0` is retained, `1`-`7` are replaced by `a`-`g` and `8`-`9` are deleted. All other characters are retained.

8.5.4.2 Building Translation Tables to Translate Based on Translation Strings

Syntax:

```
str.maketrans(lookup,replacement)
```

In the second form, given 2 strings of equal length, each character of the string `lookup` will be mapped on to the corresponding character in the string `replacement` during translation. Characters in the string to be translated that are not present in lookup will be retained during translation without any changes.

Here is an example that maps lowercase vowels to uppercase:

```
>>> t=str.maketrans("aeiou","AEIOU")
>>> "This is a demonstration".translate(t)
'ThIs Is A dEmOnstrAtIOn'
```

8.5.4.3 Building Translation Tables to Translate Based on Translation Strings and Discard Strings

Syntax:

```
str.maketrans(lookup,replacement,discard)
```

This third form is similar to the second form discussed in the previous section, except for the third parameter: a string of characters to be discarded during translation.

While the second form retains all characters not found in lookup during translation, the third form discards all characters present in the string `discard` and retains only those characters without changes that are neither present in `lookup` nor in `discard`.

Here is the same example we used in the previous section for converting lowercase vowels to uppercase, but with the added feature of discarding spaces:

```
>>> t=str.maketrans("aeiou","AEIOU"," ")
>>> "This is a demonstration".translate(t)
'ThIsIsAdEmOnstrAtIOn'
```

8.6 Padding Strings

Python has functions for filling strings with spaces in a variety of ways – and that's the focus of this section.

8.6.1 Justifying a String

A string of a given length can be increased to a specified width by filling it with any character of choice (space by default). There are 3 functions in Python that do this for you, depending on where you want the original string content to be present in the final expanded string:

`ljust()` - to left justify the content within an expanded string

`rjust()` - to right justify the content within an expanded string

`center()` - to center justify the content within an expanded string

8.6.1.1 The ljust() Function

Complete syntax:

```
str.ljust(width[,fill])
```

Forms:

1. `str.ljust(width)`
2. `str.ljust(width, fill)`

Form #1: `str.ljust(width)`

The `ljust()` function left-justifies a given string within an expanded string of the given width, filling in spaces as required and returns the new string. As a special case, if the given width is less than the length of the string, the operation is ignored and the original string is returned.

Examples:

```
>>> s="Hello"
>>> s.ljust(20)
'Hello               '
>>> s.ljust(2)
'Hello'
```

Form #2: `strl.ljust(width, fill)`

If some other character is needed for filling instead of spaces, the required character can be given as the argument `fill`:

```
>>> s.ljust(20,"=")
'Hello==============='
>>> s.ljust(2,"=")
'Hello'
```

8.6.1.2 The rjust() Function

Complete syntax:

```
str.rjust(width[,fill])
```

Forms:

1. `str.rjust(width)`
2. `str.rjust(width, fill)`

Form #1: `str.rjust(width)`

The `rjust()` function right-justifies a given string within an expanded string of the given width, filling in spaces as required and returns the new string. As a special case, if the given width is less than the length of the string, the operation is ignored and the original string is returned.

Examples:

```
>>> s.rjust(20)
'               Hello'
>>> s.rjust(2)
'Hello'
```

Form #2: `str.rjust(width, fill)`

If some other character is needed for filling instead of spaces, the required character can be given as an additional argument:

```
>>> s.rjust(20,"=")
'===============Hello'
>>> s.rjust(2,"=")
'Hello'
```

8.6.1.3 The center() Function

Complete syntax:

```
str.center(width[,fill])
```

Forms:

1. `str.center(width)`
2. `str.center(width, fill)`

Form #1: `str.center(width)`

The `center()` function center-justifies a given string within an expanded string of the given width, filling in spaces as required and returns the new string. As a special case, if the given width is less than the length of the string, the operation is ignored and the original string is returned.

Examples:

```
>>> s.center(20)
'       Hello        '
>>> s.center(2)
'Hello'
```

Form #2: `str.center(width, fill)`

If some other character is needed for filling instead of spaces, the required character can be given as an additional argument:

```
>>> s.center(20,"=")
'=======Hello========'
>>> s.center(2,"=")
'Hello'
```

8.6.2 Filling a String Using zfill()

If a given string has numeric content, probably the best method of padding is by using `zfill()` which uses '0' padding, right-justifies content, but ensures that the padding starts only after the sign, if any.

Syntax:

```
str.zfill(width)
```

Examples:

```
>>> "123".zfill(5)
'00123'
>>> "+123".zfill(5)
'+0123'
```

8.6.3 Expanding Tabs in a String Using expandtabs()

Given a string with tab characters (`\t`) within it, we can expand these tabs to spaces using the `expandtabs()` function:

```
str.expandtabs(tabsize=8)
```

Forms:

```
1.      str.expandtabs()
2.      str.expandtabs(tabsize)
```

In the first form, the `tabsize` is assumed to be `8` whereas in the second form, we can provide the `tabsize`.

The tabsize is used to divide the string into *zones*, starting from column 0. Thus, with a tabsize of 8, the zones will start from column 0, 8, 16, 24, etc. With a tabsize of 2, the zones will start from 0, 2, 4, 6, etc. A tab character is supposed to move the *cursor* to the beginning of the next zone. This effect is simulated by this function by adding spaces instead of the tabs.

Examples:

```
>>> s="a\tb\t\tc"
>>> s.expandtabs()
'a       b               c'
 012345678901234567890123 : Reference Ruler

>>> s.expandtabs(2)
'a b   c'
 0123456 : Reference Ruler
```

NOTE:

The “Reference Ruler” in the output above has been added for readability in this book and is not printed by the Python interpreter!

8.7 Formatting Strings

We have already seen the f`ormat()` function in section 2.7.2. We will examine that function in detail in this section.

We know the following about the `format()` function:

1. It allows us to substitute values inside a string
2. It allows us to substitute values in order if we use braces without identifying the argument
3. It allows us to substitute values in any order if we identify the element by it's argument number in the placeholder
4. It allows us to substitute the same value multiple times if we use the same argument number in multiple placeholders
5. It allows us to identify the argument by it's name as well as by it's position

In this section, we will explore more properties and possibilities.

Each placeholder can specify 2 things:

1. The argument to be used
2. The formatting that needs to be applied on the value of that argument

These 2 parts are separated by a colon (:) and thus have these syntax:

```
{argumentNumber:argumentFormat}
{argumentName:argumentFormat}
```

Both the parts are optional. If `argumentNumber` and `argumentName` are not given, it will extract an argument in order. If the `argumentFormat` is not given, it will not apply any special formatting – it will display the value as it is using as many columns as necessary to display that value. If `argumentFormat` is omitted, the separating colon is not required.

The complete syntax of the `argumentFormat` is shown below:

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

Different parts of this syntax will be dealt with in detail in different sections as documented in the below table:

Part	Meaning	Section
`fill`	Fill character to be used for padding	8.7.4
`align`	Alignment of the contents within the field	8.7.3
`sign`	Sign to be displayed for numeric entities	8.7.6.2
`#`	Display Base Prefix	8.7.6.4
`0`	Zero Padding	8.7.6.1
`width`	Width of the Field	8.7.2
`grouping`	Digit Grouping	8.7.6.3
`.precision`	Precision of the Field	8.7.5
`type`	Type of the Field	8.7.1

Let us start with the format type that documents the type of the field.

8.7.1 Format Types

The format type specifies how the field contents have to be represented. The possibilities depend on whether the field content is a string, an integer or a real number.

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

8.7.1.1 Format Types for Integers

Integers can be displayed in a variety of bases as shown in the table below:

Type	*Meaning*
`b`	Binary Format
`d`	Decimal Format
`o`	Octal Format
`x`	Hexadecimal Format using Lowercase Alphabets
`X`	Hexadecimal Format using Uppercase Alphabets

Table 15: Format Types for Integers

In addition to bases, the following format types are also applicable:

Type	*Meaning*
`n`	Number formatted using current locale
`c`	Character corresponding to the Unicode

Table 16: Additional Format Types for Integers

NOTE:

If no format is selected for an integer, it will be displayed in decimal (the `d` format)

Examples:

```
>>> '{}'.format(15) # Default format for integer
'15'
>>> '{:d}'.format(15) # Decimal format for integer
'15'
>>> '{:b}'.format(15) # Binary format for integer
'1111'
>>> '{:x}'.format(15) # Hexadecimal (Lowercase) format for integer
'f'
>>> '{:X}'.format(15) # Hexadecimal (Uppercase) format for integer
'F'
>>> '{:n}'.format(15000) # Numeric (Locale-Specific) format for
integer
'15000'
>>> '{:c}'.format(65) # Character Format (A=ASCII 65)
'A'
```

8.7.1.2 Format Types for Real Numbers

Real numbers can be displayed in fixed-point format, exponentiation format, general format, numeric format or as a percentage as shown in the table below:

Type	Meaning
`f`	Fixed-Point Format
`F`	Fixed-Point Format using uppercase for `inf` and `nan`
`e`	Exponentiation Format
`E`	Exponentiation Format using `E` instead of `e` to denote exponentiation
`g`	General Format (May use fixed-point or exponentiation depending on which is more appropriate)
`G`	General Format using `E` instead of `e` to denote exponentiation (if exponentiation is used)
`n`	Numeric Format (uses current locale for formatting)
`%`	Percentage Format

Table 17: Format Types for Real Numbers

NOTE:

If no format is selected for a real number, it will be displayed in general format (the `g` format), but guarantees at least 1 digit after the decimal point unlike the `g` format.

Examples:

```
>>> '{:f}'.format(12.165) # Fixed Point Format
'12.165000'
>>> '{:e}'.format(12.165) # Exponentiation Format
'1.216500e+01'
>>> '{:E}'.format(12.165) # Exponentiation Format
'1.216500E+01'
>>> '{:g}'.format(12.165) # General Format
'12.165'
>>> '{:n}'.format(12.165) # Numeric Format
'12.165'
>>> '{}'.format(12.165) # Default Format
'12.165'
>>> '{:%}'.format(12.165) # Percentage Format
'1216.500000%'
```

8.7.1.3 Format Types for Strings

The `s` format is the only format that can be applied for strings and is also the default format, and thus, is not very useful (apart from improving readability):

```
>>> '{:s}'.format("Ram") # String Format
'Ram'
>>> '{}'.format("Ram") # Default Format
'Ram'
```

NOTE:

Format types **cannot** be used for type conversion! For instance, the `d` format type cannot be used to convert a real number to an integer. Any attempt to perform such conversions will result in an error.

8.7.2 Format Width

Each field, when displayed by default, will occupy as many columns (characters) as needed to completely represent it in the chosen format type. It is possible to specify the field width, however, to ensure that at least those many columns are reserved for the display of the field.

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

NOTE:

If the given width is insufficient, it will use up as many columns as needed ignoring the field width request! Thus, the field width can be considered to be the *minimum* field width!

Examples:

```
>>> '{:3d}'.format(2) # 1-digit number with field width of 3
'  2'
>>> '{:3d}'.format(12) # 2-digit number with field width of 3
' 12'
>>> '{:3d}'.format(1234) # 4-digit number with field width of 3
'1234'
```

8.7.3 Format Alignment

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

When a width is given, and the width exceeds the default width requirement of a field, spaces are used by default to fill in the rest of the columns. In such cases, the alignment controls where the spaces get added. The possible values for this and their interpretation are shown in the table below:

Alignment	*Meaning*
<	Left-align field (add spaces to the right of the content)
>	Right-align field (add spaces to the left of the content)
^	Center-align field (add spaces on either side of the content as necessary)
=	Right-align numeric field (add spaces on the left, but after any sign)

Table 18: Format Alignment

Examples:

```
>>> '{:<10}'.format(12) # Left-align "12" in 10 columns
'12        '
>>> '{:>10}'.format(12) # Right-align "12" in 10 columns
'        12'
>>> '{:^10}'.format(12) # Center-align "12" in 10 columns
'    12    '
>>> '{:=10}'.format(12) # Right-align "12" in 10 columns, sign first
'        12'
>>> '{:=10}'.format(-12) # Right-align "-12" in 10 columns, sign first
'-       12'
```

Note:

1. The = alignment is helpful to produce a list of numbers, all aligned with their signs in one column.
2. When no alignment is specified, strings are left-aligned whereas numbers are right-aligned by default.

8.7.4 Format Fill

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

While the alignment uses spaces by default to fill the unused columns, any other character can be used to fill in the unused columns. We will revisit the same examples given in the previous section, using "*" instead of the default spaces:

```
>>> '{:*<10}'.format(12) # Left-align "12" in 10 columns
'12********'
>>> '{:*>10}'.format(12) # Right-align "12" in 10 columns
'********12'
>>> '{:*^10}'.format(12) # Center-align "12" in 10 columns
'****12****'
>>> '{:*=10}'.format(12) # Right-align "12" in 10 columns, sign first
'********12'
>>> '{:*=10}'.format(-12) # Right-align "-12" in 10 columns, sign
first
'-*******12'
```

8.7.5 Format Precision

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

The exact effect of precision depends on what we are displaying and which format type we are using. We will therefore see individual cases:

8.7.5.1 Format Precision for Fixed-Point Format

When a real number is being displayed using fixed-point format (`f` or `F`), the precision specifies the number of decimal places required, as shown in the examples below:

```
>>> '{:.2f}'.format(12.3456789)
'12.35'
>>> '{:.12f}'.format(12.3456789)
'12.345678900000'
```

Note:

1. If the precision is less than the number of fractional digits, the fractional digits are rounded.
2. If the precision is greater than the number of fractional digits, '0' is added.

8.7.5.2 Format Precision for Floating-Point General Format

When a real number is being displayed using general format (`g` or `G`), the precision specifies the maximum number of digits displayed, as shown in the examples below:

```
>>> '{:.5g}'.format(12.3456789)
'12.346'
>>> '{:.10g}'.format(12.3456789)
'12.3456789'
>>> '{:.15g}'.format(12.3456789)
'12.3456789'
```

8.7.5.3 Format Precision for Strings

When a string is being displayed, the precision specifies the maximum number of characters to be displayed from the string, as shown in the examples below:

```
>>> '{:5.2s}'.format('abcdefghijkl')
'ab   '
>>> '{:5.4s}'.format('abcdefghijkl')
'abcd '
```

NOTE:

In the above examples, the width is `5` implying that `5` columns are reserved for the field. The alignment is left by default and spaces are used by default for filling the field. The precision controls how many characters from the string are available.

8.7.6 Numeric Formatting

In this section, we will explore the left overs of the format specification.

8.7.6.1 Zero Padding

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

The presence of a `0` before the width indicates 0-padding instead of space padding:

```
>>> '{:5d}'.format(12)
'   12'
>>> '{:05d}'.format(12)
'00012'
>>> '{:05d}'.format(-12)
'-0012'
```

8.7.6.2 Sign Preservation

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

The sign controls how positive and negative numbers are differentiated and handled. The possibilities are listed in the table below:

Sign	*Meaning*
-	Negative numbers are prefixed with '-' and positive numbers have no prefix
+	Negative numbers are prefixed with '-' and positive numbers are prefixed with '+'
space (' ')	Negative numbers are prefixed with '-' and positive numbers are prefixed with a space (' ')

Table 19: Sign Preservation in Format

The default behaviour (when no `sign` is used) is that of the '-' `sign` in the table.

Examples:

```
>>> '{:d}'.format(12)
'12'
>>> '{:d}'.format(-12)
'-12'
>>> '{:-d}'.format(12)
'12'
>>> '{:-d}'.format(-12)
'-12'
>>> '{:+d}'.format(12)
'+12'
>>> '{:+d}'.format(-12)
'-12'
>>> '{: d}'.format(12)
' 12'
>>> '{: d}'.format(-12)
'-12'
```

8.7.6.3 Digit Grouping

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

The `grouping` decides what separator will be used between groups of digits. If no `grouping` is specified, by default no separator is used. If a comma (`,`) is used as `grouping`, then groups of digits will be separated by commas as shown below:

```
>>> '{:,d}'.format(123456789)
'123,456,789'
```

8.7.6.4 Alternate Representation

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

The presence of the '`#`' in the format specification indicates a request for an alternate representation – a representation different from the default one. This alternate representation exists only for numbers and differs from type to type.

For instance:

1. The alternate representation for binary, octal and hexadecimal types is the presence of '`0b`', '`0o`' and '`0x`' respectively.
2. The alternate representation for the '`f`' and '`F`' types is to display the decimal point even when there is no fractional part.
3. The alternate representation for the '`g`' and '`G`' types is to retain trailing zeros.

Examples:

```
>>> '{:b}'.format(12)
'1100'
>>> '{:#b}'.format(12)
'0b1100'
>>> '{:o}'.format(12)
'14'
>>> '{:#o}'.format(12)
'0o14'
>>> '{:x}'.format(12)
'c'
>>> '{:#x}'.format(12)
'0xc'
>>> '{:X}'.format(12)
'C'
>>> '{:#X}'.format(12)
'0XC'
```

```
>>> '{:.0f}'.format(12)
'12'
>>> '{:#.0f}'.format(12)
'12.'
>>> '{:g}'.format(1200000000)
'1.2e+09'
>>> '{:#g}'.format(1200000000)
'1.20000e+09'
>>> '{:G}'.format(1200000000)
'1.2E+09'
>>> '{:#G}'.format(1200000000)
'1.20000E+09'
```

8.8 Questions

1. Explain any 5 operations that can be done on strings in Python with examples.
2. How is the `index()` function different from `find()` function in Python?
3. Explain the `lstrip()`, `rstrip()` and `strip()` methods of `str` class.
4. Explain template processing using the `Template` class in Python.
5. Explain string translation in Python.
6. Explain the string functions used for string justification in Python.
7. Write a short note on the `format()` method of `str` class.

8.9 Exercises

1. Write a Python script to calculate the number of occurrences of a given character in an any given string.
2. Write a Python script to count the number of occurrences of each word in a given string.
3. Write a Python script to form a string out of the first 2 and last 2 characters from the given string. For example, given the string "`I am a student`", the output expected is "`I nt`".

SUMMARY

- A string is an immutable sequence of characters.
- To search within strings, these methods can be used: startswith(), endswith(), find(), rfind(), index(), rindex().
- Strings can be split using partition(), rpartition(), split(), rsplit() and splitlines() methods. Strings can be joined using join().
- Strings can be stripped using lstrip(), rstrip and strip(). Strings can be replaced using replace().
- Advanced template-based substitution is possible using string.Template. Character-wise translation is also possible using maketrans() and translate().
- Strings can be justified using ljust(), rjust() and center(). Strings can be zero-filled using zfill() and tab characters can be expanded using expandtabs().
- Strings can be formatted using the format() method.

9 Regular Expressions

In this chapter you will be able to:

- ☑ Understand what are Regular Expressions and learn to frame them.
- ☑ Learn how to search for patterns within strings.
- ☑ Deal with groups within search patterns and extract grouped parts.
- ☑ Work with matched parts within the string.
- ☑ Split strings based on string patterns.
- ☑ Replace matched patterns with custom replacement strings.
- ☑ Pre-compile frequently used regular expressions to improve efficiency.

REGULAR EXPRESSIONS

9.1 Introduction to Regular Expressions

A *regular expression* is a pattern matching expression – it is an expression made up of operators and operands that help specify what kind of pattern should be searched for and matched within a string. Applications of regular expressions include:

1. Checking whether a string conforms to a particular pattern or not (e.g. whether a string is a valid email id, valid integer, valid name, valid address, etc.)
2. Checking whether a string contains a part that matches a particular patterns (e.g. checking if a string contains coordinates, website URLs or email addresses)
3. Counting the number of occurrences of substrings that match a particular pattern (e.g. counting the number of operators present in an expression)
4. Locating the substrings that match a particular pattern within a string (e.g. locating keywords or identifiers within a computer program statement)
5. Substituting substrings that match a particular pattern with a replacement string (e.g. replacing all email addresses with the string 'EMAIL')
6. Substituting substrings that match a particular pattern with a replacement string that possibly reuses the content found in order to reorganise the string (e.g. converting dates in mm-dd-yy format to dd/mm/yy format)

What makes a regular expression based search different from the string search functions that we saw in section 8.2 is that in the case of regular expressions, we are not necessarily searching for a particular string, but are instead possibly searching for multiple strings that all follow a common pattern. If we are searching for “info@cyberplusit.com” in a string, we can use normal string functions. But if we are searching for all email addresses related to cyberplusit.com, or possibly any and every email address without knowing what email addresses to expect, we need regular expressions.

A regular expression is made up of operands and operators. Operands are merely characters that have to be found while operators are used to enforce rules pertaining to the occurrences of operands.

Before we start framing regular expressions for matching, keep these points in mind:

1. What matters currently is whether there is a match or not (Boolean).
2. It does not matter where the match is.
3. It does not matter what the match is.
4. It does not matter how many matches are there.

Of course, eventually we will learn to handle points 2-4 above. We will later learn how to:

1. Find out where the matches are present in the given string.
2. Find out what parts of the string matched.
3. Find out how many matches were there.

A good starting point is framing regular expressions that don't contain operators. Here is our first example of a regular expression (without any operators) – `not`:

Regular Expression: `not`	
Matches	***Rejects***
not	nt
can**not**	pot
nothing	nod
foot**not**e	night
not a **not**e	nest

As can be seen from these examples, the ones that are matched contain the characters `n`, `o` and `t`, exclusively in that order, with nothing in between them. What does not matter however, is where it was found within the string and how many times it was found.

We will now explore some basic operators that can be used in regular expressions.

NOTE:

We will use the abbreviation **RE** to denote **Regular Expression** from now on for brevity.

9.2 Basic Regular Expression Operators

The most basic operators used in regular expressions are `.`, `[]`, `*`, `+`, `?`, `{}`, `|`, `^`, `$` and `()`. Variants of the `()` operator will be seen in later sections.

9.2.1 Matching a Single Character

The simplest regular expressions with operators are those that employ operators for single characters.

9.2.1.1 Matching any Character Using the . Operator

The . operator matches any one character (except the newline character – a behaviour that can be changed as shown in section 10.4.4).

Thus, the RE `n.t` matches those strings that contain an `n`, followed by any one non-newline character, followed by `t`, as shown in the table below:

Regular Expression: `n.t`	
Matches	**Rejects**
not	nt
nut	night
native	spent
inter**net**	nest
agai**nst**	sat
k**not**ted	nun

9.2.1.2 Matching a Character Within a Set Using the [] Operator

The `[]` operator is similar to the . operator – it is also a placeholder for a single character, except that the character is restricted to be one amongst the characters listed within the `[]`.

Thus, the RE `n[aeiou]t` will match any string containing `n` that is followed by a lowercase vowel followed by `t`, as shown below:

Regular Expression: `n[aeiou]t`	
Matches	**Rejects**
not	nt
nut	against
net	nest
k**nit**	constant
native	Net

9.2.1.3 Matching a Character Within a Range of Characters Using the [-] Operator

To specify a range of characters within `[]`, we can use the `-` operator within `[]`.

Thus, `[a-e]` represents any one character out of `a`, `b`, `c`, `d` and `e`. `[4-6]` represents any one character out of `4`, `5` and `6`.

This can also be combined with the previous form. Thus, `[a-emx-z]` can match any of `a`, `b`, `c`, `d`, `e`, `m`, `x`, `y` and `z`.

The RE `n[a-e]t` can match any string that contains `n` followed by any one of `a`, `b`, `c`, `d` and `e` followed by `t`, as shown below:

Regular Expression: `n[a-e]t`	
Matches	**Rejects**
native	nt
net	nut
pu**nct**ual	against

9.2.1.4 Matching a Character Within a Negated Set Using the [^] Operator

Finally, the negated character set indicated by `[^]` implies a placeholder for a single character with the constraint that the character can be any one apart from the ones listed within `[]`.

Thus, while `[0-9]` represents a digit, `[^0-9]` represents a non-digit.

This can be combined with our understanding of sections 9.2.1.2 and 9.2.1.3 to create a regular expression like `[^a-emx-z]` which will match any single character that is either not a lowercase alphabet, or if it is, is only one of `f`, `g`, `h`, `i`, `j`, `k`, `l`, `n`, `o`, `p`, `q`, `r`, `s`, `t`, `u`, `v`, `w`.

Thus, the RE `n[^aeiou]t` matches any string that contains `n` followed by a character that is not a lowercase vowel, followed by `t`, as shown below:

Regular Expression: `n[^aeiou]t`	
Matches	**Rejects**
agai**nst**	not
pu**nct**ual	nt

9.2.1.5 Shortcut Escape Sequences for Characters

Certain escape sequences are understood to mean shortcuts for certain standard sets of characters as documented in the table below:

Escape Sequence	Meaning
`\d`	A Digit character; equivalent to [0-9] in ASCII
`\D`	A non-Digit character; equivalent to [^0-9] in ASCII
`\w`	A Word-forming character; equivalent to [a-zA-Z0-9_] in ASCII
`\W`	A non Word-forming character; equivalent to [^a-zA-Z0-9_] in ASCII
`\s`	A whitespace character; equivalent to [\t\n\r\f\v] in ASCII
`\S`	A non-whitespace character; equivalent to [^ \t\n\r\f\v] in ASCII

Table 20: Escape Sequences for Standard Regular Expression Character Classes

Thus, the RE `\w\D\d` will match any string that contains a word-forming character followed by a non-digit followed by a digit, as shown below:

Regular Expression: `\w\D\d`	
Matches	**Rejects**
Ab2	333
9e9	abc
cc3	c33

9.2.2 Matching Multiple Characters

Now that we've learnt some basic operators that represent placeholders for single characters, let's go on to learn operators that represent placeholders for multiple characters.

9.2.2.1 Matching 0 or More Occurrences using the * Operator

The `*` operator represents 0 or more occurrences of the preceding symbol. This also makes the preceding symbol optional (0 occurrences).

Thus, the RE `go*d` matches any string that contains `g` followed by 0 or more `o` followed by `d` as shown below:

Regular Expression: go*d	
Matches	***Rejects***
gd	g
god	go
good	gold
goooooood	old

9.2.2.2 Matching 1 or More Occurrences using the + Operator

The `+` operator represents 1 or more occurrences of the preceding symbol. Unlike the * operator, this makes the preceding symbol compulsory (at least 1 occurrence).

Thus, the RE `go+d` matches any string that contains `g` followed by 1 or more `o` followed by `d` as shown below:

Regular Expression: go*d	
Matches	***Rejects***
god	g
good	gd
goooooood	go

9.2.2.3 Matching 0 or 1 Occurrences using the ? Operator

The `?` operator represents 0 or 1 occurrence of the preceding symbol. In other words, this makes the preceding symbol optional.

Thus, the RE `goo?d` matches any string that contains `g` followed by `o`, followed by an optional `o`, followed by `d` as shown below:

Regular Expression: `go*d`	
Matches	**Rejects**
god	gd
good	goooooood

9.2.2.4 Matching Multiple Occurrences using the {} Operator

The `{m,n}` operator (where `m` and `n` are integers) represents minimum `m` occurrences and maximum `n` occurrences of the preceding symbol.

Thus, the RE `go{1,2}d` will match any string that contains `g` followed by at least 1 occurrence and at most 2 occurrences of `o`, followed by `d` as shown below:

Regular Expression: `go{1,2}d`	
Matches	**Rejects**
god	gd
good	goooooood

The form `{m,}` implies minimum `m` occurrences of the preceding symbol without any upper limit.

The `*`, `+` and `?` operators can be considered to be special shortforms of the `{}` operator as shown below:

Operator	Full form using {}
*	{0,}
+	{1,}
?	{0,1}

As a special case, the form `{m}` represents exactly `m` occurrences of the preceding symbol.

Thus, the RE `go{2}d` is exactly identical to the RE `good` – both can match strings that contain `g` followed by exactly 2 occurrences of `o`, followed by `d`.

Summary of the different forms of the `{}` operator:

Operator	*Meaning*
`{m,n}`	At least `m` occurrences and at most `n` occurrences
`{m,}`	At least `m` occurrences
`{m}`	Exactly `m` occurrences

Table 21: Summary of Different Forms of the {} RE Operator

9.2.2.5 Grouping Using the () Operator

You will notice during the discussion of the `*`, `+` and `?` operators that we used the term "preceding symbol" and not "preceding character". There is a reason for that. These operators need not work on individual characters, but can work on a group of characters – and for that matter, can work on a regular expression as well!

The `()` operator is used to create a group. Operators like `*`, `+` and `?` can then be applied on the group. Consider the RE `(a*b)+`. The `*` applies to `a`, implying 0 or more occurrences of `a`. The `+` applies on the group `(a*b)`, implying one or more occurrences of the entire group. This RE therefore means one or more occurrences of the pattern "zero of more occurrences of `a` followed by `b`". An example of a string that will be matched by this is: `abaaaabaabb`. Here is the split up of the match: `ab aaaab aab b`.

9.2.3 Matching Multiple Patterns Using the | Operator

The `|` operator represents an "OR" - a choice between patterns.

Thus, the RE `go|ld` can match any string that contains `g` followed by `o`, or `l` followed by `d`, as shown below:

Regular Expression: `go|ld`	
Matches	*Rejects*
go	greed
ld	odd
going	swallowed
bui**ld**er	gd
pan**go**	ol

When combined with the `()` operator, the choice is restricted within the parentheses. Thus, the RE `g(ol|ran)d` can match any string that contains `g` followed by either `ol` or `ran` followed by `d`, restricting the choice within the parentheses, as shown below:

Regular Expression: `g(ol|ran)d`	
Matches	**Rejects**
gold	greed
grand	golrand

9.2.4 Anchoring Operators

So far, we have emphasized that when using regular expressions, it does not matter where the match occurs. However, there are operators available to restrict the search at the beginning of a string, at the end of a string, or both.

9.2.4.1 Anchoring at the Beginning of a String

If a RE starts with a `^`, it implies that the match should occur at the beginning of the string only.

Thus, the RE `^not` will match any string that starts with `not` and will reject strings that do not start with `not`, as shown below:

Regular Expression: `^not`	
Matches	**Rejects**
not	cannot
nothing	net

NOTE:

The escape sequence `\A` at the beginning of a RE plays the same role as the `^` operator. Thus, the RE `^not` is identical to `\Anot`.

9.2.4.2 Anchoring at the End of a String Using the $ Operator

If a RE ends with a `$`, it implies that the match should occur at the end of the string only.

Thus, the RE `not$` will match any string that ends with `not` and will reject strings that do not end with `not`, as shown below:

Regular Expression: not$	
Matches	**Rejects**
not	nothing
can**not**	net

NOTE:

The escape sequence \Z at the end of a RE plays the same role as the $ operator. Thus, the RE not$ is identical to not\Z.

9.2.4.3 Anchoring at Word Boundaries

A word boundary is defined as the transition from a sequence of word-forming characters (the ones that match \w) to a sequence of non word-forming characters (the ones that match \W) or vice-versa. It also includes the special cases of the boundary between the start of a string and a word-forming character and the boundary between a word-forming character and the end of a string.

The word boundaries are highlighted in the example below:

```
"Take this 500 bucks, said he"
^   ^ ^  ^ ^ ^ ^   ^  ^  ^ ^^
```

The \b escape sequence matches a null string at a word boundary. It's purpose is to ensure a match at the right place. Thus, the RE \bnot will match those strings that contain not that starts at a word boundary, as shown below:

Regular Expression: \bnot	
Matches	**Rejects**
not	nothing
this is **not** bad	cannot
Whose **not**e is this?	This cannot happen

The RE `not\b` will match those strings that contain `not` that ends at a word boundary, as shown below:

Regular Expression: `not\b`	
Matches	**Rejects**
not	nothing
This is **not** bad	cannot
This can**not** happen	Whose note is this?

The RE `\bnot\b` will match those strings that contain `not` that starts at and ends at a word boundary – or in other words, strings that contain the *word* `not`, as shown below:

Regular Expression: `\bnot\b`	
Matches	**Rejects**
not	note this point
This is **not** bad	Tie this knot

9.2.4.4 Negated Anchoring at Word Boundaries

The `\B` escape sequence is the opposite of `\b` – it matches a null string anywhere except at word boundaries.

Thus, the RE `\Bnot` will match those strings that contain `not` that does not start at a word boundary, as shown below:

Regular Expression: `\Bnot`	
Matches	**Rejects**
This can**not** happen	not
Tie this k**not**	Whose note is this?

The RE `not\B` will match those strings that contain `not` that does not end at a word boundary, as shown below:

Regular Expression: `not\B`	
Matches	**Rejects**
note this point	This cannot happen
Did not **not**ice?	Tie this knot

The RE `\Bnot\B` will match those strings that contain `not` that neither starts nor ends at a word boundary, as shown below:

Regular Expression: `\Bnot\B`	
Matches	**Rejects**
Get me a k**not**ted rope	note this point
an**not**ate	Tie this knot

9.3 Framing Regular Expressions in Python

We have learnt some basic operators that help express regular expressions. In this section, we will see some practical examples of regular expressions, but before that, we will see two important points specific to Python:

1. The problems associated with RE strings containing backslashes and how to address them
2. Why raw string literals are better for representing a RE in Python

9.3.1 Escaping Within Regular Expressions

Some characters within a RE have special meanings. The "`+`" character, for instance, is an operator that denotes 1 or more occurrences of the preceding symbol. What if was wanted to search for the "`+`" character instead? In order to ensure that "special" characters are treated as "normal" characters, we need to *escape* the "special" characters by prefixing them with a backslash (`\`). Thus, while the RE `a+b` means "1 or more occurrences of `a` followed by `b`", the RE `a\+b` means "`a` followed by `+`, followed by `b`".

9.3.2 Dealing with Backslashes (\)

We have seen that certain escape sequences have been provided for creating regular expressions (like `\d` to represent a digit character). We have also learnt that the Python language itself understands certain escape sequences of it's own (like `\n` to represent the newline character). Thus, the Python interpreter is always ready to

process escape sequences within strings. The regular expression escape sequences are not known to the interpreter as such and could result in a problem if processed before the regular expression engine sees the RE string. The solution is to escape every backslash by prefixing each with a backslash. The table below shows some regular expressions and how they can be represented as strings in Python:

RE	*String Representation*
`\d`	`"\\d"`
`\d\d-\d\d-\d\d`	`"\\d\\d-\\d\\d-\\d\\d"`
`\\d`	`"\\\\d"`

Observation:

1. Basically, each '`\`' in the RE has to be represented as '`\\`' in a string to ensure that it remains as a backslash within the string.
2. The last example shows a RE that matches a backslash (`\`) followed by a `d`. As usual, each backslash has to be prefixed by a backslash and hence we have 4 backslashes within the string.

9.3.3 Using Raw Literals for RE

In the previous section, we have seen that backslashes within regular expressions need to be escaped by prefixing with another backslash. Python provides another alternative to this – raw strings.

A *raw string* is a string that is not further processed by Python – every character in it remains unchanged and without any special interpretation. In other words, escape sequences are not recognised. They are indicated by a prefix of '`r`' or '`R`' in front of the string literal as shown in the examples below:

```
>>> "Hello\n"
'Hello\n'
>>> r"Hello\n"
'Hello\\n'
>>> R"Hello\n"
'Hello\\n'
```

Observation:

1. Observe that the escape sequence '`\n`' is not recognised when the `r` or `R` prefix is used. Thus, '`\n`' becomes 2 characters: '`\`' and '`n`'. The '`\`' character is represented in Python as '`\\`'

NOTE:

We will use raw strings for all regular expressions henceforth as a professional practice.

9.3.4 Practical Examples of RE

Regular expressions might not be very friendly for first timers and very hard to debug! This section gives a few standard examples that can help you with common tasks and also probably help you understand how to frame them!

In all the examples that follow, we assume we are framing a RE to completely match the string and hence all the regular expressions start with `^` (to match from the beginning of the string) and end with `$` (to match till the end of the string). These can be removed if not required.

9.3.4.1 RE for Name

To start with, we can consider a name to be comprised of alphabets and can perhaps be tolerant to the case. The RE therefore would be as follows:

```
^[A-Za-z]+( [A-Za-z]+)*$
```

Observation:

1. The first part `[A-Za-z]+` is to ensure that the name starts with alphabets (and not a space)
2. The second part `( [A-Za-z ]+)*` allows for 0 or more occurrences of words (sequences of alphabets preceded by a space)

This RE can be made more robust to support the following:

1. Support for periods in the name (e.g. `M. K. Gandhi`)
2. Support for apostrophes in the name (e.g. `Francis D'Souza`)
3. Support for hyphens in the name (e.g. `Inzamam-ul-huq`)

Here is a more complex RE that handles all the above 3 enhancements as well:

```
^[A-Za-z]+([ .\-\'][A-Za-z]+)*$
```

Observation:

1. The first part `[A-Za-z]+` ensures that the name starts with alphabets
2. The part `[ .\-\']` is a placeholder for a word separator, which could be a space, a period, a hyphen or an apostrophe. The hyphen (`-`) has to be escaped as otherwise it will be wrongly construed to mean a range within the character set. The apostrophe (`'`) has to be escaped if the RE is enclosed within single quotes, else it will mean the end of the string. The period (`.`) should not be escaped though it is a RE operator as it is present within `[]`.
3. The following part `[A-Za-z]+` requires alphabets to be present after every such separator and prevents a name from ending with a stray separator.
4. Finally, the separator and the part following that is optional and can be present multiple times – thus the grouping and the `*`.

9.3.4.2 RE for Username

Normally, a username follows rules for identifiers. Let's say we have to adhere to the following rules:

1. The length should be 3-16 characters.
2. It can contain alphabets, digits and underscores but should not start with a digit.

Here is the RE for our requirement:

```
^[A-Za-z_]\w{2,15}$
```

Observation:

1. The first part `[A-Za-z_]` ensures that the username does not start with a digit.
2. The second part `\w{2,15}` permits any word forming character (alphabets, digits and underscores) with the constraint of a length between 2 and 15 characters. Together with the first character matched earlier, it ensures a length of 3-16 characters.

9.3.4.3 RE for Password

Let us say the requirements for a password are:

1. It should have a minimum length of 8 characters
2. It should support alphabets, digits and the following special characters: `!@#$%^&*()-+_=`

```
^[A-Za-z0-9!@#$%^&*()\-+_=]{8,}$
```

Observation:

1. The hyphen (`-`) within `[]` has to escaped to prevent it from being treated as an operator.

9.3.4.4 RE for Date of Birth

Let us say we want to support the following date formats:

1. dd/mm/yy and mm/dd/yy
2. dd/mm/yyyy and mm/dd/yyyy
3. dd-mm-yy and mm-dd-yy
4. dd-mm-yyyy and mm-dd-yyyy

For simplicity, let's not validate the date any further than the patterns above.

```
^(\d{2}-\d{2}-\d{2}(\d{2})?)|(\d{2}/\d{2}/\d{2}(\d{2})?)$
```

Observation:

1. We have 2 patterns – one that supports hyphens (`-`) as the separator and one that supports slashes (`/`) as the separator.
2. We provide a choice between the 2 patterns using the `|` operator. We do not prefer using `[\-/]` instead as the separator as this allows mixing the 2 separators within a single date!
3. Each digit is represented by `\d` and `{2}` ensures exactly 2 occurrences (we could have also written it as `\d\d` instead of `\d{2}`).
4. A 4-digit year is supported using an optional 2 digits after compulsory 2 digits: `\d{2}(\d{2})?`

9.3.4.5 RE for Phone

Let us say we wish to identify phone numbers with the following support:

1. An optional leading + sign to specify the country code
2. Digits can be grouped and separated by spaces and hyphens
3. 8-10 digits for the phone number and 1-4 digits for the country code, if specified

```
^(\+\d{1,4})?(\d[\- ]?){8,10}$
```

Observation:

1. The first part `(\+\d{1,4})?`, is for the country code. The `?` at the end makes it optional. The `+` is escaped to prevent it from treating it as the `+` operator of regular expressions. `\d{1,4}` allows 1-4 digits following the `+`.
2. The second part `(\d[\- ]?){8,10}` allows for 8-10 of additional digits, also permitting an optional hyphen (`-`) or space after each digit. The hyphen is escaped since it has a special meaning within the character class.

9.3.4.6 RE for Email

An email address is of the form:

```
username @ domainname . TLD
```

Here is the RE for matching an email address:

```
[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}
```

Observation:

1. The first part, `[a-z0-9._%+-]+`, permits 1 or more occurrences of the specified characters. This part matches the `username`. Of course, the characters can be tweaked as necessary.
2. The second part, `[a-z0-9.]-+`, permits 1 or more occurrences of the specified characters. We do not expect `%` and `+` to be part of `domainname`, unlike the `username`. This part matches the `domainname`. This part also takes care of subdomains like "`mail.abc`" or non-TLD's like the "`.co`" in "`.co.in`".
3. The third part, `[a-z]{2,}`, permits 2 or more characters to be part of the

TLD (like `.in` and `.com`).

4. As can be seen in the RE, these 3 parts are separated in the form of `username@domainname.TLD`. The `.` is escaped to prevent it from being considered to be a RE operator.

9.3.4.7 RE for IP Address

An IP (IPv4) address is made up of 4 integers, separated by dots, with each integer in the range of 0 to 255.

```
^(\d{1,2}|[01]\d{2}|2[0-4]\d|25[0-5]\.){3}(\d{1,2}|[01]\d{2}|2[0-4]\d|
25[0-5])$
```

Observation:

1. Each integer should be in the range of 0 to 255. This includes all single-digit numbers and all double-digit numbers. This justifies the RE `\d{1,2}`.
2. A 3-digit number that starts with 0 or 1 can be followed by any 2 digits. This justifies the RE `[01]\d{2}`.
3. A 3-digit number that starts with 2 should be confined to 255. If the second digit is in the range of 0 to 4, the third digit can be anything. This justifies the RE `2[0-4]\d`.
4. Finally, a 3-digit number that starts with 25 can only have the third digit in the range of 0 to 5. This justifies the RE `25[0-5]`.
5. Since each integer can by any of the above, we use the RE `\d{1,2}|[01]\d{2}|2[0-4]\d|25[0-5]`
6. The first 3 integers of the IP address will be followed by a dot (.), which needs to be escaped to prevent it from being treated as an operator. This justifies `(\d{1,2}|[01]\d{2}|2[0-4]\d|25[0-5]){3}`.
7. The last integers has the same pattern as the previous 3, except for the dot. This justifies the final part `(\d{1,2}|[01]\d{2}|2[0-4]\d|25[0-5])`.

9.3.4.8 RE for Integers

An integer is basically made up of digits, and thus the most fundamental RE for integers would be:

```
^\d+$
```

The above RE will match 1 or more occurrences of digits.

If we wish to also support an optional sign before the integer, the RE will become:

```
^[+-]?\d+$
```

Observation:

1. The sign can be `+` or `-`. The "`-`" is not escaped since it is the last character in the character set. When "`-`" is the first or last character in the character set, escaping is not required. In all other cases, escaping would be required. It might be a good idea for beginners to always escape "`-`" within character classes to be on the safer side. With experience, you could perhaps elect to drop escaping when not required in order to increase readability.
2. The sign is made optional using the `?` Operator.

If we wish to match a certain number of digits, like let's say a 3-digit integer, the RE could be:

```
^\d{3}$
```

Observation:

1. This RE will match an integer in the range of `000`-`999`. It will still not match integers like `25`.

If we wish to accept any integer *upto* 3 digits wide, for example, the RE can be:

```
^(\d?){2}\d$
```

Observation:

1. The `\d?` means an optional digit. The `{2}` means `2` such optional digits. The `\d` after that means a compulsory digit.
2. The RE therefore means: 2 optional digits followed by a compulsory digit. That compulsory digit matches the units place while the optional digits can match the hundreds place and tens place.

If we wish to accept an integer within arbitrary intervals, the RE becomes more complex. For example, here is a RE for accepting a 3-digit integer between `000` and `255`:

```
^([01]\d\d|2[0-4]\d|25[0-5])$
```

Observation:

1. The RE is made up of 3 choices, covered below.
2. The first choice is `[01]\d\d` which matches all 3-digit integers that start with `0` or `1`, matching `000` to `199`.
3. The second choice is `2[0-4]\d` which matches integers from `200` to `249`.
4. The third choice is `25[0-5]` which matches integers from `250` to `255`.

If we wish to match integers from 0 to 255, this is the RE:

```
^([01]?\d?\d|2[0-4]\d|25[0-5])$
```

Observation:

1. The RE is made up of 3 choices, covered below.
2. The first choice is `[01]?\d?\d`, which covers upto 2 optional digits followed by a compulsory digit, with additional constraints on the first optional digit that it should be `0` or `1`. This takes care of all single digit integers, all 2-digit integers and all 3-digit integers starting with `0` or `1`, hence matching `0`-`199` as well as `000`-`199`.
3. The second choice is `2[0-4]\d`, which matches all 3-digit integers starting with 2, and hence matches the range `200`-`249`.
4. The third choice is `25[0-5]`, which matches all 3-digit integers in the range of `250`-`255`.

9.3.4.9 RE for Real Numbers

Real numbers in programming can be represented using exponentiation notation or fixed/floating point notation. Here is a RE to match real numbers:

```
^[-+]?\d*\.?\d+([eE][-+]?\d+)?$
```

Observation:

1. The `[-+]?` takes care of an optional leading sign.
2. The `\d*\.?\d+` takes care of optional whole number part, an optional decimal point and a fractional part. It also takes care of situations where there is no fractional part – the `\d+` takes care of the integral part instead.
3. The `([eE][-+]?\d+)?` takes care of the exponent notation (`e` or `E`), followed by an optional sign, followed by an integral exponent part. This entire part is optional.

9.3.4.10 RE for Coordinates

A pair of coordinates are of the form `(x,y)` and will be matched by this RE:

```
\([-+]?\d+,[-+]?\d+\)
```

Observation:

1. The x- and y-coordinates are each matched using the same RE pattern. The "`(`", "`,`" and "`)`" are retained as characters. Since parentheses are operators, they need to be escaped. Since "`,`" is not an operator, it will not be escaped.
2. The pattern `[-+]\d+` supports an optional sign followed by an integer. This RE matches only integers, but we could also use the pattern we had used to match real numbers instead.

9.4 Searching Strings Using match(), fullmatch() and search()

Now that we know how to frame regular expressions in Python, let's put them to use! Support for regular expressions in Python is present in the `re` module, and hence this module has to be imported using the following statement:

```
import re
```

The first 3 functions we will examine are `match()`, `fullmatch()` and `search()`, all of which can search for a regular expression match within a string, but differ slightly in functionality as summarised in the table below:

Function	*Match Location*
`match()`	Matches from the beginning of the string
`fullmatch()`	Matches the entire string
`search()`	Matches anywhere and any part of the string

Table 22: Different Functions for Matching Regular Expressions in Strings

Thus, if the RE `not` is used to search within different strings, the table below summarises which functions will detect a match:

String	*match()*	*fullmatch()*	*search()*
not	✓	✓	✓
note	✓	×	✓
can**not**	×	×	✓
this, **not** that	×	×	✓

Note:

1. While `fullmatch()` is the most restrictive function, requiring a complete match, `search()` is the least restrictive function tolerating a match anywhere within the string.
2. The presence of a leading `^` or `\A` within the RE will make even the `search()` function behave like the `match()` function. The absence of these will not make any difference to the `match()` function.
3. The presence of a leading `^` or `\A` and a trailing `$` or `\Z` within the RE will make the `search()` and `match()` functions behave like the `fullmatch()` function. The absence of these will not make any difference to the `fullmatch()` function.
4. Anything matched by `fullmatch()` will definitely be matched by the `match()` and `search()` functions too.
5. Anything matched by `match()` will definitely be matched by the `search()` function too.

9.4.1 The search() Function

Complete syntax:

```
re.search(pattern,string[,flags])
```

Forms:

```
1.      re.search(pattern,string)
2.      re.search(pattern,string,flags)
```

Form #1: `re.search(pattern,string)`

In this form, the function checks whether the string `string` matches the regular expression `pattern` or not. If there is a match, the function returns a *match object* (covered in detail in section 10.6). If there is no match, the function returns `None`.

```
>>> import re
>>> re.search(r"\d","Here are 5 apples")
<_sre.SRE_Match object; span=(9, 10), match='5'>
>>> re.search(r"\d","Here are five apples")
>>>
```

NOTE:

The reason we are using a raw string (`r""`) for the regular expression has been covered in section 10.3.3.

We can use this function as though it returns a Boolean value as shown in the example below:

```
>>> import re
>>> if re.search(r"\d","5"): print("Yes")
... else: print("No")
...
Yes
```

Form #2: `re.search(pattern,string,flags)`

This form is identical in functionality to the previous form, but additionally allows us to provide *flags* that modify the search behaviour slightly. These flags are discussed in detail in section 10.4.4.

9.4.2 The fullmatch() Function

Complete syntax:

```
re.fullmatch(pattern,string[,flags])
```

Forms:

```
1.      re.fullmatch (pattern,string)
2.      re.fullmatch (pattern,string,flags)
```

Form #1: `re.fullmatch (pattern,string)`

In this form, the function checks whether the string `string` completely matches the regular expression `pattern` or not. If there is a match, the function returns a *match object* (covered in detail in section 10.6). If there is no match, the function returns `None`.

```
>>> import re
>>> re.fullmatch(r"\d+","1234")
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.fullmatch(r"\d+","12a34")
>>>
```

We can use this function as though it returns a Boolean value as shown in the example below:

```
>>> import re
>>> if re.fullmatch(r"\d+","1234"): print("Yes")
... else: print("No")
...
Yes
```

Form #2: `re.fullmatch(pattern,string,flags)`

This form is identical in functionality to the previous form, but additionally allows us to provide *flags* that modify the search behaviour slightly. These flags are discussed in detail in section 10.4.4.

9.4.3 The match() Function

Complete syntax:

```
re.match(pattern,string[,flags])
```

Forms:

```
1.      re.match (pattern,string)
2.      re.match (pattern,string,flags)
```

Form #1: `re.match (pattern,string)`

In this form, the function checks whether the string `string` starts with a match with the regular expression `pattern` or not. If there is a match, the function returns a *match object* (covered in detail in section 10.6). If there is no match, the function returns `None`.

```
>>> import re
>>> re.match(r"\d+","1234a")
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>> re.match(r"\d+","a1234")
>>>
```

We can use this function as though it returns a Boolean value as shown in the example below:

```
>>> import re
>>> if re.match(r"\d+","1234a"): print("Yes")
... else: print("No")
...
Yes
```

Form #2: `re.match(pattern,string,flags)`

This form is identical in functionality to the previous form, but additionally allows us to provide *flags* that modify the search behaviour slightly. These flags are discussed in detail in section 10.4.4.

9.4.4 Flags for Dealing with Regular Expressions

All the preceding 3 functions (`search`, `match` and `fullmatch`) and many other regular expression based functions that we are going to discuss later as well, permit us to specify an additional flags argument, which can be a combination of any of the flags discussed in this section. As a special case, a flag value of `0` indicates the

absence of any of these flags. Multiple flags can be combined by adding them using the + operator or by using the bitwise OR operator (|).

`re.I` / `re.IGNORECASE`

Regular expression matches are case-sensitive by default. This flag forces a case-insensitive match as demonstrated in the examples below:

```
>>> import re
>>> re.search("apple", "Here are some APPLES")
>>> re.search("apple", "Here are some APPLES", re.I)
<_sre.SRE_Match object; span=(14, 19), match='APPLE'>
```

Observation:

1. In the first example above, the RE "`apple`" was unable to match the substring "`APPLE`" due to the search being case-sensitive
2. In the second example, the same RE was able to match the same string due to case-insensitive comparison, specified by `re.I`

`re.M` / `re.MULTILINE`

It is typically expected when dealing with regular expressions that the string is a single line of text. The "`^`" and "`$`" operators therefore imply a match at the beginning of the string (and the line) and at the end of the string (and the line) respectively. If the string is a multi-line string that contains newline characters, these operators will be happily unaware of it. To force the "`^`" and "`$`" operators to anchor at the beginning and end of each line within the string (identified by the presence of the newline character), this flag can be specified.

```
>>> import re
>>> re.search("^This", "That is good.\nThis is bad.")
>>> re.search("^This", "That is good.\nThis is bad.", re.M)
<_sre.SRE_Match object; span=(14, 18), match='This'>
```

Observation:

1. In the first example, the RE "`^This`" is unable to match the string since the string does not start with "`This`".
2. In the second example, the same RE is able to match the same string since the string contains a *line* that starts with "`This`".

`re.S` / `re.DOTALL`

Section 10.2.1.1 introduced the fact that the "`.`" operator can match any one single

character. Since we assume that a string is a single line string, the "`.`" character does not match the newline character. When a multi-line string is provided, the "`.`" operator can match any character in that string except the newline character. To make the "`.`" operator match even the newline character, this flag can be used.

```
>>> import re
>>> re.search("e.T","One\nTwo")
>>> re.search("e.T","One\nTwo", re.S)
<_sre.SRE_Match object; span=(2, 5), match='e\nT'>
```

Observation:

1. In the first example, the "`.`" does not match the newline character between "`e`" and "`T`".
2. In the second example, the same RE is able to match the same string as the "`.`" can now match even the newline character.

There are a few more flags that can be used, but the flags discussed here are the most commonly used ones. The table below summarises all the flags:

Flag	*Meaning*
`re.A` `re.ASCII`	Ensure that the shorthand operators `\w`, `\W`, `\d`, `\D`, `\s`, `\S`, `\b` and `\B` apply only to ASCII characters in a Unicode string
`re.DEBUG`	Display debug information
`re.I` `re.IGNORECASE`	Case-insensitive comparison
`re.L` `re.LOCALE`	Ensure that `\w`, `\W`, `\s`, `\S`, `\b` and `\B` work based on the current locale
`re.M` `re.MULTILINE`	Ensure that "`^`" and "`$`" operators work at line boundaries in a multi-line string
`re.S` `re.DOTALL`	Ensure that "`.`" matches even newline characters
`re.X` `re.VERBOSE`	Support for extended or verbose regular expressions that permit 1-line comments within them using "`#`" as a comment indicator.

Table 23: Regular Expression Flags

To conclude this section, let us see how to combine flags together:

```
>>> import re
>>> re.search("E.T","One\nTwo")
>>> re.search("E.T","One\nTwo",re.S+re.I)
<_sre.SRE_Match object; span=(2, 5), match='e\nT'>
>>> re.search("E.T","One\nTwo",re.IGNORECASE|re.DOTALL)
<_sre.SRE_Match object; span=(2, 5), match='e\nT'>
```

Observation:

1. In the first example, there was no match as the "`.`" does not match the newline character and the comparison is case-sensitive.
2. In the second example, there is a match since the flags `re.S` and `re.I` take care of the 2 reasons why there wasn't a match earlier. The 2 flags were combined using the `+` operator.
3. The third example is identical to the second example, with us using the full name of the flag and combining them using the `|` operator.

9.5 Working With Groups

A group within a RE can be created using parentheses `()`. These are the reasons why we would use groups within an RE:

1. For **readability**. Grouping parts of a complex RE can make it easier for us to decipher which part of the RE is for matching what.
2. For **group operations**. Section 10.2.2.5 introduced this in detail, that quantifier operators can operate on entire groups rather than single symbols.
3. For **content extraction**. While matching regular expressions, as a secondary goal, we can also extract parts of the match. This is done using grouping. The contents of the string that matched each group can be extracted, in addition to extracting the entire match within a string. This section will show examples of what content can get matched, and section 10.6 will focus on the details on how to extract the match.
4. For **back-referencing**. It is possible to use the content found within the groups to be used within the same RE operation – for example, a substitution that uses values found within the same string. Again, this section will show examples of values of back-references, and section 10.6 will concentrate on the details.

9.5.1 Simple Group Content Extraction

Consider the following RE for matching a UK-format date:

```
\d\d-\d\d-\d{4}
```

For now, this RE can be used to search within strings for substrings that match this pattern. But using the concept of groups, it is also possible for us to extract the date, the month and the year in addition to playing the above role. The RE needs to be rewritten as follows to support group content extraction:

```
(\d\d)-(\d\d)-(\d{4})
```

The parentheses used here define 3 groups. When a match is obtained,

1. the content of the first group will give us the date found,
2. the content of the second group will give us the month found, and
3. the content of the third group will give us the year found.

Consider a sample string "`25-12-2016`". The content of the first group will be `25`, the content of the second group will be `12` and the content of the third group will be `2016`.

Section 10.6 introduces the actual functions to extract the group content.

9.5.2 Back-Reference Contents

Building on the same example and same sample string of the previous section, each group content can also be identified using back-references, whose syntax within a string is as follows:

```
\n
```

in the above syntax, `n` is the group number. Thus, `\1` refers to the content matched in the first group, `\2` refers to the content matched in the second group, and so on.

Given our sample string, `\1` will represent `25`, `\2` will represent `12` and `\3` will represent `2016`.

9.5.3 Nested Groups

Groups can also be nested, and while that might look complicated and messy, there

are rules that help restore sanity! Consider the following RE:

```
(((\d)(\d))-((\d)(\d))-(\d{4}))
```

The above RE will allow us to extract all the following:

1. The entire date that matched
2. The date, the month and the year
3. The individual digits of the date
4. The individual digits of the month

The only thing we need here is a rule for numbering, and that rule is as follows: the number of the group is assigned based on the number of it's open parentheses within the RE. Thus, the numbering is as follows:

```
(((\d)(\d))-((\d)(\d))-(\d{4}))
^^^   ^     ^^   ^     ^
123   4     56   7     8
```

For the sample string "`25-12-2016`", the content of each back-reference (and hence, group) will be as follows:

Back-reference	*Content*
`\1`	`25-12-2016`
`\2`	`25`
`\3`	`2`
`\4`	`5`
`\5`	`12`
`\6`	`1`
`\7`	`2`
`\8`	`2016`

9.6 Working with Matches

Functions like `search()`, `match()` and `fullmatch()` (covered in section 10.4) all check whether a given string matches a given RE and returns a `match` object if a match was found. This `match` object is implicitly convertible to Boolean `True`, making it convenient for using these functions in conditions, but this section focuses on other powerful capabilities of the `match` object.

The `match` object keeps track of the following:

1. The given string which was searched
2. The RE that was used to search within the string
3. The content of all the individual groups that got matched
4. Locations within the string of all the individual groups that got matched

9.6.1 Extracting the Original String and RE

Once a `match` object has been obtained, we can use it to extract the original string that was searched (`match.string`) and the RE that was used to search (`match.re`).

Syntax:

```
match.string
match.re
```

Example:

```
>>> import re
>>> m=re.fullmatch(r"(\d\d?)/(\d\d?)/(\d\d){1,2}", "25/12/2016")
>>> m.string
'25/12/2016'
>>> m.re
re.compile('(\\d\\d?)/(\\d\\d?)/(\\d\\d){1,2}')
```

Observation:

1. This example uses a RE to match a UK-format date. This time, we are storing the return value from `fullmatch` (which will be a `match` object) in a variable `m`.
2. `m.string` gives us the original string: '`25/12/2016`'
3. `m.re` gives us the original RE: `re.compile('(\\d\\d?)/(\\d\\d?)/(\\d\\d){1,2}')`

Of course, there is no significant advantage of extracted the string and RE from a

`match` object, but this is just a start. We would be more interested in extracting the contents of the groups that were found, and this is discussed in the next section,

9.6.2 Extracting the Group Contents

All the groups can be extracted as a tuple using the `match.groups()` method, which has this syntax:

```
match.groups(default=None)
```

The `match.groups()` method returns the contents of all the groups in the form of a tuple. If any group did not participate, it's content will be the `default` value provided (`None` by default).

Example:

```
>>> import re
>>> m=re.fullmatch(r"(\d\d?)/(\d\d?)/(\d\d){1,2}", "25/12/2016")
>>> m.groups()
('25', '12', '16')
```

Observation:

1. Each parenthesized group content is an item within the tuple.
2. The order of the items within the tuple matches the order of the groups in the RE.

The below example demonstrates possibilities when we have an empty group:

```
>>> import re
>>> m=re.fullmatch(r"(\d\d)/(\d\d)?/(\d\d\d\d)", "25//2016")
>>> m.groups()
('25', None, '2016')
>>> m.groups("-")
('25', '-', '2016')
```

Observation:

1. In this RE, the second group is made optional and the string does not provide any value for the same.
2. When `m.groups()` is used, the content for the second group is considered to be `None`.
3. When `m.groups("-")` is used, the content "`-`" is understood to be the

content of any group whose value that is omitted in the string.

The number of groups can be found out using `match.lastindex`, as shown in the example below:

```
>>> import re
>>> m=re.fullmatch(r"(\d\d)/(\d\d)?/(\d\d\d\d)", "25//2016")
>>> m.lastindex
3
```

Observation:

1. We have 3 sets of parentheses and therefore 3 groups.
2. Empty groups (groups without content) are also counted.

Individual group contents can be extracted using the `match.group()` method, the syntax being as follows:

```
match.group(groupNum=0,...)
```

The `match.group()` method returns the content of the specified group. If the group is not specified, it is assumed to be `0`. An argument of `0` (either implicitly or explicitly) is considered to mean the entire match that took place. If multiple arguments are provided, the return value is a tuple with as many values as the arguments passed, with corresponding content.

Examples:

```
>>> import re
>>> m=re.fullmatch(r"(\d\d)/(\d\d)?/(\d\d\d\d)", "25/12/2016")
>>> m.group(1)
'25'
>>> m.group(2)
'12'
>>>
>>> m.group(3)
'2016'
>>> m.group(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
>>> m.group(0)
'25/12/2016'
>>> m.group()
'25/12/2016'
>>> m.group(1,3)
('25', '2016')
```

```
>>> m.group(3,1)
('2016', '25')
```

Observation:

1. `m.group(1)`, `m.group(2)` and `m.group(3)` give us the content of groups `1`, `2` and `3` respectively.
2. `m.group(4)` raises an `IndexError` as the group is not found.
3. `m.group(0)` and `m.group()` are identical and give us the entire match, which in this case is the entire string.
4. `m.group(1,3)` gives us the contents of groups `1` and `3` respectively as elements within a tuple.
5. `m.group(3,1)` gives us the contents of groups `3` and `1` respectively as elements within a tuple. Note that this is different from `m.groups(1,3)` – the order of the groups can be specified.

9.6.3 Extracting the Group Locations

Each match group has a starting index and ending index within the string within which it was found, and these can be extracted using the following methods:

```
match.start(group=0)
match.end(group=0)
match.span(group=0)
```

The `match.start()` method returns the start index of the specified group in the string.

The `match.end()` method returns the end index of the specified group in the string.

The `match.span()` method returns a tuple containing the start and end indices of the specified group in the string.

In all the 3 methods, if the group is not specified, it is implied to be `0`. A group argument of `0` means the entire match.

Examples:

```
>>> import re
>>> m=re.fullmatch(r"(\d\d)/(\d\d)/(\d\d\d\d)", "25/12/2016")
>>> m.groups()
('25', '12', '2016')
>>> m.start(1)
0
>>> m.end(1)
2
>>> m.span(1)
(0, 2)
>>> m.start(2)
3
>>> m.end(2)
5
>>> m.span(2)
(3, 5)
>>> m.start(3)
6
>>> m.end(3)
10
>>> m.span(3)
(6, 10)
>>> m.start()
0
>>> m.end()
10
>>> m.span()
(0, 10)
>>> m.start(0)
0
>>> m.end(0)
10
>>> m.span(0)
(0, 10)
```

Observation:

1. As can be verified, the first group lies between indices `0` and `2`. The second group lies between indices `3` and `5`. The third group lies between indices `6` and `10`.
2. The entire string is the match and therefore the entire match is between the indices `0` and `10`.

9.6.4 Expanding a Template String

Given that a match object is aware of the contents of all the groups, it can easily allow us to expand a template with values taken from the group contents. The `match.expand()` method allows us to pass a template string which can contain references to groups, whose values will be substituted and the new string will be returned. The group contents can be identified using the syntax `\groupNum` as will be illustrated in the examples.

Syntax:

```
match.expand(template)
```

Examples:

```
>>> import re
>>> m=re.fullmatch(r"(\d\d)/(\d\d)/(\d\d\d\d)", "25/12/2016")
>>> m.expand(r"\2-\1-\3")
'12-25-2016'
>>> m.expand(r"US format: \2-\1-\3")
'US format: 12-25-2016'
```

Observation:

1. The statement `m.expand(r"\2-\1-\3")` is an attempt to convert a UK-format date to a US-format date. The RE was used to process a UK-format date (`dd/mm/yyyy`), which we now wish to convert to a US-format date (`mm-dd-yyyy`).
2. `\1`, `\2` and `\3` represent the contents of groups `1`, `2` and `3` respectively.
3. Any other characters apart from group content substitution requests are retained verbatim.

9.7 Locating Matches Using findall() and finditer()

The `re.findall()` function returns all the content that matched the given RE in the form of a list. Since the behaviour depends on the presence of groups, we will cover each case separately.

Syntax:

```
re.findall(pattern, string, flags=0)
```

Behaviour #1: If the RE does not contain any groups, this function returns a list of all contents in the string `string` that matched the pattern `pattern`:

```
>>> import re
>>> re.findall(r"\d+", "4 four 25 twenty five 12 twelve")
['4', '25', '12']
```

Observation:

1. The RE will match 1 or more occurrences of digits.
2. There are 3 such matches within the given string.
3. The function returns a list of the 3 matches found.

Behaviour #2: If the RE contains 1 group, this function returns a list of all contents in the string `string` that matched the group in the RE `pattern`:

```
>>> import re
>>> re.findall(r"(\d\d)", "4 four 25 twenty five 12 twelve")
['25', '12']
```

Observation:

1. Our RE contains 1 group: (\d\d).
2. There are 2 matches for the group: 25 and 12
3. The function returns a list of the content obtained that matched the groups.

Another example:

```
>>> import re
>>> re.findall(r"\d(\d)", "4 four 25 twenty five 12 twelve")
['5', '2']
```

Observation:

1. The RE matches 2 consecutive digits, but the second digit is a member of a group.
2. The list returned contains values that matched the group, not the entire RE.

Behaviour #3: If the RE contains more than 1 group, this function returns a list of tuples – each tuple for 1 match containing the content that matched each group:

```
>>> import re
>>> re.findall(r"(\d)(\d)", "4 four 25 twenty five 12 twelve")
[('2', '5'), ('1', '2')]
```

Observation:

1. The RE contains 2 groups.
2. Each element of the list returned contains a tuples with 2 values.
3. The list contains as many elements as the number of matches.
4. Each tuple contains as many values as the number of groups.

The `finditer()` function is a cross between `findall()` discussed in this section and `match` objects dealt with in the previous section. This function gives an iterator using which we can loop through the matches found.

Syntax:

```
re.finditer(pattern, string, flags=0)
```

Example:

```
>>> import re
>>> for m in re.finditer(r"(\d)(\d)", "4 four 25 twenty five 12
twelve"):
...     print(m.groups())
...
('2', '5')
('1', '2')
```

Observation:

1. Since `finditer()` returns in iterator, we can directly iterate through the result using a loop construct as shown.
2. Each iteration provides a single `match` object corresponding to a match found. Methods of `match` instance were dealt with in section 10.6.

9.8 Splitting Strings Using split()

A string can be split into pieces based on delimiters specified using a RE using the `re.split()` function:

```
re.split(pattern, string, parts=0, flags=0)
```

Forms:

1. `re.split(pattern, string)`
2. `re.split(pattern, string, parts)`
3. `re.split(pattern, string, parts, flags)`

Form #1: `re.split(pattern, string)`

In this form, the contents of the string `string` is split into pieces wherever it finds a substring that matches the RE `pattern` and the split parts are returned:

```
>>> import re
>>> re.split(r"\d\d", "4 four 25 twenty five 12 twelve")
['4 four ', ' twenty five ', ' twelve']
```

Observation:

1. The RE for the split specifies that splitting should occur wherever 2 consecutive digits are present - "`25`" and "`12`" in our example.
2. After splitting, there are 3 parts which are returned as a list of strings.

If the RE `pattern` contains any groups, then the contents of the groups are also returned in-between the split parts:

```
>>> import re
>>> re.split(r"(\d)(\d)", "4 four 25 twenty five 12 twelve")
['4 four ', '2', '5', ' twenty five ', '1', '2', ' twelve']
```

Form #2: `re.split(pattern, string, parts)`

In this form, we can specify the number of split parts we want. The splitting will stop after that and the remainder string will also be returned:

```
>>> import re
>>> re.split(r"\d\d", "4 four 25 twenty five 12 twelve", 1)
['4 four ', ' twenty five 12 twelve']
```

Observation:

1. We have asked for 1 split and therefore the splitting process stops after 1 split.
2. The remainder string after splitting is also returned. Thus, if we ask for `n` splits, we will get `n` split strings and the remainder strings – totally a list of `n+1` strings.

If the value of `parts` is `0`, this works the same as form #1.

Form #3: `re.split(pattern, string, parts, flags)`

This is identical to form #2, except that we can use flags to control the RE match behaviour as covered in section 10.4.4. A `flags` value of `0` makes this exactly identical to form #2.

9.9 Replacing Strings Using sub() and subn()

Now that we know how to use regular expression functions to search within a string, it is time to focus on search-replace. This section focusses on how to search for a RE match within a string and replace the match by a string. The basic function that achieves this is `re.sub()`:

```
re.sub(pattern, replacement/function, string, count=0,
flags=0)
```

Forms:

1. `re.sub(pattern, replacement, string)`
2. `re.sub(pattern, replacement , string, count)`
3. `re.sub(pattern, replacement , string, count, flags)`
4. `re.sub(pattern, function, string)`
5. `re.sub(pattern, function, string, count)`
6. `re.sub(pattern, function, string, count, flags)`

Form #1: `re.sub(pattern, replacement, string)`

Replaces all occurrences of the RE `pattern` within the string `string` by the string `replacement` and returns the new string:

```
>>> import re
>>> re.sub(r"\d+","'n'","There are 5 apples and 3 mangoes")
"There are 'n' apples and 'n' mangoes"
```

Observation:

1. The original string argument, `string`, is not changed by this function. The new string with replacements made is merely returned.

We can also use back-references (`\1`, `\2`, etc.) to use content that matched in the RE within the replacement string:

```
>>> import re
>>> re.sub(r"(\d+)/(\d+)/(\d{4})", r"\2-\1-\3", "25/12/2016")
'12-25-2016'
```

Observation:

1. We are attempting to match a UK-format date and convert it into a US-format equivalent.
2. Since the replacement string contains backslashes, either we should prefix each backslash with another backslash or use raw strings – we prefer the latter for simplicity and readability.
3. The back-reference `\1` refers to the content that matched the first group, `\2` for content that matched the second group and so on.

Form #2: `re.sub(pattern, replacement, string, count)`

In this form, we can specify the maximum number of replacements we wish within the string. If `count` is `0`, this reverts to form #1 and replaces all occurrences of the RE `pattern` in the string `string`.

```
>>> import re
>>> re.sub(r"\d+","'n'","There are 5 apples and 3 mangoes", 1)
"There are 'n' apples and 3 mangoes"
>>> re.sub(r"\d+","'n'","There are 5 apples and 3 mangoes", 5)
"There are 'n' apples and 'n' mangoes"
>>> re.sub(r"\d+","'n'","There are 5 apples and 3 mangoes", 0)
"There are 'n' apples and 'n' mangoes"
```

Observation:

1. In the first example, only the first occurrence of a sequence of digits was replaced.
2. In the second example, while we have asked for maximum `5` replacements, there were only 2 occurrences of the pattern and both were replaced.
3. The third example is identical to the example we saw for form #1.

Form #3: `re.sub(pattern, replacement, string, count, flags)`

This form is identical to form #2 and also allows us to pass various `flags`, as discussed in section 10.4.4.

Form #4: `re.sub(pattern, function, string)`

This form is similar to form #1 in that this is used to perform a replace-all task, with the difference being that instead of using a fixed string for replacement, this calls the specified function passing a `match` object corresponding to each match obtained and uses the return value from the function as a replacement string:

```
>>> import re
>>> def f(m):
...     if int(m.string[m.start():m.end()])<10: return "few"
...     else: return "many"
...
>>> re.sub(r"\d+",f,"There are 15 apples and 3 mangoes")
'There are many apples and few mangoes'
```

Observation:

1. Note that when we are invoking the `re.sub()` function, we specify `f` and not `f()` - in other words, we are providing the function name and not calling the function ourselves.
2. The function `f` will receive a `match` object `m` each time a match is found. Recall from section 10.6 that `m.string` gives us the original string that was searched, `m.start()` gives us the starting index within the string where the match was found and `m.end()` gives is the ending index within the string where the match was found. Thus, `m.string[m.begin():m.end()]` identifies the substring that was matched within the original string.
3. The function `f` converts the matched substring into an integer and compares it with `10` – if less than `10`, the function returns the string `"few"`, else it returns the string `"many"`.

Form #5: `re.sub(pattern, function, string, count)`

This form of the function replaces upto `count` number of occurrences of the RE `pattern` within the string `string`, calling the function `function` each time a match is found, passing a `match` object to the function and using the function's return value as the replacement string:

```
>>> import re
>>> def f(m):
...     if int(m.string[m.start():m.end()])<10: return "few"
...     else: return "many"
...
>>> re.sub(r"\d+",f,"There are 15 apples and 3 mangoes", 1)
'There are many apples and 3 mangoes'
```

Observation:

1. We have asked for maximum 1 replacement and hence the number "`3`" remains in the original string.
2. This form is similar to form #2, but uses a replacement function instead of a replacement string.
3. As discussed in form #2, if the `count` is `0`, this will work the same way as form #4 and replaces all occurrences of the pattern `pattern` in the string `string`.

Form #6: `re.sub(pattern, function, string, count, flags)`

This form is identical to form #5, but allows us to specify `flags` that control how the RE pattern should be treated, as discussed in section 10.4.4.

The `re.subn()` is a very similar function that works the same way as `re.sub()` does, but returns a tuple containing the replaced string as well as the total number of substitutions performed:

```
re.subn(pattern, replacement/function, string, count=0,
flags=0)
```

Example:

```
>>> import re
>>> re.subn(r"\d+","'n'","There are 5 apples and 3 mangoes")
("There are 'n' apples and 'n' mangoes", 2)
```

9.10 Compiling Regular Expressions

If a particular RE is used frequently in the program, it is possible to optimize it's use by "compiling" the regular expression and using the compiled regular expression object – technically called a regex object - to perform operations instead. Here are some points to remember about regex objects:

1. We can perform almost identical operations using the regex object instead of using the `re` library's functions.
2. Performing operations on regex objects are faster than working with `re` functions using RE strings.
3. Python does compile regular expressions automatically and maintains a cache of recently used regular expressions. Thus, if we are using just a few regular expressions in our program, we need not worry about compiling them ourselves.
4. In cases where we deal with many regular expressions in our program, we can compile the regular expressions ourselves and use the regex objects for better efficiency. At least the most frequently used regular expressions can be compiled.

We use the `compile()` function of `re` to compile a RE and obtain a reference to the regex object:

```
re.compile(pattern, flags=0)
```

Forms:

1. `re.compile(pattern)`
2. `re.compile(pattern, flags)`

Form #1: `re.compile(pattern)`

In this form, the given RE pattern is compiled and a reference to the regex object is returned:

```
>>> import re
>>> c=re.compile(r"\d\d")
```

Form #2: `re.compile(pattern, flags)`

This form is identical to form #1, but also allows us to specify flags (discussed in section 10.4.4).

Once we have access to a regex object, these are the methods we can invoke on it:

```
regexObject.search(string, pos=0, endpos=-1)
regexObject.match(string, pos=0, endpos=-1)
regexObject.fullmatch(string, pos=0, endpos=-1)
regexObject.split(string, parts=0)
regexObject.findall(string, pos=0, endpos=-1)
regexObject.finditer(string, pos=0, endpos=-1)
regexObject.sub(replacement, string, count=0)
regexObject.subn(replacement, string, count=0)
```

Observation:

1. These methods behave identical to the corresponding functions in the `re` module.
2. Most of these methods receive optional `pos` and `endpos` parameters – the `pos` parameter specifies where to start matching within the string, and the `endpos` parameter specifies where to stop matching in the string. When `pos` is omitted, the matching starts at the beginning of the string and when `endpos` is omitted, the matching ends at the end of the string. Note that this has no effect on the "`^`" and "`$`" operators which will still consider the start of the string to the be index `0` regardless of `pos` and the end of the string to be the last index of the string regardless of `endpos`.
3. The RE is not present in these methods since it is already placed within the regex object. The string against which the RE has to be matched can change and is hence passed as a parameter.
4. The flags are not present as parameters in these methods since they are already placed within the regex object. If a different set of flags are required, another regex object instance has to be created for that using `re.compile()`.

9.11 Questions

1. Explain regular expression operators with examples.
2. Write a short note on anchoring operators of regular expressions.
3. Differentiate between the `match()`, `fullmatch()` and `search()` functions of the `re` module.
4. Write a short note on flags that can be used while dealing with regular expressions.
5. Briefly explain how we can work with `match` objects in Python.

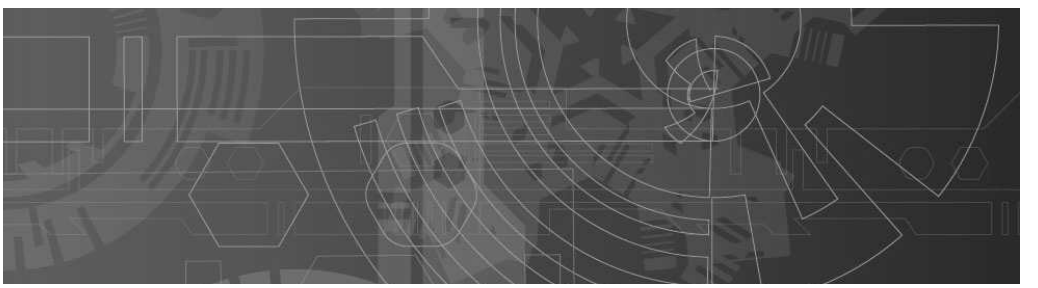
6. Write a short note on back-references in the context of regular expressions.
7. Explain the `findall()` and `finditer()` functions of the `re` module with examples.
8. Explain the `split()` function of the `re` module in detail with examples.
9. Write a short note on the `sub()` and `subn()` functions of the `re` module.
10. What is the benefit of “compiling” regular expressions? When are they useful?

9.12 Exercises

1. Write a Python script to check if a given string contains an email address or not.
2. Write a Python script that accepts a line of CSV (Comma Separated Values) and checks if the 3rd field contains exactly an email address or not, given that there are more than 3 fields.
3. Write a Python script that scans through a given piece of text and extracts all unique email addresses from it.
4. Write a Python script that reads in a piece of text and prints it out masking out email addresses. Thus, and email address “`helloworld@python.com`” should become “`h********d@python.com`”.

SUMMARY

- Regular Expressions are pattern matching expressions. They help us formally frame patterns of strings we are interested in searching.
- Certain operators like ., [], *, +, ? and {} are available that have special meanings. If these characters are to be used as operands, they need to be escaped.
- Regular expressions are case sensitive by default.
- Regular expression based search and replace functionality is provided by the re module.
- The search function searches for a pattern anywhere within a string; the match function searches from the beginning of a string; the fullmatch function searches from a match that spans the entire string.
- Parentheses can be used for grouping and substrings that matched within the groups can be retrieved.
- The split() function can be used to split a string into parts based on a RE for specifying the delimiter string.
- The sub() and subn() functions can be used for performing regular expression based substitutions.
- Regular expressions that are frequently used can be pre-compiled using the compile() function.



10 Functions

In this chapter you will be able to:

- ☑ Define Functions and call them.
- ☑ Pass and Receive Positional Arguments, Default Arguments, Keyword Arguments and Variable Arguments.
- ☑ Return single values as well as collections from functions.
- ☑ Define and use nested functions and lambda expressions.
- ☑ Unpack arguments from collections and pass them as function arguments.

FUNCTIONS

10.1 Introduction to Functions

Functions (loosely equivalent to *subroutines*/*procedures*, all of which are types of *subprograms*) are self-contained blocks of code to perform a specific task. Functions are used for the following reasons:

1. **They help reduce code redundancy:** when the same piece of code is required in multiple places within a program, the piece of code can be converted into a function, which can then be called whenever and wherever required within the program.
2. **They help increase code reusability:** a function once defined can be called any number of times and can even be called from other programs (see section 15 for a discussion of modules).
3. **They help improve program clarity:** putting all instructions in one place results in cluttering, making the program difficult to read, understand and maintain; whereas organising them into functions makes it far more manageable.

Functions, being subprograms, can assist the main program in it's activities. During execution of a Python script, control always starts flowing from the first line of the main program (the part that is outside of all functions) and flows down sequentially line by line by default, executing all the lines in a sequence. This flow can be altered by certain programming constructs like decisions and loops, and is also altered when function calls are made. Thus, a function defined in a program will not be executed unless explicitly called from the main program. Of course, a function that is called can then call other functions as well!

10.2 Function Definition

Before a function can be called, it needs to be defined. The *function definition* contains the body of the function – the instructions that constitute what will happen when the function is called. A function definition has the following syntax:

```
def function_name(parameters):
    function_body
    ...
```

The `def` keyword specifies that we are defining a function. The function name should be a valid identifier. A function can optionally receive parameters, different forms of which will be covered in the next few sections. The function body is a sequence of instructions and can span any number of lines (including blank lines). As is the case

everywhere in Python, the indentation level decides which statements belong to this function.

Here is the definition of a function called `hi` that prints "`Hello World`" when called:

```
>>> def hi():
...     print("Hello World")
...
```

10.3 Function Call

Once a function has been defined, it can be called as many times as required, from almost any place thereafter in the script and with any arguments as required.

A *function call* is identified by the parentheses `()` that immediately follow the function name (which as we know is an identifier).

Thus, the function `hi` defined earlier can be called in the following manner:

```
>>> hi()
Hello World
```

NOTE:

In an interactive Python session, once a function has been defined, it can be called any number of times without requiring a redefinition.

Functions once defined can be redefined later to perform a different task. The latest definition seen will be used when a function call is executed:

```
>>> def hi():
...     print("Hello World")
...
>>> hi()
Hello World
>>> def hi():
...     print("Heya!")
...
>>> hi()
Heya!
```

A function name without the parentheses is a *reference* to the function and as such can be stored in variables and used later as a replacement for the original function name:

```
>>> def hi():
...     print("Hello World")
...
>>> greeting=hi
>>> greeting()
Hello World
```

10.4 Positional Arguments

While a function is a self-contained piece of code, it does interact with the calling routine in 2 ways:

1. It can receive parameters from the calling routine
2. It can return a value back to the calling routine

This section concentrates on passing arguments to functions, which become parameters in the function while the sections 9.9 and 9.10 talk about returning to the caller. The following sections talk about special ways of passing arguments to functions.

A function can not only perform a specific task it was meant to do, it can also receive inputs from the caller in order to perform the task. The values that are sent to a function as part of the function call are called *arguments*, and when they are received by the function in it's definition, they are called *parameters*. There is a one-to-one correspondence between them in normal cases, and special cases will be dealt with in the sections that follow. Such arguments are called *positional arguments* in Python as their position is important for linking arguments with parameters.

We will now define the function `sum` and call it to print the sum of `2` and `3`:

```
>>> def sum(x,y):
...     print(x+y)
...
>>> sum(2,3)
5
```

Observation:

1. The values sent to the function at the place of call (`2` and `3`) are *arguments* to the function.
2. The variables in which values are received by the function in it's definition (`x` and `y`) are *parameters*.
3. At the time of function call, the argument `2` is sent to the parameter `x` and the argument `3` is sent to the parameter `y`.

10.5 Default Arguments

Ideally, the number of parameters in the function definition and the number of arguments in the function call would be the same. But Python supports many other kinds of arguments that disturb this equation but adds a lot of flexibility! The first such special kind of argument is default argument.

Default argument values are values that are assumed to be present if not provided explicitly. This makes the argument optional at the place of call, but the parameter compulsory. If the programmer is happy with the default value for the parameter, the programmer saves the effort of having to provide it manually at the place of call. On the other hand, if the programmer is not happy with the default value, it can be overridden by providing it explicitly like any other argument.

Here is an example of a function that receives 4 parameters – 2 of them are compulsory, like the previous example, while 2 of them are optional with default values:

```
>>> def f(a,b,c=2,d=3):
...     print(a,b,c,d)
...
>>> f(10,20,30,40)
10 20 30 40
>>> f(10,20,30)
10 20 30 3
>>> f(10,20)
10 20 2 3
>>> f(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required positional argument: 'b'
```

Observation:

1. The syntax of default arguments is that a value is assigned to the parameter. This is the default value which will be used when the actual argument is missing in the call.
2. A positional parameter cannot follow a default argument. Hence default arguments are always at the end of the parameter list.
3. The default value need not be used and default arguments can be treated as normal arguments as shown in the example `f(10,20,30,40)`.
4. When default arguments are omitted at the place of call, the default value is assumed - like how `3` is assumed to be the value of `d` in `f(10,20,30)`.
5. When multiple default arguments are present and some are omitted, it is assumed that the omission is from right to left. Thus, in the call `f(10,20,30)`, it is assumed that `d` is omitted and the value of `c` is `30`. Practically, we would put the default argument that is most omitted at the

rightmost end of the parameter list. (Section 10.6.2 shows how we can explicitly override a default argument without providing any value for previous default arguments).

6. The positional arguments cannot be omitted, as shown in `f(10)`.

10.6 Keyword Arguments

Keyword arguments are special arguments where the parameter name is identified at the place of call along with the value. These have various applications as shown in the following subsections.

10.6.1 Keyword Arguments for Positional Arguments

Positional Arguments are values passed for parameters on a one-to-one basis. The use of keyword arguments provides the flexibility of providing arguments in any order the programmer desires. Thus, there is no need to remember or follow the order of parameters in the function definition.

```
>>> def si(p,t,r):
...     print("Interest:",(p*t*r)/100)
...
>>> si(1000,3,9.5)
Interest: 285.0
>>> si(p=1000,r=9.5,t=3)
Interest: 285.0
>>> si(1000,r=9.5,t=3)
Interest: 285.0
```

Observation:

1. In the above example, the `si()` function is designed to receive arguments for principal, time duration and rate of interest – in that order.
2. The example `si(1000,3,9.5)` passes `1000` as `p`, `3` as `t` and `9.5` as `r`.
3. The example `si(p=1000,r=9.5,t=3)` passes `1000` as `p`, `3` as `t` and `9.5` as `r`. Note that the order of arguments given here do not match the order of parameters, but since we have explicitly identified the parameter by it's name, there is no confusion.
4. We can also mix positional parameters with keyword arguments as shown in the example `si(1000,r=9.5,t=3)`.

NOTE:

A keyword argument cannot be followed by a non-keyword argument.

10.6.2 Keyword Arguments for Default Arguments

One of the problems in using default arguments is that the order has to be retained. Thus, given 2 default consecutive arguments, it is not possible to skip the first one while providing an explicit value for the second one. But this becomes possible now using keyword arguments as shown below:

```
>>> def f(a,b,c=2,d=3):
...     print(a,b,c,d)
...
>>> f(10,20,30)
10 20 30 3
>>> f(10,20,d=30)
10 20 2 30
```

Observation:

1. In the first example, `f(10,20,30)`, `30` is assumed to be the value of `c` and `d` is assumed to have been omitted and thus assumed to be the default value `3`.
2. In the second example, `f(10,20,d=30)`, the keyword argument `d=30` explicitly states that the value of `d` should be `30`. The value of `c` is assumed to have been omitted and thus assumed to be the default value `2`.

NOTE:

A keyword argument cannot be followed by a non-keyword argument.

10.6.3 Keyword Arguments for Additional Arguments

When keyword arguments are to replace positional parameters or default arguments, as shown in the previous sections, it is an offence to provide a keyword argument for a parameter that does not exist in the parameter list, as shown in the example below:

```
>>> def f(a,b,c=2,d=3):
...     print(a,b,c,d)
...
>>> f(1,2,m=9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got an unexpected keyword argument 'm'
```

It is however possible to invent additional keyword arguments that are not even present in the formal parameter list! These additional keyword arguments can then be received as a dictionary, the keys of which are parameter names and values are argument values. The special syntax `**` prefix is used for this purpose as shown in the example below:

```
>>> def f(a,b,**x):
...     print(a,b)
...     for k,v in x.items():
...         print("The value of {} is {}".format(k,v))
...
>>> f(1,2,c=2,d=3)
1 2
The value of d is 3
The value of c is 2
>>> f(1,c=2,b=5,d=3)
1 5
The value of d is 3
The value of c is 2
```

Observation:

1. The function `f` receives 2 positional parameters `a` and `b` and any number of keyword arguments.
2. In the call `f(1,2,c=2,d=3)`, the values `1` and `2` are passed to the positional parameters `a` and `b` respectively; whereas 2 keyword arguments `c` and `d` are created with values `2` and `3` respectively.
3. The additional keyword arguments are stored in a dictionary and passed as the value for the dictionary parameter `x`. Recall from section 7.1 that the order of the keys in the dictionary (in this case, the keys are the parameter names) are not under our control.
4. The dictionary contains those parameters which are not present in the parameter list. Thus, in the example `f(1,c=2,b=5,d=3)`, the parameter `b` will not be found in the dictionary.

NOTE:

A keyword argument cannot be followed by a non-keyword argument.

10.7 Variable Arguments

A function in Python can be designed to receive any number of arguments. In other words, the number of arguments can vary from call to call. For example, consider a function `sum` that is supposed to add multiple values. We do not know how many values will be provided – in some calls there may be only 2 values, but some other calls may provide 5 values for example. Here is an example of how we could write the function `sum` using the concept of variable arguments:

```
>>> def sum(*x):
...     s=0
...     for i in x: s+=i
...     print(s)
...
>>> sum(2,3)
5
>>> sum(2,3,5,5,5,5)
25
>>> sum()
0
```

Observation:

1. The special `*` prefix is used to denote that we are dealing with a variable arguments.
2. These variable arguments are packed together into a tuple that is accessible as a single parameters in the function – in our example, `x`.
3. A simple `for` loop can help iterate through all the arguments passed, in order.

10.7.1 Variable Arguments With Positional Parameters

It is possible to combine positional parameters with variable arguments. These are helpful in cases where we expect a known number of values followed by an optional sequence of additional values. Let us redesign the `sum` function to add at least 2 values when called:

```
>>> def sum(a,b,*x):
...     s=a+b
...     for i in x: s+=i
...     print(s)
...
>>> sum(2,3)
5
>>> sum(2,3,5,5,5,5)
25
>>> sum()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() missing 2 required positional arguments: 'a' and 'b'
```

Observation:

1. In this function, `a` and `b` are positional parameters. It is mandatory to provide arguments for these or else we will get an error as demonstrated in the example `sum()`.
2. The variable `s` is initialized to the sum of `a` and `b` instead of `0` as was the case in the previous example.

The `*` parameter (`x` in our previous example) is greedy and will take up all remaining arguments. Therefore, we will never provide positional parameters after variable arguments, as demonstrated by this example:

```
>>> def sum(*x,a,b):
...     s=a+b
...     for i in x: s+=i
...     print(s)
...
>>> sum(2,3,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() missing 2 required keyword-only arguments: 'a' and
'b'
```

The problem can be solved however, using keyword arguments for the positional parameters as shown below:

```
>>> sum(2,a=3,b=4)
9
```

NOTE:

A keyword argument cannot be followed by a non-keyword argument.

10.7.2 Variable Arguments With Default Arguments

Default arguments are optional and variable arguments are greedy! Here is an example to show how the combination behaves:

```
>>> def sum(a,b=100,*x):
...     s=a+b
...     for i in x: s+=i
...     print(s)
...
>>> sum(5)
105
>>> sum(5,8)
13
>>> sum(5,8,12)
25
```

Observation:

1. In the above example, `a` is a positional parameter and is mandatory, `b` is a default argument and is optional, `x` is the collection of variable arguments and is optional. Thus, at least 1 argument in the function call is compulsory.
2. When only 1 argument is provided, the argument is considered to be the value of the positional parameter `a` and parameter `b` takes on the value `100`.
3. When 2 arguments are provided, they are assumed to be the values of `a` and `b` respectively and `x` is empty.
4. When more than 2 arguments are provided, the first 2 arguments are considered to be the values of `a` and `b` respectively and the rest will be present in `x` in the same sequence as in the call.

Of course, the positional parameter and default argument can be provided as keyword arguments if required as shown in the examples below:

```
>>> sum(b=2,a=5)
7
>>> sum(7,b=2)
9
```

Here are some more cases to consider:

```
>>> sum(3,a=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() got multiple values for argument 'a'
```

Observation:

1. If the first argument is provided, it is assigned to the positional parameter `a`. Reassignment using keyword arguments is not permitted.

```
>>> sum(3,5,b=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() got multiple values for argument 'b'
```

Observation:

1. If a second argument is provided, it is assigned to the parameter `b`. Reassignment using keyword arguments is not permitted.

10.7.3 Variable Arguments Followed by Default Arguments

As a special case, if variable arguments are followed by default arguments, the result is that the value for the default argument can only be explicitly provided using keyword arguments. Recollect the built-in `print()` function: it obviously supports variable arguments since it can print any number of values. It also uses a separator with a default value of space and a terminator with a default value of newline. The only way to specify a custom separator or terminator is by using keyword arguments:

```
>>> print(1,2,3,sep=':',end='\n*****\n')
1:2:3
*****
```

Here is a function `sum` that receives variable arguments along with a default argument following it:

```
>>> def sum(*x,negate=False):
...     s=0
...     for i in x: s+=i
...     if negate: s = -s
...     print(s)
...
>>> sum(2,3,4)
9
>>> sum(2,3,4,negate=True)
-9
>>> sum(2,3,4,True)
10
```

Observation:

1. In the example `sum(2,3,4)`, all the arguments go into `x`, leaving `negate` with the default value, `False`.
2. In the example `sum(2,3,4,negate=True)`, the keyword argument `negate=True` ensures that the parameter `negate` takes on the value `True`. All other arguments are stored in `x`.
3. In the example `sum(2,3,4,True)`, all arguments are stored in `x`. Thus, the final argument `True` is also a part of `x` and counted as `1` during addition. Specifically, the value `True` is not taken to be the value for `negate`.

10.7.4 Variable Arguments Followed by Keyword Arguments

1. Section 10.5 introduced the capability to accept default values for missing arguments.
2. Section 10.7 introduced the capability to accept variable number of arguments.
3. Section 10.6 introduced the capability of accepting arbitrary keyword arguments.

All of these can be combined as shown below:

```
>>> def sum(a,b=2,*c,**d):
...     print("a:",a,"b:",b)
...     print("c:",end='')
...     for i in c: print(i,end=' ')
...     print("\nd:",end='')
...     for k,v in d.items(): print("({}={})".format(k,v),end='')
...     print()
...
>>> sum(1,2,3,4,5,6,7,x=2,y=4,z=9)
a: 1 b: 2
c:3 4 5 6 7
d:(x=2)(y=4)(z=9)
```

10.8 Returning From Functions

When a function is called, it returns automatically after it's entire body executes by default. It is possible to ensure that the function returns immediately, whenever required, using the `return` statement. On execution, the `return` statement immediately transfers control back to the caller.

The complete syntax of `return` statement is:

```
return [value]
```

Forms:

```
1.      return
2.      return value
```

Form #1 will be covered in this section whereas form #2 will be covered in the next section.

```
>>> def hi():
...     print("Hello World")
...     return
...     print("Heya!")
...
>>> hi()
Hello World
```

Observation:

1. The function `hi` did not end up printing the string "`Heya!`" since the `return` statement preceding it resulted in a transfer of control back to the caller.
2. Unconditional `return` statements such as the one shown in this example are practically useless as the rest of the statements never get executed. It is common for a function to have multiple `return` statements, but in such cases the `return` statement would be conditional – probably present within an `if` statement's body.

10.9 Returning Single Values From Functions

The previous function did not return any value back to the caller. As per the definition of functions in classical computer science, a function must return a value. Python acknowledges this rule by ensuring that functions that return nothing end up returning `None` by default! This can be verified by assigning the function call to a variable and examining it's value or directly printing the value returned by the function call, both of which are shown below:

```
>>> def hi():
...     print("Hello World")
...
>>> x=hi()
Hello World
>>> print(x)
None
>>> print(hi())
Hello World
None
```

Observation:

1. In the above examples, each call to the function `hi` will anyway print "`Hello World`" apart from anything else that we have given.
2. There is a lot of difference between `hi` and `hi()`: `hi` is a reference to the function whereas `hi()` is a function call that results the function getting executed and it's return value getting substituted at the place of call!

But what is practically more useful however is that we can return values from functions. The `return` statement also permits the optional return of a single value that will be substituted at the place of call.

We have dealt with the `sum` function many times before, but each time it was printing the result on the standard output. This time, we will modify it to return the sum instead of printing it:

```
>>> def sum(x,y):
...     return x+y
...
>>> sum(2,3)
5
>>> s=sum(2,3)
>>> print(s)
5
```

As was discussed in the previous section, there can be multiple such `return` statements within a function, but the moment one of them is executed, control is returned to the caller and the return value is substituted at the place of call.

Let us apply this by writing a program to find the GCD of 2 integers using a function called `gcd` which returns the result to the main script.

gcd.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the GCD of 2 integers using functions
4.
5.   def gcd(x,y):
6.       if x==y: return x
7.       elif x>y: return gcd(x-y,y)
8.       else: return gcd(x,y-x)
9.
10.  x,y = input("Enter 2 integers:").split(' ')
11.  x,y = int(x),int(y)
12.
13.  print("The GCD of {} and {} is {}".format(x,y,gcd(x,y)))
```

Output:

```
Enter 2 integers:36 60
The GCD of 36 and 60 is 12
```

Observation:

1. The `gcd` function is defined in line 5. A function definition should always precede function calls.
2. The GCD is found by recursively subtracting the smaller integer from the larger integer till the 2 integers become equal, at which point they converge at the GCD. Other methods of finding the GCD can also be employed.
3. Line 10 reads in a single line as a string. This line contains 2 integers and hence the `split()` function has been used to split the string at the space to give us the 2 parts. The two parts are stored in `x` and `y` respectively, but are still strings and not integers.
4. Line 11 converts the strings in `x` and `y` into integers and stores them back into the same variables.
5. Line 13 calls the `gcd` function passing `x` and `y` as arguments and the returned value is printed as part of the formatted string.

Let us rewrite `prime3.py` using functions:

`prime7.py`

```
1.   #!/usr/bin/python
2.
3.   # Program to classify a given number
4.   # as prime or composite using functions.
5.
6.   def isPrime(n):
7.       for i in range(2,int(n/2)+1):
8.           if n%i==0: return False
9.       return True
10.
11.  n = int(input("Enter a positive integer: "))
12.
13.  if isPrime(n): print("Prime")
14.  else: print("Composite")
```

Observation:

1. The main program starts from line 11 and uses the `isPrime()` function to determine if the given number is prime or not.
2. The `isPrime()` function is loosely based on the code of `prime3.py`, with optimizations to reduce code.
3. The `int()` function is used in line 7 to prevent floating point results due to division. The `+1` is provided because the range function excludes the ending value, and we need the last value to be `n/2`.
4. If we find that the number has a factor (and hence is composite) inside the loop (in line 8), we immediately return `False` to indicate that the number is not prime. No other statements will be executed within the loop or the function and control returns to the main program (in line 13).
5. If control comes out of the loop, it is an indication that the number is prime and we unconditionally return `True` to indicate the same.

10.10 Returning Collections From Functions

There is a restriction that only a single value can be returned from a function. What if we want a function to return multiple values? An intelligent solution is to return a single collection instead! The collection could be a list, a tuple or a dictionary. Later on, after we learn OOP in section 12, we will be able to return objects too!

10.10.1 Returning Tuples From Functions

Here is a function called `calc` that returns a tuple containing the sum, difference, product and quotient after division, given 2 inputs `x` and `y`:

```
>>> def calc(x,y):
...     t=(x+y,x-y,x*y,x/y)
...     return t
...
>>> result=calc(2,3)
>>> print(result)
(5, -1, 6, 0.6666666666666666)
>>> print("Sum=",result[0],"difference=",result[1])
Sum= 5 difference= -1
```

Of course, we do not necessarily need the temporary variable `t` in the function, and the function can as well be defined as follows:

```
>>> def calc(x,y):
...     return (x+y,x-y,x*y,x/y)
...
>>> result=calc(2,3)
>>> print(result)
(5, -1, 6, 0.6666666666666666)
```

10.10.2 Returning Lists From Functions

Here is a function called `calc` that returns a list containing the sum, difference, product and quotient after division, given 2 inputs `x` and `y`:

```
>>> def calc(x,y):
...     l = [x+y,x-y,x*y,x/y]
...     return l
...
>>> result=calc(2,3)
>>> print(result)
[5, -1, 6, 0.6666666666666666]
>>> print("Sum=",result[0],"difference=",result[1])
Sum= 5 difference= -1
```

Of course, we do not necessarily need the temporary variable `l` in the function, and the function can as well be defined as follows:

```
>>> def calc(x,y):
...     return [x+y,x-y,x*y,x/y]
...
>>> result=calc(2,3)
>>> print(result)
[5, -1, 6, 0.6666666666666666]
>>> print("Sum=",result[0],"difference=",result[1])
Sum= 5 difference= -1
```

Let us write a program that uses a function to generate all prime factors of a given integer. We can then find the prime factors of 2 integers and use this information to generate the GCD of the 2 integers (which will be the product of the common prime factors):

gcd2.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the GCD using functions
4.   # to find the prime factors
5.
6.   def getPrimeFactors(n):
7.       factors = []
8.       factor=2
9.       while n>1:
10.          while n%factor == 0:
11.              factors.append(factor)
12.              n /= factor
13.          factor = factor+1
14.      return factors
15.
16.  def gcd(x,y):
17.      factors1 = getPrimeFactors(x)
18.      factors2 = getPrimeFactors(y)
19.      gcd=1
20.      for i in factors1:
21.          if i in factors2:
22.              factors2.remove(i)
23.              gcd *= i
24.      return gcd
25.
26.  x,y = input("Enter 2 integers:").split(' ')
27.  x,y = int(x),int(y)
28.  print("The GCD of {} and {} is {}".format(x,y,gcd(x,y)))
```

Output:

```
Enter 2 integers:248 356
The GCD of 248 and 356 is 4
```

Observation:

1. The `getPrimeFactors()` function defined in line 6 returns a list containing all the prime factors of a given integer. For example, the prime factors that when multiplied gives us 12 are 2, 2 and 3.
2. The `gcd()` function in line 16 finds and returns the GCD of 2 integers. It first finds the prime factors of both the integers using `getPrimeFactors()`. For each prime factor of the first number, it checks if there is an identical prime factor of the second number. If so, it removes it from the list and multiplies it to the GCD. Finally, the GCD is returned.

10.10.3 Returning Dictionaries From Functions

Here is a function called `calc` that returns a dictionary containing the sum, difference, product and quotient after division, given 2 inputs `x` and `y`:

```
>>> def calc(x,y):
...     d = {'sum':x+y,'diff':x-y,'prod':x*y,'quot':x/y}
...     return d
...
>>> result=calc(2,3)
>>> print(result)
{'quot': 0.6666666666666666, 'diff': -1, 'prod': 6, 'sum': 5}
>>> print("Sum=",result['sum'],"difference=",result['diff'])
Sum= 5 difference= -1
```

Of course, we do not necessarily need the temporary variable `d` in the function, and the function can as well be defined as follows:

```
>>> def calc(x,y):
...     return {'sum':x+y,'diff':x-y,'prod':x*y,'quot':x/y}
...
>>> result=calc(2,3)
>>> print(result)
{'quot': 0.6666666666666666, 'diff': -1, 'prod': 6, 'sum': 5}
>>> print("Sum=",result['sum'],"difference=",result['diff'])
Sum= 5 difference= -1
```

10.11 Global Variables

Global variables are variables that are accessible throughout the script – including inside functions as well as outside of all functions. While this makes it convenient for multiple functions to share the same variable(s), it is frowned upon as being an unprofessional practice for the following reasons:

1. Global variables are invisibly tied to the function – it may not be obvious to a person reading the program that the function uses/relies on a particular global variable. Parameters are far more readable, giving us a clear picture as to what inputs the function depends on.
2. Global variables are shared by all functions and when a particular function is buggy and messes with the value of the global variable, it may result in other functions also behaving in an unexpected fashion, making it difficult to debug and identify the offending function. Parameters are local to the function and are not shared across functions.
3. The dependency of the function on the global variable makes it more difficult to reuse the function across programs – you can't, for example, simply copy and paste the function to another program without also copying the global variable (though a copy-paste is not a professional way of reuse). Also, the function caller should be aware of the value that is to be stored in the global variable before the call is made. On the other hand, parameters of a function belong to the function and are copied when the function definition is copied.
4. When multiple such functions co-exist within the same program, there can be confusion related to which globals are used by which functions. On the other hand, parameters in different functions are different even if their names are the same.
5. Global variables continue to exist till the end of the program, which may be long after we require the variable, wasting memory and polluting the namespace. Parameters die the moment the called function returns.

Thus, it is a bad idea to use global variables. But for now, we will continue with this topic and see how to make it possible to use globals in functions nevertheless.

All variables used inside functions are local to the function by default. In other words, Python helps you by assuming you always want locals instead of globals. This is illustrated in the example below:

```
>>> x=2
>>> def f():
...     x=3
...     print(x)
...
>>> print(x)
2
>>> f()
3
```

```
>>> print(x)
2
```

Observation:

1. The line `x=2` introduces a global variable with name `x` and value `2`.
2. The function `f` has a reference to the variable `x`, but this is assumed to be a local variable `x` that is local to the function `f`. This variable is created when the function is called and is destroyed when the function returns. During the lifetime of the function, however, it exists with a value of `3`.
3. Any reference to a variable inside functions is assumed to be a local reference by default. Any reference to a variable outside functions is assumed to be a global reference.
4. Any reference to `x` outside of all functions is a reference to the global variable `x`, with the value `2`. Any reference to `x` inside the function `f` is a reference to the local variable `x`, with the value `3`.

What if in the previous example, we want the function to use the global variable `x` instead of creating a new local variable? We make this intention known to Python using the keyword `global` to seek access to one or more global variables as shown in the syntax below:

```
global name1[,name2[,...]]
```

This is demonstrated in the example below:

```
>>> x=2
>>> def f():
...     global x
...     x=3
...     print(x)
...
>>> print(x)
2
>>> f()
3
>>> print(x)
3
```

Observation:

1. The statement `global x` makes it possible for the function `f` to access the global variable `x`, which otherwise would have construed to mean a local variable `x`.
2. A function that needs access to a global variable has to compulsorily use the `global` statement to request access.
3. Once the `global` statement is used to request access to a global variable, it is not possible to have a local variable with the same name.
4. A function can request access to multiple global variables simultaneously using either a single `global` statement and a list of all global variables or using multiple `global` statements.

10.12 Nested Functions

Just as how the previous section introduced the concept of global variables which are defined outside of all functions and can be accessed by all function using the `global` statement, it won't be wrong to similarly assume that all functions are also defined in the global scope usually. Of course, functions do not need a `global` statement to make them accessible to other functions. Similarly, just as how we can have local variables inside functions that are accessible only within the function, we can define a function within a function, making it local to the function within which it is defined and thus accessible only to the function within which it is defined.

A function defined within another function is called a *nested function*. In such cases, we differentiate between these functions using the terminology of *outer function* and *inner function*. Thus, the inner function is the one that is defined inside of the outer function, is local to the outer function, and accessible only to the outer function. This is demonstrated below:

```
>>> def f():
...     print("This is f")
...     def g():
...             print("This is g")
...     g()
...
>>> f()
This is f
This is g
>>> g()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
```

Observation:

1. The function `f` is the outer function and the function `g` is the inner function.
2. As is the case everywhere else in Python, encountering a function definition does not automatically result in a function call. An inner function is useless unless used in some way. The typical ways of use are by making a call (as done in this example) or by returning it (as will be done in the following part).
3. The function `g` belongs to the scope of the function `f` (`g` is local to `f`) and is hence accessible only by `f`. Any attempt to access it from outside `f`, as shown in the example, will result in an error.

Here are the reasons why we might want to deal with inner functions, with each category of application covered in detail in following sections:

1. Inner functions do not pollute the global namespace. Thus, if the functionality provided by the inner function is known to benefit only the outer function, it makes sense to "hide" the inner function in the outer function without polluting the global namespace. This also makes it possible to have multiple [inner] functions with the same name since they are all in different scopes!
2. Sometimes, the outer function is simply a wrapper around the inner function with the main work being done by the inner function. The existence of the outer function could be to perform validation (verifying that the inputs to the function are acceptable), logging(recording the call in some file or data structure), choosing(selecting one inner function from amongst several), etc.
3. The outer function can also act like a generator – a function that helps manufacture a suitable (inner) function and returning that function for external use!

10.12.1 Inner Functions for Private Use

Let us revisit `gcd2.py`: we observe that the function `getPrimeFactors()` is used internally by the function `gcd()`. If we wish to "protect" it and make it available only to the `gcd` function, it can be rewritten as follows:

gcd3.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the GCD using functions
4.   # to find the prime factors, making use of inner
functions.
5.
6.   def gcd(x,y):
7.       def getPrimeFactors(n):
8.           factors = []
9.           factor=2
```

```
10.          while n>1:
11.              while n%factor == 0:
12.                  factors.append(factor)
13.                  n /= factor
14.              factor = factor+1
15.          return factors
16.
17.      factors1 = getPrimeFactors(x)
18.      factors2 = getPrimeFactors(y)
19.      gcd=1
20.      for i in factors1:
21.          if i in factors2:
22.              factors2.remove(i)
23.              gcd *= i
24.      return gcd
25.
26.  x,y = input("Enter 2 integers:").split(' ')
27.  x,y = int(x),int(y)
28.  print("The GCD of {} and {} is {}".format(x,y,gcd(x,y)))
```

Output:

```
Enter 2 integers:248 356
The GCD of 248 and 356 is 4
```

Observation:

1. This program is identical to `gcd2.py`, except for the fact that the `getPrimeFactors()` function is now defined inside the function `gcd`, in line 7.
2. In `gcd2.py`, the `getPrimeFactors()` was a global function, accessible to any other function defined after it, but in this program it is a local function, accessible only to the function `gcd()`.

10.12.2 Outer Functions for Abstraction

We revisit `prime7.py` and add error checking functionality to it without disturbing the contents of the function, as shown in the program below:

prime8.py

```
1.   #!/usr/bin/python
2.
3.   # Program to classify a given number
4.   # as prime or composite using functions.
5.   import sys
6.   def isPrime(n):
7.       def implIsPrime(n):
8.           for i in range(2,int(n/2)+1):
9.               if n%i==0: return False
10.          return True
11.
12.      if n<0: print("Negative integer specified!")
13.      elif n==0: print("Zero given!")
14.      elif n==1: print("1 is neither prime nor composite!")
15.      else: return implIsPrime(n)
16.      sys.exit()
17.
18.  n = int(input("Enter a positive integer: "))
19.
20.  if isPrime(n): print("Prime")
21.  else: print("Composite")
```

Output:

```
Enter a positive integer: -26
Negative integer specified!
```

```
Enter a positive integer: 0
Zero given!
```

```
Enter a positive integer: 1
1 is neither prime nor composite!
```

```
Enter a positive integer: 2
Prime
```

Observation:

1. The original code of `isPrime()` from `prime7.py` is now present in `implIsPrime()` inside `isPrime()` in this program.
2. The `isPrime()` function here checks the input given and validates it. If the given input is not acceptable, it prints a suitable message. If not, it uses the `implIsPrime()` function (the actual implementation of `isPrime`) to determine whether the integer is prime or not and returns the result. In case the input is not acceptable, it uses the `sys.exit()` function to terminate the script rather than returning to the caller.
3. A more professional approach of handling unacceptable input is by raising exceptions, a concept that is covered later in section 13.

10.12.3 Outer Functions as Generators

We know that `math.pow()` function allows us to find `x` to the power `y`. There are no specialised functions available for finding the square and the cube of an integer `x`. How about creating special versions of the `pow()` function for this using a generator function to help generate a specialised version of `pow()`?

generator.py

```
1.   #!/usr/bin/python
2.
3.   # Program to demonstrate outer functions
4.   # as generators using inner functions.
5.   import math
6.
7.   def power(n):
8.       def implPower(x):
9.           return int(math.pow(x,n))
10.      return implPower
11.
12.  square=power(2)
13.  cube=power(3)
14.
15.  print("Equation: ax^3 + bx^2 + cx + d")
16.  a,b,c,d = input("Enter the values of a, b, c, d:").split('
')
17.  a,b,c,d = int(a),int(b),int(c),int(d)
18.  x = int(input("Enter the value of x:"))
19.  result=a*cube(x) + b*square(x) + c*x + d
20.  print("Result:",result)
```

Output:

```
Equation: ax^3 + bx^2 + cx + d
Enter the values of a, b, c, d:3 -2 5 4
Enter the value of x:2
Result: 30
```

Observation:

1. The `power()` function is a generator function that returns a reference to function! Thus, line 12 creates a function and stores it's reference in the variable `square`, and line 13 similarly stores a function reference in the variable `cube`.
2. The variables `square` and `cube` can now be used as regular functions since they are references to functions. They are used in line 19.
3. The power function returns `implPower`, which is a reference to a function.
4. The `implPower` function is an inner function of `power` and uses the local context of `power`. The local variable `n` of `power` can be and is used in the inner function `implPower`. Thus, there can be as many unique versions of `implPower` as unique values of `n` passed to `power()`!
5. Thus, the reference in `square` is a reference to the `implPower()` function with `n=2` already substituted. Similarly in the case of `cube`, `n=3`.
6. For simplicity, we are dealing with integers here. This justifies the `int()` call in lines 9, 17 and 18.
7. We are using the `pow()` function of the `math` module in line 9. Hence we are importing the `math` module in line 5.

10.13 Lambda Expressions

Python provides a convenient mechanism for creating anonymous functions called *Lambda Expressions* and permits invoking them using their reference.

Lambda expressions are ideally used when we have something simple to be done (something that can fit within a single expression), and we are more interested in quickly getting the job done rather than formally naming the function.

Before we go on to powerful examples, let us consider some simple ones. But first, here is the syntax of a lambda expression:

```
lambda parameters: expression
```

A lambda can receive 0 or more parameters (just like functions) and end up returning the value of the expression on execution. Here is the first example:

```
>>> (lambda x: x**2)(4)
16
```

Observation:

1. We have created a lambda function using the keyword `lambda`. This function accepts a single parameter `x` and returns it's square.
2. We wish to execute the lambda function immediately with `4` as an argument. We therefore use parentheses to represent the function reference returned by the `lambda` keyword and pass `4` as an argument to it.
3. Even though this example is valid, it is useless practically as we could have as well directly used the expression `4**2` instead. But this definitely serves as a first valid example.

Since the keyword `lambda` manufactures a function and returns it's reference, we can store the function reference in any variable and then use that variable like a function. This is illustrated below:

```
>>> square=lambda x: x**2
>>> square(4)
16
```

Observation:

1. We have created the same lambda expression as in the previous example, but this time have stored it in the variable `square`.
2. We then use `square` to invoke the functionality whenever we want, how many ever times we want and with whatever arguments we want.
3. The use of lambda this way is again not very practical as we could have as well defined a function called square that works in a similar fashion. But this example illustrates the similarity in handling function references.

Lambda expressions are especially useful when function objects are required. A *function object* is a reference to a function that can be stored (in variables), passed around in the script (as arguments to functions and return values from functions) and can be invoked whenever required. We will see examples of this in section 11.6.

10.14 Unpacking Argument Lists

When a collection is passed as an argument to a function, it is received as a collection object only and not as individual elements of the collection. To illustrate, consider the following example:

```
>>> def f(a):
...     print(a)
...
>>> f((2,4,6,8))
(2, 4, 6, 8)
```

Observation:

1. The function `f` is designed to receive a single parameter.
2. We are passing a single tuple containing 4 values: `(2,4,6,8)`.
3. The function receives the entire tuple `(2,4,6,8)` as a single collection in `a`.

This behaviour is useful when we specifically want a function to receive collections like tuples, lists, sets and dictionaries. But what if we have multiple values already stored in a collection and we wish to send them as individual arguments to a function? Recollect the `calc` function discussed in section 10.10.1:

```
>>> def calc(x,y):
...     return (x+y,x-y,x*y,x/y)
```

Given 2 values, this function will return as tuple containing their sum, difference, product and quotient. If we have the 2 input values already in a list, we will not be able to directly pass the list as an argument as it will be considered to be a single list argument, as illustrated below:

```
>>> def calc(x,y):
...     return (x+y,x-y,x*y,x/y)
...
>>> L=[2,3]
>>> calc(L)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: calc() missing 1 required positional argument: 'y'
```

The solution is *argument unpacking*: extracting individual elements from a collection and considering them to be individual arguments. The syntax for argument unpacking is shown below:

```
*collection
**dictionary
```

Note:

1. The `*collection` syntax is used for tuples, lists and sets whereas the `**dictionary` syntax is used specifically for dictionaries.
2. The `*collection` syntax expands the collection into arguments that can either map on to corresponding parameters of a function or can be received as variable arguments using the `*args` syntax discussed in section 10.7.
3. The `**dictionary` syntax expands the dictionary into keyword arguments that can either map on to corresponding named parameters of a function or can be received as additional keyword arguments using the `**kargs` syntax discussed in section 10.6.

This is illustrated in the revised examples below:

```
>>> def calc(x,y):
...     return (x+y,x-y,x*y,x/y)
...
>>> L=[2,3]
>>> calc(*L)
(5, -1, 6, 0.6666666666666666)
```

```
>>> def calc(x,y):
...     return (x+y,x-y,x*y,x/y)
...
>>> D={'x':2,'y':3}
>>> calc(**D)
(5, -1, 6, 0.6666666666666666)
```

10.15 Documentation Strings

Documentation strings are strings added at strategic locations within the program to provide documentation for a program element. Such documentation strings are recognised by the Python parser and can also be parsed by various external tools.

As far as functions are concerned, a documentation string can provide documentation on what the function is all about. The documentation string is identified as a string that starts on the first line of a function definition. Since the documentation string usually spans multiple lines, it is a common practice to use triple-quotes for enclosing it.

There are certain conventions recommended for forming good documentation strings:

1. The first line should be a 1-line summary of the function's purpose. This line should typically start with an uppercase alphabet and end with a period, avoiding the function name (unless it is being used as an English word). This line must be indented as the rest of the function body is.
2. The first line (summary) is separated from the rest of the lines (description) by a blank line.
3. The description can span any number of lines and can provide additional information about the function and it's specialities. These are generally indented as the rest of the function body is (but indentation is not compulsory). Any spaces and tabs used for indentation is retained as part of the documentation string by the Python parser (but other tools can be designed to ignore the indentation)

Here is a revised `isPrime()` function definition, modified from `prime7.py`:

```
>>> def isPrime(n):
...     """Returns whether n is prime or not.
...
...     Given a non-negative integer n, this function
...     returns True if n is prime and False otherwise.
...     The argument n is assumed to be a positive integer
...     greater than 1 (since 1 is neither prime nor composite).
...     No argument validation is performed.
...     """
...     for i in range(2,int(n/2)+1):
...             if n%i==0: return False
...     return True
...
```

Since the Python parser parses through the documentation string, we can always access it using the special member `__doc__` of the function as shown below:

```
>>> isPrime.__doc__
'Returns whether n is prime or not.\n\n\tGiven a non-negative integer
n, this function\n\treturns True if n is prime and False
otherwise.\n\tThe argument n is assumed to be a positive
integer\n\tgreater than 1 (since 1 is neither prime nor
composite).\n\tNo argument validation is performed.\n\t'
```

Observation:

1. We are using the function reference (`isPrime`) to access the member `__doc__`. Note that there are no parentheses after the function name, preventing it from becoming a function call.

2. Note that every whitespace used for indentation is retained within the string by the Python parser.

Not only that, we can access the documentation string using the built-in `help()` function in the Python interpreter, as shown in the examples below:

```
>>> help(isPrime)
Help on function isPrime in module __main__:

isPrime(n)
    Returns whether n is prime or not.

    Given a non-negative integer n, this function
    returns True if n is prime and False otherwise.
    The argument n is assumed to be a positive integer
    greater than 1 (since 1 is neither prime nor composite).
    No argument validation is performed.
```

```
>>> help(range)
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Returns a virtual sequence of numbers from start to stop by step.
 |
 |  Methods defined here:
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |
 |  __eq__(...)
 |      x.__eq__(y) <==> x==y
 |
 |  __ge__(...)
 |      x.__ge__(y) <==> x>=y
 |
 |  __getattribute__(...)
 |      x.__getattribute__('name') <==> x.name
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(...)
 |      x.__gt__(y) <==> x>y
 |
 |  __hash__(...)
 |      x.__hash__() <==> hash(x)
 |
 |  __iter__(...)
```

```
 |      x.__iter__() <==> iter(x)
 |
 |  __le__(...)
 |      x.__le__(y) <==> x<=y
 |
 |  __len__(...)
 |      x.__len__() <==> len(x)
 |
 |  __lt__(...)
 |      x.__lt__(y) <==> x<y
 |
 |  __ne__(...)
 |      x.__ne__(y) <==> x!=y
 |
 |  __reduce__(...)
 |
 |  __repr__(...)
 |      x.__repr__() <==> repr(x)
 |
 |  __reversed__(...)
 |      Returns a reverse iterator.
 |
 |  count(...)
 |      rangeobject.count(value) -> integer -- return number of
occurrences of value
 |
 |  index(...)
 |      rangeobject.index(value, [start, [stop]]) -> integer -- return
index of value.
 |      Raises ValueError if the value is not present.
 |
 |
----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  start
 |
 |  step
 |
 |  stop
 |
 |
----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

10.16 Programs Based on Functions

Q. Write a program to find the combination of 2 integers using functions.

combination.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the combination of 2 integers using
functions.
4.
5.   from math import factorial
6.
7.   def ncr(n,r):
8.       return int(factorial(n)/(factorial(r)*factorial(n-r)))
9.
10.  x,y = input("Enter 2 integers:").split(' ')
11.  x,y = int(x),int(y)
12.  print("The combination of {} and {} is
{}".format(x,y,ncr(x,y)))
```

Observation:

1. Line 10 receives 2 values from the user in a single line, using the `split()` function to split the line into 2 values at the space.
2. Line 11 converts the strings in `x` and `y` into integers and stores them back into the same variables.
3. The combination is found using the `ncr` function defined in line 7. This function uses the built-in `factorial` function from the `math` module.
4. The `math.factorial` function becomes directly available in the program due to the `import` in line 5.

10.17 Questions

1. List the advantages of functions.
2. Write short notes on positional arguments and default arguments.
3. Write short notes on function definition and function call in Python.
4. Write a short note on keyword arguments.
5. Explain how variable arguments can be processed by a function in Python.
6. What will be the return value of a function in Python if no return statement is present in the function?
7. Explain 2 different ways in which a function can return multiple values back to the caller.
8. Can a nested function be called from the global scope in Python? Elaborate.

9. What are lambda expressions in python? Explain with syntax and examples.

10.18 Exercises

1. Write a program using functions to check if a given number is an ugly number. Ugly numbers are positive numbers whose only prime factors are 2, 3 or 5.
2. Write a function in Python to calculate and return the average of a given list of integers.
3. Write a function in Python called `isIn()` to check for the existence of a substring within a string returning a Boolean value.
4. Write a function in Python to demonstrate keyword arguments.
5. Write a script in Python to demonstrate nested functions.
6. Write a lambda expression to convert a given angle from degrees to radians.
7. Write a function to find the distance between 2 points, where each point is passed as a tuple to the function.

SUMMARY

- Functions need to be defined before they can be called.
- Functions are objects and references to them can be stored in variables!
- Positional Arguments are always assigned on a one-to-one basis, from the function call arguments to the function definition parameters.
- Default Arguments cannot be followed by Positional Arguments. Any default argument omitted at the place of call will take on its default value in the definition.
- Keyword Arguments can be used to change the order of Positional Arguments at the place of function call.
- Keyword Arguments can be used to specify certain Default Arguments while leaving out others.
- Keyword Arguments permit the creation of dynamic, user-defined parameters that can be received and processed by the function without documenting the parameter names in advance!
- Keyword Arguments cannot be followed by non-keyword arguments.
- Variable Arguments permit the function call to pass as many arguments as required and yet be processed using a single collection variable in the function definition.
- Variable Arguments should not be followed by Positional Arguments in the function definition parameter list.

Summary

- A function can return a single value to the caller. This single value can also be a collection! When a function does not specify what to return, the return value is taken as None. A function can return at any time using the return statement, optionally returning a value to the caller.
- Global variables can be accessed by functions by using the global keyword and specifying the global variables the function wishes to access.
- A function definition can contain other function definitions inside!
- A lambda expression is a convenient and powerful way of creating an anonymous function!
- A list of arguments can be unpacked and sent as individual arguments to a function by using the *args syntax.
- A dictionary of keyword arguments can be unpacked and sent as individual keyword arguments to a function by using the **args syntax.
- Documentation Strings can help document functions and provide help on the function usage to other programmers.

11 Practical Python

In this chapter you will be able to:

- ☑ Implement data structures like stacks and queues.
- ☑ Use the map(), filter() and reduce() functions to write powerful code.
- ☑ Use list comprehensions and nested list comprehensions to build and populate lists quickly and easily.
- ☑ Solve certain real-life problems with ease.

PRACTICAL PYTHON

11.1 Implementing Stacks

A *stack* is a linear data structure that follows the LIFO (Last In First Out) principle. Thus, the last element added into a stack is the first element to get deleted. The insertion operation is called a *push* operation whereas the deletion operation is called a *pop* operation.

There is no direct support for stacks in Python (meaning there is no class or data type called `stack`), but stacks can be housed in suitable containers and operated upon. Tuples and Lists are two containers that are linear, but tuples being immutable cannot be used to store a stack as the stack contents should be able to change with time. Lists therefore are the best choice to house a stack.

A stack has two ends: a *bottom* end which is fixed and from where the stack grows and a *top* end which is dynamic and where push and pop operations take place.

This functionality can be implemented using a list in this manner:

1. The push operation can be performed using `list.append()` which appends an element at the end of the list (the top end of the stack).
2. The pop operation can be performed using the `list.pop()` function which deletes the last element in the list (from the top of the stack).
3. The peek operation, which provides the last element of the stack (the topmost element) without deleting it from the stack, can be implemented as `list[-1]`.
4. The number of elements in the stack can be found out using `len(list)`. This can also be used to find if the stack is empty or not.
5. The stack can be cleared using the `list.clear()` function.
6. More functionality is possible that typical stacks may not provide, like searching for an element in the stack, appending stacks, etc.

Here is a simple program to demonstrate basic stack operations:

stackdemo.py

```
1.   #!/usr/bin/python
2.
3.   # Stack demonstration using lists
4.
5.   S = []       # Creation of a stack S
6.   S.append(10) # Push 10
7.   S.append(20) # Push 20
```

```
8.   S.append(30) # Push 30
9.   print(S)     # Display the stack
10.
11.  print(S.pop())     # Pop 30
12.  print(S)     # Display the stack
13.  print(S[-1]) # Peek
14.
15.  print(len(S))# Stack size
16.  print(len(S)==0)# Stack empty?
```

Output:

```
[10, 20, 30]
30
[10, 20]
20
2
False
```

11.2 Implementing Queues

A *queue* is a linear data structure that follows the FIFO (First In First Out) principle. Thus, the first element added into a queue is the first element to get deleted.

There is no direct support for queues in Python (meaning there is no class or data type called `queue`), but queues can be housed in suitable containers and operated upon. There is a class called `deque` in the `collections` module that provides support for a double-ended queue where insertions and deletions are restricted to both the ends. A more restricted use of the deque will give us the required functionality as follows:

1. To insert an element in the queue, the `deque.append()` function can be used.
2. To delete an element from the queue, the `deque.popleft()` function can be used.
3. To find the number of elements in a queue, the `len(deque)` function can be used. This function can also be used to find if the queue is empty or not.
4. To remove all elements from the queue, the `deque.clear()` function can be used.
5. In addition, the `deque.insert()` function permits insertion of an element at any place within the queue (thus allowing implementation of priority queues) and the `remove()` function deletes the first occurrence of a given value from

a queue. The `index()` function searches for the first occurrence of the given element within the queue (or within a region of the queue).

Here is a simple program to demonstrate basic queue operations:

queuedemo.py

```
1.   #!/usr/bin/python
2.
3.   # Queue demonstration using collections.deque
4.
5.   from collections import deque
6.
7.   q = deque()         # Creation of a queue
8.   q.append(10)        # Insert 10
9.   q.append(20)        # Insert 20
10.  q.append(30)        # Insert 30
11.  print(q)            # Display the queue
12.
13.  print(q.popleft()) # Delete 10
14.  print(q)            # Display the queue
15.  print(q[0])         # Peek
16.
17.  print(len(q))       # Queue size
18.  print(len(q)==0)    # Queue empty?
```

Output:

```
deque([10, 20, 30])
10
deque([20, 30])
30
2
False
```

11.3 The map() Function

The `map` function maps or transforms each element of an iterable resulting in a new iterable containing the transformed elements. The number of the elements in the output sequence will be equal to the number of elements in the input sequence.

Syntax:

```
map(func,sequence)
```

Each element in `sequence` is sent to the function `func` as an argument, and the return value from the function is sent to an output sequence that is finally returned by `map()`.

Here is an example of using `map` to multiply each element of a list by 2 and obtain a new list with the corresponding results:

```
>>> def f(x): return 2*x
...
>>> L=[1,6,4,9,7]
>>> M=list(map(f,L))
>>> M
[2, 12, 8, 18, 14]
```

Observation:

1. Our input list is `L`, comprising of the elements `1`, `6`, `4`, `9` and `7`.
2. We define a mapping function called `f`, which returns 2 times the value of the original argument.
3. We use the `map()` function passing a reference to our mapping function(`f`), as the first argument and our input list(`L`) as the second.
4. The `map()` function will map each element of `L` using `f` for individual element conversion and returns the final result as an iterable object, which we convert to a list for convenience.
5. We finally observe that the list `M` contains elements with values twice of those elements that were present in the list `L`, and in the same order as found in `L`.

Similarly, here is an example of converting a list of strings to a list of corresponding strings with all characters converted to uppercase:

```
>>> def f(x): return x.upper()
...
>>> L=["Hello","World"]
>>> M=list(map(f,L))
>>> M
['HELLO', 'WORLD']
```

11.3.1 Using map() With Multiple Iterables

A more general syntax of the `map()` function accepts multiple iterables instead of one. In such a mapping, corresponding elements from each of the iterables are taken at one time and sent to the mapping function and the return value is collected into an output sequence that is finally returned. Of course, now the mapping function has to receive as many arguments as the number of iterables passed to the `map` function!

Here is an example of using `map` to add corresponding elements of 3 lists:

```
>>> L1=[1,2,3]
>>> L2=[4,5,6]
>>> L3=[7,8,9]
>>> def f(a,b,c): return a+b+c
...
>>> M=list(map(f,L1,L2,L3))
>>> M
[12, 15, 18]
```

Observation:

1. The `map()` function is receiving the mapping function reference (`f`) and 3 lists (`L1`, `L2` and `L3`).
2. The mapping function (`f`) therefore receives 3 arguments (`a`, `b` and `c`) – one per list. The function `f` returns the sum of it's arguments.
3. The number of elements in the output sequence is the same as the number of elements in the other input lists. If the input lists did not all have the same number of elements, the smallest list is considered and the additional elements of the other lists are ignored. The length of the output list, therefore, is the same as the length of the smallest input list.

11.4 The filter() Function

As the name suggests, the `filter()` function filters a sequence – it's elements are passed through a filtering function and only those elements that pass a criteria are allowed to pass through while the rest are “blocked”.

Syntax:

```
filter(func,sequence)
```

Each element in the sequence `sequence` is sent to the function `func` as an argument, and if the function returns `True`, will be sent to an output sequence that is finally returned by `filter()`. The output sequence is returned as an iterable object

which can be sequentially accessed or can be converted to a suitable sequence like a list.

Here is an example of using `filter` to filter out odd integers from a list and retain only even elements:

```
>>> L=[1,6,4,9,7]
>>> def f(x): return x%2==0
...
>>> M=list(filter(f,L))
>>> M
[6, 4]
```

Observation:

1. Our input list is `L`, comprising of the elements `1`, `6`, `4`, `9` and `7`.
2. We define a filtering function called `f`, which returns `True` only if it's argument is even.
3. We use the `filter()` function passing a reference to our filtering function(`f`), as the first argument and our input list(`L`) as the second.
4. The `filter()` function will filter through the elements of `L` using `f` for each element of `L` and returns the final result as an iterable object, which we convert to a list for convenience.
5. We finally observe that the list `M` contains only the even elements that were present in the list `L`, and in the same order as found in `L`.

11.4.1 Special Cases of filter()

We have seen the typical use of `filter()` in the previous section. There are 2 special cases to be considered, however.

The first special case arises when the filtering function reference (first argument to `filter()`) is `None`. This will result in the filtering of all elements of the sequence based on their truth value – only those elements in the sequence that evaluate to Boolean `True` are allowed to pass through, as shown in the example below:

```
>>> L=[1,0,6,4,0,9,7]
>>> M=list(filter(None,L))
>>> M
[1, 6, 4, 9, 7]
```

For integers, `0` is considered `False` whereas any other integer value is considered `True`. Therefore, in the above example, we observe that the occurrences of `0` are filtered out.

The second special case arises when we want the reverse – we want to filter out those elements of the sequence that evaluate to Boolean `False`. This can be done using `itertools.filterfalse()` function, which is the negated form of `filter()` as demonstrated below:

```
>>> from itertools import filterfalse
>>> L=[1,0,6,4,0,9,7]
>>> M=list(filterfalse(None,L))
>>> M
[0, 0]
```

Observation:

1. Unlike the `filter()` function which is a built-in, the `filterfalse` function is present in the module `itertools` and hence needs to be imported.
2. When the filtering function reference (the first argument to `filterfalse`) is not `None`, this function filters out all those elements for which the filtering function returns `True` and only allows those elements to pass for which the filtering function return `False`, behaving in a manner opposite of `filter`.

11.5 The reduce() Function

As the name suggests, the `reduce()` function reduces a sequence to a single value – it's elements are cumulatively passed through a reducing function that reduces two arguments to a single value each time. The first argument to the reducing function is the accumulated value so far while the second argument is an element picked up from the sequence.

Syntax:

```
reduce(func,sequence[,initial])
```

The initial accumulated value is taken to be `initial` if specified, or `0` if not specified. To begin with, this initial accumulated value and the first element of sequence is passed to the function `func`. The return value from the function becomes the new accumulated value, which along with the next element from `sequence` is then passed to the function `func`. This process continues as many times as the number of elements in `sequence`. The final accumulated value is returned by `reduce()`.

Here is an example of using `reduce` to find the sum of all elements in a list:

```
>>> from functools import reduce
>>> def f(x,y): return x+y
...
>>> L=[1,6,4,9,7]
>>> sum=reduce(f,L)
>>> sum
27
```

Observation:

1. The `reduce()` function is imported from the `functools` module.
2. The reducing function (`f`) receives 2 arguments – the accumulated value and an element from the sequence – and returns their sum.
3. The `reduce` function accepts a reference to the reducing function (`f`) and the sequence to operate on (`L`)
4. Since `0` is the default initial accumulated value and is also the additive identity, we don't need to explicitly provide any initial value.

Similarly, here is an example of using `reduce` to find the product of all elements in a list:

```
>>> from functools import reduce
>>> def f(x,y): return x*y
...
>>> L=[1,6,4,9,7]
>>> product=reduce(f,L,1)
>>> product
1512
```

Observation:

1. This example is similar to the previous one, except for the third argument to `reduce()`.
2. Since 1 is the multiplicative identity and 0 is the default initial accumulated value, we need the initial accumulated value to be `1`. This is explicitly provided as the third argument to `reduce()`, failing which the product will always be 0.

11.6 Practical Use of map(), filter() and reduce() Using Lambda Expressions

The use of lambda expressions (section 10.13) in conjunction with `map()`, `filter()` and `reduce()` can help process sequences using minimal code! This section illustrates this point using a few practical examples.

First of all, let us revisit all the examples we have covered for `map()`, `filter()` and `reduce()` in the preceding sections and rewrite them using lambda expressions.

To multiply each element of a list by 2:

```
>>> L=[1,6,4,9,7]
>>> M=list(map(lambda x: 2*x, L))
>>> M
[2, 12, 8, 18, 14]
```

To convert all characters of strings in a list to uppercase:

```
>>> L=['Hello','World']
>>> M=list(map(lambda x: x.upper(), L))
>>> M
['HELLO', 'WORLD']
```

To add corresponding elements of 3 lists:

```
>>> L1=[1,2,3]
>>> L2=[4,5,6]
>>> L3=[7,8,9]
>>> M=list(map(lambda a,b,c: a+b+c, L1, L2, L3))
>>> M
[12, 15, 18]
```

To filter out the odd integers and allow only even integers of a list to pass through:

```
>>> L=[1,6,4,9,7]
>>> M=list(filter(lambda x: x%2==0, L))
>>> M
[6, 4]
```

To add all elements of a list:

```
>>> from functools import reduce
>>> L=[1,6,4,9,7]
>>> sum=reduce(lambda x,y: x+y, L)
>>> sum
27
```

To multiply all elements in a list:

```
>>> from functools import reduce
>>> L=[1,6,4,9,7]
>>> product=reduce(lambda x,y: x*y, L, 1)
>>> product
1512
```

And now, here is a program to analyse the marks of `n` students in a class:

marksAnalyzer.py

```
1.   #!/usr/bin/python
2.
3.   # Program to analyse the marks of 'n' students in a class
4.
5.   from functools import reduce
6.
7.   marks = list( map(lambda x: int(x), input("Enter the marks
of 'n' students: ").split(' ')))
8.   n = len(marks)
9.
10.  maxMarks = max(marks)
11.  minMarks = min(marks)
12.  sumMarks = reduce(lambda x,y: x+y, marks)
13.  avgMarks = sumMarks/n
14.
15.  numPasses = len(list(filter(lambda x: x>35, marks)))
16.  numFailures = n-numPasses
17.  percentPasses = numPasses/n*100
18.
19.  print("Analysed marks of {} students:".format(n))
20.  print("Minimum marks: {} Maximum marks:
{}".format(minMarks,maxMarks))
21.  print("Average marks: {}".format(avgMarks))
22.  print("Passes: {} Failures:
{}".format(numPasses,numFailures))
23.  print("Pass percentage: {}".format(percentPasses))
```

Output:

```
Enter the marks of 'n' students: 60 90 25 12 59
Analysed marks of 5 students:
Minimum marks: 12 Maximum marks: 90
Average marks: 49.2
Passes: 3 Failures: 2
Pass percentage: 60.0
```

Observation:

1. This program uses all 3 functions: `map()`, `filter()` and `reduce()`.
2. In line 7, we ask the user to enter all the marks, split it on spaces, convert the split strings to integers using map, convert the result to a list and store it in `marks`.
3. Lines 10-11 employ the built-in functions `max()` and `min()` to find the maximum and minimum marks respectively.
4. Line 12 uses the `reduce()` function to find the sum of the marks present in the list `marks`.
5. Line 15 filters the list `marks` to search for those marks that indicate a pass using the `filter()` function. Of course, we are only interested in the number of passes, which is counted by the built-in `len()` function.

11.7 List Comprehensions

A *list comprehension* is a special syntax supported by Python to quickly build lists. A list comprehension can even eliminate the need to use `map()` and `filter()` functions, providing a shorter syntax instead! Let us therefore start introducing list comprehensions as replacements to `map()` and `filter()` using the examples we saw in preceding sections.

Building a list containing element values twice those of elements in a given list:

```
>>> L=[1,6,4,9,7]
>>> M=[2*x for x in L]
>>> M
[2, 12, 8, 18, 14]
```

Converting a list of strings to a list of strings with all characters in uppercase:

```
>>> L=['Hello','World']
>>> L=[x.upper() for x in L]
>>> L
['HELLO', 'WORLD']
```

Or in fact, we could directly do the following:

```
>>> L=[x.upper() for x in ['Hello','World']]
>>> L
['HELLO', 'WORLD']
```

Here is an even shorter version that does not store the result anywhere:

```
>>> [x.upper() for x in ['Hello','World']]
['HELLO', 'WORLD']
```

Here is a dummy list comprehension that merely copies elements of a list:

```
>>> L=[1,6,4,9,7]
>>> M=[x for x in L]
>>> M
[1, 6, 4, 9, 7]
```

The previous example, though not very useful, paves way to show the next syntax of list comprehensions! Here is the replacement for the `filter()` function, to filter out the odd integers and allow only even integers of a list to pass through:

```
>>> L=[1,6,4,9,7]
>>> M=[x for x in L if x%2==0]
>>> M
[6, 4]
```

And now for the combination of `filter()` and `map()` using list comprehensions: to produce the squares of even numbers in a list:

```
>>> L=[1,6,4,9,7]
>>> M=[x*x for x in L if x%2==0]
>>> M
[36, 16]
```

How about using list comprehensions to produce a list of all primes less than 100?

```
>>> def isPrime(n):
...     for i in range(2,n):
...         if n%i==0: return False
...     else: return True
...
>>> P=[x for x in range(2,100) if isPrime(x)]
>>> P
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97]
```

Let us also add the constraint that we want to exclude those primes that end with the digit `9`:

```
>>> def isPrime(n):
...     for i in range(2,n):
...         if n%i==0: return False
...     else: return True
...
>>> P=[x for x in range(2,100) if isPrime(x) if not x%10==9]
>>> P
[2, 3, 5, 7, 11, 13, 17, 23, 31, 37, 41, 43, 47, 53, 61, 67, 71, 73,
83, 97]
```

11.7.1 List Comprehensions Returning Collections

The previous section showed how a list comprehension can be used to quickly and easily populate a list with chosen values. All the individual elements in the list were scalars – simple singular values. List comprehensions can also be used to create lists populated with collections like lists and tuples. This section shows you how.

Let's start with a list comprehension to create tuples of the form `(x,x)` with `x` ranging from `1` to `10`:

```
>>> [(x,x) for x in range(1,11)]
[(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9,
9), (10, 10)]
```

Observation::

1. The parentheses and comma in `(x,x)` are enough for Python to understand that we're dealing with a tuple. Each value generated in the list will be a tuple.
2. The first argument to `range()` is the starting number (we do not want to start from `0`).
3. The second argument to `range()` is always excluded (we want `10` to be included).

Let us modify the previous example to produce a list containing tuples of the form (x,x^2):

```
>>> [(x,x*x) for x in range(1,11)]
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64),
(9, 81), (10, 100)]
>>>
```

Let us exclude odd integers from the above list:

```
>>> [(x,x*x) for x in range(1,11) if x%2==0]
[(2, 4), (4, 16), (6, 36), (8, 64), (10, 100)]
```

Let us exclude both odd integers as well as multiples of 6 from the list:

```
>>> [(x,x*x) for x in range(1,11) if x%2==0 if not x%6==0]
[(2, 4), (4, 16), (8, 64), (10, 100)]
```

Let us work with multiple variables now: say we want to generate all permutations of coordinates of the form `(x,y)` with `x` and `y` being integers between `1` and `3` (both inclusive):

```
>>> [(x,y) for x in range(1,4) for y in range(1,4)]
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3,
3)]
```

In the above output, let's say we want to omit those coordinates that are of the form `(x,x)`:

```
>>> [(x,y) for x in range(1,4) for y in range(1,4) if not x==y]
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

Let us modify this to produce a list of coordinates of the form `(x,y)` with `x` ranging from `1` to `3` and `y` ranging from `5` to `9`:

```
>>> [(x,y) for x in range(1,4) for y in range(5,10)]
[(1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (2, 5), (2, 6), (2, 7), (2,
8), (2, 9), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9)]
```

How about generating 3D coordinates of the form (x,y,z), with x ranging from `1` to `3`, y always less than `x` and `z` always less than or equal to y?

```
>>> [(x,y,z) for x in range(1,4) for y in range(1,x) for z in
range(1,y+1)]
[(2, 1, 1), (3, 1, 1), (3, 2, 1), (3, 2, 2)]
```

While all the above examples were dealing with tuples, lists are no different – they just need to be enclosed within square brackets (`[]`):

```
>>> [[x,y,z] for x in range(1,4) for y in range(1,x) for z in
range(1,y+1)]
[[2, 1, 1], [3, 1, 1], [3, 2, 1], [3, 2, 2]]
```

11.8 Nested List Comprehensions

The previous section showed how we can use list comprehension to build a list of values. This section goes deeper, showing how we can use list comprehensions to build lists containing lists/tuples built out of list comprehensions! Or in other words, how to nest a list comprehension within another, creating *nested list comprehensions*!

Let us start with an example to create a 3x4 matrix `M` filled with the value `1` as shown in the diagram below:

1	1	1	1
1	1	1	1
1	1	1	1

```
>>> M=[[1 for j in range(4)] for i in range(3)]
>>> M
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

Observation:

1. There are 2 list comprehensions – one nested within another. The "inner" list comprehension is used for generating the different column values within a single row while the "outer" list comprehension is used for generating lists for each row.
2. The inner list comprehension (`[1 for j in range(4)]`) generates a list containing `4` elements of value `1` each. This is because each row of the matrix should contain `4` columns of value `1` each.
3. The outer list comprehension (`[[…] for i in range(3)]`) generates a list of `3` lists. This is because we want `3` rows and each row is a list of values.

Our next example will be to obtain the following matrix:

1	1	1	1
2	2	2	2
3	3	3	3

```
>>> M=[[i+1 for j in range(4)] for i in range(3)]
>>> M
[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]]
```

Observation:

1. The inner list comprehension creates a list of `4` elements with each value being `i+1`. We need `4` elements because we want `4` columns. We need the value `i+1` because we want the row number to be the content, with `i` being the row number (and since the row numbering starts from `0` in range but starts with `1` for the user, we use `i+1` instead).
2. The outer list comprehension creates `3` lists since we want `3` rows.

Our next example will be to obtain the following matrix:

1	2	3	4
1	2	3	4
1	2	3	4

```
>>> M=[[j+1 for j in range(4)] for i in range(3)]
>>> M
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
```

Observation:

1. This example is similar to the previous one, except that we substitute the column value inside the list instead of the row value (again using `j+1` since the range starts from `0` and we want to start from `1`).

Our next example will be to obtain the following matrix:

1	2	3	4
5	6	7	8
9	10	11	12

```
>>> M=[[4*i+j+1 for j in range(4)] for i in range(3)]
>>> M
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Observation:

1. It can be verified that the content of each cell can be obtained by the arithmetic expression `4*i+j+1`, where `i` is the row number starting from `0` and `j` is the column number starting from `0`.

Let us extend the previous example to also find the transpose of the matrix using nested list comprehension!

```
>>> M=[[4*i+j+1 for j in range(4)] for i in range(3)]
>>> M
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> T=[[row[i] for row in M] for i in range(4)]
>>> T
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Observation:

1. The first part (creation of the matrix `M` and it's display) is the previous example that has been reused for readability.

2. The second part creates a matrix `T` that is the transpose of the matrix `M` and displays it.
3. The outer list comprehension for creation of `T` creates `4` lists (the transpose will have `4` rows since the original matrix had `4` columns).
4. The inner list comprehension for creation of `T` is for creating each row of the transpose. Each row here will have as many columns/elements as the number of rows in the original matrix `M`. Furthermore, each element of the transpose in row `i` will have elements taken from column `i` of each row of the original matrix.

While this section focussed on matrices to demonstrate nested list comprehension, the next section concentrates on matrix operations.

11.9 Matrices

Matrices are heavily used in mathematics and programming, and programmers typically consider representing matrices as 2D arrays/lists. In Python, a matrix can be easily represented as nested lists, which we have seen in the previous section.

There are a host of operations that we would like to perform on matrices, which will be covered in the following sections, but right now we will focus on some basic aspects of matrices that will be required in the following sections, such as reading in matrices from the user, printing matrices and traversing through matrix elements. All these operations will be demonstrated by the following program:

`matrixdemo.py`

```
1.   #!/usr/bin/python
2.
3.   # Matrix demonstration
4.
5.   m,n = map(lambda x: int(x), input("Enter the order of the
matrix: ").split(' '))
6.
7.   print("Enter the matrix elements rowwise:")
8.   matrix = [list(map(lambda x: int(x),input().split(' ')))
for row in range(m)]
9.
10.  print("Here is the matrix of order {}x{}:".format(m,n))
11.  for i in range(m):
12.      for j in range(n):
13.          print("{:5}".format(matrix[i][j]),end='')
14.      print()
```

Output:

```
Enter the order of the matrix: 3 4
Enter the matrix elements rowwise:
1 2 3 4
5 6 7 8
9 10 11 12
Here is the matrix of order 3x4:
    1    2    3    4
    5    6    7    8
    9   10   11   12
```

Observation:

1. The first step while accepting a matrix is to accept the order of the matrix. This is done in line 5.
2. Each row of the matrix can be accepted as a single line and split into individual elements on spaces and converted to integers, which is done in line 8 and is repeated as many times as the number of rows. All the elements are stored as a list of lists in `matrix`.
3. The matrix can be displayed by iterating through all it's element using nested loops as shown in lines 11-13.

Lines 11-14 can be replaced by the following statements:

```
11.  for row in matrix:
12.      for value in row:
13.          print("{:5}".format(value),end=' ')
14.   print()
```

In fact, to make the code more Pythonic, lines 11-14 can be replaced by the following single statement:

```
11.  print("\n".join([" ".join(["{:5}".format(value) for value
in row]) for row in matrix]))
```

Now that we know how to accept and display matrices and how to access their individual elements, it is time to start performing operations on them.

11.9.1 Matrix Transpose

The transpose of a matrix is obtained by interchanging the rows and columns of a matrix. This, the transpose of an *m* x *n* matrix will have an order of *n* x *m*. Square matrices (whose number of rows is equal to number of columns) do not undergo any change in order, of course.

transpose.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the transpose of a matrix
4.
5.   m,n = map(lambda x: int(x), input("Enter the order of the
matrix: ").split(' '))
6.
7.   print("Enter the matrix elements rowwise:")
8.   matrix = [list(map(lambda x: int(x),input().split(' ')))
for row in range(m)]
9.   transpose = [[None]*m for i in range(n)]
10.
11.  for i in range(m):
12.      for j in range(n):
13.          transpose[j][i]=matrix[i][j]
14.
15.  print("Here is the transpose of the matrix of order
{}x{}:".format(m,n))
16.  for i in range(n):
17.      for j in range(m):
18.          print("{} ".format(transpose[i][j]),end='')
19.      print()
```

Output:

```
Enter the order of the matrix: 3 4
Enter the matrix elements rowwise:
1 2 3 4
5 6 7 8
9 10 11 12
Here is the transpose of the matrix of order 3x4:
1 5 9
2 6 10
3 7 11
4 8 12
```

Observation:

1. This program utilizes code from the previous program, `matrixdemo.py`.
2. Lines 5-8 are taken from `matrixdemo.py`, and help in reading in the matrix.
3. If the original matrix is of the order *m* x *n*, the transpose will be of the order *n* x *m*. We need to create an empty matrix of this size so that we can assign values to individual elements thereafter. This is done in line 9. The part `[None]*m` creates a list of size `m` filled with the value `None` for each element. The part `for i in range(n)` repeats this for `n` rows.
4. Lines 11-12 are similar to the part for displaying the matrix in `matrixdemo.py`, but line 13 copies an element from row `i` column `j` of the original matrix to row `j` column `i` of the transpose matrix.
5. Lines 15-19 are similar to those for displaying a matrix in `matrixdemo.py`.

Lines 11-13 can be replaced by the following statement, as covered in section 11.8:

```
11.  transpose=[[row[i] for row in matrix] for i in range(m)]
```

11.9.2 Square Matrix Diagonals

As defined earlier, a *square matrix* is one that has the same number of rows and columns. Such a matrix has 2 standard diagonals – the *primary diagonal* that extends from the top-left to the bottom-right and the *secondary diagonal* that extends from the bottom-left to the top-right. The sum of the elements that lie on the principal diagonal of a square matrix is called the *trace* of the matrix.

Here is a program to find the sum of the elements:

1. On the principal diagonal
2. Above the principal diagonal
3. Below the principal diagonal

matrix_diagonal.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the sum of all elements:
4.   # 1. On the principal diagonal
5.   # 2. Above the principal diagonal
6.   # 3. Below the principal diagonal
7.
8.   m = int(input("Enter the number of rows of the matrix: "))
9.
10.  print("Enter the matrix elements rowwise:")
11.  matrix = [list(map(lambda x: int(x),input().split(' ')))
for row in range(m)]
12.
13.  sumAbove,sumBelow,trace = 0, 0, 0
14.
15.  for i in range(m):
16.      for j in range(m):
17.          if i<j: sumAbove += matrix[i][j]
18.          elif i>j: sumBelow += matrix[i][j]
19.          else: trace += matrix[i][j]
20.
21.  print("Sum of all elements:")
22.  print("\tAbove the principal diagonal:",sumAbove)
23.  print("\tOn the principal diagonal:",trace)
24.  print("\tBelow the principal diagonal:",sumBelow)
```

Output:

```
Enter the number of rows of the matrix: 4
Enter the matrix elements rowwise:
1 2 3 4
5 6 7 8
2 2 3 3
3 3 4 4
Sum of all elements:
      Above the principal diagonal: 27
      On the principal diagonal: 14
      Below the principal diagonal: 19
```

Observation:

1. Line 8 accepts the number of rows of the matrix. Since the matrix is a square matrix, the number of columns has to be the same as the number of rows.
2. Line 13 initializes the variables `sumAbove`, `sumBelow` and `trace` to `0`. They represent the sum of all elements above, below and on the principal diagonal respectively.
3. Lines 15-16 are used to iterate through each element of the matrix.
4. Lines 17-19 utilize the fact that elements that lie on the principal diagonal have the same value for row number (`i`) and column number (`j`). Elements above the principal diagonal will have row number (`i`) less than column number (`j`), and elements below the principal diagonal will exhibit the reverse property.

11.9.3 Matrix Addition and Subtraction

Two matrices can be added or subtracted if they have the same order. The addition or subtraction is then performed on the corresponding elements. Considering that matrices will be represented as lists of lists, we can use list operations to perform the task easily!

matrix_addsub.py

```
1.   #!/usr/bin/python
2.
3.   # Matrix Addition and Subtraction
4.
5.   m,n = map(lambda x: int(x), input("Enter the order of the
matrices: ").split(' '))
6.
7.   print("Enter the elements of the first matrix rowwise:")
8.   matrix1 = [list(map(lambda x: int(x),input().split(' ')))
for row in range(m)]
9.   print("Enter the elements of the second matrix rowwise:")
10.  matrix2 = [list(map(lambda x: int(x),input().split(' ')))
for row in range(m)]
11.
12.  matrixSum = list(map(lambda rowMatrix1, rowMatrix2:
list(map(lambda a, b: a+b, rowMatrix1, rowMatrix2)), matrix1,
matrix2))
13.
14.  matrixDiff = list(map(lambda rowMatrix1, rowMatrix2:
list(map(lambda a, b: a-b, rowMatrix1, rowMatrix2)), matrix1,
matrix2))
15.
16.  print("Sum of the matrices:")
```

```
17.  for i in range(m):
18.      for j in range(n):
19.          print("{} ".format(matrixSum[i][j]),end='')
20.      print()
21.
22.  print("Difference of the matrices:")
23.  for i in range(m):
24.      for j in range(n):
25.          print("{} ".format(matrixDiff[i][j]),end='')
26.      print()
```

Output:

```
Enter the order of the matrices: 2 3
Enter the elements of the first matrix rowwise:
1 2 3
4 5 6
Enter the elements of the second matrix rowwise:
2 2 2
3 3 3
Sum of the matrices:
3 4 5
7 8 9
Difference of the matrices:
-1 0 1
1 2 3
```

Observation:

1. Line 5 stores the order of the matrices, just like our earlier programs on matrices. Both the input matrices will have the same order.
2. Lines 7-8 receive the elements of the first matrix and lines 9-10 similarly receive the elements of the second matrix.
3. Line 12 adds the 2 matrices (`matrix1` and `matrix2`) and stores the result in `matrixSum` using list operations. We use map to map each element of both the matrices using a lambda function. Remember that each element of the matrices is actually a list representing a complete row in the matrix. The corresponding rows are then passed on to another mapping, again using a lambda function which adds the corresponding elements of the two given rows and gives us a resultant row.
4. Line 14 performs a very similar operation to find the difference between the matrices – the only difference being a subtraction instead of an addition.
5. The matrices are then displayed using lines 16-20 and 22-26.

11.9.4 Matrix Multiplication

A matrix `a` of order `m` x `n` can be multiplied with a matrix `b` of order `p` x `q` only if `n`=`p`. The resultant matrix then will have an order of `m` x `q`. An element of the resultant matrix `c` at row `i` column `j` will have the value calculated by the following formula:

$$c_{ij} = a_{i0}*b_{0j} + a_{i1}*b_{1j} + a_{i2}*b_{2j} + ... + a_{in}*b_{nj}$$

matrix_mul.py

```
1.   #!/usr/bin/python
2.
3.   # Matrix Multiplication
4.   from functools import reduce
5.   import sys
6.   m,n = map(lambda x: int(x), input("Enter the order of the
first matrix: ").split(' '))
7.   p,q = map(lambda x: int(x), input("Enter the order of the
second matrix: ").split(' '))
8.
9.   if not n==p:
10.      print("Matrices are not multiplicable")
11.      sys.exit()
12.
13.  print("Enter the elements of the first matrix rowwise:")
14.  matrix1 = [list(map(lambda x: int(x),input().split(' ')))
for row in range(m)]
15.  print("Enter the elements of the second matrix rowwise:")
16.  matrix2 = [list(map(lambda x: int(x),input().split(' ')))
for row in range(p)]
17.
18.  matrixProduct = [ [ reduce(lambda x,y:x+y, [matrix1[i]
[k]*matrix2[k][j] for k in range(n)]) for j in range(q)] for i
in range(m)]
19.
20.  print("Product of the matrices:")
21.  for i in range(m):
22.      for j in range(q):
23.          print("{} ".format(matrixProduct[i][j]),end='')
24.      print()
```

Output:

```
Enter the order of the first matrix: 2 3
Enter the order of the second matrix: 2 3
Matrices are not multiplicable
```

```
Enter the order of the first matrix: 2 3
Enter the order of the second matrix: 3 4
Enter the elements of the first matrix rowwise:
1 2 3
4 5 6
Enter the elements of the second matrix rowwise:
1 2 3 4
5 6 7 8
2 3 4 5
Product of the matrices:
17 23 29 35
41 56 71 86
```

Observation:

1. Lines 6 and 7 accept the order of the matrices. Line 9 checks if the matrices are multiplicable and terminates the program in line 11 if they aren't multiplicable.
2. Lines 14 and 16 accept the elements of the 2 matrices, while line 18 performs the actual matrix multiplication using nested list comprehensions.

11.10 Powerful Scripts

This chapter has introduced many powerful concepts involving data and data structures. Let us apply these concepts to solve certain real-world problems in this section.

Let us begin this section with an application related to evaluating arithmetic expressions. You might be surprised that the approach is not direct as many might think it is, but not too complicated either!

11.10.1 Checking an Infix Arithmetic Expression

We start with a script to check whether an infix arithmetic expression (involving single-character operands and the binary operators "+", "-", "*" and "/") is correct.

An infix arithmetic expression is one in which the binary operators are used between the 2 operands on which they operate, as shown in the syntax below:

```
operand1 operator operand2
```

The following checks can be made on infix arithmetic expressions to ensure they are correct:

1. Operands are assumed to be single-character lowercase alphabets
2. Operators are assumed to be "+", "−", "*" and "/" only
3. All operators are assumed to be binary
4. Empty expressions are not allowed
5. Two operands cannot appear in succession "ab" is incorrect)
6. Two operators cannot appear in succession ("a**+*b" is incorrect), except when one of them is a parentheses ("a**+(**b)" and "(a**)+**b" are correct). However, "(a**+)**" and "**(***c)" are still incorrect
7. The expression cannot start with an operator, except when the operator is open parentheses ("*c" is incorrect)
8. The expression cannot end with an operator, except when the operator is close parentheses ("c*" is incorrect)
9. The number of open parentheses should be the same as the number of close parentheses ("(a" is incorrect)
10. When scanning the expression from left to right, a closing parentheses should not appear before it's corresponding open parentheses ("a**)(**" is incorrect though the parentheses are balanced)

infixChecker.py

```
1.   #!/usr/bin/python
2.
3.   # Infix Expression Checker
4.
5.   import sys
6.
7.   def error(msg):
```

```
8.       print(msg)
9.       sys.exit()
10.
11.  infix = input("Enter an infix expression:")
12.  mode = prevMode = ''
13.  parentheses=0
14.
15.  for i in infix:
16.      if i.isalpha():
17.          mode='operand'
18.          if prevMode == 'operand': error("Found two
operands in succession:")
19.      elif i == '(':
20.          if prevMode == 'operand': error("Found '(' after
operand")
21.          parentheses = parentheses+1
22.      elif i == ')':
23.          if prevMode == 'operator': error("Found ')' after
operator")
24.          parentheses = parentheses-1
25.          if parentheses < 0: error("Improper nesting of
parentheses")
26.      elif i in ('+','-','*','/'):
27.          mode='operator'
28.          if not prevMode == 'operand': error("Found
operator in wrong place")
29.      else:
30.          error("Invalid character found!")
31.
32.      prevMode = mode
33.
34.  if len(infix)==0: error("No expression")
35.  if mode=='operator': error("Wrong termination of
expression")
36.  if not parentheses==0: error("Improper parentheses")
37.
38.  print("Correct!")
```

Observation:

1. We accept the infix expression from the user in line 11 and finally print "`Correct!`" if the infix expression was valid in line 38.
2. We iterate through each character of the given infix expression in line 15, and test if for various possibilities within the loop, taking relevant action.
3. We have a notion of current mode (`mode`) and previous mode (`prevMode`).

The current mode is determined based on the character that is seen in the current iteration, and this becomes the previous mode in the next iteration (line 31). The values for `mode` (and `prevMode`) will either be `"operand"` or `"operator"`, as assigned in lines 17 and 27. Both these variables start with a null value in line 12.

4. Line 16 checks if the current character is alphabetic, implying that it is an operand. We set `mode` to `"operand"` to reflect this observation in line 17. We check for successive operands (which should generate an error) in line 18.
5. We have defined an `error` function (line 7-9) to handle display of suitable messages and termination of the script.
6. Line 26 checks for the supported operators. We set `mode` to `"operator"` on encountering this. We check for successive operators in line 28, generating an error if necessary.
7. We check for parentheses in lines 19 and 22. As far as balancing the parentheses are concerned, we can maintain a count of opened parentheses encountered so far (variable `parentheses`, initialised to `0` in line 13). Each time we encounter an opening parentheses, we increment this count (line 21) and each time we encounter a closing parentheses, we decrement this count (line 24). At no time should this count be negative (line 25) and finally the count must be `0` (line 36).
8. We also check for the correct placement of these parentheses. An opening parentheses cannot start after an operand (line 20) while a closing parentheses cannot be placed after an operator (line 23).
9. No other characters are expected, and this case is handled in lines 29-30.
10. Line 34 checks for the special case where no expression is provided.
11. Line 35 verifies that the expression does not end with an operator.

11.10.2 Infix to Postfix Conversion

While we humans find it convenient to deal with the infix notation for representation of an expression, computers will find it more convenient to deal with prefix and postfix notations.

In the *prefix* notation, a binary operator always precedes it's 2 operands, which immediately follow the operator as shown in the syntax below:

```
operator operand1 operand2
```

In the *postfix* notation, a binary operator always succeeds it's 2 operands, which immediately precede the operator as shown in the syntax below:

```
operand1 operand2 operator
```

Here are a few examples of infix arithmetic expressions and their postfix equivalent:

No.	Infix Notation	Postfix Notation
1	a	a
2	a+b	ab+
3	a+b*c	abc*+
4	a*b+c	ab*c+
5	a*(b+c)	abc+*
6	(a+b)*(c+d)	ab+cd+*

Observation:

1. As long as no operators are involved, both infix and postfix expressions looks the same, as illustrated in example 1. For lengthier expressions, do note that the only difference is in the placement of operators, but the operands remain in the same order within the expression!
2. Example 2 shows a basic arithmetic expression
3. In the postfix expression of example 3, the `*` operator works on `b` and `c`, while the `+` operator works on `a` and the result of `b*c`.
4. In the postfix expression of example 4, the `*` operator works on `a` and `b`, while the `+` operator works on the result of `a*b` and `c`.
5. In the postfix expression of example 5, the `+` operator works on `b` and `c`, while the `*` operator works on `a` and the result of `b+c`.
6. In the postfix expression of example 6, the first `+` operator works on `a` and `b`, the second `+` operator works on `c` and `d`, while the `*` operator works on the results of the previous 2 operations (`a+b` and `c+d`).
7. We observe that unlike infix expressions where we need to evaluate operators based on their precedence (and associativity), in postfix expressions, once framed we can evaluate easily without considering anything else.
8. We also observe, for the same reason mentioned above, that we don't need or use parentheses in postfix expressions.

Advantages of the postfix notation:

1. We need not scan the expression for considering the operator precedence and associativity. We can easily evaluate from left to right.
2. In infix expressions, parentheses are frequently used to alter operator precedence. In postfix expressions, we don't use parentheses at all since there is no precedence to deal with once the expression has been carefully framed.
3. For the above reasons, we find it easier to write programs to evaluate postfix expressions than infix expressions.
4. The above reasons also justify the use of prefix over infix, but what makes postfix slightly better than prefix is that we can directly use a stack to evaluate postfix expressions (as will be seen later in this section).

While evaluation of postfix expressions is very simple, conversion of infix expressions to postfix is slightly more complicated since we will have to deal with precedence of operators in the infix expression.

The next program converts a valid infix arithmetic expressions to it's postfix equivalent. Of course, the program makes the same assumptions as the previous one – that the operands are single-character variables and the operators are binary "+", "–", "*" and "/".

The basic logic used in the program is as follows (using a stack for storing operators temporarily):

1. Accept an infix expression from the user and scan it character by character from left to right.
2. If the character is an operand, transfer it to the output string as it is (since operands are always in place and only operator positions are to be changed).
3. On encountering an opening parentheses, push it into an operator stack (since we are interested in encountering the corresponding closing parentheses to process it completely).
4. On encountering a closing parentheses, pop all operators from the operator stack to the output string till we encounter an opening parentheses in the stack, which is discarded. This will ensure that the entire expression within the parentheses is processed.
5. On encountering an operator, we pop all operators from the stack that have a higher precedence than the operator currently being processed, transferring them to the output string. We then push the current operator into the operator stack. This will ensure that operators are presented in the right order.
6. After processing the entire infix expression, we pop out the remaining operators from the operator stack and transfer them to the output string.

infix2postfix.py

```
1.   #!/usr/bin/python
2.
3.   # Infix to Postfix Converter
4.
5.   infix = input("Enter an infix expression:")
6.   postfix = ''
7.
8.   precedence = {'(':0,'+':1,'-':1,'*':2,'/':2}
9.   operators = []
10.
11.  for i in infix:
12.      if i.isalpha(): postfix += i
13.      elif i == '(': operators.append(i)
14.      elif i == ')':
15.          while not operators[-1] == '(':
16.              postfix += operators.pop()
17.          operators.pop()
18.      else:
19.          while len(operators)>0 and precedence[operators[-
1]]>=precedence[i]:
20.              postfix += operators.pop()
21.          operators.append(i)
22.
23.  operators.reverse()
24.  postfix += ''.join(operators)
25.
26.  print(postfix)
```

Observation:

1. We accept the infix expression (`infix`) from the user in line 5 and initialise the output string (`postfix`) in line 6.
2. We maintain a stack of operators (`operators`) in line 9.
3. The precedence of operators is recorded in the form of a dictionary (`precedence`) in line 8. The key is the operator symbol and the value is the numeric precedence, with higher values indicating higher precedence. Thus, "`*`" and "`/`" have the same precedence, which is higher than the precedence of "`+`" and "`-`". The open parentheses has been added here with least precedence since that also can exist in the stack and should not be popped out due to any operator other than closing parentheses.
4. We iterate through the characters within the infix string in line 11.
5. Operands are transferred to the output string in line 12.

6. Opening parentheses are unconditionally pushed into the operator stack in line 13.
7. On encountering closing parentheses, all operators are popped from the stack to the output string in lines 15-16, till the opening parentheses operator is found in the stack, which is then discarded in line 17.
8. On encountering an operator, all operators with higher precedence are popped out from the operator stack to the output string in lines 19-20. The current operator is then pushed into the operator stack in line 21.
9. After processing the infix expression, the rest of the operators in the operator stack need to popped out into the output string. An easier operation to achieve the same effect is reversing the contents of the stack, joining them all together to form a string and appending the string to the output string, as done by lines 23-24.
10. The output string is finally printed in line 26.

11.10.3 Evaluating Postfix Arithmetic Expressions

Now that we know how to convert an infix arithmetic expression into it's postfix equivalent, let us focus on how to evaluate these postfix arithmetic expressions. Once again, we assume that the postfix arithmetic expression given to us is valid and is composed of single-character case-sensitive variables and the binary operators "+", "-", "*" and "/".

Steps for evaluating Postfix Arithmetic Expressions:

1. Scan the given postfix expression from left to right.
2. When an operand is encountered, check if we already know it's value. If not, ask the user for it's value and note it. The value of the operand is pushed on to a stack.
3. When an operator is encountered, it works on the topmost 2 values of the stack, with the topmost value of the stack being the second operand. The result of the operation is pushed on to the stack.
4. After the entire expression has been processed, the value of the expression will be the only element in the stack. We simply need to pop it out and display it.

Note that unlike the previous program where we had a stack of operators, in this program we have a stack of operand values!

postfixEval.py

```
1.   #!/usr/bin/python
2.
3.   # Postfix Expression Evaluator
4.
5.   postfix = input("Enter a postfix expression:")
6.   stack = []
7.   operands = {}
8.
9.   for i in postfix:
10.      if i.isalpha(): # Operand
11.          if i not in operands:
12.              operands[i] = int(input("Enter the value of
{}: ".format(i)))
13.          stack.append(operands[i])
14.      elif i=='+': stack.append(stack.pop() + stack.pop())
15.      elif i=='*': stack.append(stack.pop() * stack.pop())
16.      elif i=='-': stack.append(-(stack.pop() -
stack.pop()))
17.      elif i=='/': stack.append(1/(stack.pop() /
stack.pop()))
18.
19.  print("Value:",stack.pop())
```

Observation:

1. We accept the postfix expression (`postfix`) in line 5.
2. We create a stack (`stack`) to store intermediate expression values in line 6.
3. We store all variables (`operands`) with their corresponding values in a dictionary operands in line 7.
4. We iterate through the postfix expression character by character from left to right in line 9.
5. Line 10-13 deal with operands. If the operand is not present as a key in the dictionary `operands`, we ask the user to enter the value of the same and store the operand with it's associated value in `operands` and finally push the value of the operand on to the stack `stack`.
6. For each of the operators, we pop out 2 values from the stack `stack`, operate on them and push the result back. This is done in lines 14-17.
7. Finally, in line 19, we pop the result from the stack `stack` and display it.

11.10.4 Evaluating Infix Arithmetic Expressions

From the previous programs, we are now confident of evaluating an arithmetic expression given to us in the infix notation since we know how to convert it to it's postfix equivalent and how to evaluate a postfix arithmetic expression. Let us combine this all together to have a single program that evaluates an infix arithmetic expression by converting it to postfix and evaluating it.

infixEval.py

```
1.   #!/usr/bin/python
2.
3.   # Infix Expression Evaluator
4.   import sys
5.   def error(msg):
6.       print(msg)
7.       sys.exit()
8.
9.   def infixChecker(infix):
10.      mode = prevMode = ''
11.      parentheses=0
12.
13.      for i in infix:
14.          if i.isalpha():
15.              mode='operand'
16.              if prevMode == 'operand': return "Found two
operands in succession:"
17.          elif i == '(':
18.              if prevMode == 'operand': return "Found '('
after operand"
19.              parentheses = parentheses+1
20.          elif i == ')':
21.              if prevMode == 'operator': return "Found ')'
after operator"
22.              parentheses = parentheses-1
23.              if parentheses < 0: return "Improper nesting
of parentheses"
24.          elif i in ('+','-','*','/'):
25.              mode='operator'
26.              if not prevMode == 'operand': return "Found
two operators in succession"
27.          else:
28.              return "Invalid character:"
29.
30.          prevMode = mode
31.
32.      if len(infix)==0: return "No expression"
33.      if mode=='operator' and not i[-1] == ')': return
```

```
"Wrong termination of expression"
34.      if not parentheses==0: return "Improper parentheses"
35.
36.
37.  def infix2postfix(infix):
38.      postfix = ''
39.
40.      precedence = {'(':0,'+':1,'-':1,'*':2,'/':2}
41.      operators = []
42.
43.      for i in infix:
44.          if i.isalpha(): postfix += i
45.          elif i == '(': operators.append(i)
46.          elif i == ')':
47.              while not operators[-1] == '(':
48.                  postfix += operators.pop()
49.              operators.pop()
50.          else:
51.              while len(operators)>0 and
precedence[operators[-1]]>=precedence[i]:
52.                  postfix += operators.pop()
53.              operators.append(i)
54.
55.      operators.reverse()
56.      postfix += ''.join(operators)
57.      return postfix
58.
59.  def postfixEval(postfix):
60.      stack = []
61.      operands = {}
62.
63.      for i in postfix:
64.          if i.isalpha(): # Operand
65.              if i not in operands:
66.                  operands[i] = int(input("Enter the value
of {}: ".format(i)))
67.              stack.append(operands[i])
68.          elif i=='+': stack.append(stack.pop() +
stack.pop())
69.          elif i=='*': stack.append(stack.pop() *
stack.pop())
70.          elif i=='-': stack.append(-(stack.pop() -
stack.pop()))
71.          elif i=='/': stack.append(1/(stack.pop() /
stack.pop()))
72.
73.      return stack.pop()
```

```
74.
75.  infix = input("Enter an infix expression:")
76.
77.  msg = infixChecker(infix)
78.  if not msg is None: error(msg)
79.  postfix = infix2postfix(infix)
80.  value = postfixEval(postfix)
81.  print("Value:",value)
```

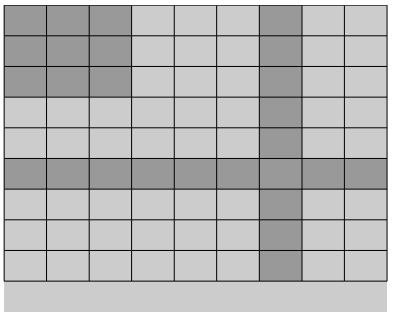
Output:

```
Enter an infix expression:a+b*(c-d)
Enter the value of a: 10
Enter the value of b: 2
Enter the value of c: 8
Enter the value of d: 5
Value: 16
```

Observation:

1. We accept an infix arithmetic expression from the user in line 75.
2. We check for the validity of the infix expression in lines 77-78 using the function `infixChecker` defined in line 9.
3. If it is valid, we proceed to convert the infix arithmetic expression into it's postfix equivalent in line 79 using the function `infix2postfix` defined in line 37.
4. After conversion, we proceed to evaluate the postfix expression in line 80 using the `postfixEval` function defined in line 59.
5. We finally print the value of the expression in line 81.

11.10.5 Checking Sudoku Solutions



Sudoku is a logical puzzle that involves putting in digits from 1-9 in a 9x9 grid of cells in such a way that each row, each column and each of the 9 3x3 grids as shown in the diagram below comprise of all digits from 1 to 9. This of course necessitates that no digit can appear twice in any row, column or grid.

We can definitely write a Python script to solve a given Sudoku problem, but that would be a little too complicated to be explained in a book like this. We

can however write a Python script to check whether a purported solution is correct or not, and that would be a little too simple to do in Python!

Here is the Pythonic logic for the solution that can get the job done in very few lines of code:

1. We accept the input from the user (9 rows of 9 digits each) and store it as a list of strings – with the list containing a string for each row and each string containing the 9 digits of that row.
2. We split the logic into 3 parts – row check (checking that each row contains digits 1-9), column check (checking that each column contains digits 1-9) and grid check (checking that each 3x3 grid contains digits 1-9).
3. We use a Boolean variable to indicate whether the solution is correct so far. The moment we realise that any check fails, we set the Boolean variable to `False`. We therefore start with the value of the Boolean variable being `True`.
4. For row check, we take the string pertaining to each row, split it into it's individual characters, sort them in ascending ASCII order, and join them back again. If the row's contents had all digits from 1 to 9, the string would now be "`123456789`"!
5. For column check, we employ a similar strategy, but have to extract the corresponding column element from each row to obtain the column contents.
6. For grid check, we similarly have to extract the contents across multiple rows and multiple columns to obtain the contents of a single grid and employ the same logic as above.

sudokuChecker.py

```
1.   #!/usr/bin/python
2.
3.   # Sudoku Solution Checker
4.   row = []
5.
6.   for i in range(9): row.append(input())
7.
8.   ok=True
9.
10.  # Row Check
11.  for i in range(9):
12.      if not ''.join(sorted(row[i])) == '123456789':
ok=False
13.
14.  # Column Check
15.  for j in range(9):
```

```
16.      if not ''.join(sorted([x for i in range(9) for x in
row[i][j]])) == '123456789': ok=False
17.
18.  # Grid Check
19.  for g in range(9):
20.      if not ''.join(sorted([x for i in range(3) for j in
range(3) for x in row[g//3*3+i][g%3*3+j]])) == '123456789':
ok=False
21.
22.  if ok: print("Correct!")
23.  else: print("Wrong!")
```

Output:

```
142576389
796384251
538912674
974235168
251698743
863741925
427153896
389467512
615829437
Correct!
```

Output:

```
142576389
427153896
796384251
538912674
974235168
251698743
863741925
389467512
615829437
Wrong!
```

Observation:

1. First of all, do note that in the first output, the given input is indeed a valid solution. In the second output, we have inserted the contents of row 7 at row 2, thereby making it an invalid solution due to grid check.
2. As suggested in the logic above, we accept the input from the user and store it in a list `row` in lines 4-6, use a Boolean flag `ok` initialised to `True` in line 8 and then proceed with row check (lines 10-12), column check (lines 14-16) and grid check (lines 18-20).
3. Finally in lines 22-23, we print a suitable message depending the the value of the Boolean variable `ok`.
4. Row check: In line 12, we take a single string, sort it (which ends up sorting the characters within the string), join the characters together and check whether it matches the string "`123456789`".
5. Column check: In line 16, we use a list comprehension to create a list of all those elements that lie in a particular column, sort the list using `sorted`, join the characters together to form a string using `join`, and compare the string against "`123456789`".
6. Grid check: Each grid's elements will be found in `row[i][j]`, where
    1. `g`, the grid number, ranges from `0` to `8`
    2. `i` ranges from `g//3 * 3` to `g//3 * 3 + 3` (considering only quotient for division)
    3. `j` ranges from `g%3 * 3` to `g%3 * 3 + 3` (considering only remainder)
7. In line 20, we use a list comprehension to create a list having all the elements of grid `g`, sort them, join them to form a string and compare against "`123456789`".

11.11 Questions

1. Explain how stacks can be implemented in Python with examples to show the various operations.
2. Explain how queues can be implemented in Python with examples to show the various operations.
3. Write short notes on:
    1. The `map()` function
    2. The `filter()` function
    3. The `reduce()` function
4. Explain how complex lists can be constructed easily using list

comprehensions with examples.

5. Explain how matrices can be represented in Python.

11.12 Exercises

1. Write a program to convert temperature from Celsius to Fahrenheit using map function.
2. Write the above program using lambda expression.
3. Write a program to convert a tuple of angles into a list of tuples with each tuple containing the sine and cosine of an angle.
4. Write a program to filter out the odd elements of the Fibonacci series for the first n terms.
5. Write a program to find the highest number in a given list using reduce function.
6. Write a program to find the sum of all the elements of a list using lambda expression and reduce function.
7. Write a program to find the product of all the elements of a list using lambda expression and reduce function.
8. Write a program to print the sum of all the numbers from 1 to 50 using lambda expression and reduce function.
9. Write a program to create a list of numbers between 1 and 50 that are neither divisible by 2 nor by 3 using filter.
10. Write a program to find all the palindromes from a list of strings.
11. Write a program to concatenate a list of strings to make a sentence using reduce.
12. Write a program to generate the square values of element of a given list using list comprehension.
13. Write a program to extract all vowels present in a given string using list comprehension.
14. Write a program to flatten a matrix (convert rows of elements into a single list of elements) using list comprehension.
15. Write a program to extract all the digits from a given string using list comprehension.
16. Write a program to generate the transpose of a matrix using list comprehension.

SUMMARY

- Stacks can be implemented using list.append() and list.pop() methods.
- Queues can be implemented using deque.append() and deque.popleft() methods.
- The map() function maps each element of a sequence to another using a mapping function. It can also be used to map multiple corresponding elements of sequences into a single corresponding element of a target sequence using a mapping function.
- The filter() function filters a sequence using a filtering function, allowing only certain values to pass through. As a special case, if no filtering function is provided, all True values pass through and False values are filtered out. The itertools.filterfalse() function does the reverse!
- The reduce() function reduces a sequence into a single value using a reducing function which takes in 2 parameters - an accumulated value (0 by default) and an element from the given sequence.
- List comprehensions are powerful mechanisms to create and populate lists with required values. They eliminate the need for elaborate lines of code to populate lists!

12 OOP in Python

In this chapter you will be able to:

- ☑ Learn the fundamental concepts of Object Oriented Programming.
- ☑ Define classes and instantiate them. Define and use Instance Variables, Class Variables, Instance Methods and Class Functions. Define and use Constructors and Destructors.
- ☑ Implement inheritance and make use of dynamic polymorphism.
- ☑ Use Attribute Handling, Magic Functions and Operator Overloading.

OOP IN PYTHON

12.1 Overview of OOP Principles

Object Oriented Programming (*OOP*) is a programming paradigm – a way of looking at problems and designing solutions. The main goal behind OOP is to model the real world within the software so that we have a parallel reality, making it easier for us to relate code to the real world. This programming paradigm essentially uses the following 8 principles to achieve this goal:

1. Classes
2. Objects
3. Data Encapsulation
4. Data Hiding
5. Data Abstraction
6. Polymorphism
7. Inheritance
8. Message Passing

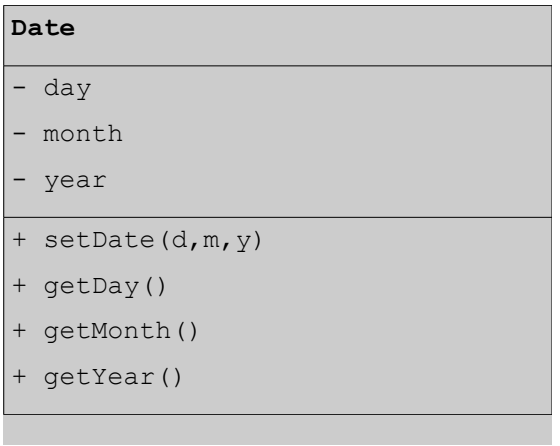
We therefore will briefly cover these 8 principles before proceeding with how OOP can be used in Python.

12.1.1 Class

A *class* can be defined as a design according to which objects can be later instantiated.

The starting point of Object Oriented Modelling is the class. The class is the design of an entity that exists in the real world. This design comprises of attributes (everything an entity has) and behaviour (everything an entity can do).

As an example, here is a diagrammatic representation of a `Date` class (technically using UML for the class representation) to represent a calendar date:



Note:

1. For now, it is sufficient to just note that `Date` is the name of a class, which has 3 attributes – `day`, `month` and `year` – and 4 member functions – `setDate()`, `getDay()`, `getMonth()` and `getYear()`.
2. Thus, the 3 pieces within the above UML diagram are: class name, data members and member functions.
3. The "−" prefix denotes that we want to keep these members private (section 12.4.6.2)
4. The "+" prefix similarly denotes that we want to keep these members public (again, section 12.4.6).

12.1.2 Object

An *object* is an instance of a class.

Once a class has been completely designed, multiple individual instances of the class can be created just as how once a car has been designed, multiple cars can be manufactured of the same design. These objects are identical to each other in terms of their design and yet independent of each other in terms of the values of their attributes. Thus, the different cars that are manufactured out of the same design can have different values for their attributes like their colour. The behaviour of all objects of the same class have to be identical, however. Thus, objects of a class typically have the same attributes and behaviour, but can differ from each other in the values of their attributes.

In our `Date` example, `day`, `month` and `year` are attributes whereas `setDate()`, `getDay()`, `getMonth()` and `getYear()` are behaviour. Multiple `Date` objects can have different values for `day`, `month` and `year`, thereby representing possibly different dates, but will have identical functionality.

12.1.3 Data Encapsulation

Data Encapsulation refers to the encapsulation of data and the code that acts on the data into a single unit. Since a class packs together attributes (data) and behaviour (code that acts on the data), we claim that classes make data encapsulation possible.

Data Encapsulation is important in the programming world as it provides an equal status for code and data.

In our `Date` example, the `Date` class encapsulates both the data (`day`, `month` and `year`) and the code that acts on the data (`setDate()`, `getDay()`, `getMonth()` and `getYear()`).

12.1.4 Data Hiding

An unwritten law in OOP is that all objects have to relate to reality at all times. Since an object in an instance of a class and a class is the design of an entity in the real world, we expect every object in our program to represent a valid real-world entity at all times. This law also entails ensuring that the attributes of an object are never meddled with in a way that it represents something absurd in reality. One of the ways of enforcing this is by hiding data that belongs to an object from the external world so that no one can accidentally tamper with it – a concept called *Data Hiding*.

In our `Date` example, we could try to hide the attributes `day`, `month` and `year` so as to prevent the users from tampering with them directly. This would prevent the user from, say, setting an invalid date like `40/30/2016`! We will instead provide functions for setting and getting the date, with validations enforced as necessary.

12.1.5 Data Abstraction

Data Abstraction refers to the hiding of implementation details while revealing a simple interface. The first advantage of Data Abstraction is that it makes the object easy to use as the users need to only be aware of the interface and need not know any implementation details. Secondly, when the implementation is insulated from the usage via the interface, it makes it possible to later change the implementation without affecting the usage by keeping the interface consistent. This adds a lot of value in practical programming, especially during the maintenance of the software.

In our `Date` example, we have provided a simple interface to set and get the date without the user having to know how we end up storing the date within the object.

Data Hiding and Data Abstraction are very much related but are not exactly the same! Both concentrate on hiding something, but with very different intentions! While Data Hiding concentrates on hiding data in order to help maintain the integrity of the object, Data Abstraction concentrates on hiding implementation details in order to simplify the usage of the object.

12.1.6 Polymorphism

The term *polymorphism* in general means multiple forms of the same entity. In OOP, it means performing the same logical operation by choosing from multiple implementations appropriately. The two forms of polymorphism that we are going to see actively in Python are:

1. *Operator overloading* – performing the same logical operation using an operator in different ways (using different implementation) depending on the type of the invoking operand
2. *Dynamic polymorphism* – performing the same logical operation in different ways (using different implementation) depending on the type of the invoking object.

In the `Date` example, an example of operator overloading would be providing an implementation for addition of days to a given date, resulting in a new Date object.

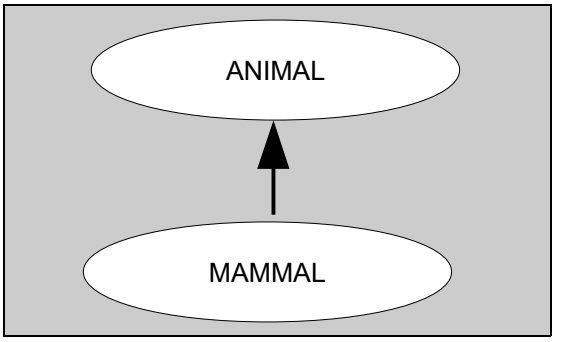
12.1.7 Inheritance

Inheritance is the mechanism wherein a class acquires all the features and properties of another class (or classes). Inheritance has the following uses:

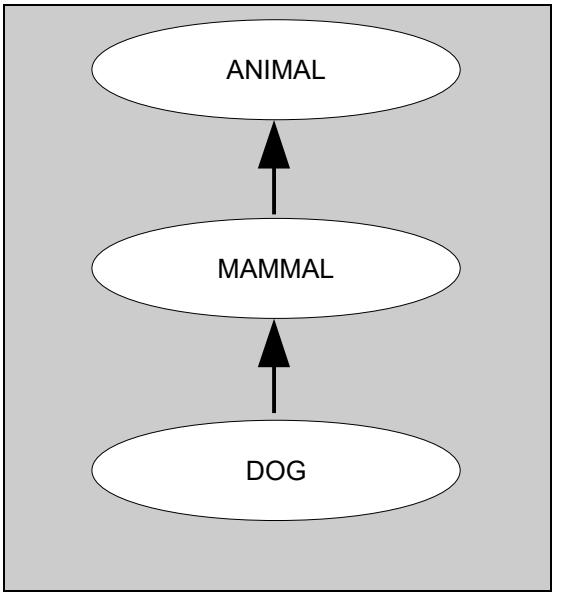
1. **Re-usability** – it helps reduce effort by reusing existing code.
2. **Extensibility** – it helps add new code or apply changes without tampering with existing code.
3. **Compartmentalisation** – it helps manage code better by compartmentalising classes.

Structurally, there are multiple types of inheritance:

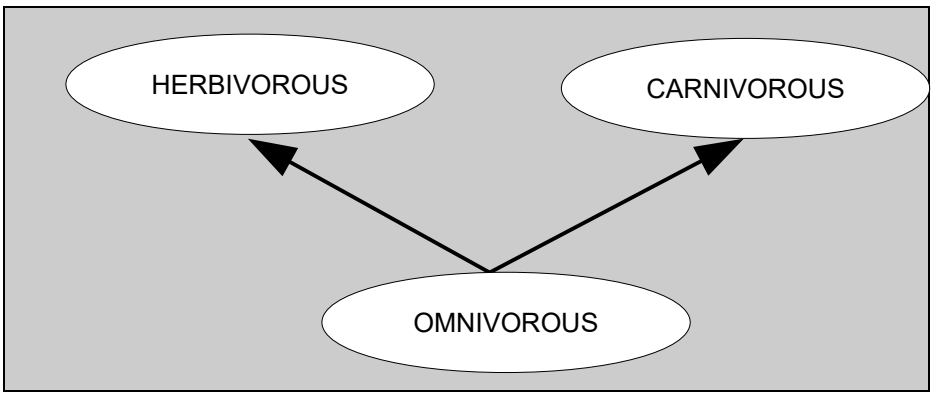
1. **Simple inheritance** – when a class inherits from another class.



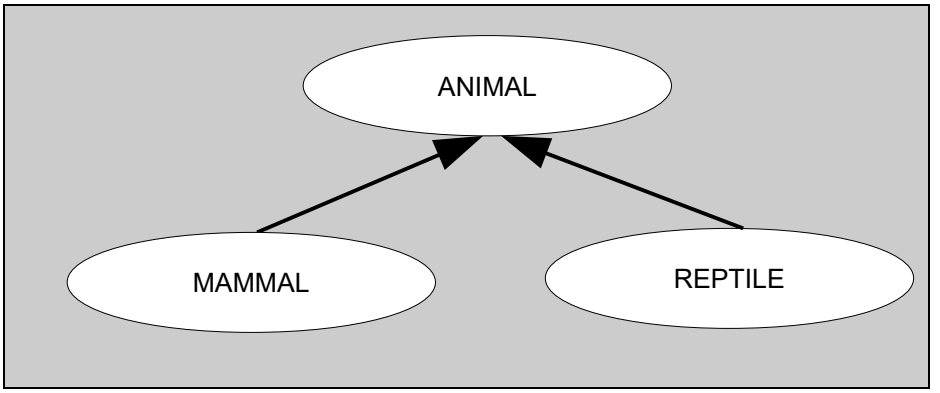
2. **Multi-level inheritance** – when a class inherits from a class that inherits from another class, thereby forming an inheritance chain.



3. **Multiple inheritance** – when a class inherits from multiple classes.



4. **Hierarchical inheritance** – When multiple classes inherit from a single class.



Inheritance is an important concept as dynamic polymorphism is totally dependent on it. In our `Date` example, we can inherit custom date format classes like `USDate` and `UKDate` from the `Date` class, adding only as much functionality as needed to support the date format.

12.1.8 Message Passing

In OOP, programs are expected to document class structures, create objects at runtime and permit these objects to communicate with each other at runtime. This inter-object communication between objects at runtime is called *message passing* and is implemented typically by making function calls, with the arguments passed being technically called the *message*.

12.2 Defining Classes

Designing a class is the first step in OOP. The simplest syntax of defining a class in Python is shown below:

```
class className:
    statements
```

The statements within a class can be any of the following:

1. Blank lines
2. Comments
3. Class variables
4. Class functions

We will cover class variables in section 12.4.3 and class functions in section 12.4.4. For now, let us start with an empty class. Since at least 1 statement is compulsory within a class definition (similar to how conditions, loops and functions need to have at least 1 statement within their body), we can use the `pass` statement to get away without causing any effect:

```
class Date:
      pass
```

Just as how a function definition can optionally start with a documentation string, even class definitions can contain documentation strings, which are then accessible using the special class variable `__doc__`:

```
>>> class Date:
...     """This is our first implementation of the Date class"""
...     pass
...
>>> Date.__doc__
'This is our first implementation of the Date class'
```

We will add more code into the class definition as we learn more features in Python.

12.3 Instantiating Classes

Once our class definition is ready, we can instantiate it and create objects. Each object will have everything documented in the class definition. In our `Date` example, since the class is empty, even the objects will be almost empty. Do note that the documentation string which was part of the class is now also accessible via the object.

The simplest syntax for instantiating a class and creating an object and storing it's reference in a variable is shown below:

```
var=ClassName()
```

This is how we can instantiate our `Date` class:

```
>>> d=Date()
>>> d
<__main__.Date object at 0x7f592e29fb50>
>>> type(d)
<class '__main__.Date'>
>>> d.__doc__
'This is our first implementation of the Date class'
```

As can be seen, the `__doc__` member of the class is accessible through the object `d` too.

12.4 Instance Variables, Class Variables, Functions and Methods

The attributes present in a class are called *Class Variables* and the functionality provided by the class using functions are called *Class Functions*.

Class variables belong to the class and are accessible using any object of the class too. In other words, class variables could be said to be shared by objects of that class.

Objects can have their own independent variables called *Instance Variables*.

The functions defined in a class are *Class Functions* and as such belong to the class and are also accessible and invokable by any object of that class, with the object playing no other role apart from identifying the class to which the Class Function belongs. When these functions work on a per-object basis (by keeping track of the invoking object and implicitly working on it), they are called *Methods*.

This section focuses on these 4 concepts:

1. Instance Variables
2. Instance Methods
3. Class Variables
4. Class Functions

12.4.1 Instance Variables

Instance Variables are variables that belong to an object.

> **For C/C++/Java programmers:**
>
> Unlike languages like C++ and Java where the class definition specifies the instance variables, in Python the class can only specify the Class Variables (which are like `static` data members in C++ and `static` fields in Java).

Python uses the Perl style, wherein the object can create whatever instance variables it desires. In fact, it is possible for different objects of the same class to have different instance variables, though doing so might not be a good idea! This creation of instance variables is typically done using methods, as will be demonstrated in an example soon.

Instance variables can also be deleted using the `del` statement!

In our `Date` example, we want the `Date` objects to have `day`, `month` and `year` as instance variables – we want each `Date` object to have it's own value for these. These instance variables can be created in the `setDate()` method as shown below:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             self.day,self.month,self.year = d,m,y
...
>>> d=Date()
>>> d.setDate(1,2,2000)
>>> d
<__main__.Date object at 0x7f592df6cb50>
>>> d.day
1
>>> d.month
2
>>> d.year
2000
```

Observation:

1. `setDate()` is a method – a function of the class `Date` that requires an invoking object to invoke it upon itself. In our example, the statement `d.setDate(1,2,2000)` shows how the `Date` object `d` invokes the function `setDate()` upon itself, passing `1`, `2` and `2000` as arguments to the method.
2. A reference to the invoking object is passed implicitly as the first argument to the method `setDate()`, and is received as the parameter `self`. While the parameter name should not matter strictly speaking, as a convention it is always named `self`. It would be wise to follow this convention.
3. Everything that belongs to the invoking object should be explicitly preceded by the reference `self`. Thus, to access the instance variable `day` of the invoking object, we would need to access `self.day`. The first time an assignment is made to an instance variable that does not exist, the instance variable is created.
4. The `setDate()` method copies the given parameters (`d`, `m` and `y`) to the corresponding instance variables (`self.day`, `self.month` and `self.year` respectively). We will improve this method later by adding validation support.
5. The above example shows that the `setDate()` works as expected.

12.4.2 Instance Methods

In the previous example, we have already seen the method `setDate()` in action. We have seen that methods are functions of the class that use invoking objects, and can access their corresponding invoking objects using the first parameter – typically called `self`.

Let us add more instance methods to our `Date` class:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             self.day,self.month,self.year = d,m,y
...     def getDay(self): return self.day
...     def getMonth(self): return self.month
...     def getYear(self): return self.year
...
>>> d = Date()
>>> d.setDate(1,2,2000)
>>> print("{}-{}-{}".format(d.getDay(),d.getMonth(),d.getYear()))
1-2-2000
```

Observation:

1. We had earlier defined the method `setDate()`. Such methods that are used to set the state of an object are called *setters* or *setter methods*.
2. We have now added the methods `getDay()`, `getMonth()` and `getYear()` to obtain or extract the state of an object. Such methods are called *getters* or *getter methods*. Since these methods have a single simple statement within their bodies, they have been defined in a single line for brevity.
3. Observe that all methods (inclusive of and not limited to getters and setters) use the first parameter (`self`) to reference the invoking object.
4. After the class definition is completed, we create an object `d`, set it's date using our setter `setDate()` and extract the values back using our getters (`getDay`, `getMonth` and `getYear`) and print them to verify the whole process.

It would be a good idea to delegate the printing of the `Date` object to the object itself via a method `print()`:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             self.day,self.month,self.year = d,m,y
...     def getDay(self): return self.day
...     def getMonth(self): return self.month
...     def getYear(self): return self.year
...     def print(self):
...             print("{}-{}-{}".format
(self.getDay(),self.getMonth(),self.getYear()))
...
>>> d = Date()
>>> d.setDate(1,2,2000)
>>> d.print()
1-2-2000
```

Observation:

1. We have added a method `print()` to the `Date` class to print out the date. This `print()` method has no connection to the global built-in `print()` function. We can of course think of a different name than `print()` if required.
2. By delegating the responsibility of printing to the object, the usage of the object becomes simpler!
3. We will see a better technique of achieving this objective of printing a string representation of an object in section 12.8.2.2.

The reason Data Hiding is considered important is that it eliminates accidental changes to data, thereby maintaining the integrity of the object. This is made possible by preventing direct access to data and providing accessible methods that permit assignment to non-accessible data after due validation. How data can be made inaccessible will be dealt with in section 12.4.6, but right now we will focus on providing a method that performs validation before assigning values to data. Here is the new code with the setter method `setDate()` modified to include validation:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             if self.isValid(d,m,y):
...                     self.day,self.month,self.year = d,m,y
...             else:
...                     print("Invalid date!")
...     def getDay(self): return self.day
...     def getMonth(self): return self.month
...     def getYear(self): return self.year
...     def print(self):
...             print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
...     def isValid(self,d,m,y):
...             if y<1 or y>9999: return False
...             if m<1 or m>12: return False
...             daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
...             if self.isLeap(y): daysInMonth[2]=29
...             if d<1 or d>daysInMonth[m]: return False
...             return True
...     def isLeap(self,y): return y%4==0 and (not y%100==0 or y
%400==0)
...
>>> d=Date()
>>> d.setDate(1,2,2000)
>>> d.setDate(29,2,2001)
Invalid date!
```

Observation:

1. The `setDate()` method performs validation of the given inputs using the `isValid()` method (defined later in the class). Only if the given values represents a valid date, it is stored in the instance variables; otherwise an error message is displayed. Instead of displaying an error message, a more professional approach would be to throw an exception. This better version is covered in section 13.
2. The `isValid()` method first verifies that the year is between 1 and 9999 and the month is between 1 and 12. It then builds the array `daysInMonth` to keep track of the maximum days in each month. We also take into consideration the fact that leap years have 29 days in February. The day is then validated. The method simply returns `True` if the given date is valid and `False` otherwise.
3. The `isLeap()` method checks if the given year is leap or not and returns

`True` if leap and `False` otherwise. The condition for a year to be leap is that it must be divisible by 4, and if divisible by 100 then must also be divisible by 400.

4. We will see a better way of implementing the `isValid()` and `isLeap()` methods in section 12.4.4.

Before proceeding to the next section, let's add some more functionality to our `Date` class and store it as a program. We will add a mechanism to add days to a `Date` object and obtain the date after the addition:

Date1.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       def setDate(self,d,m,y):
7.           if self.isValid(d,m,y):
8.               self.day,self.month,self.year = d,m,y
9.           else:
10.              print("Invalid date!")
11.
12.      def getDay(self): return self.day
13.      def getMonth(self): return self.month
14.      def getYear(self): return self.year
15.
16.      def print(self):
17.          print("{}-{}-{}".format
(self.getDay(),self.getMonth(),self.getYear()))
18.
19.      def isValid(self,d,m,y):
20.
daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
21.          if y<1 or y>9999: return False
22.          if self.isLeap(y): daysInMonth[2]=29
23.          if m<1 or m>12: return False
24.          if d<1 or d>daysInMonth[m]: return False
25.          return True
26.
27.      def isLeap(self,y): return y%4==0 and (not y%100==0 or
y%400==0)
28.
29.      def addDays(self,days):
30.          d,m,y = self.day,self.month,self.year
31.
daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
```

```
32.          if self.isLeap(y): daysInMonth[2]=29
33.
34.          for i in range(days):
35.              d=d+1
36.              if d>daysInMonth[m]:
37.                  d=1
38.                  m=m+1
39.                  if m>12:
40.                      m=1
41.                      y=y+1
42.                      if self.isLeap(y): daysInMonth[2]=29
43.                      else: daysInMonth[2]=28
44.          result = Date()
45.          result.setDate(d,m,y)
46.          return result
47.
48.  d1 = Date()
49.  d1.setDate(1,2,2000)
50.  d2 = d1.addDays(100)
51.  d2.print()
```

Output:

```
11-5-2000
```

Observation:

1. We have introduced an `addDays()` method in line 29.
2. The `addDays()` method uses a loop (line 34) to increment the day (`d`) by 1 (line 35) as many times as the number of days to be added.
3. Whenever the day (`d`) crosses the number of days in that month (tested in line 36), we reset the day to 1 and increment the month.
4. Whenever the month crosses 12 (tested in line 39), we reset the month to 1 and increment the year.
5. Whenever the year changes, we recompute the number of days in February depending on whether the year is leap or not (lines 42-43).
6. For simplicity, we have not checked if the year crosses 9999.
7. Finally, the method returns a `Date` object from the values in `d`, `m` and `y`.
8. From the output, we can confirm that the date 100 days after 1-2-2000 is indeed 11-5-2000.

12.4.3 Class Variables

As already introduced, class variables are variables that belong to the class and are shared across instances of that class. They are accessible using the class name (the class object) or any instance of that class. The following code snippets demonstrate these features:

```
>>> class A:
...     x=10
...
>>> A.x
10
```

We have created a class `A` with a class variable `x` initialized to `10`. We can access this using `A.x`, where `A` is the class object (class name for us) and `x` is the class variable. Let us create objects now:

```
(continuation)
>>> a=A()
>>> b=A()
>>> a.x
10
>>> b.x
10
```

We have created 2 objects `a` and `b`, and can see that they both have access to the class variable `x`, and provide us the same value `10`. Let us attempt to make an assignment to the class variable using an instance:

```
(continuation)
>>> a.x=20
>>> A.x
10
>>> a.x
20
>>> b.x
10
```

When an assignment is made to a variable using an instance, the variable is assumed to be an instance variable. Thus, the statement `a.x=20` ends up creating an instance variable `x` in the instance `a` and assigns the value `20` to it without disturbing the class variable `x` that is accessible via `A` as well as `b`. Any attempt to access the variable `x` using the instance a will give preference to the instance variable `x` in the instance `a`. Let us now make an assignment to the class variable using the class object:

```
(continuation)
>>> A.x=30
>>> A.x
30
>>> a.x
20
>>> b.x
30
```

We can see that when we change the class variable using the class object, the change is visible using the class object as well as all instances of that class, except those instances that also have an instance variable with the same name (in our case, the instance variable `x` in the instance `a`).

Let us apply this concept to our `Date` class in `Date1.py`. We see that the methods `isValid` and `addDays` require the same array `daysInMonth`. They are specified twice – in lines 20 and 31. Let us make this variable a class variable, accessible by any object – and therefore any method within the object. We could of course have made this an instance variable, but the fact that the values of this array does not change from instance to instance justifies making this a class variable instead.

Date2.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def setDate(self,d,m,y):
9.           if self.isValid(d,m,y):
10.              self.day,self.month,self.year = d,m,y
11.          else:
12.              print("Invalid date!")
13.
14.      def getDay(self): return self.day
15.      def getMonth(self): return self.month
16.      def getYear(self): return self.year
17.
18.      def print(self):
19.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
20.
21.      def isValid(self,d,m,y):
22.          Date.daysInMonth[2]=28
23.          if y<1 or y>9999: return False
```

```
24.          if self.isLeap(y): Date.daysInMonth[2]=29
25.          if m<1 or m>12: return False
26.          if d<1 or d>Date.daysInMonth[m]: return False
27.          return True
28.
29.      def isLeap(self,y): return y%4==0 and (not y%100==0 or
y%400==0)
30.
31.      def addDays(self,days):
32.          d,m,y = self.day,self.month,self.year
33.          Date.daysInMonth[2]=28
34.          if self.isLeap(y): Date.daysInMonth[2]=29
35.
36.          for i in range(days):
37.              d=d+1
38.              if d>Date.daysInMonth[m]:
39.                  d=1
40.                  m=m+1
41.                  if m>12:
42.                      m=1
43.                      y=y+1
44.                      if self.isLeap(y):
Date.daysInMonth[2]=29
45.                      else: Date.daysInMonth[2]=28
46.          result = Date()
47.          result.setDate(d,m,y)
48.          return result
49.
50.  d1 = Date()
51.  d1.setDate(1,2,2000)
52.  d2 = d1.addDays(100)
53.  d2.print()
```

Observation:

1. This code is basically taken from `Date.py`. The local variable `daysInMonth` has been removed from the methods `isValid` and `addDays` and instead made a class variable in line 6.
2. Each reference to `daysInMonth` elsewhere in the code has been replaced with `Date.daysInMonth` since it is a class variable now as opposed to a local variable earlier.
3. Since the class variable `daysInMonth` is shared by all instances and the methods `isValid` and `addDays`, any change made by one of them will be visible in all others. We therefore want to be careful about the number of days in February. The addition of lines 22 and 33 makes this code reliable and resilient to past changes to the number of days in February by any method.

12.4.4 Class Functions

Just as how variables defined in a class automatically become class variables, functions defined within a class also become class functions. They are methods only when an object is used to invoke the function, in which case a reference to the invoking object is passed automatically as the first argument and is typically received as the `self` parameter. Class functions can be invoked only through the class object and obviously do not receive any invoking object reference as `self`.

The following code snippets will help demonstrate these features:

```
>>> class A:
...     x=10
...     def increment():
...             A.x=A.x+1
...
>>> A.x
10
>>> A.increment()
>>> A.x
11
```

We define a class `A` with a class variable `x` that is initialized to `10`. We define a function `increment()` that increments the value of this class variable. The function is invoked using the class object (`A`) and not by using any object of the class `A`.

```
(continuation)
>>> a=A()
>>> b=A()
>>> a.x
11
>>> b.x
11
```

We then create 2 objects of the class A (`a` and `b`) and observe that they too give us the incremented value of the class variable `x`.

We cannot however invoke the class function `increment()` using any of the objects `a` and `b` as class functions are not methods:

```
(continuation)
>>> a.increment()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: increment() takes 0 positional arguments but 1 was given
```

Class functions are preferred over methods when it is obvious that the function does not depend on any particular instance of that class. In our `Date` example, we observe that the functions `isValid()` and `isLeap()` are in no way connected to any specific instance of the `Date` class, and are thus candidates for conversion into class functions.

For beginners, a simple technique to determine when to convert instance methods to class functions is when there is no usage of `self` within the function definition, except for accessing other methods which are also candidates for conversion to class functions. In our `Date` example, the method `isValid()` does use `self` to invoke the method `isLeap()`. The `isLeap()` method does not use `self` in it's definition and hence can be converted to a class function. After this change, the call to `isLeap()` within `isValid()` changes from `self.isLeap()` to `Date.isLeap()` and there is no usage of `self` within `isValid()`, making it possible to convert `isValid()` also from an instance method to a class function.

In case you are wondering why we should convert these instance methods into class functions when the code is working perfectly fine, it is because it is artificial and could also be misleading or meaningless. In the statement `self.isValid(d,m,y)`, `self` has absolutely no role to play! Similarly, in the statement `d1.isLeap(2000)`, `d1` has no role to play and could be misleading at worst and meaningless at best. It is far better to replace them with `Date.isValid(d,m,y)` and `Date.isLeap(2000)` respectively. Also note that class functions can be invoked even when the class has not been instantiated while instance methods cannot be invoked without an instance.

NOTE:

Python 2.2 and above supports the decorators `@classmethod` and `@staticmethod`, both of which are outside the purview of this book!

Here is our new `Date` class after making these changes:

Date3.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def setDate(self,d,m,y):
9.           if Date.isValid(d,m,y):
10.              self.day,self.month,self.year = d,m,y
11.          else:
12.              print("Invalid date!")
13.
14.      def getDay(self): return self.day
```

```
15.      def getMonth(self): return self.month
16.      def getYear(self): return self.year
17.
18.      def print(self):
19.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
20.
21.      def isValid(d,m,y):
22.          Date.daysInMonth[2]=28
23.          if y<1 or y>9999: return False
24.          if Date.isLeap(y): Date.daysInMonth[2]=29
25.          if m<1 or m>12: return False
26.          if d<1 or d>Date.daysInMonth[m]: return False
27.          return True
28.
29.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
30.
31.      def addDays(self,days):
32.          d,m,y = self.day,self.month,self.year
33.          Date.daysInMonth[2]=28
34.          if Date.isLeap(y): Date.daysInMonth[2]=29
35.
36.          for i in range(days):
37.              d=d+1
38.              if d>Date.daysInMonth[m]:
39.                  d=1
40.                  m=m+1
41.                  if m>12:
42.                      m=1
43.                      y=y+1
44.                      if Date.isLeap(y):
Date.daysInMonth[2]=29
45.                      else: Date.daysInMonth[2]=28
46.          result = Date()
47.          result.setDate(d,m,y)
48.          return result
49.
50.  d1 = Date()
51.  d1.setDate(1,2,2000)
52.  d2 = d1.addDays(100)
53.  d2.print()
```

Output:

```
11-5-2000
```

Observation:

1. This program is based on `Date2.py`. We have changed the instance methods `isValid()` and `isLeap()` to class functions by eliminating `self`.
2. A result of this change is that all calls to `isValid()` and `isLeap()` will now require `Date` instead of a `Date` instance.

12.4.5 Instance Methods as special Class Functions

Having understood class functions, we can view methods from a new perspective now: methods are class functions that accept a reference to the invoking object as the first argument! The following code snippet will illustrate this:

```
>>> class A:
...     def f(self):
...         print("Hello")
...
>>> a=A()
>>> a.f()
Hello
>>> A.f(a)
Hello
```

Observation:

1. We have defined a class `A` with a method `f` that prints "`Hello`" when invoked. We have an instance of this class whose reference is stored in `a`.
2. `a.f()` is a call to the method `f` of class `A` using `a` as the invoking object.
3. `A.f(a)` is a call to the class function `f` of the class `A`, passing `a` as an argument that is received as `self` in `f`.

Thus, methods are special class functions!

12.4.6 Public, Private and Protected Members

OOP generally talks about private, public and protected members of a class with this interpretation:

1. **Public** members are accessible everywhere where the class/object is accessible.
2. **Protected** members are accessible within the class that defines it as well as in all subclasses (classes that inherit it).
3. **Private** members are accessible only within the class and nowhere else.

Python does not directly support these concepts and all members of a class are inherently public. Python does support a convention however, that might help support this but does not enforce it. Since all class members are anyway public, we will only concentrate on protected and private members in this section.

12.4.6.1 Protected Members

As mentioned earlier, protected members are accessible within the class in which they are defined as well as in all it's subclasses.

Any member that starts with a single underscore indicates that it should be treated as a protected member. As already emphasized, this is only a *convention* and is not enforced by Python, but professionals treat this convention as a rule.

Syntax:

```
_member
```

Since we have not dealt with inheritance yet, protected members will be demonstrated in section 12.6.2.

12.4.6.2 Private Members

As mentioned earlier, private members are accessible only within the class in which they are defined and are not accessible anywhere else.

Any member that starts with minimum 2 underscores and ends with maximum 1 underscore indicates that it should be treated as a private member. This is achieved in Python typically by *name mangling*:

Private members of the form `__member_` get replaced by `_classname__member`. In other words, a member that starts with at least 2 underscores and ends with at most 1 underscore gets replaced to an underscore, followed by the name of the class to which the member belongs, followed by 2 underscores, followed by the member name. This prevents direct access to the member by it's name, but still does not prevent access via it's mangled name! Furthermore, this name mangling procedure may change in the future and hence programmers are advised not to use the mangled

name at any time to access a private member *illegally*!

Syntax:

```
__member
```

Here is a code snippet to demonstrate private members:

```
>>> class A:
...     def set(self,x,y):
...             self.x = x
...             self.__y = y
...     def print(self):
...             print("{},{}".format(self.x,self.__y))
...
>>> a=A()
>>> a.set(2,3)
>>> a.print()
2,3
>>> a.x
2
>>> a.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'y'
>>> a.__y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__y'
>>> a._A__y
3
```

Observation:

1. We have defined a class `A` with a setter called `set` that assigns the parameters `x` and `y` to the members `x` and `y` (called `__y`) respectively. The member `x` is considered to be public whereas the member `__y` is considered to be private.
2. The `print()` method displays the values of members `x` and `__y`, both of which are accessible because `print()` is a method of the same class.
3. From outside the class, only `x` is accessible and `__y` is not.
4. The private member `__y` is accessible outside the class with the name `_A__y`.

In our `Date` example, we can convert all class variables and instance variables to private! In general, it is always better to convert all class variables and instance variables to private to prevent accidental changes from outsiders and thus implement data hiding. Therefore, the instance variables `day`, `month` and `year` need to be renamed as `__day`, `__month` and `__year` respectively. Also the class variable `daysInMonth` will be replaced by `__daysInMonth`.

Date4.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       __daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def setDate(self,d,m,y):
9.           if Date.isValid(d,m,y):
10.              self.__day,self.__month,self.__year = d,m,y
11.          else:
12.              print("Invalid date!")
13.
14.      def getDay(self): return self.__day
15.      def getMonth(self): return self.__month
16.      def getYear(self): return self.__year
17.
18.      def print(self):
19.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
20.
21.      def isValid(d,m,y):
22.          Date.__daysInMonth[2]=28
23.          if y<1 or y>9999: return False
24.          if Date.isLeap(y): Date.__daysInMonth[2]=29
25.          if m<1 or m>12: return False
26.          if d<1 or d>Date.__daysInMonth[m]: return False
27.          return True
28.
29.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
30.
31.      def addDays(self,days):
32.          d,m,y = self.__day,self.__month,self.__year
33.          Date.__daysInMonth[2]=28
34.          if Date.isLeap(y): Date.__daysInMonth[2]=29
35.
36.          for i in range(days):
```

```
37.              d=d+1
38.              if d>Date.__daysInMonth[m]:
39.                  d=1
40.                  m=m+1
41.                  if m>12:
42.                      m=1
43.                      y=y+1
44.                      if Date.isLeap(y):
Date.__daysInMonth[2]=29
45.                      else: Date.__daysInMonth[2]=28
46.          result = Date()
47.          result.setDate(d,m,y)
48.          return result
49.
50.  d1 = Date()
51.  d1.setDate(1,2,2000)
52.  d2 = d1.addDays(100)
53.  d2.print()
```

Output:

```
11-5-2000
```

Observation:

1. This program is based on `Date3.py`. The instance variables (`day`, `month` and `year`) and class variables (`daysInMonth`) have been made private by prefixing them with __.
2. The instance and class variables can no longer be directly accessed outside the class.
3. Even class functions and instance methods can be made private by prefixing them with __. Such functions/methods can then be used internally by the class and is not directly callable from outside the class. In our `Date` example, we can consider making the class functions `isValid` and `isLeap` private if we feel that the outside world will have no interest in these.
4. Making class variables and instance variables private helps implement *data hiding*. Making class functions and instance methods private helps implement *data abstraction*.

12.5 Constructors and Destructors

In OOP, a **constructor** is a member function of a class that is automatically invoked when an instance of that class is created in order to initialize the object to a valid state. A **destructor** is a member function of a class that is automatically invoked when an instance is destroyed in order to perform any clean-up required.

Python does not have exact implementations for these, but does provide something very similar. This section focuses on Python's version of constructors and destructors.

12.5.1 Constructors

A constructor in Python is an instance method that is automatically invoked when an instance is created and permits the programmer to perform any initialization required to ensure that the instance is in a valid state.

A constructor is identified in Python by it's special name: `__init__`. Note that the leading underscores do not make this instance method private as the method name does not end with at most 1 underscore – it in fact ends with 2 underscores!

The following code snippet demonstrates how constructors can be designed and how they are automatically invoked when objects are instantiated:

```
>>> class A:
...     def __init__(self):
...             print("Constructor called!")
...
>>> a=A()
Constructor called!
>>> b=A()
Constructor called!
```

Observation:

1. The constructor is always called `__init__()` and being an instance method it will always receive `self` as the first parameter.
2. The constructor can be designed to receive any number of parameters in addition to `self`, but since Python does not support function overloading, only 1 constructor can exist in a class.
3. It is a good programming practice to use the constructor to create all instance variables that an object requires – all initialized to meaningful values.
4. In classes without explicitly defined constructors, one can imagine Python adding a dummy constructor that does nothing, i.e. an empty constructor.

In our `Date` example, we can add a constructor that permits the construction of a `Date` object by specifying the `day`, `month` and `year`. We need not worry about the implementation as the `setDate()` method can be used for this purpose. Since the objective of the constructor is to ensure that the object is in a valid state, we will ensure this even if the given values of `day`, `month` and `year` is not valid.

Date5.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       __daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def __init__(self,d,m,y):
9.           self.setDate(1,1,1970)
10.          self.setDate(d,m,y)
11.
12.      def setDate(self,d,m,y):
13.          if Date.isValid(d,m,y):
14.              self.__day,self.__month,self.__year = d,m,y
15.          else:
16.              print("Invalid date!")
17.
18.      def getDay(self): return self.__day
19.      def getMonth(self): return self.__month
20.      def getYear(self): return self.__year
21.
22.      def print(self):
23.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
24.
25.      def isValid(d,m,y):
26.          Date.__daysInMonth[2]=28
27.          if y<1 or y>9999: return False
28.          if Date.isLeap(y): Date.__daysInMonth[2]=29
29.          if m<1 or m>12: return False
30.          if d<1 or d>Date.__daysInMonth[m]: return False
31.          return True
32.
33.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
34.
35.      def addDays(self,days):
36.          d,m,y = self.__day,self.__month,self.__year
37.          Date.__daysInMonth[2]=28
```

```
38.          if Date.isLeap(y): Date.__daysInMonth[2]=29
39.
40.          for i in range(days):
41.              d=d+1
42.              if d>Date.__daysInMonth[m]:
43.                  d=1
44.                  m=m+1
45.                  if m>12:
46.                      m=1
47.                      y=y+1
48.                      if Date.isLeap(y):
Date.__daysInMonth[2]=29
49.                      else: Date.__daysInMonth[2]=28
50.          result = Date()
51.          result.setDate(d,m,y)
52.          return result
53.
54.  d1 = Date(1,2,2000)
55.  d1.print()
56.  d2 = Date(90,90,90)
57.  d2.print()
```

Output:

```
1-2-2000
Invalid date!
1-1-1970
```

Observation:

1. This program is based on `Date4.py`. We have introduced a constructor in lines 8-10.
2. The constructor receives the `day`, `month` and `year` and uses the `setDate()` method to perform the necessary validation and assignment. It is a good practice to reuse functions as it reduces our work, makes the code more robust and also makes it easy to change the program later on if required.
3. Since our current implementation of `setDate()` merely prints an error message and continues if the given values of `day`, `month` and `year` do not represent a valid date, our constructor first sets a valid date (1-1-1970 is chosen primarily because it is a valid date, and more so because it is an important reference date in computers) and then only proceeds to call `setDate()` to change the date if the given date is valid. A better implementation will use exception handling, covered in section 13.

12.5.2 Destructors

A destructor in Python is an instance method that is automatically invoked when an object is going to be destroyed and eliminated and permits the programmer to perform any desired clean-up.

A destructor is identified in Python by it's special name: `__del__`.

The following code snippet demonstrates the working of destructors in Python:

```
>>> class A:
...     def __del__(self):
...             print("Destructor called!")
...
>>> a=A()
>>> del a
Destructor called!
>>> b=A()
>>> del b
Destructor called!
```

Observation:

1. The destructor is an instance method does not receive any parameters apart from `self`.
2. Like constructors, one class can have only one destructor, and the absence of a user-defined destructor results in a dummy destructor supplied by Python that is empty.
3. While in this example the destructor merely prints a message, destructors can do many meaningful operations when an object is going to be eliminated. If no such requirement exists, then a destructor need not be defined in that class.
4. The `del` built-in function is used to delete a variable, and can also end up deleting objects in memory.

Do recollect from section 2.6 however that there is a difference between objects/instances and references and that variables in Python only hold references! Deleting a variable using `del` not only deletes that variable, but also decrements the reference count of the referenced object by 1, and only when this reference count reaches 0 (implying that the object has no active references in the program), the object is deleted. This is demonstrated below:

```
>>> class A:
...     def __del__(self):
...             print("Destructor called!")
...
>>> a=A()
>>> b=a
>>> del a
>>> del b
Destructor called!
```

Observation:

1. We create an object of class `A` and store it's reference in the variable `a`. The reference count of this object is 1 (only 1 variable – `a` – is referring to the object)
2. This reference is copied from `a` to `b`, ending up with 2 references to the object. Therefore, the reference count of the object becomes 2.
3. When we delete the variable `a`, we also decrement the reference count of the object referred to by `a` by 1. Therefore, the reference count of the object decrements to 1. Since it has not reached 0, the object continues to exist unaffected by the deletion of the variable `a`.
4. When we delete the variable `b`, we also decrement the reference count of the object referred to by `b` by 1. The reference count now reaches 0. This is when the object will be destroyed, and just prior to that the destructor gets automatically called.

Since our `Date` class does not really require a destructor, we will not attempt to demonstrate the addition of the same.

12.6 Inheritance

Inheritance is one of the most important concepts in OOP and has these advantages:

1. It helps **compartmentalise** the code, thereby helping the programmer organise the code better.
2. It allows **reusability** of code by allowing a new class to completely obtain the functionality of other existing classes and add more functionality of it's own.
3. It permits **extensibility** of code, wherein new code is added without having to modify existing code and classes.

While the different structural forms of inheritance have been covered in section 12.1.7, we will basically examine 2 forms that make all other forms possible:

1. **Simple inheritance**, wherein one class inherits from another
2. **Multiple inheritance**, wherein one class inherits from multiple classes

One point to be kept in mind as far as inheritance is concerned is that the derived class contains *all* features of the base class. We can imagine that each instance of the derived class also contains an instance of the base class. The *private* features of the base class are also present in the derived class, but are not directly accessible.

12.6.1 Simple Inheritance

In simple inheritance, one class inherits from another existing class. The class that inherits is called the *derived* class or *subclass* while the class that is inherited from is called the *base* class or *super* class. Inheritance is therefore also termed *subclassing*.

The syntax of subclassing is shown below:

```
class derived_class(base_class):
    class_definition
```

Here is a small code snippet to demonstrate simple inheritance with class `B` deriving from class `A`:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> a=A()
>>> b=B()
>>> type(a)
<class '__main__.A'>
>>> type(b)
<class '__main__.B'>
```

Observation:

1. The class definitions are empty, but as Python requires at least 1 line in the definition, we have used the `pass` statement.
2. The class `B` derives from class `A`: `class B(A)`

12.6.2 Private, Public and Protected Revisited

Section 12.4.6 had introduced the convention for *public*, *private* and *protected* access in a class. We had not seen protected access in action as inheritance was not covered then. Now that we have an idea of inheritance, let us revisit these and see them in action in an example:

inheritance1.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Public, Private and Protected Method access
5.
6.   class A:
7.    def __f1(self): print("A.f1")
8.    def f2(self): print("A.f2")
9.    def _f3(self): print("A.f3")
10.
11.  class B(A):
12.   def __g1(self): print("B.g1")
13.   def g2(self): print("B.g2")
14.   def _g3(self): print("B.g3")
15.
16.  b = B()
```

Observation:

1. The above program does not produce any output. It has been given only for us to analyse the program and draw conclusions. Therefore, there is no point in running this program.
2. Class `A` is the *base* class and class `B` is the *derived* class. Class `A` contains 3 methods - `__f1`, `f2` and `_f3` – which are *private*, *public* and *protected* respectively. Class `B` also contains 3 methods - `__g1`, `g2` and `_g3` – which are *private*, *public* and *protected* respectively.
3. Recollect that *public* members are accessible both inside as well as outside the class, *private* members are accessible only inside the class and *protected* members are accessible only inside the class and inside it's subclasses. Also recollect that the interpreter may not prohibit direct access of protected members from outside the class.
4. Line 16 creates an instance of the derived class `B` called `b`. We will now analyse which methods can be invoked using this instance `b`.
5. `b.g2()` is the most obvious valid candidate – any public method of a class is accessible from even outside the class.

6. `b.f2()` is the next obvious valid candidate – public methods of the base class and inherited as public methods of the derived class and public methods of the derived class are accessible even outside the class.
7. `b.__g1()` is an invalid candidate – private methods of a class are not accessible outside the class. We of course can invoke `b.f2()` which in turn can call `self.__f1()`.
8. `b._g3()` is supposed to be an invalid candidate – protected members of a class are not accessible outside the class, but the interpreter may not specifically prohibit it's access. We will only attempt to call `_g3()` through some other route (like through `g2()` for instance).
9. `b.__f1()` is an invalid candidate – private members of a class are accessible only within that class and are inaccessible even in it's subclasses. We can follow other routes though, like calling `__f1()` from `f2()` or from `g2()`.
10. `b._f3()` is supposed to be an invalid candidate – protected members of a class are accessible within subclasses, but not outside of these subclasses. `B` being a subclass of `A` does have access to `_f3()`, but we are not supposed to access them from outside `B`.

12.6.3 Function Overriding

The previous section illustrated which functions are accessible within derived classes and which are accessible even outside derived classes in inheritance. A special case arises however when we have functions with the same name in both the base as well as derived classes – this is called **function overriding**, wherein a derived class function is offered as a replacement for a base class function. This section concentrates on this concept.

When there are methods with the same name in a base class as well as it's derived class and we have a reference to an instance of the derived class, using which we invoke the function, the derived class function gets called. This is illustrated below:

Inheritance2.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A.f")
9.
10.  class B(A):
11.      def f(self):
12.          print("B.f")
13.
14.  b = B()
15.  b.f()
```

Output:

```
B.f
```

What if we want `B`'s `f` to also invoke `A`'s `f` as part of it's functionality? We can use the class function syntax to invoke the method (as illustrated in section 12.4.5), passing `self` as the reference to the instance. Alternatively, we can use the `super()` built-in to access the super class object of the current object (identified by `self`):

Inheritance3.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A.f")
9.
10.  class B(A):
11.      def f(self):
12.          print("B.f")
13.          super().f()
14.
15.  b=B()
16.  b.f()
```

Output:

```
B.f
A.f
```

Observation:

1. The statement `A.f(self)` is very similar to the statement `self.f()`, with `self` referring to an instance of `A`. But in our case, `self` is referring to an instance of `B` and `self.f()` will result in a call to `B.f(self)` instead!
2. The statement `super().f()` is equivalent to the statement `self.f()` with `self` referring to an instance of `A` that is housed within an instance of `B` that is currently referred to by `self`!

What if we want to directly invoke `A`'s `f` without involving `B`'s `f` using an instance of `B`? Again, we can use the class function syntax as shown below:

Inheritance4.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A.f")
9.
10.  class B(A):
11.      def f(self):
12.          print("B.f")
13.          super.f()
14.
15.  b=B()
16.  A.f(b)
```

Output:

```
A.f
```

Observation:

1. This time, we do not use `b.f()`, which will always give preference to `B`'s `f`. We directly invoke `A.f` passing `b` as the argument.

12.6.4 Constructors and Destructors in Simple Inheritance

One of the important results of inheritance is that the derived class is dependent on the base class. While the base class can exist on it's own, the derived class is dependent on the base class for it's existence. Derived class instances are dependent on corresponding base class instances. Indeed we can say that every derived class instance contains a base instance within itself, without which it cannot exist.

Building on this, we can also say then that when a derived class is instantiated, the base class should get instantiated first and when a derived class instance is destroyed, it should be followed by the destruction of the base class instance too. In other words, the order of construction/creation should be from base class to derived class whereas the order of destruction should be from derived class to base class. This order honours our agreement that the base class/instance can survive without the derived class/instance but not vice-versa.

Unlike other programming languages like C++ where this is automatically enforced by the compiler, Python does not enforce it! It therefore becomes the responsibility of the programmer to ensure that this indeed takes place by explicitly calling suitable base class functions.

Let's begin with a simple code snippet to observe Python's default behaviour when it comes to constructors and destructors under inheritance:

Inheritance5.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Constructors and Destructors
5.
6.   class A:
7.       def __init__(self):
8.           print("A constructed")
9.       def __del__(self):
10.          print("A destroyed")
11.
12.  class B(A):
13.      def __init__(self):
14.          print("B constructed")
15.      def __del__(self):
16.          print("B destroyed")
```

```
17.
18.  b=B()
19.  del(b)
```

Output:

```
B constructed
B destroyed
```

Observation:

1. We observe that when we instantiate class `B`, class `B`'s constructor is invoked, but class `A`'s constructor is not automatically invoked.
2. Similarly, when we destroy the instance of class `B`, the destructor of class `B` is invoked, but the destructor of class `A` is not automatically invoked.

Ideally, we would want the constructors of both classes to be invoked during construction and the destructors to be similarly automatically invoked upon destruction. Since this is not automatically performed by Python, we need to change the snippet as follows to obtain the desired result:

`Inheritance6.py`

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Constructors and Destructors
5.
6.   class A:
7.       def __init__(self):
8.           print("A constructed")
9.       def __del__(self):
10.          print("A destroyed")
11.
12.  class B(A):
13.      def __init__(self):
14.          super().__init__()
15.          print("B constructed")
16.      def __del__(self):
17.          print("B destroyed")
18.          super().__del__()
19.
20.  b=B()
21.  del(b)
```

Output:

```
A constructed
B constructed
B destroyed
A destroyed
```

Observation:

1. The derived class constructors calls the base class constructor *before* executing any code within it. This is ideally how derived class constructors should be!
2. The derived class destructor calls the base class destructor *after* executing any code within it. This is ideally how derived class destructors should be!
3. While it may not be considered wrong in Python if such calls are not made, it is generally required. If you are unsure of whether or not these calls are required in a particular situation, you might want to add the calls anyway. In fact, you could make it a habit of adding these calls whenever you write classes involving inheritance and remove it if and only when you have a strong reason to do so.
4. When the base class constructor requires arguments, the arguments could be received by the derived class constructor and passed on to the base class constructor in the call. In such cases, the derived class constructor is free to accept more arguments than are required by the base class constructor, but the additional arguments would be meant to be used by the derived class constructor and should not be passed to the base class constructor.

Let us reimplement the `Date` example of section 12.5.1 using inheritance. Recollect that we were printing the `Date` using the `print()` method in UK format. What if we wanted US format? Would it be a good idea to modify the code to print in US format? No! That will cease the current functionality which might be required in other places and forces us to retest the entire code again with the changes in place. Would it be a good idea to write another `Date` class with a different name like `USDate`, copying the code from the `Date` class and then making the changes? No! Tomorrow if we end up changing the code of the `Date` class, we might have to make the same changes in the `USDate` class, making maintenance difficult, changes complicated and introducing a possibility of adding bugs. The best technique would be to derive `USDate` from the `Date` class, thereby automatically having all functionality of the `Date` class, and overriding the `print()` method providing a different implementation of the same. This is illustrated in the program below.

Date6.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class and USDate class using
inheritance
4.
5.   class Date:
6.       __daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def __init__(self,d,m,y):
9.           self.setDate(1,1,1970)
10.          self.setDate(d,m,y)
11.
12.      def setDate(self,d,m,y):
13.          if Date.isValid(d,m,y):
14.              self.__day,self.__month,self.__year = d,m,y
15.          else:
16.              print("Invalid date!")
17.
18.      def getDay(self): return self.__day
19.      def getMonth(self): return self.__month
20.      def getYear(self): return self.__year
21.
22.      def print(self):
23.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
24.
25.      def isValid(d,m,y):
26.          Date.__daysInMonth[2]=28
27.          if y<1 or y>9999: return False
28.          if Date.isLeap(y): Date.__daysInMonth[2]=29
29.          if m<1 or m>12: return False
30.          if d<1 or d>Date.__daysInMonth[m]: return False
31.          return True
32.
33.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
34.
35.  class USDate(Date):
36.      def __init__(self,d,m,y):
37.          super().__init__(d,m,y);
38.
39.      def print(self):
40.          print("{}/{}/
{}".format(self.getMonth(),self.getDay(),self.getYear()))
41.
```

```
42.  d1 = Date(1,2,2000)
43.  d1.print()
44.  d2 = USDate(1,2,2000)
45.  d2.print()
```

Output:

```
1-2-2000
2/1/2000
```

Observation:

1. This program is based on `Date5.py` covered in section 12.5.1, but we have removed the `addDays()` method for simplicity. Lines 1-34 are from that program.
2. We have added a class called `USDate` in line 35, which derives from the `Date` class.
3. Line 36 provides a constructor for the derived class, which simply passes on the parameters to the constructor of the base class. Our derived class constructor has no other job to do. No destructor was present earlier and is still not required here.
4. Line 39 provides a new definition for the `print()` method, which displays the date in US format.

12.6.5 Multiple Inheritance

A class can derive from 2 or more base classes, resulting in multiple inheritance. The syntax for multiple inheritance is given below:

```
class className(baseClass1,baseClass2[,baseClass3...])
```

The above syntax shows at least 2 compulsory base classes as without that this would be called multiple inheritance in the first place! There is no limit to how many classes a class can extend from. The order of inheritance is an important property here and goes from left to right in the base class list. Thus, the first base class is `baseClass1`, the second base class is `baseClass2` and so on. The order of inheritance matters when we use built-ins like `super()`, and professionals always aim to ensure that constructor calls are always in the order of inheritance whereas destructor calls are always in the strict reverse order. These are demonstrated in the following sections.

The program given below demonstrates multiple inheritance:

Inheritance7.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance
5.
6.   class A:
7.       def fa(self):
8.           print("A called")
9.
10.  class B:
11.      def fb(self):
12.          print("B called")
13.
14.  class C(A,B):
15.      def fc(self):
16.          self.fa()
17.          self.fb()
18.          print("C called")
19.
20.  c=C()
21.  c.fc()
```

Output:

```
A called
B called
C called
```

Observation:

1. Class `A` is defined in line 6 and contains a method `fa` defined in line 7.
2. Class `B` is defined in line 10 and contains a method `fb` defined in line 11.
3. Class `C` is defined in line 14 and derives from class `A` and class `B`. The order of inheritance is `A` followed by `B`, though in this example we do not observe any result because of this order.
4. A method `fc` is defined in line 14 inside class C, which invokes methods `fa` and `fb` upon itself, available to it because of inheritance.

12.6.6 Function Overriding in Multiple Inheritance

Function Overriding was introduced in section 12.6.3 and this section will focus on function overriding in multiple inheritance. Let us start with our first program:

Inheritance8.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance - Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A")
9.
10.  class B:
11.      def f(self):
12.          print("B")
13.
14.  class C(A,B):
15.      def f(self):
16.          print("C")
17.
18.  c=C()
19.  c.f()
```

Output:

```
C
```

Observation:

1. Class `C`, defined in line 14, derives from both class `A` and class `B`.
2. All the classes (`A`, `B` and `C`) define their own respective copies of the method `f` (lines 7, 11 and 15).
3. When a call is made in line 19 using the statement `c.f()`, preference is given to the method `f` in class `C`.

The reason why we are revisiting function overriding with respect to multiple inheritance is that it gets interesting now when we use the `super()` function because the question arises: when there are 2 (or more) base classes, which base class is considered to be the super class? To answer this question is obtained by the order of inheritance, which was covered in section 12.6.5. To recall, the order of inheritance is from left to right, and in this example, the first base class of `C` is therefore `A`. The first preference is given to the method `f` in class `A`. Only if `A` (and it's super classes, in the order of inheritance) does not define method `f` will preference be given to class `B` (and it's super classes in the order of inheritance). This is demonstrated by the following program:

Inheritance9.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance - Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A")
9.
10.  class B:
11.      def f(self):
12.          print("B")
13.
14.  class C(A,B):
15.      def f(self):
16.          super().f()
17.          print("C")
18.
19.  c=C()
20.  c.f()
```

Output:

```
A
C
```

What if we want to call the method `f` of class `B` from within class `C`? We use the class function syntax: `B.f(self)`.

12.6.7 Constructors and Destructors in Multiple Inheritance

Section 12.6.4 emphasized on the order of construction and destruction and explained how this can be done by the programmer. The following program puts it all together to illustrate the best way of writing programs in Python that employ multiple inheritance:

inheritance10.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance - Constructors and Destructors
5.
6.   class A:
7.       def __init__(self):
8.           print("A constructed")
9.       def __del__(self):
10.          print("A destroyed")
11.
12.  class B:
13.      def __init__(self):
14.          print("B constructed")
15.      def __del__(self):
16.          print("B destroyed")
17.
18.  class C(A,B):
19.      def __init__(self):
20.          A.__init__(self)
21.          B.__init__(self)
22.          print("C constructed")
23.      def __del__(self):
24.          print("C destroyed")
25.          B.__del__(self)
26.          A.__del__(self)
27.
28.  c=C()
29.  del(c)
```

Output:

```
A constructed
B constructed
C constructed
C destroyed
B destroyed
A destroyed
```

Observation:

1. The constructor of class `C` first passes control to the constructor of class `A`, then the constructor of class `B` and then continues it's execution.
2. The destructor of class `C` first executes itself and finally calls the destructor of class `B` followed by the destructor of class `A`.
3. As pointed out in section 12.6.4, this is done to honour the rule that base classes can exist without derived classes but not the other way around.
4. The order of inheritance is respected: class `A` is the first base class of class `C` and hence is created first but destroyed last.

12.7 Dynamic Polymorphism

A very important feature of OOP is *dynamic polymorphism* – the mechanism of deciding which function to invoke at runtime depending on the type of the invoking object.

Before actually understanding dynamic polymorphism, it is important to realize that *subclasses are perfectly substitutable for base classes*, meaning that in any situation where we require a reference to a base class instance, a reference to a derived class instance will do equally well. Let us illustrate that through a sample program:

Inheritance11.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Dynamic Polymorphism
5.
6.   class Animal:
7.       def __init__(self,name):
8.           self.name = name
9.
10.      def speak(self):
11.          pass
12.
13.  class Dog(Animal):
14.      def __init__(self):
15.          super().__init__("Dog")
16.
17.      def speak(self):
18.          print("Bow wow!")
19.
```

```
20.  class Cat(Animal):
21.      def __init__(self):
22.          super().__init__("Cat")
23.
24.      def speak(self):
25.          print("Meow!")
26.
27.
28.  def introduce(animal):
29.      print("Hi! This animal is called",animal.name)
30.      print("This animal says: ",end='')
31.      animal.speak()
32.
33.  animal = Dog()
34.  introduce(animal)
35.  animal = Cat()
36.  introduce(animal)
```

Output:

```
Hi! This animal is called Dog
This animal says: Bow wow!
Hi! This animal is called Cat
This animal says: Meow!
```

Observation:

1. Class `Animal` is defined in line 6. It contains a constructor and a method called `speak`.
2. The constructor (defined in line 7) receives the name of the animal and stores it in the attribute `name`.
3. The `speak` method (defined in line 10) does nothing and will be overridden by the derived classes suitably.
4. The `Dog` class (defined in line 13) derives from the `Animal` class. It's constructor receives nothing, but invokes the constructor of `Animal` passing "`Dog`" as the name of the animal. It's `speak` method ends up printing "`Bow wow!`"
5. The `Cat` class (defined in line 20) derives from the `Animal` class. It's constructor receives nothing, but invokes the constructor of `Animal` passing "`Cat`" as the name of the animal. It's `speak` method ends up printing "`Meow!`"

6. The `introduce` function (defined in line 28) accepts an animal, prints it's name and invokes it's `speak` method to make the animal “speak”. While the function is designed to receive an instance of the type `Animal`, it can also receive an instance of any derived class of `Animal` because of the law of substitutability introduced at the beginning of this section. If a `Dog` instance is passed, preference is given to the `speak` method of `Dog` and if a `Cat` instance is passed, preference is given to the `speak` method of `Cat`. If `Dog` or `Cat` class does not override the `speak` method of `Animal`, then the speak method of `Animal` gets invoked which does nothing.

12.7.1 Abstract Methods and Classes

The previous program shows 2 disadvantages of the way the way the `speak` method has been defined in `Animal`:

1. We are forced to provide an empty definition of the method, which can seem a little odd at best.
2. If any of the derived classes do not override the `speak` method, there is no output on invoking the `speak` method as the `speak` method of `Animal` gets invoked and it is empty. There was no way to force the derived class to override the `speak` method.

These 2 disadvantages can be eliminated. The class can be made an abstract class using the `abc` package and the method can be made an abstract method using `@abstractmethod`, both of which are outside the scope of this book.

12.8 Attribute Handling, Magic Functions and Operator Overloading

A very interesting feature in Python is that certain operations when performed on objects will result in certain standard functions being called. What is interesting in this is that it does not appear that such a call is taking place!

This same principle is also used to implement operator overloading in Python – the mechanism by which an operator plays the same logical role but the implementation depends on the operand types on which the operator operates.

While this section focuses on operator overloading, we start with certain standard functions that are called when some basic operations are performed on objects. After an understanding of this, we will proceed to add support for operators to our classes.

12.8.1 Attribute Handling

Section 12.4.1 introduced the fact that instances can have attributes of their own. We use the syntax `object.attribute` to access such attributes. There are functions available in Python to deal with attributes of instances.

12.8.1.1 The hasattr() Function

The `hasattr()` function tells whether a particular instance has a particular attribute or not.

Syntax:

```
hasattr(object,attribute)
```

This is a Boolean function that returns true only if the given instance contains the given attribute, as shown in the code snippet below:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> hasattr(a,'x')
True
>>> hasattr(a,'y')
False
```

Observation:

1. We define a class `A` that contains a constructor which initializes an attribute `x` to `0` (thereby creating it). Since the constructor is invoked each time an instance is created, it is guaranteed that all instances of `A` will have the attribute `x` in them.
2. `hasattr(a,'x')` therefore returns `True` whereas `hasattr(a,'y')` returns `False` since no attribute `y` was created in the instance `a`.

12.8.1.2 The getattr() Function

The `getattr()` function returns the value of an attribute within an instance if it exists, returning a default value (if provided) if the attribute does not exist.

Syntax:

```
getattr(object,attribute[,default])
```

Note:

1. If the attribute `attribute` exists in the instance `object`, it's value is returned.
2. If the attribute `attribute` does not exist in the instance `object`, `default` is returned.
3. If the attribute `attribute` does not exist in the instance `object` and no `default` is provided, an `AttributeError` occurs.

Example:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> getattr(a,'x',2)
0
>>> getattr(a,'y',2)
2
>>> getattr(a,'y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'y'
```

Observation:

1. `getattr(a,'x',2)` returns the value of the attribute `x` in the instance `a`, which is `0`.
2. `getattr(a,'y',2)` returns the value `2` (the `default`) as there is no attribute `y` in the instance `a`.
3. `getattr(a,'y')` generates an `AttributeError` as there is no attribute `y` in the instance `a` and no `default` was provided either.

12.8.1.3 The setattr() Function

The `setattr()` function sets the value of an attribute in an instance. If the attribute already existed in the instance, it's value is overwritten and if it did not exist, it is created.

Syntax:

```
setattr(object,attribute,value)
```

Example:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> setattr(a,'x',2)
>>> setattr(a,'y',3)
>>> a.x
2
>>> a.y
3
```

Observation:

1. An instance `a` is created with an attribute `x` having the value `0`.
2. `setattr(a,'x',2)` overwrites the value of the attribute `x` in the instance `a` to `2`.
3. `setattr(a,'y',3)` creates an attribute `y` in the instance `a` and assigns a value `3` to it.

12.8.1.4 The delattr() Function

Finally, to remove an existing attribute from an instance, the `delattr`() function can be used.

Syntax:

```
delattr(object,attribute)
```

Note:

1. If the attribute `attribute` exists in the instance `object`, it is removed from the instance.
2. If the attribute `attribute` does not exist in the instance `object`, an `AttributeError` is generated.

Example:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> a.x
0
>>> delattr(a,'x')
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'x'
>>> delattr(a,'y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: y
```

12.8.1.5 Standard Attributes

While the programmer can create and delete attributes within an instance, there are certain standard attributes that are always present within each class (and therefore within each instance of that class), which are listed in the table below:

Attribute	*Details*
`__name__`	The name of the class
`__doc__`	The documentation string of the class (see section 12.2)
`__bases__`	A tuple containing the base classes of this class, in the order of inheritance (see section 12.6.5)
`__module__`	The name of the module to which this class belongs (see section 15)
`__dict__`	A dictionary containing the namespace of this class

Table 24: Standard Attributes

These attributes are illustrated in the program below:

StandardAttributesDemo.py

```
1.   #!/usr/bin/python
2.
3.   # Standard Attributes Demo:
4.
5.   class A:
6.       pass
7.
8.   class B:
9.       pass
10.
11.  class C(A,B):
12.      "A class to demonstrate standard attributes"
13.
14.      x=10
15.
16.      def __init__(self):
17.          pass
18.
19.      def f(self):
20.          pass
21.
22.  print("__name__:",C.__name__)
23.  print("__doc__:",C.__doc__)
24.  print("__bases__:",C.__bases__)
25.  print("__module__:",C.__module__)
26.  print("__dict__:",C.__dict__)
```

Output:

```
__name__: C
__doc__: A class to demonstrate standard attributes
__bases__: (<class '__main__.A'>, <class '__main__.B'>)
__module__: __main__
__dict__: {'f': <function C.f at 0x7f92c18564d0>, '__module__':
'__main__', '__doc__': 'A class to demonstrate standard
attributes', '__init__': <function C.__init__ at
0x7f92c1856440>, 'x': 10}
```

Observation:

1. We have defined class `A` in line 5 and class `B` in line 8. Class `C` (defined in line 11) derives from class `A` and class `B`. Class `C` contains a constructor and a method `f`. It also contains a class variable `x`.
2. We observe that the `__name__` attribute correctly gives us the class name as `C`.
3. We observe that the `__doc__` attribute picks up the documentation string from the class `C`.
4. We observe that the `__bases__` attribute has identified class `A` and class `B`, both in the module `__main__`, to be the base classes.
5. We observe that the module within which class `C` is present is `__main__`.
6. Finally, the attribute `__dict__` is a dictionary containing standard attributes as well as user-defined ones. Specifically, we observe that the dictionary contains the method `f`, the constructor and the class variable `x`.

12.8.2 Magic Functions

We use the term "*Magic Functions*" to denote those functions/methods that are automatically invoked without us explicitly naming them! All of these magic functions have the speciality that they have the following syntax for their name:

```
__name__
```

12.8.2.1 Constructors and Destructors

Here are some examples of magic functions that we have already used:

1. Whenever we create an object, it's constructor (`__init__`) is automatically invoked!
2. Whenever we delete an object reference using `del`, it's destructor (`__del__`) is automatically invoked!

Observe in the examples above that the functions being called have names that both start and end with double underscores and that we never explicitly called them! Here is a proof:

```
>>> class A:
...     def __init__(self): print("Created")
...     def __del__(self): print("Destroyed")
...
>>> a=A()
Created
>>> del(a)
Destroyed
```

Observation:

1. When we created an instance of class `A` using the statement `a=A()`, we observe that the `__init__()` method was automatically invoked!
2. When we deleted the variable a using the statement `del(a)`, we observe that the `__del__` method was automatically invoked!

12.8.2.2 Stringification

Since many a times objects are instances of custom user-defined classes that the Python interpreter had no idea about before your program could start execution, we find it convenient to have a string representation for such objects, especially when we want to display objects to the user or log them to a file. We use the term "*stringification*" to describe the conversion of an object to a string, and we use the following syntax to perform stringification:

```
str(object)
```

Note:

1. This is the constructor of the string class (`str`) that we had discussed in section 2.3.4.1.
2. The above statement does not work directly for user-defined classes, but can be made possible using the concept explained below.

Any attempt to stringify an object will automatically call the magic method `__str__` of that object. If the object does not contain this method, it will use the version provided by the (nearest) super class, calling this function of the `object` class in the worst case. This is demonstrated in the example below:

```
>>> class A: pass
...
>>> a=A()
>>> str(a)
'<__main__.A object at 0x7f390c039d30>'
```

Observation:

1. We created an empty class `A` and created an instance of it, the reference to which was stored in the variable `a`.
2. We see that though the class `A` did not provide an implementation for the `__str__` method, there was no error and we continue to get a string version due to the implementation in the `object` class.

Let us add a method by name `__str__` in class `A` to return a string and observe the behaviour:

```
>>> class A:
...     def __str__(self):
...             return "Hi"
...
>>> a=A()
>>> str(a)
'Hi'
```

Observation:

1. This time, our class `A` contains an implementation for the `__str__` method, which is designed to return the string "`Hi`" when invoked.
2. We create an instance of class `A` and store it's reference in the variable `a`.
3. We observe that `str(a)` ends up returning the string "`Hi`", proving that `str(a)` ended up making the call `a.__str__()`.

Some of the sample programs in section 12.10 will make use of this to provide appropriate string representations of objects of user-defined classes.

12.8.3 Operator Overloading

Most of the basic operators can work even with objects of user-defined classes if the operation methodology is taught to the Python interpreter using the concept of *operator overloading*.

12.8.3.1 Overloading Basic Arithmetic Operators

To illustrate the concept, if we have an object `o` and an integer `i`, it is possible to do operations like `o + i` if we overload the operator `+` in the class to which the object `o` belongs.

Here is an example without operator overloading:

```
>>> class A: pass
...
>>> o=A()
>>> i=5
>>> o+i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'A' and 'int'
```

Observation:

1. We have defined an empty class `A`.
2. The variable `o` refers to an instance of class `A` and the variable `i` refers to an instance of `int` class with value `5`.
3. Due to the fact that we have not overloaded the `+` operator in class `A`, we see that the operation `o+i` does not work.

To support overloading of the `+` operator, we need to add the method `__add__` in class `A` as follows:

```
>>> class A:
...     def __add__(self, x): pass
...
>>> o=A()
>>> i=5
>>> o+i
>>>
```

Observation:

1. This time we observe that `o+i` does not give any error. This is because `o+i` results in the following call: `o.__add__(i)`.
2. Our `__add__()` method did not do anything since this is a demonstration. For the same reason, the method also did not return anything.

Some programs in section 12.10 will demonstrate practical uses of overloading operators like these. The table below summarises the magic functions for overloading basic arithmetic operators:

Arithmetic Operator	*Magic Function*
+	__add__
-	__sub__
*	__mul__
//	__floordiv__
/	__truediv__
%	__mod__
divmod()	__divmod__
**, pow()	__pow__

Table 25: Magic Functions for Arithmetic Operators

While we know that it is now possible to evaluate `o+i` since it maps on the `o.__add__(i)`, the question now is what about evaluating expressions like `i+o`? Since `i` is a reference to the built-in `int` class, there is no way that class can define such an operation to work on an instance of a user-defined class, as illustrated below:

```
>>> class A:
...     def __add__(self, x): pass
...
>>> o=A()
>>> i=5
>>> i+a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

The solution is additional magic methods that work in the reverse order! Thus, `i+o` can be implemented using `o.__radd__(i)`, where `__radd__` is the "reverse addition" magic method, as illustrated below:

```
>>> class A:
...     def __radd__(self, x): pass
...
>>> o=A()
>>> i=5
>>> i+o
>>>
```

Observation:

1. We observe that `i+o` works without errors since it maps on to `o.__radd__(i)`.
2. The `__add__` and `__radd__` magic methods are related, but neither relies on the other and it is not mandatory to have both, though possible and in many cases recommended.
3. Our `__radd__()` method did not do anything since this is a demonstration. For the same reason, the method also did not return anything.

The table below lists the magic functions to perform reverse arithmetic operations:

Reverse Arithmetic Operator	*Magic Function*
+	__radd__
-	__rsub__
*	__rmul__
//	__rfloordiv__
/	__rtruediv__
%	__rmod__
divmod()	__rdivmod__
**, pow()	__rpow__

Table 26: Magic Functions for Reverse Arithmetic Operators

12.8.3.2 Overloading Unary Arithmetic Operators

Just as how section 12.8.3.1 introduced magic methods for basic binary arithmetic operators, the table below summarises the magic methods for basic unary arithmetic operators:

Unary Arithmetic Operator	Magic Method
`-`	`__neg__`
`+`	`__pos__`
`abs()`	`__abs__`

Table 27: Magic Methods for Unary Arithmetic Operators

NOTE:

Unlike the operators shown in section 12.8.3.1, *reverse unary arithmetic operators* don't exist since these operators, being unary, work on a single operand and that operand is the instance of the class itself!

12.8.3.3 Overloading Type Conversion Operators

When objects of our class are being converted to other primitive types, certain standard magic methods are invoked to facilitate the conversion, if defined. These magic methods are summarised in the table below:

Conversion	Magic Method
`int()`	`__int__`
`float()`	`__float__`
`bool()`	`__bool__`
`complex()`	`__complex__`
`str()`	`__str__`
`bytes()`	`__bytes__`
`index(), bin(), oct(), hex()`	`__index__`

Table 28: Magic Methods for Type and Base Conversion

Note:

1. The `__str__` magic method was already covered formally in section 12.8.2.2.
2. The `__index__` magic method is supposed to return the same value as `__int__`, and the presence of `__index__` should ideally also be accompanied by `__int__`, though the presence of `__int__` does not mandate implementing `__index__`!
3. The `__index__` magic method is used when converting the integer equivalent of an instance to other bases, as performed by the `bin()`, `oct()` and `hex()` functions. All these functions use the return value of `__index__` and perform base conversions themselves.

12.8.3.4 Overloading Comparison Operators

In order to compare two objects, the relational operators can be used provided suitable magic methods are implemented as summarised in the table below:

Comparison Operator	*Magic Method*
<	`__lt__`
>	`__gt__`
<=	`__le__`
>=	`__ge__`
==	`__eq__`
!=	`__ne__`

Table 29: Magic Methods for Comparison Operators

12.8.3.5 Overloading Bitwise Operators

When bitwise operators are used on objects, corresponding magic methods get invoked. The table below summarises the magic methods invoked when the first operand is an instance of our class:

Bitwise Operator	*Magic Method*
`&`	`__and__`
`|`	`__or__`
`^`	`__xor__`
`<<`	`__lshift__`
`>>`	`__rshift__`
`~`	`__invert__`

Table 30: Magic Methods for Bitwise Operators

If the first operand does not support the required bitwise operation, then the second operand's reverse magic method gets invoked as summarised in the table below:

Reverse Bitwise Operator	*Magic Method*
`&`	`__rand__`
`|`	`__ror__`
`^`	`__rxor__`
`<<`	`__rlshift__`
`>>`	`__rrshift__`

Table 31: Magic Methods for Reverse Bitwise Operators

Note:

1. If the second operand also does not support the reverse bitwise operation, an error is reported.
2. The `~` operator, being unary, does not have a corresponding reverse operator.

12.8.3.6 Overloading Assignment Operators

Note that in statements of the form `o1=o2`, where `o1` and `o2` are objects of any class, the assignment operator (`=`) cannot be overloaded as Python takes up the responsibility of assigning the reference from RHS to LHS. However, for the shorthand assignment operators, we do have corresponding magic methods that get invoked, as summarised in the table below:

Shorthand Assignment Operator	Magic Method
+=	__iadd__
-=	__isub__
*=	__imul__
//=	__ifloordiv__
/=	__itruediv__
%=	__imod__
**=	__ipow__
&=	__iand__
|=	__ior__
^=	__ixor__
<<=	__ilshift__
>>=	__irshift__

Table 32: Magic Methods for Shorthand Assignment Operators

NOTE:

These magic methods ideally should return a reference to a new instance of the class that represents the result of the operation.

12.9 Empty Classes

An empty class in Python can be useful that way for a variety of reasons. An empty class is defined as a class containing only 1 statement: `pass` as shown in the syntax below:

```
class className:
    pass
```

This section covers some of the valid reasons why we might encounter empty classes:

12.9.1 Empty Classes as Placeholders

Many a times while coding, the programmer wants to focus on one aspect of the code but does not want syntax errors due to absence of code in other parts of the program. A programmer who needs a particular class, but does not want to commit on it's attributes and behaviour at the moment can easily start with an empty class and fill in the class details at a later point of time. Even when the class details are not implemented, the class as such is usable and instances of that class can be created and used as well.

12.9.2 Empty Classes for Identification

Sometimes, the only reason we want a particular class is to differentiate the objects. This happens, for example, in exception handling where we want to process different exceptions in different manners. Such a class need not have any implementation at all and we can make do with an empty definition.

> **NOTE:**
>
> The example in this section involves exception handling which will be covered in section 13. Readers are advised to read this section only after completing section 13.

Here is an example illustrating operations on a stack implemented using a user-defined class `MyStack`, in which the `push()` method can raise `StackOverflowException` and the `pop()` method can raise `StackUnderflowException`:

`MyStack.py`

```
1.   #!/usr/bin/python
2.
3.   # Implementation of a fixed-length Stack
4.
5.   class MyStack:
6.
7.       def __init__(self,MAX_SIZE):
8.           self.MAX_SIZE = MAX_SIZE
9.           self.values=[]
10.
11.      def push(self,x):
12.          if len(self.values) == self.MAX_SIZE: raise
StackOverflowException()
13.          self.values.append(x)
14.
15.      def pop(self):
16.          if len(self.values) == 0: raise
StackUnderflowException()
```

```
17.          return self.values.pop()
18.
19.      def display(self):
20.          for i in self.values: print(i)
21.
22.  class StackOverflowException(Exception): pass
23.  class StackUnderflowException(Exception): pass
24.
25.  myStack = MyStack(3)
26.
27.  while True:
28.      try:
29.          print("1. Push")
30.          print("2. Pop")
31.          print("3. Display")
32.          print("4. Quit")
33.          choice = int(input("Enter choice:"))
34.
35.          if choice == 1: myStack.push(int(input("Enter
value to push:")))
36.          elif choice == 2: print("Popped:",myStack.pop())
37.          elif choice == 3: myStack.display()
38.          elif choice == 4: break
39.          else: print("Invalid Choice!")
40.      except StackOverflowException:
41.          print("Stack overflow!")
42.      except StackUnderflowException:
43.          print("Stack underflow!")
44.      except ValueError:
45.          print("Invalid number!")
46.
```

Output:

```
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:10
1. Push
```

```
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:20
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:40
Stack overflow!
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
10
20
30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Popped: 30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
10
20
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Popped: 20
1. Push
2. Pop
```

```
3. Display
4. Quit
Enter choice:2
Popped: 10
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Stack underflow!
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
Enter choice:4
```

Let us analyse the output in pieces:

```
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
```

When we start with an empty stack, we first have used option `3` to verify that we indeed have an empty stack. Let us now add items to the stack:

```
Enter choice:1
Enter value to push:10
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:20
1. Push
```

```
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:40
Stack overflow!
1. Push
2. Pop
3. Display
4. Quit
```

As can be seen, we added `10`, `20` and `30`. But when we attempt to add `40`, we get a "`Stack overflow!`" message since the maximum size of our stack is `3`! We will now use option `3` to display the stack and observe the stack contents:

```
Enter choice:3
10
20
30
1. Push
2. Pop
3. Display
4. Quit
```

We see that `10`, `20` and `30` are in the stack, but not the rejected value `40`. Let us pop a single item and display the stack contents to verify that the popped item is indeed removed from the stack:

```
Enter choice:2
Popped: 30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
10
20
1. Push
```

```
2. Pop
3. Display
4. Quit
```

The value `30` was popped out from the stack and is no longer in the stack. Let is pop out all items one by one:

```
Enter choice:2
Popped: 20
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Popped: 10
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Stack underflow!
1. Push
2. Pop
3. Display
4. Quit
```

We get a "`Stack underflow!`" message when we try to pop out an item from an empty stack. Let us verify that the stack is indeed empty:

```
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
```

Finally, let us quit the program:

```
Enter choice:4
```

Observation:

1. We have define the `MyStack` class from line 5.
2. The constructor defined in line 7 accepts `MAX_SIZE` – the maximum size of the stack – and stores it in the instance variable `MAX_SIZE`. The constructor also creates an empty list to house the stack items and stores it in the instance variable `values`.
3. The `push()` method defined in line 11 accepts a value to be pushed on to the stack. It first checks whether the stack is already full, and if so raises `StackOverflowException` instead. Otherwise it appends the given value to the list of stack values.
4. The `pop()` method defined in line 15 is supposed to pop out an item from the stack and return it. First the method checks if the stack is empty, raising `StackUnderflowException` in that case, else it pops out an element from the list of stack values and returns the element.
5. The `display()` method defined in line 19 displays the contents of the stack by iterating through the list of stack values.
6. The exceptions being raised in lines 12 and 16 are instances of classes `StackOverflowException` and `StackUnderflowException`, defined in lines 22 and 23 respectively. Recall from section 13.6 that such classes need to derive from the `Exception` class. Since no other functionality is required, the classes are empty!
7. Lines 40 and 42 show why we needed to differentiate between these classes for identification of the exception.

12.9.3 Empty Classes as Base Classes

When we have a class hierarchy and feel that certain classes “belong” to the same group, we decide to inherit them from a common base class to establish this logical belongingness. In such cases, it is perfectly valid for the common base class to have no implementation, if the programmer does not see any need. Thus, empty classes are used as base classes from which many related classes can derive.

We can change the program `MyStack.py` discussed in the previous section and ensure that the classes `StackOverflowException` and `StackUnderflowException` both derive from the common class `StackException` for better logical clarity!

MyStack2.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of a fixed-length Stack
4.
5.   class MyStack:
6.
7.       def __init__(self,MAX_SIZE):
8.           self.MAX_SIZE = MAX_SIZE
9.           self.values=[]
10.
11.      def push(self,x):
12.          if len(self.values) == self.MAX_SIZE: raise
StackOverflowException()
13.          self.values.append(x)
14.
15.      def pop(self):
16.          if len(self.values) == 0: raise
StackUnderflowException()
17.          return self.values.pop()
18.
19.      def display(self):
20.          for i in self.values: print(i)
21.
22.  class StackException(Exception): pass
23.  class StackOverflowException(StackException): pass
24.  class StackUnderflowException(StackException): pass
25.
26.  myStack = MyStack(3)
27.
28.  while True:
29.      try:
30.          print("1. Push")
```

```
31.          print("2. Pop")
32.          print("3. Display")
33.          print("4. Quit")
34.          choice = int(input("Enter choice:"))
35.
36.          if choice == 1: myStack.push(int(input("Enter
value to push:")))
37.          elif choice == 2: print("Popped:",myStack.pop())
38.          elif choice == 3: myStack.display()
39.          elif choice == 4: break
40.          else: print("Invalid Choice!")
41.      except StackOverflowException:
42.          print("Stack overflow!")
43.      except StackUnderflowException:
44.          print("Stack underflow!")
45.      except NumberFormatException:
46.          print("Invalid number!")
```

Observation:

1. This program is identical to `MyStack.py`, with changes in lines 22-24.
2. The output of the program is not shown as it is identical to the output of `MyStack.py`.
3. We decided to derive `StackOverflowException` and `StackUnderflowException` from `StackException` as we feel these 2 exception classes are logically related. The relation is established using the common base class `StackException`.
4. Due to exception handling rules (section 13.6), we need to ensure that `StackException` extends `Exception`. But apart from this, we have no need for adding any other piece of code within class `StackException`.
5. We can now also catch both `StackOverflowException` and `StackUnderflowException` using `StackException` in the `except` clause, if needed!

12.9.4 Empty Classes as Data Types

Since instance variables are not defined within classes, it is possible to start with an empty class but add instance variables directly in the instances themselves as shown in the example below:

```
>>> class A:
...     pass
...
>>> a=A()
>>> a.x=10
>>> a.y=20
>>> a.x
10
```

In such cases, having an empty class does not mean that the instances are empty!

Note for C/C++ Programmers:

The equivalent of the `struct` data type of C is an empty class whose instances can have any field required.

12.10 Programs based on OOP

This section shows some sample implementation of classes and concepts covered in this section.

12.10.1 Implementation of Counter Class

A counter is a mechanism for keeping track of an integer count that be incremented, decremented, increased, decreased and displayed. Here is an implementation of such a counter.

Counter.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Counter Class
4.
5.   class Counter:
6.       def __init__(self, count=0): self.set(count)
7.
8.       def set(self, count): self._count = count
9.       def get(self): return self._count
10.      def reset(self): self.set(0)
11.
12.      def increment(self, count=1): self.set(self.get()
+count)
13.      def decrement(self, count=1): self.set(self.get()-
count)
14.
15.      def __add__(self, x): return Counter(self.get()+x)
16.      def __radd__(self, x): return self.__add__(x)
17.      def __sub__(self, x): return Counter(self.get()-x)
18.      def __rsub__(self, x): return self.__sub__(x)
19.      def __iadd__(self, x):
20.          self.increment(x)
21.          return self
22.      def __isub__(self, x):
23.          self.decrement(x)
24.          return self
25.
26.      def __str__(self): return str(self.get())
27.      def __int__(self): return self.get()
28.      def __index__(self): return self.__int__()
29.
30.  # Basic counter setting/getting/resetting
31.  c = Counter()
32.  print(c.get())
33.  c.set(9)
34.  print(c.get())
35.  c.reset()
36.  print(c.get())
37.
38.  # Basic counter increment and decrement
39.  c = Counter(100)
40.  print(c.get())
41.  c.increment()
42.  c.increment(5)
43.  print(c.get())
```

```
44.  c.decrement()
45.  c.decrement(3)
46.  print(c.get())
47.
48.  # Basic counter operators
49.  c = Counter()
50.  c = c + 2
51.  c = 3 + c
52.  c += 5
53.  print(c.get())
54.  c = Counter()
55.  c = c - 2
56.  c = 3 - c
57.  c -= 5
58.  print(c.get())
59.
60.  # Counter type conversions
61.  c = Counter(12)
62.  print(int(c))
63.  print(str(c))
64.  print(c)
```

Output:

```
0
9
0
100
106
102
10
-10
12
12
12
```

Observation:

1. The `Counter` class is defined in line 5. The constructor in line 6 can optionally receive a count and store it within an instance variable `_count`. If this value is not given, it will be assumed to be `0`. The instance variable is *protected*, making it clear that the external world is not supposed to meddle with it directly but instead use *public* methods to gain access to it. These methods are basically `get()`, `set()` and `reset()`.
2. The `set()` method in line 8 accepts a count and sets it within the instance

variable `_count`.

3. The `get()` method in line 9 returns the count associated with the invoking object, stored in the instance variable `_count`.
4. The `reset()` method in line 10 resets the counter value back to `0`.
5. We see that the constructor is calling the `set()` method in line 6. Similarly, the `reset()` method is calling the `set()` method in line 10. We prefer reuse of method this way, even though the efficiency drops, since it makes the code more reliable and maintainable. For example, if we decide to change the name of the instance variable `_count` to `count` or `__count` or any other name, the change will be localised to `set()` (and `get()` too), but will have no impact on the constructor and `reset()` methods! We will also see to it that we don't reference the instance variable `_count` in any of our methods, preferring to use the `set()` and `get()` methods instead to indirectly access it.
6. The `increment()` method in line 12 increments the count of the invoking object by the specified value (default `1`). Again, this is done by calling the `set()` method. The `decrement()` method in line 13 similarly decrements the count of the invoking object by the specified value (default `1`).
7. Basic operator overloading is implemented in lines 15-24. Again, the focus is on reusability of methods rather than re-implementing within each method.
8. The `__add__()` method in line 15 is supposed to handle cases of the form `c+x`, where `c` is a `Counter` object and `x` is an integer. It is supposed to return the result of the expression, which we know should be a `Counter` instance.
9. The `__radd__()` method in line 16 is supposed to handle cases of the form `x+c`, where `c` is a `Counter` object and `x` is an integer. It is supposed to be identical to `c+x` in behaviour.
10. The `__sub__()` method in line 17 and `__rsub__()` method in line 18 similarly handle the cases `c-x` and `x-c` respectively where `c` is a `Counter` object and `x` is an integer.
11. The `__iadd__` method in line 19 handles the case `c+=x`, where `c` is a `Counter` object and `x` is an integer. Unlike the `__add__()` method which returns the result, here the result has to be stored within the invoking object itself, and we prefer to return a reference to the invoking object to support cascading of operations (not demonstrated here).
12. The `__isub__()` method in line 20 similarly handles the case `c-=x`, where `c` is a `Counter` object and `x` is an integer.
13. The methods in lines 26-28 handle conversion to other types. The `__str__()` method in line 26 helps convert `Counter` objects to strings. The `__int__()` method in line 27 helps convert `Counter` objects to integers.

The `__index__()` method in line 28 helps convert `Counter` objects to other bases if required (not demonstrated here).

14. The implementation of `__index__()` is supposed to be the same as that of `__int__()`.
15. The implementation of `__str__()` makes it possible for us to directly print `Counter` objects in a `print()` call by saying `print(c)`, where `c` is a `Counter` object.

12.10.2 Implementation of Distance Class

Here is an implementation of Distance class that can help represent any distance in a variety of units, and also supports the addition and subtraction of distances. The supported units currently are meters (m), kilometers (km), miles (mi), yards (yd) and feet (ft), though any other unit can be very easily added with minimal modification!

Distance.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Distance Class
4.
5.   class Distance:
6.       _factors = {"m":1, "km":1000, "mi":1609.34,
"yd":0.9144, "ft":0.3048}
7.       def __init__(self, distance=0, unit="m"):
8.           self.set(distance, unit)
9.
10.      def set(self, distance, unit="m"):
11.          self._distance = Distance._normalize(distance,
unit)
12.
13.      def get(self, unit="m"):
14.          return Distance._externalize(self._distance, unit)
15.
16.      def _normalize(distance, unit):
17.          if unit == "m": return distance
18.          return distance * Distance._factors[unit]
19.
20.      def _externalize(distance, unit):
21.          if unit == "m": return distance
22.          return distance/Distance._factors[unit]
23.
24.      def add(self, distance, unit="m"):
25.          self.set(self.get() +
Distance._normalize(distance, unit))
26.
```

```
27.      def sub(self, distance, unit="m"):
28.          self.set(self.get() -
Distance._normalize(distance, unit))
29.
30.      def __int__(self): return int(self._distance)
31.      def __float__(self): return self._distance
32.      def __str__(self): return str(self._distance)
33.
34.  # Distance getting/setting in various units
35.  d = Distance(1000) #1000m = 1km
36.  print(d.get()) #1000m
37.  print(d.get("km")) #1km
38.  d.set(2,"km") #2km = 2000m
39.  print(d.get()) #2000m
40.  print(d.get("km")) #2m
41.  print(Distance(12,"ft").get("ft")) #12ft = 12ft
42.
43.  # Distance addition and subtraction
44.  d = Distance(500) #500m
45.  d.add(0.6,"km") #0.6km = 600m, 500+600=1100m
46.  print(d.get()) #1100m
47.  d.sub(1000) #1100-1000 = 100m
48.  print(d.get("km")) #100m = 0.1km
49.
50.  # Conversion to int and str
51.  d = Distance(500)
52.  print(int(d)) #500m -> 500
53.  print(str(d)) #500m -> "500"
54.  print(d) #d -> str(d)
```

Output:

```
1000
1.0
2000
2.0
12.0
1100.0
0.1
500
500
500
```

Observation:

1. The `Distance` class is implemented in lines 5-31. The constructor defined in

line 7 is capable of accepting an optional distance (`0` by default) in any unit required (meters by default). The assignment is made using the `set()` method.

2. The `set()` method defined in line 10 is responsible for assigning the specified distance to the instance variable `_distance`, after converting it to a standard unit (meters) using the method `_normalize()` for conversion (explained later).
3. The `get()` method defined in line 13 returns the distance represented by the invoking object in the specified unit (meters by default), converting the units using the method `_externalize()` (explained later).
4. Units conversion is important here as we are dealing with various units. We have stuck to meters as being a standard unit, though the program will behave exactly the same irrespective of which unit is chosen to be the standard! Based on the standard chosen, the *class variable* `_factors` (line 6) is populated as a dictionary with the keys as unit abbreviations and values as the multiplication factor that is to be used to convert that unit to the standard unit. Thus, to convert kilometers (km) into meters (m), the multiplication factor is `1000` and to convert miles (mi) into meters (m), the multiplication factor is `1609.34`. Obviously, to convert meters into meters, the multiplication factor is `1`! More units can be added here as required! Note that we have made this a class variable because it is common to all instances of this class!
5. The `_normalize()` method defined in line 16 is a *class method* that helps convert distances from other units to the standard unit (meters). As a special case, this method does not bother to convert units if it is already in meters. This is merely an *optimization* and conversion of units in such a case is not wrong, only unnecessary! The unit conversion is done using the class variable `_factors`.
6. The `_externalize()` method defined in line 20 is a class method that helps convert distances from meters to other units. This is the opposite function of `_normalize()`!
7. The `add()` method defined in line 24 adds the specified distance in the specified units (meters by default) to the invoking object.
8. The `sub()` method defined in line 27 similarly subtracts the specified distance in the specified units (meters by default) from the invoking object.
9. The `__int__()` method defined in line 30 converts the distance in meters to an integer; the `__float__()` method defined in line 31 merely returns the distance in meters (already a `float`/`int`) and the `__str__()` method defined in line 32 converts the distance in meters to a string.

12.11 Questions

1. Explain any 5 principles of OOP.
2. What does a class contain typically in Python?
3. Differentiate between class functions and instance methods.
4. Write a short note on constructors and destructors in Python.
5. Write a short note on magic methods.
6. Write a short note on attribute handling in Python.
7. Write a short note on overloading arithmetic operators on Python.
8. Write a short note on base conversion of objects in Python.
9. Explain inheritance with an example in Python.
10. Explain dynamic polymorphism with an example in Python.
11. How can empty classes be useful in Python? Explain with examples.

12.12 Exercises

1. Write a script in Python to implement the Queue data structure using OOP.
2. Write a script in Python to implement 3 different shapes as individual classes and calculate their area using dynamic polymorphism.
3. Write a script in Python to demonstrate the overloading of the `or` operator.
4. Write a class called `Time` that helps represent the time of day in 24-hour format. Add these functionalities:
    1. Creation of a `Time` object, given the hour, minute, second and millisecond.
    2. Extraction of the different pieces of the `Time` object.
    3. Addition of the specified absolute number of milliseconds to a `Time` object.
    4. Comparison of two `Time` objects to find if they are identical.
    5. Comparison of two `Time` objects to find which comes earlier.

SUMMARY

- A Class is a design according to which objects can later be instantiated. An Object is an instance of a class.
- Data Encapsulation is the encapsulation of data and the code that acts on the data into a single unit. Data Hiding is the hiding of data from the external world with the intention of protecting the object against accidental changes, thereby maintaining the integrity of the object. Data Abstraction is the hiding of the implementation details and revealing of only an interface.
- Polymorphism refers to the existence of multiple forms of the same entity. Inheritance is the mechanism where one class derives all features of another existing class.
- Classes can be defined using the class keyword. Objects can be created using the syntax ClassName().
- Instance variables can be created explicitly using an object, or by an instance method, and most often by the constructor.
- Class variables are defined directly within the class and can be accessed by all instances of the class or by explicitly using the class name to gain access it.
- Instance methods are methods that automatically work on the invoking object. Such methods will automatically receive a reference to the invoking method as the first parameter. This parameter is usually called self by programmers.
- Class Functions are functions defined within the class that are not dependent on the invoking object.

SUMMARY

- All members within a class are assumed to be public by default. Those identifiers that start with a single underscore are assumed to be protected and those that start with 2 underscores and do not end with 2 or more underscores are assumed to be private. Python currently does not have implementation support for protected members.
- Constructors are identified by the special name __init__() while destructors are identified by the special name __del__().
- Python supports multiple inheritance too!
- All functions undergo late binding. Therefore, no special step is required to achieve dynamic polymorphism.
- The method super() can be used to reference the immediate/first base class instance.
- Constructors and Destructors of the base class are not automatically invoked during inheritance - it is the responsibility of the programmer.
- Magic functions are automatically invoked in certain situations and help implement a variety of functionality, including operator overloading. Such functions always start and end with 2 underscores!

13 Exception Handling

In this chapter you will be able to:

- ☑ Appreciate the Object Oriented Exception Handling mechanism used by Python
- ☑ Understand the exception handling block and process exceptions that occur.
- ☑ Understand the flow of control when exceptions do and don't occur.
- ☑ Create custom exception classes and create and handle custom exceptions.

Exception Handling

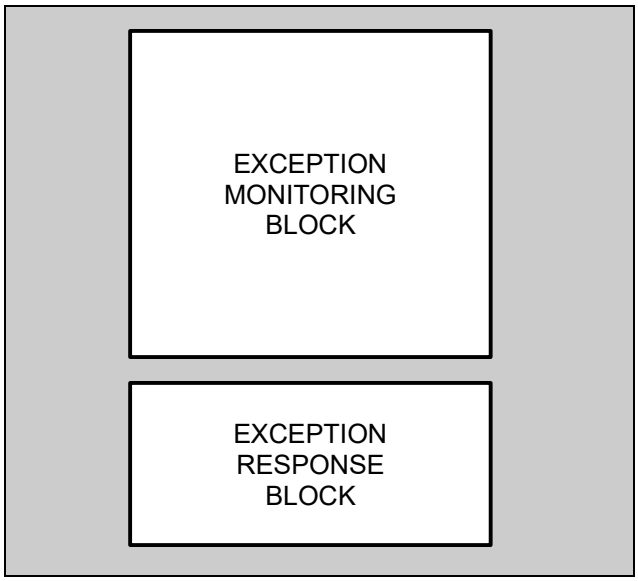
13.1 Errors vs. Exceptions

Every programmer would have definitely written programs that didn't work as expected. Sometimes, we programmers violate the syntactic rules of the language resulting in syntax errors and at other times our programs don't work correctly due to various runtime anomalies. While syntax errors are definitely the programmer's fault, various runtime anomalies can be caused due to reasons beyond the instructions in the program. For example, the program may run out of memory, or is unable to open a file, or unable to connect to a server, or the user has given invalid input. In these situations, it would be wrong to blame the programmer for the issue, though a good programmer always anticipates these issues and handles them gracefully in the program. Object Oriented programming languages like Python provide a very well-defined mechanism for dealing with these issues.

Do note however that some of Python's exception classes have names that indicate Error, but are technically exceptions! For instance, `ZeroDivisionError` is the name of an exception that is generated when we perform integer division by 0. On the other hand, `SyntaxError`, which is an error, is derived from `Exception` class and is treated as an exception. Some things are certainly strange in Python!

13.2 Handling Exceptions

As mentioned in the previous section, a good programmer anticipates situations where the program will not be able to perform as expected. Such blocks of code need to be *monitored* for exceptions. A good programmer handles these exceptions gracefully and thus has to *respond* to exceptions as they arise. This is done in a response block. Thus, the code will have the following structure:



The advantages of this mechanism are:

1. We are able to separate the exception testing code (checks in the monitoring block) and the exception handling code (the response block). This makes the code in the monitoring block easier to understand and maintain.
2. We are able to respond to multiple exceptions without multiple checks. All statements in the monitoring block are monitored. This simplifies our code and makes it easy to maintain.
3. We are able to respond to different exceptions differently, or deal with multiple types of exceptions in an identical manner, based on our requirement (section 13.2.2).
4. We are able to deal with exceptions in different levels or in multiple levels (section 13.4).

13.2.1 The Basic Exception Handling Block

The previous section introduced the exception monitoring block and the exception handling block. These blocks are called the `try` block and `except` block respectively in Python, as shown in the syntax below:

```
try:
    ...
except exceptionName:
    ...
```

Let us understand the flow of control in the above syntax. Since the flow of control depends on whether or not an exception is raised in the `try` block, we will consider both situations separately:

1. If no exception is raised in the `try` block, the statements in the `try` block are executed sequentially and the entire `except` block is ignored. This is what we normally expect most of the times, as exceptions are supposed to be raised rarely.

2. If an exception is raised anywhere within the `try` block during the normal sequential execution, control immediately exits the `try` block and starts examining the `except` block(s). If it finds an `except` block capable of handling the exception that was raised, then that `except` block is executed and control resumes outside of the `try-except` blocks. We will cover more of this in the next section where we see how to handle exceptions of multiple types. In the syntax above, the `except` block is capable of handling an exception of type `exceptionName` (and also it's derived classes as we learnt in section 12.7).

3. If an exception is raised anywhere within the `try` block during the normal sequential execution, and that exception is not handled by the `except` block(s), then control exits the entire `try-except` construct and the exception propagates. This

behaviour will be examined in detail in section 13.4.

Let us take a simple example to understand the simplest syntax of `try-except` shown above.

ExceptionHandlingDemo1.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Simple try-except block
5.
6.   try:
7.       n = int(input("Enter an integer:"))
8.       quotient = int(100/n )
9.       print("100/{} = {}".format(n,quotient))
10.  except ZeroDivisionError:
11.      print("Sorry! Division by zero is not permitted!")
12.
13.  print("Thanks for using this program!")
```

Output:

```
Enter an integer:5
100/5 = 20.0
Thanks for using this program!
```

Observation:

1. This program basically accepts an integer from the user, divides `100` by the given integer and displays the quotient.
2. We however need to be aware that if the user provides `0` as the input, then the division will result in a `ZeroDivisionError` being raised. We need to handle that case gracefully.
3. The `try` block extends from line 7 to line 9. The statements within these are monitored for any exception that may be raised. We are currently expecting `ZeroDivisionError` to be raised when the user input is `0`. Technically, lines 7 and 9 need not be inside the try block as they are not candidates for `ZeroDivisionError` to be raised. We have still kept them inside the try block for convenience and because the subsequent demo programs build upon the same program and we will require these statements also to be monitored for exceptions.
4. The `except` block contains line 11. This is our response to `ZeroDivisionError`.

5. Line 7 accepts input from the user, line 8 divides `100` by the input and line 9 prints the result. As can be seen from the output, if the input is `5`, the output is `20`. Since no exception is raised, after execution of the try block, control resumes outside the `try-except` block, in line 12.

We will now consider how the program behaves when the input given is `0`.

Output:

```
Enter an integer:0
Sorry! Division by zero is not permitted!
Thanks for using this program!
```

Observation:

1. When the input received in line 7 is `0`, it results in a `ZeroDivisionError` in line 8. This interrupts the flow of control in the `try` block. Note that line 9 did not get executed.
2. Line 10 is the beginning of an `except` block that is capable of handling `ZeroDivisionError`, which has occurred. Control therefore resumes from line 11, within the `except` block.
3. After execution of the `except` block, control comes out of the `try-except` block and resumes execution from line 12. Thus, line 13 is also executed.

13.2.2 Handling Multiple Exceptions

The previous section showed how an exception that was raised can be handled. This section shows how many such exceptions can be handled using a single `try` block. We will specifically see how to:

1. Handle the same exception from multiple lines using a single handler.
2. Handle multiple exceptions using a single handler.
3. Handle all exceptions using a single handler.
4. How to combine the above to realise our requirements.

13.2.2.1 Separate Handling of Each Exception

An extended syntax of the `try-except` block is shown below:

```
try:
    ... # try block
except ExceptionName1:
    ... # Exception handler for ExceptionName1
except ExceptionName2:
    ... # Exception handler for ExceptionName2
except ExceptionName3:
    ... # Exception handler for ExceptionName3
...
```

From this syntax, it is evident that a single `try` block can be followed by any number of `except` blocks, and each `except` block can handle a specific type of exception.

- If no exception occurs, only the `try` block is executed and the control comes out of the entire construct.
- If an exception is raised by any statement in the `try` block, the control immediately comes out of the `try` block and starts checking the `except` blocks sequentially. The first `except` block that can handle the exception that was raised will then be executed, after which control comes out of the entire construct, ignoring all other `except` blocks.
- If none of the `except` blocks can handle the exception that was raised, then the exception propagates to the next higher level, which we will examine in detail in section 13.4.

Let us revisit `ExceptionHandlingDemo1.py`: if the user does not give us a valid integer as input, then an attempt to convert such a string to an integer using the `int()` function results in `ValueError` being raised. Let us now handle both `ValueError` and `ZeroDivisionError`.

ExceptionHandlingDemo2.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Separate handling of each exception
5.
6.   try:
7.       n = int(input("Enter an integer:"))
8.       quotient = int(100/n)
9.       print("100/{} = {}".format(n,quotient))
10.  except ValueError:
11.      print("Invalid input! Please enter an integer.")
12.  except ZeroDivisionError:
13.      print("Sorry! Division by zero is not permitted!")
14.
15.  print("Thanks for using this program!")
```

Output:

```
Enter an integer:5
100/5 = 20
Thanks for using this program!
```

Observation:

1. When the input is `5`, no exceptions are raised. Therefore, only the `try` block is executed and all the `except` blocks are ignored.

Output:

```
Enter an integer:0
Sorry! Division by zero is not permitted!
Thanks for using this program!
```

Observation:

1. When the input is `0`, as seen in the case of `ExceptionHandlingDemo1.py`, a `ZeroDivisionError` is raised in the `try` block and control immediately comes out of the `try` block and the `except` blocks are examined sequentially.
2. Line 10 is an `except` block that can handle `ValueError` but not `ZeroDivisionError` and is therefore skipped.

3. Line 12 is an `except` block that can handle `ZeroDivisionError` and is hence executed, after which control resumes outside the `try-except` block from line 14.

Output:

```
Enter an integer:hi
Invalid input! Please enter an integer.
Thanks for using this program!
```

Observation:

1. When the input is `hi`, the `int()` function is unable to convert the string to an integer and raises a `ValueError`. Control immediately comes out of the `try` block and starts examining the `except` blocks sequentially.
2. Line 10 is an `except` block that can handle `ValueError` and is hence executed, after which control resumes outside the `try-except` block in line 14.

13.2.2.2 Common Handling of Multiple Exceptions

Now that we know how to handle different exceptions differently, let us consider the case that we may want to handle 2 or more exceptions in a common manner. While we have the liberty of handling different exceptions differently, it is also possible to handle multiple exceptions using a common block of code.

Syntax:

```
try:
    ... # try block
except (ExceptionName1, ExceptionName2 [,ExceptionName3...]):
    ... # except block
```

We observe in the syntax above that we can give a tuple of exception types in the `except` clause instead of a single exception type!

Let us rewrite `ExceptionHandlingDemo2.py` using a common response to both `ValueError` as well as `ZeroDivisionError`:

ExceptionHandlingDemo3.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Common handling of multiple exceptions
5.
6.   try:
7.       n = int(input("Enter an integer:"))
8.       quotient = int(100/n)
9.       print("100/{} = {}".format(n,quotient))
10.  except (ValueError, ZeroDivisionError):
11.      print("Oops! Unable to calculate!")
12.
13.  print("Thanks for using this program!")
```

Output:

```
Enter an integer:5
100/5 = 20
Thanks for using this program!
```

Output:

```
Enter an integer:0
Oops! Unable to calculate!
Thanks for using this program!
```

Output:

```
Enter an integer:hi
Oops! Unable to calculate!
Thanks for using this program!
```

Observation:

1. Line 10 now handles a tuple of exceptions! If the exception raised in the `try` block is `ValueError` or `ZeroDivisionError` (or any of their derived classes), then control will enter line 10. The response to both is identical.

Before we proceed further, do note that we can mix handling different exceptions differently with handling multiple exceptions in a common manner as shown in the sample syntax below:

```
try:
    ... # try block
except ExceptionName1:
    ... # except block
except (ExceptionName2, ExceptionName3, ExceptionName4):
    ... # except block
except (ExceptionName5, exceptionName6):
    ... # except block
except ExceptionName7:
    ... # except block
```

13.2.2.3 Handling All Exceptions

While section 13.2.2.1 showed how to handle specific exceptions, section 13.2.2.2 showed how to handle multiple specific exceptions. But both of these approaches require the programmer to explicitly specify the exception(s) to be handled. What if we do not want to name them and simply tell Python that this is how we wish to handle *all* exceptions, even if we don't know the names of the exceptions? The following syntax demonstrates this possibility:

```
try:
    ... # try block
except ExceptionName1:
    ... # Single exception handler
except (ExceptionName2, ExceptionName3):
    ... # Multiple exception handler
except:
    ... # Handles all exceptions not handled above
```

Observation:

1. As can be seen from the syntax above, the last `except` block does not specify any particular exception and is thus considered to mean any and all exceptions.
2. Such a catch-all handler should be the last `except` clause, if at all present. The reason is that no `except` handlers below this can ever get executed since the `except` blocks are tried sequentially and the catch-all handler can handle any exception.
3. In the above syntax, any exception raised in the `try` block that is not `ExceptionName1`, `ExceptionName2` and `ExceptionName3` (or it's derived classes) will be handled in the catch-all handler.

Do note that the catch-all handler does not require any previous `except` blocks. The following syntax is therefore valid, possible and useful:

```
try:
    ... # try block
except:
    ... # Handle all exceptions here
```

Let us revisit `ExceptionHandlingDemo3.py` and use this feature there.

ExceptionHandlingDemo4.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Handling all exceptions
5.
6.   try:
7.       n = int(input("Enter an integer:"))
8.       quotient = int(100/n)
9.       print("100/{} = {}".format(n,quotient))
10.  except:
11.      print("Oops! Unable to calculate!")
12.
13.  print("Thanks for using this program!")
```

Output:

```
Enter an integer:5
100/5 = 20
Thanks for using this program!
```

```
Enter an integer:0
Oops! Unable to calculate!
Thanks for using this program!
```

```
Enter an integer:hi
Oops! Unable to calculate!
Thanks for using this program!
```

Observation:

1. Any exception generated in the `try` block will now be handled in the `except` block in line 10.
2. We expect `ValueError` and `ZeroDivisionError`, both of which will be handled in line 10.

13.2.3 Dealing with Exception Instances

So far, we have responded to different exceptions in a variety of ways, but haven't considered the specific exception instance. When an exception is raised, there is an exception object behind it. Such an exception object can contain a lot of useful information about the exception that took place! In this section, we will see how we can gain access to the exception object and obtain some more information on the exception.

The following sample syntax shows how we can gain access to the exception instance:

```
try:
    ... # try block
except ExceptionName1 as e:
    ... # Handler ExceptionName1 with instance as e
except (ExceptionName2, ExceptionName3) as e:
    ... # Handle both these exceptions with instance as e
```

Let us modify `ExceptionHandlingDemo4.py` to demonstrate these features.

ExceptionHandlingDemo5.py

```
1.  #!/usr/bin/python
2.
3.  # Exception Handling Demo
4.  # Exception instances
5.
6.  import sys
7.
8.  try:
9.      n = int(input("Enter an integer:"))
10.     quotient = int(100/n)
11.     print("100/{} = {}".format(n,quotient))
12. except Exception as e:
13.     print("Oops! Unable to calculate!")
14.     print("Details:",e)
15.
16. print("Thanks for using this program!")
```

Output:

```
Enter an integer:0
Oops! Unable to calculate!
Details: division by zero
Thanks for using this program!
```

```
Enter an integer:hi
Oops! Unable to calculate!
Details: invalid literal for int() with base 10: 'hi'
Thanks for using this program!
```

Observation:

1. This program is based on `ExceptionHandlingDemo4.py`, but there are 2 changes in line 12 that are important.
2. The first change is that we are responding to exceptions of type `Exception`. Since we want to respond to both `ValueError` as well as `ZeroDivisionError` in the same manner, we would have preferred to use the catch-all handler (section 13.2.2.3). However, the catch-all handler syntax cannot be used to process instances and hence we have to think of an alternative. We utilize the fact that both `ValueError` and `ZeroDivisionError` derive from `Exception`, hence handling `Exception` will allow us to handle them both.
3. The second change in line 12 is that we have used the additional clause "`as e`". This allows us to access the exception instance using `e`. Of course, `e` is just a variable name and we could use any name for accessing the instance.
4. Line 14 prints `e` – it actually ends up stringifying `e` (section 12.8.2.2) and prints that string. We expect the string version of all built-in exceptions to be a human readable message that makes us understand the cause of the exception.
5. We can do more with exception instances, but what we can do depends on the exception. We will cover this aspect in greater detail once we learn to make our own exception classes in section 13.6.

13.2.4 The else Clause

Recollect from section 13.2 that one of the goals of exception handling is to separate normal logic from exception handling logic. This goal has been met by having normal logic code in the `try` section and the exception handling code in one of the `except` blocks.

Python provides an alternative that allows us to split the normal logic block into 2 parts

– a part that is monitored for exceptions (within the `try` block) and a part that is excluded from this monitoring (in the `else` block).

The syntax for implementing this is shown below:

```
try:
    ... # try block, monitored for exceptions
except:
    ... # except block, to respond to exceptions
else:
    ... # else block, not monitored for exceptions
```

Observation:

1. The normal code is now split into 2 parts – one part lies within the `try` block and the other part lies within the `else` part.
2. If any exceptions occur within the `try` block, the suitable `except` block gets executed, but the `else` block is skipped.
3. Only if there were no exceptions in the `try` block, all the `except` blocks are skipped and the `else` block is executed.
4. If any exceptions are raised within the `else` block, they are not handled by this piece of code (but can be handled elsewhere where the exception propagates, as covered in section 13.4).

Let us modify `ExceptionHandlingDemo4.py` and add an `else` block to it.

`ExceptionHandlingDemo6.py`

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # The else clause
5.
6.   try:
7.       n = int(input("Enter an integer:"))
8.       quotient = int(100/n)
9.   except:
10.      print("Oops! Unable to calculate!")
11.  else:
12.      print("100/{} = {}".format(n,quotient))
13.
14.  print("Thanks for using this program!")
```

Output:

```
Enter an integer:5
100/5 = 20
Thanks for using this program!
```

```
Enter an integer:0
Oops! Unable to calculate!
Thanks for using this program!
```

```
Enter an integer:hi
Oops! Unable to calculate!
Thanks for using this program!
```

Observation:

1. This program is based on `ExceptionHandlingDemo4.py`, with line 12 now being inside the `else` block instead of being within the `try` block.
2. The reason why we have separated line 12 from the rest of the `try` block is because we feel that that line has no connection with the exceptions we are expecting – `ValueError` and `ZeroDivisionError`. We do feel that it is part of the normal code of operation and hence it lies in the `else` block as an extension of the `try` block if there were no exceptions.
3. As can be seen from the various output, the behaviour is similar to that of `ExceptionHandlingDemo4.py`. The behaviour would have been different, however, if line 12 were to raise an exception in these 2 programs.

13.2.5 The finally Clause

Sometimes, there is a need to perform some task irrespective of whether an exception occurred or not. Many a times, this is generally some clean-up that needs to be performed whether or not exceptions were raised, like closing files or disconnecting from a server. The `finally` clause helps in guaranteeing the execution of it's contents in all cases – if an exception had occurred and even if it hadn't. The syntax now becomes as follows:

```
try:
    ... # try block
except:
    ... # except block
else:
    ... # else block
finally:
    ... # finally block
```

The flow of control through this construct is shown in the flowchart below. Let us revisit this once again in detail:

1. The exception monitoring block is the `try` block. The exception response is the `except` block. The `else` block is a continuation of the `try` block but without exception monitoring. The `finally` block is a continuation of both the `else` block (or the `try` block if the `else` block is not present) and the `except` block.
2. When this construct is executed, control passes on sequentially through all statements in the `try` block. If any of these statements raise an exception, control comes out of the `try` block and examines the `except` block(s).
3. If it finds a suitable `except` block, then that `except` block gets executed, ignoring all other `except` blocks and control then resumes in the `finally` block, after which control comes out of this construct.
4. If it doesn't find a suitable `except` block capable of handling the exception raised, then the exception has to be propagated to the outer level (covered in section 13.4 in detail). But before propagation, the `finally` block is executed.
5. If no exception is raised in the `try` block, control jumps to the `else` block and continues execution.
6. If any exception is raised in the `else` block, the exception propagates to the outer level (covered in section 13.4 in detail), but only after executing the `finally` block.
7. If no exceptions were raised in the `else` block also, then control resumes in the `finally` block before coming out the construct.
8. If any exceptions are raised in the `finally` block, they are propagated to the outer level (covered in section 13.4 in detail).

The only case when the `finally` block does not get executed is when the script is terminated (section 3.4.4).

Let us apply our understanding of this to `ExceptionHandlingDemo6.py`:

ExceptionHandlingDemo7.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # The finally clause
5.
6.   try:
7.       n = int(input("Enter an integer:"))
8.       quotient = int(100/n)
9.   except:
10.      print("Oops! Unable to calculate!")
11.  else:
12.      print("100/{} = {}".format(n,quotient))
13.  finally:
14.      print("Thanks for using this program!")
```

Output:

```
Enter an integer:5
100/5 = 20
Thanks for using this program!
```

```
Enter an integer:0
Oops! Unable to calculate!
Thanks for using this program!
```

```
Enter an integer:hi
Oops! Unable to calculate!
Thanks for using this program!
```

Observation:

1. This program is based on `ExceptionHandlingDemo6.py` and also produces the same output.
2. Line 14, which is inside the `finally` block, will be executed regardless of whether an exception was raised in the `try` block or not.

13.3 Raising Exceptions

So far we have seen how to handle exceptions when they are raised. We have also seen exactly when some exceptions are raised – for example, we know that `ZeroDivisionError` is raised when we attempt to perform a division by `0`. In this section, we will learn how we can raise exceptions ourselves whenever we want to.

To raise an exception, 2 steps need to be followed:

1. We need to create an exception object that has to be raised. This object will belong to a class that should derive directly or indirectly from the built-in `Exception` class. All built-in exception classes derive from the `Exception` class.
2. We need to use the `raise` statement to raise this exception object, as shown in the syntax below.

```
raise exceptionInstance
```

Example:

```
>>> e=ZeroDivisionError()
>>> raise e
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError
```

While it is necessary to follow the above mentioned 2 steps in order to raise an exception, it is more practical to combine the 2 steps into one by explicitly instantiating the exception class in the `raise` statement itself as shown below:

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

The reasons why we prefer this approach are:

1. We generally need the exception object only to pass it to `raise`. We therefore do not access the variable that references the exception object and using a variable name for this is therefore a waste.
2. We do not prefer to create an exception object in advance and then `raise` it if necessary. Remember that exceptions are supposed to be rare and creating such exception objects in advance is inefficient. Indeed, in many situations, it may be impossible to create an exception object in advance as more

information regarding the exception might be needed in order to instantiate it, and this information would not be available if the exception has not even occurred.

3. We find it more convenient to use this syntax that combines the 2 steps into one.

Let us revisit `ExceptionHandlingDemo2.py` and raise `ValueError` if the given input is greater than `100`:

ExceptionHandlingDemo8.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # The raise statement
5.
6.   try:
7.       n = int(input("Enter an integer:"))
8.       if n>100: raise ValueError()
9.       quotient = int(100/n)
10.      print("100/{} = {}".format(n,quotient))
11.  except ValueError:
12.      print("Please enter an integer less than 100!")
13.  except ZeroDivisionError:
14.      print("Sorry! Division by zero is not permitted!")
15.
16.  print("Thanks for using this program!")
```

Output:

```
Enter an integer:5
100/5 = 20
Thanks for using this program!
```

```
Enter an integer:0
Sorry! Division by zero is not permitted!
Thanks for using this program!
```

```
Enter an integer:hi
Please enter an integer less than 100!
Thanks for using this program!
```

```
Enter an integer:200
Please enter an integer less than 100!
Thanks for using this program!
```

Observation:

1. This program requires an input less than `100`. We wish to generate a `ValueError` if this is not the case and handle it appropriately.
2. Since Python by itself is unaware of this requirement of ours and has no support for testing it itself, we need to add checks of our own. In line 8, we check if `n` is greater than `100`, and if so we raise `ValueError`.
3. Lines 11-12 handle this and print a suitable message on `ValueError`. Note that `ValueError` can occur in 2 cases in our program – either the input is not a valid integer or the input is a valid integer greater than `100`. In both cases, our response is the same.

13.4 Exception Propagation

In this section, we address the question, "*what happens if an exception that is raised at a particular place is not handled in that place*." The short answer to this question is that the exception *propagates* to the outer level. To understand how it propagates and how it can be handled at an outer level, we will classify the outer levels as shown below and see them in detail in separate sections:

1. Nested `try-except` blocks
2. Functions

13.4.1 Exception Propagation Through Nested Blocks

The `try-except` blocks can be nested – one `try-except` block can be placed inside another `try` block, `except` block, `else` block or `finally` block.

Just for illustration, let us change `ExceptionHandlingDemo2.py` and add nested `try-except` blocks:

ExceptionHandlingDemo9.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Exception propagation through nested try-except blocks
5.
6.   try:
7.       try:
8.           n = int(input("Enter an integer:"))
9.           quotient = int(100/n)
10.          print("100/{} = {}".format(n,quotient))
11.      except ValueError:
12.          print("Please enter an integer!")
13.  except ValueError:
14.          print("Invalid value!")
15.  except ZeroDivisionError:
16.      print("Sorry! Division by zero is not permitted!")
17.
```

Output:

```
Enter an integer:5
100/5 = 20
```

```
Enter an integer:0
Sorry! Division by zero is not permitted!
```

```
Enter an integer:hi
Please enter an integer!
```

Observation:

1. We are expecting 2 different exceptions – `ValueError` and `ZeroDivisionError` - just like `ExceptionHandlingDemo2.py`. But in this program, instead of handling the exceptions using 2 `except` clauses of the same `try` block, we have put a `try-except` block to handle `ValueError` (lines 7-12), and put this entire block inside a `try` block that monitors and handles `ValueError` and `ZeroDivisionError` (lines 6-16).
2. If a `ValueError` is raised in lines 8-10, then it will be handled in lines 11-12. No exceptions are reported in the outer `try` block of line 6, and hence none of it's `except` blocks are executed. Note that for this reason, the `except`

block in lines 13-14 is never executed and was completely unnecessary. However, it's presence is not wrong and we will revisit this example in section 13.4.2 and see some more possibilities.

3. If a `ZeroDivisionError` is raised in lines 8-10, then the inner `try-except` block is unable to handle this exception. The exception then propagates to the outer level and is reported in the outer `try` block (lines 7-12) and is handled by it's `except` block in lines 15-16.
4. If an exception other than these 2 were to occur inside any of the `try` blocks, then that exception would have propagated further as none of these 2 blocks can handle this. Section 13.4 will throw more light on this.

13.4.2 Exception Propagation Through Functions

Another possibility in exception propagation is that if an exception raised in a particular function is not handled within that function, then it propagates to the next outer function that had called this function. This goes on recursively, function through function.

Let us modify `ExceptionHandlingDemo9.py` to demonstrate this:

`ExceptionHandlingDemo10.py`

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Exception propagation through functions
5.
6.   def divideAndPrint():
7.       try:
8.           n = int(input("Enter an integer:"))
9.           quotient = int(100/n)
10.          print("100/{} = {}".format(n,quotient))
11.      except ValueError:
12.          print("Please enter an integer!")
13.
14.  try:
15.      divideAndPrint()
16.  except ValueError:
17.          print("Invalid value!")
18.  except ZeroDivisionError:
19.      print("Sorry! Division by zero is not permitted!")
20.
21.
```

Output:

```
Enter an integer:5
100/5 = 20
```

```
Enter an integer:0
Sorry! Division by zero is not permitted!
```

```
Enter an integer:hi
Please enter an integer!
```

Observation:

1. We have modified `ExceptionHandlingDemo9.py` and put the entire inner `try-except` block into a function called `divideAndPrint`.
2. The `try` block of line 7 monitors it's contents for `ValueError` and `ZeroDivisionError`, but is capable of handling only `ValueError`. Any occurrence of `ZeroDivisionError` will propagate through the function and will be reported to it's caller instead.
3. The function `divideAndPrint` is called from line 15. This is the place where `ZeroDivisionError` is reported, if at all it is raised within `divideAndPrint`. This statement is within a `try` block and can handle `ZeroDivisionError` in lines 18-19.
4. Just like `ExceptionHandlingDemo9.py`, `ValueError` is not handled in lines 16-17 despite the `except` block, but is still not an error.

13.4.3 Unhandled Exceptions

In this section, we address the question, “what happens if an exception that was raised is not handled in the program at all?”.

The answer to this question is that the exception propagates function by function till control comes out of the program and the exception is reported at the terminal.

Let us modify `ExceptionHandlingDemo10.py` to demonstrate this:

`ExceptionHandlingDemo11.py`

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Unhandled Exceptions
```

```
5.
6.   def divideAndPrint():
7.       try:
8.           n = int(input("Enter an integer:"))
9.           quotient = int(100/n)
10.          print("100/{} = {}".format(n,quotient))
11.      except ValueError:
12.          print("Please enter an integer!")
13.
14.  divideAndPrint()
```

Output:

```
Enter an integer:5
100/5 = 20
```

```
Enter an integer:hi
Please enter an integer!
```

```
Enter an integer:0
Traceback (most recent call last):
  File "ExceptionHandlingDemo11.py", line 14, in <module>
    divideAndPrint()
  File "ExceptionHandlingDemo11.py", line 9, in divideAndPrint
    quotient = int(100/n)
ZeroDivisionError: division by zero
```

Observation:

1. The program is based on `ExceptionHandlingDemo10.py` and produces the same output whenever `ValueError` is raised, as it is handled in exactly the same way (in line 11).
2. When `ZeroDivisionError` exception is raised, however, there is no code to handle it. The exception that is raised in the `divideAndPrint` function propagates to the main program (line 14), but since it is not handled there too, it propagates beyond the program and the exception is reported.
3. Professional code never allows such exceptions to propagate beyond the program. Any exception expected must be handled appropriately.

13.5 Re-Raising Exceptions

We have so far seen that when we wish to handle an exception, we can monitor the exception in the `try` block and handle it in the `except` block. This section answers the question, "what if we want to handle the same exception in multiple places in the code?" A practical use of this would be to handle the exception at the level where it occurred as well as inform the caller that the process failed. The syntax for re-raising the exception is shown below:

```
raise
```

Note:

1. Observe that this differs from the normal syntax of `raise`, which requires an exception object. This syntax is understood to mean that we wish to raise the same exception that we are currently handling.
2. This special syntax of `raise` is only available within the `except` block, and ends up raising the same exception object that we are currently handling.

Let us modify `ExceptionHandlingDemo10.py` to incorporate this feature:

ExceptionHandlingDemo12.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # Re-raising Exceptions
5.
6.   def divideAndPrint():
7.       try:
8.           n = int(input("Enter an integer:"))
9.           quotient = int(100/n)
10.          print("100/{} = {}".format(n,quotient))
11.      except ValueError:
12.          print("Please enter an integer!")
13.          raise
14.
15.  try:
16.      divideAndPrint()
17.  except ValueError:
18.          print("Invalid value!")
19.  except ZeroDivisionError:
20.      print("Sorry! Division by zero is not permitted!")
21.
22.
```

Output:

```
Enter an integer:5
100/5 = 20
```

```
Enter an integer:0
Sorry! Division by zero is not permitted!
```

```
Enter an integer:hi
Please enter an integer!
Invalid value!
```

Observation:

1. This program is based on `ExceptionHandlingDemo10.py`, with only line 13 being an additional line, where the exception is being re-raised.
2. Any occurrence of `ZeroDivisionError` is handled in exactly the same way as in `ExceptionHandlingDemo10.py` and hence will not be discussed further here.
3. If a `ValueError` is raised within the try block in lines 8-10, it is handled in the except block in lines 11-13. However, the `raise` statement in line 13 results in the same instance of `ValueError` being raised again, which propagates beyond the function `divideAndPrint` and is reported in the `try` block in line 16. This occurrence of `ValueError` is then handled in the `except` block in lines 17-18.
4. Thus, the same exception instance of `ValueError` is handled both in lines 11-13 and 17-18. Recall that earlier in `ExceptionHandlingDemo10.py`, the `except` block in lines 17-18 would never get executed.

A more practical example is shown syntactically below:

```
def f():
      try:
            ... # try block
      except ExceptionName:
            ... # Handle the exception
            raise # Report to caller that this exception occurred

try:
      f() # Call the function f
      ... # Do other stuff if the call to f() was exception-free
except ExceptionName:
      ... # Respond to the exception
```

13.6 User-Defined Exceptions

We have learnt everything about exceptions – from monitoring them to handling them to propagating them. This section focuses on the final aspect – creating our own exception classes and raising our own exceptions. The ability to create and raise our own exceptions makes it possible to introduce meaningful and different exceptions in our program. Here are some points to be borne in mind when dealing with user-defined exceptions:

1. User-defined exceptions are never automatically raised! This means that such exceptions can only be raised by explicitly using the `raise` statement and passing an object of our exception class.
2. Such user-defined classes have to directly or indirectly derive from the built-in `Exception` class.
3. Since exceptions are rare and an indication of a specific event, raising such exceptions will generally be conditional. Combined with point 1 above, it becomes the programmer's responsibility to determine what condition needs to be used to fire the exception.

Let us revisit `ExceptionHandlingDemo8.py` and add our own exceptions for the following cases:

1. A value of greater than `100` should result in `OverflowException`
2. A negative value should result in `NegativeInputException`
3. Division by `0` will continue to give rise to `ZeroDivisionError` and a non-numeric input will continue to give rise to `ValueError`

ExceptionHandlingDemo13.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # User-defined exceptions
5.
6.   class OverflowException(Exception):
7.       pass
8.
9.   class NegativeInputException(Exception):
10.      pass
11.
12.  def check(n):
13.      if n>100: raise OverflowException()
14.      if n<0: raise NegativeInputException()
15.
16.  try:
17.      n = int(input("Enter an integer:"))
18.      check(n)
19.      quotient = int(100/n)
20.      print("100/{} = {}".format(n,quotient))
21.  except OverflowException:
22.      print("Please enter an integer less than 100!")
23.  except NegativeInputException:
24.      print("Please do not enter a negative integer!")
25.  except ValueError:
26.      print("Please enter an integer!")
27.  except ZeroDivisionError:
28.      print("Sorry! Division by zero is not permitted!")
```

Output:

```
Enter an integer:5
100/5 = 20
```

```
Enter an integer:0
Sorry! Division by zero is not permitted!
```

```
Enter an integer:hi
Please enter an integer!
```

```
Enter an integer:200
Please enter an integer less than 100!
```

```
Enter an integer:-100
Please do not enter a negative integer!
```

Observation:

1. We have defined classes `OverflowException` (line 6) and `NegativeInputException` (line 9), meant to be raised when the input is greater than `100` and when the input is negative respectively.
2. These exception classes of ours derive from the built-in `Exception` class. They can also be derived from any of the classes derived from `Exception`, like for example, `ValueError` class.
3. The classes are empty as we do not find the need to add any functionality in them. We are currently content with the fact that we are able to differentiate between these exceptions. Soon, we will see what functionality can be added to these classes to make them more useful.
4. Line 18 makes a call to our `check` function (defined in line 12) that scrutinizes the given input and raises `OverflowException` or `NegativeInputException` as required.
5. We handle `OverflowException` and `NegativeInputException` in lines 21 and 23 respectively.

The previous example showed how we can define our own exception classes to suit our requirement. Apart from playing the role of identifying the exception that was generated, the classes have no other role to play and were therefore empty. Let us now improvise upon this and add support for also storing the input that was responsible for the exception. This input can then be queried in the `except` block and used as required. For this, we will have to make a couple of changes:

1. At the time of raising the exception object, we will create the exception object passing the input as an argument so that it can be stored within the exception object.
2. To support point 1 above, we need to add a constructor to our classes that receives the input as an argument and stores it in an attribute within the object.
3. We prefer the attribute to be private to the class in order to support data hiding, and will therefore provide a public method that provides access to this private data.

The modified program is given below:

ExceptionHandlingDemo14.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # User-defined exceptions with arguments
5.
6.   class OverflowException(Exception):
7.       def __init__(self,value):
8.           self._value = value
9.
10.      def getValue(self):
11.          return self._value
12.
13.  class NegativeInputException(Exception):
14.      def __init__(self,value):
15.          self._value = value
16.
17.      def getValue(self):
18.          return self._value
19.
20.  def check(n):
21.      if n>100: raise OverflowException(n)
22.      if n<0: raise NegativeInputException(n)
23.
24.  try:
25.      n = int(input("Enter an integer:"))
26.      check(n)
27.      quotient = int(100/n)
28.      print("100/{} = {}".format(n,quotient))
29.  except OverflowException as e:
30.      print("The given input({}) is too
large!".format(e.getValue()))
31.      print("Please enter an integer less than 100!")
32.  except NegativeInputException as e:
33.      print("The given input({}) is
negative!".format(e.getValue()))
34.      print("Please do not enter a negative integer!")
35.  except ValueError:
36.      print("Please enter an integer!")
37.  except ZeroDivisionError:
38.      print("Sorry! Division by zero is not permitted!")
```

Output:

```
Enter an integer:5
100/5 = 20
```

```
Enter an integer:0
Sorry! Division by zero is not permitted!
```

```
Enter an integer:hi
Please enter an integer!
```

```
Enter an integer:200
The given input(200) is too large!
Please enter an integer less than 100!
```

```
Enter an integer:-5
The given input(-5) is negative!
Please do not enter a negative integer!
```

Observation:

1. In lines 21 and 22, when an exception object is being created, the value responsible for the exception is also passed as an argument to the constructor.
2. The constructors in lines 7-8 and 14-15 copy the given value into an attribute `_value`.
3. The `getValue` methods in lines 10-11 and 17-19 return the value stored within the object in the attribute `_value`.
4. This makes it possible for us to access and print the value in lines 30 and 33.

As a final example, let us create exception classes that do not inherit from `Exception` directly, but indirectly derive from it by deriving directly from one of it's derived classes instead. Let us design our `OverflowException` and `NegativeInputException` classes to derive from `ValueError` instead of `Exception`. If there is no other change in the program, then the behaviour and output of the program will remain the same as the previous one. We will however use this program to demonstrate one more concept that has been mentioned earlier – that the

`except` clauses can handle the specified exceptions as well as their derived classes. We will therefore handle `OverflowException`, `NegativeInputException` and `ValueError` all in the same manner in an `except` clause that handles `ValueError`. Furthermore, since we are no longer interested in the value that caused the exception to occur, we will modify `ExceptionHandlingDemo13.py` instead of `ExceptionHandlingDemo14.py`:

ExceptionHandlingDemo15.py

```
1.   #!/usr/bin/python
2.
3.   # Exception Handling Demo
4.   # User-defined exceptions - deriving indirectly from
Exception
5.
6.   class OverflowException(ValueError):
7.       pass
8.
9.   class NegativeInputException(ValueError):
10.      pass
11.
12.  def check(n):
13.      if n>100: raise OverflowException()
14.      if n<0: raise NegativeInputException()
15.
16.  try:
17.      n = int(input("Enter an integer:"))
18.      check(n)
19.      quotient = int(100/n)
20.      print("100/{} = {}".format(n,quotient))
21.  except ValueError:
22.      print("Please enter an integer between 0 and 100!")
23.  except ZeroDivisionError:
24.      print("Sorry! Division by zero is not permitted!")
```

Output:

```
Enter an integer:5
100/5 = 20
```

```
Enter an integer:0
Sorry! Division by zero is not permitted!
```

```
Enter an integer:hi
Please enter an integer between 0 and 100!
```

```
Enter an integer:200
Please enter an integer between 0 and 100!
```

```
Enter an integer:-5
Please enter an integer between 0 and 100!
```

Observation:

1. Our classes `OverflowException` (line 6) and `NegativeInputException` (line 9) now derive from `ValueError` instead of `Exception`.
2. This allows us to handle them both as well as `ValueError` in line 21.
3. If we want to handle these exceptions separately, we can do so using the same approach as the one we used in `ExceptionHandlingDemo13.py`. Care should be taken that the order in which we handle exceptions are from derived class to base class. Thus, `ValueError` should not be handled before `OverflowException`, for instance.

13.7 Questions

1. List the advantages of exception handling.
2. Write the complete syntax for exception handling block in Python and explain the same.
3. Write a short note on raising and re-raising exceptions in Python.
4. Write a short note on exception propagation.
5. Write a short note on user defined exceptions in Python?

13.8 Exercises

1. Write a Python script to demonstrate `ValueError` exception.
2. Write a Python script to demonstrate user defined exceptions.
3. Write a Python script to demonstrate exception propagation.

SUMMARY

- Exceptions can be monitored using the try block.
- Exceptions can be caught and processed using the catch block.
- The catch block can also catch multiple exception types identified by a tuple of exception class names.
- An except block capable of handling a particular exception type is also capable of handling all the sub-classes of that exception type.
- It is also possible to gain a reference to the actual exception instance in the except block.
- An except block that does not specify the exception type it can handle will end up handling all exceptions. Such an except block cannot be followed by any other except blocks.
- The else clause allows unmonitored code to exist along with monitored code!
- The finally clause provides an opportunity to clean-up irrespective of whether an exception occurs or not.
- Exceptions can be raised on demand using the raise statement.
- Exceptions that are not handled at the level where they are raised propagate to outer levels, returning from functions if required, till they are handled somewhere or are reported as unhandled exceptions.

Summary

- Unhandled exceptions result in program termination!
- An exception that is caught by an except statement can be re-raised for additional processing using the raise statement.
- User-defined exceptions can be created by creating custom classes that extend directly or indirectly from the Exception class. Instances of such classes can then be raised as required.

14 File Handling

In this chapter you will be able to:

- ☑ Differentiate between text and binary files.
- ☑ Open files in the desired mode of operation and close them when done processing.
- ☑ Read from and Write to text files.
- ☑ Read from and Write to binary files.
- ☑ Seek within files and determine the current position within the file.

File Handling

14.1 Introduction to File Handling

14.1.1 Need for File Handling

A **file** is a logical collection of data stored as a single unit inside a filesystem on a storage device. While the contents of a file are sequentially accessed, they can be randomly accessed as well. Perhaps one of the most important aspects of files are that they are persistent – available to programs and users even after long periods of time.

Files can be either:-

1. accessed directly by users,
2. or can be opened, viewed and edited using specific applications that understand that file format,
3. or can be used internally by an application (perhaps to store configuration data).

14.1.2 Logical Steps in File Handling

If a program has to access (read from or write to) a file, these are the logical steps that it should follow:

1. **Open the file.** Opening the file creates suitable data structures in memory that allow you to communicate with the file. During this process, it is necessary to identify which file you wish to open and for what reason (in which mode do you wish to open the file). On success (after due validation like file existence and permissions), a buffer is made available in memory through which you may communicate with the file contents. The reason for the buffer to be used us to improve speed by acting as an intermediary between the file and the program.
2. **Access the file.** Once the file has been successfully opened, you may then access the contents of the file. If the file has been opened in a mode that supports reading, you may read from the file. If the file has been opened in a mode that supports writing, you may write to the file.
3. **Close the file.** When you are done dealing with the file, you can close the connection with the file. This will flush out the buffers (thereby writing any data in the buffer that has not yet been written to the file if the file was opened for writing), close the file handle and release the memory occupied by the data structure used to handle the file. When the process terminates, all opened

files are automatically closed. It is a good programming practice to close a file the moment it is no longer needed to be accessed by the program. Once a file is closed, no other file operations can be performed on it. Of course, you can open the file again to access it's contents.

14.1.3 Text Files vs. Binary Files

At this point, even before we enter Python's support for files, it is better that you are aware of the 2 types of files – text and binary – and the differences between them.

A **text file** is a human readable file, comprising of strings using a known, fixed character set. No specific applications are needed to access the file contents (except perhaps a text editor).

A **binary file** on the other hand is a file that uses an encoding that is not human readable. A simple text editor will not be able to help you decipher it's contents. Therefore, specific applications that understand the file format are required to present the file contents to you for viewing or editing.

Examples of file formats that are textual in nature are:

Extension	*Format*
`.txt`	Text file
`.html, .htm`	HTML file
`.py`	Python script
`.xml`	XML file
`.sql`	SQL file

Examples of file formats that are binary in nature are:

Extension	*Format*
`.doc, .docx, .odt`	Document
`.xls, .xlsx, .ods`	Spreadsheet
`.mp3`	MP3 Audio
`.mpg, .mpeg`	MPEG Video
`.jpg, .jpeg`	JPEG Image

Do not get confused by the names ‘text’ and ‘binary’ - both of them are ultimately stored in binary format on a storage device. The distinction is not based on how they are stored, but on whether or not their contents are human readable when the individual characters of the file are displayed.

Apart from this basic difference between text files and binary files, there are technical differences that emerge on how these files are to be accessed and processed by programs. These are listed below:

1. A text file is typically made up of lines (as we humans are comfortable with dealing with a line as a unit). These lines need specific characters within the file to denote their end, and these differ between operating systems. Windows, for example, terminates lines using 2 characters – the carriage return (`\r`) and the linefeed (`\n`). Linux on the other hand, uses just the linefeed character to achieve the same purpose. Python being capable of working on any of these platforms has to support both. Therefore, Python ensures that end-of-lines are handled by it in a platform-specific manner, but for the programmer, Python gives the impression that only the newline character (`\n`) marks the end of a line. Binary files are not organised in the form of lines and any carriage return and linefeed characters found are probably not meant to convey end of line information at all and are thus not to be interpreted in any special manner.
2. Numbers in a text file need to stored in a human readable format. They are therefore converted into digits and each digit is stored as a separate character. Thus, the number `12345` is stored in the file as 5 ASCII characters (assuming ASCII encoding) - ‘`1`’, ‘`2`’, ‘`3`’, ‘`4`’ and ‘`5`’ - and will therefore occupy 5 bytes. The amount of memory required to store a number within a text file therefore depends on the number of digits it has. A binary files stores numbers in it’s direct binary form that is of fixed size independent of the number of digits in the number (provided the number is not unreasonable long).
3. Seeking within text files (covered in detail in section 14.5.4) is restricted due to point 1 above that requires Python to deal with the end of line characters in a platform dependent manner while being platform-independent to the programmer. Seeking within binary files do not have any such restriction.

14.2 Opening and Closing Files

As was pointed out in the previous section, the first operation to be performed in order to access the contents of a file is to open the file. Since there are various ways of reading from files, this section concentrates on how to open and how to close files, with later sections talking about how to read from and how to write to files.

14.2.1 Opening Files

In order to open a file to access it's contents, 2 pieces of information are required:

1. The *pathname* of the file, identifying which file you wish to open
2. The *access mode*, identifying what you wish to do with the file contents

On successfully opening a file, a **file object** is returned that is to be used as a reference to the opened file later on in the program. All other file operations on the file can be performed using this file object.

The `open` function is used to open a file and return a file object, and has the following syntax:

```
open(pathName [,mode])
```

Note:

1. The file is identified in the filesystem by it's pathname, which could be relative or absolute.
2. If the file could not be opened for any reason, `FileNotFoundError` is raised.

The permissible modes are listed in the table below:

Mode	*Meaning*
`'r'` `'rb'`	Open the file for reading. If the file exists and has read permissions, you will be permitted to read it's contents using the file object returned. If the file does not exist or does not have read permissions, this operation will fail.
`'w'` `'wb'`	Open the file for writing. If the file exists and has write permissions, you will be permitted to write to the file, but any data already present in the file is lost as the file is truncated. If the file does not exist, it will be created and you will be permitted to write new contents to the file.
`'a'` `'ab'`	Open the file for appending. If the file exists and has write permissions, you will be permitted to write to the file, and whatever is written is automatically appended to the file without disturbing it's existing contents. If the file does not exist, it will be created and you will be permitted to append to the empty file so created.

Mode	Meaning
'r+' 'r+b'	Open the file for reading and writing. This is the only mode that permits reading as well as writing using the same file object. If the file exists and has write permissions, you will be permitted to read from and write to the file, but any writes will overwrite the data already present at that position within the file. If the file does not exist, it will be created and you will be permitted to write new contents to the file and read the same by seeking (covered in section 14.5).

Table 33: File Open Modes

Note:

1. In the modes listed in the above table, a suffix of 'b' indicates that the file should be treated as a binary file, and the absence of the suffix indicates that the file should be treated as a text file. Thus, the mode 'r' opens a text file for reading whereas the mode 'rb' opens a binary file for reading.
2. The default mode is 'r'. Thus, if a file is opened using `open` without specifying the mode, it will be opened to reading as a text file.

Here is an interactive session showing a successful file open:

```
>>> f=open("fib.py",'r')
>>> f
<_io.TextIOWrapper name='fib.py' mode='r' encoding='UTF-8'>
```

Observation:

1. We are opening the Python script `fib.py` that was previously saved in the current directory. We are opening this file in read mode. Note that Python scripts are text files.
2. On successful opening of the file, a file object is returned and is stored in the variable `f` in our example.

Here is an interactive session to show what happens if the file could not be opened:

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

14.2.2 Closing Files

As discussed in section 14.1.2, when you are done accessing the contents of a file, it is recommended that you close the connection to the file. The `close` function can be used for this purpose with the following syntax:

```
fileObject.close()
```

Note:

1. The `fileObject` used must be one returned by the `open()` function only.
2. Once a file object is closed, it cannot be used to perform any more operations on the file it was connected with using this file object. You may of course call `open()` again to open any file or continue using other file objects that refer to opened files.
3. When the Python script terminates, all opened files are automatically closed.

Here is a sample interactive session to open a file and close it:

```
>>> f=open("fib.py",'r')
>>> f
<_io.TextIOWrapper name='fib.py' mode='r' encoding='UTF-8'>
>>> f.closed
False
>>> f.close()
>>> f
<_io.TextIOWrapper name='fib.py' mode='r' encoding='UTF-8'>
>>> f.closed
True
```

Observation:

1. Even after a file is closed, it's file object keeps track of the file and access mode, but does not permit operations on the file.
2. The `closed` attribute can be used at any time to find out whether the file linked with a file object is currently closed or still open.
3. Closing a closed file is not an offence, though it is obviously not required.

14.3 Reading From Text Files

Once a text file has been opened in a mode that supports reading, we can then proceed to read the file contents using the file object. This section introduces various ways by which we can read from the file:

1. Reading the entire file contents at one go
2. Reading the file contents a line at a time
3. Reading the file contents a chunk at a time
4. Reading the file contents a character at a time

14.3.1 Reading Entire File at Once

One of the simplest ways of reading a text file is to read the entire file at once in a single step. This section introduces 2 techniques by which we could achieve this:

1. Loading the entire file contents as a string
2. Loading the entire file contents as a list of lines

14.3.1.1 Reading entire file as a single string

To load the entire file contents as a single string, the `read()` method can be used that has the following syntax:

```
fileObject.read()
```

This method will read the entire file contents and will return it as a single string.

Let us write a Python script to display the contents of a file whose filename is entered by the user (the UNIX `cat` command does something similar):

cat1.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cat command to display the contents of
a file
4.
5.   filename = input("Enter filename:")
6.
7.   try:
8.       file = open(filename)
9.       contents = file.read()
10.      print(contents)
```

```
11.  except Exception as e:
12.      print("Unable to open file: {}".format(filename))
13.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:cat1.py
#!/usr/bin/python

# Implementation of cat command to display the contents of a
file

filename = input("Enter filename:")

try:
    file = open(filename)
    contents = file.read()
    print(contents)
except Exception as e:
    print("Unable to open file: {}".format(filename))
    print("Reason: {}".format(str(e)))
```

Observation:

1. We accept the filename from the user in line 5 and store it in the variable `filename`.
2. We then open the file in line 8 and store the file object on success in the variable `file`.
3. If the file could not be opened successfully, the exception handling logic in lines 12-13 print a suitable message.
4. Once the file has been opened, we use the file object obtained to read it's contents and store it in the variable `contents` in line 9.
5. The contents of the variable `contents` is then printed in line 10.
6. Note that the entire file contents will be present in the variable `contents` and the individual lines will be separated by the newline character.
7. If the file contains a newline character at the very end, our variable `contents` will also have a terminating newline character.
8. During execution, we have given our Python script filename and hence our program ends up printing itself! You may of course give any other filename, but do remember that if the file is not in the current directory, you'll need to provide the pathname to the file.

We will now see how the program behaves when it is unable to open the file specified:

Output:

```
Enter filename:blah
Unable to open file: blah
Reason: [Errno 2] No such file or directory: 'blah'
```

Observation:

1. In the above execution, we have given the name of a non-existent file and hence the program is unable to open the same for reading.
2. There are other reasons for being unable to open a file, including insufficient permissions, insufficient memory and too many files open already

14.3.1.2 Reading entire file as a list of strings

In the second approach, we can convert the lines of the file to items in a list of strings and then access the contents through the list as shown in the program below:

cat2.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cat command to display the contents of
a file
4.
5.   filename = input("Enter filename:")
6.
7.   try:
8.       file = open(filename)
9.       contents = list(file.read())
10.      for line in contents: print(line,end='')
11.  except Exception as e:
12.      print("Unable to open file: {}".format(filename))
13.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:cat2.py
#!/usr/bin/python

# Implementation of cat command to display the contents of a
file

filename = input("Enter filename:")

try:
    file = open(filename)
    contents = list(file.read())
    for line in contents: print(line,end='')
except Exception as e:
    print("Unable to open file: {}".format(filename))
    print("Reason: {}".format(str(e)))
```

```
Enter filename:blah
Unable to open file: blah
Reason: [Errno 2] No such file or directory: 'blah'
```

Observation:

1. This program is very similar to the previous one with changes only in lines 9-10.
2. A file is a sequence of lines and hence can be passed to the constructor of `list` to convert each item of the sequence (line in the file) to an item in the list. This is done in line 9.
3. Line 10 then iterates through the list and prints each item of the list. Note that each line is terminated by the newline character and hence we need to ask print not to print a newline character of it's own. The last line of the file may or may not have a terminating newline character, and will be reflected accordingly in the last item of the list.

This same functionality is obtained by using the `readlines()` method:

```
fileObject.readlines([sizeHint])
```

Note:

1. The `readlines()` method reads the entire file and returns a list of strings with each item being a line of the file.
2. Each string item of the list will have a terminating newline character, except perhaps the last line which would depend on whether or not there was a terminating newline character at the end of the file.
3. The optional `sizeHint` parameter is the suggested maximum number of bytes you are willing to read from the file, but the actual result is implementation dependent and should not be replied upon. When omitted (which is what we would be doing almost always), it means that we are interested in reading the entire file contents.

14.3.2 Reading Files a Line at a Time

The previous section showed how we can access the file contents by completely loading at one go. We saw 2 approaches that give us identical results:

1. Loading the entire content as a single string
2. Loading the entire content as a list of lines (strings)

This section focuses on how to read the file contents a line at a time, which might appear similar to approach 2 above, but differs in that we are not loading the entire file. Loading an entire file into memory might be inefficient when the file is big and could even make your program unstable as it runs out of memory. When it is not known for sure that the file is expected to be reasonably small, it is far better to load it a line at a time instead.

14.3.2.1 Reading a Line at a Time by Iterating Through the File

A file is a sequence of lines that can be directly iterated upon, as demonstrated in the program below:

cat3.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cat command to display the contents of
a file
4.
5.   filename = input("Enter filename:")
6.
```

```
7.   try:
8.       file = open(filename)
9.       for line in file: print(line,end='')
10.  except Exception as e:
11.      print("Unable to open file: {}".format(filename))
12.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:cat3.py
#!/usr/bin/python

# Implementation of cat command to display the contents of a
file

filename = input("Enter filename:")

try:
    file = open(filename)
    for line in file: print(line,end='')
except Exception as e:
    print("Unable to open file: {}".format(filename))
    print("Reason: {}".format(str(e)))
```

```
Enter filename:blah
Unable to open file: blah
Reason: [Errno 2] No such file or directory: 'blah'
```

Observation:

1. This program is similar to the previous program (`cat2.py`) with the change highlighted in line 9.
2. While the previous program read the entire file into a list and then iterated through the list, this program iterates through the file directly, giving scope for better memory handling.
3. The loop in line 9 could have been replaced by a statement like `for line in list(file)` or `for line in file.readlines()`, but these would load the entire file in memory resulting in poorer memory management.

14.3.2.2 Reading a Line at a Time From a File Using readline()

Just as how the `readlines()` method reads all lines from a file and returns it in the form of a list, the `readline()` method reads a single line from a file and returns it as a string.

Syntax:

```
fileObject.readline([size])
```

Note:

1. This method reads and returns a single line from the file identified by the file object `fileObject`.
2. The optional `size` parameter can be used to specify the maximum number of characters to read from the file. When omitted, it means read till the end of line (till the newline character is read) or till the end of file.
3. If the read line terminated with a newline character (which would be the case for all lines except possibly the last line in the file), the terminating newline character is retained as the last character of the string that is returned.
4. If already at the end of file, a null string is returned.

Here is an implementation of `cat` using `readline()`:

cat4.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cat command to display the contents of
a file
4.
5.   filename = input("Enter filename:")
6.
7.   try:
8.       file = open(filename)
9.       while 1:
10.          line = file.readline()
11.          if not line: break
12.          print(line,end='')
13.  except Exception as e:
14.      print("Unable to open file: {}".format(filename))
15.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:cat4.py
#!/usr/bin/python

# Implementation of cat command to display the contents of a
file

filename = input("Enter filename:")

try:
    file = open(filename)
    while 1:
        line = file.readline()
        if not line: break
        print(line,end='')
except Exception as e:
    print("Unable to open file: {}".format(filename))
    print("Reason: {}".format(str(e)))
```

Observation:

1. We read a line at a time in line 10 using `readline()` and store the line as a string in the variable `line`.
2. On encountering end of file, the string returned by `readline()` will be empty, which is tested in line 11.
3. Lines 9-12 constitute an infinite loop that terminates only when line 11 detects the end of file.

14.3.3 Reading Arbitrary Amounts of Data From Files

While the previous section showed how to read a line at a time, which is probably the most common way to read a text file, this section focuses on how to read any specific number of characters (bytes) from a file, giving you more control on how much data you wish to read from the file.

We revisit the `read` method covered in section 14.3.1.1 for this purpose, with it's complete syntax:

```
fileObject.read(bytes)
```

Note:

1. This method reads upto `bytes` number of bytes from the file identified by `fileObject` and returns the result as a string.
2. On encountering end of file, this method returns the read data immediately and could possibly return lesser than `bytes` number of bytes.
3. If already at the end of file at the time of making the call, this method returns a null string.
4. This method does not stop reading on encountering the newline character. Any newline character found in the file is merely considered to be a character.
5. If `bytes` is not provided or is given as `-1`, this will end up reading all the characters of the file, as covered in section 14.3.1.1.

Here is a re-implementation of the previous program using `read()` to read `100` bytes at a time:

cat5.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cat command to display the contents of
a file
4.
5.   filename = input("Enter filename:")
6.
7.   try:
8.       file = open(filename)
9.       while 1:
10.          data = file.read(100)
11.          if not data: break
12.          print(data,end='')
13.  except Exception as e:
14.      print("Unable to open file: {}".format(filename))
15.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:cat5.py
#!/usr/bin/python

# Implementation of cat command to display the contents of a
file

filename = input("Enter filename:")

try:
    file = open(filename)
    while 1:
        data = file.read(100)
        if not data: break
        print(data,end='')
except Exception as e:
    print("Unable to open file: {}".format(filename))
    print("Reason: {}".format(str(e)))
```

Observation:

1. This program is very similar to `cat4.py` with changes in lines 10-12.
2. This program reads `100` bytes at a time and prints it.
3. As an example, if the file contains `250` bytes, then line 10 of this program ends up reading `100`, `100`, `50` and `0` bytes over 4 iterations and the loop is terminated by line 11.

14.3.4 Reading Files a Character at a Time

As a specialization of the concept presented in the previous section, we can read the contents of a file a single byte (character) at a time using `read(1)`. While this is not exactly efficient, it can provide a very high level of granularity as far as reading from text files is concerned. While this can be more appropriate for use with binary files, nothing prevents us from using it on text files too.

`cat6.py`

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cat command to display the contents of
a file
4.
5.   filename = input("Enter filename:")
```

```
6.
7.   try:
8.       file = open(filename)
9.       while 1:
10.          data = file.read(1)
11.          if not data: break
12.          print(data,end='')
13.  except Exception as e:
14.      print("Unable to open file: {}".format(filename))
15.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:cat6.py
#!/usr/bin/python

# Implementation of cat command to display the contents of a
file

filename = input("Enter filename:")

try:
    file = open(filename)
    while 1:
        data = file.read(1)
        if not data: break
        print(data,end='')
except Exception as e:
    print("Unable to open file: {}".format(filename))
    print("Reason: {}".format(str(e)))
```

Observation:

1. This program is very similar to `cat5.py` with the only change being that this program reads `1` byte at a time in line 10 instead of `100` bytes.
2. This program will run slower than `cat5.py` simply due to the number of iterations of the loop and the statements within it. The speed difference cannot be easily (visibly) detected on small files.

14.4 Writing to Text Files

Unlike the many different ways of reading from files, writing to files is pretty straightforward with the use of a single function – `write()` - whose syntax is shown below:

```
fileObject.write(string)
```

Note:

1. This method writes the contents of the given string (`string`) to the file specified by the invoking file object (`fileObject`).
2. This method does not add a terminating newline character of its own. If the string terminates with a newline character, it is also written, else no newline is automatically added to the file.
3. The file identified by the file object (`fileObject`) must have been opened in a mode that supports writing, else the operation fails.

Here is a program that generates the factorial of all integers from 1 to 10 and stores them in a file called `factorials.txt`:

factorials.py

```
1.   #!/usr/bin/python
2.
3.   # Storing factorials in a text file
4.
5.   from math import factorial
6.
7.   try:
8.       file = open("factorials.txt","w")
9.       for i in range(1,11):
10.          file.write("The factorial of {} is
{}\n".format(i,factorial(i)))
11.  except Exception as e:
12.      print("Unable to open file: {}".format(filename))
13.      print("Reason: {}".format(str(e)))
14.  else:
15.      print("Factorials successfully written to file
factorials.txt")
```

Output:

```
Factorials successfully written to file factorials.txt
```

Contents of file `factorials.txt`:

```
The factorial of 1 is 1
The factorial of 2 is 2
The factorial of 3 is 6
The factorial of 4 is 24
The factorial of 5 is 120
The factorial of 6 is 720
The factorial of 7 is 5040
The factorial of 8 is 40320
The factorial of 9 is 362880
The factorial of 10 is 3628800
```

Observation:

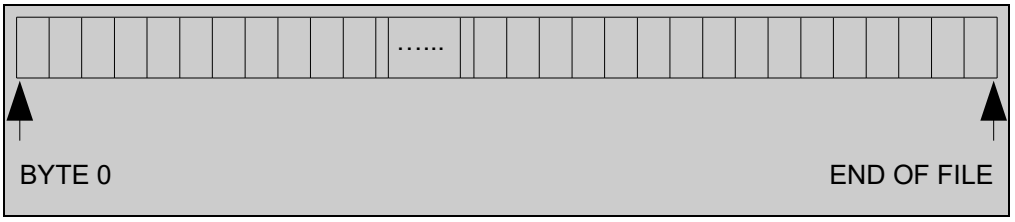
1. Line 8 opens the file `factorials.txt` in write mode (`"w"`). Instead of hard-coding the filename, we could also accept the filename from the user.
2. Line 10 uses the `write()` method to write a formatted string into the file. Note that we have taken care to terminate the string with a newline so that the file contents will be well formatted and readable.
3. Line 15 is executed only if there were no exceptions and the strings were written to the file successfully.

14.5 Seeking Within Files

While we normally prefer to read from or write to files sequentially (as we have been doing in this chapter so far), it is sometimes required to be able to “jump” to specific locations within the file, thereby providing random access on the file contents. This section shows how such jumping is possible in Python.

14.5.1 Introduction to File Offsets

If we view the file contents as a linear sequence of bytes/characters, we can visualise the contents as show in the diagram below:



When a file is opened for reading or writing, the *current position* in that file will be byte 0 (the beginning of the file). This means that next read or write will start from position 0 and will increment the position by the number of bytes successfully read or written. (An exception to this is writes to files that have been opened in append modes, in which case data will always be written at the end of the file instead of at the current position). As we continue reading or writing, the position keeps incrementing, ensuring sequential access of the file contents.

While writing to the file, it is possible to go past the end of the file, in which case the file grows with the writes. While reading from a file, it is not possible to go past the end of the file.

Any particular position in the file can be selected by means of an offset applied from one of the standard positions:

1. From the beginning of the file (identified by the standard position `SEEK_SET`)
2. From the current position within the file (identified by the standard position `SEEK_CUR`)
3. From the end of the file (identified by the standard position `SEEK_END`)

A positive offset implies those many bytes after the selected standard position whereas a negative offset implies those many bytes before the selected standard position. An offset value of 0 implies exactly at the selected standard position.

The table below shows some examples of how these offsets combined with standard positions can help locate a position in a file:

Offset	*Standard Position*	*Meaning*
`0`	`SEEK_SET`	Beginning of the file
`10`	`SEEK_SET`	`10` bytes after the beginning of the file (positioned at the beginning of the 11th byte of the file)
`0`	`SEEK_CUR`	Current position within the file (no change in position)
`10`	`SEEK_CUR`	`10` bytes after the current position in the file (useful for the skipping over the next 10 bytes in the file)

Offset	Standard Position	Meaning
-10	SEEK_CUR	10 bytes before the current position in the file (useful for re-reading or overwriting the previous section of the file)
0	SEEK_END	End of the file
-10	SEEK_END	10 bytes before the end of the file
10	SEEK_END	10 bytes after the end of the file (valid only if the file has been opened for writing, in which case the file will grow along with the change in position)

Table 34: Examples of Seeking

14.5.2 Seeking to a Particular Position Within a File

In order to seek to a specific position within a file (using the concepts of the previous section), the `seek()` method can be used, which has the following syntax:

```
fileObject.seek(offset [,whence])
```

Note:

1. The `offset` parameter specifies an offset relative to a standard position that is selected in the second parameter, `whence`.
2. The `whence` parameter is the standard position selection relative to which the offset is applied. The values for this parameter has to be one of `os.SEEK_SET` (`0`), `os.SEEK_CUR`(`1`) or `os.SEEK_END`(`2`). If this parameter is not provided, the default standard position assumed in `os.SEEK_SET`.
3. For text files, the offset must be `0` or a value that was previous returned by the `tell()` method (covered in section 14.5.3). For binary files, the offset can be any arbitrary number.

14.5.3 Finding the Current Location Within a File

Just as how the `seek()` method allows us to change the position within an opened file, the `tell()` method tells us the current location in an opened file and has the following syntax:

```
fileObject.tell()
```

Note:

1. This method returns the current position within the file represented by the file object (`fileObject`), measured as the number of bytes from the beginning of the file.
2. The value returned by the `tell()` method can be used directly as an offset in the `seek()` method with the standard position as `SEEK_SET` to return to this position whenever required irrespective of where the current position is later on during the execution of the script.
3. For text files, the value returned by the `tell()` method and the special offset of `0` are the only valid values that can be used as offset in the `seek()` method.

14.5.4 Special Notes for Seeking Within Text Files

Section 14.1.3 introduced the difference between text and binary files. One of the differences between them that affects seeking within files is the fact that end of line detection is done internally by Python depending on the platform. The implementation can vary and hence seeking will also be affected. To keep things simple, Python asks programmers to stick to the following rules as far as seeking in text files are concerned:

1. It is safe to use an offset of `0` relative to any standard position. Such a seek is guaranteed to work.
2. Any other offset used must be relative to the start of the file (`SEEK_SET`).
3. If a non-zero offset is used for seeking, the offset should be one that was previously returned from `tell()`. No other arbitrary offsets are to be used.

Binary files do not undergo any end-of-line processing by Python and hence do not have the above restrictions.

14.6 Writing to Binary Files

While text files are designed to be human-readable, it is sometimes preferable to store data in an efficient way that our application can read and understand but not humans. Such files would be called binary files just to differentiate from text files that contain human-readable text. In order to write to binary files and read from binary files, the `pickle` module can be used.

The `pickle` module is capable of converting an object to a stream of bytes and then writing them to a binary file and is also capable of reading a sequence of bytes from a binary file and converting them into an object.

In order to write an object to a binary file, the `dump()` function can be used, which has the following syntax:

```
pickle.dump(object,file)
```

Note:

1. In the above syntax, `pickle` is the name of the module and `dump` is the name of a function in that module. There are other ways of pickling an object, but we will stick to this syntax.
2. The parameter `object` is any object you wish to store in the file and the parameter `file` is the file object identifying the file to which the object needs to be written.
3. The file is expected to be opened in a mode that supports writing in binary.

Here is a program that uses a class called `Employee` to represent an employee, and allows the user to add one such employee record to a binary file called `employees.dat`:

add_employee.py

```
1.  #!/usr/bin/python
2.
3.  # Program to write employee records to a file
4.
5.  import pickle
6.
7.  class Employee:
8.      def __init__(self,name,id,designation):
9.          self.name = name
10.         self.id = id
11.         self.designation = designation
12.
13. try:
14.     file = open("employees.dat","ab")
15.     name = input("Enter employee name:")
16.     id = int(input("Enter employee ID:"))
17.     designation = input("Enter employee designation:")
18.
19.     pickle.dump(Employee(name,id,designation),file)
20. except Exception as e:
21.     print("Unable to open file!")
22.     print("Reason: {}".format(str(e)))
23. else:
24.     print("Employee record successfully added to file!")
```

Output:

```
Enter employee name:Ram
Enter employee ID:1
Enter employee designation:Manager
Employee record successfully added to file!
```

Observation:

1. We import the `pickle` module in line 5. We define our `Employee` class in line 7.
2. The `Employee` class contains a constructor (line 8) that accepts the name, ID and designation of the employee and stores these details within the object.
3. We open the file `employees.dat` in append binary (`ab`) mode in line 14. Since we are planning to add records to the file without overwriting any existing records, the append mode is necessary. Since we plan to write objects, the file should be binary in nature. Note that this mode creates the file if it is not already present.
4. We accept the employee details from the user in lines 15-17.
5. We construct an `Employee` object in line 19 using the details the user had given and write them to the file `file` using `pickle.dump` function.
6. Any errors are handled in lines 21-22. On success, a message is printed by line 24.

14.7 Reading from Binary Files

Once objects have been stored in files using `pickle.dump()`, the `pickle.load()` function can be used to load them back from file.

```
pickle.load(file)
```

Note:

1. The parameter `file` identifies the file from which an object has to be loaded from the current position in the file. The file is expected to be binary and should have been opened in a mode that supports reading.
2. The object that is successfully loaded from the file is returned by this function.

We will now write a program that lists all employee records in the file `employees.dat` to which records were written by the previous program, `add_employee.py`.

list_employees.py

```
1.   #!/usr/bin/python
2.
3.   # Program to list employee records from a file
4.
5.   import pickle
6.
7.   class Employee:
8.       def __init__(self,name,id,designation):
9.           self.name = name
10.          self.id = id
11.          self.designation = designation
12.
13.  try:
14.      file = open("employees.dat","rb")
15.
16.      while 1:
17.          employee = pickle.load(file)
18.          print("Name: {} ID: {} Designation:
{}".format(employee.name,employee.id,employee.designation))
19.
20.  except EOFError: pass
21.  except Exception as e:
22.      print("Unable to open file!")
23.      print("Reason: {}".format(str(e)))
```

Output:

```
Name: Ram ID: 1 Designation: Manager
```

Observation:

1. We define the same `Employee` class as was defined in `add_employee.py` in line 7. Of course, practically a better option would be to save the `Employee` class in a module and load it in both programs using concepts taken from section 15!
2. We open the file `employees.dat` for reading in binary mode in line 14.
3. We have a loop set up in line 16 to iterate as many times as required in the file, reading records one at a time. Each employee record is loaded as an `Employee` object in line 17 using `pickle.load()`.
4. On end of file, an `EOFError` is generated by `pickle.load()`, which is silently handled in line 20. This results in the termination of the loop.

14.8 Programs based on Files

14.8.1 Implementation of head: Extract First Few Lines of a File

The UNIX `head` command displays the first 10 lines of a file by default. Let us write a Python script that does the same.

head.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of head command to display the first 10
lines of a file
4.
5.   filename = input("Enter filename:")
6.
7.   try:
8.       file = open(filename)
9.       for i in range(10):
10.          data = file.readline()
11.          if not data: break
12.          print(data,end='')
13.  except Exception as e:
14.      print("Unable to open file: {}".format(filename))
15.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:head.py
#!/usr/bin/python

# Implementation of head command to display the first 10 lines
of a file

filename = input("Enter filename:")

try:
    file = open(filename)
    for i in range(10):
        data = file.readline()
```

Observation:

1. We ask the user for the filename in line 5 and open the file in line 8. Errors are handled in lines 14-15.
2. We run a loop that executes 10 times in line 9. In each iteration, we read 1 line from the file (line 10), check whether we have reached the end of file (line 11) and print the line read if successful (line 12).
3. If the file contains less than 10 lines, the loop terminates prematurely in line 11 and we end up displaying as many lines as were present.

14.8.2 Implementation of tail: Extract Last Few Lines of a File

The UNIX `tail` command displays the last 10 lines of a given file by default. Let us see how we can write a Python script to do the same.

tail.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of tail command to display the last 10
lines of a file
4.
5.   filename = input("Enter filename:")
6.
7.   try:
8.       file = open(filename)
9.       lines = file.readlines()
10.      if len(lines)>10: lines = lines[-10:]
11.      print("".join(lines),end='')
12.  except Exception as e:
13.      print("Unable to open file: {}".format(filename))
14.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:tail.py
filename = input("Enter filename:")

try:
    file = open(filename)
    lines = file.readlines()
    if len(lines)>10: lines = lines[-10:]
    print("".join(lines),end='')
except Exception as e:
    print("Unable to open file: {}".format(filename))
    print("Reason: {}".format(str(e)))
```

Observation:

1. We accept the filename from the user in line 5 and open the file in line 8. Errors are handled in lines 13-14.
2. We load the entire file into a list in line 9. We need the last 10 entries of this list.
3. If there are less than 10 lines in the file, they should be displayed as it is. This check is made in line 10 and if there are more than 10 lines, we pick the last 10 lines of the list.
4. The list contents are converted into a single string and printed in line 11.
5. This approach is simple but inefficient as the entire file is loaded into memory. Especially when dealing with big files, it is better to develop an alternate logic that loads a single line at a time and maintains at most 10 lines in memory.

14.8.3 Combination of head and tail Utilities

In UNIX, the `head` and `tail` commands can be piped together to obtain any consecutive sequence of lines from a file. We can implement the same as a Python script.

headtail.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of head and tail command combination to
display consecutive lines of a file
4.
5.   filename = input("Enter filename:")
6.   start = int(input("Enter starting line:"))
7.   length = int(input("Enter number of lines:"))
8.   try:
9.       file = open(filename)
10.      lines = file.readlines()
11.      lines = lines[start-1:start+length-1]
12.      print("".join(lines),end='')
13.  except Exception as e:
14.      print("Unable to open file: {}".format(filename))
15.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:headtail.py
Enter starting line:6
Enter number of lines:4
start = int(input("Enter starting line:"))
length = int(input("Enter number of lines:"))
try:
    file = open(filename)
```

Observation:

1. We accept the filename (line 5), starting line number (line 6) and number of lines to be displayed (line 7) from the user and open the file in line 9 and read all its contents into a list in line 10.
2. Since the user numbers from 1 while the list index starts from 0, the first line required from the list is at index `start-1`. We extract the required lines from the list in line 11 and display them after converting it to a string in line 12.
3. This program is simple but inefficient as the entire file contents are loaded to memory. A more memory efficient alternative would be to load the file contents line by line and conditionally display it by keeping track of the current line number.

14.8.4 Implementation of wc: Word Count

The UNIX `wc` utility counts the number of characters, words and lines in a given file. Here is a Python script that does the same.

wc.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of wc command to display
4.   # the number of characters, words and lines in a file.
5.
6.   filename = input("Enter filename:")
7.   chars,words,lines = 0,0,0
8.
9.   try:
10.      file = open(filename)
11.      while 1:
12.          line = file.readline()
13.          if not line: break
14.          lines += 1
```

```
15.          chars += len(line)
16.          words += len(line.split())
17.  except Exception as e:
18.      print("Unable to open file: {}".format(filename))
19.      print("Reason: {}".format(str(e)))
20.  else:
21.      print("Number of characters :",chars)
22.      print("Number of words :",words)
23.      print("Number of lines :",lines)
```

Output:

```
Enter filename:wc.py
Number of characters : 613
Number of words : 73
Number of lines : 23
```

Observation:

1. We obtain the filename from the user in line 6 and open the file in line 10. Errors are handled in lines 18-19.
2. We read the file contents a line at a time (line 12). Each time we read a line, we increment the number of lines read (`lines`) in line 14.
3. We add the number of characters found in that line to the variable `chars` in line 15.
4. We use the `split()` function (section 8.3.2) to split the line into a list of words and count the number of words using `len()` and add that to `words` in line 16.
5. The results are displayed in lines 21-23 if there are no errors.

14.8.5 Implementation of cp: Copy Files

The UNIX `cp` command copies one file to another. Here is a Python script to do something similar.

cp.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cp command to copy files
4.
```

```
5.   source_filename = input("Enter source filename:")
6.   dest_filename = input("Enter destination filename:")
7.
8.   try:
9.       source_file = open(source_filename,"rb")
10.      dest_file = open(dest_filename,"wb")
11.      while 1:
12.          byte = source_file.read(1)
13.          if not byte: break
14.          dest_file.write(byte)
15.  except Exception as e:
16.      print("Unable to open file!")
17.      print("Reason: {}".format(str(e)))
18.  else:
19.      print("File successfully copied!")
```

Output:

```
Enter source filename:cp.py
Enter destination filename:copy.py
File successfully copied!
```

Observation:

1. We obtain the source filename and destination filename from the user in lines 5-6.
2. We open the source file for reading in binary mode in line 9. The binary mode ensures that our program works for all files – whether textual or not. Similarly, we open the destination file for writing in binary mode in line 10. All errors are handled in lines 16-17.
3. We read 1 byte at a time from the source file (line 12) and write the byte read into the destination file (line 14). Line 13 ensures that this loop is broken on encountering the end of file on the source file.

14.8.6 Implementation of cmp: Compare Files

The UNIX `cmp` command compares two files and reports whether they are identical or not. If they are identical, no output is generated, else the position of the first difference is reported. Here is a Python script to do the same.

cmp.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cmp command to compare files
4.
5.   filename1 = input("Enter first filename:")
6.   filename2 = input("Enter second filename:")
7.   byte,line = 1,1
8.
9.   try:
10.      file1 = open(filename1)
11.      file2 = open(filename2)
12.      while 1:
13.          char1 = file1.read(1)
14.          char2 = file2.read(1)
15.          if not char1 and not char2: break
16.          elif not char1 or not char2 or char1 != char2:
17.              print("{} {} differ: byte {}, line
{}".format(filename1,filename2,byte,line))
18.              break
19.          byte += 1
20.          if char1 == '\n': line += 1
21.  except Exception as e:
22.      print("Unable to open file")
23.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter first filename:cmp.py
Enter second filename:cmp.py
```

```
Enter first filename:cmp.py
Enter second filename:cp.py
cmp.py cp.py differ: byte 41, line 3
```

Observation:

1. We accept the 2 filenames from the user in lines 5-6. Error handling is performed by lines 22-23.
2. We open both the files for reading in lines 10-11. We have assumed the files to be textual. If binary files are to be supported, we can change the mode to "rb".

3. We read 1 character from each of the files in lines 13-14. To speed up execution, we can also read a line at a time (for text files only) or multiple bytes at a time.
4. If the files are identical, all pairs of characters will be identical and we will encounter end of file on both files simultaneously. This is checked in line 15.
5. If the files are not identical, either some pair of characters will differ or one of the files will end with data remaining in the other file. These checks are made in line 16.
6. If a difference is encountered, we display a suitable message along with the location of the difference (using variables `byte` and `line` to keep track of the byte number and line number respectively) in line 17. The `break` statement in line 18 ensures that only the first difference is reported.
7. On successfully reading each byte, the variable `byte` is incremented in line 19. On reading a newline character, the variable `line` is incremented in line 20.

14.8.7 Implementation of cut: Extract Vertical Slices of Files

The UNIX `cut` command displays vertical slices of data from the given file. While the actual command is very powerful and supports various options, we will stick to extracting a simple range of bytes/characters from each line of a given file.

cut.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of cut command to extract vertical slices
from files
4.
5.   filename = input("Enter filename:")
6.   start = int(input("Enter starting column:"))
7.   end = int(input("Enter ending column:"))
8.
9.   try:
10.      file = open(filename)
11.      while 1:
12.          line = file.readline()
13.          if not line: break
14.          print(line[start-1:end])
15.  except Exception as e:
16.      print("Unable to open file: {}".format(filename))
17.      print("Reason: {}".format(str(e)))
```

Output:

```
Enter filename:cut.py
Enter starting column:4
Enter ending column:8
usr/b

mplem

ename
rt =
 = in

:

 file
 whil



ept E
 prin
 prin
```

Observation:

1. We accept the filename, starting byte and ending byte from the user in lines 5-7.
2. We open the file for reading in line 10. Error handling is performed by lines 16-17.
3. We read line by line from the file till the end of file using lines 11-13.
4. We display only a selected portion of the line (line 14). We use `start-1` as the index starts from 0 whereas the user uses a 1-based numbering.
5. This program works with only a simple byte range. The actual `cut` command also supports extraction based on fields and also supports specification of the field delimiter. The same functionality can be implemented by making use of the `split()` function to split the read line into pieces and then displaying the required field range.

14.8.8 Employee Record Management Using Files

For the final sample program on file handling, let us build on the programs `add_employee.py` (section 14.6) and `list_employees.py` (section 14.7) and create a complete menu-based employee management program that allows us to perform the following using files:

- Add employee records
- List all employee records
- Search for a specific employee
- Delete selected employee record

employee_management.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of an employee management system
4.
5.   import os
6.   import pickle
7.
8.   class Employee:
9.       def __init__(self,name,id,designation):
10.          self.name = name
11.          self.id = id
12.          self.designation = designation
13.
14.  def menu():
15.      print("Employee Management System")
16.      print("==========================")
17.      print("1. Add Employee")
18.      print("2. List Employees")
19.      print("3. Search Employee")
20.      print("4. Delete Employee")
21.      print("5. Quit")
22.      return int(input("Enter choice:"))
23.
24.  def do_add_employee(file):
25.      name = input("Enter employee name:")
26.      id = int(input("Enter employee ID:"))
27.      designation = input("Enter employee designation:")
28.
29.      file.seek(0,os.SEEK_END)
30.      pickle.dump(Employee(name,id,designation),file)
31.
```

```
32.
33.  def do_list_employees(file):
34.      try:
35.          file.seek(0)
36.          while 1:
37.              employee = pickle.load(file)
38.              print("Name: {} ID: {} Designation:
{}".format(employee.name,employee.id,employee.designation))
39.      except EOFError: pass
40.
41.  def do_search_employee(file):
42.      id = int(input("Enter ID to search:"))
43.      try:
44.          file.seek(0)
45.          while 1:
46.              employee = pickle.load(file)
47.              if id == employee.id:
48.                  print("Name: {} ID: {} Designation:
{}".format(employee.name,employee.id,employee.designation))
49.                  break
50.      except EOFError: print("Record not found!")
51.
52.
53.  def do_delete_employee(file):
54.      id = int(input("Enter ID to delete:"))
55.      file.seek(0)
56.      tempfile = open("temp.dat","w+b")
57.
58.      try:
59.          while 1:
60.              employee = pickle.load(file)
61.              if not employee.id == id:
62.                  pickle.dump(employee,tempfile)
63.      except EOFError: pass
64.
65.      file.seek(0)
66.      tempfile.seek(0)
67.      try:
68.          while 1:
69.              employee = pickle.load(tempfile)
70.              pickle.dump(employee,file)
71.      except EOFError:
72.          file.flush()
73.          file.truncate()
74.          tempfile.close()
75.          os.remove("temp.dat")
76.
```

```
77.  filename="employees.dat"
78.  try:
79.      if os.path.isfile(filename): mode="r+b"
80.      else: mode="w+b"
81.      file = open(filename,mode)
82.      while 1:
83.          choice = menu()
84.          if choice == 1: do_add_employee(file)
85.          elif choice == 2: do_list_employees(file)
86.          elif choice == 3: do_search_employee(file)
87.          elif choice == 4: do_delete_employee(file)
88.          elif choice == 5: break
89.          else: print("Invalid choice!")
90.  except Exception as e:
91.      print("Unable to open file: {}".format(filename))
92.      print("Reason: {}".format(str(e)))
```

Output:

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:1
Enter employee name:Sham
Enter employee ID:2
Enter employee designation:Manager
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
```

```
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Sham ID: 2 Designation: Manager
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:1
Enter employee name:Anthony
Enter employee ID:5
Enter employee designation:Team Lead
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Sham ID: 2 Designation: Manager
Name: Anthony ID: 5 Designation: Team Lead
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:3
Enter ID to search:2
Name: Sham ID: 2 Designation: Manager
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:3
Enter ID to search:3
Record not found!
Employee Management System
==========================
```

```
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:4
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Sham ID: 2 Designation: Manager
Name: Anthony ID: 5 Designation: Team Lead
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:2
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Anthony ID: 5 Designation: Team Lead
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:5
Employee Management System
```

```
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:1
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:5
```

Notes on Output:

1. This program uses the same data file that was used by the programs `add_employee.py` (section 14.6) and `list_employees.py` (section 14.7). This means that all the 3 programs can work with each other if necessary.
2. Since there are multiple operations permissible, the sample output demonstrates many of these operations. We will examine the output piece by piece.
3. We assume that this program is executed after what we had demonstrated in `add_employee.py` (section 14.6). The file `employees.dat` therefore contains 1 employee record (of the employee named Ram).

4. We will begin by listing out the file contents. We expect the employee details of “Ram” to be listed.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
```

5. We will now use the “Add Employee” option to add our second employee to the file.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:1
Enter employee name:Sham
Enter employee ID:2
Enter employee designation:Manager
```

6. Let us now verify that the employee details of “Sham” are indeed present in the file.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Sham ID: 2 Designation: Manager
```

7. We will now similarly add another employee record and verify that it has been successfully saved.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:1
Enter employee name:Anthony
Enter employee ID:5
Enter employee designation:Team Lead
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Sham ID: 2 Designation: Manager
Name: Anthony ID: 5 Designation: Team Lead
```

8. Now that we are convinced that the “Add Employee” and “List Employees” options work perfectly fine, let us explore the “Search Employee” option and search for the employee details of the employee whose ID is 2.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:3
Enter ID to search:2
Name: Sham ID: 2 Designation: Manager
```

9. There, we retrieved the details of the employee named “Sham”. If we attempt to search for a non-existent ID, there will be no output. Let us verify this too.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
```

```
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:3
Enter ID to search:3
Record not found!
```

10. Now let us explore deletion. We will start by attempting to delete a non-existent employee and verify that the file contents are not disturbed.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:4
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Sham ID: 2 Designation: Manager
Name: Anthony ID: 5 Designation: Team Lead
```

11. Despite the attempted deletion, we continue to get the same employee details on listing. We will now delete the employee with ID 2.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:2
Employee Management System
==========================
```

```
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
Name: Anthony ID: 5 Designation: Team Lead
```

12. We observe that the employee record did get deleted! Deletion is the most complex operation in this program, so let us verify that we can indeed delete any record. The previous deletion was an example of deletion of a record somewhere in between the file. Let us verify deletion at the end.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:5
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
Name: Ram ID: 1 Designation: Manager
```

13. Worked! Now finally, deletion of the first record (and the only record of the file)

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:4
Enter ID to delete:1
Employee Management System
```

```
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:2
```

14. Now the file is empty and we have verified all operations. Let us terminate the program.

```
Employee Management System
==========================
1. Add Employee
2. List Employees
3. Search Employee
4. Delete Employee
5. Quit
Enter choice:5
```

Now that we're convinced the program works perfectly, let is analyse the code.

Observation:

1. We import the `os` module for gaining access to the constants used for seeking, and the `pickle` module to be able to read and write objects from/to binary files in lines 5-6.
2. The same `Employee` class that was used in earlier programs is defined in lines 8-12.
3. The `menu()` function in lines 14-22 is used for displaying the program menu, waiting for the user to enter the menu choice and returning the choice made by the user.
4. The `do_add_employee()` function in lines 24-30 is used to add an employee record into the file by taking input from the user.
5. The `do_list_employees()` function in lines 33-39 is used to list all employee records from the file.
6. The `do_search_employee()` function in lines 41-50 searches for an employee with the ID specified by the user and displays it if present, or an error message otherwise.
7. The `do_delete_employee()` function in lines 53-75 deletes an employee

record based on an ID that is provided by the user. For simplicity, if the record is not found, no message is displayed!

8. We will examine each of these functions in subsequent points, but let us first focus on the main program code in lines 77-92.
9. Line 77 specifies the name of the file we will be using in the program. Since we have many different functions that have to work on the same file, and too in perhaps different modes, we prefer to open the file once in a mode that allows reading as well as writing in binary mode and use the file object across all the functions. This file object is created in line 81. If the file exists, it's contents should not be disturbed; whereas if the file does not exist, it should be created. Line 79 verifies the existence of the file using `os.path.isfile()` function, and the mode decision is taken suitably in lines 79-80.
10. Lines 82-89 provide a continuous menu system, using the `menu()` function to display the menu and allow the user to make a choice, and calling other suitable functions based on the user's choice.
11. The `open()` function in line 81 can fail and this exception handling (along with any other like `ValueError` when the user does not provide a numeric input for the menu choice) is performed in lines 90-92.
12. Let us make observations on each of the functions by providing their code again for quick reference.

```
1. def menu():
2.     print("Employee Management System")
3.     print("==========================")
4.     print("1. Add Employee")
5.     print("2. List Employees")
6.     print("3. Search Employee")
7.     print("4. Delete Employee")
8.     print("5. Quit")
9.     return int(input("Enter choice:"))
```

1. The `menu()` function prints a menu and waits the user for making a choice in lines 2-8.
2. The choice is converted to an integer and returned in line 9.

```
1. def do_add_employee(file):
2.     name = input("Enter employee name:")
3.     id = int(input("Enter employee ID:"))
```

```
4.      designation = input("Enter employee designation:")
5.
6.      file.seek(0,os.SEEK_END)
7.      pickle.dump(Employee(name,id,designation),file)
```

1. The `do_add_employee()` function accepts input from the user for a single employee record in lines 2-4.
2. Since we need to add this employee record to the given file and appending is the simplest way of adding to a file, we prefer to seek to the end of the file in line 6.
3. We then create an `Employee` object using the input the user provided and write it to the file in line 7.

```
1. def do_list_employees(file):
2.      try:
3.          file.seek(0)
4.          while 1:
5.              employee = pickle.load(file)
6.              print("Name: {} ID: {} Designation:
   {}".format(employee.name,employee.id,employee.designation
   ))
7.      except EOFError: pass
```

1. The `do_list_employees()` function has to list all employee records from the given file and hence we would want to seek to the beginning of the file to ensure that we start from the first record in the file. This is done in line 3.
2. We read a single employee record at a time in line 5 and display the employee details in line 6. This is repeated infinitely (till `EOFError` occurs on end of file and removes control from the loop).
3. Since we will eventually reach the end of file, we handle this in line 7 and ignore the exception.

```
1. def do_search_employee(file):
2.      id = int(input("Enter ID to search:"))
3.      try:
4.          file.seek(0)
5.          while 1:
6.              employee = pickle.load(file)
7.              if id == employee.id:
```

```
8.                  print("Name: {} ID: {} Designation:
   {}".format(employee.name,employee.id,employee.designation
   ))
9.                  break
10.          except EOFError: print("Record not found!")
```

1. The `do_search_employee()` function allows the user to provide the ID of the employee to search for in line 2.
2. In order to search for that employee within the given file, we need to seek to the beginning in order to start from the beginning of the file. This is done in line 4.
3. Line 5 sets an infinite loop within which we load one employee record at a time from the file (line 6) and check whether the ID matches (line 7). If so, we display the employee details of that employee (line 8) and terminate the loop (line 9). If not, we simply continue within the loop.
4. If the employee record is not found in the file at all, the loop is broken because of the `EOFError` that is raised when we try to go past the end of file, which is handled in line 10.

```
1. def do_delete_employee(file):
2.     id = int(input("Enter ID to delete:"))
3.     file.seek(0)
4.     tempfile = open("temp.dat","w+b")
5.
6.     try:
7.         while 1:
8.             employee = pickle.load(file)
9.             if not employee.id == id:
10.                    pickle.dump(employee,tempfile)
11.          except EOFError: pass
12.
13.          file.seek(0)
14.          tempfile.seek(0)
15.          try:
16.              while 1:
17.                  employee = pickle.load(tempfile)
18.                  pickle.dump(employee,file)
19.          except EOFError:
20.              file.flush()
21.              file.truncate()
22.              tempfile.close()
23.              os.remove("temp.dat")
```

1. The most complex function in this program is the `do_delete_employee()` function that has to delete an employee record from the given file identified by the ID provided by the user. This input is taken in line 2.
2. What makes this function complex is the fact that there is no direct support for deletion of content within files! The technique that we use here is to read each employee record and write it to a temporary file if it is not to be deleted. Thus, the record to be deleted will not be present in the temporary file while all other records would be present. We then transfer all records one by one from the temporary file to the given file and truncate the file to the current position after that to ensure that no other content will be there in the file beyond that point. We can then safely delete the temporary file created.
3. Line 3 ensures that we start from the beginning of the given file and line 4 creates a temporary file.
4. Lines 7-11 read one employee record at a time from the given file and writes it to the temporary file if the ID does not match the ID given by the user.
5. Lines 13-14 then seek back to the beginning of both the files, preparing to transfer content in the other direction now.
6. Lines 68-70 transfer the records from the temporary file to the original file.
7. Line 72 then flushes the contents from the buffer to the file to ensure that data is indeed present in the file rather than just in memory and to ensure that the following call to `truncate()` in line 73 works correctly. The `truncate()` function ensures that the file size is changed to the current position thereby effectively removing any content in the file beyond the current position, if any.
8. Lines 74-75 then close the temporary file and delete it from the filesystem.

14.9 Questions

1. What are the 3 logical steps involved in accessing a file?
2. How would you differentiate between text & binary files? Give examples of text & binary file formats.
3. List and explain all the access modes used to operate on files.
4. How can we confirm if a file has been closed or not?
5. Is it advisable at all times to read the entire contents of a file in one go? If not, why would you avoid it and how?
6. How do we read a file's contents a line at a time?
7. Which function in Python would you use to read few bytes from a file? Give its syntax along with an example.

8. Write a short note on seeking within files.
9. How would you read & write binary data to a file? Illustrate with suitable programs.

14.10 Exercises

1. Accept details of all your friends from standard input (e.g. Name, Date of birth, Hobbies, occupation, residence address) and store it in a binary file as individual records. Allow the user to list out all details of a friend, given his/her name.
2. Create a text file with content as a short summary highlighting your profile and the interesting events that took place in your lifetime. Write a Python program that lists out all repetitive words (e.g '`I`' in the sentence "`I think I will go`").
3. Write a Python script that lists out each line of a file prefixed with it's line number.

SUMMARY

- Text files are human readable whereas binary files require custom applications to read the render their contents.
- The open() function opens the specified file in the specified mode (read mode by default) and returns a file object on success. This file object can then be used to interact with the file contents.
- When done with interacting with a file contents, the file object can be used to invoke the close() method to close the connection with the file. Doing so is optional and all open files are eventually closed when the script ends or when the file object is no longer in use by the program, but explicitly closing files when they are no longer needed can prove efficient in certain situations.
- The read() method can be used to read any number of bytes from a file.
- The readlines() method can be used to read the entire file as a list of lines.
- The readline() method can be used to read a single line at a time. The same effect can be achieved using a for loop to iterate over the file object.
- The write() method can be used to write a string to a file.
- The seek() method can be used to seek to any position within a file, specified as an offset relative to a chosen reference point.
- The tell() method can be used to determine the current position within the file.

SUMMARY

- For text files, seeking is allowed only to one of the standard reference points, or to an offset previously returned by the tell() method on the same file object.
- The pickle module provides a simple and convenient method to read from and write to binary files!
- The pickle.dump() method can be used to write an object to a binary file.
- The pickle.load() method can be used to read an object from a binary file.

15 Modules

In this chapter you will be able to:

- ☑ Appreciate the need for Modules.
- ☑ Understand how to create and import modules.
- ☑ Learn how symbol tables help implement namespaces.
- ☑ Understand module execution and initialisation.
- ☑ Create packages and subpackages to organise modules.
- ☑ Figure out how Python locates modules, packages and subpackages.

MODULES

15.1 Need for Modules

So far, we have seen 2 ways of executing Python code:

1. By typing Python instructions directly into the interpreter and getting it executed immediately
2. By storing Python instructions in a file and running it as a scripting

While the former approach has the advantage of quickly getting the output for any instruction, when it comes to writing larger pieces of code, the latter approach is definitely preferred. In the former approach, all code and data (functions and variables) introduced are forgotten the moment the user quits the interpreter. In the latter approach, the program runs the same way each time the user runs it.

When we write bigger pieces of code, we observe the following:

1. Our program might become too long and we want a way to organise it into manageable chunks (files)
2. We might have multiple Python scripts with various functions already defined that we wish to use in another piece of work

The solution for both the problems above is *modules*! A **module** is a reusable piece of Python code that can be used by other Python programs (and modules). In fact, when we run a Python script, it is assumed to be running as a default module called `__main__`. The name of the current module is available in the variable `__name__`.

```
>>> print(__name__)
__main__
```

By storing a Python script as a module, we gain the following abilities:

1. The contents of the module can be reused by other Python scripts
2. The module so defined is considered to be a separate logical unit, having it's own symbol table where identifiers are stored, preventing clashes with other modules that might have the same name(s) for their identifiers.

By the way, there is nothing much to be done to “convert” a Python script into a Python module! Any Python script can be considered to be a module! However, you might still be interested in differentiating between a module and an executable script, and that is covered in detail in section 15.3.3.

15.2 Creating Modules

Armed with this information of the advantages of using modules and the fact that any Python script can be considered to be a module, let us write our first module! Let us create a module that defines a function `factorial` to find the factorial of an integer. We can then use this module and call the function `factorial` from other Python scripts.

fact.py

```
1.   #!/usr/bin/python
2.
3.   # Module to find the factorial of an integer
4.
5.   def factorial(n):
6.       if n==0: return 1
7.       return n*factorial(n-1)
```

Observation:

1. This is a simple Python script, despite we calling it a module. No special changes were made to convert this script into a module.
2. This module is named `fact` because the filename is `fact.py`. Thus, module names are derived from filenames.
3. This module defines a function called `factorial` that returns the factorial of the given integer.
4. When this script is executed, there is no output as there is no code in the main script. This can be changed however, as covered in section 15.3.3, but for the moment we are happy with the way this module is coded.

15.3 Importing Modules

The previous sections have shown us that modules are meant to be reusable and module names are derived from filenames. Let us now write a Python script that reuses the module `fact` defined earlier. In order to use a module, the module has to be *imported*. Importing a module makes the module name available to us in our script, and using the module name we will be able to access the module contents, as we will soon discover. The syntax for importing a module is:

```
import moduleName
```

In our example, since `fact` is the name of the module, this is how we'd import it before using the factorial function:

```
>>> import fact
```

15.3.1 Module Symbol Tables

Section 15.1 mentioned that one of the advantages of using modules is that each module has it's own symbol table where it's identifiers are stored, thus eliminating clashes with any identifiers with the same name in other modules and scripts. This section will throw more light on this concept.

An *identifier* is a name given by the programmer to a program element, and includes names of functions and variables. These identifiers are stored in a data structure called a *symbol table*. Several symbol tables can co-exist and contain their own identifiers as the symbol tables themselves are identified by a name – the name of the *module* that defined them! Section 15.2 introduced the fact that the module name is derived from the filename. Section 15.1 introduced the fact that executed scripts are assumed to be a part of the module named `__main__`. Thus, every identifier is present within a symbol table identified by a module name or the default module name `__main__`.

Importing a module only adds the module name as an identifier to the symbol table of the importer. The contents of a module are hidden within the symbol table of the module. In our example, the statement `import fact` ensures that the identifier `fact` is present in the symbol table of `__main__`, making it possible for us to access `fact`. The `factorial` function is an identifier within the module `fact` and is thus not directly accessible to us in `__main__`, but can be accessed via `fact`, which is now available to us.

We will now write a program to find the combination using the `factorial` function of the `fact` module:

combination2.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the combination of 2 integers using
modules.
4.
5.   import fact
6.
7.   def ncr(n,r):
8.       return int(fact.factorial(n)/
(fact.factorial(r)*fact.factorial(n-r)))
9.
```

```
10.  x,y = input("Enter 2 integers:").split(' ')
11.  x,y = int(x),int(y)
12.  print("The combination of {} and {} is
{}".format(x,y,ncr(x,y)))
```

Output:

```
Enter 2 integers:5 3
The combination of 5 and 3 is 10
```

Observation:

1. Line 5 imports the `fact` module, making it's content available to us via the identifier `fact`.
2. Line 8 uses the `factorial` function of the `fact` module using the name `fact.factorial()`.

Python does not exactly differentiate between the built-in modules and user-defined modules. You might recall that the built-in `math` module does have a similar `factorial` function defined. However, without importing the `math` module, we won't be able to access the `factorial` function, as shown below:

```
>>> factorial(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'factorial' is not defined
>>> import math
>>> factorial(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'factorial' is not defined
>>> math.factorial(5)
120
```

15.3.2 Importing Names From Module Symbol Tables

The previous section showed how to import modules and the fact that the module's identifiers are not directly available to us. There is a solution, however. If we wish to import a particular identifier (or a set of identifiers) from a module directly to our current symbol table, there is a variant of `import` available with the following syntax:

```
from moduleName import identifier
```

This will copy the identifier `identifier` from the symbol table of module `moduleName` into our current symbol table, making it possible for us to directly access `identifier`, as shown below for the `factorial` function of the `math` module:

```
>>> from math import factorial
>>> factorial(5)
120
```

While this has the advantage that it simplifies the way we use the identifier (`factorial`) without using the module name each time to access it, it also has the disadvantage that it can pollute the current symbol table and increase the number of identifiers, thereby making it possible for clashes to occur as more identifiers from more modules join in similarly. Though convenient, this practice of importing identifiers is frowned upon by professionals.

The question arises: what will happen if we import identifiers with the same name from different modules? The answer is that each `import` can potentially overwrite existing identifiers in the symbol table, thus giving most preference to the last import. This is illustrated below:

```
>>> from math import factorial
>>> factorial(5)
120
>>> from fact import factorial
>>> factorial(5)
120
>>> factorial.__doc__
>>> from math import factorial
>>> factorial.__doc__
'factorial(x) -> Integral\n\nFind x!. Raise a ValueError if x is
negative or non-integral.'
```

Observation:

1. We first imported the `factorial` function from the `math` module and verified that it worked correctly.
2. We then imported the `factorial` function from the `fact` module and verified that we still get the correct output. The fact that there is a clash in the identifier name `factorial` did not result in any error. The fact that both `math.factorial` and `fact.factorial` give us the same output might make it a little difficult for us to figure out which of the two was invoked.
3. Recollect from section 10.15 that functions can have documentation strings. We would expect `math.factorial` to also have it, but our `fact.factorial` did not have one. We can use this knowledge to figure out which module's `factorial` function was called.

4. From the output above, we can see that each `import` overwrites the previous `factorial` with a new copy taken from the specified module.

Here is `combination2.py` rewritten with this neater (but probably unprofessional) syntax:

combination3.py

```
1.   #!/usr/bin/python
2.
3.   # Program to find the combination of 2 integers using
modules.
4.
5.   from fact import factorial
6.
7.   def ncr(n,r):
8.       return int(factorial(n)/(factorial(r)*factorial(n-r)))
9.
10.  x,y = input("Enter 2 integers:").split(' ')
11.  x,y = int(x),int(y)
12.  print("The combination of {} and {} is
{}".format(x,y,ncr(x,y)))
```

Output:

```
Enter 2 integers:5 3
The combination of 5 and 3 is 10
```

Observation:

1. This program is based on `combination2.py`.
2. We have changed line 5 to import only the identifier `factorial` and make it directly available to us.
3. Line 8 uses the `factorial` function directly without using the module name `fact` to identify it.

Before we conclude this section, let us introduce other related forms of importing as well:

It is possible to import multiple identifiers into the current symbol table by giving a tuple of identifiers as shown in the following syntax:

```
from moduleName import identifier1 [,identifier2...]
```

NOTE:

We can import as many identifiers from a module as required. This syntax reduces the number of `import` statements required to achieve this goal.

As a special case, we can import all permissible identifiers from a module into the current symbol table using the following syntax:

```
from moduleName import *
```

NOTE:

While this statement should ideally import all identifiers from the module `moduleName` into the current symbol table, the fact is that the module controls which identifiers are imported when `*` is used! If the module does not explicitly specify what all is permissible to be imported when this syntax is used, the default rule is to import all identifiers that do not have a leading single underscore! Also note that this syntax of importing is frowned upon by professionals as it pollutes the importer's namespace!

15.3.3 Executing Modules

From the previous sections, we have learnt the following:

1. Modules are merely normal Python scripts meant to be reusable
2. Before we can use the contents of a module, we need to import

This section delves deeper into what happens when we import a module.

Importing a module gives a chance for the module to initialize itself. This is done by executing the contents of the module, just as how someone would execute a Python script!

Let us modify our module `fact.py` to demonstrate this property:

fact2.py

```
1.   #!/usr/bin/python
2.
3.   # Module to find the factorial of an integer
4.   # Demonstration of module execution
5.
6.   def factorial(n):
7.       if n==0: return 1
8.       return n*factorial(n-1)
9.
10.  print("Module loaded successfully!")
```

Output:

```
Module loaded successfully!
```

Observation:

1. This program is based on `fact.py` with only 1 addition of line 10.
2. Line 10 prints a message when executed, which we can use as a proof as execution of statements.

We get the above output when we directly run the Python script as a program. But the point is that we will continue to get the same output even if this module is imported, as shown in the below interactive session:

```
>>> import fact2
Module loaded successfully!
```

We therefore conclude that *a module is executed when imported*. However, this execution is performed only when the module is imported for the first time. Thereafter, subsequent imports of the same module do not result in execution of the module.

Interestingly, a module can differentiate between the two forms of execution as follows:

1. If the module is being executed because the user is trying to run it as a script (which we shall call *direct execution*), the value of the variable `__name__` would be `__main__` (as discussed in section 15.1).
2. If the module is being executed because some other script imported it (which we shall call *indirect execution*), the value of the variable `__name__` would be the module name (in this example, `fact2`).

Let us now finally rewrite the previous module so that the output is different for direct and indirect execution.

fact3.py

```
1.   #!/usr/bin/python
2.
3.   # Module to find the factorial of an integer
4.   # Demonstration of module execution
5.
6.   def factorial(n):
7.       if n==0: return 1
8.       return n*factorial(n-1)
9.
10.  if __name__ == "__main__":
11.      # Direct execution
12.      print("Script executed successfully!")
13.  else:
14.      # Indirect execution
15.      print("Module loaded successfully!")
```

Output:

```
Script executed successfully!
```

Interactive session:

```
>>> import fact3
Module loaded successfully!
```

Observation:

1. Line 10 checks if the execution is direct (`__name__` == `__main__`) or indirect (`__name__` would contain the module name).
2. If the script is directly executed, lines 11-12 will be executed, as can be seen from the output above.
3. If the script is indirectly executed by importing it, lines 14-15 will be executed, as can be seen from the interactive session above.

15.3.4 Accessing Symbols From Namespaces

Considering that a module defines a namespace, we will now explore techniques to find out the complete contents of any namespace using the `dir()` function which has the following syntax:

```
dir([moduleName])
```

Forms:

1. `dir()`
2. `dir(moduleName)`

Form #1: `dir()`

This function returns a list of all the identifiers present in the current local scope of the current namespace, excluding the built-in identifiers, as shown below:

```
>>> x=10
>>> def f(x): return x
...
>>> import fact
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'f', 'fact', 'x']
```

Observation:

1. We have created a variable `x`, defined a function `f` and imported a module `fact`.
2. We see that the `dir()` function returns a list of strings with each string naming an identifier in our current namespace. This list includes names of variables (like `x`), names of functions (like `f`) and names of modules (like `fact`).
3. Certain standard identifiers are always included – like `__builtins__` for example!

NOTE:

If you want to see the list of all built-in identifiers, import the module `builtins` (`import builtins`) and pass `builtins` as an argument to `dir()` using the 2nd form (`dir(builtins)`).

Form #2: `dir(moduleName)`

This function lists out all the identifiers present in the namespace of the specified module. Here is an example showing all the built-in identifiers by explicitly checking the contents of the module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError',
'OSError', 'OverflowError', 'PendingDeprecationWarning',
'PermissionError', 'ProcessLookupError', 'RecursionError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '__build_class__', '__debug__', '__doc__',
'__import__', '__loader__', '__name__', '__package__', '__spec__',
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

NOTE:

The second form actually accepts any namespace, not necessarily a module name! Given a reference to an object, for instance, this will return a list of all symbols present within that object!

15.4 Packages

If a module is a collection of reusable code, what is a collection of modules called? Package!

A *package* is a logical collection of modules (and as will be covered in the next section, a package can also contain subpackages). A package makes it possible to logically group related modules so that managing and distributing them becomes easier. Physically in the file system, a package is a *directory* and the modules are *files* within that directory.

Given that a directory is to be considered to be a package, all the Python scripts within that will be considered to be modules. We should therefore have a mechanism to tell Python which directories are to be considered as packages and which are to be treated as merely directories without any other inference. This is done by the presence of a special file called `__init__.py`. The presence of this file in a directory is an indication to Python that the directory is to be considered to be a package. The file could well be empty, but being a Python script, it could also contain executable Python code that is executed when the package is loaded for the first time (just as how modules are executed when imported for the first time). Similarly, the absence of this file in a directory will prevent Python from considering that directory to be a package.

Let us create some sample files to demonstrate packages and accessing it's modules and their content. We will create a package called `pkg1` that contains 2 modules named `mod1` and `mod2` respectively, each containing a function called `f`. We will then see various ways of invoking these functions.

pkg1/__init__.py

```
1.
```

pkg1/mod1.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of modules within packages
4.   # Module mod1 containing function f
5.
6.   def f():
7.       print("This is mod1's f")
```

pkg1/mod2.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of modules within packages
4.   # Module mod2 containing function f
5.
6.   def f():
7.       print("This is mod2's f")
```

Interactive session:

```
>>> import pkg1
>>> from pkg1 import mod1, mod2
>>> mod1.f()
This is mod1's f
>>> mod2.f()
This is mod2's f
```

Observation:

1. The file `pkg1/__init__.py` is empty. We will see what code could be added later on.
2. The statement `import pkg1` makes the symbol `pkg1` available in the program. This statement will also be responsible for executing the file `pkg1/__init__.py`.
3. The statement `from pkg1 import mod1, mod2` makes the symbols `mod1` and `mod2` from within `pkg1` available to us directly. This statement is also responsible for executing the modules `mod1` and `mod2` within the package `pkg1`.
4. The statement `mod1.f()` invokes the function `f` of module `mod1` of package `pkg1`. Similarly, the statement `mod2.f()` invokes the function `f` of module `mod2` of package `pkg1`.

In the above setup, if our goal is to invoke the function `f` of module `mod1` of package `pkg1`, these are a couple of ways of doing it:

```
>>> import pkg1.mod1
>>> pkg1.mod1.f()
This is mod1's f
```

Observation:

1. Do note that each of the examples given here are in different interactive sessions, as in a single session all previous imports are remembered!
2. The statement `import pkg1.mod1` imports the entire symbol `pkg1.mod1`. It is understood from this syntax that `pkg1` is the name of a package whereas `mod1` is the name of a module.
3. The statement `pkg1.mod1.f()` invokes the function `f` of the imported symbol `pkg1.mod1`, which is nothing but the module `mod1` within the package `pkg1`.

```
>>> from pkg1 import mod1
>>> mod1.f()
This is mod1's f
```

Observation:

1. The statement `from pkg1 import mod1` imports the symbol `mod1` from within `pkg1`, making `mod1` directly accessible.
2. The statement `mod1.f()` then invokes the function `f` using the module name that was imported.

```
>>> from pkg1.mod1 import f
>>> f()
This is mod1's f
```

Observation:

1. The statement `from pkg1.mod1 import f` imports the symbol `f` from the module `mod1` within the package `pkg1`.
2. The function `f` can now be directly invoked.

We therefore conclude that the following syntaxes are possible:

```
packageName.moduleName
packageName.moduleName.functionName
packageName.moduleName.variableName
```

We will now demonstrate the execution of packages (`__init__.py`) and modules by making changes in the above setup, creating a new package `pkg2` to demonstrate this:

pkg2/__init__.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of packages
4.   # Package execution
5.
6.   print("Package loaded successfully!")
```

pkg2/mod1.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of modules within packages
4.   # Module mod1 containing function f
5.
6.   def f():
7.       print("This is mod1's f")
8.
9.   print("Module mod1 loaded successfully!")
```

pkg2/mod2.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of modules within packages
4.   # Module mod2 containing function f
5.
6.   def f():
7.       print("This is mod2's f")
8.
9.   print("Module mod2 loaded successfully!")
```

Observation:

1. The filenames `mod1.py` and `mod2.py` do not clash with what we had done previously as they are in a different directory `pkg2`. For the same reason, the modules `mod1` and `mod2` of package `pkg2` will not clash with the modules of the same name in the package `pkg1`.
2. We have added `print` statements in these 3 files to find out when they execute.

The same interactive sessions given for the previous package demo is presented here for this new package:

```
>>> import pkg2.mod1
Package loaded successfully!
Module mod1 loaded successfully!
>>> pkg2.mod1.f()
This is mod1's f
```

```
>>> from pkg2 import mod1
Package loaded successfully!
Module mod1 loaded successfully!
>>> mod1.f()
This is mod1's f
```

```
>>> from pkg2.mod1 import f
Package loaded successfully!
Module mod1 loaded successfully!
>>> f()
This is mod1's f
```

```
>>> from pkg2 import mod1, mod2
Package loaded successfully!
Module mod1 loaded successfully!
Module mod2 loaded successfully!
```

Observation:

1. When a package (or any of it's modules) is imported for the first time, the package gets executed first (and by that we mean that the file `__init__.py` within that package gets executed).
2. A package always gets executed before any of it's modules.

15.5 Subpackages

A package can not only contain modules, but can also contain subpackages (which recursively can contain subpackages and modules). This concept is identical to the physical concept in a filesystem that a directory can not only contain files, but can also contain subdirectories (which recursively can contain subdirectories and files).

Thus, some more valid syntaxes that now become possible are:

```
packageName.subpackageName
packageName.subpackageName.moduleName
packageName.subPackageName.moduleName.functionName
packageName.subPackageName.moduleName.variableName
packageName.subpackageName.subpackageName.moduleName
```

Here is a demonstration of a package `pkg3` contain a subpackage `subpkg`, containing a module `mod`, containing a function `f`:

pkg3/__init__.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of packages
4.   # Subpackage demo
5.
6.   print("Package pkg3 loaded successfully!")
```

pkg3/subpkg/__init__.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of packages
4.   # Subpackage demo
5.
6.   print("Subpackage subpkg loaded successfully!")
```

pkg3/subpkg/mod.py

```
1.   #!/usr/bin/python
2.
3.   # Demonstration of modules within subpackages
4.   # Module mod containing function f
5.
6.   def f():
7.       print("This is mod1's f")
8.
9.   print("Module mod loaded successfully!")
```

Interactive session:

```
>>> from pkg3.subpkg.mod import f
Package pkg3 loaded successfully!
Subpackage subpkg loaded successfully!
Module mod loaded successfully!
>>> f()
This is mod1's f
```

Observation:

1. Subpackages are also packages and must contain the file `__init__.py`.
2. Packages are always executed before their subpackages

15.6 Locating Modules and Packages

We have seen how Python locates and imports packages and modules when using the import statement. What we haven't formally discussed is how exactly Python searches for these files and where all does it search. We do know the following:

1. Section 15.2 introduced the fact that module names match filenames and helps Python locate them.
2. Section 15.4 introduced the fact that package names match directory names and helps Python locate them.

The question we are addressing in this section is: which all directories does Python search for packages and modules, and if we wish to add certain additional directories to the search path, how do we do so? Let's address this piece by piece.

Whenever a package or module is imported, Python first checks if it refers to a built-in package/module, and if so imports it directly. Otherwise, Python assumes it to be a library package/module or user-defined package/module and relies on a variable `sys.path`.

Python searches for packages and modules in the list of directories stored in the

variable `sys.path`. The contents of `sys.path` on my system is shown below:

```
>>> import sys
>>> sys.path
['', '/usr/lib64/python35.zip', '/usr/lib64/python3.5',
'/usr/lib64/python3.5/plat-linux', '/usr/lib64/python3.5/lib-dynload',
'/usr/lib64/python3.5/site-packages', '/usr/lib/python3.5/site-
packages']
```

Observation:

1. Any package/module imported will be searched through the above directories in sequential order till it is found. Thus for example, preference is given to the directory `/usr/lib64/python35.zip` rather than `/usr/lib64/python3.5`. If the package/module is not found in any of directories listed above in the output, it is an error.
2. The first entry in `sys.path` is empty and represents the current working directory. This means that packages and modules in the current working directory is given more preference than the library modules! What this means is that we can provide replacements for library packages and modules!

The next question is how is `sys.path` built? The contents of `sys.path` are loaded as follows:

1. The first entry is always the current working directory.
2. The second set of entries are picked up from the environment variable `PYTHONPATH`, if present. This variable has the same usage as `PATH` – it is a list of directories separated by a delimiter. The delimiter is '`:`' is UNIX/Linux and '`;`' in Windows. This would be a good place to store locations of packages and modules without affecting the source code.
3. The standard library locations are then appended to `sys.path`. These are dependent on the installation.

We can change the contents of `sys.path` in our script. Modifying `sys.path` directly in our script can be a replacement for loading them from `PYTHONPATH`. Let us remove the first entry in `sys.path` (referring to the current directory) and see the impact on importing our packages/modules stored in the current directory:

```
>>> import sys
>>> sys.path
['', '/usr/lib64/python35.zip', '/usr/lib64/python3.5',
'/usr/lib64/python3.5/plat-linux', '/usr/lib64/python3.5/lib-dynload',
'/usr/lib64/python3.5/site-packages', '/usr/lib/python3.5/site-
packages']
>>> del sys.path[0]
>>> sys.path
['/usr/lib64/python35.zip', '/usr/lib64/python3.5',
'/usr/lib64/python3.5/plat-linux', '/usr/lib64/python3.5/lib-dynload',
'/usr/lib64/python3.5/site-packages', '/usr/lib/python3.5/site-
packages']
>>> import pkg1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'pkg1'
```

As can be seen, Python is now unable to import the package `pkg1` from our current directory as the current directory is no longer searched in. Let us start a new session and add `pkg1` to `sys.path` and see how we can load the module `mod1`:

```
>>> import sys
>>> sys.path.append('pkg1')
>>> import mod1
>>> mod1.f()
This is mod1's f
```

Adding `pkg1` in `sys.path` makes Python search for packages and modules within the directory `pkg1` also. This makes it possible to directly load the module `mod1` from within the directory `pkg1`.

As our last demo in this section, let us add `pkg1` to `PYTHONPATH` and import `mod1` from `pkg1` directly.

The following statement is being typed in a Linux shell (**not in Python interpreter**):

```
export PYTHONPATH="pkg1"
```

This is then followed by a Python interactive session:

```
>>> import mod1
>>> mod1.f()
This is mod1's f
```

As can be seen, we were able to locate `mod1` within `pkg1`. As further proof that `sys.path` was modified with the inputs taken from `PYTHONPATH`, here is a continuation of the same session:

```
>>> import sys
>>> sys.path
['', '/home/nagesh/PythonBook/programs/pkg1',
'/usr/lib64/python35.zip', '/usr/lib64/python3.5',
'/usr/lib64/python3.5/plat-linux', '/usr/lib64/python3.5/lib-dynload',
'/usr/lib64/python3.5/site-packages', '/usr/lib/python3.5/site-
packages']
```

While the first entry is for the current working directory, the second entry was taken from `PYTHONPATH`. The rest of the entries are as before.

15.7 Installing Packages and Modules

There are many packages available for use with Python – some of them are available when you install Python whereas others may have to be manually installed. This section shows how to install packages and modules using Python's installer – PIP (a recursive acronym for PIP Installs Packages).

The syntax for installing a package using PIP is shown below:

```
python -m pip install packageName
```

Observation:

1. We are running the Python interpreter (`python`) explicitly. If there are multiple Python interpreters installed, we can choose the appropriate one to be used for package installation.
2. The `-m` option specifies that we are interested in running a named module (`pip`) as a script.
3. The argument `install` is an argument for the PIP script that denotes that we wish to install a package.
4. Finally, the `packageName` argument specifies the name of the package to be downloaded and installed.

On many systems, PIP might be directly available as a command. On systems where both Python 2 and Python 3 are installed, PIP might be available as `pip` and `pip3` for these two version respectively. On such systems, the syntax for package installation is as follows:

```
pip install packageName
pip3 install packageName
```

15.8 Questions

1. How are modules helpful in managing large projects?
2. Write a short note on how a module can be created and used in Python.
3. Explain the various ways of importing module contents with examples.
4. How does Python know where to search for modules and packages? Explain various techniques to help Python locate custom modules.
5. How are packages different from modules?

15.9 Exercises

1. Define a module called `prime` that contains a function `isPrime()` that returns whether the passed argument is prime or not. Using this module and function, write another program containing a function `printPrimes()` that prints the first `n` prime numbers.
2. Define a module called `factorial` that contains a function to find the factorial of the given integer. Using this function, find the permutation and combination of the given inputs.
3. Create a package `P` with module `M` that contains a function `F`. Demonstrate various ways of calling the function `F` using different programs.

SUMMARY

- A module makes its contents reusable for other programs.
- No special step is needed to convert a Python script into a module!
- Each module has a symbol table of its own to maintain its identifiers. The import statement helps transfer symbols into the current namespace.
- Modules get executed when imported for the first time! The special symbol __name__ will contain the module name when executed due to an import, and will have the value "__main__" instead when being executed as a script.
- The dir() function can be used to list out any namespace content.
- A package is a logical collection of modules and is physically a directory. A sub-package is similarly a package within a package, stored as a sub-directory.
- A directory is considered to be a package only if it contains the file __init__.py, which is executed when the package is loaded for the first time.
- The environment variable PYTHONPATH and the Python variable sys.path can be used to specify where to locate packages and modules.

16 Working with Databases

In this chapter you will be able to:

- ☑ Recap the fundamentals of SQL, DDL, DML and DQL.
- ☑ Create and work with SQLite databases
- ☑ Execute queries and fetch query results
- ☑ Create in-memory temporary databases

Working with Databases

16.1 Introduction

This chapter is not intended as a complete material on databases and SQL programming. That would perhaps require a complete book of it's own! We will nevertheless explore some basic concepts rapidly that will allow us to focus on how we can deal with SQLite databases in Python in the main part of this chapter.

16.1.1 Introduction to Databases

Whenever we find the need to deal with huge amounts of data, we find a need for organising them in a logical fashion that makes it manageable to store and access that data. A **database** is precisely that: an organised collection of data. In more complicated systems, a **Database Management System (DBMS)** takes care of interfacing with and managing the database.

A **Relational DBMS (RDBMS)** is a type of database system that is based on the concept of relations that help link data stored across different tables. These are by far the most popular form of databases, though of late NewSQL is gaining prominence.

Since Object Oriented Programming became very popular, databases also evolved to natively support objects – such a database system is called an **Object Database** (if objects can be stored and retrieved directly) or an **Object Relational Database** (if the underlying storage is relational but the view provided is object-oriented).

As mentioned earlier, what is gaining prominence is NewSQL and NoSQL. **NoSQL** deals with key-value pairs as opposed to the standard table structure used in RDBMS. **NewSQL** emphasises on the use of SQL to deal with NoSQL databases, keeping scalability and compatibility in mind.

The most common language used for programming and interacting with databases is **SQL – Structured Query Language**. SQL comprises of a set of standard statements having a pre-defined format. The language is English-like and case insensitive. SQL statements are broadly classified into:

1. **Data Definition Language (DDL)** that comprises of statements that deal with tables – their structures, creation, destruction, etc. Example: the `CREATE TABLE` statement.
2. **Data Control Language (DCL)** that comprises of statements that deal with access rights and permissions. Example: the `GRANT` statement.
3. **Data Manipulation Language (DML)** that comprises of statements that deal with changes to data. Example: the `INSERT` statement.
4. **Data Query Language (DQL)** that comprises of the `SELECT` statement in all it's forms and basically deals with the extraction of data.

The next few sections will deal with DDL, DML and DQL in brief. We will not be considering DCL as access rights is not a critical issue when simple databases are created using SQLite.

16.1.2 An Overview of Data Definition Language (DDL)

The most commonly used DDL SQL statements are `CREATE TABLE` to create a table and `DROP TABLE` to delete and remove a table.

16.1.2.1 The CREATE TABLE Statement

Here is an example of a DDL statement to create an `Employee` table:

```
CREATE TABLE Employee
(
      name  TEXT,
      id    INTEGER,
      email TEXT
)
```

Observation:

1. SQL is not case sensitive and therefore we can use any case we wish, but we will stick to upper-case for keywords and lower-case for identifier names as a convention. This convention is followed by many.
2. This `Employee` table will have 3 columns or fields: `name`, `id` and `email`.
3. `TEXT` and `INTEGER` are standard data types. `TEXT` can help represent any textual content of varying length whereas `INTEGER` can help represent any normal integer.

It is possible to create a table conditionally only if it does not already exist. This uses a slightly different syntax:

```
CREATE TABLE IF NOT EXISTS Employee
(
      name  TEXT,
      id    INTEGER,
      email TEXT
)
```

16.1.2.2 The DROP TABLE Statement

In order to remove a table that is no longer required, the `DROP TABLE` statement can be used as follows:

```
DROP TABLE Employee
```

> **NOTE:**
>
> Dropping a table will not only remove the table, but also all data along with it! This is potentially dangerous to be executed accidentally!

It is possible to drop a table conditionally only if it exists. Any attempt to drop a table that does not exist will otherwise give rise to a syntax error. Conditional dropping of a table is demonstrated below:

```
DROP TABLE IF EXISTS Employee
```

16.1.3 An Overview of Data Manipulation Language (DML)

The most commonly DML SQL statements are `INSERT` to add rows to a table, `UPDATE` to modify rows in a table and `DELETE` to remove rows from a table.

16.1.3.1 The INSERT Statement

Once the `Employee` table is created, we can add rows into it as demonstrated below:

```
INSERT INTO Employee VALUES("Ram",1,"ram@company.com")
```

16.1.3.2 The UPDATE Statement

In order to change an existing row in the `Employee` table, we can use the `UPDATE` statement as demonstrated below:

```
UPDATE Employee SET email="ram2@company.com" WHERE id=1
```

The above statement will replace the email address of any record(s) to "`ram2@company.com`" whenever it finds that the `id` in that record is `1`.

16.1.3.3 The DELETE Statement

In order to delete 1 or more rows from a table, the `DELETE` statement can be used as follows:

```
DELETE FROM Employee WHERE name=”Ram”
```

As a special case, in order to delete all records of the table, the following `DELETE` statement can be used:

```
DELETE FROM Employee
```

NOTE:

This is not the same as `DROP TABLE Employee`, which not only deletes the rows but also removes the table!

16.1.4 An Overview of Data Query Language (DQL)

Perhaps the most powerful statement – and also the most frequently used one – in SQL is the `SELECT` statement, capable of retrieving data from tables:

```
SELECT * FROM Employee
```

The above statement fetches all records from the table `Employee`. If we wish to filter the results, we could do so as shown in the following example:

```
SELECT * FROM Employee WHERE id<10
```

16.1.5 Introduction to SQLite

All the above examples in this chapter are valid SQL examples, but were more fine tuned for use with SQLite. We thus saw data types like `TEXT` instead of the more traditional `VARCHAR`. Though SQLite supports primary keys the way other popular RDBMS do, joins across tables might be less frequently used here. Perhaps we need to first understand what is SQLite in order to better understand why things are slightly different in this RDBMS!

SQLite is a C implementation of an embeddable RDBMS. It is thus highly efficient, highly portable (is shipped with the application and is not dependent on any databases to be installed/deployed there) and pretty simple (since a single application

might rarely want to create large databases and store huge amounts of data). The installation process is simple and direct and so are the databases. Each database is stored locally as a file. In fact, if required, a database can be temporarily created directly in memory! This of course will have the limitation that when the application terminates, the database vanishes too!

16.2 Installation

SQLite comes built-in in Python 3 and hence requires no installation. More information can be obtained on downloading the source and documentation at https://www.sqlite.org/.

16.3 Connecting and Closing

All functionality of SQLite is provided by the `sqlite3` module, which has to be imported first.

Before we can access the contents of a database, we need to establish a connection with. This is done by the `connect` function whose syntax is shown below:

```
sqlite3.connect(databaseName)
```

Recall from section 16.1.5 that SQLite stores databases as files. The `databaseName` parameter shown in the syntax above is the filename (or pathname if the file is not in the current directory) of the database file to be used. If the file does not exist, it will be created.

NOTE:

As a special case, if the `databaseName` parameter is "`:memory:`", the database is created in memory!

The `connect()` function returns a `Connection` object that will be used further in section 16.4, but for now let us also see how to close a database connection using the `close()` method, whose syntax is shown below:

```
connection.close()
```

From the time a database connection is opened till the time it is closed, there could be any number of read/write operations on the database. None of the write operations would be committed to the database without an explicit call to the `commit()` method! It might be a good idea to follow the practice of committing before closing the database connection!

The syntax of the `commit()` method is shown below:

```
connection.commit()
```

Here is a sample run using the above functions:

```
>>> import sqlite3
>>> conn=sqlite3.connect("test.db")
>>> conn
<sqlite3.Connection object at 0x7f596a4183b0>
>>> conn.commit()
>>> conn.close()
```

16.4 Executing Queries

16.4.1 Working with Cursors

Now that we know how to create a connection with a database, let us see how we can execute queries on that database using the connection. A `Cursor` instance acts as an interface to execute queries on a `Connection` and can be created from a `Connection` using the `cursor()` method which has the following syntax:

```
connection.cursor()
```

We can create any number of `Cursor` objects from a single connection and using each we can execute any number of queries through the connection. While it is not necessary to create multiple `Cursor` objects to execute multiple queries, do bear in mind that if we are not done extracting the complete result of the previous query and execute the next query, we will lose the remaining results from the previous query. There are therefore situations wherein we may have to create multiple `Cursor` objects.

Retrieving results from `Cursor` objects will be dealt with later in section 16.5, but for now let us see how to execute queries that don't produce results (DML queries).

16.4.2 Executing a Single Query

In order to execute a query, we use the `execute()` method of `Cursor` which has the following syntax:

```
cursor.execute(query)
```

We will first demonstrate how to create the `Employee` table discussed in section 16.1.2.1 using this method:

```
>>> import sqlite3
>>> conn=sqlite3.connect("employee.db")
>>> cursor=conn.cursor()
>>> query="CREATE TABLE Employee( name TEXT, id INTEGER, email
TEXT)"
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x7f596a1ba490>
>>> conn.commit()
>>> conn.close()
```

Now that we have created the table and committed it, it will be available for us anytime we want, even in a different Python session!

Let us launch another Python interpreter session and continue writing a record into the table created.

```
>>> import sqlite3
>>> conn=sqlite3.connect("employee.db")
>>> cursor=conn.cursor()
>>> query="""INSERT INTO Employee
VALUES("Ram",1,"ram@company.com")"""
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x7f35d86e7490>
>>> conn.commit()
>>> conn.close()
```

16.4.3 Using Placeholders

Many a times, we wish to frame a query template containing placeholders for values that will be filled in dynamically with suitable values when the query is to be executed. It is possible to frame such a query, using "`?`" as placeholders, as shown in the example below that continues to write 2 more records to the `Employee` database using a single query template:

```
>>> import sqlite3
>>> conn=sqlite3.connect("employee.db")
>>> cursor=conn.cursor()
>>> query="""INSERT INTO Employee VALUES(?,?,?)"""
>>> cursor.execute(query,("Sham",2,"sham@company.com"))
<sqlite3.Cursor object at 0x7f35d86e7500>
>>> cursor.execute(query,("Balram",3,"ballu@company.com"))
<sqlite3.Cursor object at 0x7f35d86e7490>
>>> conn.commit()
>>> conn.close()
```

Observation:

1. We have framed the query only once without hard-coding the values, and used it twice with different values.
2. The `execute()` method now expects a second argument – a tuple of values – to be assigned on a one-to-one basis for each placeholder found in the query.
3. In case you are wondering that `String.format()` could also have been used instead, here's the difference: using placeholders is far safer as it protects you from SQL injection! Without going into the details of that, here's the short conclusion: it is not safe to use `String.format()` as a replacement for the placeholders concept!

When using placeholders, we use a slightly different syntax of `execute()` as shown below:

```
cursor.execute(query_with_placeholders, tuple_of_values)
```

16.4.4 Executing Multiple Queries

The previous section showed how we can frame a single query template but provide a different tuple of values each time we call the execute() method. In this section we will examine how we can pass an entire sequence of such tuples to run the same query template multiple times on different tuples of values!

We use the `executemany()` method which has the following syntax:

```
cursor.executemany(query_with_placeholders, seq_of_tuples)
```

Here's a demonstration of adding 2 more records into our `Employee` database using this concept:

```
>>> import sqlite3
>>> conn=sqlite3.connect("employee.db")
>>> cursor=conn.cursor()
>>> query="""INSERT INTO Employee VALUES(?,?,?)"""
>>> employees=[("Sita",10,"sita@company.com"),
("Gita",11,"gita@company.com")]
>>> cursor.executemany(query,employees)
<sqlite3.Cursor object at 0x7f35d86e7490>
>>> conn.commit()
>>> conn.close()
```

Observation:

1. We have used the same query template as before
2. We have framed a list of tuples, with each tuple representing 1 `Employee` record. This need not be a list – it can be any sequence, for example another tuple.
3. We have passed this sequence as an argument to `executemany()` the same way we pass a single tuple as an argument to `execute()`.
4. We now should be having totally 5 records in our database if we have executed all the above code snippets. The employee names in our database would be `Ram`, `Sham`, `Balram`, `Sita` and `Gita`.
5. So far, we haven't been able to verify whether our data in indeed present in the database. The next section will demonstrate how we can extract data from the database and verify the working of all the code we executed so far.

16.5 Fetching Responses

Each `Cursor` object, after executing a query, is capable of providing the result of the query execution. That is why in section 16.4.1 we had mentioned the need for multiple `Cursor` objects in some cases.

We can either:

1. Fetch the result a row at a time using `Cursor.fetchone()`
2. Fetch many rows at a time using `Cursor.fetchmany()`
3. Fetch all result rows at a time using `Cursor.fetchall()`

The following sections demonstrate how to use these methods.

16.5.1 Fetching Results Using fetchone()

The `Cursor.fetchone()` method fetches a single row of result and returns it. As a special case, where there are no more result rows to fetch, it returns `None`. This is demonstrated below:

```
>>> import sqlite3
>>> conn=sqlite3.connect("employee.db")
>>> cursor=conn.cursor()
>>> query="SELECT * FROM Employee"
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x7fb7281ed490>
>>> while True:
...     row=cursor.fetchone()
...     if row is None: break
...     print(row)
...
('Ram', 1, 'ram@companycom')
('Sham', 2, 'sham@company.com')
('Balram', 3, 'ballu@company.com')
('Sita', 10, 'sita@company.com')
('Gita', 11, 'gita@company.com')
```

Observation:

1. We have connected to the same database – `employee.db` – and are trying to list out all records in the database.
2. Each row, when printed, is printed as a tuple of column values.
3. We iterate row by row till the `fetchone()` method returns `None`.
4. Since we have not written anything into the database, there is no need to invoke `commit()`.

16.5.2 Fetching Results By Iterating

Another alternative to fetch result rows sequentially is to iterate through the `Cursor` object as demonstrated below:

```
(continuation of previous session)
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x7fb7281ed490>
>>> for row in cursor:
...     print(row)
...
('Ram', 1, 'ram@companycom')
('Sham', 2, 'sham@company.com')
('Balram', 3, 'ballu@company.com')
('Sita', 10, 'sita@company.com')
('Gita', 11, 'gita@company.com')
>>>
```

Observation:

1. This is a continuation of the previous session and hence we don't have to re-establish connection with the database or create a new `Cursor` object.
2. We need to execute the query again so as to obtain the results. We can either reuse the same `Cursor` object as has been done, or can create a new `Cursor` object for this (which is unnecessary).
3. We are merely iterating over the `Cursor` object and end up iterating over the result rows. This is perhaps more readable and simple than using `fetchone()`.

16.5.3 Fetching Results Using fetchmany()

The previous sections showed how to fetch a single result record at a time. While doing so might be simple, it is also inefficient. It is more efficient to fetch a block of rows at a time using `fetchmany()` which has the following syntax:

```
Cursor.fetchmany([size])
```

Observation:

1. The optional `size` argument specifies the number of result rows we wish to fetch. In case those many rows are unavailable, it will anyway return how many ever rows are available.
2. It is best to omit the `size` parameter and allow the `Cursor` object to decide the best size. This is guaranteed to be optimal.

Let us continue the Python session and fetch all result rows using `fetchmany()` this time:

```
(continuation of previous session)
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x7fb7281ed490>
>>> while True:
...     rows=cursor.fetchmany(3)
...     if len(rows) == 0: break
...     print(rows)
...
[('Ram', 1, 'ram@companycom'), ('Sham', 2, 'sham@company.com'),
('Balram', 3, 'ballu@company.com')]
[('Sita', 10, 'sita@company.com'), ('Gita', 11,
'gita@company.com')]
```

Observation:

1. Again, this is a continuation of the previous session and we have executed the query again using the same `Cursor` object.
2. We are attempting to read `3` rows at a time. We break out of the loop when the number of rows returned in `0`.
3. Since there were 5 rows in the database, we see that the first iteration produced the first 3 rows, the second iteration produced the remainder 2 rows and the third iteration produced 0 rows (and that's when we break out of the loop).

16.5.4 Fetching Results Using fetchall()

Another version of the `fetchmany()` method is the `fetchall()` method that returns all the result rows at one go. Note that this is not advisable when we are expecting a lot of rows as the entire list of rows will be stored in memory!

We revisit the previous session to demonstrate the use of `fetchall()` to fetch all the result rows:

```
(continuation of previous session)
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x7fb7281ed490>
>>> rows=cursor.fetchall()
>>> print(rows)
[('Ram', 1, 'ram@companycom'), ('Sham', 2, 'sham@company.com'),
('Balram', 3, 'ballu@company.com'), ('Sita', 10,
'sita@company.com'), ('Gita', 11, 'gita@company.com')]
```

Observation:

1. As we can see, all the result rows have been returned at one go. In the special case that there were no result rows at all, we would have obtained an empty list.

16.6 Working With Row Objects

Section 16.5 showed a variety of ways by which we can extract query result rows using the `Cursor` object. For better ease of use, it is possible to deal with each result row as a `Row` object instead of a tuple. This section focusses on how we can extract columns of a `Row` object.

In order to deal with rows as `Row` objects instead of tuples, the following statement must be executed before we attempt to create the `Cursor` object from the `Connection` object:

```
>>> conn.row_factory=sqlite3.Row
```

Here is a small session that demonstrates the use of `Row` objects:

```
>>> import sqlite3
>>> conn=sqlite3.connect("employee.db")
>>> conn.row_factory=sqlite3.Row
>>> cursor=conn.cursor()
>>> query="SELECT * FROM Employee"
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x7fb46b1d9490>
>>> row=cursor.fetchone()
>>> row
<sqlite3.Row object at 0x7fb469682a90>
```

16.6.1 Row Objects as Tuples

`Row` objects behave like tuples. We can use the `len()` function to find the number of columns within the row and can iterate over the row contents, which will give us the column values in sequence, as demonstrated below:

```
(continuation of previous session)
>>> tuple(row)
('Ram', 1, 'ram@companycom')
>>> len(row)
3
>>> for value in row:
...     print(value)
...
Ram
1
ram@companycom
```

16.6.2 Accessing Columns by Indices

Just like tuples, individual elements of a `Row` object can be accessed via the subscript operator:

```
(continuation of previous session)
>>> row[0]
'Ram'
>>> row[1]
1
>>> row[2]
'ram@companycom'
```

16.6.3 Accessing Columns by Name

What makes the `Row` object more powerful than tuples is that it is possible to also treat a `Row` object like a dictionary, with the keys being the column names and the values being the corresponding value taken from the result:

```
(continuation of previous session)
>>> row['name']
'Ram'
>>> row['id']
1
>>> row['email']
'ram@companycom'
```

16.7 Questions

1. Explain how SQL queries can be executed in Python using SQLite.
2. Explain the usage of placeholders in SQL query execution using examples.
3. Write a short note on extracting SQL query results as rows in Python.
4. Write a short note on extracting column values from within an SQLite Row object.

16.8 Exercises

1. Write a program to create an SQLite database in the file `employee.db` that contains a table called `Employee`, with fields `ID`, `name`, `department`, `designation` and `city`.
2. Write a program that allows the user to add multiple records into the `employee.db` file created earlier. After every record, the user should be asked whether he/she wants to add another record.
3. Write a program that allows the user to edit an entry present in the `employee.db` file created earlier. The program should ask for the `ID` of the employee, and should take in the new details for that employee.
4. Write a program that allows the user to delete 1 or more records from the file `employee.db` created earlier. The input is a single line containing the `ID` of the employees to be deleted, separated by spaces.
5. Write a program that displays all the records present in the file `employee.db` created earlier in a formatted manner.
6. Write a menu-driven program that works with the `employee.db` file created earlier and provides the following options:
    1. Searching for a particular employee by `ID`
    2. Listing all employees belonging to a particular department
    3. Listing all employees belonging to a particular city
    4. Listing the number of people in a particular city having a particular designation
    5. Listing the total number of employees in each city

Summary

- SQLite is an embedded RDBMS that stores databases as files. It is also capable of storing databases directly in memory!
- The built-in module sqlite3 provides the functionality to work with SQLite.
- Connections are objects of the Connection class and can be obtained using the connect() function, and can be closed using the close() method.
- Queries can be executed using execute() or executemany() methods of the Cursor object, which can be obtained from a Connection using it's cursor() method.
- Results from query execution can be obtained using the fetchone(), fetchmany() and fetchall() methods of Cursor object. It is also possible to iterate over a Cursor object directly to process the query results row-wise.
- We can also deal with query results using Row objects instead of the default tuples. Row objects can behave both as tuples and as dictionaries.

17 Parsing HTML

In this chapter you will be able to:

- ☑ Parse through HTML content and write custom parsers using HTMLParser
- ☑ Read and Write HTML content using the BeautifulSoup module
- ☑ Extract details from a HTML parse tree
- ☑ Navigate through a HTML parse tree
- ☑ Search for tags, attributes and content within a HTML parse tree
- ☑ Download HTML content from a URL using the urllib package
- ☑ Parse through remote HTML content
- ☑ Download images from a remote URL

Parsing HTML

17.1 Introduction

HyperText Markup Language (HTML) is the language of the web. Whenever content has to be made available across the web, the most preferred form is through HTML output. This chapter concentrates on how HTML can be parsed in Python and how it can be created.

Python 3 comes with a built-in HTML parser. We will first explore how to parse HTML content using this. Thereafter, we will explore another library called **BeautifulSoup** (which internally uses Python's HTML parser) that provides more powerful and easier to use interface to HTML documents.

17.2 Using the HTML Parser

Python's built-in HTML parser is available through the `html.parser` module's `HTMLParser` class. This is the procedure for parsing HTML code using this:

1. Define a class that derives from `HTMLParser`
2. Override the methods we are interested in and provide suitable implementation
3. Create an instance of our class
4. Feed in HTML data to that instance
5. Suitable methods of our class get called automatically depending on the HTML content parsed

The following sections show how we can achieve each of these steps.

17.2.1 Deriving a Class from HTMLParser

The following sections cover the frequently called methods of `HTMLParser` that we might be interested in overriding:

17.2.1.1 Handling Tags

The most commonly occurring content in HTML is tags. Tags may be paired (Example: `<b>Hello world</b>`) or unpaired (Example: `<br/>`).

Whenever the parser finds the beginning of a tag, it will invoke the `handle_starttag` method which has the following syntax:

```
HTMLParser.handle_starttag(self, tag, attrs)
```

While the first parameter has to be `self` (as this is an instance method), the second parameter (`tag`) identifies the name of the tag found and the third parameter (`attrs`) is a list containing all the attributes present with the tag, with each attribute-value pair being stored as a tuple.

Here is a sample code to demonstrate how this works:

```
>>> from html.parser import HTMLParser
>>> class Parser(HTMLParser):
...     def handle_starttag(self, tag, attrs):
...             print("Found tag:", tag)
...             print(" Attributes:", attrs)
...
>>> data='<html><body><p class="para text" id="1">This is a
<b>test</b><p><br/></body></html>'
>>> parser=Parser()
>>> parser.feed(data)
Found tag: html
 Attributes: []
Found tag: body
 Attributes: []
Found tag: p
 Attributes: [('class', 'para text'), ('id', '1')]
Found tag: b
 Attributes: []
Found tag: p
 Attributes: []
Found tag: br
 Attributes: []
```

Observation:

1. We imported the `HTMLParser` class from the `html.parser` module, defined our class (`Parser`) that derived from it, overrode the method `handle_starttag` and instantiated our class.
2. We used the `feed()` method to feed in HTML data. We will explore this in detail later in section 17.2.2.
3. We are merely printing the tag name and it's attribute as and when the parser encounters it.
4. We observe that parsing commences immediately after feeding in data.
5. We observe that we encounter the tags in the same order as they are present

in the HTML input.

6. We observe that all attributes are also made available as a list of tuples. Of course most of the tags did not have any attributes and therefore the list was empty.

Similarly, whenever an ending tag is encountered, the `handle_endtag` method is invoked, which has the following syntax:

```
HTMLParser.handle_endtag(self, tag)
```

Note that the end tag cannot contain any attributes and therefore we don't have the additional `attrs` parameter that we had in `handle_starttag`.

We will now demonstrate the handling of end tags in addition to start tags. Since we are reusing the code written earlier, it would be easier for us to write this as a program instead:

htmlDemo1.py

```
1.   #!/usr/bin/python
2.
3.   # Program to demonstrate handling of start tags
4.   # and end tags in HTML content
5.
6.
7.   from html.parser import HTMLParser
8.
9.   class Parser(HTMLParser):
10.      def handle_starttag(self, tag, attrs):
11.          print("Found tag:", tag)
12.          print(" Attributes:", attrs)
13.
14.      def handle_endtag(self, tag):
15.          print("End tag:", tag)
16.
17.  data='<html><body><p class="para text" id="1">This is a
<b>test</b><p><br/></body></html>'
18.  parser=Parser()
19.  parser.feed(data)
```

Output:

```
Found tag: html
 Attributes: []
Found tag: body
 Attributes: []
Found tag: p
 Attributes: [('class', 'para text'), ('id', '1')]
Found tag: b
 Attributes: []
End tag: b
Found tag: p
 Attributes: []
Found tag: br
 Attributes: []
End tag: br
End tag: body
End tag: html
```

Observation:

1. In addition to `handle_starttag`, we have now defined even `handle_endtag`.
2. The start tags and end tags are printed in the order they occur in the HTML content.

Unpaired tags of the form `<br/>` are reported in the `handle_startendtag` method that has the following syntax:

```
HTMLParser.handle_startendtag(self, tag, attrs)
```

The role of each parameter is exactly the same as in `handle_starttag`.

NOTE:

If we do not override `handle_startendtag`, the default implementation of this in `HTMLParser` class is invoked, which ends up calling both `handle_starttag` and `handle_endtag` for this tag!

Here is a program with `handle_startendtag` defined:

htmlDemo2.py

```
1.   #!/usr/bin/python
2.
3.   # Program to demonstrate handling of start tags
4.   # and end tags in HTML content
5.
6.
7.   from html.parser import HTMLParser
8.
9.   class Parser(HTMLParser):
10.      def handle_starttag(self, tag, attrs):
11.          print("Found tag:", tag)
12.          print(" Attributes:", attrs)
13.
14.      def handle_endtag(self, tag):
15.          print("End tag:", tag)
16.
17.      def handle_startendtag(self, tag, attrs):
18.          print("Found unpaired tag:", tag)
19.          print(" Attributes:", attrs)
20.
21.
22.  data='<html><body><p class="para text" id="1">This is a
<b>test</b><p><br/></body></html>'
23.  parser=Parser()
24.  parser.feed(data)
```

Output:

```
Found tag: html
 Attributes: []
Found tag: body
 Attributes: []
Found tag: p
 Attributes: [('class', 'para text'), ('id', '1')]
Found tag: b
 Attributes: []
End tag: b
Found tag: p
 Attributes: []
Found unpaired tag: br
 Attributes: []
End tag: body
End tag: html
```

Observation:

1. This program is pretty much like the previous one, except for the addition of the `handle_startendtag` method in line 17.
2. From the output, it is now evident that the HTML content `<br/>` did not invoke `handle_starttag` and `handle_endtag`, but instead only invoked `handle_startendtag`.

17.2.1.2 Handling Data

After tags, the next important piece of information in an HTML document is the textual content itself. Whenever the parser finds textual information, it invokes the `handle_data` method, which has the following syntax:

```
HTMLParser.handle_data(self, data)
```

The actual text content obtained is sent via the `data` parameter.

Let us rewrite our program to also handle text:

htmlDemo3.py

```
1.   #!/usr/bin/python
2.
3.   # Program to demonstrate handling of tags
4.   # and text content in HTML content
5.
6.
7.   from html.parser import HTMLParser
8.
9.   class Parser(HTMLParser):
10.      def handle_starttag(self, tag, attrs):
11.          print("Found tag:", tag)
12.          print(" Attributes:", attrs)
13.
14.      def handle_endtag(self, tag):
15.          print("End tag:", tag)
16.
17.      def handle_startendtag(self, tag, attrs):
18.          print("Found unpaired tag:", tag)
19.          print(" Attributes:", attrs)
20.
21.      def handle_data(self, data):
22.          print("Found data:", data)
23.
24.  data='<html><body><p class="para text" id="1">This is a
<b>test</b><p><br/></body></html>'
```

```
25.  parser=Parser()
26.  parser.feed(data)
```

Output:

```
Found tag: html
 Attributes: []
Found tag: body
 Attributes: []
Found tag: p
 Attributes: [('class', 'para text'), ('id', '1')]
Found data: This is a
Found tag: b
 Attributes: []
Found data: test
End tag: b
Found tag: p
 Attributes: []
Found unpaired tag: br
 Attributes: []
End tag: body
End tag: html
```

Observation:

1. We have added the `handle_data` method in this program in line 21.
2. We see that even the textual content is being reported now.

17.2.1.3 Handling Special Input

Tags and textual content make up most of the content in HTML. What's remaining are the following:

1. Comments (Example: `<!-- This is a comment -->`)
2. Entities (Examples: ` `, `&#32` and `&#x20`)
3. Declarations and processing instructions (Examples: `<!DOCTYPE HTML>` and `<?php echo ?>`

There are additional methods to handle these and their syntax is shown below:

```
HTMLParser.handle_comment(self, commentText)
HTMLParser.handle_entityref(self, entityName)
HTMLParser.handle_charref(self, entityCode)
HTMLParser.handle_decl(self, declarationText)
HTMLPartser.unknown_decl(self, declarationText)
HTMLParser.handle_pi(self, processingInstruction)
```

17.2.2 Feeding HTML Data

We have already seen how to respond to various parts in an HTML document. We have also seen the `feed()` method that is used to provide the HTML data and has the following syntax:

```
HTMLParser.feed(self, data)
```

This method feeds in the given `data` to the parser and that results in an immediate execution of suitable methods. Data can be "added" to the parser by calling this method several times. Technically, data is buffered internally since this method can be called any time with more content. This means that text content can remain in the buffer, unreported via any of the methods discussed. The buffer is guaranteed to be flushed out when we invoke the `close()` method on the parser object:

```
HTMLParser.close(self)
```

17.3 Working with Beautiful Soup

BeautifulSoup is another HTML/XHTML & XML parser that uses another parser as it's backend and provides many useful easy-to-use functionalities. BeautifulSoup is not in-built in Python and hence has to be manually installed. The next section shows you how to do it. Documentation is available at https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

Here is a list of important capabilities of BeautifulSoup:

1. Can parse HTML, XHTML and XML documents
2. Permits extraction of details in a very simple manner
3. Permits navigation between nodes in the parse tree
4. Permits a powerful search mechanism to directly locate nodes the programmer is interested in
5. Permits writing back into the parse tree (modification and creation of documents)
6. Can convert the tree into a string and can even prettify it!

17.3.1 Installing BeautifulSoup

The simplest way of installing BeautifulSoup is by using PIP (section 15.7) as follows:

```
pip3 install beautifulsoup4
```

While BeautifulSoup uses `lxml` parser or `HTMLParser` as it's default backend engine, it can be configured to work with other parsers as well, in which case those parsers also need to be installed if not already present.

17.3.2 Reading HTML Content

If the HTML content is small (as was demonstrated in earlier sections), we can store that HTML content to be parsed in a string and pass it as an argument to the constructor of BeautifulSoup as illustrated in the syntax below:

```
BeautifulSoup(htmlString [,parser])
```

The `parser` argument can be either "`lxml`" to select the lxml HTML parser or "`html.parser`" to select the `HTMLParser` covered in section 17.2 before. If omitted, it will automatically pick a parser from amongst the available parsers in the system.

Here is a demonstration:

```
>>> from bs4 import BeautifulSoup
>>> data='<html><body><p class="para text" id="1">This is a
<b>test</b><p><br/></body></html>'
>>> soup=BeautifulSoup(data)
/usr/lib/python3.6/site-packages/bs4/__init__.py:181:
UserWarning: No parser was explicitly specified, so I'm using
the best available HTML parser for this system ("lxml"). This
usually isn't a problem, but if you run this code on another
system, or in a different virtual environment, it may use a
different parser and behave differently.

The code that caused this warning is on line 1 of the file
<stdin>. To get rid of this warning, change code that looks
like this:

 BeautifulSoup(YOUR_MARKUP})

to this:

 BeautifulSoup(YOUR_MARKUP, "lxml")

  markup_type=markup_type))
>>> soup
<html><body><p class="para text" id="1">This is a
<b>test</b></p><p><br/></p></body></html>
```

Observation:

1. We imported the `BeautifulSoup` class from the `bs4` module (`bs4` stands for BeautifulSoup 4)
2. We used the same data that was used earlier in section 17.2.1.1.
3. When we constructed the `BeautifulSoup` object, we got a warning because we had not explicitly selected the backend HTML parser. We see from the message that it has selected the `lxml` parser in this case because it was already installed and available. It also displays the message that if we explicitly pass the parser ("`lxml`" or "`html.parser`") then we won't get this message. This is not a grave issue and we can ignore it for the moment.
4. We see that the constructed object, `soup`, does indeed have the HTML content we had passed for parsing.

For larger HTML content, we can store the contents in a file and pass on the file handle to the constructor as shown in the syntax below:

```
BeautifulSoup(fileHandle [,parser])
```

As an example, if we have an HTML file called `test.html` in the current directory, we can load it as follows:

```
>>> from bs4 import BeautifulSoup
>>> f=open("test.html")
>>> soup=BeautifulSoup(f)
```

Observation:

1. We open the file "`test.html`" and pass the file handle to BeautifulSoup.
2. If the file "`test.html`" is in a different directory, we can always specify the pathname of the file.
3. We will see in later sections how to extract various parts of this HTML document.

17.3.3 Writing HTML Content

Once a BeautifulSoup object represents an HTML document, it can be converted to HTML text simply by converting the BeautifulSoup object into a string. This is demonstrated below:

```
>>> from bs4 import BeautifulSoup
>>> data='<html><body><p class="para text" id="1">This is a
<b>test</b><p><br/></body></html>'
>>> soup=BeautifulSoup(data)
>>> str(soup)
'<html><body><p class="para text" id="1">This is a
<b>test</b></p><p><br/></p></body></html>'
```

It is also possible to prettify this output using the `prettify` method as follows:

```
>>> from bs4 import BeautifulSoup
>>> data='<html><body><p class="para text" id="1">This is a
<b>test</b><p><br/></body></html>'
>>> soup=BeautifulSoup(data)
>>> print(soup.prettify())
<html>
 <body>
  <p class="para text" id="1">
   This is a
   <b>
    test
   </b>
  </p>
  <p>
   <br/>
  </p>
 </body>
</html>
```

The `prettify` method formats the HTML content and returns it as a string. The complete syntax is given below:

```
BeautifulSoup.prettify([formatter=formatter])
```

The formatter controls how the HTML is prettified. The default value for `formatter` is "`minimal`".

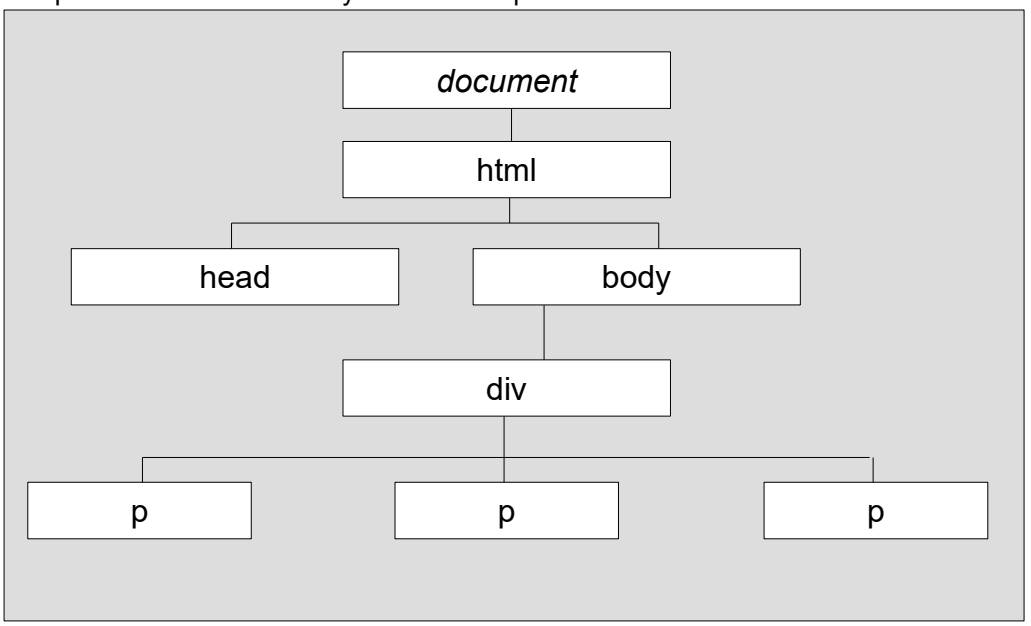
17.3.4 Extracting Details from the Parse Tree

Section 17.3.2 introduced how we can read an HTML string or file into a BeautifulSoup object, but a lot of stuff was hidden in the background! It was not visible to us that BeautifulSoup has, in the background, used another HTML parser to parse through the content and has built a parse tree representing the entire HTML document! This section will show how to explore that parse tree and extract values from it.

In order to demonstrate all these capabilities, we will use a sample HTML content like this:

```
>>> from bs4 import BeautifulSoup
>>> html="""
... <!DOCTYPE html>
... <!-- Test file -->
... <html>
...  <head>
...   <title>HTML Demonstration</title>
...  </head>
...  <body>
...   <div id="main" class="main content">
...    <p>This is <b>Bold</b></p>
...    <p>This is <i>Italics</i></p>
...    <p>This is <u>Underlined</u></p>
...   </div>
...  </body>
... </html>
... """
>>> soup=BeautifulSoup(html)
```

The parse tree constructed by BeautifulSoup will look like this:



Observation:

1. This tree is a collection of nodes. Not all nodes are shown here for readability. We have chosen to merely show all the tag objects in the tree.
2. What is specifically missing in this tree is all the text nodes. Text nodes are not tag objects, but are nodes in the tree nevertheless. Though not shown in the tree, we emphasize that they exist and will show in the following sections how to extract them as well.
3. The root of this hierarchy is the "`document`" object, maintained internally by BeautifulSoup.
4. The single child of the "`document`" object is the `html` tag, which is the root of the HTML content.
5. The `html` tag contains a `head` tag and `body` tag as children.
6. The `body` tag contains a `div` tag as it's child, which in turn contains 3 `p` tags.

All further demonstration code will be continuation of the same session. We know the an HTML page generally contains a `title` tag. Let us extract that tag:

```
(continuation of previous session)
>>> tag=soup.title
>>> tag
<title>HTML Demonstration</title>
>>> type(tag)
<class 'bs4.element.Tag'>
```

Observation:

1. The expression `soup.title`, where `soup` refers to the BeautifulSoup object that has parsed through some HTML content, returns a reference to a `Tag` object that represents the `title` tag in the HTML content.
2. We see that the string representation of a `Tag` object is the tag along with it's content – in this example, "`<title>HTML Demonstration</title>`"
3. We can verify that the type of the `Tag` object is indeed `bs4.element.Tag`.

Given a `Tag` object, as we had extracted above, we can obtain the following information:

1. The name of the tag
2. The attributes of the tag
3. The string content of the tag
4. The child tags within the tag
5. The complete recursive string content within the tag

17.3.4.1 Extracting the Tag Name

The name of the tag can be extracted simply by using the `name` attribute as shown in the example below:

```
(continuation of previous session)
>>> tag.name
'title'
```

17.3.4.2 Extracting the Tag Attributes

All the attributes of a `Tag` are available as a dictionary in the attribute `attrs`. The key in the dictionary is the attribute name and the value is the corresponding value of that attribute. Since the `title` tag did not have any attributes, we will demonstrate using the `div` tag:

```
(continuation of previous session)
>>> tag=soup.div
>>> tag.attrs
{'id': 'main', 'class': ['main', 'content']}
```

Observation:

1. The `div` tag had 2 attributes: `id` and `class`.
2. While the `id` attribute had a single value, the `class` attribute had 2 values. In HTML, multiple values for an attribute are space-separated. BeautifulSoup gives us a list of values instead!
3. Do note however that BeautifulSoup will give us a list of values only if and when it feels that multiple values are supported for that attribute and at least one space is found in the value. If multiple values are not expected for that attribute, BeautifulSoup will retain the space in the value instead and does not return a list!

An easier and more direct way of interacting with known attributes of a tag is by simply using the attribute name as a subscript on the `Tag` object as demonstrated below:

```
(continuation of previous session)
>>> tag['id']
'main'
>>> tag['class']
['main', 'content']
>>> tag['class'][0]
'main'
```

Observation:

1. The above approach of course only works when we know the name of the attribute we are expecting.
2. While `tag['class']` gives us the value of the attribute class in the form of a list, `tag['class'][0]` gives us the first value for that the attribute class.

17.3.4.3 Extracting the String Content of a Tag

To extract the text content within a tag, the `string` attribute can be used as shown below:

```
(continuation of previous session)
>>> tag=soup.title
>>> tag
<title>HTML Demonstration</title>
>>> tag.string
'HTML Demonstration'
>>> type(tag.string)
<class 'bs4.element.NavigableString'>
```

Observation:

1. The `string` attribute provides the text content within a `Tag`. The type of the text content is `NavigableString`.
2. This text content is available only if the `Tag` contains nothing else apart from text! Section 17.3.4.5 shows how to extract text from within the child nodes recursively.
3. The string equivalent of a `NavigableString` object is the text content.

If the given `Tag` contains multiple child text nodes, then we can iterate over all the `NavigableString` nodes of the given `Tag` using the `strings` attribute as follows:

```
(continuation of previous session)
>>> tag=soup.div
>>> for string in tag.strings:
...     print("["+str(string)+"]")
...
[
]
[This is ]
[Bold]
[
]
[This is ]
[Italics]
[
]
[This is ]
[Underlined]
[
]
```

Observation:

1. We are iterating over the text content of the `div Tag`.
2. We see that this iterates not only within the `div Tag`, but also it's children!
3. We display the string equivalent of each child, enclosed in square brackets for readability.
4. We need to explicitly convert each object to a string using the `str()` function since strings cannot be concatenated against other objects. We never found the need to convert these objects to strings earlier as in all those examples, the conversion to string was implicit!

Each newline character after the HTML tag is being considered text content and this makes the output less readable. We can instead choose `stripped_strings` over `strings` to iterate over strings that have these whitespaces stripped:

```
>>> for string in tag.stripped_strings:
...     print(string)
...
This is
Bold
This is
Italics
This is
Underlined
```

Observation:

1. We no longer display the square brackets as there is no need!
2. Since we are anyway not performing string concatenation, we don't need to explicitly convert the object to a string.
3. We observe that we don't obtain the blank lines we had obtained earlier.

Even comments are considered to be (special kind of) strings! They belong to the type `Comment`, which is a subclass of `PreformattedString`, which is a subclass of `NavigableString`, as demonstrated below:

```
>>> tag=BeautifulSoup("<!-- This is a comment -->")
>>> tag.string
' This is a comment '
>>> type(tag.string)
<class 'bs4.element.Comment'>
>>> type(tag.string).__bases__
(<class 'bs4.element.PreformattedString'>,)
>>> type(tag.string).__bases__[0].__bases__
(<class 'bs4.element.NavigableString'>,)
```

Observation:

1. We create a `Comment` tag by parsing through a HTML comment.
2. We observe that the string equivalent of the comment is the comment text.
3. The type of a comment object is `Comment`, which is derived from `PreformattedString`, which is derived from `NavigableString`.

17.3.4.4 Extracting the Child Tags Within a Tag

To extract a specific child tag within a tag, we can simply use the name of the tag as an attribute of the `Tag` object, as shown below:

```
(continuation of previous session)
>>> tag=soup.html.body
>>> tag.name
'body'
```

Observation:

1. We are trying to extract the `body` tag from within the `html` tag.
2. We have used `tag.name` to demonstrate the correctness here.

We need not, strictly speaking, use the proper hierarchy to locate the `body` tag. Thus, the following code will also work:

```
(continuation of previous session)
>>> tag=soup.body
>>> tag.name
'body'
```

Here is another example of extracting a `p` tag:

```
(continuation of previous session)
>>> tag=soup.p
>>> tag
<p>This is <b>Bold</b></p>
```

Observation:

1. These techniques for extracting a child tag work only when there is a single child tag with the given name. If there are multiple tags with the given name, only the first one is returned.
2. Section 17.3.6 will cover how to extract all tags with a given name.

17.3.4.5 Extracting the Recursive String Content Within a Tag

A very powerful capability of BeautifulSoup is the functionality to extract all text within a tag and it's child tags! This is demonstrated below:

```
(continuation of previous session)
>>> tag=soup.div
>>> tag
<div class="main content" id="main">
<p>This is <b>Bold</b></p>
<p>This is <i>Italics</i></p>
<p>This is <u>Underlined</u></p>
</div>
>>> print(tag.get_text())

This is Bold
This is Italics
This is Underlined

>>>
```

Observation:

1. We have extracted the `div` tag.
2. We see that the `div` tag contains 3 `p` tags inside it, which in turn contain text. In addition, there are newlines.
3. The `get_text()` method has concatenated and returned the strings within each `p` tag inside the `div` tag.

17.3.5 Navigating Through the Parse Tree

The previous section showed how we can extract certain `Tag` objects from within the parse tree and then proceed to extract details from within the `Tag` object. This section shows how we can navigate from one `Tag` object to another in the parse tree.

Given that the parse tree is a tree, there are 3 types of navigation possible:

1. Navigation from a `Tag` to it's children
2. Navigation from a `Tag` to it's parent
3. Navigation from a `Tag` to it's siblings

The following sections will cover all these possibilities using the following attributes of the `Tag` class:

```
Tag.contents
Tag.children
Tag.descendants
Tag.parent
Tag.parents
Tag.next_sibling
Tag.previous_sibling
Tag.next_siblings
Tag.previous_siblings
Tag.next_element
Tag.previous_element
Tag.next_elements
Tag.previous_elements
```

17.3.5.1 Navigating to Children

Each tag can have any number of child nodes and they can be accessed as a list using the `contents` attribute as shown below:

```
(continuation of previous session)
>>> tag=soup.div
>>> tag.contents
['\n', <p>This is <b>Bold</b></p>, '\n', <p>This is
<i>Italics</i></p>, '\n', <p>This is <u>Underlined</u></p>,
'\n']
```

Observation:

1. The child nodes of the `div Tag` includes string nodes and other `Tag` nodes.
2. The string nodes merely contain the text that was present within the `Tag` whereas the `Tag` nodes can contain additional content within them.
3. Though the output looks like a list of string objects, in reality it is a list of `NavigableString` and `Tag` objects
4. The order of the elements in the list is the same as the order in which they appear in the HTML content parsed.

If the goal is to iterate over the children, the `children` attribute can be used instead, which will prove more efficient:

```
(continuation of previous session)
>>> tag=soup.div
>>> for child in tag.children:
...     print("["+str(child)+"]")
...
[
]
[<p>This is <b>Bold</b></p>]
[
]
[<p>This is <i>Italics</i></p>]
[
]
[<p>This is <u>Underlined</u></p>]
[
]
```

Observation:

1. We iterate over the children of the given `Tab` object.
2. This code that uses `children` is slightly more efficient than a similar code written using `contents`, but the output is the same. We are avoiding the creation of a list of child elements here.

While the previous example showed how to visit the immediate children using the children attribute, it is also possible to similarly visit all the descendants (children recursively) using the `descendants` attribute as shown below:

```
(continuation of previous session)
>>> tag=soup.div
>>> for child in tag.descendants:
...     print("["+str(child)+"]")
...
[
]
[<p>This is <b>Bold</b></p>]
[This is ]
[<b>Bold</b>]
[Bold]
[
]
[<p>This is <i>Italics</i></p>]
```

```
[This is ]
[<i>Italics</i>]
[Italics]
[
]
[<p>This is <u>Underlined</u></p>]
[This is ]
[<u>Underlined</u>]
[Underlined]
[
]
```

Observation:

1. This not only shows the immediate children, but their children too!
2. Nested tags are shown as single entries.

17.3.5.2 Navigating to Parent

Being a tree structure, each `Tag` object in the parse tree will have exactly one parent, and tracing and ancestry will take us finally to the "`document`" node that has no parent. The immediate parent of a `Tag` can be accessed using the `parent` attribute as shown below:

```
(continuation of previous session)
>>> tag=soup.div
>>> tag.parent.name
'body'
>>> tag.parent.parent.name
'html'
>>> tag.parent.parent.parent.name
'[document]'
```

Observation:

1. The parent of the `div Tag` is the `body Tag`.
2. The parent of the `body Tag` is the `html Tag`.
3. The parent of the `html` Tag is the `document` element.
4. The parent of the `document` element is `None` (not demonstrated here).

It is possible to iterate over the ancestors of a `Tag` object using the `parents` attribute:

```
>>> for parent in tag.parents:
...     print(parent.name)
...
body
html
[document]
```

Observation:

1. We see the ancestors of the `div Tag` in order: `body`, `html` and `document`.

17.3.5.3 Navigating to Siblings

Having seen how to traverse up and down the tree, we will now see how to traverse sideways in the tree to the peer nodes. Nodes sharing a common parent are technically siblings and can be accessed using the `next_sibling` and `previous_sibling` attributes as demonstrated below:

```
(continuation of previous session)
>>> tag=soup.p
>>> tag
<p>This is <b>Bold</b></p>
>>> tag.next_sibling
'\n'
>>> tag.next_sibling.next_sibling
<p>This is <i>Italics</i></p>
>>> tag.previous_sibling
'\n'
>>> tag.previous_sibling.previous_sibling
>>>
```

Observation:

1. We are starting with the first `p` tag within the `div` tag.
2. It's next sibling is a text node with content "`\n`".
3. The sibling after that is the next `p` tag.
4. The previous sibling to the first `p` tag is a text node with content "`\n`" - this is the newline after the `<div>` tag!

Once again, we can choose to iterate over the siblings using `next_siblings` and `previous_siblings`:

```
(continuation of previous session)
>>> tag=soup.p
>>> for sibling in tag.next_siblings:
...     print(sibling)
...


<p>This is <i>Italics</i></p>


<p>This is <u>Underlined</u></p>


>>>
```

Observation:

1. We start with the first `p` tag and iterate over the siblings ahead.
2. This gives is all the text nodes and other `p` tags as well.
3. The blank lines we see is because of the text nodes that contain "`\n`" as their content.

Finally, just as how we accessed the next and previous siblings, we can also access the previous and next elements (nodes) of the parse tree using `next_element` and `previous_element`:

```
(continuation of previous session)
>>> tag=soup.p
>>> tag
<p>This is <b>Bold</b></p>
>>> tag.next_element
'This is '
>>> tag.previous_element
'\n'
```

Observation:

1. The next element in the parse tree need not be the next sibling, as can be seen in the output above.
2. The previous element in the parse tree need not be the previous sibling (but is indeed the previous sibling in the output above).

We can through all the elements following a particular one using `next_elements` till the end of the parse tree (the last leaf node), and can iterate over all previous elements till the document element using `previous_elements`. Here is a

demonstration of `next_elements`:

```
(continuation of previous session)
>>> tag=soup.p
>>> for element in tag.next_elements:
...     print(element)
...
This is
<b>Bold</b>
Bold


<p>This is <i>Italics</i></p>
This is
<i>Italics</i>
Italics


<p>This is <u>Underlined</u></p>
This is
<u>Underlined</u>
Underlined









>>>
```

Observation:

1. We see that we get all the HTML content that appeared after the `p` tag.
2. The blank lines we see are the text content (newlines) that exist after every tag in the HTML content.

Here is a demonstration of `previous_elements`:

```
(continuation of previous session)
>>> tag=soup.p
>>> for element in tag.previous_elements:
...     print(element)
...


<div class="main content" id="main">
<p>This is <b>Bold</b></p>
<p>This is <i>Italics</i></p>
<p>This is <u>Underlined</u></p>
</div>


<body>
<div class="main content" id="main">
<p>This is <b>Bold</b></p>
<p>This is <i>Italics</i></p>
<p>This is <u>Underlined</u></p>
</div>
</body>




HTML Demonstration
<title>HTML Demonstration</title>


<head>
<title>HTML Demonstration</title>
</head>


<html>
<head>
<title>HTML Demonstration</title>
</head>
<body>
<div class="main content" id="main">
<p>This is <b>Bold</b></p>
<p>This is <i>Italics</i></p>
<p>This is <u>Underlined</u></p>
</div>
</body>
```

```
</html>
 Test file
html
```

Observation:

1. We see all the HTML content before the `p` tag, from the `div` tag all the way till the document tag!
2. Note that when a previous element is accessed and printed, it will end up printing all the HTML content contained within it and it's children recursively.

17.3.6 Searching in the Parse Tree

While the previous section showed how to traverse through the parse tree, it is often handy to be able to directly access a particular node within the parse tree. We did see a mechanism to do this in section 17.3.4.4, but that only fetched the first such tag in the parse tree. In this section, we will see how to extract nodes of interest to us directly.

In order to directly obtain all the tags that meet our criteria, we can use the `find_all()` method which has the following syntax:

```
BeautifulSoup.find_all(name, attrs, recursive, string, limit,
**kwargs)
```

NOTE:

All the arguments shown above are optional! We frequently use keyword arguments to specify required arguments.

Instead of invoking this method on the BeautifulSoup object and searching through the entire parse tree, it is also possible to invoke this method on any `Tag` object and search through that sub-tree.

Forms:

```
1.     BeautifulSoup.find_all()
2.     BeautifulSoup.find_all(name)
3.     BeautifulSoup.find_all(name, attrs)
4.     BeautifulSoup.find_all(name, attrs, recursive)
5.     BeautifulSoup.find_all(name, attrs, recursive,
   string)
6.     BeautifulSoup.find_all(name, attrs, recursive,
   string, limit)
7.     BeautifulSoup.find_all(name, attrs, recursive,
   string, limit, **kwargs)
```

Form #1: `BeautifulSoup.find_all()`

This returns a list of all tags in the HTML content parsed, as demonstrated below:

```
(continuation of previous session)
>>> for tag in soup.find_all():
...     print(tag.name)
...
html
head
title
body
div
p
b
p
i
p
u
```

Observation:

1. We are iterating through all the tags of the HTML content and displaying the name of the tag.
2. We observe that we get all the tags in the same order as present in the HTML content.

Form #2: `BeautifulSoup.find_all(name)`

This returns a list of all tags that match the given name, as demonstrated below:

```
(continuation of previous session)
>>> for tag in soup.find_all("p"):
...     print(tag.name)
...
p
p
p
```

Observation:

1. This lists out all the `p` tags in the HTML content.
2. We had 3 such `p` tags and each is listed.

Instead of searching for a single tag with the specified name, we can search for tags that match any of a given list of names as demonstrated below:

```
(continuation of previous session)
>>> for tag in soup.find_all(["p","b","u"]):
...     print(tag.name)
...
p
b
p
p
u
```

Observation:

1. This lists out all `p`, `b` and `u` tags.
2. We observe that this lists out the tags in the same order as they are found in the HTML content.
3. We observe that this does not print the `i` tag as that was not specified in the list of tags.

It is also possible to use a regular expression for specifying the tags to match, as demonstrated below:

```
(continuation of previous session)
>>> import re
>>> for tag in soup.find_all(re.compile("^.$")):
...     print(tag.name)
...
p
b
p
i
p
u
```

Observation:

1. We have imported the `re` module in order to deal with regular expressions.
2. The regex we have given is one that matches a single character – we are thus searching for tags that are single characters.
3. We compile the regular expression and pass the regex object as the first argument to the `find_all()` method.
4. We observe that this lists out all those tags that are made up of single characters!

Finally, it is possible to pass a function object as the first argument, where the function is used for filtering as demonstrated below:

```
(continuation of previous session)
>>> def filter_tag(tag):
...     return tag.name=="b" or tag.name=="u"
...
>>> for tag in soup.find_all(filter_tag):
...     print(tag.name)
...
b
u
```

Observation:

1. We are passing a function name without the parentheses (i.e. a function object) as the first argument to the `find_all()` method.
2. This function (`filter_tag`) will receive each tag found in the HTML content and has to return `True` or `False` for each tag received. If the function returns `True`, the tag will be listed by `find_all()`, else it will be skipped.
3. The filtering condition we have given in the `filter_tag` function is that it will return true only when the given tag is either `b` or `u`.
4. We therefore observe that only `b` and `u` tags are listed.

Form #3: `BeautifulSoup.find_all(name, attrs)`

This returns a list of all tags with the given name (`name`) and the given attributes (`attrs`). The attributes are passed as a dictionary with the attribute name as the key and it's value as the value. This is demonstrated in the examples below:

```
(continuation of previous session)
>>> for tag in soup.find_all(attrs={"id":"main"}):
...     print(tag.name)
...
div
```

Observation:

1. We have passed only the `attrs` argument in the above example and have not passed the `name` argument. From the previous forms, we know that when the name is not provided, this will give a list of all the tags encountered in the HTML content. However, we are filtering by attributes now. Only those tags that have an "`id`" attribute with the value "`main`" will be selected.
2. Verify from the HTML content assigned in section 17.3.4 that the `div` tag is the only one that has an "`id`" attribute with value "`main`", and hence gets processed.

Form #4: `BeautifulSoup.find_all(name, attrs, recursive)`

The `recursive` flag is used to specify whether we want to descend recursively into the child tags or not. The default is `True`.

```
>>> for tag in soup.div.find_all("b"):
...     print(tag.name)
...
b
>>> for tag in soup.div.find_all("b",recursive=False):
...     print(tag.name)
...
>>>
```

Observation:

1. We are trying to find all the "`b`" tags within the "`div`" tag. In the first case, the `div` tag contains a `p` tag that in turn contains a `b` tag, and therefore we see it being listed.
2. In the second case where we pass `recursive=False`, we are confining the search to the immediate children of the `div` tag and hence do not get to see the `b` tag.

Form #5: `BeautifulSoup.find_all(name, attrs, recursive, string)`

The `string` argument can be used to search for nodes that contain specific strings instead of searching for tags, as illustrated below:

```
(continuation of previous session)
>>> for node in soup.find_all(string="Bold"):
...     print(node)
...
Bold
```

Observation:

1. We are searching for any node that contains the text "`Bold`".
2. Note that in previous versions of BeautifulSoup, the `string` argument was called `text`.

Form #6: `BeautifulSoup.find_all(name, attrs, recursive, string, limit)`

By default, the `find_all()` method returns a list of all tags that match the requirement. We can restrict the length of this list by specifying the maximum number of tags required using the `limit` argument. Note that if there are lesser tags than the limit, they are returned nevertheless.

```
(continuation of previous session)
>>> for tag in soup.div.find_all("p"):
...     print(tag)
...
<p>This is <b>Bold</b></p>
<p>This is <i>Italics</i></p>
<p>This is <u>Underlined</u></p>
>>> for tag in soup.div.find_all("p",limit=2):
...     print(tag)
...
<p>This is <b>Bold</b></p>
<p>This is <i>Italics</i></p>
```

Observation:

1. The first example attempts to list out all the `p` tags within the `div` tag – there are 3 of them.
2. In the second example, we use `limit=2` to limit the tag list to maximum 2 entries and observe that only the first 2 `p` tags within the `div` tag are reported.

Form #7: `BeautifulSoup.find_all(name, attrs, recursive, string, limit, **kwargs)`

Finally, if we use any keyword argument that is not one of `name`, `attrs`, `recursive`, `string` and `limit`, it will be assumed to refer to an attribute with a particular value as demonstrated below:

```
(continuation of previous session)
>>> for tag in soup.find_all(id="main"):
...     print(tag.name)
...
div
```

Observation:

1. We are recursively searching for all those tags that contain an attribute `id` with value `main`.
2. The `div` tag is the only one that contains such an attribute and is hence listed.

The `find_all()` method is perhaps the single method we need to perform any search in the parse tree. Despite this, there are other possibilities when it comes to searching. Let us have a quick look at these before we conclude this section.

As a short-cut for the `find_all()` method, we can directly use parentheses on the `BeautifulSoup` document or `Tag` object, as shown below:

```
(continuation of previous session)
>>> soup.find_all("p")
[<p>This is <b>Bold</b></p>, <p>This is <i>Italics</i></p>,
<p>This is <u>Underlined</u></p>]
>>> soup("p")
[<p>This is <b>Bold</b></p>, <p>This is <i>Italics</i></p>,
<p>This is <u>Underlined</u></p>]
```

Observation:

1. We see that the output is the same in both the above examples.
2. Some people find the parentheses syntax simpler as it is briefer; others prefer the `find_all()` method for readability. The choice, of course, is left to you since there is no difference in their behaviour or efficiency.

If we are only interested in searching one tag that matches a particular requirement, we can either use `limit=1` as shown in form #6, or can use the `find()` method instead, which is identical to the `find_all()` method except for these differences:

1. The `find()` method returns a single tag that matched (and the tag would be the first match found)
2. Even when there is a single match, the `find_all()` method returns a list containing that single tag whereas the `find()` method never returns a list.

The syntax of the `find()` method is the same as the `find_all()` method:

```
BeautifulSoup.find_all(name, attrs, recursive, string, limit,
**kwargs)
```

As earlier, all the arguments are optional and the method can also be directly invoked on a `Tag` object.

There are variants of `find()` and `find_all()` that have exactly the same syntax but work on different tags, summarised in the table below:

Search Range	Single Item	All Items
All descendants	`find()`	`find_all()`
All ancestors	`find_parent()`	`find_parents()`
All subsequent siblings	`find_next_sibling()`	`find_next_siblings()`
All previous siblings	`find_previous_sibling()`	`find_previous_siblings()`
All subsequent nodes	`find_next()`	`find_all_next()`
All previous nodes	`find_previous()`	`find_all_previous()`

17.4 Accessing Web Content Over HTTP

The previous sections all concentrated on HTML content that was either hard coded in a Python script or was loaded from a local HTML file. This section will focus on how to access and download HTML content over a HTTP connection to a remote system.

17.4.1 Downloading a HTML File Over a HTTP Connection

The built-in `urllib` package helps in establishing HTTP connections to remote systems and transferring content. The most basic method we need to employ is `urllib.request.urlopen()`, which opens a connection to the given URL and returns a `HTTPResponse` object using which we can later query the response as illustrated below:

```
>>> import urllib.request
>>>
response=urllib.request.urlopen("https://en.wikipedia.org/wiki/
Python_(programming_language)")
>>> type(response)
<class 'http.client.HTTPResponse'>
```

Observation:

1. The `urllib` package comprises of multiple modules like `request`, `error`, `parse` and `robotparser`. We are interested in the `request` module here.
2. We use the `urlopen()` function to establish a connection with the Wikipedia page for Python!
3. We obtain a `HTTPResponse` object in return that is stored in the variable `response` in the example.

This was pretty simple but we still haven't managed to extract any content from the web page. We can read from the response object the same way as how we read from files (covered in section 14.3).

17.4.1.1 Reading a Line at a Time

A very simple of extracting the HTML content line by line is by iterating through the response object as demonstrated below:

```
(continuation of previous session)
>>> for line in response:
...     print(line.decode().strip())
```

Observation:

1. The output of these statements is not shown here as it is large, does not add value and can change with changes to the Wikipedia page! Nevertheless, when you run these statements you will definitely see the HTML page content.
2. We are iterating through each line of the HTML page.
3. Each line is obtained as a `bytes` object. It is better to convert the same into a `str` object using the `decode()` method. The default character set used for this conversion is UTF-8.
4. The `strip()` method removes any excess whitespaces at the beginning and end of each line, giving rise to a cleaner output.
5. If we attempt to repeat this loop once again, we will not get the same output again as all data from the `response` object would have already been read out by this loop.

17.4.1.2 Reading Entire Content at Once

The entire HTML content can be extracted at one go using the `read` method as shown below:

```
>>> import urllib.request
>>>
response=urllib.request.urlopen("https://en.wikipedia.org/wiki/
Python_(programming_language)")
>>> data=response.read()
>>> len(data)
355637
```

Observation:

1. The `read()` method returns the entire HTML data. The output of these statements is not shown here as it is large, does not add value and can change with changes to the Wikipedia page! Nevertheless, when you print the value of data, you will definitely see the HTML page content.
2. We observe that the length of the data loaded is `355637` bytes.

Reading a line at a time might not be very time efficient (though convenient) and reading the entire content at once may not be very memory efficient. There is a third possibility: reading data a block at a time using the `read()` method as shown below:

```
(continuation of previous session)
>>>
response=urllib.request.urlopen("https://en.wikipedia.org/wiki/
Python_(programming_language)")
>>> while True:
...     data=response.read(10000)
...     if not data: break
...     print(data.decode().strip())
...
```

Observation:

1. Again, remember that we need to use `urlopen()` calls as once we have read out the content from the response object, we won't be able to extract it again using the same object.
2. We are reading the content `10000` bytes at a time. This is an arbitrary number that can be controlled by us as required.

Here is a program that accepts a URL and a filename from the user, downloads the HTML content from the URL and saves it to the given file:

htmlDownloader.py

```
1.   #!/usr/bin/python
2.
3.   # Program to download HTML content from a URL
4.
5.   import urllib.request
6.
7.   url = input("Enter URL:")
8.   filename = input("Enter filename:")
9.
10.  try:
11.      response = urllib.request.urlopen(url)
12.      fileHandle = open(filename,"w")
13.      fileHandle.write(response.read().decode().strip())
14.      fileHandler.close()
15.  except Exception as e:
16.      print("Error:",e)
```

Output:

```
Enter
URL:https://en.wikipedia.org/wiki/Python_(programming_language)
Enter filename:Python.wiki
```

Observation:

1. We accept the URL to read from and the filename to write to in lines 7-8.
2. We open the connection to the URL and obtain the response in line 11.
3. We open the output file in write mode in line 12.
4. We read the HTML content, convert it to string from bytes, remove leading and trailing whitespaces and write the content to the file in line 13.
5. We have added exception handling logic to handle any errors during the process.

17.4.2 Parsing Through Remote Content

Combining this with BeautifulSoup covered earlier, here is a program to list out all the hyperlinks found in a given URL (this could be useful as a basic web crawler):

hyperlinkLister.py

```
1.   #!/usr/bin/python
2.
3.   # Program to list all hyperlinks referenced from a URL
4.
5.   import urllib.request
6.   from bs4 import BeautifulSoup
7.
8.   url = input("Enter URL:")
9.
10.  try:
11.      response = urllib.request.urlopen(url)
12.      soup =  BeautifulSoup(response)
13.      hyperlinks = set()
14.
15.      for tag in soup("a"):
16.          if "href" in tag.attrs:
17.              hyperlinks.add(tag["href"])
18.
19.      for hyperlink in hyperlinks:
20.          print(hyperlink)
21.
22.  except Exception as e:
23.      print("Error:",e)
```

Observation:

1. You can enter any valid URL – perhaps you would like to try the URL that we gave as input to the previous program: `https://en.wikipedia.org/wiki/Python_(programming_language)`. We have not produced the output here as it can be too large.
2. We accept the URL from the user in line 8.
3. We obtain the contents from that URL in line 11 and construct a BeautifulSoup object based on that in line 12.
4. Since hyperlinks can be repeated in the HTML content, we construct a set of hyperlinks in line 13. Remember that sets contain unique elements and duplicates are ignored.
5. We iterate through each of the `a` tags in line 15.
6. Not all `a` tags will have the `href` attribute (some might just have the `id` attribute). We therefore check if the given `a` tag contains an `href` attribute in line 16, and if so, we add it to the set of hyperlinks in line 17.
7. We then iterate through the set of hyperlinks in line 19 and print them in line 20.
8. We process all exceptions by printing them in line 23.

17.4.3 Downloading Binary Files

Similarly, here is a program to download an image from a given URL:

imageDownloader.py

```
1.   #!/usr/bin/python
2.
3.   # Program to download an image file from a URL
4.
5.   import urllib.request
6.
7.   url = input("Enter URL:")
8.
9.   try:
10.      response = urllib.request.urlopen(url)
11.
12.      index = url.rfind("/")
13.      filename = url[index+1:]
14.      fileHandle = open(filename, "wb")
15.      fileHandle.write(response.read())
```

```
16.      fileHandle.close()
17.
18.  except Exception as e:
19.      print("Error:",e)
```

Observation:

1. This program asks the user to enter the URL of the image to download. Perhaps you can try with the URL of the Python logo image in Wikipedia's site:
   `https://upload.wikimedia.org/wikipedia/commons/thumb/f/f8/Python_logo_and_wordmark.svg/200px-Python_logo_and_wordmark.svg.png`
2. After taking the input from the user in line 7, we obtain a connection in line 10.
3. We wish to create a local file to store the image. We would prefer to have a file with the same name as the image file being downloaded, but without the leading host and path details. We therefore extract only the part after the last "`/`" character in the URL. This is done in lines 12 and 13.
4. We open the local file in line 14 for writing in binary mode. The binary mode is important here as image data is not textual.
5. We read the image data from the URL HTTP response in line 15 and write it as it is to the local file created.
6. The file is closed in line 16. Exceptions are handled and reported as a text message in line 19.

17.5 Questions

1. Explain how a simple HTML parser can be built in Python using the `HTMLParser` class.
2. List out some of the advantages of using the `BeautifulSoup` library over `HTMLParser`.
3. Explain the services provided by the Tag class of `BeautifulSoup`.
4. Write a short note on navigation through the parse tree using `BeautifulSoup`.
5. Write a short note on searching in the parse tree using `BeautifulSoup`.
6. Explain how HTML files can be downloaded and parsed in Python with an example.
7. Explain how an image can be downloaded over the Internet with an example.

17.6 Exercises

1. Write a Python script to load an HTML file from the filesystem and convert the `BODY` contents into text by discarding all tags.
2. Rewrite the above program to download HTML content from a given URL, printing the textual content in the `BODY`.
3. Write a Python script to extract all important keywords within a HTML document (assuming that such keywords are within `B`, `STRONG`, `I` or `EM` tags).
4. Write a Python script that downloads all images referenced within an HTML page given it's URL.
5. Write a Python script that parses through an HTML file and lists out all the unique CSS `class` references.
6. Write a Python script that parses through an HTML file and identifies the `DIV` Tag that contains the most number of descendent tags.

SUMMARY

- To develop your custom HTML parser, create a class that extends the html.parser.HTMLParser class and override the needed methods as required. The most common methods are handle_starttag(), handle_endtag() and handle_data(). The HTML data is fed in to the parser using the feed() method.
- The bs4.BeautifulSoup class provides a much easier to use parser that internally uses another available HTML parser.
- The prettify() method returns pretty-printed HTML content.
- The Tag object has various attributes of use like name, attrs, string, etc.
- The get_text() method of Tag returns the recursively collected string content of the Tag while the contents attribute contains the list of all child nodes of that Tag.
- The most commonly used searching method of BeautifulSoup is find_all(), which returns a list of tags that matched the required criteria.
- The urllib.request.urlopen() function establishes a connection with the given URL and returns a HTTPResponse object which behaves like a file handle representing the contents of the URL.

18 Parsing XML

In this chapter you will be able to:

- ☑ Parse through XML content from both files and strings
- ☑ Use the BeautifulSoup module as well as the ElementTree module
- ☑ Extract details from an XML parse tree
- ☑ Search for specific XML tags within an XML parse tree

PARSING XML

18.1 Introduction

XML (eXtensible Markup Language) is the mother of many languages (including HTML covered in section 17) used for representation of data. This chapter shows how XML data can be parsed in Python.

Python 3 comes with a built-in XML parser called `ElementTree` to parse XML documents, but BeautifulSoup covered in section 17.3 can also be used if the `lxml` HTML/XML parser is installed. This chapter focusses on how to use the `ElementTree` XML API.

18.2 Reading XML Content

XML content can be read either from a file or from a string. We will examine both approaches.

18.2.1 Reading XML Content From a File

Reading XML content from a file is recommended if we want higher levels of reusability and portability. Recollect the HTML example of section 17.3.4 that was used for demonstration. If that HTML content is stored in a file called `test.html` in the current directory, then we can load it using the statements below:

```
>>> from xml.etree import ElementTree
>>> tree=ElementTree.parse("test.html")
>>> root=tree.getroot()
>>> root
<Element 'html' at 0x7f519ae16e08>
```

Observation:

1. All valid and well-formed HTML documents are also valid XML documents.
2. We import the `ElementTree` class from the `xml.etree` module.
3. The `ElementTree.parse` function can parse through XML content from a file, given the pathname of the file or a file object. It returns an `ElementTree` object that represents the parsed XML document.
4. Note that a well-formed and valid HTML document is a form of XML document! We are merely reusing examples discussed earlier in chapter ***.
5. We extract the root node of this parse tree using the `getroot()` method of

`ElementTree`, which returns the root node of the XML parse tree as an `Element` object.

6. The root node of the XML parse tree is the `html` tag.

18.2.2 Reading XML Content From a String

If the required XML content to be parsed is already available as a string, we can directly parse it as follows:

```
>>> from xml.etree import ElementTree
>>> data="""
... <!DOCTYPE html>
... <!-- Test file -->
... <html>
...  <head>
...   <title>HTML Demonstration</title>
...  </head>
...  <body>
...   <div id="main" class="main content">
...    <p>This is <b>Bold</b></p>
...    <p>This is <i>Italics</i></p>
...    <p>This is <u>Underlined</u></p>
...   </div>
...  </body>
... </html>
... """
>>> root=ElementTree.fromstring(data)
>>> root
<Element 'html' at 0x7f519255ce58>
```

Observation:

1. Instead of using the `parse()` function, we are now using the `fromstring()` function and passing the XML string instead.
2. Once again, remember that HTML content can also be considered to be valid XML content if it is well-formed and valid.
3. Unlike the `parse()` function that returns an `ElementTree` object from which we can extract the root node, the `fromstring()` method directly returns the root node!
4. In this example, once again, the root node is the `html` tag.

18.2.3 Extracting Details from the Parse Tree

Given any `Element` object, as we had extracted above, we can obtain the following information:

1. The name of the tag
2. The attributes of the tag
3. The string content of the tag
4. The child tags within the tag
5. The complete recursive string content within the tag

18.2.3.1 Extracting the Tag Name

The name of the tag can be extracted simply by using the `tag` attribute as shown in the example below:

```
(continuation of previous session)
>>> root.tag
'html'
```

18.2.3.2 Extracting the Child Tags Within a Tag

Refer to the parse tree shown in section 17.3.4. The `ElementTree` is the document element and the root node is the `html` tag. This html tag has 2 children: `head` and `body`. The `body` tag has 1 child – the `div` tag. We can use the subscript operator and access the child nodes as shown below:

```
(continuation of previous session)
>>> root
<Element 'html' at 0x7f519255ce58>
>>> root.tag
'html'
>>> root[1]
<Element 'body' at 0x7f518d0bd098>
>>> root[1][0]
<Element 'div' at 0x7f518d0bd0e8>
```

Observation:

1. The root node is the `html` tag.
2. The expression `root[0]` would have given us the `head` tag. We have used `root[1]` which gave us the `body` tag.
3. The expression `root[1][0]` gives us the first child of `root[1]`, where `root[1]` represents the `body` tag. This first child of the `body` tag is the `div` tag.

Another method to access the immediate children of an `Element` is to iterate over it as shown below:

```
(continuation of previous session)
>>> for child in root[1][0]:
...     print(child.tag)
...
p
p
p
```

Observation:

1. The expression `root[1][0]` gives us an `Element` object that represents the `div` tag.
2. The `div` tag has 3 children and all of them are `p` tags.

18.2.3.3 Extracting the Tag Attributes

All the attributes of an `Element` are available as a dictionary in the attribute `attrib`. The key in the dictionary is the attribute name and the value is the corresponding value of that attribute. Since the `html` tag did not have any attributes, we will demonstrate using the `div` tag:

```
(continuation of previous session)
>>> root[1][0].attrib
{'id': 'main', 'class': 'main content'}
```

Observation:

1. The previous section showed us that `root[1][0]` gives us an `Element` object representing the `div` tag.
2. The `attrib` attribute is a dictionary containing the attributes of the `div` tag and their corresponding values.

We can also access an individual attribute's value from an Element object using it's `get()` method as shown below:

```
(continuation of previous session)
>>> root[1][0].get("id")
'main'
```

Observation:

1. The expression `root[1][0]` gives us an `Element` object representing the `div` tag.
2. The `div` tag has an attribute called `id` whose value is `main`.

18.2.3.4 Extracting the String Content of a Tag

To extract the text content within a tag, the `text` attribute can be used as shown below:

```
(continuation of previous session)
>>> root[0]
<Element 'head' at 0x7f518edc9f98>
>>> root[0][0]
<Element 'title' at 0x7f518d0bd048>
>>> root[0][0].text
'HTML Demonstration'
```

Observation:

1. The expression `root[0]` gives us an `Element` representing the `head` tag.
2. The expression `root[0][0]` gives us an `Element` representing the `title` tag.
3. The `text` attribute of this `Element` object gives us the string content within that tag.

Let's take another example – this time we will extract the text content of the `div` tag as shown below:

```
(continuation of previous session)
>>> root[1][0].text
'\n   '
```

Observation:

1. The expression `root[1][0]` gives us an `Element` object that represents the `div` tag.
2. The text content of the `div` tag is reported as merely newlines and spaces, without considering the text content within its children.

If a tag contains multiple child tags that contain text, we can iterate through all the nested text content as demonstrated below:

```
(continuation of previous session)
>>> for string in root[1][0].itertext():
...     print(string.strip())
...

This is
Bold

This is
Italics

This is
Underlined
```

Observation:

1. The `iternext()` method allows us to iterate through nested text content.
2. The `strip()` method has been used to remove the additional whitespaces (especially the newline characters) in the beginning and end of each of the text content.
3. We see that this lists out even the text content present within the `p` tags (and their child tags) that are present within the `div` tag.

18.2.3.5 Extracting the Recursive String Content Within a Tag

The previous section showed how we can recursively access the string content within a tag and it's children. While there is no separate method available that can do this, we can use a simple `join` call to extract the entire text as a single string as shown below:

```
(continuation of previous session)
>>> "".join(root[1][0].itertext())
'\n   This is Bold\n   This is Italics\n   This is Underlined\n
'
```

Observation:

1. Recall from the previous example that `root[1][0]` represents the `div` tag and `itertext()` allows us to iterate through all it's nested text nodes.
2. The `join()` method will combine all those pieces of text together (using the null string separator, `""`).

18.2.4 Searching in the Parse Tree

Support for navigation in `ElementTree` is not as rich as in BeautifulSoup. Support for searching in the parse tree is also not as powerful as in BeautifulSoup, but is done using the `findall()` and `find()` methods, similar to BeautifulSoup.

18.2.4.1 The findall() Method

The `findall()` method returns a list of all `Element` objects that match the given name and are direct children of the invoking `Element`:

```
(continuation of previous session)
>>> for element in root[1][0].findall("p"):
...     print("".join(element.itertext()))
...
This is Bold
This is Italics
This is Underlined
```

Observation:

1. We are iterating over all `p` tags that are the direct children of the `div` tag.
2. We are combining all text within the `p` tags and printing them using the technique covered in section 18.2.3.5.

NOTE:

The `findall()` method also supports Xpath and namespaces, both of which are outside the scope of this chapter.

18.2.4.2 The find() Method

The `find()` method also searches the same way as the `findall()` method, but with 2 differences:

1. This method returns the first match only.
2. Even if there was a single match, `findall()` returns a list containing that match whereas `find()` returns a single `Element` object.

The same sample code above is repeated here using `find()` instead of `findall()`:

```
(continuation of previous session)
>>> print("".join(root[1][0].find("p").itertext()))
This is Bold
```

Observation:

1. Since we are not expecting a list of `Element` objects, we are not using a loop. We instead directly work with the `Element` object returned.
2. As can be seen from the output, this returns the first `p` tag's contents instead of any or all of the `p` tags within the `div` tag.

18.3 Questions

1. Compare the `ElementTree` module with `BeautifulSoup` for parsing XML documents.
2. Explain with examples how `ElementTree` can be used to parse through XML content:
    1. From a file
    2. From a string
3. Write a short note on extracting the contents of a `Tag` using `ElementTree`.
4. Write a short note on searching within the XML parse tree using `ElementTree`.

18.4 Exercises

1. Write a Python script to verify if the given XML content is actually HTML by maintaining a small list of sample standard HTML tags. Any tag found in the XML content that is not present in the list of standard tags should result in the conclusion that the content is not valid HTML.
2. Write a Python script to convert an XML file into HTML using the following rules:
    1. The `<head>` section of the HTML content has to be created by the script.
    2. Every `<employee>` tag in XML should become `<div class="employee">` in HTML.
    3. Every `<name>` and `<id>` tag within `<employee>` in XML should become `<p class="name">` and `<p class="id">` respectively in HTML.
    4. Any other child tag found within `<employee>` in XML should become a `<p>` tag in HTML.
3. Write a Python script that analyses an XML file and displays the following:
    1. The total number of tags encountered
    2. The most frequently occurring tag
    3. The maximum level of nesting of tags
4. Write a Python script that reads an XML file and lists out the names of tags that contain the string "`Python`" in it's text content.
5. Write a Python script that reads an XML file and lists out all tags that contain the `class` attribute with value "`test`". Note that a single tag can contain multiple values for the `class` attribute. It would be required to check if "`test`" is present as a value for the `class` attribute.

SUMMARY

- While the bs4.BeautifulSoup module can be used to parse XML, it is also possible to parse through XML content using the xml.etree.ElementTree module.
- The Element object has various attributes of use like name, attrib, text, etc.
- The itertext() method of Element allows us to iterate through the string content of all child nodes recursively.
- The subscript operator can be used to access individual child elements of a given Element.
- The most commonly used searching method of ElementTree is findall(), which returns a list of Element objects that matched the given tag name.
- The find() method is similar to the findall() method, but returns only the first match.

19 Parsing JSON

In this chapter you will be able to:

- ☑ Work with JSON using the json module
- ☑ Convert Python objects into their corresponding JSON string representation
- ☑ Parse JSON strings and extract data in the form of Python objects
- ☑ Read from and write to JSON files in Python

Parsing JSON

19.1 Introduction

The **JavaScript Object Notation (JSON)** has become a popular textual representation of data. Like XML, JSON too is textual and structured, but differs from XML by requiring significantly lesser metadata within the document! Originally developed as a textual representation of JavaScript objects that can be transmitted through streams, it is now an effective replacement for XML in many cases!

A JSON document basically comprises of collections and individual values. The supported collections are objects and arrays that map on to Python's `dict` and `list` types respectively. The scalar values could be numbers or strings, mapping to Python's `int`/`float` and `str` types. In addition, JSON also supports Boolean literals (`true` and `false`) and a special `null` literal, which map on to Python's `bool` and `None` respectively.

JSON objects and Python objects can be inter-converted using the built-in `json` module. The interface of this module is very similar to the pickle module covered in section 14.6 & 14.7 where we had used the methods `dump()` and `load()`, but a significant difference is that multiple JSON objects written one after another to a single file will not make it a JSON file! A single JSON file is only supposed to have a single JSON object within it (which in turn can have anything!). Multiple JSON objects written sequentially into a file can have it's applications, but is technically not a single JSON file.

19.2 Converting From Python to JSON

19.2.1 Converting Python Objects into JSON Strings

Before we examine the methods that help convert Python objects into their JSON equivalents, we will first see the JSON equivalents of each type of Python object:

Python Object	JSON Representation
`dict`	Object
`list`, `tuple`	Array
`str`	String
`int`, `float`	Number
`True`, `False`	`true`, `false`

Python Object	*JSON Representation*
None	null

In order to convert an existing Python object into it's equivalent JSON representation, we use the `json.dumps()` method (which stands for "dump string") that has the following syntax:

```
json.dumps(obj)
```

Here is an example to convert various Python objects to JSON :

```
>>> import json
>>> json.dumps({"a":"1","b":"2"})
'{"a": "1", "b": "2"}'
>>> json.dumps([1,2,3,4])
'[1, 2, 3, 4]'
>>> json.dumps("Hello")
'"Hello"'
>>> json.dumps(25)
'25'
>>> json.dumps(True)
'true'
>>> json.dumps(None)
'null'
```

Observation:

1. We have imported the `json` module.
2. We see that the JSON output matches the Python representation pretty well. Dictionaries become objects, lists and tuples become arrays, `int` and `str` objects remain strings and numbers, `True` and `False` become `true` and `false` respectively and `None` becomes `null`.

Here is a more complex example:

```
(continuation of previous session)
>>> json.dumps({"Ram":{"age":25,"hobbies":
["reading","singing"]}, "Sham":{"age":32,"hobbies":
["swimming","singing","dancing"]}})
'{"Ram": {"age": 25, "hobbies": ["reading", "singing"]},
"Sham": {"age": 32, "hobbies": ["swimming", "singing",
"dancing"]}}'
```

Observation:

1. We have a dictionary whose keys are "`Ram`" and "`Sham`".
2. The values of this dictionary are again dictionaries with keys "`age`" and "`hobbies`".
3. The values corresponding to key "`hobbies`" are arrays of strings.

19.2.2 Pretty-Printing JSON

The `json.dumps()` function supports pretty-printing, with the indentation level specified through the `indent` keyword argument as shown below, changing the previous example:

```
(continuation of previous session)
>>> result=json.dumps({"Ram":{"age":25,"hobbies":
["reading","singing"]}, "Sham":{"age":32,"hobbies":
["swimming","singing","dancing"]}},indent=4)
>>> print(result)
{
    "Ram": {
        "age": 25,
        "hobbies": [
            "reading",
            "singing"
        ]
    },
    "Sham": {
        "age": 32,
        "hobbies": [
            "swimming",
            "singing",
            "dancing"
        ]
    }
}
```

Observation:

1. This is the same example as before, but we have used an indentation level of 4 (`indent=4`).
2. This indentation results in `4` spaces per level of indentation.
3. We also see that the string now contains newline characters to support pretty-printing.

19.2.3 Encoding to Files

Just as how the `dumps()` function dumps a JSON representation as a string, the `dump()` function dumps the JSON string representation to a file, identified by a file object. It has the following syntax:

```
json.dump(obj, file)
```

Here is a simple example that writes a list into a file in JSON format:

```
>>> import json
>>> f=open("test.json","w")
>>> json.dump([1,3,5,7],f)
>>> f.close()
```

Observation:

1. We import the `json` module. We open the file "`test.json`" in (textual) write mode and obtain a file handle that is stored in the variable `f`.
2. We dump a list contents to the file represented by the file handle `f`.
3. We close the file to be sure that the data is indeed flushed and written to the file.

The file "`test.json`" will contain the following text:

```
[1, 3, 5, 7]
```

We will see how to read back the JSON content in section 19.3.2.

19.3 Converting From JSON to Python

19.3.1 Converting JSON Strings into Python Objects

Converting from JSON to Python also uses a data type matching that is summarised in the table below:

JSON Representation	*Python Object*
Object	`dict`
Array	`list`
String	`str`

JSON Representation	Python Object
Number (integral)	`int`
Number (non-integral)	`float`
`true`, `false`	`True`, `False`
`null`	`None`

In order to convert a JSON string into it's equivalent Python object representation, we use the `json.loads()` method (which stands for "load string") that has the following syntax:

```
json.loads(jsonString)
```

Here are some examples to convert various JSON strings into Python objects:

```
>>> import json
>>> json.loads('{"a":"1","b":"2"}')
{'a': '1', 'b': '2'}
>>> json.loads('[1,2,3,4]')
[1, 2, 3, 4]
>>> json.loads('"Hello"')
'Hello'
>>> json.loads('25')
25
>>> json.loads('true')
True
>>> json.loads('false')
False
>>> json.loads('null')
>>>
```

Observation:

1. We have imported the `json` module.
2. We see that the Python output matches the JSON representation pretty well. Objects become dictionaries, arrays become lists, integers and strings become `int` and `str` objects, `true` and `false` become `True` and `False` respectively and `null` becomes `None` (which of course, is not visible in the output).

19.3.2 Decoding from Files

Just as how the `loads()` function loads a JSON representation from a string, the `load()` function loads a JSON string representation from a file, identified by a file object. It has the following syntax:

```
json.load(file)
```

Here is a simple example that reads the previously written list from the file `test.json` (in section 19.2.3):

```
>>> import json
>>> f=open("test.json")
>>> obj=json.load(f)
>>> print(obj)
[1, 3, 5, 7]
```

Observation:

1. We import the `json` module. We load the file "`test.json`" for reading and store the file handle in the variable `f`.
2. We load an object from the file identified by the file handle `f`.
3. On printing the object, we see the same list we had stored in the JSON file.
4. Closing the file is not critical as we have opened it in read mode.

19.3.3 Exception Handling

It is possible that the JSON representation we are reading from while attempting to convert it to a Python object is malformed. In such situations, an exception of type `json.decoder.JSONDecodeError` (which is a subclass of `ValueError`) is raised, as demonstrated below:

```
>>> import json
>>> json.loads("[1,2,3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.6/json/__init__.py", line 354, in
loads
    return _default_decoder.decode(s)
  File "/usr/lib64/python3.6/json/decoder.py", line 339, in
decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
  File "/usr/lib64/python3.6/json/decoder.py", line 355, in
```

```
raw_decode
    obj, end = self.scan_once(s, idx)
json.decoder.JSONDecodeError: Expecting ',' delimiter: line 1
column 7 (char 6)
```

Observation:

1. We have imported the `json` module. We are attempting to convert a JSON string into it's Python equivalent.
2. The JSON string is malformed as the open square bracket does not have a matching close square bracket.
3. We see the `JSONDecodeError` exception being raised, which identifies the error and it's location within the string.
4. In programs, we might be interested in processing this exception in the `except` block suitably.

19.4 Questions

1. Explain the following functions of the `json` module in Python:
    1. `dump()`
    2. `dumps()`
2. Explain the following functions of the `json` module in Python:
    1. `load()`
    2. `loads()`

19.5 Exercises

1. Write a Python script to demonstrate the saving and loading of data in JSON format using the `json` module.
2. Write a Python script to load XML data of a known format and write it to a JSON file.
3. Write a Python script to load JSON data of a known format and write it to an XML file.

SUMMARY

- JSON parsing and conversion support is provided by the built-in json module.
- Python objects and JSON objects have a close semblance - they are almost inter-convertible, thanks to the json module!
- A Python object can be converted into a JSON string representation using json.dumps(), or can be stored in a file in JSON format using json.dump().
- A JSON string can be converted into a corresponding Python object using json.loads(), or from a JSON file using json.load().

20 Miscellaneous

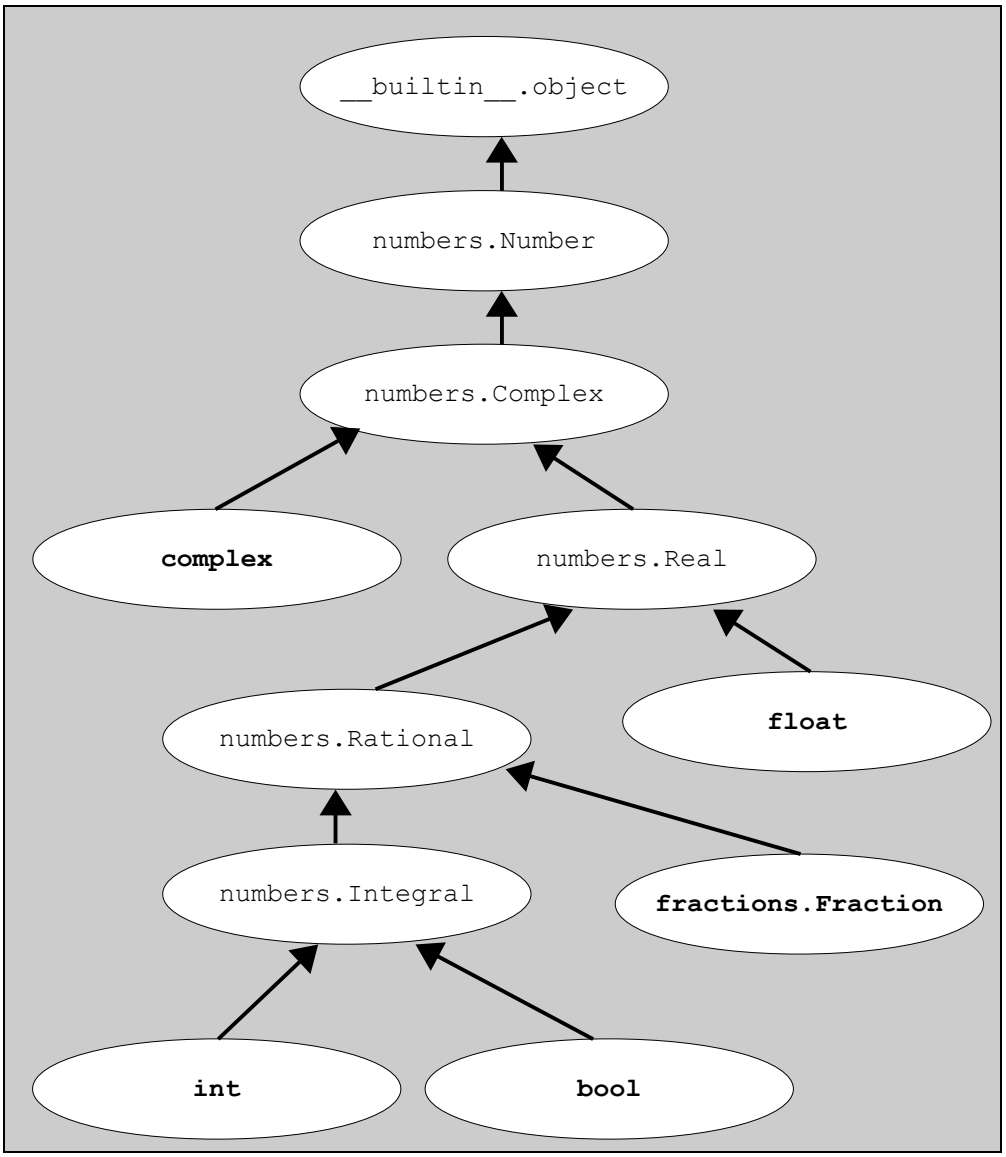
In this chapter you will be able to:

- ☑ Understand the built-in data type hierarchy
- ☑ Convert a data item from one built-in type to another and know how the conversion takes place.
- ☑ Deal with bit operations on data items.
- ☑ Deal with bytes instead of strings.

MISCELLANEOUS

20.1 Data Types Revisited

The diagram below shows the various built-in **numeric data types** available in Python:



We can make out the following from the diagram:

1. This is a **class hierarchy** where each class **derives** features from another.
2. The root of this hierarchy is a class called **`object`**.
3. The classes in bold are **concrete classes** that we can use whereas the rest are merely **abstract base classes** that contain functionality that can be derived by other classes, but can't be directly used.
4. **`numbers`** and **`fractions`** are names of **packages**. **`__builtin__`** is a pseudo-package whose contents are known to the interpreter without having to import any package.

Here are some more observations on the class hierarchy:

1. All objects in Python ultimately are instances of `__builtin__.object`.
2. All numbers in Python are instances of `numbers.Number`
3. The `numbers.Complex` class is an abstract class representing complex numbers and derives from `numbers.Number`
4. If the programmer wants to specifically deal with complex numbers, the `complex` class is a concrete representation of `numbers.Complex`
5. If the programmer does not want to deal specifically with complex numbers, then the programmer wishes to deal with real numbers, represented using `numbers.Real`, which is a subclass of `numbers.Complex` since all real numbers are complex numbers with the imaginary part being 0.
6. Real numbers can be rational or irrational. Rational numbers are those that are representable as `x/y`, where `x` and `y` are integers and `y` is non-zero. Irrational numbers cannot be represented that way, but can be approximated that way. If the programmer does not want to bother about whether the real number is rational or irrational, the `float` class is provided as a subclass of `numbers.Real` to deal with any real number.
7. If the programmer is interested specifically in rational numbers or integers, then the `numbers.Rational` subclass of `numbers.Real` is useful. If the programmer is not particularly interested in integers, then the `fractions.Fraction` subclass of `numbers.Rational` class permits access to the numerator and denominator.
8. As a special case of rational numbers, if the denominator is 1, then mathematically we are dealing with integers! The `numbers.Integral` subclass of `numbers.Rational` describes those numbers that are rational with denominator 1 – integers!
9. While the most common implementation of `numbers.Integral` is the `int` class, `bool` is also considered to be an integral type whose values can only be `0` or `1`, but called `False` and `True` respectively!

20.1.1 Data Type Conversions

20.1.1.1 Conversion to int

To convert an object `x` to it's integer equivalent, use `int(x)`. Here is how it works on primitives:

```
>>> int(4) # int to int does not require any conversion
4
>>> int(4.9) #float is truncated to int
4
>>> int("25") #str is parsed to convert to int
25
>>> int("25.6") #integer parsing in strings does not support float
values!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '25.6'
>>> int(float("25.6")) #This is how to parse a string that contains a
float!
25
>>> int(True) #False=0, True=1
1
>>> int(None)#Null references are not allowed!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string, a bytes-like object or a
number, not 'NoneType'
>>> int(bool(None)) #bool() supports null references, and False=0!
0
```

Objects of type `complex` cannot be directly converted to `int` – their `real` and `imag` parts can be!

If `x` is an instance of a user-defined class, it's `__int__()` method is invoked to determine the integer equivalent as covered in section 12.8.3.3. If this method is not implemented, a `TypeError` occurs!

20.1.1.2 Conversion to float

To convert an object `x` to it's float equivalent, use `float(x)`. Here is how it works on primitives:

```
>>> float(5) #Conversion of int to float
5.0
>>> float("56.5") #Parsing a string to extract a float
56.5
>>> float(False) #False = int(0) = float(0.0)
0.0
>>> float(None) #Null references not supported!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float() argument must be a string or a number, not
'NoneType'
>>> float(bool(None)) #bool() permits null references!
0.0
```

If `x` is an instance of a user-defined class, it's `__float__()` method is invoked to determine the `float` equivalent as covered in section 12.8.3.3. If this method is not implemented, a `TypeError` occurs!

20.1.1.3 Conversion to str

Conversion to string is a very common operation and is implicitly performed in a few places (like `print(x)`). When we explicitly convert an object `x` into a string by using the statement `str(x)`, the following examples show how the conversion takes for primitive types:

```
>>> str(True)
'True'
>>> str(12)
'12'
>>> str(12.5)
'12.5'
>>> str(2+3j)
'(2+3j)'
>>> str(None)
'None'
```

Note that `x` being a null reference is also demonstrated above.

If x is a sequence, set or mapping, the following examples demonstrate the conversion:

```
>>> str([2,3,4])
'[2, 3, 4]'
>>> str((2,3,4))
'(2, 3, 4)'
>>> str({1:2,3:4})
'{1: 2, 3: 4}'
>>> str({1,2,3,4})
'{1, 2, 3, 4}'
```

If x is an instance of a user-defined class, it's magic function `__str__()` is invoked to extract the string representation of the object, as detailed in section 12.8.2.2.

If the class does not implement the `__str__()` method, the `__str__()` methods of superclasses are tested for existence till it finally reaches the class `object` if necessary. The `object` class provides a default implementation of `__str__()` as demonstrated below:

```
>>> class A: pass
...
>>> str(A())
'<__main__.A object at 0x7f8e6bcaf240>'
```

20.1.1.4 Conversion to bool

Conversion to type `bool` is involved when we explicitly attempt to convert an object to type `bool` (like in the call `bool(x)`), or implicitly in a Boolean context (like in the condition of an `if` statement). If the object is already an instance of class `bool`, it's value will definitely be either `True` or `False`. This section concentrates on objects of other types and their conversion to `bool` using the example `bool(x)`:

1. If x is a null reference (`None`), it is evaluated as `False`.
2. If x is a number, then values `0` (integer), `0.0` (float) and `0j` (complex) are considered `False`, all other values are considered `True`.
3. If x is a string, null string (`''`) is considered `False` and any other string is considered `True`.
4. If x is a sequence or mapping, empty sequence/mapping is considered `False` while non-empty sequence/mapping is considered `True`. Thus, null lists (`[]`), null tuples (`()`) and null dictionaries (`{}`) are evaluated `False`.
5. If x is an object of any other class, if that class implements the magic function `__bool__()`, that method's return value is used to determine the Boolean value (the function must return a Boolean value).

6. If the class does not implement `__bool__()` above but implements the magic function `__len__()`, that method's return value is used to determine the Boolean value (the function must return an integer, which if `0` is evaluated as `False` and `True` otherwise).
7. If all the above checks fail, it is evaluated as `True`!

20.2 Dealing with Bits

While section 2.3.5.2 concentrated on Boolean values, this section concentrates on bits!

The two might seem similar since a Boolean value can well be represented by a single bit, but there are 2 core differences:

1. We deal with `1` and `0` in bits instead of `True` and `False`.
2. We deal with sequences of bits that collectively define a single value rather than individual Boolean values that have meaning of their own.

For the sake of providing common, consistent examples for the following sections, we are going to use these 2 statements:

```
>>> x=0b1100
>>> y=0b0110
```

We use these values as they help demonstrate all possible cases in the truth table! Observe the left-most bits of `x` and `y` – they are `1` and `0` respectively. The next bit in each of them is `1` and `1` respectively. The next bit in each of them is `0` and `1` respectively. And finally, the last bit in each of them is `0` and `0` respectively. Thus, we have all the 4 permutations of `0`s and `1`s we want.

Just before we proceed, let us print the decimal equivalent of these:

```
(continuation)
>>> x
12
>>> y
6
```

NOTE:

Conversion from binary to decimal (and vice versa) is beyond the scope of this book.

20.2.1 Bitwise AND

The bitwise AND operator (`&`) works just like the logical AND operator (`and`), except for these 2 differences:

1. They work on corresponding bits of 2 integers.
2. They work on `1`s and `0`s.

Here is an example of performing bitwise AND on the variables `x` and `y` defined earlier:

```
>>> x & y
4
```

Let us convert the result `4` to binary and see the bit pattern:

```
>>> bin(4)
'0b100'
```

NOTE:

The binary value `0b100` is the same as `0b0100`, which we shall use to understand how the operation took place.

The table below summarises how the operation took place. Do note that the operation took place column by column:

x	1	1	0	0
y	0	1	1	0
x & y	**0**	**1**	**0**	**0**

20.2.2 Bitwise OR

The bitwise OR operator (`|`) works very similar to the logical OR operator (`or`), except for these 2 differences:

1. They work on corresponding bits of 2 integers.
2. They work on `1`s and `0`s.

Here is an example of performing bitwise OR on the variables `x` and `y` defined earlier:

```
>>> x | y
14
```

Let us convert the result `14` to binary and see the bit pattern:

```
>>> bin(14)
'0b1110'
```

The table below summarises how the operation took place. Do note that the operation took place column by column:

x	1	1	0	0
y	0	1	1	0
x | y	**1**	**1**	**1**	**0**

20.2.3 Bitwise XOR

The bitwise XOR operator (^) works as per the truth table given alongside. Note that this is very similar to OR, except that when both the conditions are `True`, the result is `False`. In other words, the result is `True` only when only one of the inputs is `True`. Yet another way of phrasing it is that the XOR operator is the inequality operator: the result is `True` only when the inputs are unequal!

A	*B*	*A XOR B*
0	0	0
0	1	1
1	0	1
1	1	0

Here is an example of performing bitwise XOR on the variables `x` and `y` defined earlier:

```
>>> x ^ y
10
```

Let us convert the result `10` to binary and see the bit pattern:

```
>>> bin(10)
'0b1010'
```

The table below summarises how the operation took place. Do note that the operation took place column by column:

x	1	1	0	0
y	0	1	1	0
x ^ y	**1**	**0**	**1**	**0**

20.2.4 Bitwise Complement

Finally, the bitwise complement operator (`~`) is similar in nature to the logical NOT operator (`not`): `0` becomes `1` and vice-versa, as demonstrated in the example below:

```
>>> ~x
-13
```

Here is a tabular representation of the bits:

x	1	1	0	0
~x	**0**	**0**	**1**	**1**

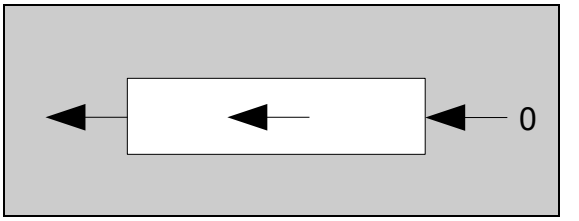
NOTE:

To understand the binary pattern of the result and how the result is `-13`, a knowledge of the 2's complement system is required, which is outside the scope of this book!

20.2.5 Bit Shifting

20.2.5.1 Left Bit Shift

The value of the expression `x << y` is determined by left-shifting all the bits of `x y` times as shown in the diagram:



Each time all bits are shifted left, the leftmost bit is dropped off and a 0 is inserted from the right.

NOTE:

The variable `x` is not affected by this operation unless the expression is `x <<= y`!

Example:

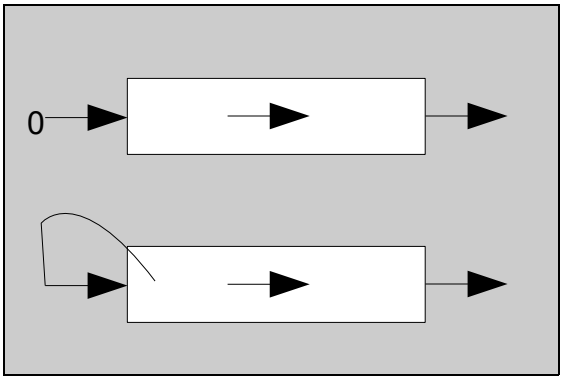
```
>>> x=10
>>> bin(x)
'0b1010'
>>> x<<2
40
>>> bin(40)
'0b101000'
```

Observation:

1. The binary pattern of the integer value `10` is `1010`.
2. When this pattern is shifted left `2` times, with 0 being inserted from the right end, the pattern becomes `101000`

20.2.5.2 Right Bit Shift

The value of `x >> y` is determined by right-shifting all the bits of `x y` times. However, the actual procedure depends on whether the number is positive or negative! The diagram shows both possibilities:



For positive integers, each time all bits are shifted right, the rightmost bit is dropped off and a 0 is inserted from the left.

For negative integers, each time all bits are shifted right, the rightmost bit is dropped off and the leftmost bit is re-inserted as shown in the diagram.

> **NOTE:**
>
> The variable `x` is not affected by this operation unless the expression is `x >>= y`!

Example:

```
>>> x=10
>>> bin(x)
'0b1010'
>>> x>>2
2
>>> bin(2)
'0b10'
```

Observation:

1. The binary pattern of the integer value `10` is `1010`.
2. When this pattern is shifted right `2` times, with 0 being inserted from the left, the pattern becomes `10`.

20.3 Dealing with Bytes

Recollect from section 8 that strings in Python are immutable sequences of Unicode characters. If we are dealing with ASCII characters instead, we could probably same memory and time by having a similar immutable sequence of ASCII characters instead – and that's what the `bytes` class offers!

20.3.1 Literals of Type bytes

Just as how quotes are used to specify string literals, bytes literals can be specified in the same way but with a prefix of '`b`' as shown in the examples below:

```
>>> b'Hello'
b'Hello'
>>> b"Hello"
b'Hello'
>>> b'''Hello'''
b'Hello'
>>> b"""Hello"""
b'Hello'
```

Objects of type `bytes` can also be created using the `bytes()` constructor. The default constructor creates a `bytes` object representing a single null character (ASCII `0`)as a byte:

```
>>> bytes()
b''
```

An integer argument can be provided to specify the length of the `bytes` sequence:

```
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
```

More commonly, a string can be passed as the first argument to construct the equivalent `bytes` sequence, with a second argument specifying the encoding, as shown in the example below:

```
>>> bytes("Hello","utf-8")
b'Hello'
```

Considering that each byte can be specified using 2 hexadecimal digits, it is also possible to build a `bytes` object using the `fromhex` class function, passing a hexadecimal string sequence with 2 hex digits per byte and optional spaces between bytes for readability:

```
>>> bytes.fromhex("10 a96d")
b'\x10\xa9m'
```

Observation:

1. Being a class function, we use the class name (`bytes`) to invoke it.
2. We have specified hex values for 3 bytes – `10`, `a9` and `6d`. These are represented as `\x10`, `\xa9` and `m` respectively. (`6d` is the hex ASCII code of the character `m`!)

20.4 Questions

1. Explain the hierarchy of built-in basic numeric data types with the help of a diagram.
2. Write a short note on conversion to the `bool` type.
3. Explain bitwise operations in Python with examples.
4. Explain the `bytes` class of Python.

20.5 Exercises

1. Write a menu-driven Python program that supports the following operations on a given integer and displays the result after each operation in binary:
    1. Setting a particular bit of an integer
    2. Resetting a particular bit of an integer
    3. Toggling a particular bit of an integer
    4. Extracting a particular bit of an integer

SUMMARY

- All objects are ultimate instances of classes that derive from __builtin__.object. All primitive data types derive from numbers.Number (which is a subclass of __builtin__.object), or more specifically from the numbers.Complex class, of which complex is a direct subclass.
- All non-complex numbers are represented by the numbers.Real class, of which float is a direct subclass.
- All non-float real numbers are represented by the numbers.Rational class, of which fractions.Fraction is a direct subclass.
- All integral classes (int and bool) are subclasses of numbers.Integral, which derives from numbers.Rational.
- The constructors int(), bool(), float(), complex() and str() help manufacture objects of their respective classes, and also help in converting from other types.
- The bitwise operators have been dealt in detail in this section.
- The bytes class helps implement a sequence of bytes and can be used to represent ASCII strings as well.

21 Appendix – Python 2 Vs. Python 3

In this chapter you will be able to:

- ☑ Learn some of the most important differences between Python 2 and Python 3 and know the implications of using a particular version.
- ☑ Understand how to convert Python 2.x code into Python 3.x code or vice versa.
- ☑ Write code that is compatible with either version.

Appendix – Python 2 Vs. Python 3

21.1 Introduction

Though we have stable releases of Python 3.x (3.6) and Python 2.x(2.7) available, the first question that comes to the mind of an individual is “Which version of Python do I use for my day to day activities?”. The short answer is “Python 2 is legacy and therefore Python 3 is the way to go!”. The long answer is “it depends on what your needs are!”. However, if a beginner intends to start learning Python, he or she should consider learning and building on Python 3 and only if required, get gradually acquainted with Python 2 and its differences from Python 3.

Python 3 evolved when the inventor of Python – Guido Van Rossum – decided to clean up Python 2 and in this journey gave very less importance to backward compatibility! It is therefore necessary to choose the right version and stick to it right from the beginning!

It is interesting to note that few of the improvements in Python 3.0 & 3.1 were back-ported to Python 2.6 & 2.7 respectively! Hence, if there is a need to use Python 2, it is recommended to use Python 2.7.x.

This module will first show you some fundamental and obvious differences between Python 2 and Python 3, then will proceed to show you how to convert legacy Python 2 code into Python 3 with least effort, and finally, will show how to write code that runs equally well on both versions, if need be.

21.2 The Fundamental Differences

This section covers the fundamental and obvious changes that got introduced in Python 3.0 from Python 2.5

21.2.1 The print() Function

The `print` statement of Python 2 has been replaced by a `print()` function in Python 3.

Python 2	Python 3
`>>> print "2+3=", 2+3` `2+3=5`	`>>> print("2+3=", 2+3)` `2+3=5`

In Python 2, a trailing comma at the end of a print statement suppresses the newline character at the end. This has been replaced by the `end` keyword argument of the `print()` function in Python 3:

Python 2	*Python 3*

`>>> print 25,`
`25`
`>>>`
	`>>> print(25,end="")`
	`25>>>`

NOTE:

In Python 2, the interpreter will display the prompt in the next line even when the print statement did not print a newline. This newline character will not be printed when executed as a script, however!

To print an empty line, Python 3 uses `print()` whereas Python 2 uses `print`:

Python 2	*Python 3*

`>>> print`
 	
`>>>`
	`>>> print()`
	
	`>>>`

Keyword arguments like `end` were introduced in Python 3. We similarly can now provide a separator between items using the `sep` keyword argument (section 2.7.1.3) as follows:

```
>>> print("I have", 5, "items", sep=":")
I have:5:items
```

21.2.2 Views and Iterators Instead of Lists

Some methods that earlier returned lists/tuples in Python 2 have been replaced by methods that return views and iterators instead. What this means is that instead of getting an entire collection at one go, we instead get an object through which we can iterate and extract a single element at a time. This results in better efficiency. Of course, if required, we can construct a list or tuple out of it and access all the elements at once. This section shows some of the most noticeable changes introduced in Python 3 in this regard.

The `dict` methods `dict.keys()`, `dict.values()` and `dict.items()` (section 7.4) return views instead of lists:

Python 2	Python 3

`>>> d={}`
`>>> type(d.keys())`
`<type 'list'>`
	`>>> d={}`
	`>>> type(d.keys())`
	`<class 'dict_keys'>`

Python 2 had provided `dict.iterkeys()`, `dict.itervalues()` and `dict.iteritems()` methods for the same reason and are no longer supported in Python 3.

The `map()` and `filter()` functions (section 11.3 & 11.4) return iterators in Python 3:

Python 2	Python 3

`>>> L=map(lambda x:x, [])`
`>>> type(L)`
`<type 'list'>`
	`>>> L=map(lambda x:x, [])`
	`>>> type(L)`
	`<class 'map'>`

The `range()` function (section 3.3.5) returns an iterator instead of a list in Python 3:

Python 2	Python 3
`>>> type(range(5))` `<type 'list'>`	`>>> type(range(5))` `<class 'range'>`

Python 2 provided the `xrange()` for the same reason, which no longer exists in Python 3.

21.2.3 Comparisons

The ordering comparison operators (<, <=, >=, >) raise a `TypeError` exception when the operands don't have a meaningful natural ordering.:

Python 2	*Python 3*

`>>> 1<""`
`True`
`>>> 0>None`
`True`
`>>> None<None`
`False`
	`>>> 1<""`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 1, in <module>`
	`TypeError: '<' not supported between instances of 'int' and 'str'`
	`>>> 0>None`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 1, in <module>`
	`TypeError: '>' not supported between instances of 'int' and 'NoneType'`
	`>>> None<None`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 1, in <module>`
	`TypeError: '<' not supported between instances of 'NoneType' and 'NoneType'`

Furthermore, the `cmp()` function (and the underlying `__cmp__()` magic method) has been removed in Python 3:

Python 2	*Python 3*

`>>> cmp(2,3)`
`-1`
	`>>> cmp(2,3)`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 1, in <module>`
	`NameError: name 'cmp' is not defined`

If there is a need to use `cmp()` in Python 3, the expression to use is `(a > b) - (a < b)` instead of `cmp(a, b)`.

21.2.4 Integers

The `long` data type has been renamed to `int` in Python 3. Regardless of the "size" of the integer, the data type in Python 3 is `int`, which can handle arbitrarily large integers (section 2.3.1):

Python 2	Python 3

`>>> type(29482364)`
`<type 'int'>`
`>>>`
`type(2948236423472903749234)`
`<type 'long'>`
	`>>> type(29482364)`
	`<class 'int'>`
	`>>>`
	`type(2948236423472903749234)`
	`<class 'int'>`

The `sys.maxint` constant was removed, since there is no longer a limit to the value of integers.

The `repr()` of a long integer doesn't include the trailing `L` anymore, so code that unconditionally strips that character will chop off the last digit instead. Use `str()` instead to obtain the string representation of an integer in Python 3.

In Python 3, the division operator (`/`) can return a `float`. To guarantee integer division, the `//` operator can be used:

Python 2	Python 3

`>>> 1/2`
`0`
	`>>> 1/2`
	`0.5`
	`>>> 1//2`
	`0`

Octal literals in Python 3 no longer use the prefix "`0`" - "`0o`" needs to be used instead:

Python 2	Python 3

`>>> 077`
`63`
	`>>> 077`
	`  File "<stdin>", line 1`
	`    077`
	`      ^`
	`SyntaxError: invalid token`
	`>>> 0o77`
	`63`

21.2.5 Unicode and ASCII

Python 3 uses the concepts of text and binary data instead of Unicode strings and 8-bit strings. All text is Unicode; however *encoded* Unicode is represented as binary data. The type used to hold text is `str`; while the type used to hold data is `bytes`. The biggest difference with Python 2 is that any attempt to mix text and data in Python 3 raises `TypeError`, whereas if you were to mix Unicode and 8-bit strings in Python 2, it would work fine if the 8-bit string happened to contain only 7-bit ASCII bytes, but you would get `UnicodeDecodeError` if it contained non-ASCII values.

Python 2	*Python 3*

`>>> "abc"+b"def"`
`'abcdef'`
	`>>> "abc"+b"def"`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 1, in <module>`
	`TypeError: must be str, not bytes`

21.2.6 Syntax Changes

Some of the significant syntactic changes are highlighted in this section.

List comprehensions no longer support the syntactic form `[... for var in item1, item2, ...]`. Use `[... for var in (item1, item2, ...)]` instead in Python 3:

Python 2	*Python 3*

`>>> [x for x in 1,2,3]`
`[1, 2, 3]`
	`>>> [x for x in 1,2,3]`
	`  File "<stdin>", line 1`
	`  [x for x in 1,2,3]`
	`  ^`
	`SyntaxError: invalid syntax`
	`>>> [x for x in (1,2,3)]`
	`[1, 2, 3]`

In Python 3, `nonlocal` is a reserved word. Using `nonlocal x` you can now assign directly to a variable in an outer (but non-global) scope.

Tuple parameter unpacking has been removed in Python 3. You can no longer write `def f(a, (b, c)): ....` Use `def f(a, b_c): b, c = b_c` instead:

Python 2	Python 3

`>>> def f(a,(b,c)):`
`...  print "a=",a`
`...  print "b=",b`
`...  print "c=",c`
`...`
`>>> f(1,(2,3))`
`a= 1`
`b= 2`
`c= 3`
	`>>> def f(a,b_c):`
	`...  b,c=b_c`
	`...  print("a=",a)`
	`...  print("b=",b)`
	`...  print("c=",c)`
	`...`
	`>>> f(1,(2,3))`
	`a= 1`
	`b= 2`
	`c= 3`

Python 3 uses `!=` for comparison instead of `<>` in Python 2:

Python 2	Python 3
`>>> 1<>2` `True`	`>>> 1!=2` `True`

Integer literals no longer support a trailing `l` or `L` (section 2.3.1). String literals no longer support a leading `u` or `U` (section 2.3.4).

21.2.7 Exception Handling

Python 3 no longer supports the syntax `raise exceptionClass message`. We need to stick to the constructor syntax of `raise exceptionClass(message)`:

Python 2	Python 3

`>>> raise ValueError, "Invalid Value"`
`Traceback (most recent call last):`
`  File "<stdin>", line 1, in <module>`
`ValueError: Invalid Value`
	`>>> raise ValueError, "Invalid Value"`
	`  File "<stdin>", line 1`
	`  raise ValueError, "Invalid Value"`
	`  ^`
	`SyntaxError: invalid syntax`

`>>> raise ValueError("Invalid Value")` `Traceback (most recent call`	`>>> raise ValueError("Invalid Value")` `Traceback (most recent call`

```
last):
  File "<stdin>", line 1, in
<module>
ValueError: Invalid Value
```

```
last):
  File "<stdin>", line 1, in
<module>
ValueError: Invalid Value
```

Also, Python 3 requires the usage of the `as` keyword in the `except` block:

Python 2	*Python 3*

`>>> try: 2/0`
`... except ZeroDivisionError, e:`
`...  print e`
`...`
`integer division or modulo by zero`
	`>>> try: 2/0`
	`... except ZeroDivisionError as e:`
	`...  print(e)`
	`...`
	`division by zero`

21.2.8 Library Changes

In Python 3, many old modules were removed like `gopherlib` and `md5` (replaced by `hashlib`). The `bsddb3` package was removed because its presence in the core standard library has proved over time to be a particular burden for the core developers.

Some modules were renamed:

Python 2 Module	*Python 3 Module*
`_winreg`	`winreg`
`ConfigParser`	`configparser`
`copy_reg`	`copyreg`
`Queue`	`queue`
`SocketServer`	`socketserver`
`markupbase`	`_markupbase`
`repr`	`reprlib`
`test.test_support`	`test.support`

Some related modules have been grouped into packages and the submodule names have been simplified. The resulting new packages are:

- `dbm` (`anydbm`, `dbhash`, `dbm`, `dumbdbm`, `gdbm`, `whichdb`).
- `html` (`HTMLParser`, `htmlentitydefs`).
- `http` (`httplib`, `BaseHTTPServer`, `CGIHTTPServer`, `SimpleHTTPServer`, `Cookie`, `cookielib`).
- `tkinter` (all `Tkinter` related modules except `turtle`).
- `urllib` (`urllib`, `urllib2`, `urlparse`, `robotparse`).
- `xmlrpc` (`xmlrpclib`, `DocXMLRPCServer`, `SimpleXMLRPCServer`).

21.2.9 Miscellaneous Changes

This section shows some miscellaneous changes brought in by Python 3.

21.2.9.1 Operators and Magic Methods

The `!=` operator now returns the opposite of `==`, unless `==` returns `NotImplemented`.

The concept of “unbound methods” has been removed from the language. When referencing a method as a class attribute, you now get a plain function object.

The `__getslice__()`, `__setslice__()` and `__delslice__()` magic methods have been removed. The syntax `a[i:j]` now translates to `a.__getitem__(slice(i, j))` (or `__setitem__()` or `__delitem__()`, when used as an assignment or deletion target, respectively).

The standard `next()` method has been renamed to `__next__()`.

The `__oct__()` and `__hex__()` special methods are removed – `oct()` and `hex()` use `__index__()` now to convert the argument to an integer.

Support for `__members__` and `__methods__` have been removed.

The function attributes of the form `func_X` have been renamed to the form `__X__`, freeing up these names in the function attribute namespace for user-defined attributes. This is illustrated in the table below:

Python 2 Method	Python 3 Method
`func_closure`	`__closure__`
`func_code`	`__code__`
`func_defaults`	`__defaults__`
`func_dict`	`__dict__`
`func_doc`	`__doc__`
`func_globals`	`__globals__`
`func_name`	`__name__`

The `__nonzero__()` magic method has now been replaced by `__bool__()`.

21.2.9.2 The super() Call

You can now invoke `super()` without arguments and (assuming this is in a regular instance method defined inside a class statement) the right class and instance will automatically be chosen. With arguments, the behaviour of `super()` is unchanged.

21.2.9.3 The input() Function

The `raw_input()` function was renamed as input(). That is, the new `input()` function reads a line from `sys.stdin` and returns it with the trailing newline stripped. It raises `EOFError` if the input is terminated prematurely. To get the old behaviour of `input()`, use `eval(input())`.

21.2.9.4 The round() Function

The `round()` function rounding strategy and return type have changed. Exact halfway cases are now rounded to the nearest even result instead of away from zero. (For example, `round(2.5)` now returns `2` rather than `3`). The call `round(x[, n])` now delegates to `x.__round__([n])` instead of always returning a `float`. It generally returns an `int` when called with a single argument and a value of the same type as `x` when called with two arguments.

21.2.9.5 The reduce() Function

The `reduce()` function has been removed from the global scope and placed in the `functools` module.

21.2.9.6 The reload() Function

The `reload()` function has been removed from the global scope and placed in the `imp` module.

21.2.9.7 The dict.has_key() Method

The `dict.has_key()` method has been removed. Use the `in` operator instead.

21.3 Porting Code From Python 2 to Python 3

In situations where you want to use Python 3 and there is a library in your codebase that is Python 2 compliant only, it is advisable to port that particular library to Python 3 first. The solution adopted here is to run the library against all the unit tests by using the `-3` command line switch in Python 2. If tests fail or emit warnings, modify the code and try again.

Once the code runs without warnings, then try it with Python 3. If the tests still fail in Python 3, then the standard `2to3` utility can come handy here to convert the version to Python 3. In addition to the `2to3` tool that allows code to be generated from Python 2, there's also the `3to2` tool which converts Python 3 code back to Python 2 code!

NOTE:

More information on the `2to3` tool can be found at `https://docs.python.org/2/library/2to3.html`.

More information on the `3to2` tool can be found at `https://pypi.python.org/pypi/3to2`.

21.4 Writing Common Code for Python 2 and Python 3

If there is a need to maintain your code base in Python 2 as well as Python 3, you may come across a few items which have different names in different versions. Instead of changing your code, you may use the `six` compatibility library. It is a single Python file and can be downloaded from `http://pypi.python.org/pypi/six/`. This section shows how we can write code that works across Python 2 and Python 3 using the `six` compatibility module.

The below code snippet illustrates the challenge in using `intern("mysting")` in Python 3 directly as it is not readily available (you'll need to import it from `sys`). Conversely, if we try to import the same in Python 2, we get an `ImportError` exception:

Python 2	*Python 3*

`>>> intern('mystring')`
`'mystring'`
`>>> from sys import intern`
`Traceback (most recent call last):`
`File "<stdin>", line 1, in <module>`
`ImportError: cannot import name intern`
	`>>> intern('mystring')`
	`Traceback (most recent call last):`
	`File "<stdin>", line 1, in <module>`
	`NameError: name 'intern' is not defined`
	`>>> from sys import intern`
	`>>> intern('mystring')`
	`'mystring'`

To make the above code work with both Python 2 and Python 3, we could use the `six` module as follows:

```
>>> from six.moves import intern
>>> intern('mystring')
mystring
```

Another demonstration of the `six` module is with respect to a few constants that come handy as replacements for certain objects (that got deprecated in Python 3):

`six.class_types`

`six.integer_types`

`six.string_types`

`six.text_type`

`six.binary_type`

Here is a demonstration in Python 2:

```
>>> a=u'Hello World'
>>> isinstance(a, basestring)
True
>>> isinstance(a, str)
False
>>> import six
```

```
>>> isinstance(a, six.string_types)
True
```

Observation:

1. In Python 2, a Unicode string is definitely derived from `basestring`, but does not qualify as being a `str` object.
2. A Unicode string in Python 2 qualifies as being `six.string_types` nevertheless.

Here is a similar piece of code in Python 3:

```
>>> a=u'Hello World'
>>> isinstance(a, basestring)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'basestring' is not defined
>>> isinstance(a, str)
True
>>> import six
>>> isinstance(a, six.string_types)
True
```

Observation:

1. In Python 3, all strings are Unicode strings and are thus instances of the `str` class.
2. There is no base class called `basestring` in Python 3.
3. However, Unicode strings in Python 3 still qualify as being instances of `six.string_types`.

NOTE:

More information on the `six` module can be found at `https://pypi.python.org/pypi/six`.

SUMMARY

- Python 2 is the past whereas Python 3 is the present and future!
- Python 3 is not backward compatible with Python 2, and this is the main reason why Python 2 still lives!
- Even basic statements like printing and reading in input have undergone changes from Python 2 to Python 3.
- Python 2 code can be converted to Python 3 code - either manually or by using the 2to3 tool.
- To some extent, Python code common to Python 2 and Python 3 can be written using the six compatibility library.