Learning Python

eBook

[Table of Contents](#_Toc350788838)

[[Chapter 01 – Introduction to Python 5](#_Toc350788838)](#_Toc515037797)

[[About Python 5](#_Toc350788838)](#_Toc515037798)

[[Python and its Origin 5](#_Toc350788838)](#_Toc515037799)

[[Is Python a good language for beginning programmers? 5](#_Toc350788838)](#_Toc515037800)

[[Features of Python 6](#_Toc350788838)](#_Toc515037801)

[[When is Python Used? 7](#_Toc350788838)](#_Toc515037802)

[[History of Python 8](#_Toc350788838)](#_Toc515037803)

[[Chapter 02 – Setting up Python Environment 11](#_Toc350788838)](#_Toc515037804)

[[Installing Python 11](#_Toc350788838)](#_Toc515037805)

[[Installing Python on Windows 11](#_Toc350788838)](#_Toc515037806)

[[Installing Python on Linux 14](#_Toc350788838)](#_Toc515037807)

[[Setting Up System Path 15](#_Toc350788838)](#_Toc515037808)

[[Python Virtual Machine 16](#_Toc350788838)](#_Toc515037809)

[[Executing Steps in Python 18](#_Toc350788838)](#_Toc515037810)

[[Using the Interactive Shell 18](#_Toc350788838)](#_Toc515037811)

[[Using Executable or Script Files 19](#_Toc350788838)](#_Toc515037812)

[[Using User Interface or IDE 19](#_Toc350788838)](#_Toc515037813)

[[Chapter 03 – Getting Started with Python Programming 31](#_Toc350788838)](#_Toc515037814)

[[Basic Syntax 31](#_Toc350788838)](#_Toc515037815)

[[Python Identifier 31](#_Toc350788838)](#_Toc515037816)

[[Python Keywords 31](#_Toc350788838)](#_Toc515037817)

[[Lines and Multi-line Statements 32](#_Toc350788838)](#_Toc515037818)

[[Comments 33](#_Toc350788838)](#_Toc515037819)

[[Blocks 33](#_Toc350788838)](#_Toc515037820)

[[Indentation 34](#_Toc350788838)](#_Toc515037821)

[[Quotations 34](#_Toc350788838)](#_Toc515037822)

[[Expressions 35](#_Toc350788838)](#_Toc515037823)

[[Types of Errors 35](#_Toc350788838)](#_Toc515037824)

[[Operators 37](#_Toc350788838)](#_Toc515037825)

[[Arithmetic Operators 37](#_Toc350788838)](#_Toc515037826)

[[Relational/Conditional Operators 37](#_Toc350788838)](#_Toc515037827)

[[Logical Operators 38](#_Toc350788838)](#_Toc515037828)

[[Bitwise Operators 38](#_Toc350788838)](#_Toc515037829)

[[Assignment Operators 39](#_Toc350788838)](#_Toc515037830)

[[Identity Operators 40](#_Toc350788838)](#_Toc515037831)

[[Membership Operators 40](#_Toc350788838)](#_Toc515037832)

[[Operator Precedence and Associativity 41](#_Toc350788838)](#_Toc515037833)

[[Chapter 04 – Conditions and Loops 42](#_Toc350788838)](#_Toc515037834)

[[Conditions 42](#_Toc350788838)](#_Toc515037835)

[[The Simple if Statement 42](#_Toc350788838)](#_Toc515037836)

[[The if else Statement 42](#_Toc350788838)](#_Toc515037837)

[[The elif Statement 43](#_Toc350788838)](#_Toc515037838)

[[Nested if Statements 43](#_Toc350788838)](#_Toc515037839)

[[Loops 44](#_Toc350788838)](#_Toc515037840)

[[while Loops 44](#_Toc350788838)](#_Toc515037841)

[[for Loops 45](#_Toc350788838)](#_Toc515037842)

[[Nested Loops 47](#_Toc350788838)](#_Toc515037843)

[[Loop Control Statements 48](#_Toc350788838)](#_Toc515037844)

[[Chapter 05 – Data Structures 50](#_Toc350788838)](#_Toc515037845)

[[Introduction 50](#_Toc350788838)](#_Toc515037846)

[[Built-in Functions 51](#_Toc350788838)](#_Toc515037847)

[[Numbers 54](#_Toc350788838)](#_Toc515037848)

[[Types of Numbers 54](#_Toc350788838)](#_Toc515037849)

[[Number Methods 54](#_Toc350788838)](#_Toc515037850)

[[string 55](#_Toc350788838)](#_Toc515037851)

[[string Methods 59](#_Toc350788838)](#_Toc515037852)

[[list 61](#_Toc350788838)](#_Toc515037853)

[[list Methods 64](#_Toc350788838)](#_Toc515037854)

[[tuple 66](#_Toc350788838)](#_Toc515037855)

[[tuple Methods 66](#_Toc350788838)](#_Toc515037856)

[[dictionary 67](#_Toc350788838)](#_Toc515037857)

[[Mapping Types 68](#_Toc350788838)](#_Toc515037858)

[[dictionary View Objects 71](#_Toc350788838)](#_Toc515037859)

[[dictionary Methods 72](#_Toc350788838)](#_Toc515037860)

[[set 73](#_Toc350788838)](#_Toc515037861)

[[Built-in set Types 74](#_Toc350788838)](#_Toc515037862)

[[boolean 77](#_Toc350788838)](#_Toc515037863)

[[Data Type Conversions 78](#_Toc350788838)](#_Toc515037864)

[[Chapter 06 – Functions of Python 79](#_Toc350788838)](#_Toc515037865)

[[What are Functions? 79](#_Toc350788838)](#_Toc515037866)

[[Defining a Function 79](#_Toc350788838)](#_Toc515037867)

[[Calling a Function 80](#_Toc350788838)](#_Toc515037868)

[[Function Scope 81](#_Toc350788838)](#_Toc515037869)

[[Scope of Variables 81](#_Toc350788838)](#_Toc515037870)

[[Global vs. Local variables 81](#_Toc350788838)](#_Toc515037871)

[[Function Arguments 82](#_Toc350788838)](#_Toc515037872)

[[Default Argument Values 82](#_Toc350788838)](#_Toc515037873)

[[Keyword Arguments 83](#_Toc350788838)](#_Toc515037874)

[[Arbitrary Argument Lists 86](#_Toc350788838)](#_Toc515037875)

[[Unpacking Argument Lists 86](#_Toc350788838)](#_Toc515037876)

[[Difference between Arguments and Parameters 87](#_Toc350788838)](#_Toc515037877)

[[Function Objects 88](#_Toc350788838)](#_Toc515037878)

[[The Anonymous Functions 88](#_Toc350788838)](#_Toc515037879)

[[Lambda Functions 88](#_Toc350788838)](#_Toc515037880)

[[Recursive Functions 89](#_Toc350788838)](#_Toc515037881)

[[Chapter 07 – Working with Modules and Packages 91](#_Toc350788838)](#_Toc515037882)

[[Modules 91](#_Toc350788838)](#_Toc515037883)

[[Introduction 91](#_Toc350788838)](#_Toc515037884)

[[Executing Modules as Scripts 92](#_Toc350788838)](#_Toc515037885)

[[Setting Search Path to Access a Module 93](#_Toc350788838)](#_Toc515037886)

[[Importing a Module 93](#_Toc350788838)](#_Toc515037887)

[[Packages 95](#_Toc350788838)](#_Toc515037888)

[[Introduction 95](#_Toc350788838)](#_Toc515037889)

[[Importing \* From a Package 97](#_Toc350788838)](#_Toc515037890)

[[Intra-package References 98](#_Toc350788838)](#_Toc515037891)

[[Packages in Multiple Directories 98](#_Toc350788838)](#_Toc515037892)

[[Chapter 08 – Python File I/O 99](#_Toc350788838)](#_Toc515037893)

[[Introduction 99](#_Toc350788838)](#_Toc515037894)

[[Modes of Files 99](#_Toc350788838)](#_Toc515037895)

[[File Handling 100](#_Toc350788838)](#_Toc515037896)

[[Opening a File 100](#_Toc350788838)](#_Toc515037897)

[[Read from a File 100](#_Toc350788838)](#_Toc515037898)

[[Writing a File 101](#_Toc350788838)](#_Toc515037899)

[[Append to a Module 101](#_Toc350788838)](#_Toc515037900)

[[Closing a Module 102](#_Toc350788838)](#_Toc515037901)

[[OS Module 103](#_Toc350788838)](#_Toc515037902)

[[Other File Tools: shutil 106](#_Toc350788838)](#_Toc515037903)

[[Directory and Files Operations 106](#_Toc350788838)](#_Toc515037904)

[[Archiving Operations 107](#_Toc350788838)](#_Toc515037905)

[[File Operations: Examples 108](#_Toc350788838)](#_Toc515037906)

[[Chapter 09 – Exception Handling in Python 111](#_Toc350788838)](#_Toc515037907)

[[Overview of Exception Handling 111](#_Toc350788838)](#_Toc515037908)

[[About Exceptions 111](#_Toc350788838)](#_Toc515037909)

[[Standard Exceptions 112](#_Toc350788838)](#_Toc515037910)

[[Argument of an Exception 113](#_Toc350788838)](#_Toc515037911)

[[Default Exception Handler 114](#_Toc350788838)](#_Toc515037912)

[[Handling Exceptions 115](#_Toc350788838)](#_Toc515037913)

[[Catching Exceptions 115](#_Toc350788838)](#_Toc515037914)

[[Raising an Exception 119](#_Toc350788838)](#_Toc515037915)

[[User-deﬁned Exception 120](#_Toc350788838)](#_Toc515037916)

[[Chapter 10 – Understanding Object-Oriented Concepts in Python 122](#_Toc350788838)](#_Toc515037917)

[[Object-Oriented Programming 122](#_Toc350788838)](#_Toc515037918)

[[General Introduction 122](#_Toc350788838)](#_Toc515037919)

[[OOP in Python 124](#_Toc350788838)](#_Toc515037920)

[[First-Class Everything 124](#_Toc350788838)](#_Toc515037921)

[[A Minimal Class in Python 125](#_Toc350788838)](#_Toc515037922)

[[Attributes 127](#_Toc350788838)](#_Toc515037923)

[[Methods 130](#_Toc350788838)](#_Toc515037924)

[[The \_\_init\_\_ Method 132](#_Toc350788838)](#_Toc515037925)

[[Data Abstraction, Data Encapsulation, and Information Hiding 134](#_Toc350788838)](#_Toc515037926)

[[Definitions of Terms 134](#_Toc350788838)](#_Toc515037927)

[[\_\_str\_\_ and \_\_repr\_\_ Methods 137](#_Toc350788838)](#_Toc515037928)

[[Public- Protected- and Private Attributes 143](#_Toc350788838)](#_Toc515037929)

[[Destructor 148](#_Toc350788838)](#_Toc515037930)

[[Chapter 11 – Advance Python Concepts 150](#_Toc350788838)](#_Toc515037931)

[[Command Line Arguments 150](#_Toc350788838)](#_Toc515037932)

[[sys.argv 150](#_Toc350788838)](#_Toc515037933)

[[Parsing Command-line Arguments in sys.argv 150](#_Toc350788838)](#_Toc515037934)

[[Working with Iterators, Generators, Closure, Decorators 154](#_Toc350788838)](#_Toc515037935)

[[Iterators 154](#_Toc350788838)](#_Toc515037936)

[[Generators 157](#_Toc350788838)](#_Toc515037937)

[[Closure 159](#_Toc350788838)](#_Toc515037938)

[[Decorators 162](#_Toc350788838)](#_Toc515037939)

[[Chapter 12 – Introduction to Threads in Python 168](#_Toc350788838)](#_Toc515037940)

[[Threading 168](#_Toc350788838)](#_Toc515037941)

[[Thread Objects 168](#_Toc350788838)](#_Toc515037942)

[[Determining the Current Thread 170](#_Toc350788838)](#_Toc515037943)

[[Daemon vs. Non-Daemon Threads 173](#_Toc350788838)](#_Toc515037944)

[[Enumerating all Threads 177](#_Toc350788838)](#_Toc515037945)

[[Subclassing Thread 178](#_Toc350788838)](#_Toc515037946)

[[Timer Threads 180](#_Toc350788838)](#_Toc515037947)

[[Signaling Between Threads 182](#_Toc350788838)](#_Toc515037948)

[[Controlling Access to Resources 184](#_Toc350788838)](#_Toc515037949)

[[Re-entrant Locks 188](#_Toc350788838)](#_Toc515037950)

[[Lock as Context Managers 190](#_Toc350788838)](#_Toc515037951)

[[Synchronizing Threads 191](#_Toc350788838)](#_Toc515037952)

[[Limiting Concurrent Access to Resources 196](#_Toc350788838)](#_Toc515037953)

[[Thread-specific Data 199](#_Toc350788838)](#_Toc515037954)

[[Chapter 13 – Internet Programming 202](#_Toc350788838)](#_Toc515037955)

[[Overview of Python CGI Programming 202](#_Toc350788838)](#_Toc515037956)

[[About Common Gateway Interface (CGI) 202](#_Toc350788838)](#_Toc515037957)

[[Configuring Apache Web Server for CGI 202](#_Toc350788838)](#_Toc515037958)

[[Writing the First CGI Program 204](#_Toc350788838)](#_Toc515037959)

[[CGI Environment Variables 205](#_Toc350788838)](#_Toc515037960)

[[Working with Databases 206](#_Toc350788838)](#_Toc515037961)

[[pymysql 206](#_Toc350788838)](#_Toc515037962)

[[Installing pymysql 206](#_Toc350788838)](#_Toc515037963)

[[Database Operations 206](#_Toc350788838)](#_Toc515037964)

[[Setting Database Connection and Creating Database Table 207](#_Toc350788838)](#_Toc515037965)

[[Disconnecting from Database 209](#_Toc350788838)](#_Toc515037966)

[[Testing the Final Product 209](#_Toc350788838)](#_Toc515037967)

[[Chapter 14 – Network and Graphical User Interface Programming 211](#_Toc350788838)](#_Toc515037968)

[[Socket Programming 211](#_Toc350788838)](#_Toc515037969)

[[Overview of Sockets 211](#_Toc350788838)](#_Toc515037970)

[[The socket Module 211](#_Toc350788838)](#_Toc515037971)

[[The socket Methods 212](#_Toc350788838)](#_Toc515037972)

[[Creating Server Socket and Client Socket 214](#_Toc350788838)](#_Toc515037973)

[[Working with XML Files 215](#_Toc350788838)](#_Toc515037974)

[[About XML 215](#_Toc350788838)](#_Toc515037975)

[[Parsing XML with SAX APIs 215](#_Toc350788838)](#_Toc515037976)

[[Parsing XML with DOM APIs 216](#_Toc350788838)](#_Toc515037977)

[[Parsing XML Document Using Python 218](#_Toc350788838)](#_Toc515037978)

[[Developing GUIs 221](#_Toc350788838)](#_Toc515037979)

[[Overview 221](#_Toc350788838)](#_Toc515037980)

[[Tkinter Programming 221](#_Toc350788838)](#_Toc515037981)

[[Sending Emails in Python (SMTP) 224](#_Toc350788838)](#_Toc515037982)

[[Overview of SMTP 224](#_Toc350788838)](#_Toc515037983)

[[Sending an E-mail using Python 224](#_Toc350788838)](#_Toc515037984)

[[Sending Attachments as an E-mail 225](#_Toc350788838)](#_Toc515037985)

[[Integrating Python with other Language 228](#_Toc350788838)](#_Toc515037986)

[[C/C++ Tools 228](#_Toc350788838)](#_Toc515037987)

[[C/C++ BindingGenerators 229](#_Toc350788838)](#_Toc515037988)

[[Java Integration 229](#_Toc350788838)](#_Toc515037989)

[[Sources 230](#_Toc350788838)](#_Toc515037990)

# Chapter 01 – Introduction to Python

## About Python

### Python and its Origin

Python is a general-purpose, cross-platform, and object-oriented scripting language. Python is a high-level, interpreted language where the translation of the code to machine language happens at runtime.

Python was created in the early 1980s by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands. Guido named it after ‘Monty Python Flying Circus’, a British comedy series, the scripts of which he was reading when he began implementing Python.

Python was released to the public in 1991 and was introduced as a successor of the ABC (All Basic Code) language.

### Is Python a good language for beginning programmers?

While it is very common to start students with procedural and statically-typed languages, Python is one of the most popular languages among beginners. Python strives to provide a simple, yet powerful syntax in contrast to other popular languages such as Java, C, C++, and C#.

There are various reasons for the popularityof Python among beginner programmers.

* It uses simple and consistent English-like syntax that is easy to read and write.
* It has a large standard library of predefined functions and modules that can be used to build realistic projects/applications early on.
* Using the standard library also teaches students about code reuse.
* It works equally well for simple and complex applications.
* It allows you to concentrate on programming skills likes problem decomposition and data type design rather than getting lost in difficult syntaxes.
* It can be extended by adding third-party modules such as PyGame (used for developing games) to help increase students’ reach.

### Features of Python

Let us now take a look at the key features of the Python language:

* **Open source:**Python is an open source software. The source code of Python is available to everyoneat no cost.
* **Object-Oriented:** Python is an object-oriented programming language that includes features such as classes, inheritance, objects, and overloading.
* **Ease of learning:** As discussed earlier,Python has easy-to-learn syntax (more English like syntax) and has a large standard library that makes it a beginner’s language.
* **Interactive:**Python's interactive mode makes it easy to test short snippets of code. There's also a bundled development environment called IDLE.
* **Extendable:** Python is often referred to as a ‘glue’ language. It is easily extended by adding new modules implemented in a compiled language such as C or C++.
* **Standard Libraries:** Python comes with a large standard library that supports many common programming tasks such as connecting to web servers, searching text with regular expressions, reading and modifying files.
* **Support**: You can get continuous support from the online Python community in case you are stuck anywhere while developing your code.
* **High-Level Language:** Python isahigh-level language, which means it’s closer to human language and is usually independent of computer hardware.
* **Portable:** Python is aportable language, which means that it can run on most of the platforms without making a lot of changes to the source code. Python can be used on Linux, Windows, Macintosh, Solaris, OS/2and many such platforms.

### When is Python Used?

Python is a commonly-used language for scripting in the following cases:

* System utilities (system admin tools, command line programs).
* Web development
* Graphical User Interfaces (Tkinter, gtk, Qt).
* Internet scripting
* Embedded scripting
* Database access and programming
* Game programming
* Rapid prototyping and development
* Distributed programming

Let’s take a deeper look at role of python in the following domains:

**Python on the Desktop**: You can write desktop applications in Python using frameworks such as WxPython, Tkinter, or PyGTK. However, most desktop applications are still written in compiled languages such as C, C++ or C#. The frameworks for these languages tend to have more sophisticated development tools and the resulting programs are often easier to distribute, as they do not require the user to have Python installed.

Python has good graphical development tools, including Wing IDE and the Eclipse PyDev extensions. However, most Python developers work "Unix style" with standalone text editors and terminals. On platforms like Java or .Net, environments such as Microsoft's Visual Studio will always offer tighter integration with the programming language.

**Python for the Web:** In fact, several successful rapid Web application frameworks are available for Python, each with its own slant. Many also share components. The most popular include Django, Pylons, TurboGears, CherryPy, Zope and Grok (which is based on Zope).

These frameworks are all suitable for serious applications. For example,

* Zope, was a pioneering open-source application server that helped prove Python's viability in the enterprise (although many Python developers these days feel it is a little "unPythonic").
* Plone, a popular open-source content management system—to which the author is a contributor—runs on Zope and has been implemented in organizations such as Novell and Oxfam.
* The high-traffic Reddit.com runs Pylons.
* The Revver.com video sharing site uses Django.

Deploying a Python Web application is usually straightforward, although not quite as easy as deploying a PHP application in Apache. Database connectivity is very well catered to by object/relational mappers such as SQLAlchemy. However, most Python Web frameworks have yet to catch up to enterprise-grade application servers for Java or .Net in terms of support for high-availability clustering, failover and server management.

The main disadvantage of using Python in an enterprise setting is that Python programmers can be harder to find than, say, Java developers. Python is easy to pick up for an experienced programmer, but the plethora of books, training courses and certifications in the Java world cannot be matched by Python.

Furthermore, the power and expressivity that Python offers means that it may require more skilled developers. Java or C# are more restrictive by design, forcing programmers to adhere to stricter rules around type safety and interface compliance. For some, that hinders productivity. For others, it reduces mistakes or accidents of design.

Finally, application integration concerns may dictate a certain language or platform. However, in today's service-oriented, heterogeneous systems landscape, it is entirely possible to—for example—write a Web service in Python that plugs into a Java service bus and is ultimately consumed by a Visual Basic program.**Python for Data Science**

Python is a popular scientific language and a rising star for machine learning. It is often compared with R andthe matrix handling in NumPy may challenge MATLAB and communication tools like IPython are very attractive and a step into the future of reproducibility.

The SciPy stack for machine learning and data analysis can be used for one-off projects, and frameworks like scikit-learn are mature enough to be used in production systems.

**Python for Artificial Intelligence (AI)**

Python is a language with the best compilation of Lisp and Java both (these languages are used for AI as well).According to Peter Norvig (*an American computer scientist and fellow and councilor of the Association for the Advancement of Artificial Intelligence*) is his text comparing Lisp to Python, these two languages are very similar to each other with some minor differences. There also exists JPython, giving access to the Java GUIs. This is the reason behind Peter Norvig choosing JPython to translate his programs from his AI book. As JPython allowed him to have portable GUI demos, and portable http/ftp/html libraries. Therefore, it is very good to use as AI language.

Benefits of Using Python over the Other Programming Languages for AI:

* Good quality documentation.
* Platform agnostic, and present in virtually every \*nix distribution.
* Easy and fast to learn in comparison to any other OOP language.
* Python has many image intensive libraries like Python Imaging Library, VTK and Maya 3D Visualization Toolkits, Numeric Python, Scientific Python and many other tools available for numeric and scientific applications.
* Python is very well designed, fast, robust, portable, and scalable. These are evidently the most important factors for AI applications.
* Useful for a really broad range of programming tasks from little shell scripts to enterprise web applications to scientific uses.
* Last but not the least, it is Open Source and has good community support available.

### History of Python

Python was released to the public in 1991 and was introduced as a successor of the ABC (All Basic Code) language.

Since 1991, several versions of Python have been released. The current stable release is version 3.2 and the most active release in the market is 2.x.

Python is a very stable language. Since 1991, Python has been released at regular intervals of about 6 to 18 months. The time difference between major releases is usually 18 months.

You can download the current version of Python from the [Python Download page](https://www.python.org/downloads/). On the download page, you will find production ready versions of Python 2.x and 3.x. While Python 3 represents the future of the language, availability of third party software for Python 2 is higher than that for Python 3.

***Note:*** *Usually, the code in Python 2 code must be changed to be able to run on Python 3.*

Let us take a look at some of the differences between Python 2.x and Python 3.x.

|  |  |  |  |
| --- | --- | --- | --- |
| Points of Difference | Explanation | Python 2  (Example) | Python 3  (Example) |
| Print Statement | In Python 3, the print statement has been replaced with a print() function. | >>> print "Hello World"  Hello World | >>> print("Hello World")  Hello World |
| Unicode | In Python 2, astr() type is used for text and bytes/binary values. To save a string value as Unicode, you need to indicate the same.  In Python 3, the str() type stores Unicode values, by default. | >>> print "Python" + b" Unicode"  Python Unicode  >>> | >>> print(" Python", type(b" Unicode"))  Python <class 'bytes'>  >>> |
| Data input | Python 2: raw\_input([prompt]) - If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised.  Python 3 : input([prompt]) - If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised. | >>>data\_input2 = raw\_input() | >>>data\_input3 = input() |
| Integer Division | In Python 2, the result of any division was rounded off to the nearest integer; whereas, in Python 3, the exact value of the division is shown as the output. | >>3/2  1 | >>>3/2  1.5 |
| xrange() Function Removed | In Python 2, the range() function is used to represent a list and the xrange() function is used to display an xrange()object.  In Python 3, the xrange() function is removed and the range() function will behave like the xrange() function of Python 2. | >>>for i in xrange(3):  ... print(i)  ...  0  1  2 | >>>for i in range(3):  ... print(i)  ...  0  1  2 |
| Raise Exception | In Python 2, exception arguments are accepted with or without parenthesis; whereas, in Python 3, a syntax error is raised if the exception arguments are not enclosed in parenthesis. | raise IOError, "file error" #This is accepted in Python 2  raise IOError("file error") #This is also accepted in Python 2 | raise IOError, "file error" #syntax error is raised in Python 3 |
| Arguments in Exceptions | In Python 3, you need to use the ‘as’ keyword to declare arguments to exceptions. | except testerror, e:  print “exception is”, e | except testerror as e:  print(“exception is”, e) |
| next() Function and .next() Method | In Python 2, next() as a method of generator object, is allowed. In Python 2, the next() function, to iterate over generator object, is also accepted.  In Python 3, however, next() as a generator method is discontinued and raises AttributeError. | gen = (letter for letter in 'Hello World') # creates generator object  next(my\_generator)  #allowed in Python 2 and Python 3 | my\_generator.next() #allowed in Python 2. raises AttributeError in Python 3 |
| 2to3 Utility | This utility makes it easy to convert a Python 2.x code to Python 3.x. It reads Python 2.x source code and applies a series of fixers to transform it into a valid Python 3.x code. | | |

# Chapter 02 – Setting up Python Environment

## Installing Python

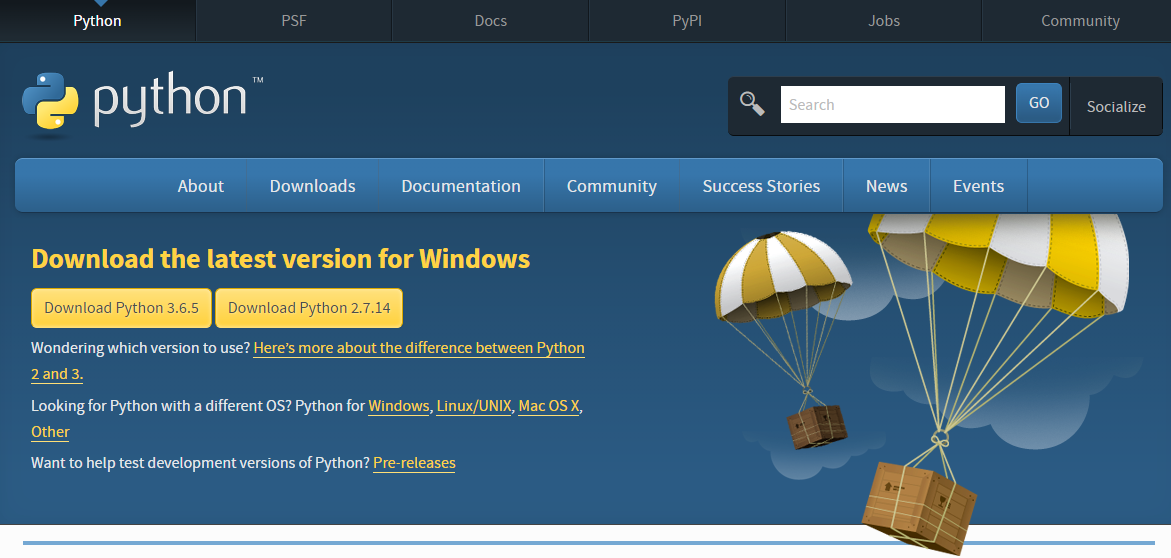
Python is a portable or cross-platform language, which means that it can run on multiple platforms, including Windows, Macintosh, UNIX variants, MS-DOS, and OS/2. Let us see the steps to install Python on Windows and Macintosh systems.

### Installing Python on Windows

Here are the steps to install Python on Windows machine.

1. Open a Web browser and go to <https://www.python.org/downloads/>.

The Downloads page of the Python Web site opens.

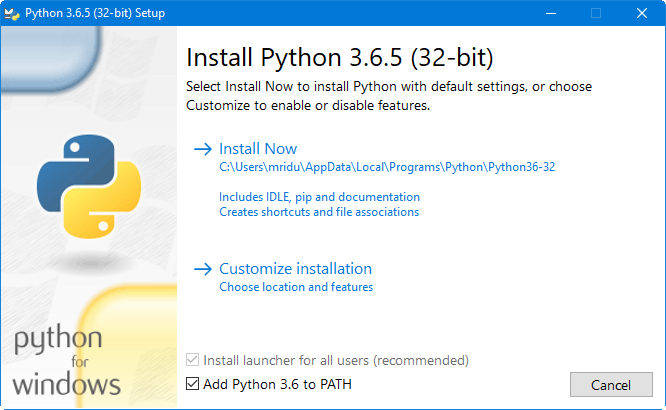


1. Click the **Download Python** button for the version that you want to install.

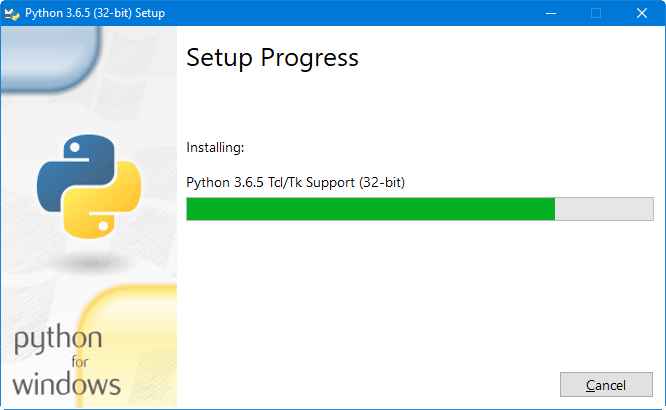
**Note**: A Windows system must support Microsoft installer 2.0 to run Python installer.

1. Run the Python installer (python-xyz.exe) and click **Install Now.**

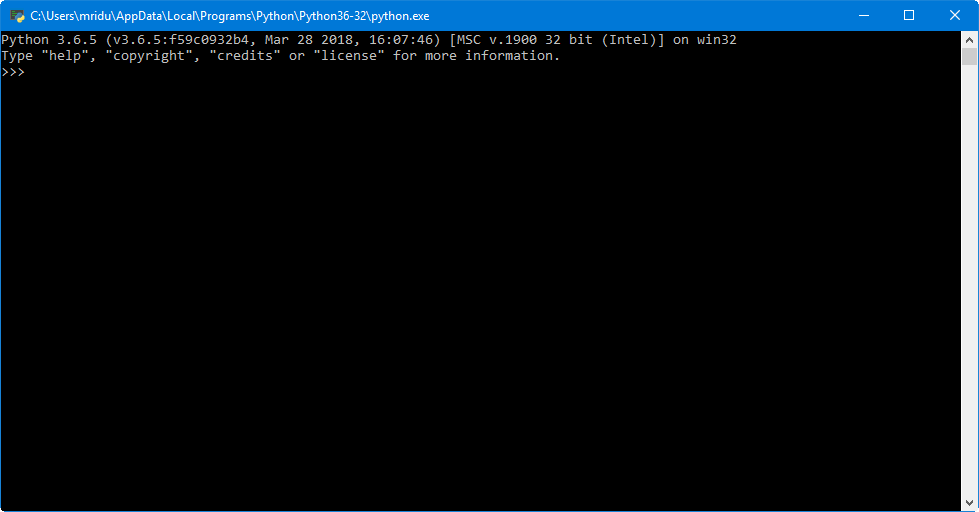
**Note**: *You can also check the* ***Add Python 3.6 to PATH*** *option to add the path at the time of installation only.*



1. To start with, follow the steps in the installation wizard and accept the default settings to complete the installation.



1. Navigate to C:\Users\<your username>\AppData\Local\Programs\Python\Python36-32 on your Windows system.
2. Double-click **python.exe** and the following window will appear.



### Installing Python onLinux

#### Prerequisites

To follow these installation steps, you will need:

* One Ubuntu 14.04 Droplet.
* A sudo non-root user, which you can set up by following[the steps listed below](https://www.digitalocean.com/community/tutorials/how-to-add-and-delete-users-on-an-ubuntu-14-04-vps).

#### Making Python 3 the Default

In this step, we will set Python 3 as the default for ourpythoncommand.

First, check your current Python version.

python --version

On a fresh Ubuntu 14.04 server, this will output:

Python 2.7.6

We would like to havepythonrun Python 3. So first, let's remove the old 2.7 binary.

sudo rm /usr/bin/python

Next, create a symbolic link to the Python 3 binary in its place.

sudo ln -s /usr/bin/python3 /usr/bin/python

If you run python –versionagain, you will now see Python 3.4.0.

### Setting Up System Path

To run Python conveniently from a command prompt, you might consider changing some default environment variables in Windows. While the installer provides an option to configure the PATH and PATHEXT variables for you, this is only reliable for a single, system-wide installation. If you regularly use multiple versions of Python, consider using the Python Launcher for Windows.

To add the Python directory to the path for a particular session in Windows, at the command prompt, type:

path %path%;C:\Python

and press **Enter**.

**Note:***Here, C:\Python is the path of the Python directory.*

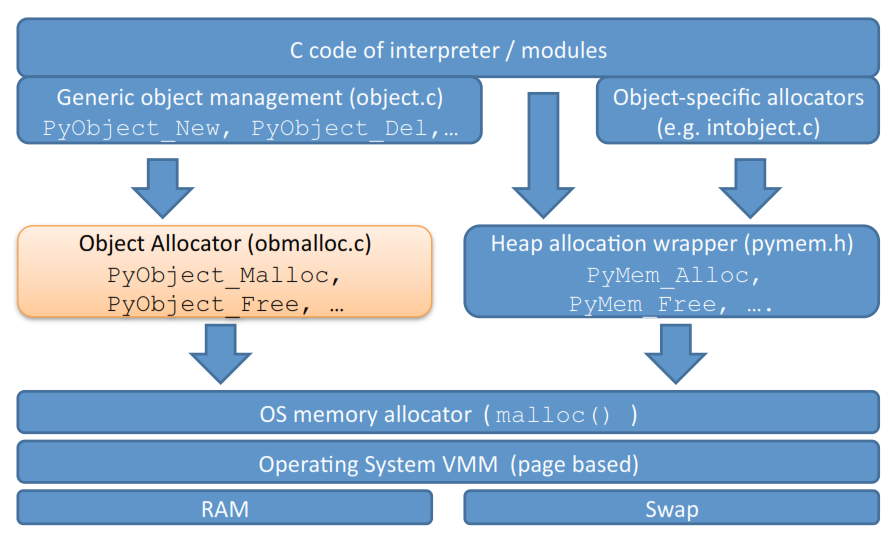
Here are important environment variables, which are recognized by Python:

|  |  |
| --- | --- |
| Variable | Description |
| PYTHONPATH | It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer. |
| PYTHONSTARTUP | It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH. |
| PYTHONCASEOK | It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it. |
| PYTHONHOME | It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy |

### Python Virtual Machine

* Virtual machine part of interpreter executes Python byte code
  + Simple stack machine
  + PyObject\* stack – byte codes operate on objects
    - Stack frames are allocated on the heap
  + C stack frames point to heap stack frames
* Some high-level byte codes (“PRINT”)
* VM knows nearly nothing about C representation of specific Python types
* Python objects know nothing about the VM.

#### Memory Management



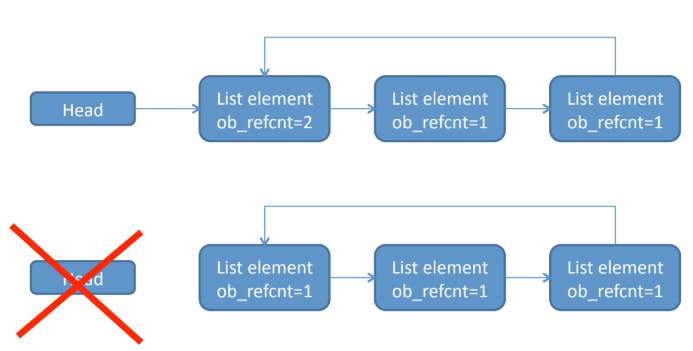
#### Object Allocator

* By design, many very small allocation requests
  + Everything is an object!
* Special optimization for performance (obmalloc.c)
  + Requests >256 bytes handled by malloc
  + Smaller requests sizes are grouped (8 bytes apart)
    - Memory pools of 4k length each (VMM page size), with own free list
    - Pools are used by different request size allocators
    - 8 byte alignment of returned address

#### Garbage Collector

* Traditional garbage collection (e.g. mark and sweep) would demand a set of root objects
  + Extension modules can create own Python objects
  + GC for allocated C objects not really portable
  + Traversing all objects is expensive
* Instead: Simple reference counting
  + In ob->\_refcnt from PyObject\_HEAD
  + Works with every malloc()/free()
* PY\_DECREF() – (object.h)
  + Call finalizer when reference count comes to zero
* Functions that create an object set the on\_refcnt to 1, and store it or destroy it by calling PY\_DECREF
  + Some store functions therefore don’t increase the reference counter (e,g, PyList\_SetItem())
* Objects can be stucked u tracebacks
* Weakref module (PEP 205)
  + For object caches (weak dictionaries)
  + For circular references (DOM node relations)

#### Circular References



#### Cyclic Garbage Collector

* Reference cycle: Unused object(s) even though reference counter is not zero
  + Test is only relevant for container types
* Usage of double-linked list of all container objects (gc\_next,gc\_pref)
  + Determine all containers which are only referenced by themselves
* Objects in cycles with finalizers \_\_del()\_\_ are added to set of uncollectable objects
  + Order of finalizer call in the cycle unclear

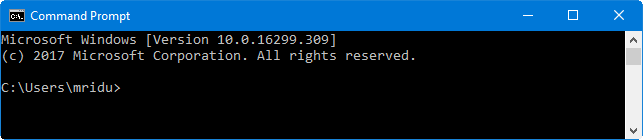
## Executing Steps in Python

There are different ways in which you can execute steps in Python. Let us take a look at each of the ways.

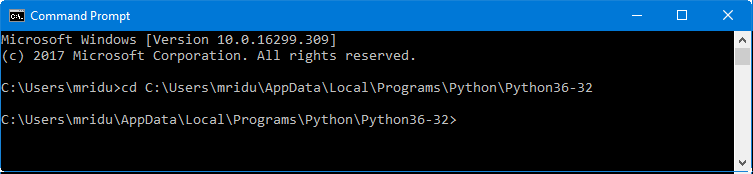
### Using the Interactive Shell

To invoke the Python interpreter on your Windows system:

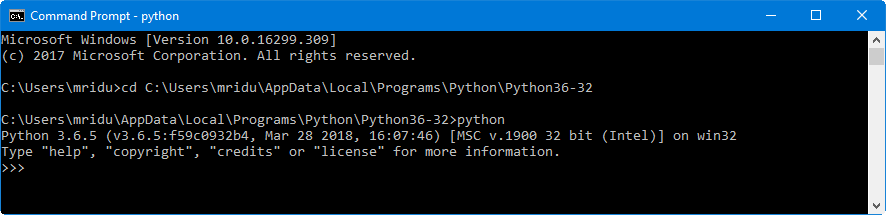
1. Open the command line.



1. Use the cd command to change the directory to the folder where Python is installed.



1. Type python in the command line and press Enter.



This will open the interactive shell. You can then type your scripts at the Python prompt >>> and execute them.

### 

### Using Executable or Script Files

You can also use any text editing software to write a Python script in a file and then save the file with the .py extension. You can then run the .py file in the Python interactive shell.

To run the file in the interactive shell, type the .py file name and press Enter.

### Using User Interface or IDE

Integrated Development Environment (IDE) is a software that makes life easier for a programmer by providing features like code editing, code hints, indentation, and syntax highlighting. An IDE not only provides text editing capabilities but has integrated compiler and debugger that allows you to compile a code on-the-fly.

Here are some of the preferred IDEs for Python:

* **Anaconda**
* **Spyder**
* **Jupyter**
* **Sublime Text**

Let’s take a look at these IDEs in more details.

#### **Anaconda**

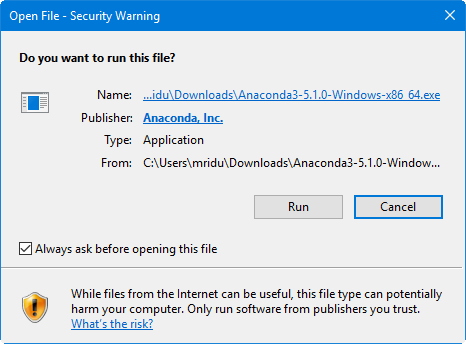
Anaconda Distribution 5 is a free, easy-to-install package manager, environment manager, and Python distribution with a collection of 1,000+ open source packages with free community support. Anaconda is platform-agnostic, so you can use it whether you are on Windows, macOS or Linux.

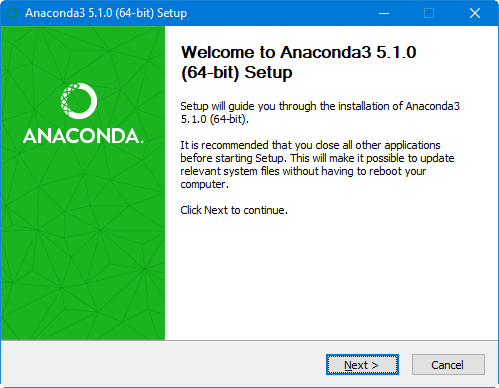
System requirements

* **License**: Free use and redistribution under the terms of the Anaconda End User License Agreement.
* **Operating system**: Windows Vista or newer, 64-bit macOS 10.10+, or Linux, including Ubuntu, RedHat, CentOS 6+, and others.
* Windows XP supported on Anaconda versions 2.2 and earlier. See Old package lists. Download it from our archive.
* **System architecture**: 64-bit x86, 32-bit x86 with Windows or Linux, Power8 or Power9.
* Minimum **3 GB** disk space to download and install.

Installing Anaconda on Windows

1. Download the Anaconda Installer.
2. Double click the installer to launch.

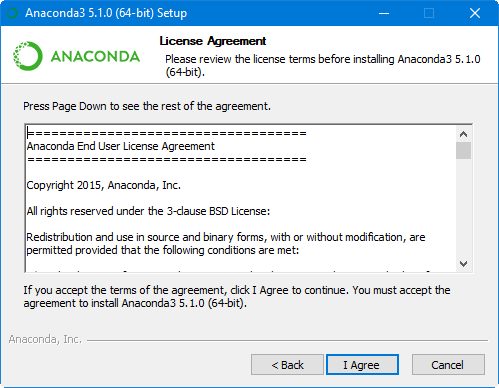




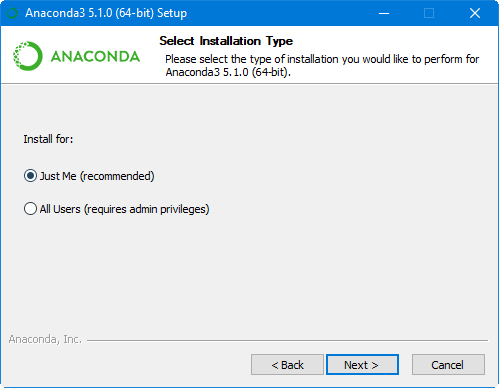
**NOTE**: *To prevent permission errors, do not launch the installer from the Favorites folder.*

**NOTE**: *If you encounter issues during installation, temporarily disable your anti-virus software during install, then re-enable it after the installation concludes. If you installed for all users, uninstall Anaconda and re-install it for your user only and try again.*

1. Click **Next**.
2. Read the licensing terms and click **“I Agree”**.



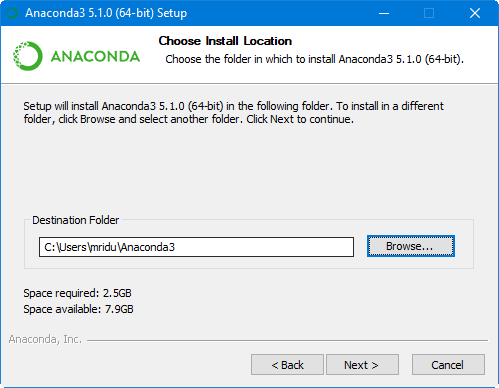
1. Select an install for **“Just Me”** unless you’re installing for all users (which requires Windows Administrator privileges) and click **Next**.



1. Select a destination folder to install Anaconda and click the **Next** button.

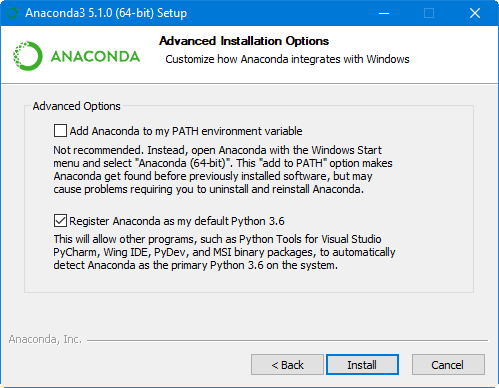
**NOTE**: *Install Anaconda to a directory path that does not contain spaces or unicode characters.*

**NOTE**: *Do not install as Administrator unless admin privileges are required.*

**

1. Choose whether to add Anaconda to your PATH environment variable.

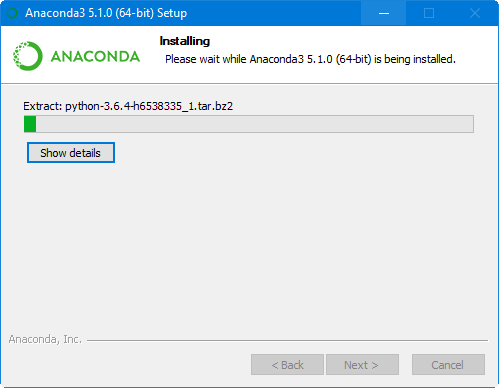
**NOTE**: *We recommend not adding Anaconda to the PATH environment variable, since this can interfere with other software. Instead, use Anaconda software by opening Anaconda Navigator or the Anaconda Prompt from the Start Menu.*

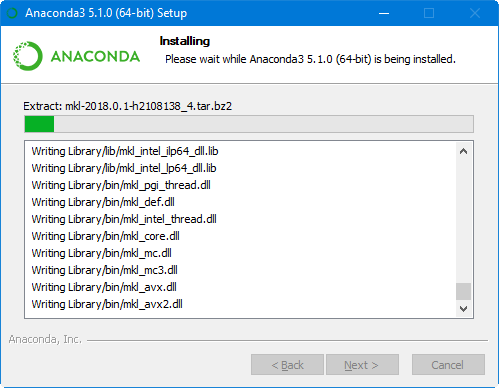


Choose whether to register Anaconda as your default Python 3.6.

Unless you plan on installing and running multiple versions of Anaconda, or multiple versions of Python, accept the default and leave this box checked.

1. Click the **Install** button. If you want to watch the packages Anaconda is installing, click **Show Details**.



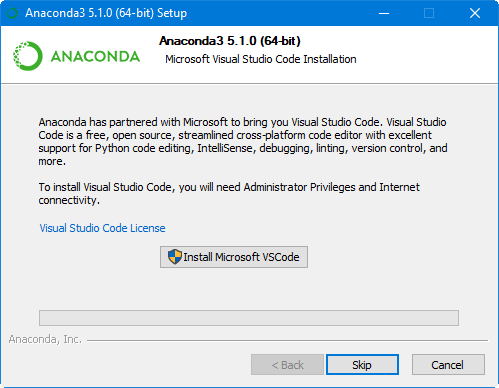


1. Click the **Next** button.

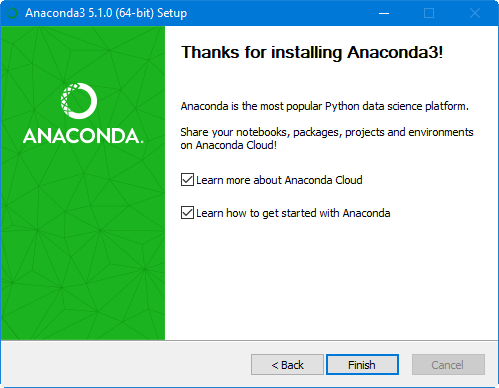
*Optional: To install VS Code, click the Install Microsoft VS Code button. After the install completes click the Next button.*

*Or to install Anaconda without VS Code, click the* ***Skip*** *button.*

***NOTE****: Installing VS Code with the Anaconda installer requires an internet connection. Offline users may be able to find an offline VS Code installer from Microsoft.*

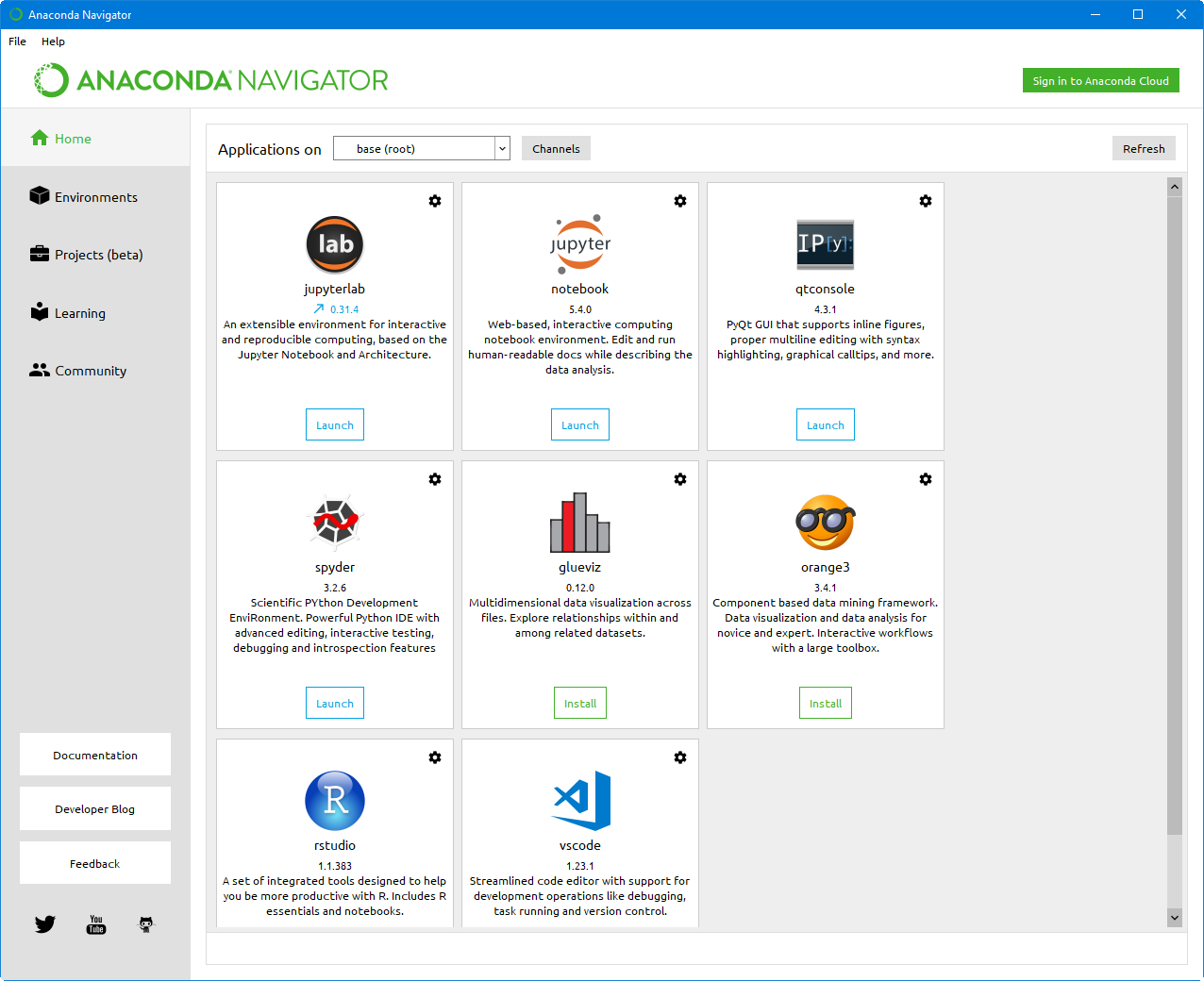


1. After a successful installation you will see the **“Thanks for installing Anaconda”** dialog box. If you wish to read more about Anaconda Cloud package management service and Anaconda support, check the boxes “Learn more about Anaconda Cloud” and “Learn how to get started with Anaconda”. Click the **Finish** button.



1. After your install is complete, verify it by opening Anaconda Navigator, a program that is included with Anaconda: from your Windows Start menu, select the shortcut Anaconda Navigator.

**Note:***If Navigator opens, you have successfully installed Anaconda. If not, check that you completed each step above.*



#### **spyder**

spyder stands for **S**cientific **PY**thon **D**evelopment **E**nvi**R**onment. It is a powerful IDE for writing Python codes and offers features like code editing, testing, and debugging.

spyder is already included in these Python Scientific Distributions:

* Anaconda
* WinPython
* Python(x,y)

You can start using it immediately after installing one of them (you only need to install one!).

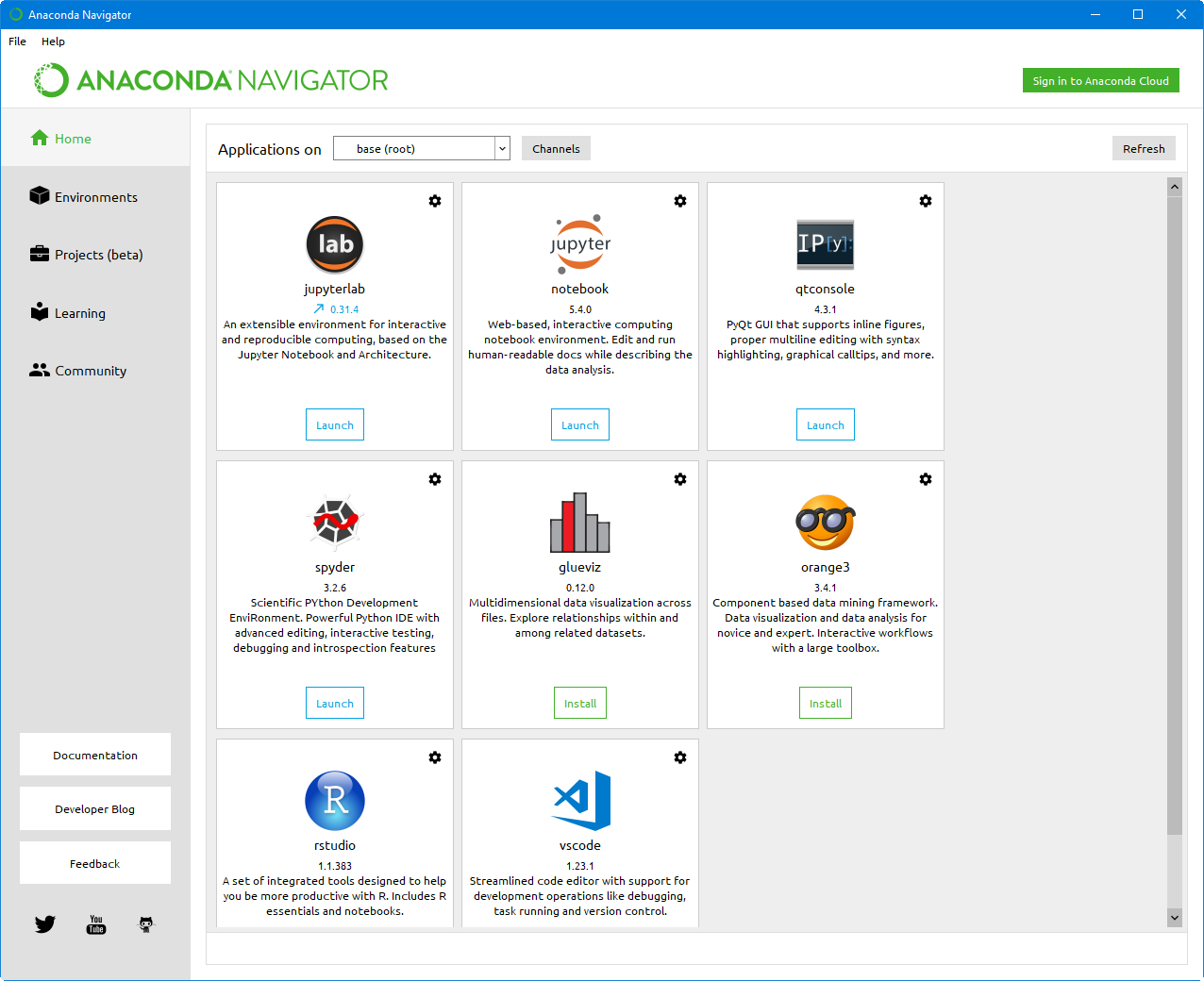
If you want to install spyder directly, you need to follow these steps:

1. Install the essential requirements:
   * The Python programming language
   * PyQt5 (recommended) or PyQt4
2. Install spyder and its dependencies by running this command:

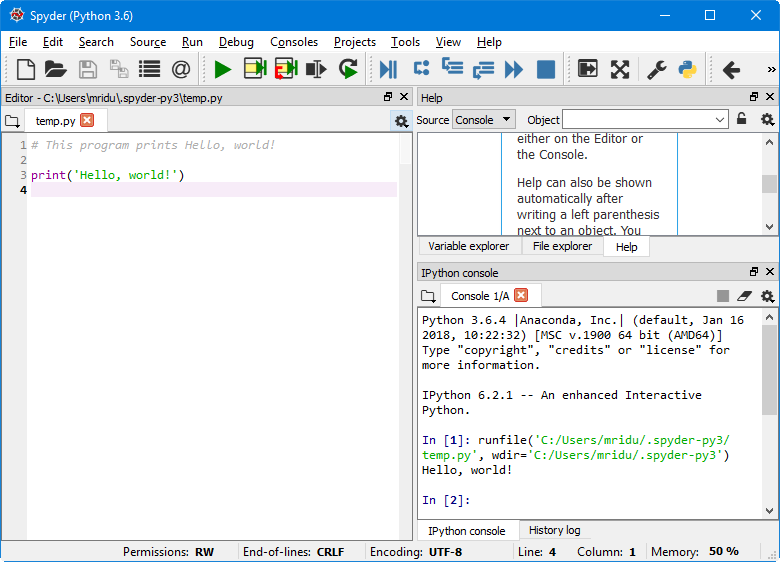
pip install spyder

Executing Program in spyder

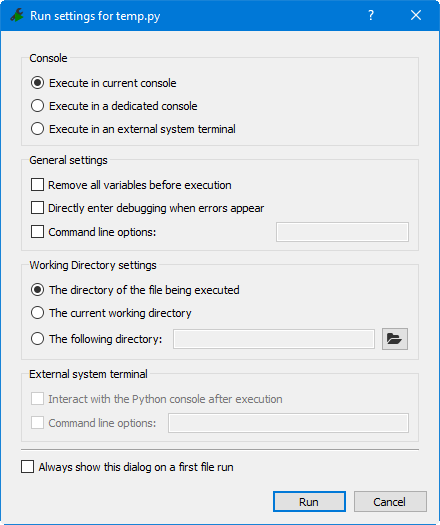
1. In the Anaconda Navigator window, click **Launch** under spyder.



1. In the Spyder editor window, type the program as shown in the image below and press F5 to execute the program.

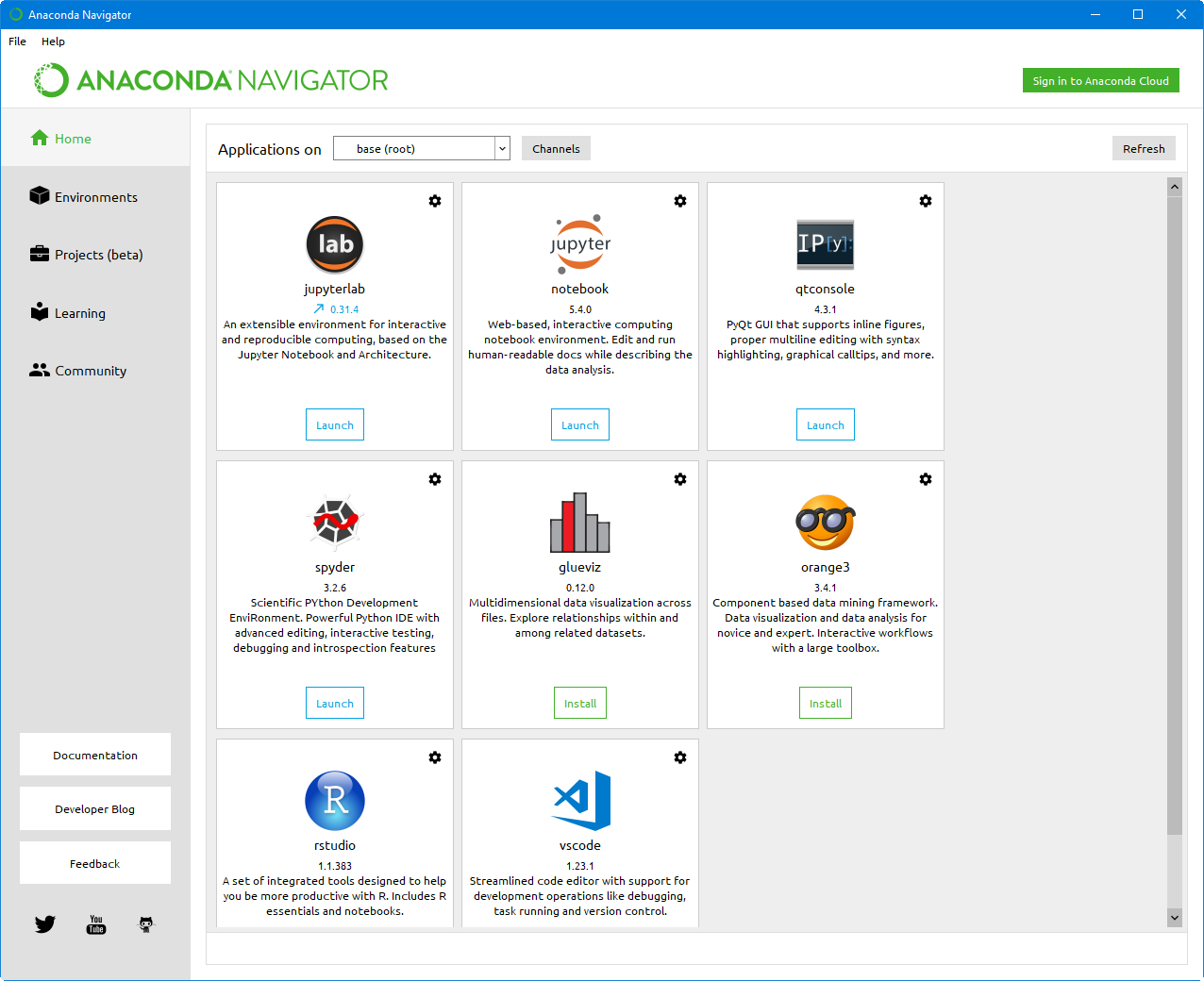


**Note:***When you execute a program in Spyder editor for the first time, you get the Run settings window where you can modify the run settings for your program.*

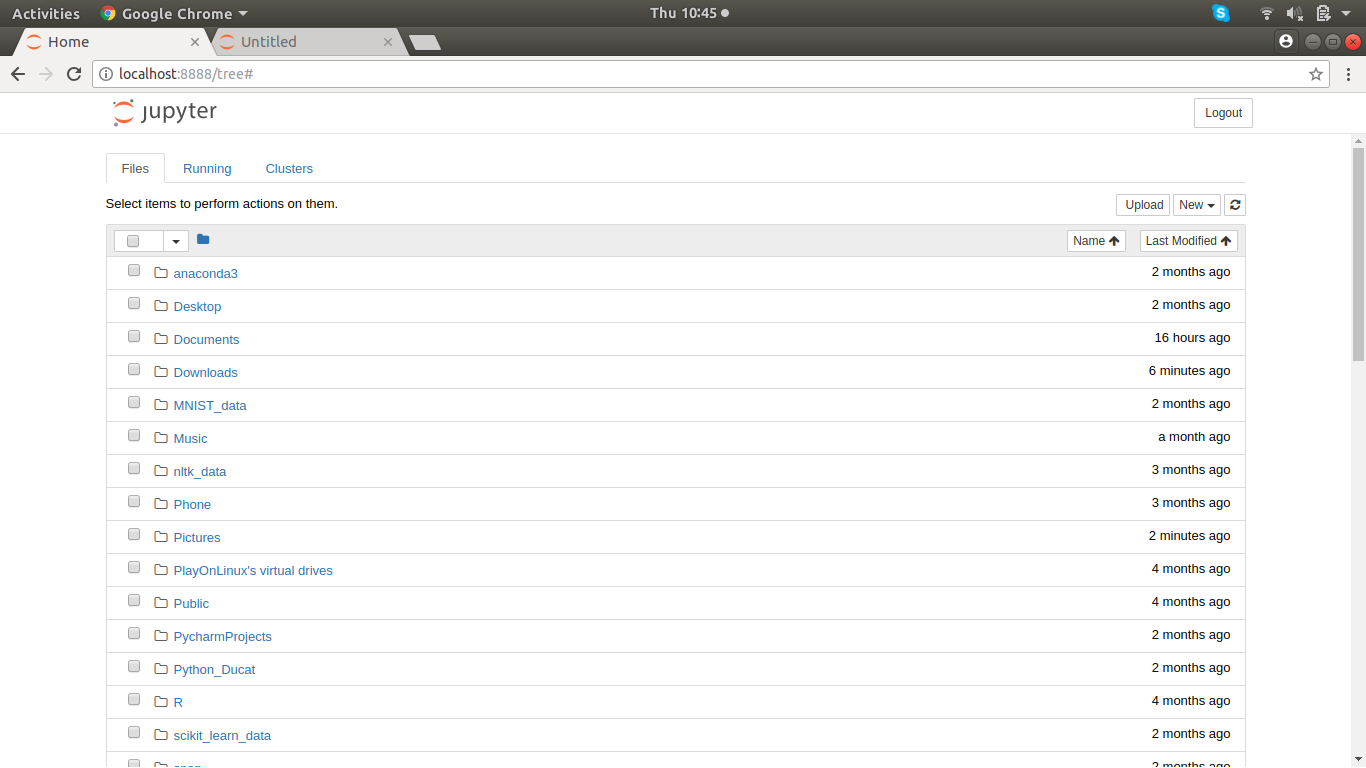


#### **Jupyter**

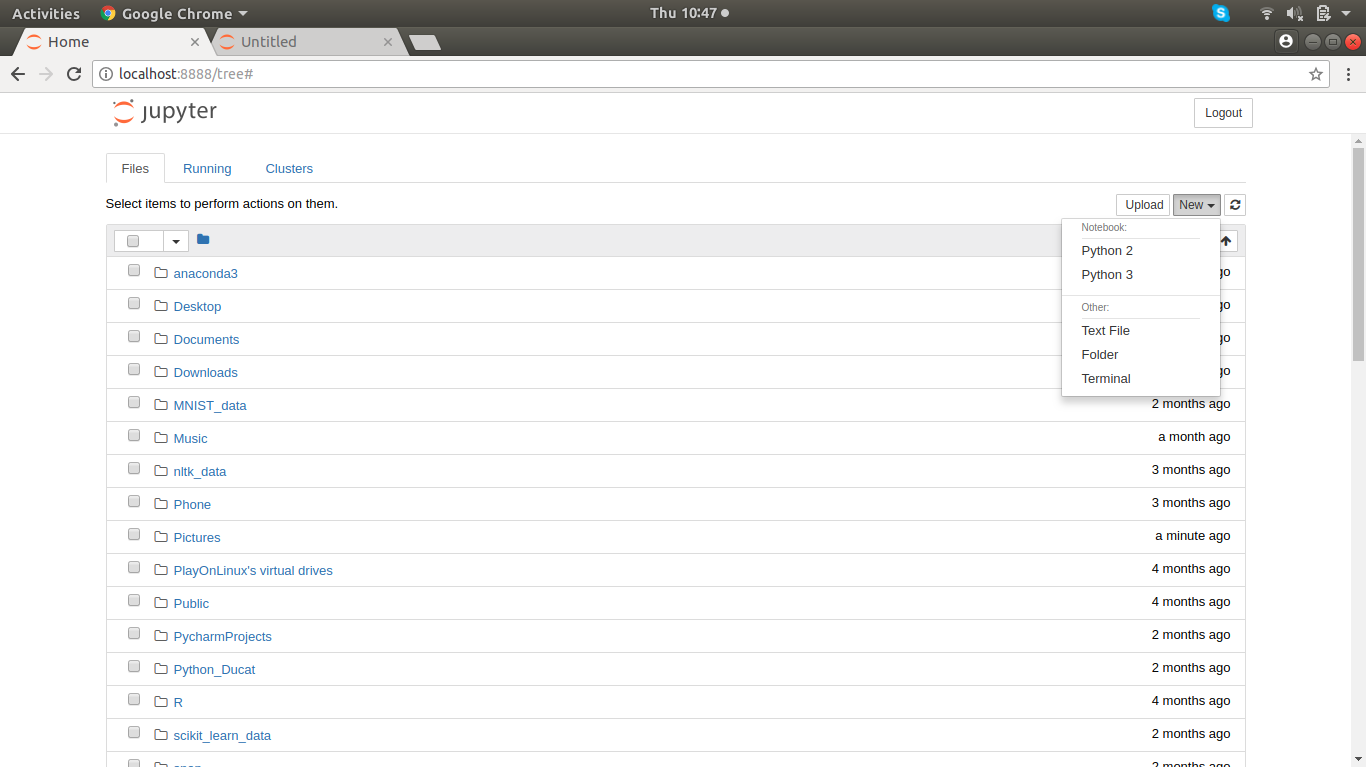
1. Just like Spyder, Jupyter can also be launched from Anaconda Navigator.



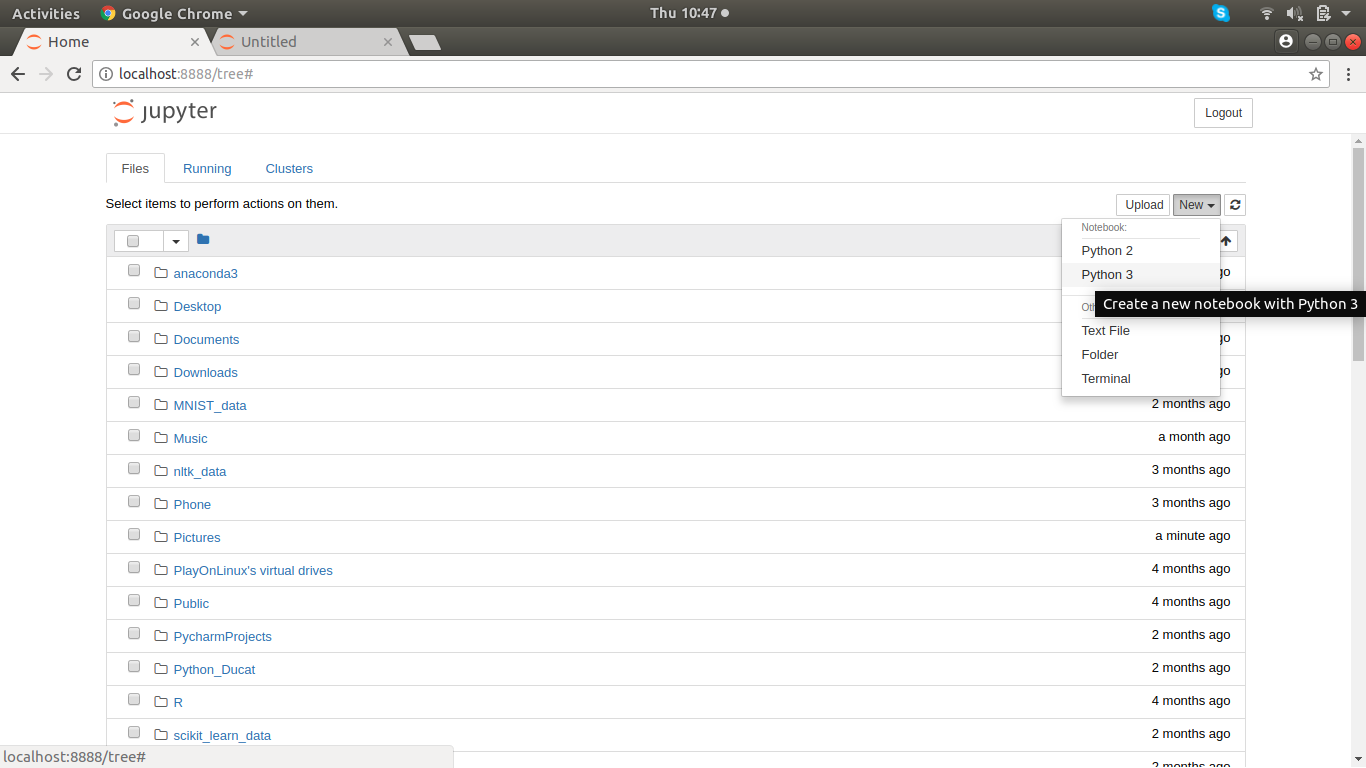
Click **Launch** to open the Jupyter window.



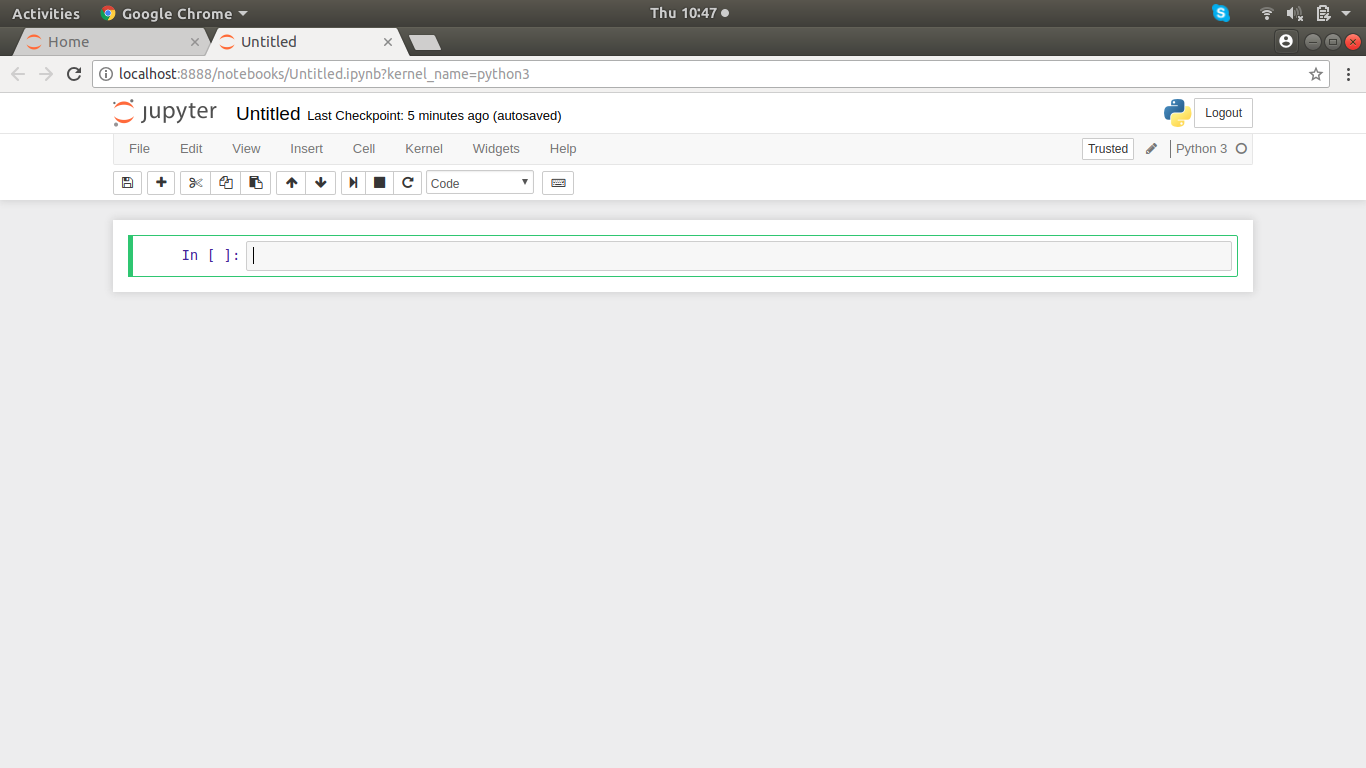
1. Click **New**.



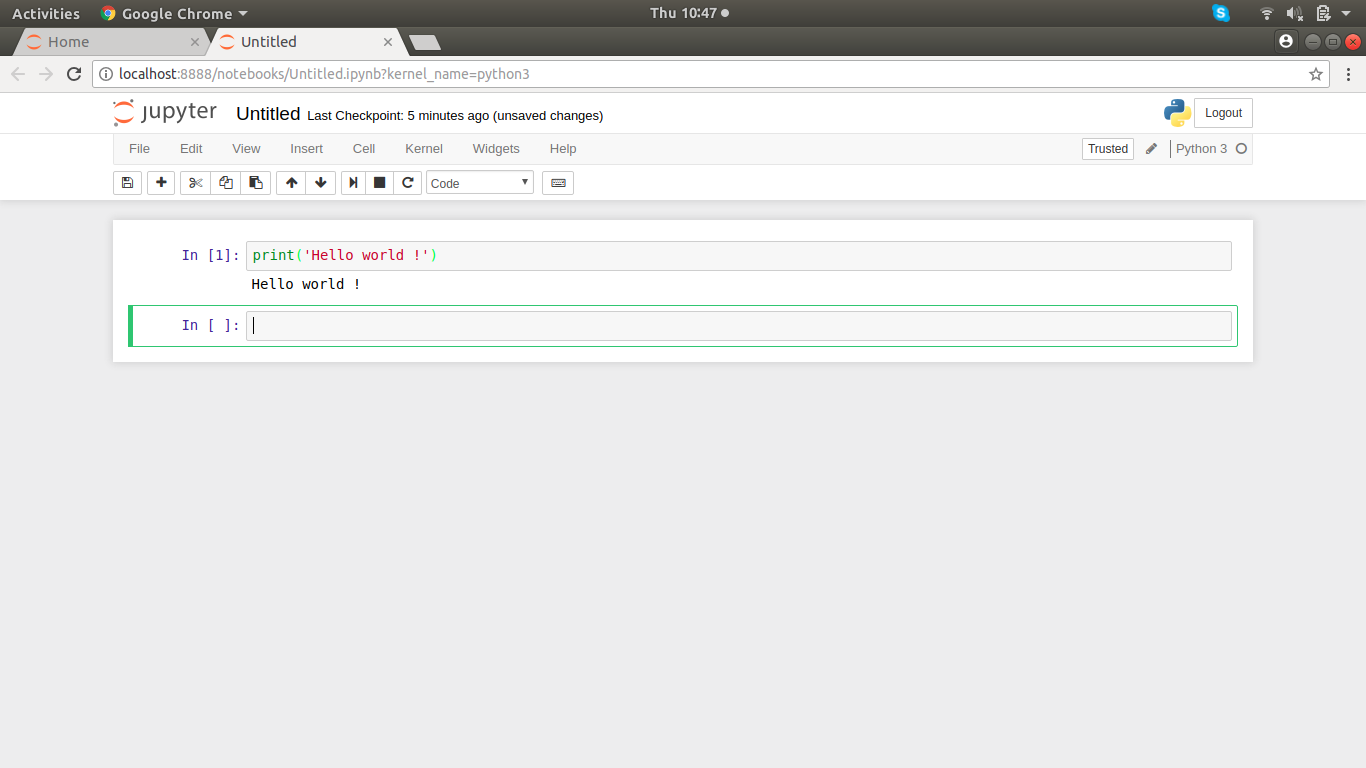
1. Click **Python 3** to open a new notebook with Python 3.



The Jupyter notebook will appear.



1. In the prompt, type print (‘Hello world!’) and press Shift + Enter.



#### **Sublime Text**

The Sublime Text 3 editor Python is an easy-to-use, fast, and lightweight IDE. It also allows to enhance the functionality of the editor by using packages and customizing settings.

# Chapter 03 – Getting Started with Python Programming

## Basic Syntax

Before you start writing codes in Python, it’s important to understand the basic concepts and syntaxes. Let us take a look at these concepts one-by-one with the help of examples.

### Python Identifier

Identifiers are the unique names that are used to identify objects, classes, modules, functions or variables. Identifiers are unlimited in length and are case-sensitive.

The naming conventions for Python identifiers are as follows:

* An identifier name can be a combination of uppercase or lowercase letters A through Z, an underscore, and digits 0 through 9. For example, myCode, code\_test\_1, and this\_is\_my\_code.
* An identifier name cannot start with a digit/number.
* An identifier name cannot have a special character, such as @, $, and %.

Here are naming conventions for Python identifiers-

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

### Python Keywords

Python keywords are the reserved words that cannot be used as ordinary identifiers. They must be spelled exactly as written here:

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

### Lines and Multi-line Statements

A Python program is divided into a number of logical lines.

The end of a logical line is represented by the NEWLINE token. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (for example, between statements in compound statements).

A logical line is constructed from one or more physical lines by following the explicit or implicit line joining rules.A physical line is a sequence of characters terminated by an   
end-of-line sequence.

Two or more physical lines may be joined into logical lines using backslash characters (\).

**Example:**

weekDays = ['Sunday', 'Monday', 'Tuesday',\

'Wednesday', 'Thursday', 'Friday'\

'Saturday']

**Note:***A line ending in a backslash cannot carry a comment.*

Expressions in parentheses, square brackets, or curly braces can be split over more than one physical line without using backslashes.

**Example:**

weekDays = ['Sunday', 'Monday', 'Tuesday', # These are the

'Wednesday', 'Thursday', 'Friday', # days

'Saturday'] # in a week

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines.

**Note:***Implicitly continued lines can also occur within triple-quoted strings, but in that case they cannot carry comments.*

### Comments

Using comments in a code is a recommended practice because it helps explain the purpose of the code or provide descriptions for the sections of code to anyone else looking at your code for review, fix, or modify.

A comment starts with a hash character (#) and ends at the end of the physical line.

Just like statements, comments can also span into multiple lines. In case of multiline comments, start each line with a hash character (#).

Comments are ignored by the Python Interpreter.

**Examples:**

1. Single line comment:

#This is a single line comment.

1. Multiline comment

#This comment will

#span into two lines.

### Blocks

A Python program is constructed from code blocks. A block is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Some more examples of a block include:

* A script file (a file given as standard input to the interpreter or specified as a command line argument to the interpreter) is a code block.
* A script command (a command specified on the interpreter command line with the ‘-c’ option) is a code block.
* The string argument passed to the built-in functions eval() and exec() is a code block.  
  **Note:** *You will learn about the built-in functions (including eval() and exec())later in the course.*

A code block is executed in an execution frame. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block’s execution has completed.In Python, a code block is denoted with the help of indented statements. All statements within a block are indented at the same level.

**Example:**

x = 20

y = 20

if x>y:

print("x is greater")

elif x == y:

print("x and y are equal")

else:

print("y is greater")

x and y are equal

### Indentation

In Python, a code is structured by using indentation. The level of indentation is determined by the whitespace (spaces or tabs) at the beginning of a logical line in a code. Indentation in a Python code determines the grouping of statements. If a code has inconsistent tabs and spaces, the indentation is rejected and a IndentationError is raised.

**Example:**

Here is an example of a correctly (though confusingly) indented piece of Python code:

def perm(l):

# Compute the list of all permutations of l

if len(l) <= 1:

return [l]

r = []

for i in range(len(l)):

s = l[:i] + l[i+1:]

p = perm(s)

for x in p:

r.append(l[i:i+1] + x)

return r

The following example shows various indentation errors:

def perm(l): # error: first line indented

for i in range(len(l)): # error: not indented

s = l[:i] + l[i+1:]

p = perm(l[:i] + l[i+1:]) # error: unexpected indent

for x in p:

r.append(l[i:i+1] + x)

return r # error: inconsistent dedent

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer — the indentation of return r does not match a level popped off the stack.)

### Quotations

In Python, you can use single, double, or triple quotes for strings. You have to ensure that the string is enclosed within the same type of quote.

You can use triple quotes if the string spans across multiple lines.

**Examples:**

1. single = 'Single quote'
2. double = "Double quotes"
3. triple = """There are multiple lines

in this paragraph."""

### Expressions

In Python, expressions are anything that returns a value. Expressions are created by using variables, functions, operators, constants, and evaluations.

**Examples:**

1. >>>4+4
2. >>>squareAddition((p\*p)+(q\*q))

**Note:**For this example to run correctly, the variables p and q must have a value assigned and squareAddition should be a previously deﬁned function.

### Types of Errors

There are three types of errors that you encounter in executing a Python code: Syntax errors, runtime errors, and logical errors. Let’s understand these types in more detail.

Syntax Errors

Syntax errors or parsing errors are the most common type of errors that you come across when you are learning any language.

Syntax errors in Python commonly occur when you:

* leave out a keyword
* put a keyword in the wrong place
* leave out a symbol, such as a colon, comma, or brackets
* misspell a keyword
* put incorrect indentation
* include an empty block

**Example:**

>>>print'Hello world'

File "<stdin>", line 1

print'Hello world'

^

SyntaxError: invalid syntax

In case of a syntax error, the parser points at the line where the error was detected.

Runtime Errors

There may be cases where the code is syntactically correct but an error message is thrown during code execution. Such error messages are called runtime errors or exceptions.

Runtime errors in Python commonly occur when you:

* Divide a number by zero
* Perform an operation on incompatible types
* Use an identifier which has not been defined
* Access a list element, dictionary value, or object attribute which doesn’t exist
* Try to access a file which doesn’t exist

Most of the times, exceptions are not handled by programs and an error message is thrown.

Logical Errors

The syntax and runtime errors are easy to find because the interpreter indicates exactly where there is an issue with the code. However, there are errors that are not detected by the interpreter and these are known as logical errors.

The error is caused by a mistake in the program’s logic. You won’t get an error message, because no syntax or runtime error has occurred. You will have to find the problem on your own by reviewing all the relevant parts of your code – although some tools can flag suspicious code which looks like it could cause unexpected behavior.

Logical errors in Python may occur when you:

* Use the wrong variable name
* Indent a block to the wrong level
* Use integer division instead of floating-point division
* Get operator precedence wrong
* Make a mistake in a boolean expression

## Operators

### Arithmetic Operators

The Arithmetic operators are used for performing mathematical operations. The table below shows the types of arithmetic operators used in Python.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + | Adds the values on the left and right side of the operator. | >>>5+7  12 |
| - | Subtracts values of right side operand from left side operand. | >>>7-5  2 |
| \* | Multiplies the values on left and right side of the operator. | >>>7\*5  35 |
| / | Divides the left-side value with the right-side value and returns a float value. | >>>7/5  1.4 |
| % | Divides the left-side value with right-side value and returns the remainder. | >>>7%5  2 |
| \*\* | Calculates the exponential power of the left-side value. | >>>2\*\*3  8 |
| // | Calculates quotient in which the digits after decimal points are not taken into account. | >>>7//5  1 |

### Relational/Conditional Operators

Relational operators and used for comparing values. The table below lists the relational operators used in Python.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| == | If the two value of its operands are equal, then the condition becomes true, otherwise false | >>>7==5  False |
| != | If two operands values are not equal, then condition becomes true. Both the operators define the same meaning and function. | >>>7!=5  True |
| > | Returns true if the left-side value is greater than the right-sidevalue. | >>>7>5  True |
| < | Returns true if the left-sidevalue is less than the right-side value. | >>>7<5  False |
| <= | Returns true if left-side value is less than or equal to the right-side value. | >>>7<=5  False |
| >= | Returns true if the left-side value is greater than or equal to the right-side value. | >>>7>=5  True |

### Logical Operators

These operators help validate relationship between two operands and returns true or false. The table below lists the logical operators used in Python.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| or | Returns true if any one of the conditions or operands is true. | >>>a=10  >>>b=20  >>>print(a>10 or b>10)  True |
| and | Returns true if both the conditions/operands are true. | >>>a=10  >>>b=20  >>>print(a>10 and b>10)  False |
| not | It is used to reverse the logical state of its operand. | >>>a=10  >>>b=20  >>>print(not a>10 and b>10)  True |

### Bitwise Operators

Logical operators are used to combine conditional statements. The table below lists the bitwise operators used in Python.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| & | Yields the bitwise AND of its arguments, which must be integers. | >>>8&1  0 |
| | | Yields the bitwise (inclusive) OR of its arguments, which must be integers. | >>>8|1  9 |
| ^ | Yields the bitwise XOR (exclusive OR) of its arguments, which must be integers. | >>>8^1  9 |
| ~ | This unary operator yields the bitwise inversion of its integer argument.  The bitwise inversion of x is defined as -(x+1). It only applies to integral numbers. | >>>~4  -5 |
| << | Shifts the first argument to the left by the number of bits given by the second argument. | >>>2<<1  4 |
| >> | Shifts the first argument to the right by the number of bits given by the second argument. | >>>2>>1  1 |

### Assignment Operators

Assignment operators are used to assign values to variables. The table below lists the assignment operators used in Python.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| = | Assigns the values of right side operand to left side operand. | >>>a=10  >>>a  10 |
| += | Adds right side operand value to the left side operand value and assigns the results to the left operand. | >>>a+=10  >>>a  20 |
| -= | Subtracts right side operand value to the left side operand value and assigns the results to the left operand. | >>>a-=5  >>>a  15 |
| \*= | Multiplies right side operand value to the left side operand value and assigns the results to the left operand. | >>>a\*=2  >>>a  30 |
| /= | Divides right side operand value to the left side operand value and assigns the results to the left operand. | >>>a/=5  >>>a  6.0 |
| %= | Similarly does their respective operations and assigns the operator value to the left operand. | >>>a%=4  >>>a  2.0 |
| \*\*= | >>>a\*\*=6  >>>a  64.0 |
| //= | >>>a//=6  >>>a  10.0 |

### Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location. The table below lists the identity operators used in Python.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| is | The result becomes true if values on either side of the operator point to the same object and False otherwise. | >>>a=4  >>>print(type(a) is int)  True |
| is not | The result becomes False if the variables on either side of the operator point to the same object. | >>>print(type(a) is not float)  True |

### Membership Operators

Membership operators are used to test if a sequence is presented in an object. The table below lists the membership operators used in Python.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| in | The result of this operation becomes True if it finds a value in a specified sequence & False otherwise. | >>>list1=[1,2,3,4,5,6]  >>>print(1 in list1)  True |
| not in | Result of this operation becomes True if it doesn't find a value in a specified sequence & False otherwise. | >>>print(7 not in list1)  True |

### Operator Precedence and Associativity

In cases where multiple operators appear in the same expression, the Python Interpreter applies the rules of arithmetic to define precedence and associativity.

Precedence rules are applied when two types of operators are used and the Interpreter has to decide which one to apply first.

Associativity rules are applied when two operators of the same precedence are included in an expression and the Interpreter has to decide which one to apply first.

The following table summarizes the operator precedence in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left).

**Note**: *Comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature.*

|  |  |
| --- | --- |
| Operator | Description |
| lambda | Lambda expression |
| [if](https://docs.python.org/3.6/reference/compound_stmts.html#if) – [else](https://docs.python.org/3.6/reference/compound_stmts.html#else) | Conditional expression |
| [or](https://docs.python.org/3.6/reference/expressions.html?highlight=operator%20precedence#or) | Boolean OR |
| [and](https://docs.python.org/3.6/reference/expressions.html?highlight=operator%20precedence#and) | Boolean AND |
| [not](https://docs.python.org/3.6/reference/expressions.html?highlight=operator%20precedence#not) x | Boolean NOT |
| [in](https://docs.python.org/3.6/reference/expressions.html?highlight=operator%20precedence#in), [not in](https://docs.python.org/3.6/reference/expressions.html?highlight=operator%20precedence#not-in), [is](https://docs.python.org/3.6/reference/expressions.html?highlight=operator%20precedence#is), [is not](https://docs.python.org/3.6/reference/expressions.html?highlight=operator%20precedence#is-not), <, <=, >, >=, !=, == | Comparisons, including membership tests and identity tests |
| | | Bitwise OR |
| ^ | Bitwise XOR |
| & | Bitwise AND |
| <<, >> | Shifts |
| +, - | Addition and subtraction |
| \*, @, /, //, % | Multiplication, matrix multiplication, division, floor division, remainder |
| +x, -x, ~x | Positive, negative, bitwise NOT |
| \*\* | Exponentiation |
| await x | Await expression |
| x[index], x[index:index], x(arguments...), x.attribute | Subscription, slicing, call, attribute reference |
| (expressions...), [expressions...], {key: value...}, {expressions...} | Binding or tuple display, list display, dictionary display, set display |

# Chapter 04 – Conditions and Loops

## Conditions

Conditions are used in programming languages to execute code and produce the result only if a certain condition is met or satisfied.

In Python, if statements are used for conditional execution. The types of if statements used are as follows:

* Simple if statements
* if else Statement
* elif Statement
* Nested if Statements

### The Simple if Statement

In a simple if statement, the program evaluates the if expression and executes statements only if the specified condition is true.

**Example:**

x = 20

if x> 10:

print("x is greater")

x is greater

### The if else Statement

The if else statement also contains a Boolean expression. In this case, the if statement is followed by an optional else statement and if the expression results in false, then the elsestatement is executed. The if else statement is also called alternative execution in which there are two possibilities of the condition determined and any one of them will get executed.

**Example:**

x = 20

y = 25

if x>y:

print("x is greater")

else:

print("y is greater")

y is greater

### The elif Statement

The elif keyword is used in Python as a replacement of the else ifstatements used in other programming languages. The keyword is used to add another condition in the program.

**Example:**

x = 20

y = 20

if x>y:

print("x is greater")

elif x == y:

print("x and y are equal")

else:

print("y is greater")

x and y are equal

### Nested if Statements

A code is said to have nested if statements when you have an if statement and/or an if else statement inside another if or if else statement.

In this case, more than one if conditions are applied and there can be more than one if within elif.

Example:

# In this program, we check

# if the given number is +ve, -ve, or 0

# and display a message.

num = 10

if num >= 0:

if num == 0:

print("Zero")

else:

print("Positive number")

else:

print("Negative number")

Positive number

## Loops

Loops are a sequence of instructions that executes a specific set of tasks based on some condition. The tasks are executed till the time the condition is met or is true and stops when the condition evaluates to false.

Loops are helpful when you do not know how many iterations will be required.

The types of loop statements used in Python are as follows:

* while loops
* for Loops
* Nested Loops

### while Loops

The while statement is used for repeated execution as long as an expression is true.

**Example:**

# Program to add natural

# numbers upto

# sum = 1+2+3+...+n

# To take input from the user,

# n = int(input("Enter n: "))

n = 10

# initialize sum and counter

sum = 0

i = 1

while i <= n:

sum = sum + i

i = i+1 # update counter

# print the sum

print("The sum is", sum)

Enter n: 10

The sum is 55

The while loop repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates.

### for Loops

The for statement in Python differs a bit from the for loop used in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

**Example:**

>>> # Measure the length of some strings:

... words = ['dog', 'tiger', 'elephant']

>>> for w in words:

... print(w, len(w))

...

dog 3

tiger 5

elephant 8

#### The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

**Syntax:**

class range(start, stop[, step])

The range() function generates arithmetic progressions.

**Example:**

>>> for i in range(5):

... print(i)

...

0

1

2

3

4

**Note:** *The range() function of Python 3 was called xrange() in Python 2. xrange() function has now removed in Python 3.*

The arguments to the range constructor must be integers (either built-in int or any object that implements the \_\_index\_\_ special method). If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. If step is zero, ValueError is raised.

For a positive step, the contents of a range r are determined by the formula   
r[i] = start + step\*i where i >= 0 and r[i] < stop.

For a negative step, the contents of the range are still determined by the formula   
r[i] = start + step\*i, but the constraints are i >= 0 and r[i] > stop.

A range object will be empty if r[0] does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than sys.maxsize are permitted but some features (such as len()) may raise OverflowError.

**Examples:**

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(1, 11))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> list(range(0, 30, 5))

[0, 5, 10, 15, 20, 25]

>>> list(range(0, 10, 3))

[0, 3, 6, 9]

>>> list(range(0, -10, -1))

[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

>>> list(range(0))

[]

>>> list(range(1, 0))

[]

**Note:***Ranges implement all of the common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).*

### Nested Loops

In Python, you can create one loop inside another loop. In other words, you can create:

* A for loop inside another for loop
* A for loop inside a while loop
* A while loop inside another while loop
* A while loop inside a for loop

**Example:**

for g in range(1, 6):

for k in range(1, 3):

print ("%d \* %d = %d" % ( g, k, g\*k))

1 \* 1 = 1

1 \* 2 = 2

2 \* 1 = 2

2 \* 2 = 4

3 \* 1 = 3

3 \* 2 = 6

4 \* 1 = 4

4 \* 2 = 8

5 \* 1 = 5

5 \* 2 = 10

### Loop Control Statements

In Python, if you want to change the flow if a normal loop, you can use loop control statements. The loop control statements used in Python are:

* break statements
* continuestatements
* pass statements

Let’s understand these statements in detail.

#### break statement

The break statement occurs syntactically nested in a for or while loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional else clause if the loop has one.

This is exemplified by the following loop, which searches for prime numbers:

**Example:**

>>> for n in range(2, 10):

... for x in range(2, n):

... if n % x == 0:

... print(n, 'equals', x, '\*', n//x)

... break

... else:

... # loop fell through without finding a factor

... print(n, 'is a prime number')

...

2 is a prime number

3 is a prime number

4 equals 2 \* 2

5 is a prime number

6 equals 2 \* 3

7 is a prime number

8 equals 2 \* 4

9 equals 3 \* 3

#### continue statement

The continue statementcontinues with the next iteration of the loop.

**Example:**

>>> for num in range(2, 10):

... if num % 2 == 0:

... print("Found an even number", num)

... continue

... print("Found a number", num)

Found an even number 2

Found a number 3

Found an even number 4

Found a number 5

Found an even number 6

Found a number 7

Found an even number 8

Found a number 9

#### pass statement

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

**Example:**

>>> while True:

... pass # Busy-wait for keyboard interrupt (Ctrl+C)

...

This is commonly used for creating minimal classes.

**Example:**

>>> class MyEmptyClass:

... pass

...

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

**Example:**

>>> def initlog(\*args):

... pass # Remember to implement this!

...

# Chapter 05 – Data Structures

## Introduction

Data types in Python language are simple to understand and easy to use. The Python Interpreter can identify the types of data you want to store and hence does not require you to define the data type.

The data type determines:

* The possible values for that type.
* The operations that can be done with that values.
* Conveys the meaning of data.
* The way values of that type can be stored.

Everything in Python programming is an object, and each object has its own unique identity (a type and a value).

Python has following built-in data types:

* Numbers
* int (signed integer)
* float
* complex
* Sequences
* string
* list
* tuple
* set
* dictionary
* boolean

***Note:*** *Sequence data types can be of two types:*

* ***Immutable****: An object with a fixed value. Immutable objects include numbers, string and tuple. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.*
* ***Mutable****:Mutable objects can change their value but keep their id()*

## Built-in Functions

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.

|  |  |  |
| --- | --- | --- |
| Function | Description | Data Type Supported |
| [abs()](https://www.programiz.com/python-programming/methods/built-in/abs) | Returns absolute value of a number |  |
| [all()](https://www.programiz.com/python-programming/methods/built-in/all) | Returns true when all elements in iterable is true | List, String, Tuple, Set, Dictionary |
| [any()](https://www.programiz.com/python-programming/methods/built-in/any) | Checks if any element of an iterable is True | List, String, Tuple, Set, Dictionary |
| [ascii()](https://www.programiz.com/python-programming/methods/built-in/ascii) | Returns string containing printable representation | List, String, Tuple, Set, Dictionary |
| [bin()](https://www.programiz.com/python-programming/methods/built-in/bin) | Converts integer to binary string |  |
| [bool()](https://www.programiz.com/python-programming/methods/built-in/bool) | Coverts a value to boolean | List, String, Tuple, Set, Dictionary |
| [bytearray()](https://www.programiz.com/python-programming/methods/built-in/bytearray) | Returns array of given byte size | String |
| [bytes()](https://www.programiz.com/python-programming/methods/built-in/bytes) | Returns immutable bytes object | String |
| [callable()](https://www.programiz.com/python-programming/methods/built-in/callable) | Checks if the object is callable |  |
| [chr()](https://www.programiz.com/python-programming/methods/built-in/chr) | Returns a character (a string) from an integer |  |
| [classmethod()](https://www.programiz.com/python-programming/methods/built-in/classmethod) | Returns class method for given function |  |
| [compile()](https://www.programiz.com/python-programming/methods/built-in/compile) | Returns a Python code object | String |
| [complex()](https://www.programiz.com/python-programming/methods/built-in/complex) | Creates a complex number | String |
| [delattr()](https://www.programiz.com/python-programming/methods/built-in/delattr) | Deletes attribute from the object |  |
| [dict()](https://www.programiz.com/python-programming/methods/built-in/dict) | Creates a dictionary | Dictionary |
| [dir()](https://www.programiz.com/python-programming/methods/built-in/dir) | Tries to return attributes of object |  |
| [divmod()](https://www.programiz.com/python-programming/methods/built-in/divmod) | Returns a tuple of quotient and remainder |  |
| [enumerate()](https://www.programiz.com/python-programming/methods/built-in/enumerate) | Returns an enumerate object | List, String, Tuple, Set, Dictionary |
| [eval()](https://www.programiz.com/python-programming/methods/built-in/eval) | Runs Python code within program |  |
| [exec()](https://www.programiz.com/python-programming/methods/built-in/exec) | Executes dynamically created program |  |
| [filter()](https://www.programiz.com/python-programming/methods/built-in/filter) | Constructs iterator from elements which are true | List, String, Tuple, Set, Dictionary |
| [float()](https://www.programiz.com/python-programming/methods/built-in/float) | Returns floating point number from number, string | String |
| [format()](https://www.programiz.com/python-programming/methods/built-in/format) | Returns formatted representation of a value |  |
| [frozenset()](https://www.programiz.com/python-programming/methods/built-in/frozenset) | Returns immutable frozenset object | Set |
| [getattr()](https://www.programiz.com/python-programming/methods/built-in/getattr) | Returns value of named attribute of an object |  |
| [globals()](https://www.programiz.com/python-programming/methods/built-in/globals) | Returns dictionary of current global symbol table |  |
| [hasattr()](https://www.programiz.com/python-programming/methods/built-in/hasattr) | Returns whether object has named attribute |  |
| [hash()](https://www.programiz.com/python-programming/methods/built-in/hash) | Returns hash value of an object |  |
| [help()](https://www.programiz.com/python-programming/methods/built-in/help) | Invokes the built-in help system |  |
| [hex()](https://www.programiz.com/python-programming/methods/built-in/hex) | Converts to integer to hexadecimal |  |
| [id()](https://www.programiz.com/python-programming/methods/built-in/id) | Returns identity of an object |  |
| [input()](https://www.programiz.com/python-programming/methods/built-in/input) | Reads and returns a line of string | String |
| [int()](https://www.programiz.com/python-programming/methods/built-in/int) | Returns integer from a number or string | String |
| [isinstance()](https://www.programiz.com/python-programming/methods/built-in/isinstance) | Checks if an object is an instance of class |  |
| [issubclass()](https://www.programiz.com/python-programming/methods/built-in/issubclass) | Checks if an object is subclass of a class |  |
| [iter()](https://www.programiz.com/python-programming/methods/built-in/iter) | Returns iterator for an object | List, String, Tuple, Set, Dictionary |
| [len()](https://www.programiz.com/python-programming/methods/built-in/len) | Returns length of an object | List, String, Tuple, Set, Dictionary |
| [list() Function](https://www.programiz.com/python-programming/methods/built-in/list) | Creates list in Python | List |
| [locals()](https://www.programiz.com/python-programming/methods/built-in/locals) | Returns dictionary of current local symbol table |  |
| [map()](https://www.programiz.com/python-programming/methods/built-in/map) | Applies function and returns a list | List, String, Tuple, Set, Dictionary |
| [max()](https://www.programiz.com/python-programming/methods/built-in/max) | Returns largest element | List, String, Tuple, Set, Dictionary |
| [memoryview()](https://www.programiz.com/python-programming/methods/built-in/memoryview) | Returns memory view of an argument |  |
| [min()](https://www.programiz.com/python-programming/methods/built-in/min) | Returns smallest element | List, String, Tuple, Set, Dictionary |
| [next()](https://www.programiz.com/python-programming/methods/built-in/next) | Retrieves next element from iterator |  |
| [object()](https://www.programiz.com/python-programming/methods/built-in/object) | Creates a featureless object |  |
| [oct()](https://www.programiz.com/python-programming/methods/built-in/oct) | Converts integer to octal |  |
| [open()](https://www.programiz.com/python-programming/methods/built-in/open) | Returns a file object |  |
| [ord()](https://www.programiz.com/python-programming/methods/built-in/ord) | Returns unicode code point for unicode character | String |
| [pow()](https://www.programiz.com/python-programming/methods/built-in/pow) | Returns x to the power of y |  |
| [print()](https://www.programiz.com/python-programming/methods/built-in/print) | Prints the given object |  |
| [property()](https://www.programiz.com/python-programming/methods/built-in/property) | Returns a property attribute |  |
| [range()](https://www.programiz.com/python-programming/methods/built-in/range) | Return sequence of integers between start and stop |  |
| [repr()](https://www.programiz.com/python-programming/methods/built-in/repr) | Returns printable representation of an object |  |
| [reversed()](https://www.programiz.com/python-programming/methods/built-in/reversed) | Returns reversed iterator of a sequence | List, String, Tuple, |
| [round()](https://www.programiz.com/python-programming/methods/built-in/round) | Rounds a floating point number to ndigits places. |  |
| [set()](https://www.programiz.com/python-programming/methods/built-in/set) | Returns a Python set | Set |
| [setattr()](https://www.programiz.com/python-programming/methods/built-in/setattr) | Sets value of an attribute of object |  |
| [slice()](https://www.programiz.com/python-programming/methods/built-in/slice) | Creates a slice object specified by range() | List, String, Tuple |
| [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) | Returns sorted list from a given iterable | List, String, Tuple, Set, Dictionary |
| [staticmethod()](https://www.programiz.com/python-programming/methods/built-in/staticmethod) | Creates static method from a function |  |
| [str()](https://www.programiz.com/python-programming/methods/built-in/str) | Returns informal representation of an object |  |
| [sum()](https://www.programiz.com/python-programming/methods/built-in/sum) | Add items of an iterable | List, String, Tuple, Set, Dictionary |
| [super()](https://www.programiz.com/python-programming/methods/built-in/super) | Allow you to refer parent class by super |  |
| [tuple() Function](https://www.programiz.com/python-programming/methods/built-in/tuple) | Creates a tuple | Tuple |
| [type()](https://www.programiz.com/python-programming/methods/built-in/type) | Returns type of an object |  |
| [vars()](https://www.programiz.com/python-programming/methods/built-in/vars) | Returns \_\_dict\_\_ attribute of a class |  |
| [zip()](https://www.programiz.com/python-programming/methods/built-in/zip) | Returns an iterator of tuple | List, String, Tuple, Set, Dictionary |
| [\_\_import\_\_()](https://www.programiz.com/python-programming/methods/built-in/__import__) | Advanced function called by import |  |

## Numbers

### Types of Numbers

Python supports three types of numbers:

* **Integers**: Integers are whole numbers without decimal point. They can be positive or negative as long as they don’t contain a decimal point that would make a number a floating number, a distinct numeric type. Integers have unlimited size in Python 3.

**Example**:

myint = 7  
print(myint)

7

* **Floating point numbers**: Also known as floats, floating-point numbers signify real numbers. Floats are written with a decimal point that segregates the integer from the fractional numbers. They may also be written in scientific notation where the uppercase or lowercase letter ‘e’ signifies the 10 power.

**Example**:

1. myfloat = 7.0  
   print(myfloat)

7.0

1. myfloat = float(7)  
   print(myfloat)

7.0

* **Complex numbers**: Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number z, use z.real and z.imag.

class complex([real[, imag]])return a complex number with the value real + imag\*1j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float. If both arguments are omitted, returns 0j.

**Note:***When converting from a string, the string must not contain whitespace around the central + or- operator. For example, complex('1+2j') is fine, but complex('1 + 2j') raises ValueError.*

### Number Methods

|  |  |
| --- | --- |
| Functions | Description |
| abs() | Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned. |

**Note**: *By default, Python provides theabs() method for numbers data type. You can add the Math module to add more methods including ceil(), cmp(), exp(), fabs(),floor(), log(), log10(), max(x1,x2…), min(), modf(), pow(), round(), and sqrt().*

## string

A string is usually a bit of text in programming that is written to be displayed to users. It is known to Python when you want to display a string. This is because programmers use either double quote " or single quote ' to enclose a word or group of words to express a string.

A string is defined either with a single quote or a double quotes.

**Example:**

>>>mystring = 'hello'  
... print(mystring)

hello  
>>>mystring = "hello"  
... print(mystring)

hello

**Note:** *The difference between the two is that using double quotes makes it easy to include apostrophes (whereas these would terminate the string if using single quotes).*

Let us take a look at what all you can do with a string:

1. A string can be **concatenated** (glued together). Here are different ways concatenation happens:
   1. with the + operator, and repeated with \*:

**Example**:

>>> # 3 times 'un', followed by 'ium'

>>> 3 \* 'un' + 'ium'

'unununium'

* 1. Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

**Example:**

>>> 'Py' 'thon'

'Python'

This feature is particularly useful when you want to break long string:

**Example:**

>>> text = ('Put several string within parentheses '

... 'to have them joined together.')

>>> text

'Put several string within parentheses to have them joined together.'

* 1. This only works with two literals though, not with variables or expressions:

**Example**:

>>> prefix = 'Py'

>>> prefix 'thon' # can't concatenate a variable and a string literal

...

SyntaxError: invalid syntax

>>> ('un' \* 3) 'ium'

...

SyntaxError: invalid syntax

If you want to concatenate variables or a variable and a literal, use +:

**Example:**

>>> prefix + 'thon'

'Python'

1. A string can be **indexed** (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

**Example:**

>>> word = 'Python'

>>> word[0] # character in position 0

'P'

>>> word[5] # character in position 5

'n'

Indices may also be negative numbers, to start counting from the right:

**Example:**

>>> word[-1] # last character

'n'

>>> word[-2] # second-last character

'o'

>>> word[-6]

'P'

**Note:***Since -0 is the same as 0, negative indices start from -1.*

1. In addition to indexing, **slicing** is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

**Example:**

>>> word[0:2] # characters from position 0 (included) to 2 (excluded)

'Py'

>>> word[2:5] # characters from position 2 (included) to 5 (excluded)

'tho'

Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

**Example:**

>>> word[:2] + word[2:]

'Python'

>>> word[:4] + word[4:]

'Python'

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

**Example:**

>>> word[:2] # character from the beginning to position 2 (excluded)

'Py'

>>> word[4:] # characters from position 4 (included) to the end

'on'

>>> word[-2:] # characters from the second-last (included) to the end

'on'

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n.

**Example:**

+---+---+---+---+---+---+

| P | y | t | h | o | n |

+---+---+---+---+---+---+

0 1 2 3 4 5 6

-6 -5 -4 -3 -2 -1

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of word[1:3] is 2.

Attempting to use an index that is too large will result in an error:

**Example:**

>>> word[42] # the word only has 6 characters

Traceback (most recent call last):

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

IndexError: string index out of range

However, out of range slice indexes are handled gracefully when used for slicing:

**Example:**

>>> word[4:42]

'on'

>>> word[42:]

''

1. Python string cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

**Example:**

>>> word[0] = 'J'

...

TypeError: 'str' object does not support item assignment

>>> word[2:] = 'py'

...

TypeError: 'str' object does not support item assignment

If you need a different string, you should create a new one:

**Example:**

>>> 'J' + word[1:]

'Jython'

>>> word[:2] + 'py'

'Pypy'

1. The built-in function **len()** returns the length of a string:

**Example:**

>>> s = 'supercalifragilisticexpialidocious'

>>> len(s)

34

### string Methods

Apart from the string methods listed in the built-in methods, the table below lists the methods that can take string as a parameter and perform some task.

|  |  |
| --- | --- |
| Method | Description |
| [capitalize()](https://www.programiz.com/python-programming/methods/string/capitalize) | Converts first character to capital letter |
| [center()](https://www.programiz.com/python-programming/methods/string/center) | Pads string with specified character |
| [casefold()](https://www.programiz.com/python-programming/methods/string/casefold) | Converts to casefolded string |
| [count()](https://www.programiz.com/python-programming/methods/string/count) | Returns occurrences of substring in string |
| [endswith()](https://www.programiz.com/python-programming/methods/string/endswith) | Checks if string ends with the specified suffix |
| [expandtabs()](https://www.programiz.com/python-programming/methods/string/expandtabs) | Replaces Tab character With Spaces |
| [encode()](https://www.programiz.com/python-programming/methods/string/encode) | Returns encoded string of given string |
| [find()](https://www.programiz.com/python-programming/methods/string/find) | Returns the Highest Index of Substring |
| [format()](https://www.programiz.com/python-programming/methods/string/format) | Formats string into nicer output |
| [index()](https://www.programiz.com/python-programming/methods/string/index) | Returns index of substring |
| [isalnum()](https://www.programiz.com/python-programming/methods/string/isalnum) | Checks alphanumeric character |
| [isalpha()](https://www.programiz.com/python-programming/methods/string/isalpha) | Checks if all characters are alphabets |
| [isdecimal()](https://www.programiz.com/python-programming/methods/string/isdecimal) | Checks decimal characters |
| [isdigit()](https://www.programiz.com/python-programming/methods/string/isdigit) | Checks digit characters |
| [isidentifier()](https://www.programiz.com/python-programming/methods/string/isidentifier) | Checks for valid identifier |
| [islower()](https://www.programiz.com/python-programming/methods/string/islower) | Checks if all alphabets in a string are lowercase |
| [isnumeric()](https://www.programiz.com/python-programming/methods/string/isnumeric) | Checks numeric characters |
| [isprintable()](https://www.programiz.com/python-programming/methods/string/isprintable) | Checks printable character |
| [isspace()](https://www.programiz.com/python-programming/methods/string/isspace) | Checks whitespace characters |
| [istitle()](https://www.programiz.com/python-programming/methods/string/istitle) | Checks for titlecased string |
| [isupper()](https://www.programiz.com/python-programming/methods/string/isupper) | Returns if all characters are uppercase characters |
| [join()](https://www.programiz.com/python-programming/methods/string/join) | Returns a concatenated string |
| [ljust()](https://www.programiz.com/python-programming/methods/string/ljust) | Returns left-justified string of given width |
| [rjust()](https://www.programiz.com/python-programming/methods/string/rjust) | Returns right-justified string of given width |
| [lower()](https://www.programiz.com/python-programming/methods/string/lower) | Returns lowercased string |
| [upper()](https://www.programiz.com/python-programming/methods/string/upper) | Returns uppercased string |
| [swapcase()](https://www.programiz.com/python-programming/methods/string/swapcase) | Swap uppercase characters to lowercase; vice versa |
| [lstrip()](https://www.programiz.com/python-programming/methods/string/lstrip) | Removes leading characters |
| [rstrip()](https://www.programiz.com/python-programming/methods/string/rstrip) | Removes trailing characters |
| [strip()](https://www.programiz.com/python-programming/methods/string/strip) | Removes both leading and trailing characters |
| [partition()](https://www.programiz.com/python-programming/methods/string/partition) | Returns a tuple |
| [maketrans()](https://www.programiz.com/python-programming/methods/string/maketrans) | Returns a translation table |
| [rpartition()](https://www.programiz.com/python-programming/methods/string/rpartition) | Returns a tuple |
| [translate()](https://www.programiz.com/python-programming/methods/string/translate) | Returns mapped charactered string |
| [replace()](https://www.programiz.com/python-programming/methods/string/replace) | Replaces substring inside |
| [rfind()](https://www.programiz.com/python-programming/methods/string/rfind) | Return the highest index in the string where substring sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure. |
| [rindex()](https://www.programiz.com/python-programming/methods/string/rindex) | Like rfind() but raises ValueError when the substring sub is not found. |
| [split()](https://www.programiz.com/python-programming/methods/string/split) | Splits string from left |
| [rsplit()](https://www.programiz.com/python-programming/methods/string/rsplit) | Splits string from right |
| [splitlines()](https://www.programiz.com/python-programming/methods/string/splitlines) | Splits string at line boundaries |
| [startswith()](https://www.programiz.com/python-programming/methods/string/startswith) | Checks if string starts with the specified string |
| [title()](https://www.programiz.com/python-programming/methods/string/title) | Returns a title cased string |
| [zfill()](https://www.programiz.com/python-programming/methods/string/zfill) | Returns a copy of the string padded with zeros |
| [format\_map()](https://www.programiz.com/python-programming/methods/string/format_map) | Formats the string using dictionary |

## list

A list is a container data type that acts as a dynamic array, or it’s a sequence that can be indexed into and that can grow and shrink.

List data types are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

A list can be written as a list of comma-separated values between square brackets.

**Example:**

>>> squares = [1, 4, 9, 16, 25]

>>> squares

[1, 4, 9, 16, 25]

A list may be constructed in several ways:

* Using a pair of square brackets to denote the empty list: []
* Using square brackets, separating items with commas: [a], [a, b, c]
* Using a list comprehension: [x for x in iterable]
* Using the type constructor: list() or list(iterable)

The constructor builds a list whose items are the same and in the same order as iterable’s items. iterable may be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a list, a copy is made and returned, similar to iterable[:]. For example, list('abc') returns ['a', 'b', 'c'] and list( (1, 2, 3) ) returns [1, 2, 3]. If no argument is given, the constructor creates a new empty list, [].

Many other operations also produce list, including the sorted() built-in.

Let us take a look at what all can be done with list data types:

1. Like a string (and all other built-in sequence type), a list can be **indexed and sliced**:

**Example:**

>>> squares[0] # indexing returns the item

1

>>> squares[-1]

25

>>> squares[-3:] # slicing returns a new list

[9, 16, 25]

1. All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

**Example:**

>>> squares[:]

[1, 4, 9, 16, 25]

1. List also supports operations like **concatenation**:

**Example:**

>>> squares + [36, 49, 64, 81, 100]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

1. Unlike a string, which is immutable, a list is a **mutable** type, i.e. it is possible to **change its content**:

**Example:**

>>> cubes = [1, 8, 27, 65, 125] # something's wrong here

>>> 4 \*\* 3 # the cube of 4 is 64, not 65!

64

>>> cubes[3] = 64 # replace the wrong value

>>> cubes

[1, 8, 27, 64, 125]

1. You can also **add new items** at the end of the list, by using the append() method:

**Example:**

>>> cubes.append(216) # add the cube of 6

>>> cubes.append(7 \*\* 3) # and the cube of 7

>>> cubes

[1, 8, 27, 64, 125, 216, 343]

1. **Assignment to slices** is also possible, and this can even change the size of the list or clear it entirely:

**Example:**

>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

>>> letters

['a', 'b', 'c', 'd', 'e', 'f', 'g']

>>> # replace some values

>>> letters[2:5] = ['C', 'D', 'E']

>>> letters

['a', 'b', 'C', 'D', 'E', 'f', 'g']

>>> # now remove them

>>> letters[2:5] = []

>>> letters

['a', 'b', 'f', 'g']

>>> # clear the list by replacing all the elements with an empty list

>>> letters[:] = []

>>> letters

[]

1. The built-in function **len()** also applies to list.

**Example:**

>>> letters = ['a', 'b', 'c', 'd']

>>> len(letters)

4

1. It is possible to **nest list** (create list containing other list).

**Example:**

>>> a = ['a', 'b', 'c']

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

>>> x = [a, n]

>>> x

[['a', 'b', 'c'], [1, 2, 3]]

>>> x[0]

['a', 'b', 'c']

>>> x[0][1]

'b'

### list Methods

List implements all of the common and mutable sequence operations as indicated in the built-in methods table discussed earlier. List also provides the following additional method:

|  |  |
| --- | --- |
| Methods | Description |
| append() | Add an item to the end of the list. |
| extend() | Extend the list by appending all the items from another list. |
| insert() | Insert an item at a given position. |
| remove(x) | Remove an item from a list. |
| pop() | Remove the item at the given position in the list, and return it. |
| clear() | Remove all items from the list. |
| index() | Returns smallest index of element in list.  The optional argumentsstart and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument. |
| count() | Return the number of times an item appears in a list. |
| sort() | Sort the items of the list in place (the arguments can be used for sort customization). |
| reverse() | Reverse the elements of the list in place. |
| copy() | Return a shallow copy of the list. |

**Example** (*An example that uses most of the list methods.*)**:**

>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']

>>> fruits.count('apple')

2

>>> fruits.count('tangerine')

0

>>> fruits.index('banana')

3

>>> fruits.index('banana', 4) # Find next banana starting a position 4

6

>>> fruits.reverse()

>>> fruits

['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']

>>> fruits.append('grape')

>>> fruits

['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']

>>> fruits.sort()

>>> fruits

['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']

>>> fruits.pop()

'pear'

**Note**: *You might have noticed that methods like insert, remove or sort that only modify the list have no return value printed – they return the default None. This is a design principle for all mutable data structures in Python.*

## tuple

Tuple data types are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuple produced by the enumerate() built-in). Tupleis also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a set or dict instance).

Tuple may be constructed in a number of ways:

* Using a pair of parentheses to denote the empty tuple: ()
* Using a trailing comma for a singleton tuple: a, or (a,)
* Separating items with commas: a, b, c or (a, b, c)
* Using the tuple() built-in: tuple() or tuple(iterable)

The constructor builds a tuple whose items are the same and in the same order as *iterable’s* items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For example, tuple('abc') returns ('a', 'b', 'c') and tuple( [1, 2, 3] ) returns (1, 2, 3). If no argument is given, the constructor creates a new empty tuple, ().

***Note:****It is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, f(a, b, c) is a function call with three arguments, while f((a, b, c)) is a function call with a 3-tuple as the sole argument.*

For heterogeneous collections of data where access by name is clearer than access by index, collections.namedtuple() may be a more appropriate choice than a simple tuple object.

### tuple Methods

Apart from the tuple methods listed in the built-in methods, the table below lists the methods that can take tuple as a parameter and perform some task.

|  |  |
| --- | --- |
| Method | Description |
| count() | Returns occurrences of element in a tuple |
| index() | Returns smallest index of element in tuple |

## dictionary

Another useful data type built into Python is the dictionary. Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionary is indexed by keys, which can be any immutable type.

* A string and numbers can always be keys.
* Tuple can be used as a key if it contains only string, numbers, or tuple; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.
* List can’t be used as keys, since list can be modified in place using index assignments, slice assignments, or methods like append() and extend().

It is best to think of a dictionary as an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionary is written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d.keys()) on a dictionary returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just use sorted(d.keys()) instead). To check whether a single key is in the dictionary, use the in keyword.

**Example:**

>>> tel = {'jack': 4098, 'sape': 4139}

>>> tel['guido'] = 4127

>>> tel

{'sape': 4139, 'guido': 4127, 'jack': 4098}

>>> tel['jack']

4098

>>> del tel['sape']

>>> tel['irv'] = 4127

>>> tel

{'guido': 4127, 'irv': 4127, 'jack': 4098}

>>> list(tel.keys())

['irv', 'guido', 'jack']

>>> sorted(tel.keys())

['guido', 'irv', 'jack']

>>> 'guido' in tel

True

>>> 'jack' not in tel

False

The dict() constructor builds dictionary directly from sequences of key-value pairs.

**Example:**

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

{'sape': 4139, 'jack': 4098, 'guido': 4127}

In addition, dict comprehensions can be used to create dictionary from arbitrary key and value expressions.

**Example:**

>>> {x: x\*\*2 for x in (2, 4, 6)}

{2: 4, 4: 16, 6: 36}

When the keys are simple string, it is sometimes easier to specify pairs using keyword arguments.

**Example:**

>>> dict(sape=4139, guido=4127, jack=4098)

{'sape': 4139, 'jack': 4098, 'guido': 4127}

### Mapping Types

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary.

A dictionary’s keys are almost arbitrary values. Values that are not hashable, that is, values containing list, dictionary, or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry.

**Note:***Since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.*

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces or by the dict constructor.

class dict(\*\*kwarg), class dict(mapping, \*\*kwarg), or class dict(iterable, \*\*kwarg) return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

**Example:**

The following examples all return a dictionary equal to {"one": 1, "two": 2, "three": 3}:

>>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

>>> a == b == c == d == e

True

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

The table below lists the operations that the dictionary data type support (and therefore, custom mapping types should support too):

|  |  |
| --- | --- |
| Operation | Description |
| len(d) | Return the number of items in the dictionary d. |
| d[key] | Return the item of d with key key. Raises a KeyError if key is not in the map.  If a subclass of dict defines a method \_\_missing\_\_() and key is not present, the d[key] operation calls that method with the key key as argument. The d[key] operation then returns or raises whatever is returned or raised by the \_\_missing\_\_(key) call. No other operations or methods invoke \_\_missing\_\_(). If \_\_missing\_\_() is not defined, KeyError is raised. \_\_missing\_\_() must be a method; it cannot be an instance variable: |
| d[key] = value | Set d[key] to value. |
| del d[key] | Remove d[key] from d. Raises a KeyError if key is not in the map. |
| key in d | Return True if d has a key key, else False. |
| key not in d | Equivalent to not key in d. |
| iter(d) | Return an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()). |
| clear() | Remove all items from the dictionary. |
| copy() | Return a shallow copy of the dictionary. |
| classmethod fromkeys(seq[, value]) | Create a new dictionary with keys from seq and values set to value.  fromkeys() is a class method that returns a new dictionary. value defaults to None. |
| get(key[, default]) | Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError. |
| items() | Return a new view of the dictionary’s items ((key, value) pairs). |
| keys() | Return a new view of the dictionary’s keys. |
| pop(key[, default]) | If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised. |
| popitem() | Remove and return an arbitrary (key, value) pair from the dictionary.  popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling popitem() raises a KeyError. |
| setdefault(key[, default]) | If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None. |
| update([other]) | Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.  update() accepts either another dictionary object or an iterable of key/value pairs (as tupleq or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: d.update(red=1, blue=2). |
| values() | Return a new view of the dictionary’s values. |

**Note:***Dictionaries compare equal if and only if they have the same (key, value) pairs. Order comparisons (‘<’, ‘<=’, ‘>=’, ‘>’) raise TypeError.*

### dictionary View Objects

The objects returned by dict.keys(), dict.values() and dict.items() are view objects. They provide a dynamic view on the dictionary’s entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

* len(dictview): Return the number of entries in the dictionary.
* iter(dictview): Return an iterator over the keys, values or items (represented as tuple of (key, value)) in the dictionary.

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of (value, key) pairs using zip():pairs = zip(d.values(), d.keys()). Another way to create the same list is pairs = [(v, k) for (k, v) in d.items()].

Iterating views while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

* x in dictview:Return True if x is in the underlying dictionary’s keys, values or items (in the latter case, x should be a (key, value) tuple).

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class collections.abc.Set are available (for example, ==, <, or ^).

**Example:**

>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}

>>> keys = dishes.keys()

>>> values = dishes.values()

>>> # iteration

>>> n = 0

>>> for val in values:

... n += val

>>> print(n)

504

>>> # keys and values are iterated over in the same order

>>> list(keys)

['eggs', 'bacon', 'sausage', 'spam']

>>> list(values)

[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes

>>> del dishes['eggs']

>>> del dishes['sausage']

>>> list(keys)

['spam', 'bacon']

>>> # set operations

>>> keys & {'eggs', 'bacon', 'salad'}

{'bacon'}

>>> keys ^ {'sausage', 'juice'}

{'juice', 'sausage', 'bacon', 'spam'}

### dictionary Methods

The table below lists the dictionary methods in addition to the dictionary methods listed in the built-in methods table shown earlier in the chapter.

|  |  |
| --- | --- |
| Methods | Description |
| [clear()](https://www.programiz.com/python-programming/methods/dictionary/clear) | Removes all items |
| [copy()](https://www.programiz.com/python-programming/methods/dictionary/copy) | Returns shallow copy of a dictionary |
| [fromkeys()](https://www.programiz.com/python-programming/methods/dictionary/fromkeys) | Creates dictionary from given sequence |
| [get()](https://www.programiz.com/python-programming/methods/dictionary/get) | Returns value of the key |
| [items()](https://www.programiz.com/python-programming/methods/dictionary/items) | Returns view of dictionary's (key, value) pair |
| [keys()](https://www.programiz.com/python-programming/methods/dictionary/keys) | Returns view object of all keys |
| [popitem()](https://www.programiz.com/python-programming/methods/dictionary/popitem) | Returns and removes element from dictionary |
| [setdefault()](https://www.programiz.com/python-programming/methods/dictionary/setdefault) | Inserts key with a value if key is not present |
| [pop()](https://www.programiz.com/python-programming/methods/dictionary/pop) | Removes and returns element having given key |
| [values()](https://www.programiz.com/python-programming/methods/dictionary/values) | Returns view of all values in dictionary |
| [update()](https://www.programiz.com/python-programming/methods/dictionary/update) | Updates the dictionary |

## set

A set is an unordered collection with no duplicate elements. Basic uses of a set include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create a set.

**Note**: *To create an empty set you have to use set(), not set{}; the latter creates an empty dictionary.*

**Example 1:**

>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

>>> print(basket) # show that duplicates have been removed

{'orange', 'banana', 'pear', 'apple'}

>>> 'orange' in basket # fast membership testing

True

>>> 'crabgrass' in basket

False

**Example 2:**

>>> # Demonstrate set operations on unique letters from two words

...

>>> a = set('abracadabra')

>>> b = set('alacazam')

>>> a # unique letters in a

{'a', 'r', 'b', 'c', 'd'}

>>> a - b # letters in a but not in b

{'r', 'd', 'b'}

>>> a | b # letters in a or b or both

{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

>>> a & b # letters in both a and b

{'a', 'c'}

>>> a ^ b # letters in a or b but not both

{'r', 'd', 'b', 'm', 'z', 'l'}

Similar to list comprehensions, set comprehensions are also supported:

**Example:**

>>> a = {x for x in 'abracadabra' if x not in 'abc'}

>>> a

{'r', 'd'}

**Note:***Like other collections, setsupportsx in set, len(set), and for x in set. Being an unordered collection, set do not record element position or order of insertion. Accordingly, set do not support indexing, slicing, or other sequence-like behavior.*

### Built-in set Types

There are currently two built-in set types:

* set: The set type is mutable, which means that the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set.
* frozenset: The frozenset type is immutable and hashable, which means that its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

**Note:***Non-empty set (not frozenset) can be created by placing a comma-separated list of elements within braces, for example: {'jack', 'sjoerd'}, in addition to the set constructor.*

The constructors for both classes class set([iterable])and class frozenset([iterable])work in the same way. They return a new set or frozenset object whose elements are taken from *iterable*. The elements of a set must be *hashable*. To represent a set of set, the inner set must be frozensetobjects. If *iterable* is not specified, a new empty set is returned.

The table below lists the methods that you can use for the set and frozensettypes:

|  |  |
| --- | --- |
| Method | Description |
| isdisjoint() | ReturnsTrue if the set has no elements in common with another set. Sets are disjoint if and only if their intersection is the empty set. |
| issubset() | Checks if a set is subset of another set |
| issuperset() | Tests whether the set is a superset of other |
| union() | Returns a new set with elements from the set and all others. |
| intersection() | Returns a new set with elements common to the set and all others. |
| difference() | Returns a new set with elements in the set that are not in the others. |
| symmetric\_difference(other) | Returnssymmetric difference |
| copy() | Returns a new set with a shallow copy of s. |

**Note:***The non-operator versions of union(), intersection(), difference(), and symmetric\_difference(), issubset(), and issuperset() methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be a set. This precludes error-prone constructions like set('abc') & 'cbs' in favor of the more readable set('abc').intersection('cbs').*

Both set and frozenset support set to set comparisons.

* Two set are equal if and only if every element of each set is contained in the other (each is a subset of the other).
* A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal).
* A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of set are compared to instances of frozenset based on their members. For example, set('abc') == frozenset('abc') returns True and so does set('abc') in set([frozenset('abc')]).

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint set are not equal and are not subsets of each other, so all of the following return False: a<b, a==b, or a>b.

Since set only define partial ordering (subset relationships), the output of the list.sort() method is undefined for lists of set.

Set elements, like dictionary keys, must be hashable.

Binary operations that mix set instances with frozenset return the type of the first operand. For example: frozenset('ab') | set('bc') returns an instance of frozenset.

The following table lists operations available for set that do not apply to immutable instances of frozenset:

|  |  |
| --- | --- |
| Operation | Description |
| update() | Updates the set, adding elements from all other set. |
| intersection\_update() | Updates calling set with intersection of set. |
| difference\_update() | Updates the set, removing elements found in others. |
| symmetric\_difference\_update() | Updates the set, keeping only elements found in either set, but not in both. |
| add() | Adds element to the set. |
| remove() | Removes element from the set. Raises KeyError if element is not contained in the set. |
| discard() | Removes element from the set if it is present. |
| pop() | Removes and returns an arbitrary element from the set. Raises KeyError if the set is empty. |
| clear() | Remove all elements from the set. |

**Note:**

* *The non-operator versions of the update(), intersection\_update(), difference\_update(), and symmetric\_difference\_update() methods will accept any iterable as an argument.*
* *The argument to the \_\_contains\_\_(), remove(), and discard() methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from the argument.*

## boolean

Boolean values are the two constant objects False and True. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function bool() can be used to convert any value to a Boolean, if the value can be interpreted as a truth value.

They are written as False and True, respectively.

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below.

By default, an object is considered true unless its class defines either a \_\_bool\_\_() method that returns False or a \_\_len\_\_() method that returns zero, when called with the object. [1] Here are most of the built-in objects considered false:

* Constants defined to be false: None and False.
* Zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
* Empty sequences and collections: '', (), [], {}, set(), range(0)

Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated.

**Note:***Important exception: the Boolean operations or and and always return one of their operands.*

## Data Type Conversions

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

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

|  |  |
| --- | --- |
| Function | Description |
| int(x [,base]) | Converts x to an integer. The base specifies the base if x is a string. |
| float(x) | Converts x to a floating-point number. |
| complex(real [,imag]) | Creates a complex number. |
| str(x) | Converts object x to a string representation. |
| repr(x) | Converts object x to an expression string. |
| eval(str) | Evaluates a string and returns an object. |
| tuple(s) | Converts s to a tuple. |
| list(s) | Converts s to a list. |
| set(s) | Converts s to a set. |
| dict(d) | Creates a dictionary. d must be a sequence of (key,value) tuple. |
| frozenset(s) | Converts s to a frozen set. |
| chr(x) | Converts an integer to a character. |
| unichr(x) | Converts an integer to a Unicode character. |
| ord(x) | Converts a single character to its integer value. |
| hex(x) | Converts an integer to a hexadecimal string. |
| oct(x) | Converts an integer to an octal string. |

# Chapter 06 – Functions of Python

## What are Functions?

Python functions are techniques used to combine a set of statements within a program. Functions also let programmers compute a result-value and give parameters that serve as function inputs that may change each time the code runs. Functions prove to be a useful tool when the operations are coded in it and can be used in a variety of scenarios.

Functions are an alternative method of cutting-and-pasting codes, rather than typing redundant copies of the same instruction or operation; which further reduces the future work for programmers. They are the most basic structure of a program, and so Python provides this technique for code re-use.

Functions help in:

* Maximizing code reusability
* Minimizing redundancy
* Procedural decomposition
* Make programs simpler to read and understand

Function objects are created by function definitions and the only operation on a function object is to call it. Let’s take a look at how a function is defined and called.

## Defining a Function

A function definition defines a user-defined function object. A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

In Python a function is defined using the def keyword.

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

* Function blocks begin with the keyword def followed by the function name and parentheses ( ( ) ).
* Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
* The first statement of a function can be an optional statement - the documentation string of the function or docstring.
* The code block within every function starts with a colon (:) and is indented.
* The statement return[expression]exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

**Example:**

>>>def my\_function():

... print("Hello from a function")

The keyword [def](https://docs.python.org/3/reference/compound_stmts.html#def) introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or docstring.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function.

Let’s now take a look at how a function is called.

## Calling a Function

The function definition does not execute the function body; this gets executed only when the function is called.

Functions can be called in the following ways.

* By providing the mandatory argument only
* By providing one of the optional argument
* By giving all the arguments

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

**Example:**

>>>my\_function()

Hello from a function

Function bodies can contain one or more return statement. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller. If the return statement is without an expression, the special value None is returned. If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value "None" will be returned.

## Function Scope

### Scope of Variables

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

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

* Global variables
* Local variables

### Global vs. Local variables

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

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

**Example:**

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

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

print ("Inside the function local total : ", total)

return total;

# Now you can call sum function

sum( 10, 20 );

print ("Outside the function global total : ", total)

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

Inside the function local total : 30

Outside the function global total : 0

## Function Arguments

In Python, it is also possible to define functions with a variable number of arguments. There are three forms, which can be combined: Default argument values, keyword arguments, and arbitrary arguments list.

### Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. Example:

def ask\_ok(prompt, retries=4, reminder='Please try again!'):

while True:

ok = input(prompt)

if ok in ('y', 'ye', 'yes'):

return True

if ok in ('n', 'no', 'nop', 'nope'):

return False

retries = retries - 1

if retries < 0:

raise ValueError('invalid user response')

print(reminder)

This function can be called in several ways:

* By giving only the mandatory argument:  
  ask\_ok('Do you really want to quit?')
* giving one of the optional arguments:  
  ask\_ok('OKtooverwritethefile?',2)
* or even giving all arguments:   
  ask\_ok('OKtooverwritethefile?',2,'Comeon,onlyyesorno!')

This example also introduces the [in](https://docs.python.org/3/reference/expressions.html#in) keyword that tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the defining scope, so that the out of the following example is 5.

**Example:**

i = 5

def f(arg=i):

print(arg)

i = 6

f()

**Important warning:** The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. The function in the following example accumulates the arguments passed to it on subsequent calls:

**Example:**

def f(a, L=[]):

L.append(a)

return L

print(f(1))

print(f(2))

print(f(3))

This will print

[1]

[1, 2]

[1, 2, 3]

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

def f(a, L=None):

if L is None:

L = []

L.append(a)

return L

### Keyword Arguments

Functions can also be called using keyword arguments of the form kwarg=value.

A keyword argument is an argument preceded by an identifier (e.g. name=) in a function call or passed as a value in a dictionary preceded by \*\*.

**Example:**

def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):

print("-- This parrot wouldn't", action, end=' ')

print("if you put", voltage, "volts through it.")

print("-- Lovely plumage, the", type)

print("-- It's", state, "!")

In the example, the function accepts one required argument (voltage) and three optional arguments (state, action, and type).

This function can be called in any of the following ways:

**Example:**

parrot(1000) #1 positional argument

parrot(voltage=1000) #1 keyword argument

parrot(voltage=1000000, action='VOOOOOM') #2 keyword arguments

parrot(action='VOOOOOM', voltage=1000000) #2 keyword arguments

parrot('a million', 'bereft of life', 'jump') #3 positional arguments

parrot('a thousand', state='pushing up the daisies') #1 positional, 1 keyword

However, all the following calls would be invalid:

**Example:**

parrot() #required argument missing

parrot(voltage=5.0, 'dead') #non-keyword argument after a keyword argument

parrot(110, voltage=220) #duplicate value for the same argument

parrot(actor='John Cleese') #unknown keyword argument

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. actoris not a valid argument for the parrotfunction), and their order is not important. This also includes non-optional arguments   
(e.g. parrot(voltage=1000) is valid too). No argument may receive a value more than once. Here’s an example that fails due to this restriction:

**Example:**

>>> def function(a):

... pass

...

>>> function(0, a=0)

Traceback (most recent call last):

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

TypeError: function() got multiple values for keyword argument 'a'

When a final formal parameter of the form \*\*name is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form \*name, which receives a tuple containing the positional arguments beyond the formal parameter list. (\*name must occur before \*\*name.)

**Example**(if we define a function as shown below)**:**

def cheeseshop(kind, \*arguments, \*\*keywords):

print("-- Do you have any", kind, "?")

print("-- I'm sorry, we're all out of", kind)

for arg in arguments:

print(arg)

print("-" \* 40)

for kw in keywords:

print(kw, ":", keywords[kw])

It could be called as shown in the example below:

**Example:**

cheeseshop("Limburger", "It's very runny, sir.",

"It's really very, VERY runny, sir.",

shopkeeper="Michael Palin",

client="John Cleese",

sketch="Cheese Shop Sketch")

And, it will print the output as shown below:

-- Do you have any Limburger ?

-- I'm sorry, we're all out of Limburger

It's very runny, sir.

It's really very, VERY runny, sir.

----------------------------------------

shopkeeper : Michael Palin

client : John Cleese

sketch : Cheese Shop Sketch

***Note:*** *The order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.*

### Arbitrary Argument Lists

A function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.

**Example:**

def write\_multiple\_items(file, separator, \*args):

file.write(separator.join(args))

Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the \*args parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

**Example:**

>>> def concat(\*args, sep="/"):

... return sep.join(args)

...

>>> concat("earth", "mars", "venus")

'earth/mars/venus'

>>> concat("earth", "mars", "venus", sep=".")

'earth.mars.venus'

### Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in [range()](https://docs.python.org/3/library/stdtypes.html#range)function expects separate start and stop arguments. If they are not available separately, write the function call with the \*-operator to unpack the arguments out of a list or tuple.

**Example:**

>>> list(range(3, 6)) # normal call with separate arguments

[3, 4, 5]

>>> args = [3, 6]

>>> list(range(\*args)) #call with arguments unpacked from a list

[3, 4, 5]

In the same fashion, dictionaries can deliver keyword arguments with the \*\*-operator.

**Example:**

>>> def parrot(voltage, state='a stiff', action='voom'):

... print("-- This parrot wouldn't", action, end=' ')

... print("if you put", voltage, "volts through it.", end=' ')

... print("E's", state, "!")

...

>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}

>>> parrot(\*\*d)

-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !

### Difference between Arguments and Parameters

Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it. Parameters define what types of arguments a function can accept.

**Example***(given the function definition)*:

def func(foo, bar=None, \*\*kwargs):

pass

foo, bar and kwargs are parameters of func. However, when calling func,

func(42, bar=314, extra=somevar)

the values 42, 314, and somevar are arguments.

## Function Objects

A function object refers to the fact that Python treats functions first class objects. This allows Python to use certain techniques from functional programming, particularly the passing of functions in ways similar to how you would pass any other object like a String or an Array

## The Anonymous Functions

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

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression

## Lambda Functions

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression lambda arguments: expression yields a function object. The unnamed object behaves like a function object defined with:

def <lambda>(arguments):

return expression

Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

**Example**:

# Program to show the use of lambda functions

double = lambda x: x \* 2

# Output: 10

print(double(5))

The above example uses a lambda expression to double the input value.

Another use is to pass a small function as an argument.

**Example:**

>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

>>> pairs.sort(key=lambda pair: pair[1])

>>> pairs

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

## Recursive Functions

Recursion is the process of defining something in terms of itself.

A real world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

In Python, a function can call other functions as well as it can call itself. A function that can call itself is termed as a recursive function.

**Example** *(a recursive function to find the factorial of an integer)***:**

**Note:***Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1\*2\*3\*4\*5\*6 = 720.*

# An example of a recursive function to

# find the factorial of a number

def calc\_factorial(x):

"""This is a recursive function

to find the factorial of an integer"""

if x == 1:

return 1

else:

return (x \* calc\_factorial(x-1))

num = 4

print("The factorial of", num, "is", calc\_factorial(num))

In the above example, calc\_factorial() is a recursive functions as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

**Example:**

calc\_factorial(4) # 1st call with 4

4 \* calc\_factorial(3) # 2nd call with 3

4 \* 3 \* calc\_factorial(2) # 3rd call with 2

4 \* 3 \* 2 \* calc\_factorial(1) # 4th call with 1

4 \* 3 \* 2 \* 1 # return from 4th call as number=1

4 \* 3 \* 2 # return from 3rd call

4 \* 6 # return from 2nd call

24 # return from 1st call

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of Recursion

* Recursive functions make the code look clean and elegant.
* A complex task can be broken down into simpler sub-problems using recursion.
* Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

* Sometimes the logic behind recursion is hard to follow through.
* Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
* Recursive functions are hard to debug.

# Chapter 07 – Working with Modules and Packages

## Modules

### Introduction

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module’s name (as a string) is available as the value of the global variable \_\_name\_\_. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents.

**Example:**

# Fibonacci numbers module

def fib(n): # write Fibonacci series up to n

a, b = 0, 1

while b < n:

print(b, end=' ')

a, b = b, a+b

print()

def fib2(n): # return Fibonacci series up to n

result = []

a, b = 0, 1

while b < n:

result.append(b)

a, b = b, a+b

return result

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

>>> import fibo

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

>>> fibo.fib(1000)

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

>>> fibo.fib2(100)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

>>> fibo.\_\_name\_\_

'fibo'

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

>>> fib = fibo.fib

>>> fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

### Executing Modules as Scripts

When you run a Python module with:

python fibo.py <arguments>

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

if \_\_name\_\_ == "\_\_main\_\_":

import sys

fib(int(sys.argv[1]))

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

$ python fibo.py 50

1 1 2 3 5 8 13 21 34

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

>>> import fibo

>>>

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

### Setting Search Path to Access a Module

When a module named spam is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named spam.py in a list of directories given by the variable sys.path.

sys.path is initialized from these locations:

* The directory containing the input script (or the current directory when no file is specified).
* PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
* The installation-dependent default.

**Note:** *On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is* ***not*** *added to the module search path.*

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

### Importing a Module

The basic import statement (no from clause) is executed in two steps:

1. Find a module, loading and initializing it, if necessary
2. Define a name or names in the local namespace for the scope where the import statement occurs.

**Note:***When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.*

**Note:***Failures in executing the first step may indicate either that the module could not be located, or that an error occurred while initializing the module, which includes execution of the module’s code.*

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:

* If the module name is followed by as, then the name following as is bound directly to the imported module.
* If no other name is specified, and the module being imported is a top level module, the module’s name is bound in the local namespace as a reference to the imported module.
* If the module being imported is not a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly.

The from form uses a slightly more complex process:

1. Find the module specified in the from clause, loading and initializing it if necessary;
2. For each of the identifiers specified in the import clauses:
3. Check if the imported module has an attribute by that name
4. If not, attempt to import a submodule with that name and then check the imported module again for that attribute
5. If the attribute is not found, ImportError is raised.
6. Otherwise, a reference to that value is stored in the local namespace, using the name in the as clause if it is present, otherwise using the attribute name

**Examples:**

import abc # abc imported and bound locally

import abc.bar.baz # abc.bar.baz imported, abc bound locally

import abc.bar.baz as fbb # abc.bar.baz imported and bound as fbb

from abc.bar import baz # abc.bar.baz imported and bound as baz

from abc import attr # abc imported and abc.attr bound as attr

If the list of identifiers is replaced by a star ('\*'), all public names defined in the module are bound in the local namespace for the scope where the import statement occurs.

The public names defined by a module are determined by checking the module’s namespace for a variable named \_\_all\_\_; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in \_\_all\_\_ are all considered public and are required to exist. If \_\_all\_\_ is not defined, the set of public names includes all names found in the module’s namespace which do not begin with an underscore character ('\_'). \_\_all\_\_ should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The wild card form of import — from module import \* — is only allowed at the module level. Attempting to use it in class or function definitions will raise a SyntaxError.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after from you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute from . import mod from a module in the pkg package then you will end up importing pkg.mod. If you execute from ..subpkg2 import mod from within pkg.subpkg1 you will import pkg.subpkg2.mod.

importlib.import\_module() is provided to support applications that determine dynamically the modules to be loaded.

## Packages

### Introduction

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

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

**Example:**

sound/ Top-level package

\_\_init\_\_.py Initialize the sound package

formats/ Subpackage for file format conversions

\_\_init\_\_.py

wavread.py

wavwrite.py

aiffread.py

aiffwrite.py

auread.py

auwrite.py

...

effects/ Subpackage for sound effects

\_\_init\_\_.py

echo.py

surround.py

reverse.py

...

filters/ Subpackage for filters

\_\_init\_\_.py

equalizer.py

vocoder.py

karaoke.py

...

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

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

There are different ways to import a package.

1. Users of the package can import individual modules from the package.

**Example:**

import sound.effects.echo

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

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

1. An alternative way of importing the submodule is:

from sound.effects import echo

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

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

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

from sound.effects.echo import echofilter

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

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

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

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

### Importing \* From a Package

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

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

\_\_all\_\_ = ["echo", "surround", "reverse"]

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

If \_\_all\_\_ is not defined, the statement from sound.effects import \* does not import all submodules from the package sound.effects into the current namespace; it only ensures that the package sound.effects has been imported (possibly running any initialization code in \_\_init\_\_.py) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by \_\_init\_\_.py. It also includes any submodules of the package that were explicitly loaded by previous import statements. Consider this code:

import sound.effects.echo

import sound.effects.surround

from sound.effects import \*

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

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

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

### Intra-package References

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

You can also write relative imports, with the from module import name form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import.

**Example** (*From the surround module, you might use*):

from . import echo

from .. import formats

from ..filters import equalizer

**Note:** *Relative imports are based on the name of the current module. Since the name of the main module is always "\_\_main\_\_", modules intended for use as the main module of a Python application must always use absolute imports.*

### Packages in Multiple Directories

Packages support one more special attribute, \_\_path\_\_. This is initialized to be a list containing the name of the directory holding the package’s \_\_init\_\_.py before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and sub packages contained in the package.

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

# Chapter 08 – Python File I/O

## Introduction

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

## Modes of Files

In Python, a file can be opened in the following modes:

|  |  |
| --- | --- |
| Mode | Description |
| r | Opens a file for reading only. (It's a default mode.) |
| w | Opens a file for writing. (If a file doesn't exist already, then it creates a new file. Otherwise, it's truncate a file.) |
| x | Opens a file for exclusive creation. (Operation fails if a file does not exist in the location.) |
| a | Opens a file for appending at the end of the file without truncating it. (Creates a new file if it does not exist in the location.) |
| T | Opens a file in text mode. (It's a default mode.) |
| b | Opens a file in binary mode. |
| + | Opens a file for updating (reading and writing.) |

## File Handling

### Opening a File

In Python, you use the open() function to open a file. The open() function requires two arguments, file name and file opening mode.

**Syntax:**

file\_object = open(filename [,mode] [,buffering])

In the above syntax, the parameters used are:

* filename: It is the name of the file.
* mode: It tells the program in which mode the file has to be open.
* buffering: Here, if the value is set to zero(0), no buffering will occur while accessing a file, if the value is set to top one (1), line buffering will be performed while accessing a file.

**Example:**

>>>fh = open("hello.txt", "r")

### Read from a File

For reading and writing text data, different text-encoding schemes are used such as ASCII (American Standard Code for Information Interchange), UTF-8 (Unicode Transformation Format), UTF-16.

Once a file is opened using the open() method then it can be read by a method called read().

**Example:**

#read a text file

fh = open("hello.txt","r")

print fh.read()

#To read one line at a time

fh = open("hello".txt", "r")

print fh.readline()

#read a list of lines

fh = open("hello.txt.", "r")

print fh.readlines()

### Writing a File

Similarly, for writing data to files, we have to use open() with 'wt' mode, clearing and overwriting the previous content. Also, we have to use write() function to write into a file.

**Examples:**

* #To write to a file

fh = open("hello.txt","w")

write("Hello World")

fh.close()

* #To write to a file

fh = open("hello.txt", "w")

lines\_of\_text = ["a line of text", "another line of text", "a third line"]

fh.writelines(lines\_of\_text)

fh.close()

By default, in Python - using the system default text encoding files are read/written. Though Python can understand several hundred text-encodings but the most common encoding techniques used are ASCII, Latin-1, UTF-8, UTF-16, etc.

#### Use of with Option

The use of 'with' statement in the example establishes a context in which the file will be used. As the control leaves the 'with' block, the file gets closed automatically.

The problem can be easily solved by using another mode - technique, i.e., the 'x' mode to open a file instead of 'w' mode.

Let's see two examples to differentiate between them.

**Examples:**

with open('filename' , 'wt') as f:

f.write ('Hello, This is sample content.\n')

# This will create an error that the file 'filename' doesn't exist.

with open ('filename.txt' , 'xt') as f:

f.write ('Hello, This is sample content.\n')

**Note***: In binary mode, you should use 'xb' instead of 'xt'*.

### Append to a Module

Here is an example of a program that appends to an existing file:

**Example:**

#append to file

fh = open("Hello.txt", "a")

write("Hello World again")

fh.close

### Closing a Module

In Python, it is not system critical to close all your files after using them, because the file will auto close after Python code finishes execution. You can close a file by using the close() method.

**Example:**

file\_object.close();

try:

# Open a file

fo = open("sample.txt", "wb")

# perform file operations

finally:

# Close opened file

fo.close()

## OS Module

The OS module in python provides functions for interacting with the operating system. OS, comes under Python’s standard utility modules. This module provides a portable way of using operating system dependent functionality. The os and os.path modules include many functions to interact with the file system.

Following are some functions in OS module:

1. **os.name**: This function gives the name of the operating system dependent module imported. The following names have currently been registered: ‘posix’, ‘nt’, ‘os2’, ‘ce’, ‘java’ and ‘riscos’

**Example:**

import os

print(os.name)

1. **os.getcwd()**: The os.getcwd()function, returns the Current Working Directory(CWD) of the file used to execute the code, can vary from system to system.

**Example:**

import os

print(os.getcwd())

# To print absolute path on your system

# os.path.abspath('.')

# To print files and directories in the current directory

# on your system

# os.listdir('.')

1. **os.error:** All functions in this module raise OSError in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system. os.error is an alias for built-in OSError exception.

**Example:**

importos

try:

    # If the file does not exist,

    # then it would throw an IOError

    filename ='GFG.txt'

    f =open(filename, 'rU')

    text =f.read()

    f.close()

# Control jumps directly to here if

#any of the above lines throws IOError.

exceptIOError:

    # print(os.error) will <class 'OSError'>

    print('Problem reading: '+filename)

# In any case, the code then continues with

# the line after the try/except

1. **os.popen():**This method opens a pipe to or from command. The return value can be read or written depending on whether mode is ‘r’ or ‘w’.  
   **Syntax:**

os.popen(command[, mode[, bufsize]])

Parameters mode & bufsize are not necessary parameters, if not provided, default ‘r’ is taken for mode.

importos

fd ="GFG.txt"

# popen() is similar to open()

file=open(fd, 'w')

file.write("Hello")

file.close()

file=open(fd, 'r')

text =file.read()

print(text)

# popen() provides a pipe/gateway and accesses the file directly

file=os.popen(fd, 'w')

file.write("Hello")

# File not closed, shown in next function.

**Note:***Output for popen() will not be shown, there would be direct changes into the file.*

1. **os.close():**Close file descriptor fd. A file opened using open(), can be closed by close()only. But file opened through os.popen(), can be closed with close() or os.close(). If we try closing a file opened with open(), using os.close(), Python would throw TypeError.

**Example:**

import os

fd = "GFG.txt"

file = open(fd, 'r')

text = file.read()

print(text)

os.close(file)

1. **os.rename():**A file old.txt can be renamed to new.txt, using the function os.rename(). The name of the file changes only if, the file exists and user has sufficient privilege permission to change the file.

**Example:**

importos

fd ="GFG.txt"

os.rename(fd,'New.txt')

os.rename(fd,'New.txt')

## Other File Tools: shutil

The shutil module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal.

**Warning:***Warning Even the higher-level file copying functions (shutil.copy(), shutil.copy2()) cannot copy all file metadata.*

*On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.*

### Directory and Files Operations

Let us take a look at the commonly-used directory and file operations:

* shutil.copy

**Syntax:**shutil.copy(src, dst, \*, follow\_symlinks=True)

Copies the file src to the file or directory dst. src and dst should be strings. If dst specifies a directory, the file will be copied into dst using the base filename from src. Returns the path to the newly created file.

If follow\_symlinks is false, and src is a symbolic link, dst will be created as a symbolic link. If follow\_symlinks is true and src is a symbolic link, dst will be a copy of the file src refers to.

copy() copies the file data and the file’s permission mode (see os.chmod()). Other metadata, like the file’s creation and modification times, is not preserved. To preserve all file metadata from the original, use copy2() instead.

* shutil.copy2

**Syntax:** shutil.copy2(src, dst, \*, follow\_symlinks=True)

Identical to copy() except that copy2() also attempts to preserve all file metadata.

When follow\_symlinks is false, and src is a symbolic link, copy2() attempts to copy all metadata from the src symbolic link to the newly-created dst symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, copy2() will preserve all the metadata it can; copy2() never returns failure.

copy2() uses copystat() to copy the file metadata.

* shutil.copyfile  
  **Syntax:**shutil.copyfile(src, dst, \*, follow\_symlinks=True)

Copies the contents (no metadata) of the file named src to a file named dst and return dst. src and dst are path names given as strings. dst must be the complete target file name; look at shutil.copy() for a copy that accepts a target directory path. If src and dst specify the same file, SameFileError is raised.

The destination location must be writable; otherwise, an OSError exception will be raised. If dst already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If follow\_symlinks is false and src is a symbolic link, a new symbolic link will be created instead of copying the file src points to.

### Archiving Operations

High-level utilities to create and read compressed and archived files are also provided in Python. They rely on the zipfile and tarfile modules.

The most-commonly used archiving operation is as follows:

* shutil.get\_archive\_formats()

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple (name,description).

By default [shutil](https://docs.python.org/3/library/shutil.html#module-shutil) provides these formats:

* + zip: ZIP file (if the [zlib](https://docs.python.org/3/library/zlib.html#module-zlib) module is available).
  + tar: uncompressed tar file.
  + gztar: gzip’ed tar-file (if the [zlib](https://docs.python.org/3/library/zlib.html#module-zlib) module is available).
  + bztar: bzip2’ed tar-file (if the [bz2](https://docs.python.org/3/library/bz2.html#module-bz2) module is available).
  + xztar: xz’ed tar-file (if the [lzma](https://docs.python.org/3/library/lzma.html#module-lzma) module is available).

## File Operations: Examples

1. from itertools import islice

f1=open("ravan.txt",'a+')

with open("test.txt",'r') as f:

for line in islice(f,3):

f1.write(line+"\n")

f1.close()

1. count=0

with open('test.txt') as f:

for x in f:

count+=1

print('No of Lines ',count)

1. try:

f=open('data.txt','r+')

f.seek(0)

print(f.read())

#print(f.closed)

finally:

f.close()

print('Now File is closed')

i=10

j=1

try:

k=i/j

print(k)

#finally:

except Exception as ee:

print('No is Divided by Zero please change the value of j')

1. import sys

import os

bufsize = 9825

fsize = os.stat('test.txt').st\_size

iter = 0

lines=2

with open('test.txt') as f:

if bufsize > fsize:

bufsize = fsize-1

data = []

while True:

iter +=1

f.seek(fsize-bufsize\*iter)

data.extend(f.readlines())

if len(data) >= lines or f.tell() == 0:

print(''.join(data[-lines:]))

break

1. def file\_read(fname):

content\_array = []

with open(fname) as f:

#Content\_list is the list that contains the read lines.

for line in f:

content\_array.append(line)

print(content\_array)

file\_read('test.txt')

1. with open('test.txt', 'r') as infile:

fileword=infile.read()

words = fileword.split()

#print(words)

max\_len = len(max(words, key=len))

print([word for word in words if len(word) == max\_len])

1. with open('test.txt') as f:

i=0

for i,l in enumerate(f):

#pass

#print(l)

#print(i)

i+=1

print("Number of lines in the file: ",i)

1. from collections import Counter

with open('test.txt') as f:

s=f.read()

print('Number of Words ',Counter(s.split()))

1. color = ['Red', 'Green', 'White', 'Black', 'Pink','White','Yellow','Aman','Bhushan','White']

with open('jkt.txt', "w") as myfile:

for c in color:

if (c=='White'):

myfile.write("%s\n" % c)

content = open('jkt.txt')

print(content.read())

1. with open('c1.py') as fh1, open('c2.py') as fh2:

with open('ra.txt','a+') as file:

for line1, line2 in zip(fh1, fh2):

file.write(line1+'\n'+line2)

# line1 from abc.txt, line2 from test.txtg

#print(line1+line2)

1. A Python program to assess if a file is closed or not.

f = open('abc.txt','r')

print(f.closed)

f.close()

print(f.closed)

1. #Write a Python program to remove newline characters from a file.

def remove\_newlines(fname):

flist = open(fname).readlines()

print(flist)

return [s.rstrip('\n') for s in flist]

print(remove\_newlines("test.txt"))

1. digit=[1,2,3,4,5,6,7]

for x in digit:

if x==4:

pass

print(x)

else:

print("There is no value left in digit")

print("Out from for loop")

# Chapter 09 – Exception Handling in Python

## Overview of Exception Handling

### About Exceptions

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal. Most exceptions are not handled by programs; however, they result in error messages.

**Example:**

>>> 10 \* (1/0)

Traceback (most recent call last):

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

ZeroDivisionError: division by zero

>>> 4 + spam\*3

Traceback (most recent call last):

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

NameError: name 'spam' is not defined

>>> '2' + 2

Traceback (most recent call last):

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

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

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, [NameError](https://docs.python.org/3/library/exceptions.html#NameError) and [TypeError](https://docs.python.org/3/library/exceptions.html#TypeError). The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

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

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

In Python, exceptions are processed using the following statements:

* try/except: catch the error and recover from exceptions hoist by programmers or Python itself.
* try/finally: Whether exception occurs or not, it automatically performs the clean-up action.
* assert: triggers an exception conditionally in the code.
* raise: manually triggers an exception in the code.

### Standard Exceptions

The table below lists the standard exceptions in Python.

|  |  |
| --- | --- |
| Exception | Description |
| ArithmeticError | Base class for all errors that occur for numeric calculation. |
| AssertionError | Raised in case of failure of the Assert statement. |
| AttributeError | Raised in case of failure of attribute reference or assignment. |
| EnvironmentError | Base class for all exceptions that occur outside the Python environment. |
| EOFError | Raised when there is no input from the input() function and the end of file is reached. |
| Exception | Base class for all exceptions |
| FloatingPointError | Raised when a floating point calculation fails. |
| ImportError | Raised when an import statement fails. |
| IndentationError | Raised when indentation is not specified properly. |
| IndexError | Raised when an index is not found in a sequence. |
| IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
| IOError | Raised for operating system-related errors. |
| KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| KeyError | Raised when the specified key is not found in the dictionary. |
| LookupError | Base class for all lookup errors. |
| NameError | Raised when an identifier is not found in the local or global namespace. |
| NotImplementedError | Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |
| OverflowError | Raised when a calculation exceeds maximum limit for a numeric type. |
| RuntimeError | Raised when a generated error does not fall into any category. |
| StandardError | Base class for all built-in exceptions except StopIteration and SystemExit. |
| StopIteration | Raised when the next() method of an iterator does not point to any object. |
| SyntaxError | Raised when there is an error in Python syntax. |
| SystemError | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |
| SystemExit | Raised by the sys.exit() function. |
| SystemExit | Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit. |
| TypeError | Raised when an operation or function is attempted that is invalid for the specified data type. |
| UnboundLocalError | Raised when trying to access a local variable in a function or method but no value has been assigned to it. |
| ValueError | Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| ZeroDivisionError | Raised when division or modulo by zero takes place for all numeric types. |

### Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause.

Example:

try:

You do your operations here;

......................

except ExceptionType, Argument:

You can print value of Argument here...

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

### Default Exception Handler

Sometimes, you want to catch all errors that could possibly be generated, but usually you don't. In most cases, you want to be as specific as possible; however, there are some situations where it's best to catch all errors.

For example, suppose you are writing an extension module to a web service. You want the error information to output the output web page, and the server to continue to run, if at all possible. But you have no idea what kind of errors you might have put in your code.

In situations like these, you may want to code something like this:

**Example:**

import sys

try:

untrusted.execute()

except: # catch \*all\* exceptions

e = sys.exc\_info()[0]

write\_to\_page( "<p>Error: %s</p>" % e )

Another case is when you want to do something when code fails:

**Example:**

try:

do\_some\_stuff()

except:

rollback()

raise

else:

commit()

By using raise with no arguments, you will re-raise the last exception. A common place to use this would be to roll back a transaction, or undo operations. If it's a matter of cleanup that should be run regardless of success or failure, then you would do.

**Example:**

try:

do\_some\_stuff()

finally:

cleanup\_stuff()

## Handling Exceptions

### Catching Exceptions

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

**Example:**

>>> while True:

... try:

... x = int(input("Please enter a number: "))

... break

... except ValueError:

... print("Oops! That was no valid number. Try again...")

...

The try statement works as follows:

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

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

**Example:**

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

... pass

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class).

**Example:**

class B(Exception):

pass

class C(B):

pass

class D(C):

pass

for cls in [B, C, D]:

try:

raise cls()

except D:

print("D")

except C:

print("C")

except B:

print("B")

The code here prints B, C, D in that order.

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

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

**Example:**

import sys

try:

f = open('myfile.txt')

s = f.readline()

i = int(s.strip())

except OSError as err:

print("OS error: {0}".format(err))

except ValueError:

print("Could not convert data to an integer.")

except:

print("Unexpected error:", sys.exc\_info()[0])

raise

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

**Example:**

for arg in sys.argv[1:]:

try:

f = open(arg, 'r')

except OSError:

print('cannot open', arg)

else:

print(arg, 'has', len(f.readlines()), 'lines')

f.close()

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

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

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

**Example:**

>>> try:

... raise Exception('spam', 'eggs')

... except Exception as inst:

... print(type(inst)) # the exception instance

... print(inst.args) # arguments stored in .args

... print(inst) # \_\_str\_\_ allows args to be printed directly,

... # but may be overridden in exception subclasses

... x, y = inst.args # unpack args

... print('x =', x)

... print('y =', y)

...

<class 'Exception'>

('spam', 'eggs')

('spam', 'eggs')

x = spam

y = eggs

If an exception has arguments, they are printed as the last part (‘detail’) of the message for unhandled exceptions. Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause.

**Example:**

>>> def this\_fails():

... x = 1/0

...

>>> try:

... this\_fails()

... except ZeroDivisionError as err:

... print('Handling run-time error:', err)

...

Handling run-time error: division by zero

### Raising an Exception

The raise statement allows the programmer to force a specified exception to occur. **Example:**

>>> raise NameError('HiThere')

Traceback (most recent call last):

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

NameError: HiThere

The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

raise ValueError # shorthand for 'raise ValueError()'

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the raise statement allows you to re-raise the exception.

**Example:**

>>> try:

... raise NameError('HiThere')

... except NameError:

... print('An exception flew by!')

... raise

...

An exception flew by!

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

NameError: HiThere

### User-deﬁned Exception

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions.

**Example:**

class Error(Exception):

"""Base class for exceptions in this module."""

pass

class InputError(Error):

"""Exception raised for errors in the input.

Attributes:

expression -- input expression in which the error occurred

message -- explanation of the error

"""

def \_\_init\_\_(self, expression, message):

self.expression = expression

self.message = message

class TransitionError(Error):

"""Raised when an operation attempts a state transition that's not

allowed.

Attributes:

previous -- state at beginning of transition

next -- attempted new state

message -- explanation of why the specific transition is not allowed

"""

def \_\_init\_\_(self, previous, next, message):

self.previous = previous

self.next = next

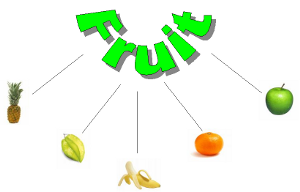
self.message = message

Most exceptions are defined with names that end in “Error,” similar to the naming of the standard exceptions.

# Chapter 10 – Understanding Object-Oriented Concepts in Python

## Object-Oriented Programming

### General Introduction



Though Python is an object-oriented language without fuss or quibble, we have so far intentionally avoided the treatment of object-oriented programming (OOP) in the previous chapters of our Python tutorial. We skipped OOP, because we are convinced that it is easier and more fun to start learning Python without having to know about all the details of object-oriented programming.

But even though we have avoided OOP, it has nevertheless always been present in the exercises and examples of our course. We used objects and methods from classes without properly explaining their OOP background. In this chapter, we will catch up on what has been missing so far. We will provide an introduction into the principles of object oriented programming in general and into the specifics of the OOP approach of Python. OOP is one of the most powerful tools of Python, but nevertheless you don't have to use it, i.e. you can write powerful and efficient programs without it as well.

Though many computer scientists and programmers consider OOP to be a modern programming paradigm, the roots go back to 1960s. The first programming language to use objects was Simula 67. As the name implies, Simula 67 was introduced in the year 1967. A major breakthrough for object-oriented programming came with the programming language Smalltalk in the 1970s.

You will learn to know the four major principles of object-orientation and the way Python deals with them in the next section of this tutorial on object-oriented programming:



* Encapsulation
* Data Abstraction
* Polymorphism
* Inheritance

Before we start with the section on the way OOP is used in Python, we want to give you a general idea about object-oriented programming. For this purpose, we would like to draw your attention to a public library. Let's think about a huge one, like the "British Library" in London or the "New York Public Library" in New York. If it helps, you can imagine the libraries in Paris, Berlin, Ottawa, or Torontoas well. Each of these contain an organized collection of books, periodicals, newspapers, audiobooks, films and so on.

Generally, there are two opposed ways of keeping the stock in a library. You can use a "closed access" method that is the stock is not displayed on open shelves. In this system, trained staff brings the books and other publications to the users on demand. Another way of running a library is open-access shelving, also known as "open shelves". "Open" means open to all the users of the library not only specially trained staff. In this case the books are openly displayed. Imperative languages like C could be seen as open-access shelving libraries. The user can do everything. It's up to the user to find the books and to put them back at the right shelf. Even though this is great for the user, it might lead to serious problems in the long run. For example, some books will be misplaced, so it's hard to find them again. As you may have guessed already, "closed access" can be compared to object oriented programming. The analogy can be seen like this: The books and other publications, which a library offers, are like the data in an object-oriented program. Access to the books is restricted like access to the data is restricted in OOP. Getting or returning a book is only possible via the staff. The staff functions like the methods in OOP, which control the access to the data. So, the data, often called attributes, in such a program can be seen as being hidden and protected by a shell, and it can only be accessed by special functions, usually called methods in the OOP context. Putting the data behind a "shell" is called Encapsulation.   
So a library can be regarded as a class and a book is an instance or an object of this class. Generally speaking, an object is defined by a class. A class is a formal description of how an object is designed, i.e. which attributes and methods it has. These objects are called instances as well. The expressions are in most cases used synonymously. A class should not be confused with an object.

## OOP in Python

### First-Class Everything

Even though we haven't talked about classes and object orientation in previous chapters, we have worked with classes all the time. In fact, everything is a class in Python. Guido van Rossum has designed the language according to the principle "first-class everything". He wrote: "One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." (Blog, The History of Python, February 27, 2009) This means that "everything" is treated the same way, everything is a class: functions and methods are values just like lists, integers or floats. Each of these are instances of their corresponding classes.

**Example:**

>>> x = 42

>>> type(x)

<class 'int'>

>>> y = 4.34

>>> type(y)

<class 'float'>

>>> def f(x):

... return x + 1

...

>>> type(f)

<class 'function'>

>>> import math

>>> type(math)

<class 'module'>

>>>

One of the many integrated classes in Python is the list class, which we have quite often used in our exercises and examples. The list class provides a wealth of methods to build lists, to access and change elements, or to remove elements.

**Example:**

>>> x = [3,6,9]

>>> y = [45, "abc"]

>>> print(x[1])

6

>>> x[1] = 99

>>> x.append(42)

>>> last = y.pop()

>>> print(last)

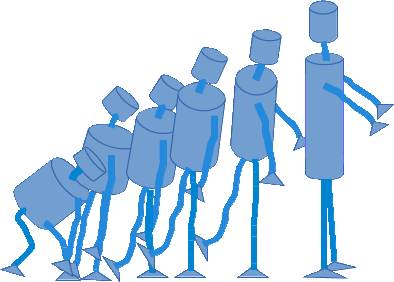
abc

>>>

The variables x and y of the previous example denote two instances of the list class. In simplified terms, we have said so far that "x and y are lists". We will use the terms "object" and "instance" synonymously in the following chapters, as it is often done.

pop and append of the previous example are methods of the list class. pop returns the most upper (or you might think of it as the "rightest") element of the list and removes this element from the list. We will not explain how Python has implemented lists internally. We don't need this information, because the list class provides us with all the necessary methods to access the data indirectly. This means that the encapsulation details are encapsulated. We will learn about encapsulation later.

### A Minimal Class in Python



We will design and use a robot class in Python as an example to demonstrate the most important terms and ideas of object orientation. We will start with the simplest class in Python.

class Robot:

pass

We can realize the fundamental syntactical structure of a class in Python. A class consists of two parts:

* the header, and
* the body

The header usually consists of just one line of code. It begins with the keyword "class" followed by a blank and an arbitrary name for the class. The class name is "Robot" in our case. The class name is followed by a listing of other class names, which are classes from which the defined class inherits from. These classes are called superclasses, base classes or sometimes parent classes. If you look at our example, you will see that this listing of superclasses is not obligatory. You don't have to bother about inheritance and superclasses for the time being. We will introduce them later.

The body of a class consists of an indented block of statements. In our case a single statement, the "pass" statement.

A class object is created, when the definition is left normally, i.e. via the end. This is basically a wrapper around the contents of the namespace created by the class definition.

It's hard to believe, especially for C++ or Java programmers, but we have already defined a complete class with just three words and two lines of code. We are capable of using this class as well:

class Robot:

pass

if \_\_name\_\_ == "\_\_main\_\_":

x = Robot()

y = Robot()

y2 = y

print(y == y2)

print(y == x)

We have created two different robots x and y in our example. Besides this, we have created a reference y2 to y, i.e. y2 is an alias name for y. The output of this example program looks like this:

True

False

### Attributes

Those who have learned already another object-oriented language, will have realized that the terms attributes and properties are usually used synonymously. It may even be used in the definition of an attribute, like Wikipedia does: *"In computing, an attribute is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such."*

Even in normal English usage the words "attribute" and "property" can be used in some cases as synonyms. Both can have the meaning "An attribute, feature, quality, or characteristic of something or someone". Usually an "attribute" is used to denote a specific ability or characteristic which something or someone has, like black hair, no hair, or a quick perception, or "her quickness to grasp new tasks". So, think a while about your outstanding attributes. What about your "outstanding properties"? Great, if one of your strong points is your ability to quickly understand and adapt to new situations! Otherwise, you would not learn Python!

Let's get back to Python: We will learn later that properties and attributes are essentially different things in Python. This subsection of our tutorial is about attributes in Python. So far our robots have no attributes. Not even a name, like it is customary for ordinary robots, isn't it? So, let's implement a name attribute. "type designation", "build year" and so on are easily conceivable as further attributes as well.

Attributes are created inside of a class definition, as we will soon learn. We can dynamically create arbitrary new attributes for existing instances of a class. We do this by joining an arbitrary name to the instance name, separated by a dot ".". In the following example, we demonstrate this by created an attribute for the name and the build year:

>>> class Robot:

... pass

...

>>> x = Robot()

>>> y = Robot()

>>>

>>> x.name = "Marvin"

>>> x.build\_year = "1979"

>>>

>>> y.name = "Caliban"

>>> y.build\_year = "1993"

>>>

>>> print(x.name)

Marvin

>>> print(y.build\_year)

1993

>>>

As we have said before: This is not the way to properly create instance attributes. We introduced this example, because we think that it may help to make the following explanations easier to understand.

If you want to know, what's happening internally: The instances possess dictionaries \_\_dict\_\_, which they use to store their attributes and their corresponding values.

**Example:**

>>> x.\_\_dict\_\_

{'name': 'Marvin', 'build\_year': '1979'}

>>> y.\_\_dict\_\_

{'name': 'Caliban', 'build\_year': '1993'}

Attributes can be bound to class names as well. In this case, each instance will possess this name as well. Watch out, what happens, if you assign the same name to an instance:

>>> class Robot(object):

... pass

...

>>> x = Robot()

>>> Robot.brand = "Kuka"

>>> x.brand

'Kuka'

>>> x.brand = "Thales"

>>> Robot.brand

'Kuka'

>>> y = Robot()

>>> y.brand

'Kuka'

>>> Robot.brand = "Thales"

>>> y.brand

'Thales'

>>> x.brand

'Thales'

If you look at the \_\_dict\_\_ dictionaries, you can see what's happening.

>>> x.\_\_dict\_\_

{'brand': 'Thales'}

>>> y.\_\_dict\_\_

{}

>>>

>>> Robot.\_\_dict\_\_

mappingproxy({'\_\_module\_\_': '\_\_main\_\_', '\_\_weakref\_\_': , '\_\_doc\_\_': None, '\_\_dict\_\_': , 'brand': 'Thales'})

If you try to access y.brand, Python checks first, if "brand" is a key of the y.\_\_dict\_\_ dictionary. If it is not, Python checks, if "brand" is a key of the Robot.\_\_dict\_\_. If so, the value can be retrieved.

If an attribute name is not in included in either of the dictionary, the attribute name is not defined. If you try to access a non-existing attribute, you will raise an AttributeError:

>>> x.energy

Traceback (most recent call last):

File "<stdin>", line 1, in

AttributeError: 'Robot' object has no attribute 'energy'

>>>

By using the function getattr, you can prevent this exception, if you provide a default value as the third argument:

>>> getattr(x, 'energy', 100)

100

>>>

Binding attributes to objects is a general concept in Python. Even function names can be attributed. You can bind an attribute to a function name in the same way, we have done so far.

**Example:**

>>> def f(x):

... return 42

...

>>> f.x = 42

>>> print(f.x)

42

>>>

This can be used as a replacement for the static function variables of C and C++, which are not possible in Python. We use a counter attribute in the following example:

def f(x):

f.counter = getattr(f, "counter", 0) + 1

return "Monty Python"

for i in range(10):

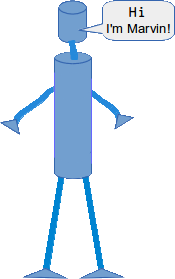
f(i)

print(f.counter)

If you call this little script, it will output 10.

As we have already mentioned, we do not create instance attributes like this. To properly create instances we need methods. You will learn in the following subsection of our tutorial, how you can define methods.

### Methods



We want to demonstrate now, how we can define methods in classes.

Methods in Python are essentially functions in accordance with Guido's saying "first-class everything".

Let's define a function "hi", which takes an object "obj" as an argument and assumes that this object has an attribute "name". We will also define again our basic Robot class.

def hi(obj):

print("Hi, I am " + obj.name + "!")

class Robot:

pass

x = Robot()

x.name = "Marvin"

hi(x)

If we call this code, we get the result:

Hi, I am Marvin!

We will now bind the function “hi” to a class attribute “say\_hi”!

def hi(obj):

print("Hi, I am " + obj.name)

class Robot:

say\_hi = hi

x = Robot()

x.name = "Marvin"

Robot.say\_hi(x)

"say\_hi" is called a method. Usually, it will be called like this:

x.say\_hi()

It is possible to define methods like this, but you shouldn't do it.

The proper way to do it:

* Instead of defining a function outside of a class definition and binding it to a class attribute, we define a method directly inside (indented) of a class definition.
* A method is "just" a function which is defined inside of a class.
* The first parameter is used a reference to the calling instance.
* This parameter is usually called self.
* Self corresponds to the Robot object x.

We have seen that a method differs from a function only in two aspects:

* It belongs to a class, and it is defined within a class
* The first parameter in the definition of a method has to be a reference to the instance, which called the method. This parameter is usually called "self".

As a matter of fact, "self" is not a Python keyword. It's just a naming convention! So C++ or Java programmers are free to call it "this", but this way they are risking that others might have greater difficulties in understanding their code!

Most other object-oriented programming languages pass the reference to the object (self) as a hidden parameter to the methods.

You saw before that the calls Robot.say\_hi(x)". and "x.say\_hi()" are equivalent. "x.say\_hi()" can be seen as an "abbreviated" form, i.e. Python automatically binds it to the instance name. Besides this "x.say\_hi()" is the usual way to call methods in Python and in other object oriented languages.

For a Class C, an instance x of C and a method m of C the following three method calls are equivalent:

* type(x).m(x, ...)
* C.m(x, ...)
* x.m(...)

Before you proceed with the following text, you may mull over the previous example for awhile. Can you figure out, what is wrong in the design?

There is more than one thing about this code, which may disturb you, but the essential problem at the moment is the fact that we create a robot and that after the creation, we shouldn't forget about naming it! If we forget it, say\_hi will raise an error.

We need a mechanism to initialize an instance right after its creation. This is the \_\_init\_\_-method, which we cover in the next section.

### The \_\_init\_\_ Method

We want to define the attributes of an instance right after its creation. \_\_init\_\_ is a method which is immediately and automatically called after an instance has been created. This name is fixed and it is not possible to chose another name. \_\_init\_\_ is one of the so-called magic methods, of which we will get to know some more details later. The \_\_init\_\_ method is used to initialize an instance. There is no explicit constructor or destructor method in Python, as they are known in C++ and Java. The \_\_init\_\_ method can be anywhere in a class definition, but it is usually the first method of a class, i.e. it follows right after the class header.

**Example:**

>>> class A:

... def \_\_init\_\_(self):

... print("\_\_init\_\_ has been executed!")

...

>>> x = A()

\_\_init\_\_ has been executed!

>>>

We add an \_\_init\_\_method to our robot class:

class Robot:

def \_\_init\_\_(self, name=None):

self.name = name

def say\_hi(self):

if self.name:

print("Hi, I am " + self.name)

else:

print("Hi, I am a robot without a name")

x = Robot()

x.say\_hi()

y = Robot("Marvin")

y.say\_hi()

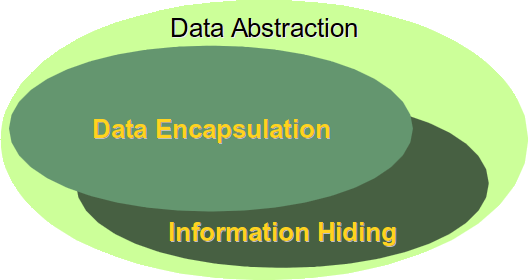
This little program returns the following:

Hi, I am a robot without a name

Hi, I am Marvin

## Data Abstraction, Data Encapsulation, and Information Hiding

### Definitions of Terms



Data Abstraction, Data Encapsulation and Information Hiding are often synonymously used in books and tutorials on OOP. But there is a difference. Encapsulation is seen as the bundling of data with the methods that operate on that data. Information hiding on the other hand is the principle that some internal information or data is "hidden", so that it can't be accidentally changed. Data encapsulation via methods doesn't necessarily mean that the data is hidden. You might be capable of accessing and seeing the data anyway, but using the methods is recommended. Finally, data abstraction is present, if both data hiding and data encapsulation is used. This means data abstraction is the broader term:

**Data Abstraction = Data Encapsulation + Data Hiding**

Encapsulation is often accomplished by providing two kinds of methods for attributes: The methods for retrieving or accessing the values of attributes are called getter methods. Getter methods do not change the values of attributes, they just return the values. The methods used for changing the values of attributes are called setter methods.

We will define now a Robot class with a Getter and a Setter for the name attribute. We will call them get\_name and set\_name accordingly.

class Robot:

def \_\_init\_\_(self, name=None):

self.name = name

def say\_hi(self):

if self.name:

print("Hi, I am " + self.name)

else:

print("Hi, I am a robot without a name")

def set\_name(self, name):

self.name = name

def get\_name(self):

return self.name

x = Robot()

x.set\_name("Henry")

x.say\_hi()

y = Robot()

y.set\_name(x.get\_name())

print(y.get\_name())

Hopefully, it will be easy for you to see, that this program prints the following:

Hi, I am Henry

Henry

Before you go on, you can do a little exercise. You can add an additional attribute "build\_year" with Getters and Setters to the Robot class.

**Example:**

class Robot:

def \_\_init\_\_(self,

name=None,

build\_year=None):

self.name = name

self.build\_year = build\_year

def say\_hi(self):

if self.name:

print("Hi, I am " + self.name)

else:

print("Hi, I am a robot without a name")

if self.build\_year:

print("I was built in " + str(self.build\_year))

else:

print("It's not known, when I was created!")

def set\_name(self, name):

self.name = name

def get\_name(self):

return self.name

def set\_build\_year(self, by):

self.build\_year = by

def get\_build\_year(self):

return self.build\_year

x = Robot("Henry", 2008)

y = Robot()

y.set\_name("Marvin")

x.say\_hi()

y.say\_hi()

The program returns the following output:

Hi, I am Henry

I was built in 2008

Hi, I am Marvin

It's not known, when I was created!

There is still something wrong with our Robot class. The Zen of Python says: "There should be one-- and preferably only one --obvious way to do it." Our Robot class provides us with two ways to access or to change the "name" or the "build\_year" attribute. This can be prevented by using private attributes, which we will explain later.

### \_\_str\_\_ and \_\_repr\_\_Methods

We will have a short break in our treatise on data abstraction for a quick side-trip. We want to introduce two important magic methods "\_\_str\_\_" and "\_\_repr\_\_", which we will need in future examples. In the course of this tutorial, we have already encountered the \_\_str\_\_ method. We had seen that we can depict various data as string by using the str function, which uses "magically" the internal \_\_str\_\_ method of the corresponding data type. \_\_repr\_\_is similar. It also produces a string representation.

>>> l = ["Python", "Java", "C++", "Perl"]

>>> print(l)

['Python', 'Java', 'C++', 'Perl']

>>> str(l)

"['Python', 'Java', 'C++', 'Perl']"

>>> repr(l)

"['Python', 'Java', 'C++', 'Perl']"

>>> d = {"a":3497, "b":8011, "c":8300}

>>> print(d)

{'a': 3497, 'c': 8300, 'b': 8011}

>>> str(d)

"{'a': 3497, 'c': 8300, 'b': 8011}"

>>> repr(d)

"{'a': 3497, 'c': 8300, 'b': 8011}"

>>> x = 587.78

>>> str(x)

'587.78'

>>> repr(x)

'587.78'

>>>

If you apply str or repr to an object, Python is looking for a corresponding method \_\_str\_\_ or \_\_repr\_\_ in the class definition of the object. If the method does exist, it will be called.   
In the following example, we define a class A, having neither a \_\_str\_\_ nor a \_\_repr\_\_ method. We want to see, what happens, if we use print directly on an instance of this class, or if we apply str or repr to this instance:

>>> class A:

... pass

...

>>> a = A()

>>> print(a)

<\_\_main\_\_.A object at 0xb720a64c>

>>> print(repr(a))

<\_\_main\_\_.A object at 0xb720a64c>

>>> print(str(a))

<\_\_main\_\_.A object at 0xb720a64c>

>>> a

<\_\_main\_\_.A object at 0xb720a64c>

>>>

As both methods are not available, Python uses the default output for our object "a".If a class has a \_\_str\_\_ method, the method will be used for an instance x of that class, if either the function str is applied to it or if it is used in a print function. \_\_str\_\_ will not be used, if repr is called, or if we try to output the value directly in an interactive Python shell:

>>> class A:

... def \_\_str\_\_(self):

... return "42"

...

>>> a = A()

>>> print(repr(a))

<\_\_main\_\_.A object at 0xb720a4cc>

>>> print(str(a))

42

>>> a

<\_\_main\_\_.A object at 0xb720a4cc>

Otherwise, if a class has only the \_\_repr\_\_ method and no \_\_str\_\_ method, \_\_repr\_\_ will be applied in the situations, where \_\_str\_\_would be applied, if it were available:

>>> class A:

... def \_\_repr\_\_(self):

... return "42"

...

>>> a = A()

>>> print(repr(a))

42

>>> print(str(a))

42

>>> a

42

A frequently asked question is when to use \_\_repr\_\_ and when \_\_str\_\_. \_\_str\_\_ is always the right choice, if the output should be for the end user or in other words, if it should be nicely printed. \_\_repr\_\_ on the other hand is used for the internal representation of an object. The output of \_\_repr\_\_ should be - if feasible - a string which can be parsed by the python interpreter. The result of this parsing is in an equal object.

This means that the following should be true for an object "o":

o == eval(repr(o))

This is shown in the following interactive Python session:

>>> l = [3,8,9]

>>> s = repr(l)

>>> s

'[3, 8, 9]'

>>> l == eval(s)

True

>>> l == eval(str(l))

True

>>>

We show in the following example with the datetime module that eval can only be applied on the strings created by repr:

>>> import datetime

>>> today = datetime.datetime.now()

>>> str\_s = str(today)

>>> eval(str\_s)

Traceback (most recent call last):

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

File "<string>", line 1

2014-01-26 17:35:39.215144

^

SyntaxError: invalid token

>>> repr\_s = repr(today)

>>> t = eval(repr\_s)

>>> type(t)

<class 'datetime.datetime'>

>>>

We can see that eval(repr\_s) returns again a datetime.datetime object. The String created by str can't be turned into a datetime.datetime object by parsing it.

We will extend our robot class with a repr method. We dropped the other methods to keep this example simple.

**Example:**

class Robot:

def \_\_init\_\_(self, name, build\_year):

self.name = name

self.build\_year = build\_year

def \_\_repr\_\_(self):

return "Robot('" + self.name + "', " + str(self.build\_year) + ")"

if \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

x\_str = str(x)

print(x\_str)

print("Type of x\_str: ", type(x\_str))

new = eval(x\_str)

print(new)

print("Type of new:", type(new))

x\_str has the value Robot('Marvin', 1979). eval(x\_str) converts it again into a Robot instance.

The script returns the following output:   
$ python3 robot\_class5.py

Robot("Marvin",1979)

Type of x\_str: <class 'str'>

Robot("Marvin",1979)

Type of new: <class '\_\_main\_\_.Robot'>

Now it's time to extend our class with a user friendly \_\_str\_\_ method:

class Robot:

def \_\_init\_\_(self, name, build\_year):

self.name = name

self.build\_year = build\_year

def \_\_repr\_\_(self):

return "Robot('" + self.name + "', " + str(self.build\_year) + ")"

def \_\_str\_\_(self):

return "Name: " + self.name + ", Build Year: " + str(self.build\_year)

if \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

x\_str = str(x)

print(x\_str)

print("Type of x\_str: ", type(x\_str))

new = eval(x\_str)

print(new)

print("Type of new:", type(new))

When we start this program, we can see that it is not possible to convert our string x\_str, created via str(x), into a Robot object anymore.

$ python3 robot\_class6.py

Name: Marvin, Build Year: 1979

Type of x\_str: <class 'str'>

Traceback (most recent call last):

File "robot\_class6.py", line 19, in <module>

new = eval(x\_str)

File "<string>", line 1

Name: Marvin, Build Year: 1979

^

SyntaxError: invalid syntax

We show in the following program that x\_repr can still be turned into a Robot object:

class Robot:

def \_\_init\_\_(self, name, build\_year):

self.name = name

self.build\_year = build\_year

def \_\_repr\_\_(self):

return "Robot(\"" + self.name + "\"," + str(self.build\_year) + ")"

def \_\_str\_\_(self):

return "Name: " + self.name + ", Build Year: " + str(self.build\_year)

if \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

x\_repr = repr(x)

print(x\_repr, type(x\_repr))

new = eval(x\_repr)

print(new)

print("Type of new:", type(new))

The output looks like this:

$ python3 robot\_class6b.py

Robot("Marvin",1979) <class 'str'>

Name: Marvin, Build Year: 1979

Type of new: <class '\_\_main\_\_.Robot'>

### Public- Protected- and Private Attributes



Who doesn't know those trigger-happy farmers from films. Shooting as soon as somebody enters their property. This "somebody" has of course neglected the "no trespassing" sign, indicating that the land is private property. Maybe he hasn't seen the sign, maybe the sign is hard to be seen? Imagine a jogger, running the same course five times a week for more than a year, but than he receives a $50 fine for trespassing in the Winchester Fells. Trespassing is a criminal offence in Massachusetts. He was innocent anyway, because the signage was inadequate in the area.4   
  
Even though no trespassing signs and strict laws do protect the private property, some surround their property with fences to keep off unwanted "visitors". Should the fence keep the dog in the yard or the burglar in the street? Choose your fence: Wood panel fencing, post-and-rail fencing, chain-link fencing with or without barbed wire and so on.   
  
We have a similar situation in the design of object-oriented programming languages. The first decision to take is how to protect the data which should be private. The second decision is what to do if trespassing, i.e. accessing or changing private data, occurs. Of course, the private data may be protected in a way that it can't be accessed under no circumstances. This is hardly possible in practice, as we know from the old saying "Where there's a will, there's a way"!   
  


Some owners allow a restricted access to their property. Joggers or hikers may find signs like "Enter at your own risk". A third kind of property might be public property like streets or parks, where it is perfectly legal to be.

We have the same classification again in object-oriented programming:

* Private attributes should only be used by the owner, i.e. inside of the class definition itself.
* Protected (restricted) Attributes may be used, but at your own risk. Essentially, this means that they should only be used under certain conditions.
* Public Attributes can and should be freely used.

Python uses a special naming scheme for attributes to control the accessibility of the attributes. So far, we have used attribute names, which can be freely used inside or outside of a class definition, as we have seen. This corresponds to public attributes of course.

There are two ways to restrict the access to class attributes:

* First, we can prefix an attribute name with a leading underscore "\_". This marks the attribute as protected. It tells users of the class not to use this attribute unless, somebody writes a subclass. We will learn about inheritance and subclassing in the next chapter of our tutorial.
* Second, we can prefix an attribute name with two leading underscores "\_\_". The attribute is now inaccessible and invisible from outside. It's neither possible to read nor write to those attributes except inside of the class definition itself.5

To summarize the attribute types:

|  |  |  |
| --- | --- | --- |
| Naming | Type | Meaning |
| name | Public | These attributes can be freely used inside or outside of a class definition. |
| \_name | Protected | Protected attributes should not be used outside of the class definition, unless inside of a subclass definition. |
| \_\_name | Private | This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside of the class definition itself. |

We want to demonstrate the behaviour of these attribute types with an example class:

class A():

def \_\_init\_\_(self):

self.\_\_priv = "I am private"

self.\_prot = "I am protected"

self.pub = "I am public"

We store this class (attribute\_tests.py) and test its behaviour in the following interactive Python shell:

>>> from attribute\_tests import A

>>> x = A()

>>> x.pub

'I am public'

>>> x.pub = x.pub + " and my value can be changed"

>>> x.pub

'I am public and my value can be changed'

>>> x.\_prot

'I am protected'

>>> x.\_\_priv

Traceback (most recent call last):

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

AttributeError: 'A' object has no attribute '\_\_priv'

>>>

The error message is very interesting. One might have expected a message like "\_\_priv is private". We get the message "AttributeError: 'A' object has no attribute '\_\_priv'" instead, which looks like a "lie". There is such an attribute, but we are told that there isn't. This is perfect information hiding. Telling a user that an attribute name is private, means that we make some information visible, i.e. the existence or non-existence of a private variable.

Our next task consists in rewriting our Robot class. Though we have Getter and Setter methods for the name and the build\_year, we can access the attributes directly as well, because we have defined them as public attributes. Data Encapsulation means, that we should only be able to access private attributes via getters and setters.

We have to replace each occurrence of self.name and self.build\_year by self.\_\_name and self.\_\_build\_year.

The listing of our revised class:

class Robot:

def \_\_init\_\_(self, name=None, build\_year=2000):

self.\_\_name = name

self.\_\_build\_year = build\_year

def say\_hi(self):

if self.\_\_name:

print("Hi, I am " + self.\_\_name)

else:

print("Hi, I am a robot without a name")

def set\_name(self, name):

self.\_\_name = name

def get\_name(self):

return self.\_\_name

def set\_build\_year(self, by):

self.\_\_build\_year = by

def get\_build\_year(self):

return self.\_\_build\_year

def \_\_repr\_\_(self):

return "Robot('" + self.\_\_name + "', " + str(self.\_\_build\_year) + ")"

def \_\_str\_\_(self):

return "Name: " + self.\_\_name + ", Build Year: " + str(self.\_\_build\_year)

if \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

y = Robot("Caliban", 1943)

for rob in [x, y]:

rob.say\_hi()

if rob.get\_name() == "Caliban":

rob.set\_build\_year(1993)

print("I was built in the year " + str(rob.get\_build\_year()) + "!")

We get the following out, if we call this program:

Hi, I am Marvin

I was built in the year 1979!

Hi, I am Caliban

I was built in the year 1993!

Every private attribute of our class has a getter and a setter. There are IDEs for object-oriented programming languages, who automatically provide getters and setters for every private attribute as soon as an attribute is created.

This may look like the following class:

class A():

def \_\_init\_\_(self, x, y):

self.\_\_x = x

self.\_\_y = y

def GetX(self):

return self.\_\_x

def GetY(self):

return self.\_\_y

def SetX(self, x):

self.\_\_x = x

def SetY(self, y):

self.\_\_y = y

There are at least two good reasons against such an approach. First of all not every private attribute needs to be accessed from outside. Second, we will create non-pythonic Code this way, as you will learn soon.

### Destructor

What we said about constructors holds true for destructors as well. There is no "real" destructor, but something similar, i.e. the method \_\_del\_\_. It is called when the instance is about to be destroyed and if there is no other reference to this instance. If a base class has a \_\_del\_\_() method, the derived class's \_\_del\_\_() method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

The following script is an example with \_\_init\_\_ and \_\_del\_\_:

class Robot():

def \_\_init\_\_(self, name):

print(name + " has been created!")

def \_\_del\_\_(self):

print ("Robot has been destroyed")

if \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Tik-Tok")

y = Robot("Jenkins")

z = x

print("Deleting x")

del x

print("Deleting z")

del z

del y

The output of the previous program:

$ python3 del\_example.py

Tik-Tok has been created!

Jenkins has been created!

Deleting x

Deleting z

Robot has been destroyed

Robot has been destroyed

The usage of the \_\_del\_\_method is very problematic. If we change the previous code to personalize the deletion of a robot, we create an error:

class Robot():

def \_\_init\_\_(self, name):

print(name + " has been created!")

def \_\_del\_\_(self):

print (self.name + " says bye-bye!")

if \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Tik-Tok")

y = Robot("Jenkins")

z = x

print("Deleting x")

del x

print("Deleting z")

del z

del y

We get the following output with error messages:

$ python3 del\_example.py

Tik-Tok has been created!

Jenkins has been created!

Deleting x

Deleting z

Exception AttributeError: "'Robot' object has no attribute 'name'" in <bound method Robot.\_\_del\_\_ of <\_\_main\_\_.Robot object at 0xb71da3cc>> ignored

Exception AttributeError: "'Robot' object has no attribute 'name'" in <bound method Robot.\_\_del\_\_ of <\_\_main\_\_.Robot object at 0xb71da36c>> ignored

We are accessing an attribute which doesn't exist anymore. We will learn later, why this is the case.

# Chapter 11 – Advance Python Concepts

## Command Line Arguments

### sys.argv

sys.argv is the list of command line arguments passed to a Python script. argv[0] is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the -c command line option to the interpreter, argv[0] is set to the string '-c'. If no script name was passed to the Python interpreter, argv[0] is the empty string.

### Parsing Command-line Arguments in sys.argv

The getopt module helps scripts to parse the command line arguments in sys.argv. It supports the same conventions as the Unix getopt() function (including the special meanings of arguments of the form ‘-‘ and ‘--‘). Long options similar to those supported by GNU software may be used as well via an optional third argument.

**Note:** *The getopt module is a parser for command line options whose API is designed to be familiar to users of the C getopt()function. Users who are unfamiliar with the C getopt() function or who would like to write less code and get better help and error messages should consider using the argparse module instead.*

The getopt module provides two functions and an exception:

**Functions**

1. getopt.getopt(args, shortopts, longopts=[])

Parses command line options and parameter list. args is the argument list to be parsed, without the leading reference to the running program. Typically, this means sys.argv[1:]. shortopts is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (':'; i.e., the same format that Unix getopt() uses).

**Note*:*** *Unlike GNU getopt(), after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.*

longopts, if specified, must be a list of strings with the names of the long options which should be supported. The leading '--' characters should not be included in the option name. Long options which require an argument should be followed by an equal sign ('='). Optional arguments are not supported. To accept only long options, shortopts should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if longopts is ['foo', 'frob'], the option --fo will match as --foo, but --f will not match uniquely, so GetoptError will be raised.

The return value consists of two elements:

* The first is a list of (option, value) pairs;
* The second is the list of program arguments left after the option list was stripped (this is a trailing slice of args).

Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option'), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

1. getopt.gnu\_getopt(args, shortopts, longopts=[])

This function works like getopt(), except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The getopt() function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is '+', or if the environment variable POSIXLY\_CORRECT is set, then option processing stops as soon as a non-option argument is encountered.

**Exception**

1. exception getopt.GetoptError

This exception is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes msg and opt give the error message and related option; if there is no specific option to which the exception relates, opt is an empty string.

**Note:***exception getopt.error is alias for*[*GetoptError*](https://docs.python.org/3/library/getopt.html#getopt.GetoptError)*; for backward compatibility.*

In a script, typical usage is something like this:

import getopt, sys

def main():

try:

opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])

except getopt.GetoptError as err:

# print help information and exit:

print(err) # will print something like "option -a not recognized"

usage()

sys.exit(2)

output = None

verbose = False

for o, a in opts:

if o == "-v":

verbose = True

elif o in ("-h", "--help"):

usage()

sys.exit()

elif o in ("-o", "--output"):

output = a

else:

assert False, "unhandled option"

# ...

if \_\_name\_\_ == "\_\_main\_\_":

main()

An equivalent command line interface could be produced with less code and more informative help and error messages by using the argparse module:

**Example:**

import argparse

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument('-o', '--output')

parser.add\_argument('-v', dest='verbose', action='store\_true')

args = parser.parse\_args()

# ... do something with args.output ...

# ... do something with args.verbose ..

## Working with Iterators, Generators, Closure, Decorators

### Iterators

An iterator is an object representing a stream of data. This object returns the data one element at a time. A Python iterator must support a method called \_\_next\_\_() that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, \_\_next\_\_() must raise the StopIteration exception. Iterators don’t have to be finite, though; it’s perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in iter() function takes an arbitrary object and tries to return an iterator that will return the object’s contents or elements, raising TypeError if the object doesn’t support iteration. Several of Python’s built-in data types support iteration, the most common being lists and dictionaries. An object is called iterable if you can get an iterator for it.

Example:

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

>>> it = iter(L)

>>> it

<...iterator object at ...>

>>> it.\_\_next\_\_() # same as next(it)

1

>>> next(it)

2

>>> next(it)

3

>>> next(it)

Traceback (most recent call last):

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

StopIteration

>>>

Python expects iterable objects in several different contexts, the most important being the for statement. In the statement for X in Y, Y must be an iterator or some object for which iter() can create an iterator. These two statements are equivalent:

* for i in iter(obj):

print(i)

* for i in obj:

print(i)

Iterators can be materialized as lists or tuples by using the list() or tuple() constructor functions:

**Example:**

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

>>> iterator = iter(L)

>>> t = tuple(iterator)

>>> t

(1, 2, 3)

Sequence unpacking also supports iterators. If you know an iterator will return N elements, you can unpack them into an N-tuple.

**Example:**

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

>>> iterator = iter(L)

>>> a,b,c = iterator

>>> a,b,c

(1, 2, 3)

Built-in functions such as max() and min() can take a single iterator argument and will return the largest or smallest element.

The "in" and "not in" operators also support iterators: X in iterator is true if X is found in the stream returned by the iterator. You’ll run into obvious problems if the iterator is infinite; max(), min() will never return, and if the element X never appears in the stream, the "in" and "not in" operators won’t return either.

**Note:***You can only go forward in an iterator; there’s no way to get the previous element, reset the iterator, or make a copy of it. Iterator objects can optionally provide these additional capabilities, but the iterator protocol only specifies the \_\_next\_\_() method. Functions may therefore consume all of the iterator’s output, and if you need to do something different with the same stream, you’ll have to create a new iterator.*

Data Types that Support Iterators

We’ve already seen how lists and tuples support iterators. In fact, any Python sequence type, such as strings, will automatically support creation of an iterator.

Calling iter() on a dictionary returns an iterator that will loop over the dictionary’s keys.

**Example:**

>>>

>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,

... 'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}

>>> for key in m:

... print(key, m[key])

Mar 3

Feb 2

Aug 8

Sep 9

Apr 4

Jun 6

Jul 7

Jan 1

May 5

Nov 11

Dec 12

Oct 10

**Note:***The order is essentially random, because it’s based on the hash ordering of the objects in the dictionary.*

Applying iter() to a dictionary always loops over the keys, but dictionaries have methods that return other iterators. If you want to iterate over values or key/value pairs, you can explicitly call the values() or items() methods to get an appropriate iterator.

The dict() constructor can accept an iterator that returns a finite stream of (key, value) tuples:

**Example:**

>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]

>>> dict(iter(L))

{'Italy': 'Rome', 'US': 'Washington DC', 'France': 'Paris'}

Files also support iteration by calling the readline() method until there are no more lines in the file. This means you can read each line of a file like this:

**Example:**

for line in file:

# do something for each line

...

Sets can take their contents from an iterable and let you iterate over the set’s elements.

Example:

S = {2, 3, 5, 7, 11, 13}

for i in S:

print(i)

### Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the yield statement whenever they want to return data. Each time next() is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed).

**Example:**

def reverse(data):

for index in range(len(data)-1, -1, -1):

yield data[index]

>>>

>>> for char in reverse('golf'):

... print(char)

...

f

l

o

g

Anything that can be done with generators can also be done with class-based iterators. What makes generators so compact is that the \_\_iter\_\_() and \_\_next\_\_() methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much clearer than an approach using instance variables like self.index and self.data.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise StopIteration. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

#### Generator Expressions

Some simple generators can be coded as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Example:

>>> sum(i\*i for i in range(10)) # sum of squares

285

>>> xvec = [10, 20, 30]

>>> yvec = [7, 5, 3]

>>> sum(x\*y for x,y in zip(xvec, yvec)) # dot product

260

>>> from math import pi, sin

>>> sine\_table = {x: sin(x\*pi/180) for x in range(0, 91)}

>>> unique\_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'

>>> list(data[i] for i in range(len(data)-1, -1, -1))

['f', 'l', 'o', 'g']

### Closure

#### Nonlocal Variable in a Nested Function

Before seeing what a closure is, we have to first understand what are nested functions and non-local variables.

A function which is defined inside another function is known as nested function. Nested functions are able to access variables of the enclosing scope.

In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

Following is an example of a nested function accessing a non-local variable.

**Example:**

defprint\_msg(msg):

# This is the outer enclosing function

defprinter():

# This is the nested function

print(msg)

printer()

# We execute the function

# Output: Hello

print\_msg("Hello")

We can see that the nested function printer() was able to access the non-local variable msg of the enclosing function.

#### Defining a Closure Function

In the example above, what would happen if the last line of the function print\_msg()returned the printer() function instead of calling it? This means the function was defined as follows.

defprint\_msg(msg):

# This is the outer enclosing function

defprinter():

# This is the nested function

print(msg)

returnprinter# this got changed

# Now let's try calling this function.

# Output: Hello

another=print\_msg("Hello")

another()

That's unusual.

Theprint\_msg()function was called with the string"Hello"and the returned function was bound to the nameanother. On callinganother(), the message was still remembered although we had already finished executing theprint\_msg()function.

This technique by which some data ("Hello") gets attached to the code is called**closure in Python**.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

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

>>>del print\_msg

>>> another()

Hello

>>> print\_msg("Hello")

Traceback(most recent call last):

...

NameError: name 'print\_msg'isnotdefined

#### When do we have a Closure?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.

The criteria that must be met to create closure in Python are summarized in the following points.

* We must have a nested function (function inside a function).
* The nested function must refer to a value defined in the enclosing function.
* The enclosing function must return the nested function.

#### When to use Closures?

Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions. But when the number of attributes and methods get larger, better implement a class.

Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.

defmake\_multiplier\_of(n):

defmultiplier(x):

returnx\*n

returnmultiplier

# Multiplier of 3

times3=make\_multiplier\_of(3)

# Multiplier of 5

times5=make\_multiplier\_of(5)

# Output: 27

print(times3(9))

# Output: 15

print(times5(3))

# Output: 30

print(times5(times3(2)))

Decorators in Pythonmake an extensive use of closures as well.

On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out.

All function objects have a\_\_closure\_\_attribute that returns a tuple of cell objects if it is a closure function. Referring to the example above, we knowtimes3andtimes5are closure functions.

>>> make\_multiplier\_of.\_\_closure\_\_

>>> times3.\_\_closure\_\_

(<cell at 0x0000000002D155B8:intobject at 0x000000001E39B6E0>,)

The cell object has the attribute cell\_contents which stores the closed value.

>>> times3.\_\_closure\_\_[0].cell\_contents

3

>>> times5.\_\_closure\_\_[0].cell\_contents

5

### Decorators

Python has an interesting feature called decorators to add functionality to an existing code.

This is also called metaprogramming as a part of the program tries to modify another part of the program at compile time.

#### Prerequisites for Learning Decorators

In order to understand about decorators, we must first know a few basic things in Python.

We must be comfortable with the fact that, everything in Python (Yes! Even classes), are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object.

**Example:**

deffirst(msg):

print(msg)

first("Hello")

second=first

second("Hello")

When you run the code, both functions first and second gives same output. Here, the names first and second refer to the same function object.

Now things start getting weirder.Functions can be passed as arguments to another function.If you have used functions likemap, filter, and reducein Python, then you already know about this.

Such function that take other functions as arguments are also called **higher order functions**.

**Example:**

definc(x):

returnx+1

defdec(x):

returnx-1

defoperate(func, x):

result=func(x)

returnresult

We invoke the function as follows.

>>> operate(inc,3)

4

>>> operate(dec,3)

2

Furthermore, a function can return another function.

defis\_called():

defis\_returned():

print("Hello")

returnis\_returned

new=is\_called()

#Outputs "Hello"

new()

Here, is\_returned()is a nested function which is defined and returned, each time we call is\_called().

Finally, we must also know about closures in Python.

#### Getting back to Decorators

Functions and methods are called **callable** as they can be called.

In fact, any object which implements the special method \_\_call\_\_()is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it.

**Example:**

defmake\_pretty(func):

definner():

print("I got decorated")

func()

returninner

defordinary():

print("I am ordinary")

When you run the following codes in shell,

>>> ordinary()

I am ordinary

>>># let's decorate this ordinary function

>>> pretty = make\_pretty(ordinary)

>>> pretty()

I got decorated

I am ordinary

In the example shown above,make\_pretty()is a decorator. In the assignment step.

pretty = make\_pretty(ordinary)

The function ordinary()got decorated and the returned function was given the name pretty.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

ordinary = make\_pretty(ordinary).

This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the @symbol along with the name of the decorator function and place it above the definition of the function to be decorated.

**Example:**

@make\_pretty

def ordinary():

print("I am ordinary")

is equivalent to

def ordinary():

print("I am ordinary")

ordinary = make\_pretty(ordinary)

This is just a syntactic sugar to implement decorators.

#### Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like below?

def divide(a, b):

return a/b

This function has two parameters, a and b. We know, it will give error if we pass in bas 0.

>>> divide(2,5)

0.4

>>> divide(2,0)

Traceback (most recent call last):

...

ZeroDivisionError: division by zero

Now let's make a decorator to check for this case that will cause the error.

defsmart\_divide(func):

definner(a,b):

print("I am going to divide",a,"and",b)

ifb==0:

print("Whoops! cannot divide")

return

returnfunc(a,b)

returninner

@smart\_divide

defdivide(a,b):

returna/b

This new implementation will return Noneif the error condition arises.

>>> divide(2,5)

I am going to divide 2and5

0.4

>>> divide(2,0)

I am going to divide 2and0

Whoops! cannot divide

In this manner we can decorate functions that take parameters.

A keen observer will notice that parameters of the nested inner()function inside the decorator is same as the parameters of functions it decorates. Taking this into account, now we can make general decorators that work with any number of parameter.

In Python, this magic is done as function(\*args, \*\*kwargs). In this way, argswill be the[tuple](https://www.programiz.com/python-programming/tuple) of positional arguments and kwargswill be the[dictionary](https://www.programiz.com/python-programming/dictionary)of keyword arguments.

**Example:**

def works\_for\_all(func):

def inner(\*args,\*\*kwargs):

print("I can decorate any function")

return func(\*args,\*\*kwargs)

return inner

#### Chaining Decorators in Python

Multiple decorators can be chained in Python.

This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

defstar(func):

definner(\*args, \*\*kwargs):

print("\*"\*30)

func(\*args, \*\*kwargs)

print("\*"\*30)

returninner

defpercent(func):

definner(\*args, \*\*kwargs):

print("%"\*30)

func(\*args, \*\*kwargs)

print("%"\*30)

returninner

@star

@percent

defprinter(msg):

print(msg)

printer("Hello")

This will give the output.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Hello

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The above syntax of,

@star

@percent

def printer(msg):

print(msg)

is equivalent to

def printer(msg):

print(msg)

printer = star(percent(printer))

The order in which we chain decorators matter. If we had reversed the order as,

@percent

@star

def printer(msg):

print(msg)

The execution would take place as,

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hello

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

# Chapter 12 – Introduction to Threads in Python

### Threading

Threading is the process to manage several threads of execution.Using threads allows a program to run multiple operations concurrently in the same process space.

### Thread Objects

The simplest way to use athreadis to instantiate it with a target function and call start() to let it begin working.

**Example:**

threading\_simple.py

importthreading

def worker():

"""thread worker function"""

print('Worker')

threads = []

for i in range(5):

t = threading.Thread(target=worker)

threads.append(t)

t.start()

The output is five lines with "Worker"on each.

$ python3 threading\_simple.py

Worker

Worker

Worker

Worker

Worker

It is useful to be able to spawn a thread and pass it arguments to tell it what work to do. Any type of object can be passed as argument to the thread.

This example passes a number, which the thread then prints.

**Example:**

threading\_simpleargs.py

importthreading

def worker(num):

"""thread worker function"""

print('Worker: %s' % num)

threads = []

for i in range(5):

t = threading.Thread(target=worker, args=(i,))

threads.append(t)

t.start()

The integer argument is now included in the message printed by each thread.

$ python3 threading\_simpleargs.py

Worker: 0

Worker: 1

Worker: 2

Worker: 3

Worker: 4

### Determining the Current Thread

Using arguments to identify or name the thread is cumbersome and unnecessary. Each Threadinstance has a name with a default value that can be changed as the thread is created. Naming threads is useful in server processes with multiple service threads handling different operations.

**Example:**

threading\_names.py

importthreading

importtime

def worker():

print(threading.current\_thread().getName(), 'Starting')

time.sleep(0.2)

print(threading.current\_thread().getName(), 'Exiting')

def my\_service():

print(threading.current\_thread().getName(), 'Starting')

time.sleep(0.3)

print(threading.current\_thread().getName(), 'Exiting')

t = threading.Thread(name='my\_service', target=my\_service)

w = threading.Thread(name='worker', target=worker)

w2 = threading.Thread(target=worker) # use default name

w.start()

w2.start()

t.start()

The debug output includes the name of the current thread on each line. The lines with "Thread-1"in the thread name column correspond to the unnamed thread w2.

$ python3 threading\_names.py

worker Starting

Thread-1 Starting

my\_service Starting

worker Exiting

Thread-1 Exiting

my\_service Exiting

Most programs do not useprint() to debug. The loggingmodule supports embedding the thread name in every log message using the formatter code %(threadName)s. Including thread names in log messages makes it possible to trace those messages back to their source.

**Example:**

threading\_names\_log.py

importlogging

importthreading

importtime

def worker():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

def my\_service():

logging.debug('Starting')

time.sleep(0.3)

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='[%(levelname)s] (%(threadName)-10s) %(message)s',

)

t = threading.Thread(name='my\_service', target=my\_service)

w = threading.Thread(name='worker', target=worker)

w2 = threading.Thread(target=worker) # use default name

w.start()

w2.start()

t.start()

loggingis also thread-safe, so messages from different threads are kept distinct in the output.

**Example:**

$ python3 threading\_names\_log.py

[DEBUG] (worker ) Starting

[DEBUG] (Thread-1 ) Starting

[DEBUG] (my\_service) Starting

[DEBUG] (worker ) Exiting

[DEBUG] (Thread-1 ) Exiting

[DEBUG] (my\_service) Exiting

### Daemon vs. Non-Daemon Threads

Up to this point, the example programs have implicitly waited to exit until all threads have completed their work. Sometimes programs spawn a thread as a daemonthat runs without blocking the main program from exiting. Using daemon threads is useful for services where there may not be an easy way to interrupt the thread, or where letting the thread die in the middle of its work does not lose or corrupt data (for example, a thread that generates “heart beats” for a service monitoring tool). To mark a thread as a daemon, pass daemon=Truewhen constructing it or call its set\_daemon() method with True. The default is for threads to not be daemons.

**Example:**

threading\_daemon.py

importthreading

importtime

importlogging

def daemon():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

def non\_daemon():

logging.debug('Starting')

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

d = threading.Thread(name='daemon', target=daemon, daemon=True)

t = threading.Thread(name='non-daemon', target=non\_daemon)

d.start()

t.start()

The output does not include the "Exiting"message from the daemon thread, since all of the non-daemon threads (including the main thread) exit before the daemon thread wakes up from the sleep()call.

**Example:**

$ python3 threading\_daemon.py

(daemon ) Starting

(non-daemon) Starting

(non-daemon) Exiting

To wait until a daemon thread has completed its work, use the join()method.

**Example:**

threading\_daemon\_join.py

importthreading

importtime

importlogging

def daemon():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

def non\_daemon():

logging.debug('Starting')

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

d = threading.Thread(name='daemon', target=daemon, daemon=True)

t = threading.Thread(name='non-daemon', target=non\_daemon)

d.start()

t.start()

d.join()

t.join()

Waiting for the daemon thread to exit using join()means it has a chance to produce its "Exiting"message.

**Example:**

$ python3 threading\_daemon\_join.py

(daemon ) Starting

(non-daemon) Starting

(non-daemon) Exiting

(daemon ) Exiting

By default, join()blocks indefinitely. It is also possible to pass a float value representing the number of seconds to wait for the thread to become inactive. If the thread does not complete within the timeout period, join()returns anyway.

**Example:**

threading\_daemon\_join\_timeout.py

importthreading

importtime

importlogging

def daemon():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

def non\_daemon():

logging.debug('Starting')

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

d = threading.Thread(name='daemon', target=daemon, daemon=True)

t = threading.Thread(name='non-daemon', target=non\_daemon)

d.start()

t.start()

d.join(0.1)

print('d.isAlive()', d.isAlive())

t.join()

Since the timeout passed is less than the amount of time the daemon thread sleeps, the thread is still “alive” after join()returns.

**Example:**

$ python3 threading\_daemon\_join\_timeout.py

(daemon ) Starting

(non-daemon) Starting

(non-daemon) Exiting

d.isAlive() True

### Enumerating all Threads

It is not necessary to retain an explicit handle to all of the daemon threads in order to ensure they have completed before exiting the main process. enumerate() returns a list of active Threadinstances. The list includes the current thread, and since joining the current thread introduces a deadlock situation, it must be skipped.

**Example:**

threading\_enumerate.py

importrandom

importthreading

importtime

importlogging

def worker():

"""thread worker function"""

pause = random.randint(1, 5) / 10

logging.debug('sleeping %0.2f', pause)

time.sleep(pause)

logging.debug('ending')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

for i in range(3):

t = threading.Thread(target=worker, daemon=True)

t.start()

main\_thread = threading.main\_thread()

for t in threading.enumerate():

if t is main\_thread:

continue

logging.debug('joining %s', t.getName())

t.join()

Because the worker is sleeping for a random amount of time, the output from this program may vary.

$ python3 threading\_enumerate.py

(Thread-1 ) sleeping 0.20

(Thread-2 ) sleeping 0.30

(Thread-3 ) sleeping 0.40

(MainThread) joining Thread-1

(Thread-1 ) ending

(MainThread) joining Thread-3

(Thread-2 ) ending

(Thread-3 ) ending

(MainThread) joining Thread-2

### 

### Subclassing Thread

At start-up, a Thread does some basic initialization and then calls its run()method, which calls the target function passed to the constructor. To create a subclass of Thread, override run()to do whatever is necessary.

**Example:**

threading\_subclass.py

importthreading

importlogging

classMyThread(threading.Thread):

def run(self):

logging.debug('running')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

for i in range(5):

t = MyThread()

t.start()

The return value of run()is ignored.

$ python3 threading\_subclass.py

(Thread-1 ) running

(Thread-2 ) running

(Thread-3 ) running

(Thread-4 ) running

(Thread-5 ) running

Because the args and kwargs values passed to the Threadconstructor are saved in private variables using names prefixed with '\_\_', they are not easily accessed from a subclass. To pass arguments to a custom thread type, redefine the constructor to save the values in an instance attribute that can be seen in the subclass.

**Example:**

threading\_subclass\_args.py

importthreading

importlogging

classMyThreadWithArgs(threading.Thread):

def \_\_init\_\_(self, group=None, target=None, name=None,

args=(), kwargs=None, \*, daemon=None):

super().\_\_init\_\_(group=group, target=target, name=name,

daemon=daemon)

self.args = args

self.kwargs = kwargs

def run(self):

logging.debug('running with %s and %s',

self.args, self.kwargs)

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

for i in range(5):

t = MyThreadWithArgs(args=(i,), kwargs={'a': 'A', 'b': 'B'})

t.start()

MyThreadWithArgs uses the same API as Thread, but another class could easily change the constructor method to take more or different arguments more directly related to the purpose of the thread, as with any other class.

**Example:**

$ python3 threading\_subclass\_args.py

(Thread-1 ) running with (0,) and {'b': 'B', 'a': 'A'}

(Thread-2 ) running with (1,) and {'b': 'B', 'a': 'A'}

(Thread-3 ) running with (2,) and {'b': 'B', 'a': 'A'}

(Thread-4 ) running with (3,) and {'b': 'B', 'a': 'A'}

(Thread-5 ) running with (4,) and {'b': 'B', 'a': 'A'}

### Timer Threads

One example of a reason to subclass Thread is provided by Timer, also included in threading. A Timerstarts its work after a delay, and can be canceled at any point within that delay time period.

**Example:**

threading\_timer.py

importthreading

importtime

importlogging

def delayed():

logging.debug('worker running')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

t1 = threading.Timer(0.3, delayed)

t1.setName('t1')

t2 = threading.Timer(0.3, delayed)

t2.setName('t2')

logging.debug('starting timers')

t1.start()

t2.start()

logging.debug('waiting before canceling %s', t2.getName())

time.sleep(0.2)

logging.debug('canceling %s', t2.getName())

t2.cancel()

logging.debug('done')

The second timer in this example is never run, and the first timer appears to run after the rest of the main program is done. Since it is not a daemon thread, it is joined implicitly when the main thread is done.

**Example:**

$ python3 threading\_timer.py

(MainThread) starting timers

(MainThread) waiting before canceling t2

(MainThread) canceling t2

(MainThread) done

(t1 ) worker running

### Signaling Between Threads

Although the point of using multiple threads is to run separate operations concurrently, there are times when it is important to be able to synchronize the operations in two or more threads. Event objects are a simple way to communicate between threads safely. An Eventmanages an internal flag that callers can control with the set() and clear() methods. Other threads can use wait()to pause until the flag is set, effectively blocking progress until allowed to continue.

**Example:**

threading\_event.py

importlogging

importthreading

importtime

def wait\_for\_event(e):

"""Wait for the event to be set before doing anything"""

logging.debug('wait\_for\_event starting')

event\_is\_set = e.wait()

logging.debug('event set: %s', event\_is\_set)

def wait\_for\_event\_timeout(e, t):

"""Wait t seconds and then timeout"""

whilenot e.is\_set():

logging.debug('wait\_for\_event\_timeout starting')

event\_is\_set = e.wait(t)

logging.debug('event set: %s', event\_is\_set)

if event\_is\_set:

logging.debug('processing event')

else:

logging.debug('doing other work')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

e = threading.Event()

t1 = threading.Thread(

name='block',

target=wait\_for\_event,

args=(e,),

)

t1.start()

t2 = threading.Thread(

name='nonblock',

target=wait\_for\_event\_timeout,

args=(e, 2),

)

t2.start()

logging.debug('Waiting before calling Event.set()')

time.sleep(0.3)

e.set()

logging.debug('Event is set')

The wait()method takes an argument representing the number of seconds to wait for the event before timing out. It returns a Boolean indicating whether or not the event is set, so the caller knows why wait() returned. The is\_set()method can be used separately on the event without fear of blocking.

In this example, wait\_for\_event\_timeout()checks the event status without blocking indefinitely. The wait\_for\_event() blocks on the call to wait(), which does not return until the event status changes.

**Example:**

$ python3 threading\_event.py

(block ) wait\_for\_event starting

(nonblock ) wait\_for\_event\_timeout starting

(MainThread) Waiting before calling Event.set()

(MainThread) Event is set

(nonblock ) event set: True

(nonblock ) processing event

(block ) event set: True

### Controlling Access to Resources

In addition to synchronizing the operations of threads, it is also important to be able to control access to shared resources to prevent corruption or missed data. Python’s built-in data structures (lists, dictionaries, etc.) are thread-safe as a side-effect of having atomic byte-codes for manipulating them (the global interpreter lock used to protect Python’s internal data structures is not released in the middle of an update). Other data structures implemented in Python, or simpler types like integers and floats, do not have that protection. To guard against simultaneous access to an object, use a Lockobject.

**Example:**

threading\_lock.py

importlogging

importrandom

importthreading

importtime

classCounter:

def \_\_init\_\_(self, start=0):

self.lock = threading.Lock()

self.value = start

def increment(self):

logging.debug('Waiting for lock')

self.lock.acquire()

try:

logging.debug('Acquired lock')

self.value = self.value + 1

finally:

self.lock.release()

def worker(c):

for i in range(2):

pause = random.random()

logging.debug('Sleeping %0.02f', pause)

time.sleep(pause)

c.increment()

logging.debug('Done')

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

counter = Counter()

for i in range(2):

t = threading.Thread(target=worker, args=(counter,))

t.start()

logging.debug('Waiting for worker threads')

main\_thread = threading.main\_thread()

for t in threading.enumerate():

if t isnot main\_thread:

t.join()

logging.debug('Counter: %d', counter.value)

In this example, the worker() function increments a Counter instance, which manages a Lockto prevent two threads from changing its internal state at the same time. If the Lockwas not used, there is a possibility of missing a change to the value attribute.

**Example:**

$ python3 threading\_lock.py

(Thread-1 ) Sleeping 0.18

(Thread-2 ) Sleeping 0.93

(MainThread) Waiting for worker threads

(Thread-1 ) Waiting for lock

(Thread-1 ) Acquired lock

(Thread-1 ) Sleeping 0.11

(Thread-1 ) Waiting for lock

(Thread-1 ) Acquired lock

(Thread-1 ) Done

(Thread-2 ) Waiting for lock

(Thread-2 ) Acquired lock

(Thread-2 ) Sleeping 0.81

(Thread-2 ) Waiting for lock

(Thread-2 ) Acquired lock

(Thread-2 ) Done

(MainThread) Counter: 4

To find out whether another thread has acquired the lock without holding up the current thread, pass False for the blocking argument to acquire(). In the next example, worker()tries to acquire the lock three separate times and counts how many attempts it has to make to do so. In the mean time, lock\_holder()cycles between holding and releasing the lock, with short pauses in each state used to simulate load.

**Example:**

threading\_lock\_noblock.py

importlogging

importthreading

importtime

def lock\_holder(lock):

logging.debug('Starting')

whileTrue:

lock.acquire()

try:

logging.debug('Holding')

time.sleep(0.5)

finally:

logging.debug('Not holding')

lock.release()

time.sleep(0.5)

def worker(lock):

logging.debug('Starting')

num\_tries = 0

num\_acquires = 0

while num\_acquires < 3:

time.sleep(0.5)

logging.debug('Trying to acquire')

have\_it = lock.acquire(0)

try:

num\_tries += 1

if have\_it:

logging.debug('Iteration %d: Acquired',

num\_tries)

num\_acquires += 1

else:

logging.debug('Iteration %d: Not acquired',

num\_tries)

finally:

if have\_it:

lock.release()

logging.debug('Done after %d iterations', num\_tries)

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

lock = threading.Lock()

holder = threading.Thread(

target=lock\_holder,

args=(lock,),

name='LockHolder',

daemon=True,

)

holder.start()

worker = threading.Thread(

target=worker,

args=(lock,),

name='Worker',

)

worker.start()

It takes worker()more than three iterations to acquire the lock three separate times.

**Example:**

$ python3 threading\_lock\_noblock.py

(LockHolder) Starting

(LockHolder) Holding

(Worker ) Starting

(LockHolder) Not holding

(Worker ) Trying to acquire

(Worker ) Iteration 1: Acquired

(LockHolder) Holding

(Worker ) Trying to acquire

(Worker ) Iteration 2: Not acquired

(LockHolder) Not holding

(Worker ) Trying to acquire

(Worker ) Iteration 3: Acquired

(LockHolder) Holding

(Worker ) Trying to acquire

(Worker ) Iteration 4: Not acquired

(LockHolder) Not holding

(Worker ) Trying to acquire

(Worker ) Iteration 5: Acquired

(Worker ) Done after 5 iterations

### Re-entrant Locks

Normal Lockobjects cannot be acquired more than once, even by the same thread. This can introduce undesirable side-effects if a lock is accessed by more than one function in the same call chain.

**Example:**

threading\_lock\_reacquire.py

importthreading

lock = threading.Lock()

print('First try :', lock.acquire())

print('Second try:', lock.acquire(0))

In this case, the second call to acquire()is given a zero timeout to prevent it from blocking because the lock has been obtained by the first call.

**Example:**

$ python3 threading\_lock\_reacquire.py

First try : True

Second try: False

In a situation where separate code from the same thread needs to “re-acquire” the lock, use an RLockinstead.

**Example:**

threading\_rlock.py

importthreading

lock = threading.RLock()

print('First try :', lock.acquire())

print('Second try:', lock.acquire(0))

The only change to the code from the previous example was substituting RLock for Lock.

**Example:**

$ python3 threading\_rlock.py

First try : True

Second try: True

### Lock as Context Managers

Locks implement the context manager API and are compatible with the with statement. Using withremoves the need to explicitly acquire and release the lock.

**Example:**

threading\_lock\_with.py

importthreading

importlogging

def worker\_with(lock):

with lock:

logging.debug('Lock acquired via with')

def worker\_no\_with(lock):

lock.acquire()

try:

logging.debug('Lock acquired directly')

finally:

lock.release()

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

lock = threading.Lock()

w = threading.Thread(target=worker\_with, args=(lock,))

nw = threading.Thread(target=worker\_no\_with, args=(lock,))

w.start()

nw.start()

The two functions worker\_with() and worker\_no\_with()manage the lock in equivalent ways.

**Example:**

$ python3 threading\_lock\_with.py

(Thread-1 ) Lock acquired via with

(Thread-2 ) Lock acquired directly

### Synchronizing Threads

In addition to using Events, another way of synchronizing threads is through using a Condition object. Because the Condition uses a Lock, it can be tied to a shared resource, allowing multiple threads to wait for the resource to be updated. In this example, the consumer()threads wait for the Conditionto be set before continuing. The producer()thread is responsible for setting the condition and notifying the other threads that they can continue.

**Example:**

threading\_condition.py

importlogging

importthreading

importtime

def consumer(cond):

"""wait for the condition and use the resource"""

logging.debug('Starting consumer thread')

with cond:

cond.wait()

logging.debug('Resource is available to consumer')

def producer(cond):

"""set up the resource to be used by the consumer"""

logging.debug('Starting producer thread')

with cond:

logging.debug('Making resource available')

cond.notifyAll()

logging.basicConfig(

level=logging.DEBUG,

format='%(asctime)s (%(threadName)-2s) %(message)s',

)

condition = threading.Condition()

c1 = threading.Thread(name='c1', target=consumer,

args=(condition,))

c2 = threading.Thread(name='c2', target=consumer,

args=(condition,))

p = threading.Thread(name='p', target=producer,

args=(condition,))

c1.start()

time.sleep(0.2)

c2.start()

time.sleep(0.2)

p.start()

The threads use withto acquire the lock associated with the Condition. Using the acquire() and release()methods explicitly also works.

**Example:**

$ python3 threading\_condition.py

2016-07-10 10:45:28,170 (c1) Starting consumer thread

2016-07-10 10:45:28,376 (c2) Starting consumer thread

2016-07-10 10:45:28,581 (p ) Starting producer thread

2016-07-10 10:45:28,581 (p ) Making resource available

2016-07-10 10:45:28,582 (c1) Resource is available to consumer

2016-07-10 10:45:28,582 (c2) Resource is available to consumer

Barriers are another thread synchronization mechanism. A Barrierestablishes a control point and all participating threads block until all of the participating “parties” have reached that point. It lets threads start up separately and then pause until they are all ready to proceed.

**Example:**

threading\_barrier.py

importthreading

importtime

def worker(barrier):

print(threading.current\_thread().name,

'waiting for barrier with {} others'.format(

barrier.n\_waiting))

worker\_id = barrier.wait()

print(threading.current\_thread().name, 'after barrier',

worker\_id)

NUM\_THREADS = 3

barrier = threading.Barrier(NUM\_THREADS)

threads = [

threading.Thread(

name='worker-%s' % i,

target=worker,

args=(barrier,),

)

for i in range(NUM\_THREADS)

]

for t in threads:

print(t.name, 'starting')

t.start()

time.sleep(0.1)

for t in threads:

t.join()

In this example, the Barrieris configured to block until three threads are waiting. When the condition is met, all of the threads are released past the control point at the same time. The return value from wait()indicates the number of the party being released, and can be used to limit some threads from taking an action like cleaning up a shared resource.

**Example:**

$ python3 threading\_barrier.py

worker-0 starting

worker-0 waiting for barrier with 0 others

worker-1 starting

worker-1 waiting for barrier with 1 others

worker-2 starting

worker-2 waiting for barrier with 2 others

worker-2 after barrier 2

worker-0 after barrier 0

worker-1 after barrier 1

The abort()method of Barriercauses all of the waiting threads to receive a BrokenBarrierError. This allows threads to clean up if processing is stopped while they are blocked on wait().

**Example:**

threading\_barrier\_abort.py

importthreading

importtime

def worker(barrier):

print(threading.current\_thread().name,

'waiting for barrier with {} others'.format(

barrier.n\_waiting))

try:

worker\_id = barrier.wait()

except threading.BrokenBarrierError:

print(threading.current\_thread().name, 'aborting')

else:

print(threading.current\_thread().name, 'after barrier',

worker\_id)

NUM\_THREADS = 3

barrier = threading.Barrier(NUM\_THREADS + 1)

threads = [

threading.Thread(

name='worker-%s' % i,

target=worker,

args=(barrier,),

)

for i in range(NUM\_THREADS)

]

for t in threads:

print(t.name, 'starting')

t.start()

time.sleep(0.1)

barrier.abort()

for t in threads:

t.join()

This example configures the Barrierto expect one more participating thread than is actually started so that processing in all of the threads is blocked. The abort()call raises an exception in each blocked thread.

**Example:**

$ python3 threading\_barrier\_abort.py

worker-0 starting

worker-0 waiting for barrier with 0 others

worker-1 starting

worker-1 waiting for barrier with 1 others

worker-2 starting

worker-2 waiting for barrier with 2 others

worker-0 aborting

worker-2 aborting

worker-1 aborting

### Limiting Concurrent Access to Resources

Sometimes it is useful to allow more than one worker access to a resource at a time, while still limiting the overall number. For example, a connection pool might support a fixed number of simultaneous connections, or a network application might support a fixed number of concurrent downloads. A Semaphoreis one way to manage those connections.

**Example:**

threading\_semaphore.py

importlogging

importrandom

importthreading

importtime

classActivePool:

def \_\_init\_\_(self):

super(ActivePool, self).\_\_init\_\_()

self.active = []

self.lock = threading.Lock()

def makeActive(self, name):

with self.lock:

self.active.append(name)

logging.debug('Running: %s', self.active)

def makeInactive(self, name):

with self.lock:

self.active.remove(name)

logging.debug('Running: %s', self.active)

def worker(s, pool):

logging.debug('Waiting to join the pool')

with s:

name = threading.current\_thread().getName()

pool.makeActive(name)

time.sleep(0.1)

pool.makeInactive(name)

logging.basicConfig(

level=logging.DEBUG,

format='%(asctime)s (%(threadName)-2s) %(message)s',

)

pool = ActivePool()

s = threading.Semaphore(2)

for i in range(4):

t = threading.Thread(

target=worker,

name=str(i),

args=(s, pool),

)

t.start()

In this example, the ActivePoolclass simply serves as a convenient way to track which threads are able to run at a given moment. A real resource pool would allocate a connection or some other value to the newly active thread, and reclaim the value when the thread is done. Here, it is just used to hold the names of the active threads to show that at most two are running concurrently.

**Example:**

$ python3 threading\_semaphore.py

2016-07-10 10:45:29,398 (0 ) Waiting to join the pool

2016-07-10 10:45:29,398 (0 ) Running: ['0']

2016-07-10 10:45:29,399 (1 ) Waiting to join the pool

2016-07-10 10:45:29,399 (1 ) Running: ['0', '1']

2016-07-10 10:45:29,399 (2 ) Waiting to join the pool

2016-07-10 10:45:29,399 (3 ) Waiting to join the pool

2016-07-10 10:45:29,501 (1 ) Running: ['0']

2016-07-10 10:45:29,501 (0 ) Running: []

2016-07-10 10:45:29,502 (3 ) Running: ['3']

2016-07-10 10:45:29,502 (2 ) Running: ['3', '2']

2016-07-10 10:45:29,607 (3 ) Running: ['2']

2016-07-10 10:45:29,608 (2 ) Running: []

### Thread-specific Data

While some resources need to be locked so multiple threads can use them, others need to be protected so that they are hidden from threads that do not own them. The local()class creates an object capable of hiding values from view in separate threads.

**Example:**

threading\_local.py

importrandom

importthreading

importlogging

def show\_value(data):

try:

val = data.value

except AttributeError:

logging.debug('No value yet')

else:

logging.debug('value=%s', val)

def worker(data):

show\_value(data)

data.value = random.randint(1, 100)

show\_value(data)

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

local\_data = threading.local()

show\_value(local\_data)

local\_data.value = 1000

show\_value(local\_data)

for i in range(2):

t = threading.Thread(target=worker, args=(local\_data,))

t.start()

The attribute local\_data.valueis not present for any thread until it is set in that thread.

**Example:**

$ python3 threading\_local.py

(MainThread) No value yet

(MainThread) value=1000

(Thread-1 ) No value yet

(Thread-1 ) value=33

(Thread-2 ) No value yet

(Thread-2 ) value=74

To initialize the settings so all threads start with the same value, use a subclass and set the attributes in \_\_init\_\_().

**Example:**

threading\_local\_defaults.py

importrandom

importthreading

importlogging

def show\_value(data):

try:

val = data.value

except AttributeError:

logging.debug('No value yet')

else:

logging.debug('value=%s', val)

def worker(data):

show\_value(data)

data.value = random.randint(1, 100)

show\_value(data)

classMyLocal(threading.local):

def \_\_init\_\_(self, value):

super().\_\_init\_\_()

logging.debug('Initializing %r', self)

self.value = value

logging.basicConfig(

level=logging.DEBUG,

format='(%(threadName)-10s) %(message)s',

)

local\_data = MyLocal(1000)

show\_value(local\_data)

for i in range(2):

t = threading.Thread(target=worker, args=(local\_data,))

t.start()

\_\_init\_\_()is invoked on the same object (note the id()value), once in each thread to set the default values.

**Example:**

$ python3 threading\_local\_defaults.py

(MainThread) Initializing <\_\_main\_\_.MyLocal object at

0x101c6c288>

(MainThread) value=1000

(Thread-1 ) Initializing <\_\_main\_\_.MyLocal object at

0x101c6c288>

(Thread-1 ) value=1000

(Thread-1 ) value=18

(Thread-2 ) Initializing <\_\_main\_\_.MyLocal object at

0x101c6c288>

(Thread-2 ) value=1000

(Thread-2 ) value=77

# Chapter 13 – Internet Programming

### Overview of Python CGI Programming

### About Common Gateway Interface (CGI)

CGI stands for Common Gateway Interface. CGI is one of the essential parts of HTTP (Hyper-Text Transfer Protocol).

It is a set of standards that define a standard way of passing information or web-user request to an application program and to get data back to forward it to users. This is the exchange of information between web-server and a custom script. When the users requested the web-page, the server sends the requested web-page. The web server usually passes the information to all application programs that process data and sends back an acknowledged message; this technique of passing data back-and-forth between server and application is the Common Gateway Interface.

When a user clicks a hyperlink to browse a particular web-page or URL (Uniform Resource Locator), the following steps happen:

* Browser contacts the HTTP web server for demanding the URL
* Parsing the URL
* Look for the filename
* If it finds that file, a request is sent back
* Web browser takes a response from the web server
* As the server response, it either shows the received file or an error message

It may become possible to set-up an HTTP server because when a certain directory is requested that file is not sent back; instead it is executed as a program and that program's output is displayed back to your browser.

The cgi module is the support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

### Configuring Apache Web Server for CGI

In this section, you will install Apache 2, and ensure that it recognizes Python files as executables.

Install Apache using apt-get:

sudo apt-get install apache2

Like MySQL, the Apache server will start once the installation completes.

**Note**: After installation, several ports are open to the Internet. Make sure to see the conclusion of this tutorial for resources on security.

It’s important to place your website's root directory in a safe location. The server is by default at /var/www/html. To keep convention, we will create a new directory for testing purposes, called test, in the same location.

sudo mkdir /var/www/test

Finally, we must register Python with Apache.

To start, you need to disable multithreading processes.

sudo a2dismod mpm\_event

Then, you should give Apache explicit permission to run scripts by using the following command:

sudo a2enmod mpm\_prefork cgi

Next, we modify the actual Apache configuration, to explicitly declare Python files as runnable file and allow such executables.

Open the configuration file using nano or your favorite text editor.

sudo nano /etc/apache2/sites-enabled/000-default.conf

Add the following right after the first line, which reads

<VirtualHost \*:80\>.

<Directory /var/www/test>

Options +ExecCGI

DirectoryIndex index.py

</Directory>

AddHandler cgi-script .py

Make sure that your <Directory> block is nested inside the <VirtualHost>block, like so. Make sure to indent correctly with tabs, too.

/etc/apache2/sites-enabled/000-default.conf

<VirtualHost \*:80>

<Directory /var/www/test>

Options +ExecCGI

DirectoryIndex index.py

</Directory>

AddHandler cgi-script .py

...

This Directory block allows us to specify how Apache treats that directory. It tells Apache that the /var/www/testdirectory contains executables, considers index.pyto be the default file, then defines the executables.

We also want to allow executables in our website directory, so we need to change the path for DocumentRoot, too. Look for the line that reads DocumentRoot /var/www/html, a few lines below the long comment at the top of the file, and modify it to read /var/www/testinstead.

DocumentRoot /var/www/test

Your file should now resemble the following:

/etc/apache2/sites-enabled/000-default.conf

<VirtualHost \*:80>

<Directory /var/www/test>

Options +ExecCGI

DirectoryIndex index.py

</Directory>

AddHandler cgi-script .py

...

DocumentRoot /var/www/test

...

Save and exit the file. To put these changes into effect, restart Apache.

sudo service apache2 restart

**Note**: *Apache 2 may throw a warning which says about the server's fully qualified domain name; this can be ignored as the ServerName directive has little application as of this moment. They are ultimately used to determine subdomain hosting, after the*[*necessary records*](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-host-name-with-digitalocean)*are created.*

If the last line of the output reads [ OK ], Apache has restarted successfully.

### Writing the First CGI Program

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML <FORM> or <ISINDEX> element.

Most often, CGI scripts live in the server’s special cgi-bin directory. The HTTP server places all sorts of information about the request (such as the client’s hostname, the requested URL, the query string, and lots of other goodies) in the script’s shell environment, executes the script, and sends the script’s output back to the client.

The script’s input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the “query string” part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

print("Content-Type: text/html") # HTML is following

print() # blank line, end of headers

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here’s Python code that prints a simple piece of HTML:

print("<TITLE>CGI script output</TITLE>")

print("<H1>This is my first CGI script</H1>")

print("Hello, world!")

To create a CGI script in Python, begin your script by importing the cgi module. Use the following syntax for the same:

import cgitb

cgitb.enable()

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you do not want to show the details of your program to users of your script, you can have the reports saved to files instead, with code like this:

import cgitb

cgitb.enable(display=0, logdir="/path/to/logdir")

It’s very helpful to use this feature during script development. The reports produced by cgitb provide information that can save you a lot of time in tracking down bugs. You can always remove the cgitb line later when you have tested your script and are confident that it works correctly.

### CGI Environment Variables

|  |  |
| --- | --- |
| Environment Variable | Description |
| content\_type | Describes the data-type of the content |
| http\_cookie | Returns the visitor's cookie if one is set |
| content\_length | It is available for POST request to define the length of query information |
| http\_user\_agent | Defines the browser type of the visitor |
| path\_info | Defines the path of a CGI script |
| remote\_host | Defines the host-name of the visitor |
| remote\_addr | Defines the IP Address of the visitor |
| request\_method | Is used to make request & the most common methods are - GET and POST |

## Working with Databases

### pymysql

pymysqlpackage contains a pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb and work on CPython, PyPy and IronPython.

#### Prerequisites: Installing pip

Before installing pymysql, you need to install pip, the recommended package installer for Python.

You need to first, update the system's package index. This will ensure that old or outdated packages do not interfere with the installation.

sudo apt-get update

pip allows you to easily manage any Python 3 package you would like to have.

To install pip, simply run the following:

sudo apt-get install python3-pip

### Installing pymysql

In this section, we will install and configure MySQL.

To install SQL, write the following command:

sudo apt-get install mysql-server

Enter a strong password for the MySQL root user when prompted, and remember it, because you will need it later.

The MySQL server will start once installation completes. After installation, run:

mysql\_secure\_installation

This setup will take you through a series of self-explanatory steps. First, you'll need to enter the root password you picked a moment ago. The first question will ask if you want to change the root password, but because you just set it, enter **n**. For all other questions, press **ENTER**to accept the default response.

Python 3 requires a way to connect with MySQL. There are a number of options, like MySQLclient, but for the module's simplicity, we will usepymysql.

To install pymysql using pip, type:

sudo pip3 install pymysql

### Database Operations

You can perform following database operations within a Python program.

|  |  |
| --- | --- |
| Operation | Description |
| insert() | Creates a record into a table. |
| read() | Fetches useful information from the database. |
| update() | It is used update those available or already existing record(s). |
| delete() | It is used to delete records from the database. |
| commit() | Gives a green signal to the database to finalize the changes, and after this operation, no change can be reverted back. |
| rollback() | It works like "undo", which reverts all the changes that you have made. |

### Setting Database Connection and Creating Database Table

The following examples make use of a simple table:

**Example 1:**

CREATE TABLE `users` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`email` varchar(255) COLLATE utf8\_bin NOT NULL,

`password` varchar(255) COLLATE utf8\_bin NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8\_bin

AUTO\_INCREMENT=1 ;

The following example sets a connection with the database and makes use of a simple table.

**Example 2:**

import pymysql.cursors

# Connect to the database

connection = pymysql.connect(host='localhost',

user='user',

password='passwd',

db='db',

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor)

try:

with connection.cursor() as cursor:

# Create a new record

sql = "INSERT INTO `users` (`email`, `password`) VALUES (%s, %s)"

cursor.execute(sql, ('webmaster@python.org', 'very-secret'))

# connection is not autocommit by default. So you must commit to save

# your changes.

connection.commit()

with connection.cursor() as cursor:

# Read a single record

sql = "SELECT `id`, `password` FROM `users` WHERE `email`=%s"

cursor.execute(sql, ('webmaster@python.org',))

result = cursor.fetchone()

print(result)

finally:

connection.close()

### Disconnecting from Database

To disconnect the Database connection, use the close() method.

db.close()

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of the DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

## Testing the Final Product

In this section, we will confirm that individual components (Python, MySQL, and Apache) can interact with one another by creating an example webpage and database.

First, let's create a database. Log in to MySQL. You'll need to enter the MySQL root password you set earlier.

mysql -u root -p

Add an example database called example.

CREATE DATABASE example;

Switch to the new database.

USE example;

Add a table for some example data that we'll have the Python app add.

CREATE TABLE numbers (num INT, word VARCHAR(20));

Press **CTRL+D**to exit. Now, create a new file for our simple Python app.

sudo nano /var/www/test/index.py

Copy and paste the following code in. The in-line comments describe what each piece of the code does. Make sure to replace the passwdvalue with the root MySQL password you chose earlier.

#!/usr/bin/python

# Turn on debug mode.

import cgitb

cgitb.enable()

# Print necessary headers.

print("Content-Type: text/html")

print()

# Connect to the database.

import pymysql

conn = pymysql.connect(

db='example',

user='root',

passwd='your\_root\_mysql\_password',

host='localhost')

c = conn.cursor()

# Insert some example data.

c.execute("INSERT INTO numbers VALUES (1, 'One!')")

c.execute("INSERT INTO numbers VALUES (2, 'Two!')")

c.execute("INSERT INTO numbers VALUES (3, 'Three!')")

conn.commit()

# Print the contents of the database.

c.execute("SELECT \* FROM numbers")

print([(r[0], r[1]) for r in c.fetchall()])

Save and exit.

Next, fix permissions on the newly-created file. For more information on the three-digit permissions code, see the tutorial on[Linux permissions](https://www.digitalocean.com/community/tutorials/linux-permissions-basics-and-how-to-use-umask-on-a-vps).

sudo chmod 777 /var/www/test/index.py

Now, access your server's by going to <http://your_server_ip>using your favorite browser. You should see the following:

your\_server\_ip'>http://your\_server\_ip

[(1, 'One!'), (2, 'Two!'), (3, 'Three!')]

Congratulations! Your server is now online.

# Chapter 14 – Network and Graphical User Interface Programming

## Socket Programming

### Overview of Sockets

A socket is the end-point in a flow of communication between two programs or communication channels operating over a network. They are created using a set of programming request called socket API (Application Programming Interface). Python's socket library offers classes for handling common transports as a generic interface.

Sockets use protocols for determining the connection type for port-to-port communication between client and server machines. The protocols are used for:

* Domain Name Servers (DNS)
* IP addressing
* E-mail
* FTP (File Transfer Protocol), etc...

### The socket Module

The socket module provides access to the BSD (Berkeley Software Distribution) socket interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

**Note:***Some behavior may be platform dependent, since calls are made to the operating system Enable a server to accept connections. If backlog is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.*

The socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface, as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

### The socket Methods

#### Server Socket Methods

|  |  |
| --- | --- |
| Method | Description |
| socket.bind() | Binds the socket to address. The socket must not already be bound. |
| socket.listen() | Enables a server to accept connections. If backlog is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen. |
| socket.accept() | Accepts a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.  The newly created socket is non-inheritable. |

#### Client Socket Methods

|  |  |
| --- | --- |
| Method | Description |
| socket.connect() | Connects to a remote socket at address.  If the connection is interrupted by a signal, the method waits until the connection completes, or raise a socket.timeout on timeout, if the signal handler doesn’t raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an InterruptedError exception if the connection is interrupted by a signal (or the exception raised by the signal handler). |

#### General Socket Methods

|  |  |
| --- | --- |
| Method | Description |
| socket.recv() | Receives data from the socket. The return value is a bytes object representing the data received. |
| socket.send() | Sends data to the socket. The socket must be connected to a remote socket. |
| socket.recvfrom() | Receive data from the socket. The return value is a pair (bytes, address) where bytes is a bytes object representing the data received and address is the address of the socket sending the data. |
| socket.sendto() | Sends data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by the address parameter of the method. |
| socket.close() | Marks the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).  Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly, or to use a with statement around them. |
| socket.gethostnamee() | Return a string containing the hostname of the machine where the Python interpreter is currently executing.  **Note:***gethostname() doesn’t always return the fully qualified domain name; use getfqdn() for that.* |

### Creating Server Socket and Client Socket

#### Server Socket: Here is a simple program to open a web server at port 60.

**Example:**

import socket

T\_PORT = 60

TCP\_IP = '127.0.0.1'

BUF\_SIZE = 30

# create a socket object name 'k'

k = socket.socket (socket.AF\_INET, socket.SOCK\_STREAM)

k.bind((TCP\_IP, T\_PORT))

k.listen(1)

con, addr = k.accept()

print ('Connection Address is: ' , addr)

while True :

data = con.recv(BUF\_SIZE)

if not data:

break

print ("Received data", data)

con.send(data)

con.close()

#### Client Socket

The following example shows how sockets work.

**Example:**

import socket

T\_PORT = 5006

TCP\_IP = '127.0.0.1'

BUF\_SIZE = 1024

MSG = "Hello karl"

# create a socket object name 'k'

k = socket.socket (socket.AF\_INET, socket.SOCK\_STREAM)

k.connect((TCP\_IP, T\_PORT))

k.send(MSG)

data = k.recv(BUF\_SIZE)

k.close

## Working with XML Files

### About XML

XML stands for eXtensible Markup Language. It was designed to store and transport data. It was designed to be both human- and machine-readable. That’s why, the design goals of XML emphasize simplicity, generality, and usability across the Internet.

The XML file to be parsed in this tutorial is actually a RSS feed.

Python’s interfaces for processing XML are grouped in the xml package.

### Parsing XML with SAX APIs

The xml.sax package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

The convenience functions are:

* The make\_parser Method: Creates and returns a SAX XMLReader object. The first parser found will be used. If parser\_list is provided, it must be a sequence of strings which name modules that have a function named create\_parser(). Modules listed in parser\_list will be used before modules in the default list of parsers.
* The parse Method: Creates a SAX parser and use it to parse a document. The document, passed in as filename\_or\_stream, can be a filename or a file object. The handler parameter needs to be a SAX ContentHandler instance. If error\_handler is given, it must be a SAX ErrorHandler instance; if omitted, SAXParseException will be raised on all errors. There is no return value; all work must be done by the handler passed in.
* The parseString Method: Similar to parse(), but parses from a buffer string received as a parameter. string must be a str instance or a bytes-like object.

A typical SAX application uses three kinds of objects: readers, handlers and input sources.

“Reader” in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

### Parsing XML with DOM APIs

The Document Object Model, or “DOM,” is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program’s position in the document somewhere in your own code. SAX does not do it for you.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a DOMImplementation object class which provides access to Document creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing Document object. In Python, each DOM implementation will provide a function getDOMImplementation(). DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

The xml.dom contains the following functions:

1. xml.dom.registerDOMImplementation(name, factory): Register the factory function with the name name. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).
2. xml.dom.getDOMImplementation(name=None, features=()): Return a suitable DOM implementation. The name is either well-known, the module name of a DOM implementation, or None. If it is not None, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON\_DOM is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an ImportError. The features list must be a sequence of (feature, version) pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

|  |  |
| --- | --- |
| Constant | Description |
| xml.dom.EMPTY\_NAMESPACE | The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the namespaceURI of a node, or used as the namespaceURI parameter to a namespaces-specific method. |
| xml.dom.XML\_NAMESPACE | The namespace URI associated with the reserved prefix xml, as defined by Namespaces in XML (section 4). |
| xml.dom.XMLNS\_NAMESPACE | The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification. |
| xml.dom.XHTML\_NAMESPACE | The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language. |

In addition, xml.dom contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the nodeType attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.

### Parsing XML Document Using Python

While developing a web application in most cases we need to build and parse XML document using Python.

Python supports to work with various forms of structured data markup. This includes modules to work with the Hypertext Markup Language (HTML), Extensible Markup Language (XML).

In addition to parsing XML, xml.etree.ElementTree used for creating well-formed documents from Element objects in an application. The Element class used when a document is parsed also knows how to generate a serialized form of its contents, which can then be written to a file.

For creating element instance, we need to use the Element constructor or subelement()

**Example**(*to create an element instance factory function.*)**:**

import xml.etree.ElementTree as xml

filename = "/home/abc/Desktop/test\_xml.xml"

root = xml.Element("Users")

userelement = xml.Element("user")

root.append(userelement)

When you run this you can see the output as below:

<Users>

<user>

</user>

</Users>

Let us add user children.

**Example:**

uid = xml.SubElement(userelement, "uid")

uid.text = "1"

FirstName = xml.SubElement(userelement, "FirstName")

FirstName.text = "testuser"

LastName = xml.SubElement(userelement, "LastName")

LastName.text = "testuser"

Email = xml.SubElement(userelement, "Email")

Email.text = "testuser@test.com"

state = xml.SubElement(userelement, "state")

state.text = "xyz"

location = xml.SubElement(userelement, "location")

location.text = "abc"

tree = xml.ElementTree(root)

with open(filename, "w") as fh:

tree.write(fh)

First we create the root element by using ElmeentTree’sElement function. Then we create a user element and append it to the root. For creating SubElement we will use the Element object as first argument and name as second argument. Then for each SubElement, we set its text property to give it a value.

If you run this the response will be as follows:

<Users>

<user>

<uid>1</uid>

<FirstName>testuser</FirstName>

<LastName>testuser</LastName>

<Email>testuser@test.com</Email>

<state>xyz</state>

<location>abc</location>

</user>

</Users>

To parse the XML document, use the following code:

**Example:**

import xml.etree.ElementTree as ET

tree = ET.parse('Your\_XML\_file\_path')

root = tree.getroot()

Here getroot() will return the root Element of the XML document.

<Users version="1.0" language="SPA">

<user>

<uid>1</uid>

<FirstName>testuser</FirstName>

<LastName>testuser</LastName>

<Email>testuser@test.com</Email>

<state>xyz</state>

<location>abc</location>

</user>

</Users>

## Developing GUIs

### Overview

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.

* Tkinter − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this chapter.
* wxPython − This is an open-source Python interface for Windows http://wxpython.org.
* JPython − JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine http://www.jython.org.

There are many other interfaces available, which you can find on the Internet.

### Tkinter Programming

#### Overview

The tkinter package (“Tk interface”) is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running python -m tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

#### Tkinter Widgets

The tkinter.ttk module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if Tile has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for tkinter.ttk is to separate, to the extent possible, the code implementing a widget’s behavior from the code implementing its appearance.

To start using Ttk, import its module:

from tkinter import ttk

To override the basic Tk widgets, the import should follow the Tk import:

from tkinter import \*

from tkinter.ttk import \*

That code causes several tkinter.ttk widgets (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as “fg”, “bg” and others related to widget styling are no longer present in Ttk widgets. Instead, use the ttk.Style class for improved styling effects.

Ttk comes with 17 widgets, 11 of which already existed in tkinter: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar. The other six are new: Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview. And all them are subclasses of Widget.

ttk.Widget defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

The table below provides a brief description of the widgets:

|  |  |
| --- | --- |
| Widgets | Description |
| Button | Displays buttons in the application. |
| Checkbutton | Displays a number of options as checkboxes. The user can select multiple options at a time. |
| Entry | Displays a single-line text field for accepting values from a user. |
| Frame | Is used as a container widget to organize other widgets. |
| Label | Provides a single-line caption for other widgets. It can also contain images. |
| LabelFrame | Is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. |
| Menubutton | Displays menus in your application. |
| PanedWindow | Is a container widget that may contain any number of panes, arranged horizontally or vertically. |
| Radiobutton | Displays a number of options as radio buttons. The user can select only one option at a time. |
| Scale | Is used to provide a slider widget. |
| Scrollbar | Is used to add scrolling capability to various widgets, such as list boxes. |
| Combobox | Combines a text field with a pop-down list of values. This widget is a subclass of Entry. |
| Notebook | Manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window. |
| Progressbar | Shows the status of a long-running operation. It can operate in two modes:   1. The determinate mode which shows the amount completed relative to the total amount of work to be done and 2. The indeterminate mode which provides an animated display to let the user know that work is progressing. |
| Separator | Displays a horizontal or vertical separator bar. It has no other methods besides the ones inherited from ttk.Widget. |
| Sizegrip | Allows the user to resize the containing toplevel window by pressing and dragging the grip. |
| Treeview | Displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label. |

#### Standard Attributes

Some of the standard attributes available for the widgets are as follows:

* Dimensions
* Colors
* Fonts
* Anchors
* Relief styles
* Bitmaps
* Cursors

#### Geometry Management

Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual master. In contrast to the more cumbersome placer (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - above, to the left of, filling, etc - and works everything out to determine the exact placement coordinates for you.

**Note:** *Widgets do not appear until they have had their geometry specified with a geometry manager. It’s a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer’s pack() method applied to it.*

Tkinter exposes the following geometry manager classes:

* The pack() method: Controls where the widget is to appear within its container, and how it is to behave when the main application window is resized. It can be called with keyword-option/value pairs.
* The grid() method − Organizes widgets in a table-like structure in the parent widget.
* The place() method − Organizes widgets by placing them in a specific position in the parent widget.

## Sending Emails in Python (SMTP)

### Overview of SMTP

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.

The [smtplib](https://docs.python.org/3/library/smtplib.html#module-smtplib) module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon.

Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail −

import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local\_hostname]]] )

Here is the detail of the parameters −

* host − This is the host running your SMTP server. You can specify IP address of the host or a domain name like tutorialspoint.com. This is optional argument.
* port − If you are providing host argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
* local\_hostname − If your SMTP server is running on your local machine, then you can specify just localhost as of this option.

An SMTP object has an instance method called sendmail, which is typically used to do the work of mailing a message. It takes three parameters −

* The sender − A string with the address of the sender.
* The receivers − A list of strings, one for each recipient.
* The message − A message as a string formatted as specified in the various RFCs.

### Sending an E-mail using Python

Let’s see how to create and send a simple text message (both the text content and the addresses may contain unicode characters)

**Example:**

# Import smtplib for the actual sending function

import smtplib

# Import the email modules we'll need

from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.

with open(textfile) as fp:

# Create a text/plain message

msg = EmailMessage()

msg.set\_content(fp.read())

# me == the sender's email address

# you == the recipient's email address

msg['Subject'] = 'The contents of %s' % textfile

msg['From'] = me

msg['To'] = you

# Send the message via our own SMTP server.

s = smtplib.SMTP('localhost')

s.send\_message(msg)

s.quit()

### Sending Attachments as an E-mail

1. For adding an attachment, you need to import:

import smtplib

from email.mime.multipart import MIMEMultipart

from email.mime.text import MIMEText

from email.mime.base import MIMEBase

from email import encoders

These are some libraries which will make our work simple. These are the native libraries and you don’t need to import any external library for this.

1. Firstly, create an instance of MIMEMultipart, namely “msg” to begin with.
2. Mention the sender’s email id, receiver’s email id and the subject in the “From”, “To” and “Subject” key of the created instance “msg”.
3. In a string, write the body of the message you want to send, namely body. Now, attach the body with the instance msg using attach function.
4. Open the file you wish to attach in the “rb” mode. Then create an instance of MIMEBase with two parameters. First one is ‘\_maintype’ and the other one is ‘\_subtype’. This is the base class for all the MIME-specific sub-classes of Message.

Note that ‘\_maintype’ is the Content-Type major type (e.g. text or image), and ‘\_subtype’ is the Content-Type minor type (e.g. plain or gif or other media).

1. set\_payload is used to change the payload the encoded form. Encode it in encode\_base64. And finally attach the file with the MIMEMultipart created instance msg.

**Example:**

# Python code to illustrate Sending mail with attachments

# from your Gmail account

# libraries to be imported

import smtplib

from email.mime.multipart import MIMEMultipart

from email.mime.text import MIMEText

from email.mime.base import MIMEBase

from email import encoders

fromaddr = "EMAIL address of the sender"

toaddr = "EMAIL address of the receiver"

# instance of MIMEMultipart

msg = MIMEMultipart()

# storing the senders email address

msg['From'] = fromaddr

# storing the receivers email address

msg['To'] = toaddr

# storing the subject

msg['Subject'] = "Subject of the Mail"

# string to store the body of the mail

body = "Body\_of\_the\_mail"

# attach the body with the msg instance

msg.attach(MIMEText(body, 'plain'))

# open the file to be sent

filename = "File\_name\_with\_extension"

attachment = open("Path of the file", "rb")

# instance of MIMEBase and named as p

p = MIMEBase('application', 'octet-stream')

# To change the payload into encoded form

p.set\_payload((attachment).read())

# encode into base64

encoders.encode\_base64(p)

p.add\_header('Content-Disposition', "attachment; filename= %s" % filename)

# attach the instance 'p' to instance 'msg'

msg.attach(p)

# creates SMTP session

s = smtplib.SMTP('smtp.gmail.com', 587)

# start TLS for security

s.starttls()

# Authentication

s.login(fromaddr, "Password\_of\_the\_sender")

# Converts the Multipart msg into a string

text = msg.as\_string()

# sending the mail

s.sendmail(fromaddr, toaddr, text)

# terminating the session

s.quit()

**Notes:**

* *You can use loops to send mails to a number of people.*
* *This code is simple to implement. But it will not work if you have enabled 2-step verification on your gmail account. It is required to switch off the 2-step verification first.*
* *Using this method, Gmail will always put your mail in the primary section and the mails sent will not be Spam.*

## Integrating Python with other Language

Python is an advanced scripting language that is being used successfully to glue together large software components. It spans multiple platforms, middleware products, and application domains. Python is an object-oriented language with high-level data structures, dynamic typing, and dynamic binding. Python has been around since 1991, and has a very active user community.

Python is easily extensible with C/C++/Java code, and easily embeddable in applications. Python even uses Tk, the Tcl GUI toolkit, for a de-facto standard portable GUI toolkit. Unlike Tcl, however, Python supports object-oriented programming. Python programmers can create classes, use multiple inheritance, define methods, overload operators, and so on.

### C/C++ Tools

There a various tools which make it easier to bridge the gap between Python and C/C++:

|  |  |
| --- | --- |
| Tools | Description |
| Pyrex | Write your extension module on Python. |
| Cython | Is an improved version of Pyrex. |
| CXX | PyCXX is helper lib for writing Python extensions in C++. |
| ctypes | Is a Python module allowing to create and manipulate C data types in Python. These can then be passed to C-functions loaded from dynamic link libraries. |
| elmer | Compile and run python code from C, as if it was written in C |
| PicklingTools | Is a collection of libraries for exchanging Python Dictionaries between C++ and Python. |
| weave | Include C code lines in Python program (deprecated in favor of Cython) |
| ackward | Exposes parts of Python's standard library as idiomatic C++ |
| CFFI | Interact with almost any C code from Python, based on C-like declarations that you can often copy-paste from header files or documentation. |

### C/C++ BindingGenerators

Tools to make C/C++ functions/methods accessible from Python by generating binding (Python extension or module) from header files.

|  |  |
| --- | --- |
| Tools | Description |
| boost.python | Expose C++ classes functions and objects to Python, and vice-versa, using just C++ compiler. |
| PyAutoC | Automatically wrap C functions and structs, using just C compiler. |
| pwig | Is a SWIG extension for writing new language modules in Python. |
| PyBindGen | Python bindings code generator for pure C or C++ APIs. The generator is written in Python and has low complexity. The generated code is lean, efficient, and highly readable |
| shiboken | Binding Generator used to create PySide PYTHON bindings for Qt |
| SIP | Similar to SWIG but specialized for Python and C++.  Used to create PyQt, the Qt API wrapper library. |
| SWIG | Generate extension module from your .h files. |
| pybind11 | Similar to Boost.Python, but with a lean header-only implementation for C++11-capable compilers. |
| pyclif | It’s a Google tool. Similar to SWIG, but user-friendly and targeted only C++11 well-written libs. |
| cppyy | Interact with C/C++ through LLVMs JIT with automatic pythonizations for a "python feel." Support tools are available for large scale projects. |

### Java Integration

|  |  |
| --- | --- |
| Tools | Description |
| Jython | Python implemented in Java. |
| JPype | Allows Python to run java commands. |
| Jepp | Java embedded Python. |
| JCC | A C++ code generator for calling Java from C++/Python. |
| Javabridge | A package for running and interacting with the JVM from CPython. |
| py4j | Allows Python to run java commands. |
| voc | Part of[BeeWare](https://wiki.python.org/moin/BeeWare)suite. Converts python code to Java bytecode. |
| p2j | Converts Python code to Java. No longer developed. |

# Sources

<https://machinelearningmastery.com/best-programming-language-for-machine-learning/>

<http://www.cuelogic.com/blog/role-of-python-in-artificial-intelligence/>

<https://www.tutorialspoint.com/python3/python_environment.htm>

https://docs.anaconda.com/anaconda/install/windows

<https://www.w3schools.in/python-tutorial/functions/>

<https://docs.python.org/3/library/stdtypes.html#functions>

<https://docs.python.org/3/reference/datamodel.html#types>

<https://www.tutorialspoint.com/python/python_functions.htm>

<https://www.python-course.eu/python3_functions.php>

<https://www.tutorialspoint.com/python/python_functions.htm>

<https://www.tutorialspoint.com/python/python_functions.htm>

<https://www.programiz.com/python-programming/recursion>

<https://docs.python.org/3/tutorial/modules.html>

<https://docs.python.org/3/tutorial/modules.html#standard-modules>

<https://www.programiz.com/python-programming/file-operation>

<https://www.w3schools.in/python-tutorial/file-handling/>

<https://www.w3schools.in/python-tutorial/file-handling/>

<http://www.pythonforbeginners.com/cheatsheet/python-file-handling>

<https://www.geeksforgeeks.org/os-module-python-examples/>

<https://docs.python.org/3/library/shutil.html#archiving-operations>

<https://www.w3schools.in/python-tutorial/exception-handling/>

<https://www.tutorialspoint.com/python/python_exceptions.htm>

<https://docs.python.org/3/library/exceptions.html#bltin-exceptions>

<https://www.tutorialspoint.com/python/python_exceptions.htm>

<https://wiki.python.org/moin/HandlingExceptions>

<https://docs.python.org/3.6/library/getopt.html>

<https://docs.python.org/3/howto/functional.html>

<https://www.geeksforgeeks.org/python-closures/>

<https://www.w3schools.in/python-tutorial/cgi-programming/>

<https://www.w3schools.in/python-tutorial/network-programming/>

<https://www.tutorialspoint.com/python/python_networking.htm>

<https://docs.python.org/3/library/socket.html?highlight=socket%20programming#socket.gethostname>

<https://www.w3schools.in/python-tutorial/network-programming/>

<https://www.geeksforgeeks.org/xml-parsing-python/>

<https://micropyramid.com/blog/building-and-parsing-xml-document-using-python/>

<https://www.tutorialspoint.com/python/python_gui_programming.htm>

<https://docs.python.org/3/library/tkinter.ttk.html>

* <https://www.tutorialspoint.com/python/python_gui_programming.htm>

<https://www.tutorialspoint.com/python/python_gui_programming.htm>

<https://www.tutorialspoint.com/python/python_gui_programming.htm>

<https://www.tutorialspoint.com/python/python_sending_email.htm>