Using Python libraries

This lab covers

- [Managing various data types---strings, numbers, and more]
- [Manipulating files and storage]
- [Accessing operating system services]
- [Using internet protocols and formats]
- [Developing and debugging tools]
- [Accessing PyPI (a.k.a. "The Cheese Shop")]
- [Installing Python libraries and virtual environments using pip and venv]

Python has long proclaimed that one of its key advantages is its "batteries included" philosophy. This means that a stock install of Python comes with a rich standard library that lets you handle a wide variety of situations without the need to install additional libraries. This lab gives you a high-level survey of some of the contents of the standard library, as well as some suggestions on finding and installing external modules.

"Batteries included": The standard library

In Python, what's considered to be the *library* consists of several components, including built-in data types and constants that can be used without an [import] statement, such as numbers and lists, as well as some built-in functions and exceptions. The largest part of the library is an extensive collection of modules. If you have Python, you also have libraries to manipulate diverse types of data and files, to interact with your operating system, to write servers and clients for many internet protocols, and to develop and debug your code.

What follows is a survey of the high points. Although most of the major modules are mentioned, for the most complete and current information I recommend that you spend time on your own exploring the library reference that's part of the Python documentation. In particular, before you go in search of an external library, be sure to scan through what Python already offers. You may be surprised by what you find.

Managing various data types

The standard library naturally contains support for Python's built-in types, which I touch on in this section. In addition, three categories in the standard library deal with various data types: string services, data types, and numeric modules.

String services include the modules in [table 19.1] that deal with bytes as well as strings. The three main things these modules deal with are strings and text, sequences of bytes, and Unicode operations.

Table 19.1. String services modules

Module Description and possible uses

---

string Compare with string constants, such as digits or whitespace; format strings (see [chapter 6]) re Search and replace text using regular expressions (see [chapter 16]) struct Interpret bytes as packed binary data, and read and write structured data to/from files difflib Use helpers for computing deltas, find differences between strings or sequences, and create patches and diff files textwrap Wrap and fill text, and format text by breaking lines or adding spaces

The data types category is a diverse collection of modules covering various data types, particularly time, date, and collections, as shown in [table 19.2].

Table 19.2. Data types modules

Module Description and possible uses

---

datetime, calendar	Date, time, and calendar operations
collections	Container data types
enum	Allows creation of enumerator classes that bind symbolic names to constant values
array	Efficient arrays of numeric values
sched	Event scheduler
queue	Synchronized queue class
copy	Shallow and deep copy operations
pprint	Data pretty printer
typing	Support for annotating code with hints as to the types of objects, particularly of function parameters and return values

As the name indicates, the numeric and mathematical modules deal with numbers and mathematical operations, and the most common of these modules are listed in [table 19.3]. These modules have everything you need to create your own numeric types and handle a wide range of math operations.

Table 19.3. Numeric and mathematical modules

Module	Description and possible uses
numbers	Numeric abstract base classes
math, cmath	Mathematical functions for real and complex numbers
decimal	Decimal fixed-point and floating-point arithmetic
statistics	Functions for calculating mathematical statistics
fractions	Rational numbers
random	Generate pseudorandom numbers and choices, and shuffle sequences
itertools	Functions that create iterators for efficient looping
functools	Higher-order functions and operations on callable objects
operator	Standard operators as functions

Manipulating files and storage

Another broad category in the standard library covers files, storage, and data persistence and is summarized in [table 19.4]. This category ranges from modules for file access to modules for data persistence and compression and handling special file formats.

Table 19.4. File and storage modules

Module	Description and possible uses
os.path	Perform common pathname manipulations
pathlib	Deal with pathnames in an object-oriented way
fileinput	Iterate over lines from multiple input streams
filecmp	Compare files and directories
tempfile	Generate temporary files and directories
glob, fnmatch	Use UNIX-style pathname and filename pattern handling
linecache	Gain random access to text lines
shutil	Perform high-level file operations
pickle, shelve	Enable Python object serialization and persistence
sqlite3	Work with a DB-API 2.0 interface for SQLite databases
zlib, gzip, bz2, zipfile, tarfile	Work with archive files and compressions
csv	Read and write CSV files
configparser	Use a configuration file parser; read/write Windows-style configuration .ini files

Accessing operating system services

This category is another broad one, containing modules for dealing with your operating system. As shown in [table 19.5], this category includes tools for handling command-line parameters, redirecting file and print output and input, writing to log files, running multiple threads or processes, and loading non-Python (usually, C) libraries for use in Python.

Table 19.5. Operating system modules

Module	Description
os	Miscellaneous operating system interfaces
io	Core tools for working with streams
time	Time access and conversions
optparse	Powerful command-line option parser
logging	Logging facility for Python
getpass	Portable password input
curses	Terminal handling for character-cell displays
platform	Access to underlying platform's identifying data
ctypes	Foreign function library for Python
select	Waiting for I/O completion
threading	Higher-level threading interface
multiprocessing	Process-based threading interface
subprocess	Subprocess management

Using internet protocols and formats

The internet protocols and formats category is concerned with encoding and decoding the many standard formats used for data exchange on the internet, from MIME and other encodings to JSON and XML. This category also has modules for writing servers and clients for common services, particularly HTTP, and a generic socket server for writing servers for custom services. The most commonly used of these modules are listed in [table 19.6].

Table 19.6. Modules supporting internet protocols and formats

Module	Description
socket, ssl	Low-level networking interface and SSL wrapper for socket objects
email	Email and MIME handling package
json	JSON encoder and decoder
mailbox	Manipulate mailboxes in various formats
mimetypes	Map filenames to MIME types
base64, binhex, binascii, quopri, uu	Encode/decode files or streams with various encodings
html.parser, html.entities	Parse HTML and XHTML
xml.parsers.expat, xml.dom, xml.sax, xml.etree.ElementTree	Various parsers and tools for XML
cgi, cgitb	Common Gateway Interface support
wsgiref	WSGI utilities and reference implementation
urllib.request, urllib.parse	Open and parse URLs
ftplib, poplib, imaplib, nntplib, smtplib, telnetlib	Clients for various internet protocols
socketserver	Framework for network servers
http.server	HTTP servers
xmlrpc.client, xmlrpc.server	XML-RPC client and server

Development and debugging tools and runtime services

Python has several modules to help you debug, test, modify, and otherwise interact with your Python code at runtime. As shown in [table 19.7], this category includes two testing tools, profilers, modules to interact with error tracebacks, the interpreter's garbage collection, and so on, as well as modules that let you tweak the importing of other modules.

Table 19.7. Development, debugging, and runtime modules

Module	Description
pydoc	Documentation generator and online help system
doctest	Test interactive Python examples
unittest	Unit testing framework
test.support	Utility functions for tests
pdb	Python debugger
profile, cProfile	Python profilers
timeit	Measure execution time of small code snippets
trace	Trace or track Python statement execution
sys	System-specific parameters and functions
atexit	Exit handlers
__future__	Future statement definitions---features to be added to Python
gc	Garbage collector interface
inspect	Inspect live objects
imp	Access the import internals
zipimport	Import modules from zip archives
modulefinder	Find modules used by a script

Moving beyond the standard library

Although Python's "batteries included" philosophy and well-stocked standard library mean that you can do a lot with Python out of the box, there will inevitably come a situation in which you need some functionality that doesn't come with Python. This section surveys your options when you need to do something that isn't in the standard library.

Adding more Python libraries

Finding a Python package or module can be as easy as entering the functionality you're looking for (such as [mp3 tags] and [Python]) in a search engine and then sorting through the results. If you're lucky, you may find the module you need packaged for your OS---with an executable Windows or macOS installer or a package for your Linux distribution.

This technique is one of the easiest ways to add a library to your Python installation, because the installer or your package manager takes care of all the details of adding the module to your system correctly. It can also be the answer for installing more complex libraries, such as scientific libraries with complex build requirements and dependencies.

In general, except for scientific libraries, such prebuilt packages aren't the rule for Python software. Such packages tend to be a bit older, and they offer less flexibility in where and how they're installed.

Installing Python libraries using pip and venv

If you need a third-party module that isn't prepackaged for your platform, you'll have to turn to its source distribution. This fact presents a couple of problems:

- [To install the module, you must find and download it.]
- [Installing even a single Python module correctly can involve a certain amount of hassle in dealing with Python's paths and your system's permissions, which makes a standard installation system helpful.]

Python offers [pip] as the current solution to both problems. [pip] tries to find the module in the Python Package index (more about that soon), downloads it and any dependencies, and takes care of the installation. The basic syntax of [pip] is quite simple. To install the popular requests library from the command line, for example, all you have to do is

```
$ python3.6 -m pip install requests
```

Upgrading to the library's latest version requires only the addition of the [---upgrade] switch:

```
$ python3.6 -m pip install --upgrade requests
```

Finally, if you need to specify a particular version of a package, you can append it to the name like this:

```
$ python3.6 -m pip install requests==2.11.1
$ python3.6 -m pip install requests>=2.9
```

Installing with the --user flag

On many occasions, you can't or don't want to install a Python package in the main system instance of Python. Maybe you need a bleeding-edge version of the library, but some other application (or the system itself) still uses an older version. Or maybe you don't have access privileges to modify the system's default Python. In cases like those, one answer is to install the library with the [---user] flag. This flag installs the library in the user's home directory, where it's not accessible by any other users. To install [requests] for only the local user:

```
$ python3.6 -m pip install --user requests
```

As I mentioned previously, this scheme is particularly useful if you're working on a system on which you don't have sufficient administrator rights to install software, or if you want to install a different version of a module. If your needs go beyond the basic installation methods discussed here, a good place to start is "Installing Python Modules," which you can find in the Python documentation.

Virtual environments

You have another, better option if you need to avoid installing libraries in the system Python. This option is called a virtual environment (virtualenv). A *virtual environment* is a self-contained directory structure that contains both an installation of Python and its additional packages. Because the entire Python environment is contained in the virtual environment, the libraries and modules installed there can't conflict with those in the main system or in other virtual environments, allowing different applications to use different versions on both Python and its packages.

Creating and using a virtual environment takes two steps. First, you create the environment:

```
$ python3.6 -m venv test-env
```

This step creates the environment with Python and [pip] installed in a directory called test-env. Then, when the environment is created, you activate it. On Windows, you do this:

```
> test-env\Scripts\activate.bat
```

On Unix or MacOS systems, you source the activate script:

```
$ source test-env/bin/activate
```

When you've activated the environment, you can use [pip] to manage packages as earlier, but in the virtual environment [pip] is a standalone command:

```
$ pip install requests
```

In addition, whatever version of Python you used to create the environment is the default Python for that environment, so you can use just [python] instead of [python3] or [python3.6].

Virtual environments are very useful for managing projects and their dependencies and are very much a standard practice, particularly for developers working on multiple projects. For more information, look at the "[Virtual Environments and Packages]" section of the Python tutorial in the Python online documentation.

PyPI (a.k.a. "The Cheese Shop")

Although [distutils] packages get the job done, there's one catch: You have to find the correct package, which can be a chore. And after you've found a package, it would be nice to have a reasonably reliable source from which to download that package.

To meet this need, various Python package repositories have been made available over the years. Currently, the official (but by no means the only) repository for Python code is the Python Package Index, or PyPI (formerly also known as "The Cheese Shop," after the Monty Python sketch) on the Python website. You can access it from a link on the main page or directly at https://pypi.python.org. PyPI contains more than 6,000 packages for various Python versions, listed by date added and name, but also searchable and broken down by category.

At this writing, a new version of PyPI is in the wings; currently, it's called "The Warehouse." This version is still in testing but promises to provide a much smoother and friendlier search experience.

PyPI is the logical next stop if you can't find the functionality you want with a search of the standard library.

Summary

- [Python has a rich standard library that covers more common situations than many other languages, and you should check what's in the standard library carefully before looking for external modules.]
- [If you do need an external module, prebuilt packages for your operating system are the easiest option, but they're sometimes older and often hard to find.]
- [The standard way to install from source is to use pip, and the best way to prevent conflicts among multiple projects is to create virtual environments with the venv module.]
- [Usually, the logical first step in searching for external modules is the Python Package Index (PyPI).]