

Setting up Python for NLP

All the code examples showcased in this chapter are available on the book's official GitHub repository which you can access here: <https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition>.

Getting to know Python

Before we can dive into the Python ecosystem and look at the various components associated with it, we will take a brief look at the origins and philosophy behind Python, and how it has evolved over time to be the choice of language powering many applications, servers and systems today. Python is a high-level open source general purpose programming language widely used as a scripting as well as a programming language across different domains. Python is the brainchild of Guido Van Rossum and was conceived in the late 1980s as a successor to the ABC language, both developed at the Centrum Wiskunde and Informatica (CWI), Netherlands. Python was originally designed to be a scripting and interpreted language and to this day it still retains the essence of being one of the most popular scripting languages out there but with Object Oriented Principles (OOP) and constructs, you can use it just like any other object oriented language e.g. Java. The name Python coined by Guido for the language actually does not come from the snake but the hit comedy show, Monty Python's Flying Circus since he was a big fan of the same.

Like we mentioned before, Python is a general purpose programming language which supports multiple programming paradigms. The popular programming paradigms supported are mentioned as follows.

- Object Oriented Programming
- Functional Programming
- Procedural Programming
- Aspect Oriented Programming

A lot of Object Oriented Programming concepts are present in Python including classes, objects, data and methods. Principles like abstraction,

encapsulation, inheritance and polymorphism can also be implemented and exhibited using Python. There are several advanced features in Python including iterators, generators, list comprehensions, lambda expressions and several modules like collections, itertools and functools which provide the ability to write code following the Functional Programming paradigm. Python has been designed keeping in mind that simple and beautiful code is more elegant and easy to use rather than doing premature optimization and writing hard to interpret code.

Python's standard libraries are power-packed with a wide variety of capabilities and features ranging from low level hardware interfacing to handling files and working with text data. Easy extensibility and integration was considered when developing Python such that it can be easily integrated with existing applications and even rich application programming interfaces (APIs) can be created to provide interfaces to other applications and tools. Python also has a thriving and helpful developer community which makes sure there is a ton of helpful resources and documentation out there on the internet and also organizes various workshops and conferences throughout the world!

Besides being a multi-purpose language, the wide variety of frameworks, libraries and platforms which have been developed by using Python and to be used for Python form a complete robust ecosystem around Python. These libraries make our life easier by giving us a wide variety of capabilities and functionality to perform various tasks with minimal code. Some examples would be libraries for handling databases, text data, machine learning, signal processing, image processing, deep learning, artificial intelligence and the list goes on.

The Zen of Python

You might be wondering what on earth the Zen of Python might be. However if you are somewhat familiar with Python, this is one of the first things you might get to know. The beauty of Python lies in its simplicity and elegance. "*The Zen of Python*" is a set of 20 guiding principles, also known as aphorisms, which have been influential behind Python's design. Long time Pythoneer Tim Peters however documented only 19 of them in 1999 and they can be accessed here <https://hg.python.org/peps/file/tip/pep-0020.txt> as a part of the Python Enhancement Proposals (PEP) number 20 (PEP 20). The best part is, if you already have Python installed, you can access the "Zen of Python" by running the following code in the Python or IPython shell or a Jupyter notebook.

```
# zen of python
import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
```

```

Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

```

The above output shows us the 19 principles which form the Zen of Python and is included in the Python language itself as an easter egg. The principles are written in simple English and a lot of them are pretty self-explanatory even if you have not written code before and many of them contain inside jokes! Python focuses on writing simple and clean code which is readable. It also mentions to make sure you focus a lot on error handling and implementing code which is easy to interpret and understand. The one principle I would like you to remember is "Simple is better than complex" which is applicable not only for Python but for a lot of things especially when you are out there in the world solving problems. Sometimes a simple approach beats a more complex one as long as you know what you are doing and it helps you from not overcomplicating things.

Setup a Robust Python Environment

Now that you have been acquainted with Python and know more about the language, its capabilities, implementations and versions, we will be covering some essentials on how to setup your development environment and handle package management and virtual environments. This section will give you a good head start on getting things ready for following along with the various hands-on examples which we will be covering in this book.

Recommended Setup (Use Google Colab)

We recommend running these notebooks in Google Colab so you don't have to install dependencies manually in your environment or face challenges with a few dependencies on windows environments.

Google provides you with an easy to use free python environment with jupyter notebooks to run your code.

You can go to <https://colab.research.google.com/> and open up a new notebook and start working on it just like a Jupyter notebook or upload an already existing notebook into Colab using the upload option and start working on it in the Colab environment.

Standalone Setup: Which Python version?

We have previously talked about two major Python versions one being the 2.x series and the other being 3.x series. Both the versions are quite similar however, there have been several backward incompatible changes in the 3.x version which has led to a huge drift between people who use 2.x and people who use 3.x since most legacy code and a large majority of Python packages on PyPI are developed in Python 2.7.x and the package owners either do not have the time or will to port all their codebases to Python 3.x since the effort required will not be minimal. Some of the changes in 3.x are mentioned as follows.

- All text strings are Unicode by default
- `print` and `exec` are now functions and no longer statements
- Several methods like `range()` returns a memory-efficient iterable instead of a list
- The style for classes have changed
- Library and name changes based on convention and style violations
- More features, modules and enhancements

To know more about all the changes introduced in Python 3.x you can check this link here <https://docs.python.org/3/whatsnew/3.7.html> which is the official documentation listing the changes. This should give you a pretty good idea of what changes can break your code if you are porting it from Python 2 to Python 3.

Now addressing the problem of selecting which version, we would definitely recommend to using Python 3 at all times. The primary reason behind this is a recent announcement from the Python core group members which mentioned that support for Python 2 will be ending by 2020 and no new features or enhancements will be pushed to Python 2. We recommend checking out PEP 373 <https://legacy.python.org/dev/peps/pep-0373/> which talked about the same in further detail. However if you are working on a large legacy codebase with Python 2.x, you might need to stick to it until porting the same to Python 3 is possible. We will be using Python 3.x in this book and recommend you to do the same. For code in Python 2.x, you can refer to the previous release of this book as needed.

Which Operating System?

There are several popular Operating Systems (OS) out there and each person has their own preference. The beauty of Python is that it can run seamlessly on any OS out there without much hassle. Some of the different OS which are used the most include,

- Windows
- Linux
- macOS (also known as Mac OS and OS X)

You can choose any OS of your choice and use it for following along with the examples. We usually use a combination of Linux and Windows as our OS platforms. Usually Python external packages are really easy to install on UNIX based OS like Linux and macOS. However sometimes there are major issues in installing them for Windows, so we want to highlight such instances and make sure we address them such that executing any of the code snippets and samples here becomes easy for the readers. You are most welcome to use any OS of your choice when following the examples in this book!

Integrated Development Environments

Integrated Development Environments (IDEs) are software products which enable developers to be highly productive by providing a complete suite of tools and capabilities necessary for writing, managing and executing code. The usual components of an IDE include a source editor, debugger, compiler, interpreter, refactoring and build tools. They also have other capabilities like code-completion, syntax highlighting, error highlighting and checks, objects and variable explorers. IDEs can be used to manage entire codebases and is much better than trying to write code in a simple text editor which takes more time. However more experienced developers often use simple plain text editors to write code especially if they are working in server environments. The link mentioned here <https://wiki.python.org/moin/IntegratedDevelopmentEnvironments> provides a list of IDEs used specially for Python. We usually use a combination of PyCharm, Spyder and Jupyter notebooks. The code examples in this book will be demonstrated on Jupyter notebooks mostly, which usually comes pre-installed along with the Anaconda Python distribution for writing and executing our code.

Standalone Environment Setup

In this section, we will cover details regarding how to setup your Python environment with minimal effort and the main components required for the same. You can head over to the official python website and download Python 3.7 from <https://www.python.org/downloads/> or you can download a complete Python distribution with over hundreds of packages specially built for data science and AI, known as the Anaconda Python distribution from Anaconda (formerly known as Continuum Analytics). This provides a lot of advantages especially to Windows users where installing some of the packages like numpy and scipy can sometimes cause major issues. You can get more information about Anaconda and the excellent

■ Setting up Python for NLP

work they are doing by visiting <https://www.anaconda.com> which also tells us that Anaconda comes with conda an open source package and environment management system and also Spyder (Scientific Python Development Environment) an IDE for writing and executing your code.

To start our environment setup, you can follow along the instructions mentioned at <https://docs.anaconda.com/anaconda/install/windows> for Windows or use the instructions for any other OS of your choice. Head over to <https://www.anaconda.com/download/> and download the 64 or 32 bit Python 3 installer for Windows depending on your OS version. For other OS you can check the relevant instructions on the website. Start the executable and follow the instructions on the screen clicking the next button at each stage. Remember to set a proper install location as depicted in Figure 2-1.

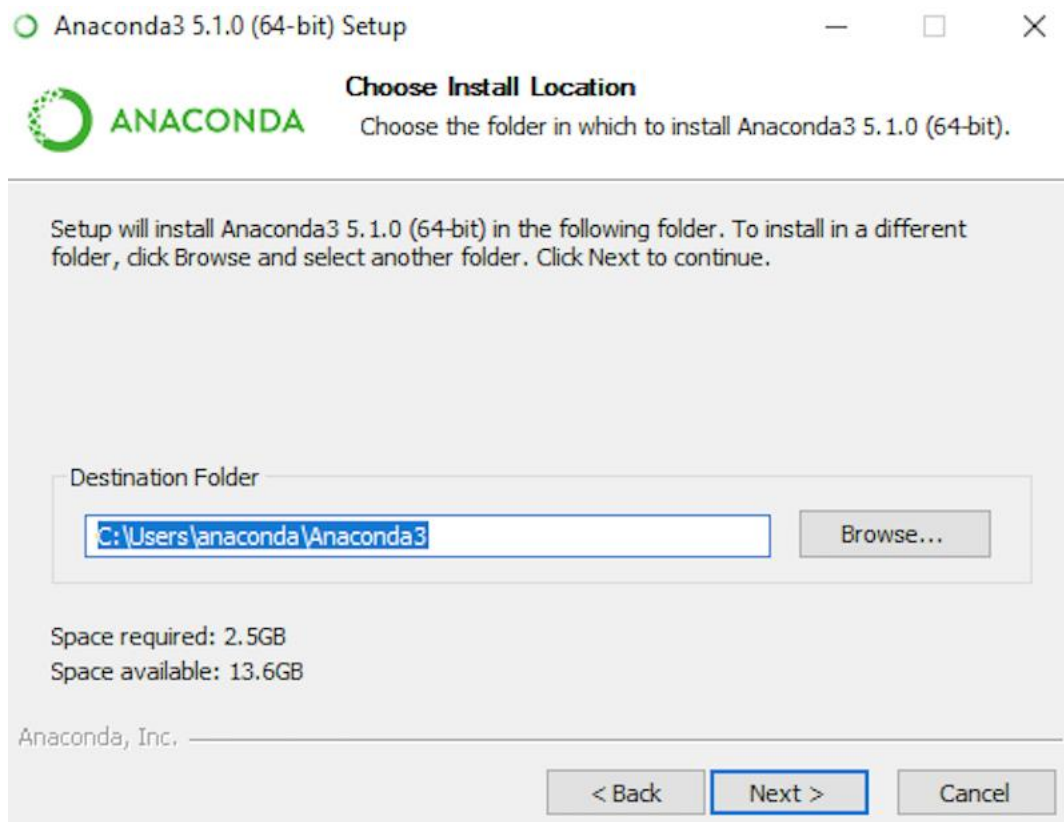


Figure 2-1. Installing the Anaconda Python distribution - setting up an install location

Before starting the actual installation, remember to check the following two options as shown in Figure 2-2.

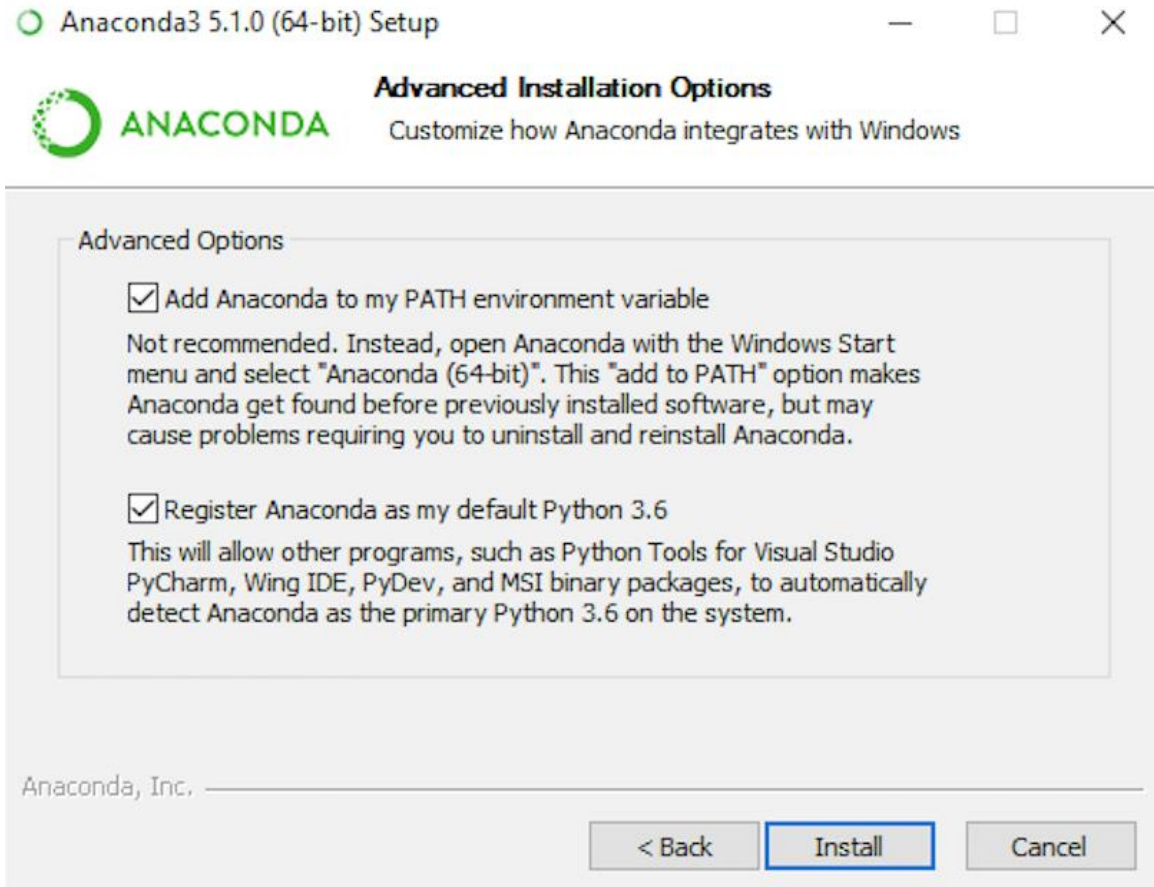
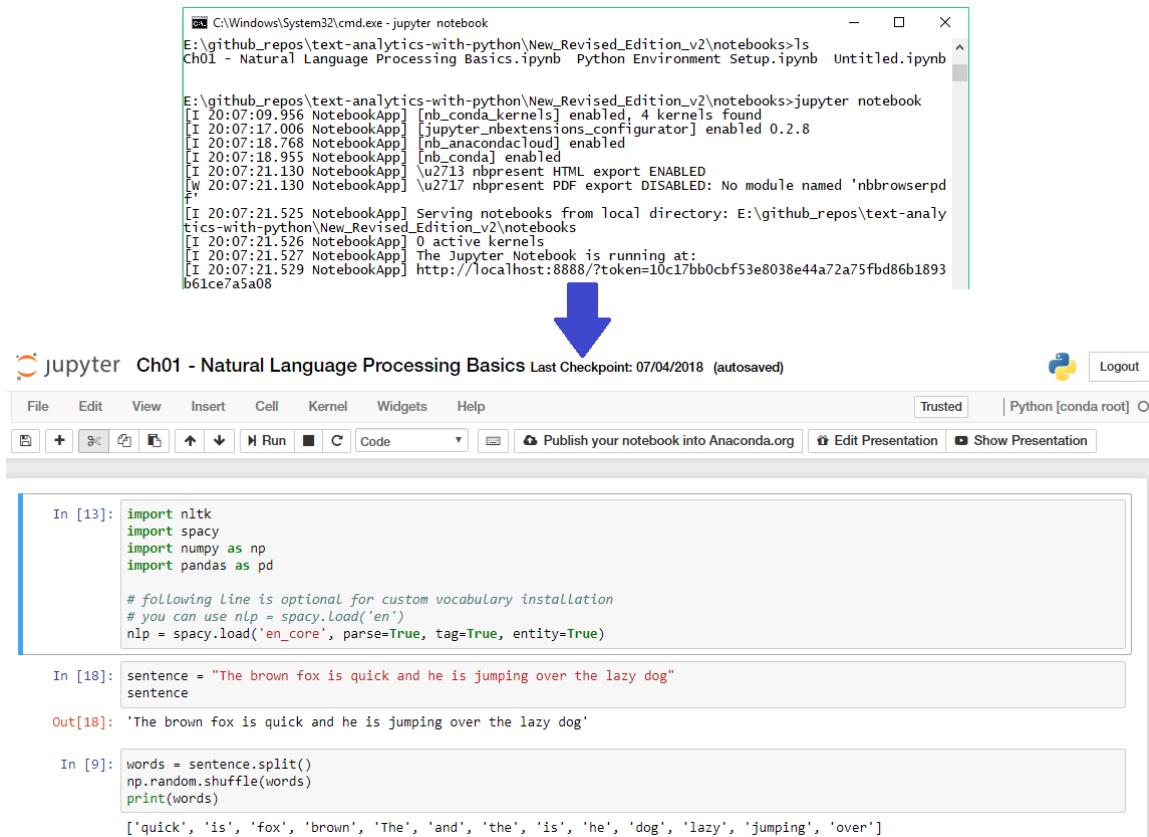


Figure 2-2. *Installing the Anaconda Python distribution - add to system path*

Once the installation is complete, you can either start up Spyder by double clicking the relevant icon or start the Python or IPython shell from the command prompt. For starting Jupyter notebooks you can just type `jupyter notebook` from a terminal or command prompt. Spyder provides you a complete IDE to write and execute code in both the regular Python and IPython shell. Figure 2-3 shows us how to start a jupyter notebook.

■ Setting up Python for NLP



```

C:\Windows\System32\cmd.exe - jupyter notebook
E:\github_repos\text-analytics-with-python\New_Revised_Edition_v2\notebooks>ls
Ch01 - Natural Language Processing Basics.ipynb  Python Environment Setup.ipynb  Untitled.ipynb

E:\github_repos\text-analytics-with-python\New_Revised_Edition_v2\notebooks>jupyter notebook
[I 20:07:09.956 NotebookApp] [nb_conda_kernels] enabled, 4 kernels found
[I 20:07:17.006 NotebookApp] [jupyter_nbextensions_configurator] enabled 0.2.8
[I 20:07:18.768 NotebookApp] [nb_anacondacloud] enabled
[I 20:07:18.955 NotebookApp] [nb_conda] enabled
[I 20:07:21.130 NotebookApp] \u2713 nbpresent HTML export ENABLED
[W 20:07:21.130 NotebookApp] \u2717 nbpresent PDF export DISABLED: No module named 'nbbrowserpd
f'
[I 20:07:21.525 NotebookApp] Serving notebooks from local directory: E:\github_repos\text-analy
tics-with-python\New_Revised_Edition_v2\notebooks
[I 20:07:21.526 NotebookApp] 0 active kernels
[I 20:07:21.527 NotebookApp] The Jupyter Notebook is running at:
[W 20:07:21.529 NotebookApp] http://localhost:8888/?token=10c17bb0cbf53e8038e44a72a75fbd86b1893
b61ce7a5a08

jupyter Ch01 - Natural Language Processing Basics Last Checkpoint: 07/04/2018 (autosaved)
Python [conda root]

File Edit View Insert Cell Kernel Widgets Help Trusted Python [conda root]

In [13]: import nltk
import spacy
import numpy as np
import pandas as pd

# following line is optional for custom vocabulary installation
# you can use nlp = spacy.load('en')
nlp = spacy.load('en_core', parse=True, tag=True, entity=True)

In [18]: sentence = "The brown fox is quick and he is jumping over the lazy dog"
sentence

Out[18]: 'The brown fox is quick and he is jumping over the lazy dog'

In [9]: words = sentence.split()
np.random.shuffle(words)
print(words)

['quick', 'is', 'fox', 'brown', 'The', 'and', 'the', 'is', 'he', 'dog', 'lazy', 'jumping', 'over']
```

Figure 2-3. Starting a Jupyter notebook

This should give you an idea how easy it is to showcase code and outputs interactively in Jupyter notebooks. We will be showcasing our code usually through these notebooks. To test if python is properly installed you can just type `python --version` from the terminal or even from your notebook as depicted in Figure 2-4.

```
In [1]: !python --version
Python 3.5.2 :: Anaconda custom (64-bit)

In [2]: print('Welcome to Python')
Welcome to Python
```


Figure 2-4. Checking your Python installation

Package Management

We will now cover package management briefly, you can use either the pip or conda command to install, uninstall and upgrade packages. The following shell command depicts installing the pandas library via pip. You can check if a package is installed using the pip freeze <package_name> command and install packages using the pip install <package_name> command as depicted in Figure 2-5. If we already have a package/library installed, you can use the --upgrade flag.

```
In [3]: !pip freeze | grep pandas
pandas==0.20.3

You are using pip version 9.0.1, however version 10.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

In [4]: !pip install pandas
Requirement already satisfied: pandas in c:\program files\anaconda3\lib\site-packages
Requirement already satisfied: python-dateutil>=2 in c:\program files\anaconda3\lib\site-packages (from pandas)
Requirement already satisfied: pytz>=2011k in c:\program files\anaconda3\lib\site-packages (from pandas)
Requirement already satisfied: numpy>=1.7.0 in c:\program files\anaconda3\lib\site-packages (from pandas)
Requirement already satisfied: six>=1.5 in c:\program files\anaconda3\lib\site-packages (from python-dateutil>=2->pandas)

You are using pip version 9.0.1, however version 10.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

In [4]: !pip install pandas --upgrade
Collecting pandas
  Downloading https://files.pythonhosted.org/packages/24/f1/bbe61db3ab675ae612d5261e69cff05f1ff0a7638469a9faalc9cdclcd5/pandas-0.23.3-cp35-cp35m-win_amd64.whl (7.6MB)
Collecting python-dateutil>=2.5.0 (from pandas)
  Downloading https://files.pythonhosted.org/packages/cf/f5/af2b09c957ace60dcfac112b669c45c8c97e32f94aa8b56da4c6d1682825/python_dateutil-2.7.3-py2.py3-none-any.whl (211kB)
Collecting numpy>=1.9.0 (from pandas)
  Downloading https://files.pythonhosted.org/packages/f3/71/94628784c3f07d4bc0dd38f8753e3f751d66cfd5a6823591179608c27f09/numpy-1.14.5-cp35-none-win_amd64.whl (13.4MB)
Collecting pytz>=2011k (from pandas)
  Downloading https://files.pythonhosted.org/packages/30/4e/27c34b62430286c6d59177a0842ed90dc789ce5d1ed740887653b898779a/pytz-2018.5-py2.py3-none-any.whl (510kB)
Collecting six>=1.5 (from python-dateutil>=2.5.0->pandas)
  Downloading https://files.pythonhosted.org/packages/67/4b/141a581104b1f6397bfa78ac9d43d8ad29a7ca43ea90a2d863fe3056e86a/six-1.11.0-py2.py3-none-any.whl
Installing collected packages: six, python-dateutil, numpy, pytz, pandas
```

Figure 2-5. Python package management with pip

The conda package manager is better than pip in several aspects since it provides a holistic view of what dependencies are going to be upgraded and the specific versions and other details during installation. Also pip often fails to install some packages in Windows however conda usually has no such issues during installation. Figure 2-6 depicts how to install and manage packages using conda.

■ Setting up Python for NLP

```
C:\> conda install pandas
Solving environment: done

## Package Plan ##

environment location: C:\Program Files\Anaconda3

added / updated specs:
- pandas

The following packages will be downloaded:

package                        | build
-----|-----
pandas-0.23.3                  | py35_0      8.6 MB  conda-forge

The following packages will be UPDATED:

pandas: 0.20.3-py35_1 conda-forge --> 0.23.3-py35_0 conda-forge

Proceed ([y]/n)? y

Downloading and Extracting Packages
pandas-0.23.3      | 8.6 MB | ##### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

!pip freeze | grep pandas
pandas==0.23.3
```

Figure 2-6. Python package management with conda

Now you have a much better idea of how to install external packages and libraries in Python. This will be useful later whenever you want to install any external python packages in your environment. Your Python environment should now be setup and ready for executing code. Before we dive into techniques for handling text data in Python, we will conclude with a discussion about virtual environments.

Virtual Environments (Optional)

A virtual environment also termed as **venv** is a complete isolated Python environment with its own Python interpreter, libraries, modules and scripts. This

environment is a standalone environment isolated from other virtual environments and the default system level Python environment. Virtual environments are extremely useful when you have multiple projects or codebases which have dependencies on different versions of the same packages or libraries. For example if my project TextApp1 depends on nltk 2.0 and another project TextApp2 depends on nltk 3.0 then it would be impossible to run both the projects on the same system. Hence the need for virtual environments which provide complete isolated environments which can be activated and deactivated as needed.

To setup a virtual environment, you need the virtualenv package. We will create a new directory also where we want to keep our virtual environment and install virtualenv as follows.

```
E:\>mkdir Apress
E:\>cd Apress
E:\Apress>pip install virtualenv
```

```
Collecting virtualenv
Installing collected packages: virtualenv
Successfully installed virtualenv-16.0.0
```

Once installed, you can create a virtual environment as follows where we create a new project directory called test_proj and create the virtual environment inside the directory.

```
E:\Apress>mkdir test_proj && chdir test_proj
E:\Apress\test_proj>virtualenv venv
```

```
Using base prefix 'c:\\program files\\anaconda3'
New python executable in E:\Apress\test_proj\venv\Scripts\python.exe
Installing setuptools, pip, wheel...done.
```

Now that you have installed the virtual environment successfully, let's try and observe the major differences between the global system python environment and our virtual environment. If you remember, we recently updated our global system python's pandas package version to 0.23 in the previous section. We can verify it using the following commands.

```
E:\Apress\test_proj>echo 'This is Global System Python'
'This is Global System Python'
```

```
E:\Apress\test_proj>pip freeze | grep pandas
pandas==0.23.3
```

Now supposed we wanted an older version of pandas in our virtual environment but we don't want to affect our global system python environment, we can do so by activating our virtual environment and installing pandas.

```
E:\Apress\test_proj>venv\Scripts\activate
(venv) E:\Apress\test_proj>echo 'This is VirtualEnv Python'
'This is VirtualEnv Python'
```

```
(venv) E:\Apress\test_proj>pip install pandas==0.21.0
```

■ Setting up Python for NLP

```
Collecting pandas==0.21.0
  100% |#####| 9.0MB 310kB/s
Collecting pytz>=2011k (from pandas==0.21.0)
Collecting python-dateutil>=2 (from pandas==0.21.0)
Collecting numpy>=1.9.0 (from pandas==0.21.0)
Collecting six>=1.5 (from python-dateutil>=2->pandas==0.21.0)

Installing collected packages: pytz, six, python-dateutil, numpy, pandas
Successfully installed numpy-1.14.5 pandas-0.21.0 python-dateutil-2.7.3 pytz-
2018.5 six-1.11.0
```

```
(venv) E:\Apress\test_proj>pip freeze | grep pandas
pandas==0.21.0
```

For other OS platforms, you might need to use the command `source venv/bin/activate` to activate the virtual environment. Once the virtual environment is active, you can see the `(venv)` notation as shown in the preceding code output and any new packages you install will be placed in the `venv` folder in complete isolation from the global system Python environment. You can see from the above code snippets how the `pandas` package has different versions in the same machine, 0.23.3 for global Python and 0.21.0 for the virtual environment Python. Hence these isolated virtual environments can run seamlessly on the same system. Once you have finished working in the virtual environment, you can deactivate it again as follows.

```
(venv) E:\Apress\test_proj>venv\Scripts\deactivate
E:\Apress\test_proj>pip freeze | grep pandas
pandas==0.23.3
```

This will bring you back to the system's default Python version with all its installed libraries and as expected the `pandas` version is the newer one which we had installed for the same. This gives us a good idea about the utility and advantages of virtual environments and once you start working on several projects, you should definitely consider using it. To know more about virtual environments, you can head here <http://docs.python-guide.org/en/latest/dev/virtualenvs/> which is the official documentation for the `virtualenv` package. This brings us to the end of our installation and setup activities and now we will be looking into some basic concepts around Python syntax and structure before diving into handling text data with python using hands on examples.

Python Syntax and Structure

We will discuss briefly about the basic syntax, structure and design philosophies which are followed when writing Python code for applications and systems. There is a defined hierarchical syntax for Python code which you should remember when writing code. Any big Python application or system is built using several modules which are themselves comprised of Python statements. Each statement is like a command or direction to the system directing what operations it should perform and these statements are comprised of expressions and objects. Everything in Python is an object including functions, data structures, types, classes and so on. This hierarchy can be visualized better in Figure 2-7.

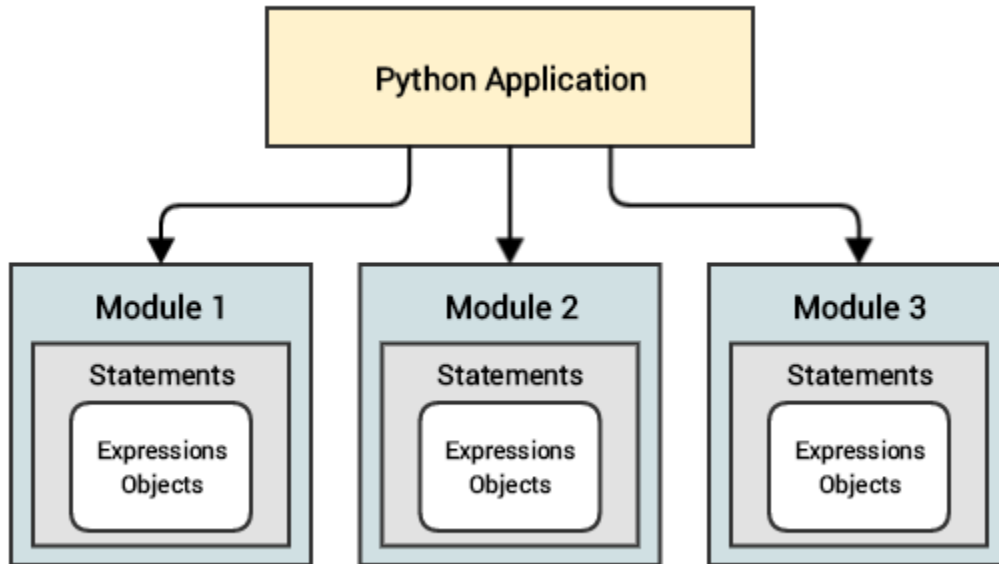


Figure 2-7. *Python program structure hierarchy*

The basic statements consist of objects, expressions which usually make use of objects and process and perform operations on them and objects can be anything from simple data types and structures to complex objects including functions and reserved words which have their own specific roles. Python has around 30+ keywords or reserved words which have their own designated role and function. We assume that you have some knowledge of basic programming constructs but in case you do not, don't despair! In the next section we will showcase how to work with text data using detailed hands-on examples.