Numerical\_Python\_2E\_Johansson\_C01

Numerical Python

Scientific Computing and Data

Science Applications with Numpy,

SciPy and Matplotlib

Second Edition

Robert Johansson

**Introduction**

Scientific and numerical computing is a booming field in research, engineering, and

analytics. The revolution in the computer industry over the last several decades has

provided new and powerful tools for computational practitioners. This has enabled

computational undertakings of previously unprecedented scale and complexity. Entire

fields and industries have sprung up as a result. This development is still ongoing, and

it is creating new opportunities as hardware, software, and algorithms keep improving.

Ultimately the enabling technology for this movement is the powerful computing

hardware that has been developed in recent decades. However, for a computational

practitioner, the software environment used for computational work is as important as, if

not more important than, the hardware on which the computations are carried out. This

book is about one popular and fast-growing environment for numerical computing: the

Python programming language and its vibrant ecosystem of libraries and extensions for

computational work.

Computing is an interdisciplinary activity that requires experience and expertise

in both theoretical and practical subjects: a firm understanding of mathematics and

scientific thinking is a fundamental requirement for effective computational work.

Equally important is solid training in computer programming and computer science.

The role of this book is to bridge these two subjects by introducing how scientific

computing can be done using the Python programming language and the computing

environment that has appeared around this language. In this book the reader is assumed

to have some previous training in mathematics and numerical methods and basic

knowledge about Python programming. The focus of the book is to give a practical

introduction to computational problem-solving with Python. Brief introductions to the

theory of the covered topics are given in each chapter, to introduce notation and remind

readers of the basic methods and algorithms. However, this book is not a self-consistent

treatment of numerical methods. To assist readers that are not previously familiar with

some of the topics of this book, references for further reading are given at the end of each

chapter. Likewise, readers without experience in Python programming will probably find

it useful to read this book together with a book that focuses on the Python programming

language itself.

**How This Book Is Organized**

The first chapter in this book introduces general principles for scientific computing

and the main development environments that are available for work with computing in

Python: the focus is on IPython and its interactive Python prompt, the excellent Jupyter

Notebook application, and the Spyder IDE.

In Chapter 2, an introduction to the NumPy library is given, and here we also

discuss more generally array-based computing and its virtues. In Chapter 3, we turn our

attention to symbolic computing – which in many respects complements array-based

computing – using the SymPy library. In Chapter 4, we cover plotting and visualization

using the Matplotlib library. Together, Chapters 2 to 4 provide the basic computational

tools that will be used for domain-specific problems throughout the rest of the book:

numerics, symbolics, and visualization.

In Chapter 5, the topic of study is equation solving, which we explore with both

numerical and symbolic methods, using the SciPy and SymPy libraries. In Chapter 6, we

explore optimization, which is a natural extension of equation solving. Here we mainly

work with the SciPy library and briefly with the cvxopt library. Chapter 7 deals with

interpolation, which is another basic mathematical method with many applications of

its own, and important roles in higher-level algorithms and methods. In Chapter 8, we

cover numerical and symbolic integration. Chapters 5 to 8 cover core computational

techniques that are pervasive in all types of computational work. Most of the methods

from these chapters are found in the SciPy library.

In Chapter 9, we proceed to cover ordinary differential equations. Chapter 10 is

a detour into sparse matrices and graph methods, which helps prepare the field for

the following chapter. In Chapter 11, we discuss partial differential equations, which

conceptually are closely related to ordinary differential equations, but require a different

set of techniques that necessitates the introduction of sparse matrices, the topic of

Chapter 10.

Starting with Chapter 12, we make a change of direction and begin exploring data

analysis and statistics. In Chapter 12, we introduce the Pandas library and its excellent

data analysis framework. In Chapter 13, we cover basic statistical analysis and methods

from the SciPy stats package. In Chapter 14, we move on to statistical modeling,

using the statsmodels library. In Chapter 15, the theme of statistics and data analysis

is continued with a discussion of machine learning, using the scikit-learn library. In

Chapter 16, we wrap up the statistics-related chapters with a discussion of Bayesian

statistics and the PyMC library. Together, Chapters 12 to 16 provide an introduction to

the broad field of statistics and data analytics: a field that has been developing rapidly

within and outside of the scientific Python community in recent years.

In Chapter 17, we briefly return to a core subject in scientific computing: signal

processing. In Chapter 18, we discuss data input and output, and several methods for

reading and writing numerical data to files, which is a basic topic that is required for

most types of computational work. In Chapter 19, the final regular chapter in this book,

two methods for speeding up Python code are introduced, using the Numba and Cython

libraries.

The Appendix covers the installation of the software used in this book. To install

the required software (mostly Python libraries), we use the conda package manager.

Conda can also be used to create virtual and isolated Python environments, which is an

important topic for creating stable and reproducible computational environments. The

Appendix also discusses how to work with such environments using the conda package

manager.

**CHAPTER 1**

**Introduction to Computing**

**with Python**

This book is about using Python for numerical computing. Python is a high-level,

general-purpose interpreted programming language that is widely used in scientific

computing and engineering. As a general-purpose language, Python was not specifically

designed for numerical computing, but many of its characteristics make it well suited

for this task. First and foremost, Python is well known for its clean and easy-to-read

code syntax. Good code readability improves maintainability, which in general results in

fewer bugs and better applications overall, but it also enables rapid code development.

This readability and expressiveness are essential in exploratory and interactive

computing, which requires fast turnaround for testing various ideas and models.

In computational problem-solving, it is, of course, important to consider the

performance of algorithms and their implementations. It is natural to strive for

efficient high-performance code, and optimal performance is indeed crucial for many

computational problems. In such cases it may be necessary to use a low-level program

language, such as C or Fortran, to obtain the best performance out of the hardware that

runs the code. However, it is not always the case that optimal runtime performance is the

most suitable objective. It is also important to consider the development

time required to implement a solution to a problem in a given programming language

or environment. While the best possible runtime performance can be achieved in a

low-level programming language, working in a high-level language such as Python usually

reduces the development time and often results in more flexible and extensible code.

These conflicting objectives present a trade-off between high performance and

long development time and lower performance but shorter development time. See

Figure 1-1 for a schematic visualization of this concept. When choosing a computational

environment for solving a particular problem, it is important to consider this trade-off

and to decide whether man-hours spent on the development or CPU-hours spent on

running the computations is more valuable. It is worth noting that CPU-hours are cheap

already and are getting even cheaper, but man-hours are expensive. In particular, your

own time is of course a very valuable resource. This makes a strong case for minimizing

development time rather than the runtime of a computation by using a high-level

programming language and environment such as Python and its scientific computing

libraries.

Figure 1-1. Trade-off between low- and high-level programming languages.

While a low-level language typically gives the best performance when a significant

amount of development time is invested in the implementation of a solution to a

problem, the development time required to obtain a first runnable code that solves

the problem is typically shorter in a high-level language such as Python.

A solution that partially avoids the trade-off between high- and low-level languages

is to use a multilanguage model, where a high-level language is used to interface

libraries and software packages written in low-level languages. In a high-level scientific

computing environment, this type of interoperability with software packages written in

low-level languages (e.g., Fortran, C, or C++) is an important requirement. Python excels

at this type of integration, and as a result, Python has become a popular “glue language”

used as an interface for setting up and controlling computations that use code written

in low-level programming languages for time-consuming number crunching. This is an

important reason for why Python is a popular language for numerical computing. The

multilanguage model enables rapid code development in a high-level language while

retaining most of the performance of low-level languages.

As a consequence of the multilanguage model, scientific and technical computing

with Python involves much more than just the Python language itself. In fact, the Python

language is only a piece of an entire ecosystem of software and solutions that provide a

complete environment for scientific and technical computing. This ecosystem includes

development tools and interactive programming environments, such as Spyder and

IPython, which are designed particularly with scientific computing in mind. It also

includes a vast collection of Python packages for scientific computing. This ecosystem

of scientifically oriented libraries ranges from generic core libraries – such as NumPy,

SciPy, and Matplotlib – to more specific libraries for particular problem domains.

Another crucial layer in the scientific Python stack exists below the various Python

modules: many scientific Python library interface, in one way or another; low-level

high-performance scientific software packages, such as for example optimized LAPACK

and BLAS libraries for low-level vector, matrix, and linear algebra routines; or other

specialized libraries for specific computational tasks. These libraries are typically

implemented in a compiled low-level language and can therefore be optimized and

efficient. Without the foundation that such libraries provide, scientific computing with

Python would not be practical. See Figure 1-2 for an overview of the various layers of the

software stack for computing with Python.

1

For example, MKL, the Math Kernel Library from Intel, https://software.intel.com/en-us/

intel-mkl; openBLAS, https://www.openblas.net; or ATLAS, the Automatically Tuned Linear

Algebra Software, available at http://math-atlas.sourceforge.net

**Tip** The SciPy organization and its web site www.scipy.org provide a

centralized resource for information about the core packages in the scientific

Python ecosystem, and lists of additional specialized packages, as well as

documentation and tutorials. As such, it is a valuable resource when working

with scientific and technical computing in Python. Another great resource is the

Numeric and Scientific page on the official Python Wiki: http://wiki.python.

org/moin/NumericAndScientific.

Apart from the technical reasons for why Python provides a good environment for

computational work, it is also significant that Python and its scientific computing libraries

are free and open source. This eliminates economic constraints on when and how

applications developed with the environment can be deployed and distributed by its users.

Equally significant, it makes it possible for a dedicated user to obtain complete insight on

how the language and the domain-specific packages are implemented and what methods

are used. For academic work where transparency and reproducibility are hallmarks, this

is increasingly recognized as an important requirement on software used in research. For

commercial use, it provides freedom on how the environment is used and integrated into

products and how such solutions are distributed to customers. All users benefit from the

relief of not having to pay license fees, which may otherwise inhibit deployments on large

computing environments, such as clusters and cloud computing platforms.

The social component of the scientific computing ecosystem for Python is another

important aspect of its success. Vibrant user communities have emerged around the core

packages and many of the domain-specific projects. Project-specific mailing lists, Stack

Overflow groups, and issue trackers (e.g., on Github, www.github.com) are typically very

active and provide forums for discussing problems and obtaining help, as well as a way of

getting involved in the development of these tools. The Python computing community also

organizes yearly conferences and meet-ups at many venues around the world, such as the

SciPy (http://conference.scipy.org) and PyData (http://pydata.org) conference series.

**Environments for Computing with Python**

There are a number of different environments that are suitable for working with

Python for scientific and technical computing. This diversity has both advantages

and disadvantages compared to a single endorsed environment that is common in

proprietary computing products: diversity provides flexibility and dynamism that lends

itself to specialization for particular use-cases, but on the other hand, it can also be

confusing and distracting for new users, and it can be more complicated to set up a

full productive environment. Here I give an orientation of common environments for

scientific computing, so that their benefits can be weighed against each other and an

informed decision can be reached regarding which one to use in different situations and

for different purposes. The three environments discussed here are

* The Python interpreter or the IPython console to run code interactively. Together with a text editor for writing code, this provides a lightweight development environment.
* The Jupyter Notebook, which is a web application in which Python code can be written and executed through a web browser. This environment is great for numerical computing, analysis, and problem-solving, because it allows one to collect the code, the output produced by the code, related technical documentation, and the analysis and interpretation, all in one document.
* The Spyder Integrated Development Environment, which can be used to write and interactively run Python code. An IDE such as Spyder is a great tool for developing libraries and reusable Python modules.

All of these environments have justified use-cases, and it is largely a matter of

personal preference which one to use. However, I do in particular recommend exploring

the Jupyter Notebook environment, because it is highly suitable for interactive and

exploratory computing and data analysis, where data, code, documentation, and results

are tightly connected. For development of Python modules and packages, I recommend

using the Spyder IDE, because of its integration with code analysis tools and the Python

debugger.

Python, and the rest of the software stack required for scientific computing with

Python, can be installed and configured in a large number of ways, and in general the

installation details also vary from system to system. In Appendix 1, we go through one

popular cross-platform method to install the tools and libraries that are required for

this book.

**Python**

The Python programming language and the standard implementation of the Python

interpreter are frequently updated and made available through new releases. Currently,

there are two active versions of Python available for production use: Python 2 and

Python 3. In this book we will work with Python 3, which by now has practically

superseded Python 2. However, for some legacy applications, using Python 2 may still be

the only option, if it contains libraries that have not been made compatible with Python

3. It is also sometimes the case that only Python 2 is the available in institutionally

provided environments, such as on high-performance clusters or universities’ computer

systems. When developing Python code for such environments, it might be necessary

to use Python 2, but otherwise, I strongly recommend using Python 3 in new projects. It

should also be noted that support for Python 2 has now been dropped by many major

2

The Python language and the default Python interpreter are managed and maintained by the

Python Software Foundation: http://www.python.org.

Chapter 1 Introduction to Computing with Python

Python libraries, and the vast majority of computing-oriented libraries for Python now

support Python 3. For the purpose of this book, we require version 2.7 or greater for the

Python 2 series or Python 3.2 or greater for the preferred Python 3 series.

**Interpreter**

The standard way to execute Python code is to run the program directly through the

Python interpreter. On most systems, the Python interpreter is invoked using the python

command. When a Python source file is passed as an argument to this command, the

Python code in the file is executed.

$ python hello.py

Hello from Python!

Here the file hello.py contains the single line:

print("Hello from Python!")

To see which version of Python is installed, one can invoke the python command

with the --version argument:

$ python --version

Python 3.6.5

It is common to have more than one version of Python installed on the same system.

Each version of Python maintains its own set of libraries and provides its own interpreter

command (so each Python environment can have different libraries installed). On many

systems, specific versions of the Python interpreter are available through the commands

such as, for example, python2.7 and python3.6. It is also possible to set up virtual

python environments that are independent of the system-provided environments. This

has many advantages and I strongly recommend to become familiar with this way of

working with Python. Appendix A provides details of how to set up and work with these

kinds of environments.

In addition to executing Python script files, a Python interpreter can also be used

as an interactive console (also known as a REPL: Read–Evaluate–Print–Loop). Entering

python at the command prompt (without any Python files as argument) launches the

Python interpreter in an interactive mode. When doing so, you are presented with a

prompt:

$ python

Python 3.6.1 |Continuum Analytics, Inc.| (default, May 11 2017, 13:04:09)

[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>>

From here Python code can be entered, and for each statement, the interpreter

evaluates the code and prints the result to the screen. The Python interpreter itself

already provides a very useful environment for interactively exploring Python code,

especially since the release of Python 3.4, which includes basic facilities such as a

command history and basic autocompletion (not available by default in Python 2).

**IPython Console**

Although the interactive command-line interface provided by the standard Python

interpreter has been greatly improved in recent versions of Python 3, it is still in certain

aspects rudimentary, and it does not by itself provide a satisfactory environment for

interactive computing. IPython is an enhanced command-line REPL environment for

Python, with additional features for interactive and exploratory computing. For example,

IPython provides improved command history browsing (also between sessions), an

input and output caching system, improved autocompletion, more verbose and helpful

exception tracebacks, and much more. In fact, IPython is now much more than an

enhanced Python command-line interface, which we will explore in more detail later

in this chapter and throughout the book. For instance, under the hood IPython is a

3

See the IPython project web page, http://ipython.org, for more information and its official

documentation.

client-server application, which separates the frontend (user interface) from the backend

(kernel) that executes the Python code. This allows multiple types of user interfaces

to communicate and work with the same kernel, and a user-interface application can

connect multiple kernels using IPython’s powerful framework for parallel computing.

Running the ipython command launches the IPython command prompt:

$ ipython

Python 3.6.1 |Continuum Analytics, Inc.| (default, May 11 2017, 13:04:09)

Type 'copyright', 'credits' or 'license' for more information

IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:

**Caution** Note that each IPython installation corresponds to a specific version

of Python, and if you have several versions of Python available on your system,

you may also have several versions of IPython as well. On many systems, IPython

for Python 2 is invoked with the command ipython2 and for Python 3 with

ipython3, although the exact setup varies from system to system. Note that here

the “2” and “3” refer to the Python version, which is different from the version of

IPython itself (which at the time of writing is 6.4.0).

In the following sections, I give a brief overview of some of the IPython features

that are most relevant to interactive computing. It is worth noting that IPython is used

in many different contexts in scientific computing with Python, for example, as a

kernel in the Jupyter Notebook application and in the Spyder IDE, which are covered

in more detail later in this chapter. It is time well spent to get familiar with the tricks

and techniques that IPython offers to improve your productivity when working with

interactive computing.

**Input and Output Caching**

In the IPython console, the input prompt is denoted as In [1]: and the corresponding

output is denoted as Out [1]:, where the numbers within the square brackets are

incremented for each new input and output. These inputs and outputs are called cells in

IPython. Both the input and the output of previous cells can later be accessed through

the In and Out variables that are automatically created by IPython. The In and Out

variables are a list and a dictionary, respectively, that can be indexed with a cell number.

For instance, consider the following IPython session:

In [1]: 3 \* 3

Out[1]: 9

In [2]: In[1]

Out[2]: '3 \* 3'

In [3]: Out[1]

Out[3]: 9

In [4]: In

Out[4]: [", '3 \* 3', 'In[1]', 'Out[1]', 'In']

In [5]: Out

Out[5]: {1: 9, 2: '3 \* 3', 3: 9, 4: [", '3 \* 3', 'In[1]', 'Out[1]', 'In', 'Out']}

Here, the first input was 3 \* 3 and the result was 9, which later is available as In[1]

and Out[1]. A single underscore \_ is a shorthand notation for referring to the most

recent output, and a double underscore \_\_ refers to the output that preceded the most

recent output. Input and output caching is often useful in interactive and exploratory

computing, since the result of a computation can be accessed even if it was not explicitly

assigned to a variable.

Note that when a cell is executed, the value of the last statement in an input cell

is by default displayed in the corresponding output cell, unless the statement is an

assignment or if the value is Python null value None. The output can be suppressed by

ending the statement with a semicolon:

In [6]: 1 + 2

Out[6]: 3

In [7]: 1 + 2; # output suppressed by the semicolon

In [8]: x = 1 # no output for assignments

In [9]: x = 2; x # these are two statements. The value of 'x' is shown in

the output

Out[9]: 2

**Autocompletion and Object Introspection**

In IPython, pressing the TAB key activates autocompletion, which displays a list of

symbols (variables, functions, classes, etc.) with names that are valid completions of

what has already been typed. The autocompletion in IPython is contextual, and it will

look for matching variables and functions in the current namespace or among the

attributes and methods of a class when invoked after the name of a class instance. For

example, os.<TAB> produces a list of the variables, functions, and classes in the os

module, and pressing TAB after having typed os.w results in a list of symbols in the os

module that starts with w:

In [10]: import os

In [11]: os.w<TAB>

os.wait os.wait3 os.wait4 os.waitpid os.walk os.write os.writev

This feature is called object introspection, and it is a powerful tool for interactively

exploring the properties of Python objects. Object introspection works on modules,

classes, and their attributes and methods and on functions and their arguments.

**Documentation**

Object introspection is convenient for exploring the API of a module and its member

classes and functions, and together with the documentation strings, or “docstrings”, that

are commonly provided in Python code, it provides a built-in dynamic reference manual

for almost any Python module that is installed and can be imported. A Python object

followed by a question mark displays the documentation string for the object. This is

similar to the Python function help. An object can also be followed by two question

marks, in which case IPython tries to display more detailed documentation, including

the Python source code if available. For example, to display help for the cos function in

the math library:

In [12]: import math

In [13]: math.cos?

Type: builtin\_function\_or\_method

String form: <built-in function cos>

Docstring:

cos(x)

Return the cosine of x (measured in radians).

Docstrings can be specified for Python modules, functions, classes, and their

attributes and methods. A well-documented module therefore includes a full API

documentation in the code itself. From a developer’s point of view, it is convenient to be

able to document a code together with the implementation. This encourages writing and

maintaining documentation, and Python modules tend to be well documented.

**Interaction with the System Shell**

IPython also provides extensions to the Python language that makes it convenient

to interact with the underlying system. Anything that follows an exclamation mark

is evaluated using the system shell (such as bash shell). For example, on a UNIX-like

system, such as Linux or Mac OS X, listing files in the current directory can be done using

In[14]: !ls

file1.py file2.py file3.py

On Microsoft Windows, the equivalent command would be !dir. This method for

interacting with the OS is a very powerful feature that makes it easy to navigate the file

system and to use the IPython console as a system shell. The output generated by a

command following an exclamation mark can easily be captured in a Python variable.

For example, a file listing produced by !ls can be stored in a Python list using

In[15]: files = !ls

In[16]: len(files)

3

In[17] : files

['file1.py', 'file2.py', 'file3.py']

Likewise, we can pass the values of Python variables to shell commands by prefixing

the variable name with a $ sign:

In[18]: file = "file1.py"

In[19]: !ls -l $file

-rw-r--r-- 1 rob staff 131 Oct 22 16:38 file1.py

This two-way communication with the IPython console and the system shell can be

very convenient when, for example, processing data files.

**IPython Extensions**

IPython provides extension commands that are called magic functions in IPython

terminology. These commands all start with one or two % signs. A single % sign is used

for one-line commands, and two % signs are used for commands that operate on cells

(multiple lines). For a complete list of available extension commands, type %lsmagic,

and the documentation for each command can be obtained by typing the magic

command followed by a question mark:

In[20]: %lsmagic?

Type: Magic function

String form: <bound method BasicMagics.lsmagic of <IPython.core.magics.

basic.BasicMagics object at 0x10e3d28d0>>

Namespace: IPython internal

File: /usr/local/lib/python3.6/site-packages/IPython/core/magics/

basic.py

Definition: %lsmagic(self, parameter\_s=")

Docstring: List currently available magic functions.

**File System Navigation**

In addition to the interaction with the system shell described in the previous section,

IPython provides commands for navigating and exploring the file system. The

commands will be familiar to UNIX shell users: %ls (list files), %pwd (return current

working directory), %cd (change working directory), %cp (copy file), %less (show the

content of a file in the pager), and %%writefile filename (write content of a cell to the

file filename). Note that autocomplete in IPython also works with the files in the current

4

When %automagic is activated (type %automagic at the IPython prompt to toggle this feature), the % sign that precedes the IPython commands can be omitted, unless there is a name conflict with a Python variable or function. However, for clarity, the % signs are explicitly shown here.

working directory, which makes IPython as convenient to explore the file system as is the

system shell. It is worth noting that these IPython commands are system independent

and can therefore be used on both UNIX-like operating systems and on Windows.

**Running Scripts from the IPython Console**

The command %run is an important and useful extension, perhaps one of the most

important features of the IPython console. With this command, an external Python

source code file can be executed within an interactive IPython session. Keeping a session

active between multiple runs of a script makes it possible to explore the variables and

functions defined in a script interactively after the execution of the script has finished.

To demonstrate this functionality, consider a script file fib.py that contains the

following code:

def fib(n):

"""

Return a list of the first n Fibonacci numbers.

"""

f0, f1 = 0, 1

f = [1] \* n

for i in range(1, n):

f[i] = f0 + f1

f0, f1 = f1, f[i]

return f

print(fib(10))

It defines a function that generates a sequence of n Fibonacci numbers and prints

the result for n = 10 to the standard output. It can be run from the system terminal using

the standard Python interpreter:

$ python fib.py

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

It can also be run from an interactive IPython session, which produces the

same output, but also adds the symbols defined in the file to the local namespace,

so that the fib function is available in the interactive session after the %run command

has been issued.

In [21]: %run fib.py

Out[22]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

In [23]: %who

fib

In [23]: fib(6)

Out[23]: [1, 1, 2, 3, 5, 8]

In the preceding example, we also made use of the %who command, which lists

all defined symbols (variables and functions). The %whos command is similar, but

also gives more detailed information about the type and value of each symbol, when

applicable.

**Debugger**

IPython includes a handy debugger mode, which can be invoked postmortem after

a Python exception (error) has been raised. After the traceback of an unintercepted

exception has been printed to the IPython console, it is possible to step directly into the

Python debugger using the IPython command %debug. This possibility can eliminate

the need to rerun the program from the beginning using the debugger or after having

employed the common debugging method of sprinkling print statements into the code.

If the exception was unexpected and happened late in a time-consuming computation,

this can be a big time-saver.

To see how the %debug command can be used, consider the following incorrect

invocation of the fib function defined earlier. It is incorrect because a float is passed to

the function while the function is implemented with the assumption that the argument

passed to it is an integer. On line 7 the code runs into a type error, and the Python

interpreter raises an exception of the type TypeError. IPython catches the exception and

5

The Python function dir provides a similar feature.

prints out a useful traceback of the call sequence on the console. If we are clueless as

to why the code on line 7 contains an error, it could be useful to enter the debugger by

typing %debug in the IPython console. We then get access to the local namespace at

the source of the exception, which can allow us to explore in more detail why the

exception was raised.

In [24]: fib(1.0)

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

TypeError Traceback (most recent call last)

<ipython-input-24-874ca58a3dfb> in <module>()

----> 1 fib.fib(1.0)

/Users/rob/code/fib.py in fib(n)

5 """

6 f0, f1 = 0, 1

----> 7 f = [1] \* n

8 for i in range(1, n):

9 f[n] = f0 + f1

TypeError: can't multiply sequence by non-int of type 'float'

In [25]: %debug

> /Users/rob/code/fib.py(7)fib()

6 f0, f1 = 0, 1

----> 7 f = [1] \* n

8 for i in range(1, n):

ipdb> print(n)

1.0

■■ **Tip** Type a question mark at the debugger prompt to show a help menu that

lists available command

ipdb> ?

More information about the Python debugger and its features is also available in

the Python Standard Library documentation: http://docs.python.org/3/

library/pdb.html.

**Reset**

Resetting the namespace of an IPython session is often useful to ensure that a program

is run in a pristine environment, uncluttered by existing variables and functions. The

%reset command provides this functionality (use the flag –f to force the reset). Using

this command can often eliminate the need for otherwise common exit-restart cycles of

the console. Although it is necessary to reimport modules after the %reset command has

been used, it is important to know that even if the modules have changed since the last

import, a new import after a %reset will not import the new module but rather reenable

a cached version of the module from the previous import. When developing Python

modules, this is usually not the desired behavior. In that case, a reimport of a previously

imported (and since updated) module can often be achieved by using the reload

function from IPython.lib.deepreload. However, this method does not always work, as

some libraries run code at import time that is only intended to run once. In this case, the

only option might be to terminate and restart the IPython interpreter.

**Timing and Profiling Code**

The %timeit and %time commands provide simple benchmarking facilities that are

useful when looking for bottlenecks and attempting to optimize code. The %timeit

command runs a Python statement a number of times and gives an estimate of the

runtime (use %%timeit to do the same for a multiline cell). The exact number of times

the statement is ran is determined heuristically, unless explicitly set using the –n and –r

flags. See %timeit? for details. The %timeit command does not return the resulting

value of the expression. If the result of the computation is required, the %time or %%time

(for a multiline cell) commands can be used instead, but %time and %%time only run the

statement once and therefore give a less accurate estimate of the average runtime.

The following example demonstrates a typical usage of the %timeit and %time

commands:

In [26]: %timeit fib(100)

100000 loops, best of 3: 16.9 μs per loop

In [27]: result = %time fib(100)

CPU times: user 33 μs, sys: 0 ns, total: 33 μs

Wall time: 48.2

While the %timeit and %time commands are useful for measuring the elapsed

runtime of a computation, they do not give any detailed information about what part

of the computation takes more time. Such analyses require a more sophisticated code

profiler, such as the one provided by Python standard library module cProfile. The

Python profiler is accessible in IPython through the commands %prun (for statements)

and %run with the flag –p (for running external script files). The output from the profiler

is rather verbose and can be customized using optional flags to the %prun and %run -p

commands (see %prun? for a detailed description of the available options).

As an example, consider a function that simulates N random walkers each taking M

steps and then calculates the furthest distance from the starting point achieved by any of

the random walkers:

In [28]: import numpy as np

In [29]: def random\_walker\_max\_distance(M, N):

...: """

...: Simulate N random walkers taking M steps, and return the

largest distance

...: from the starting point achieved by any of the random walkers.

...: """

...: trajectories = [np.random.randn(M).cumsum() for \_ in range(N)]

...: return np.max(np.abs(trajectories))

Calling this function using the profiler with %prun results in the following output,

which includes information about how many times each function was called and

a breakdown of the total and cumulative time spent in each function. From this

information we can conclude that in this simple example, the calls to the function

np.random.randn consume the bulk of the elapsed computation time.

In [30]: %prun random\_walker\_max\_distance(400, 10000)

20008 function calls in 0.254 seconds

Ordered by: internal time

6

Which can, for example, be used with the standard Python interpreter to profile scripts by

running python -m cProfile script.py

ncalls tottime percall cumtime percall filename:lineno(function)

10000 0.169 0.000 0.169 0.000 {method 'randn' of 'mtrand.

RandomState' objects}

10000 0.036 0.000 0.036 0.000 {method 'cumsum' of 'numpy.

ndarray' objects}

1 0.030 0.030 0.249 0.249 <ipython-input-30>:18(random\_

walker\_max\_distance)

1 0.012 0.012 0.217 0.217 <ipython-input-30>:

19(<listcomp>)

1 0.005 0.005 0.254 0.254 <string>:1(<module>)

1 0.002 0.002 0.002 0.002 {method 'reduce' of 'numpy.

ufunc' objects}

1 0.000 0.000 0.254 0.254 {built-in method exec}

1 0.000 0.000 0.002 0.002 \_methods.py:25(\_amax)

1 0.000 0.000 0.002 0.002 fromnumeric.py:2050(amax)

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_

lsprof.Profiler' objects}

**Interpreter and Text Editor as Development Environment**

In principle, the Python or the IPython interpreter and a good text editor are all that are

required for a full productive Python development environment. This simple setup is,

in fact, the preferred development environment for many experienced programmers.

However, in the following sections, we will look into the Jupyter Notebook and the

integrated development environment Spyder. These environments provide richer

features that improve productivity when working with interactive and exploratory

computing applications.

**Jupyter**

The Jupyter project7 is a spin-off from the IPython project that includes the Python

independent frontends – most notably the notebook application which we discuss in

more detail in the following section – and the communication framework that enables

7

For more information about Jupyter, see http://jupyter.org.

the separation of the frontend from the computational backends, known as kernels.

Prior to the creation of the Jupyter project, the notebook application and its underlying

framework were a part of the IPython project. However, because the notebook frontend

is language agnostic – it can also be used with a large number of other languages, such as

R and Julia – it was spun off a separate project to better cater to the wider computational

community and to avoid a perceived bias toward Python. Now, the remaining role of

IPython is to focus on Python-specific applications, such as the interactive Python

console, and to provide a Python kernel for the Jupyter environment.

In the Jupyter framework, the frontend talks to computational backends known as

kernels. The frontend can have multiple kernels registered, for example, for different

programming languages, for different versions of Python, or for different Python

environments. The kernel maintains the state of the interpreter and performs the actual

computations, while the frontend manages how code is entered and organized and how

the results of calculations are visualized to the user.

In this section, we will discuss the Jupyter QtConsole and Notebook frontends and

give a brief introduction to some of their rich display and interactivity features, as well

as the workflow organization that the notebook provides. The Jupyter Notebook is the

Python environment for computation work that I generally recommend in this book,

and the code listings in the rest of this book are understood to be read as if they are cells

in a notebook.

**The Jupyter QtConsole**

The Jupyter QtConsole is an enhanced console application that can serve as a substitute

to the standard IPython console. The QtConsole is launched by passing the qtconsole

argument to the jupyter command:

$ jupyter qtconsole

This opens up a new IPython application in a console that is capable of displaying

rich media objects such as images, figures, and mathematical equations. The Jupyter

QtConsole also provides a menu-based mechanism for displaying autocompletion

results, and it shows docstrings for functions in a pop-up window when typing the

opening parenthesis of a function or a method call. A screenshot of the Jupyter

Qtconsole is shown in Figure 1-3.

**The Jupyter Notebook**

In addition to the interactive console, Jupyter also provides the web-based notebook

application that has made it famous. The notebook offers many advantages over

a traditional development environment when working with data analysis and

computational problem-solving. In particular, the notebook environment allows to

write and to run code, to display the output produced by the code, and to document

and interpret the code and the results: all in one document. This means that the entire

analysis workflow is captured in one file, which can be saved, restored, and reused later

on. In contrast, when working with a text editor or an IDE, the code, the corresponding

data files and figures, and the documentation are spread out over multiple files in the file

system, and it takes a significant effort and discipline to keep such a workflow organized.

The Jupyter Notebook features a rich display system that can show media such as

equations, figures, and videos as embedded objects in the notebook. It is also possible

to create user interface (UI) elements with HTML and JavaScript, using Jupyter’s widget

system. These widgets can be used in interactive applications that connect the web

application with Python code that is executed in the IPython kernel (on the server side).

These and many other features of the Jupyter Notebook make it a great environment for

interactive and literate computing, as we will see examples of throughout this book.

To launch the Jupyter Notebook environment, the notebook argument is passed to

the jupyter command-line application.

$ jupyter notebook

This launches a notebook kernel and a web application that, by default, will serve

up a web server on port 8888 on localhost, which is accessed using the local address

http://localhost:8888/ in a web browser. By default, running jupyter notebook

will open a dashboard web page in the default web browser. The dashboard lists

all notebooks that are available in the directory from where the Jupyter Notebook

was launched, as well as a simple directory browser that can be used to navigate

subdirectories, relative to the location where the notebook server was launched, and to

open notebooks from therein. Figure 1-4 shows a screenshot of a web browser and the

Jupyter Notebook dashboard page.

8

This web application is by default only accessible locally from the system where the notebook

application was launched.

Clicking the “New” button creates a new notebook and opens it in a new page in the

browser (see Figure 1-5). A newly created notebook is named Untitled, or Untitled1,

etc., depending on the availability of unused filenames. A notebook can be renamed by

clicking the title field on the top of the notebook page. The Jupyter Notebook files are

stored in a JSON file format using the filename extension ipynb. A Jupyter Notebook file

is not pure Python code, but if necessary the Python code in a notebook can easily be

extracted using either “File ➤ Download as ➤ Python” or the Jupyter utility nbconvert

(see in the following section).

**Jupyter Lab**

Jupyter Lab is a new alternative development environment from the Jupyter project.

It combines the Jupyter Notebook interface with a file browser, text editor, shell, and

IPython consoles, in a web-based IDE-like environment; see Figure 1-6.

The Jupyter Lab environment consolidates the many advantages of the notebook

environment and the strengths of traditional IDEs. Having access to shell consoles and

text editors all within the same web frontend is also convenient when working on a

Jupyter server that runs on a remote system, such as a computing cluster or in the cloud.

**Cell Types**

The main content of a notebook, below the menu bar and the toolbar, is organized as

input and output cells. The cells can be of several types, and the type of the selected

cell can be changed using the cell-type drop-down menu in the toolbar (which initially

displays “Code”). The most important types are

* Code: A code cell can contain an arbitrary amount of multiline Python code. Pressing Shift-Enter sends the code in the cell to the kernel process, where the kernel evaluates it using the Python interpreter. The result is sent back to the browser and displayed in the corresponding output cell.
* Markdown: The content of a Markdown cell can contain marked- up plain text, which is interpreted using the Markdown language and HTML. A Markdown cell can also contain LaTeX formatted equations, which are rendered in the notebook using the JavaScript-based LaTeX engine MathJax.
* Headings: Heading cells can be used to structure a notebook into sections.
* Raw: A raw text cell is displayed without any processing.

**Editing Cells**

Using the menu bar and the toolbar, cells can be added, removed, moved up and down,

cut and pasted, and so on. These functions are also mapped to keyboard shortcuts,

which are convenient and time-saving when working with Jupyter Notebooks. The

notebook uses a two-mode input interface, with an edit mode and a command mode.

The edit mode can be entered by clicking a cell or by pressing the Enter key on the

keyboard when a cell is in focus. Once in edit mode, the content of the input cell can be

edited. Leaving the edit mode is done by pressing the ESC key or by using Shift-Enter to

execute the cell. When in command mode, the up and down arrows can be used to move

focus between cells, and a number of keyboard shortcuts are mapped to the basic cell

manipulation actions that are available through the toolbar and the menu bar. Table 1-1

summarizes the most important Jupyter Notebook keyboard shortcuts for the

command mode.

Table 1-1. A Summary of Keyboard Shortcuts in the Jupyter Notebook

Command Mode

Keyboard Shortcut Description

b Create a new cell below the currently selected cell.

a Create a new cell above the currently selected cell.

d-d Delete the currently selected cell.

1 to 6 Heading cell of level 1 to 6.

x Cut currently selected cell.

c Copy currently selected cell.

v Paste cell from the clipboard.

m Convert a cell to a Markdown cell.

y Convert a cell to a code cell.

Up Select previous cell.

Down Select next cell.

Enter Enter edit mode.

Escape Exit edit mode.

Shift-Enter Run the cell.

h Display a help window with a list of all available keyboard

shortcuts.

0-0 Restart the kernel.

i-i Interrupt an executing cell.

s Save the notebook.

While a notebook cell is being executed, the input prompt number is represented

with an asterisk, In[\*], and an indicator in the upper right corner of the page signals that

the IPython kernel is busy. The execution of a cell can be interrupted using the menu

option “Kernel ➤ Interrupt” or by typing i-i in the command mode (i.e., press the i key

twice in a row).

**Markdown Cells**

One of the key features of the Jupyter Notebook is that code cells and output cells

can be complemented with documentation contained in text cells. Text input cells

are called Markdown cells. The input text is interpreted and reformatted using the

Markdown markup language. The Markdown language is designed to be a lightweight

typesetting system that allows text with simple markup rules to be converted to HTML

and other formats for richer display. The markup rules are designed to be user-friendly

and readable as is in plain-text format. For example, a piece of text can be made italics

by surrounding it with asterisks, \*text\*, and it can be made bold by surrounding it

with double asterisks, \*\*text\*\*. Markdown also allows creating enumerated and

bulleted lists, tables, and hyper-references. An extension to Markdown supported by

Jupyter is that mathematical expressions can be typeset in LaTeX, using the JavaScript

LaTeX library MathJax. Taking full advantage of what Jupyter Notebooks offer includes

generously documenting the code and resulting output using Markdown cells and the

many rich display options they provide. Table 1-2 introduces basic Markdown and

equation formatting features that can be used in a Jupyter Notebook Markdown cell.

Table 1-2. Summary of Markdown Syntax for Jupyter Notebook Markdown Cells

Function Syntax by Example

Italics \*text\*

Bold \*\*text\*\*

Strike-through ~~text~~

Fixed-width font `text`

URL [URL text](http://www.example.com)

New paragraph Separate the text of two paragraphs with an empty line.

Verbatim Lines that start with four blank spaces are displayed as is, without

any further processing, using a fixed-width font. This is useful for

code-like text segments.

␣␣␣␣def func(x):

␣␣␣␣ return x \*\* 2

(continued)

Chapter 1 Introduction to Computing with Python

29

Function Syntax by Example

Table | A | B | C |

|---|---|---|

| 1 | 2 | 3 |

| 4 | 5 | 6 |

Horizontal line A line containing three dashes is rendered as a horizontal line

separator:

---

Heading # Level 1 heading

## Level 2 heading

### Level 3 heading

...

Block quote Lines that start with a “>” are rendered as a block quote.

> Text here is indented and offset

> from the main text body.

Unordered list \* Item one

\* Item two

\* Item three

Ordered list 1. Item one

2. Item two

3. Item three

Image ![Alternative text](image-file.png)9

or

![Alternative text](http://www.example.com/image.png)

Inline LaTeX equation $\LaTeX$

Displayed LaTeX equation

(centered and on a new line)

$$\LaTeX$$ or \begin{env}...\end{env} where env can be a

LaTeX environment such as equation, eqnarray, align, etc.

Table 1-2. (continued)

9

The path/filename is relative to the notebook directory.

Chapter 1 Introduction to Computing with Python

30

Markdown cells can also contain HTML code, and the Jupyter Notebook interface

will display it as rendered HTML. This is a very powerful feature for the Jupyter

Notebook, but its disadvantage is that such HTML code cannot be converted to other

formats, such as PDF, using the nbconvert tool (see later section in this chapter).

Therefore, it is in general better to use Markdown formatting when possible and resort to

HTML only when absolutely necessary.

More information about MathJax and Markdown is available at the projects web

pages at www.mathjax.com and http://daringfireball.net/projects/markdown,

respectively.

**Rich Output Display**

The result produced by the last statement in a notebook cell is normally displayed in the

corresponding output cell, just like in the standard Python interpreter or the IPython

console. The default output cell formatting is a string representation of the object,

generated, for example, by the \_\_repr\_\_ method. However, the notebook environment

enables a much richer output formatting, as it in principle allows displaying arbitrary

HTML in the output cell area. The IPython.display module provides several classes and

functions that make it easy to programmatically render formatted output in a notebook.

For example, the Image class provides a way to display images from the local file system

or online resources in a notebook, as shown in Figure 1-7. Other useful classes from

the same module are HTML, for rendering HTML code, and Math, for rendering LaTeX

expressions. The display function can be used to explicitly request an object to be

rendered and displayed in the output area.

Figure 1-7. An example of rich Jupyter Notebook output cell formatting, where an

image has been displayed in the cell output area using the Image class

An example of how HTML code can be rendered in the notebook using the HTML class

is shown in Figure 1-8. Here we first construct a string containing HTML code for a table

with version information for a list of Python libraries. This HTML code is then rendered

in the output cell area by creating an instance of the HTML class, and since this statement

is the last (and only) statement in the corresponding input cell, Jupyter will render the

representation of this object in the output cell area.

For an object to be displayed in an HTML formatted representation, all we need to

do is to add a method called \_repr\_hmtl\_ to the class definition. For example, we can

easily implement our own primitive version of the HTML class and use it to render the

same HTML code as in the previous example, as demonstrated in Figure 1-9.

Figure 1-9. Another example of how to render HTML code in the Jupyter

Notebook, using a class that implements the \_repr\_hmtl\_ method

Jupyter supports a large number of representations in addition to the \_repr\_hmtl\_

shown in the preceding text, for example, \_repr\_png\_, \_repr\_svg\_, and \_repr\_latex\_,

to mention a few. The former two can be used to generate and display graphics in the

notebook output cell, as used by, for example, the Matplotlib library (see the following

interactive example and Chapter 4). The Math class, which uses the \_repr\_latex\_

method, can be used to render mathematical formulas in the Jupyter Notebook. This is

often useful in scientific and technical applications. Examples of how formulas can be

rendered using the Math class and the \_repr\_latex\_ method are shown in Figure 1-10.

Figure 1-10. An example of how a LaTeX formula is rendered using the Math class

and how the \_repr\_latex\_ method can be used to generate a LaTeX formatted

representation of an object

Using the various representation methods recognized by Jupyter, or the convenience

classes in the IPython.display module, we have great flexibility in shaping how results

are visualized in the Jupyter Notebook. However, the possibilities do not stop there: an

exciting feature of the Jupyter Notebook is that interactive applications, with two-way

communication between the frontend and the backend kernel, can be created using, for

example, a library of widgets (UI components) or directly with Javascript and HTML. For

example, using the interact function from the ipywidgets library, we can very easily

create an interactive graph that takes an input parameter that is determined from a UI

slider, as shown in Figure 1-11.

In the example in Figure 1-11, we plot the distribution functions for the Normal

distribution and the Poisson distribution, where the mean and the variance of

the distributions are taken as an input from the UI object created by the interact

function. By moving the slider back and forth, we can see how the Normal and Poisson

distributions (with equal variance) approach each other as the distribution mean is

increased and how they behave very differently for small values of the mean. Interactive

graphs like this are a great tool for building intuition and for exploring computation

problems, and the Jupyter Notebook is a fantastic enabler for this kind of investigations.10

**nbconvert**

Jupyter Notebooks can be converted to a number of different read-only formats

using the nbconvert application, which is invoked by passing nbconvert as the first

argument to the jupyter command line. Supported formats include, among others,

PDF and HTML. Converting Jupyter Notebooks to PDF or HTML is useful when sharing

notebooks with colleagues or when publishing them online, when the reader does

not necessarily need to run the code, but primarily view the results contained in the

notebooks.

**HTML**

In the notebook web interface, the menu option “File ➤ Download as ➤ HTML” can

be used to generate an HTML document representing a static view of a notebook. An

HTML document can also be generated from the command prompt using the nbconvert

application. For example, a notebook called Notebook.ipynb can be converted to HTML

using the command:

$ jupyter nbconvert --to html Notebook.ipynb

This generates an HTML page that is self-contained in terms of style sheets and

JavaScript resources (which are loaded from public CDN servers), and it can be

published as is online. However, image resources that are using Markdown or HTML tags

are not included and must be distributed together with the resulting HTML file.

For public online publishing of Jupyter Notebooks, the Jupyter project provides a

convenient web service called nbviewer, available at http://nbviewer.jupyter.org.

By feeding it a URL to a public notebook file, the nbviewer application automatically

10For more information about how to create interactive applications using Jupyter and IPython

widgets, see the documentation for the ipywidgets library https://ipywidgets.readthedocs.

io/en/latest.

converts the notebook to HTML and displays the result. One of the many benefits of

this method of publishing Jupyter Notebooks is that the notebook author only needs to

maintain one file – the notebook file itself – and when it is updated and uploaded to its

online location, the static view of the notebook provided by nbviewer is automatically

updated as well. However, it requires publishing the source notebook at a publicly

accessible URL, so it can only be used for public sharing.

Tip The Jupyter project maintains a Wiki page that indexes many interesting

Jupyter Notebooks that are published online at http://github.com/jupyter/

jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks. These

notebooks demonstrate many of IPython’s and Jupyter’s more advanced features

and can be a great resource for learning more about Jupyter Notebooks as well as

the many topics covered by those notebooks.

**PDF**

Converting a Jupyter Notebook to PDF format requires first converting the notebook

to LaTeX and then compiling the LaTeX document to PDF format. To be able to do

the LaTeX to PDF conversion, a LaTeX environment must be available on the system

(see Appendix A for pointers on how to install these tools). The nbconvert application

can do both the notebook-to-LaTeX and the LaTeX-to-PDF conversions in one go,

using the --to pdf argument (the --to latex argument can be used to obtain the

intermediate LaTeX source):

$ jupyter nbconvert --to pdf Notebook.ipynb

The style of the resulting document can be specified using the --template name

argument, where built-in templates include base, article, and report (these templates

can be found in the nbconvert/templates/latex directory where Jupyter is installed).

By extending one of the existing templates,11 it is easy to customize the appearance of

11The IPython nbconvert application uses the jinja2 template engine. See http://jinja.pocoo.org

for more information and documentation of its syntax.

the generated document. For example, in LaTeX it is common to include additional

information about the document that is not available in Jupyter Notebooks, such as a

document title (if different from the notebook filename) and the author of the document.

This information can be added to a LaTeX document that is generated by the nbconvert

application by creating a custom template. For example, the following template extends

the built-in template article and overrides the title and author blocks:

((\*- extends 'article.tplx' -\*))

((\* block title \*)) \title{Document title} ((\* endblock title \*))

((\* block author \*)) \author{Author's Name} ((\* endblock author \*))

Assuming that this template is stored in a file called custom\_template.tplx,

the following command can be used to convert a notebook to PDF format using this

customized template:

$ jupyter nbconvert --to pdf --template custom\_template.tplx Notebook.ipynb

The result is LaTeX and PDF documents where the title and author fields are set as

requested in the template.

**Python**

A Jupyter Notebook in its JSON-based file format can be converted to a pure Python code

using the nbconvert application and the python format:

$ jupyter nbconvert --to python Notebook.ipynb

This generates the file Notebook.py, which only contains executable Python code (or

if IPython extensions were used in the notebook; a file that is executable with ipython).

The noncode content of the notebook is also included in the resulting Python code file in

the form of comments that do not prevent the file from being interpreted by the Python

interpreter. Converting a notebook to pure Python code is useful, for example, when

using the Jupyter Notebooks to develop functions and classes that need to be imported

in other Python files or notebooks.

**Spyder: An Integrated Development Environment**

An integrated development environment is an enhanced text editor that also provides

features such as integrated code execution, documentation, and debugging. Many

free and commercial IDE environments have good support for Python-based projects.

Spyder12 is an excellent free IDE that is particularly well suited for computing and data

analysis using Python. The rest of this section focuses on Spyder and explores its features

in more detail. However, there are also many other suitable IDEs. For example, Eclipse13

is a popular and powerful multilanguage IDE, and the PyDev14 extension to Eclipse

provides a good Python environment. PyCharm15 is another powerful Python IDE that

has gained a significant popularity among Python developers recently, and the Atom

IDE16 is yet another great option. For readers with previous experience with any of these

tools, they could be a productive and familiar environment also for computational work.

However, the Spyder IDE was specifically created for Python programming and in

particular for scientific computing with Python. As such it has features that are useful for

interactive and exploratory computing: most notably, integration with the IPython console

directly in the IDE. The Spyder user interface consists of several optional panes, which can

be arranged in different ways within the IDE application. The most important panes are

• Source code editor

• Consoles for the Python and the IPython interpreters and the system shell

• Object inspector, for showing documentation for Python objects

• Variable explorer

• File explorer

• Command history

• Profiler

12http://code.google.com/p/spyderlib

13http://www.eclipse.org

14http://pydev.org

15http://www.jetbrains.com/pycharm

16https://atom.io

Each pane can be configured to be shown or hidden, depending on the user’s

preferences and needs, using the “View ➤ Panes” menu option. Furthermore, panes

can be organized together in tabbed groups. In the default layout, three pane groups are

displayed: The left pane group contains the source code editor. The top-right pane group

contains the variable explorer, the file explorer, and the object inspector. The bottom

right pane group contains Python and IPython consoles.

Running the command spyder at the shell prompt launches the Spyder IDE. See

Figure 1-12 for a screenshot of the default layout of the Spyder application.

Figure 1-12. A screenshot of the Spyder IDE application. The code editor is shown

in the left panel, the top-right panel shows the object inspector (help viewer), and

the bottom right panel shows an IPython console.

**Source Code Editor**

The source code editor in Spyder supports code highlighting, intelligent autocompletion,

working with multiple open files simultaneously, parenthesis matching, indentation

guidance, and many other features that one would expect from a modern source code

editor. The added benefit from using an IDE is that code in the editor can be run – as a

whole (shortcut F5) or a selection (shortcut F9) – in attached Python or IPython consoles

with persistent sessions between successive runs.

In addition, the Spyder editor has very useful support for static code checking with

pylint,17 pyflakes,18 and pep8,19 which are external tools that analyze Python source code

and report errors such as undefined symbols, syntax errors, coding style violations, and

more. Such warnings and errors are shown on a line-by-line basis as a yellow triangle

with an exclamation mark in the left margin of the editor, next to the line number. Static

code checking is extremely important in Python programming. Since Python is an

interpreted and lazily evaluated language, simple bugs like undefined symbols may not

be discovered until the offending code line is reached at runtime, and for rarely used

code paths, sometimes such bugs can be very hard to discover. Real-time static code

checking and coding style checks in the Spyder editor can be activated and deactivated

in the “Editor” section of the preference windows (Python ➤ Preferences, in the menu

on OS X, and Tools ➤ Preferences on Linux and Windows). In the Editor section,

I recommend checking the “Code analysis” and “Style analysis” boxes in the “Code

Introspection/Analysis” tab.

**Tip** The Python language is versatile, and equivalent Python source code can

be written in a vast variety of styles and manners. However, a Python coding style

standard, PEP8, has been put forward to encourage a uniform appearance of

Python code. I strongly recommend studying the PEP8 coding style standard and

complying to it in your code. The PEP8 is described at www.python.org/dev/

peps/pep-0008.

17http://www.pylint.org

18http://github.com/pyflakes/pyflakes

19http://pep8.readthedocs.org

**Consoles in Spyder**

The integrated Python and IPython consoles can be used to execute a file that is being

edited in the text editor window, or for running interactively typed Python code. When

executing Python source code files from the editor, the namespace variables created

in the script are retained in the IPython or Python session in the console. This is an

important feature that makes Spyder an interactive-computing environment, in addition

to a traditional IDE application, since it allows exploring the values of variables after

a script has finished executing. Spyder supports having multiple Python and IPython

consoles opened simultaneously, and, for example, a new IPython console can be

launched through the “Consoles ➤ Open an IPython console” menu. When running a

script from the editor, by pressing F5 or pressing the run button in the toolbar, the script

is by default ran in the most recently activated console. This makes it possible to maintain

different consoles, with independent namespaces, for different scripts or projects.

When possible, use the %reset command and the reload function to clear a

namespace and reload updated modules. If that is insufficient, it is possible to restart the

IPython kernel corresponding to an IPython console, or the Python interpreter, via the

drop-down menu for the top-right icon in the console panel. Finally, a practical feature

is that IPython console sessions can be exported as an HTML file by right-clicking the

console window and selecting “Save as HTML/XML” in the pop-up menu.

**Object Inspector**

The object inspector (Help pane) is a great aid when writing Python code. It can display

richly formatted documentation strings for objects defined in source code created with

the editor and for symbols defined in library modules that are installed on the system. The

object text field at the top of the object inspector panel can be used to type the name of

a module, function, or class for which to display the documentation string. Modules and

symbols do not need to be imported into the local namespace to be able to display their

docstrings using the object inspector. The documentation for an object in the editor or the

console can also be opened in the object inspector by selecting the object with the cursor

and using the shortcut Ctrl-i (Cmd-i on OS X). It is even possible to automatically display

docstrings for callable objects when its opening left parenthesis is typed. This gives an

immediate reminder of the arguments and their order for the callable object, which can

be a great productivity booster. To activate this feature, navigate to the “Help” page in the

“Preferences” window and check the boxes in the “Automatic connections” section.

**Summary**

In this chapter we introduced the Python environment for scientific and technical

computing. This environment is, in fact, an entire ecosystem of libraries and tools

for computing, which includes not only Python software but everything from low-

level number crunching libraries up to graphical user interface applications and web

applications. In this multilanguage ecosystem, Python is the language that ties it all

together into a coherent and productive environment for computing. IPython is a core

component of Python’s computing environment, and we briefly surveyed some of its

most important features before covering the higher-level user environments provided

by the Jupyter Notebook and the Spyder IDE. These are the tools in which the majority

of exploratory and interactive computing is carried out. In the rest of this book, we focus

on computing using Python libraries, assuming that we are working within one of the

environments provided by IPython, the Jupyter Notebook, or Spyder.

**Further Reading**

The Jupyter Notebook is a particularly rich platform for interactive computing, and it is

also a very actively developed software. One of the most recent developments within the

Jupyter Notebook is its widget system, which are user-interface components that can be

used to create interactive interfaces within the browser that is displaying the notebook.

In this book we just briefly touch upon Jupyter widgets, but it is a very interesting and

rapidly developing part of the Jupyter project, and I do recommend exploring their

potential applications for interactive computing. The Jupyter Notebook widgets, and

many other parts of Jupyter, are documented through examples in Jupyter Notebook form

that are available here: http://nbviewer.ipython.org/github/ipython/ipython/tree/

master/examples. There are also two interesting books on this topic (Rossant, Learning

IPython for Interactive Computing and Data Visualization, 2013; Rossant, IPython

Interactive Computing and Visualization Cookbook, 2014) that I highly recommend.

**References**

1. Rossant, C. (2014). IPython Interactive Computing and Visualization Cookbook. Mumbai: Packt.
2. Rossant, C. (2013). Learning IPython for Interactive Computing and Data Visualization. Mumbai: Packt.