

Migrating MATLAB® to Python

Strategies, Comparisons and a Guide to Converting for Experts

```
pi
from traits.api import HasTraits, Instance,
from traitsui.api import View, Item, RangeEditor
from enable.api import ComponentEditor
from chaco.api import Plot, ArrayPlotData
7
8 # Local import
9 from damped_osc import DampedOsc
10
11
12 class DampedOscView(HasTraits):
13     """
14     Access the model traits using DelegatesTo().
15     """
16
17     model = Instance(DampedOsc)
18
19     x = DelegatesTo('model')
20     y = DelegatesTo('model')
21     a = DelegatesTo('model')
22     b = DelegatesTo('model')
23     omega = DelegatesTo('model')
24     phase = DelegatesTo('model')
25
26     plot = Instance(Plot)
27
28     traits_view = \
29         View(
30             Group(
31                 UItem('plot', editor=ComponentEditor),
32                 UItem('a', label='a', editor=RangeEditor),
33                 UItem('b', label='b', editor=RangeEditor),
34                 UItem('omega', label='omega', editor=RangeEditor),
35                 UItem('phase', label='phase', editor=RangeEditor)
36             )
37         )
```



Migrating MATLAB® to Python
Strategies, Comparisons and a Guide to Converting for Experts
Alexandre Chabot-Leclerc
Enthought, Inc.

©2020 Enthought, Inc.

Written by Enthought, Inc.

All Rights Reserved. Use only permitted under license. Copying, sharing, redistributing, or other unauthorized use strictly prohibited. All trademarks and registered trademarks are the property of their respective owners. MATLAB and Simulink are registered trademark of The MathWorks, Inc.

Enthought, Inc.
200 W Cesar Chavez St Suite 202
Austin, TX 78701
United States
www.enthought.com

Version 1.2.0



Introduction	1
Why Python	2
Differences Between Python and MATLAB®	4
Fundamental Data Types	4
Organizing Code in Packages, not Toolboxes	6
Syntax	6
Indexing and Slicing: Why Zero-Based Indexing	8
NumPy Arrays Are Not Matrices	10
Programming Paradigm: Object-Oriented vs. Procedural	13
Anti-Patterns	15
How Do I?	17
Load Data	17
Signal Processing	19
Linear Algebra	19
Machine Learning	20
Statistical Analysis	21
Image Processing and Computer Vision	21
Optimization	22
Natural Language Processing	22
Data Visualization	22
Save Data	25
What Else?	26
Strategies for Converting to Python	27
From the Bottom Up: Converting One Function at a Time	27
From the Top Down: Calling Python from MATLAB®	33
What Next?	37
Acknowledgments	37
Appendix	37
Code Example: Profiling Contiguous Array Operations	37
Complete Version of main.py, in Chapter 4	38
References	39

**Accelerate your Python migration with Enthought's
Python for Scientists and Engineers training course!**



Introduction

This document will guide you through the transition from MATLAB® to Python. The first section presents some reasons why you would want to do so.

The second section highlights some of the most important differences between the two languages: the fundamental data types; how code is organized in packages; an overview of the syntax differences; how indexing and slicing work; NumPy arrays; and how Python mainly uses an object-oriented programming paradigm.

The third section is structured around vignettes for common tasks when doing data analysis or running simulations. The vignettes highlight the most common packages used for each task, such as loading data, cleaning and reformatting data, performing analysis or simulation, plotting, and saving data.

The fourth section introduces two strategies to transition gradually to Python. Both rely on testing to validate that the new Python code works the same way as your MATLAB® code. They approach the problem by either converting all functions directly to Python or by calling Python from MATLAB®. You should use the strategy that is most convenient for your project.

Who This Guide Is For

Long-time MATLAB® users who want to migrate to Python, either partially or entirely.

Who This Document Is Not For

Those who rely heavily on the Simulink® graphical programming environment, as there is no good Simulink equivalent in the Python ecosystem.

Those who rely on MATLAB®'s automatic C and C++ code generation for embed systems, or its FPGA support, as there is no good tool for this purpose in the Python ecosystem.

Hardware and Software Requirements

Nothing in this guide is platform specific. All the code is written to run under Python 3. At the time of writing, the latest version is 3.8. The only version-specific feature used in this guide is the “@” operator for matrix multiplication. It was introduced in Python 3.5. Otherwise, the code should run under Python 2 and Python 3.



Conventions Used in This Document

Italic text is used for new terms, emphasis, and variables that do not appear in any code listing. **Constant width text** is used for program listings, as well as within paragraphs to refer to program variables, functions, data types, keywords, etc.

Why Python

There are a number of reasons why one might want to switch from MATLAB® to Python. Typically, they fall into four different categories: financial, freedom, technical, and social.

Financial:

Cost is often the first reason given for switching away from MATLAB®, as licensing fees add up quickly and may account for a significant part of a small organization's budget. Python certainly has the appeal of being free, because you do not have to pay a license fee and you have access to many free open source packages. However, be aware of challenges associated with transitioning from a language with which you are familiar, to one with which you are not. Once that transition is complete and the skillset has improved, Python will allow you to be more agile and productive in the long term.

Freedom:

Choosing Python, or any other open source language, lets you run your code without being locked-in with a given provider. There is no need to pay a license fee in order to keep your software running. More importantly, it means that colleagues, and others, can run Python code without needing a license. This can greatly improve the chances of survival for your project.

Technical:

Python has the benefit of being a *general purpose* programming language. Though it is an excellent language for scientific computing, it is not solely a scientific computing language. It can be used to do everything from building a file synchronization system (Dropbox), a photo-sharing service (Instagram), a 3D modeling and video-editing application (Blender), and a video hosting platform (YouTube), to discovering gravitational waves. Python was also used in most components of the Laser Interferometer Gravitational-Wave Observatory (LIGO) project.

The consequence of such varied uses is that you can find tools to do almost all common tasks. This allows you to use Python for your entire application, from hardware control and number crunching, to web API and desktop application. And for cases when a feature or a library exists only in another language, Python can easily interface with C/C++ and Fortran libraries. There are also Python implementations for some of the major other languages, such as IronPython for C, and Jython for Java.

Social:

The Python community is certainly a great reason to pick the language. There are the multiple PyCon conferences around the world, from the main conference in North America to PyCon Zimbabwe, PyCon Pakistan, and Kiwi PyCon. There are also the various SciPy conferences, which focus on the scientific Python ecosystem, or the PyData events about data science. Another aspect of having a vibrant community is the large number of



libraries available. As of August 18, 2017, there are 114,910 packages on the Python Package Index, the official repository for the Python language. This number does not include all the packages available on code hosting sites such as GitHub or Bitbucket. Finally, there are over 1.3 million questions tagged with “Python” on Stack Overflow—compared to 86,000 questions for MATLAB®—and countless articles, books, and blog posts about Python.



Differences Between Python and MATLAB®

Fortunately, Python and MATLAB® are not that different from each other, save for a few areas. Some are simple to assimilate, such as Python using zero-based indexing, or the meaning of the different types of brackets. Others are more challenging, such as how NumPy's row-major orientation impacts the way you think about your multidimensional data, or thinking in terms of objects and methods, rather than functions.

Fundamental Data Types

Python is designed as a *general purpose* language, not a numerical computing one like MATLAB®, therefore its basic types are also more general. Out of the box, there are no arrays or matrices. This is addressed by the NumPy package, which provides multidimensional arrays. With NumPy, numerical computing will be as fast and as concise as in MATLAB®.

Let's first focus on how the fundamental types in Python map to those in MATLAB®. They are numbers: real, float, and complex; strings; lists and tuples, which are two types of ordered sequences; dictionaries, which are “associative arrays” or mappings; and sets, which are unordered collections of unique items.

Numbers are scalars. They don't have a shape, they are zero dimensional. This is different from scalars in MATLAB®, which are 1-by-1 matrices.

```
int_number = 1
float_number = 1.0
complex_number = 1 + 2j
```

Strings can be written with either single or double quotes. They are immutable data structures, which means that they cannot be modified. Instead, new strings are created based on the contents of an existing one. They are “sequences” of characters, which means they can be indexed and sliced, both of which return a new string that is a subset of the original.



```
s1 = 'a string'
s2 = "also a string"
s3 = """A (possibly) multiline string"""
s3[0] returns 'A'
s2[:4] returns 'also'
```

Lists are similar to cell arrays in MATLAB®, except they are only one-dimensional. They can contain anything, including other lists. Even though they can contain different types of items at the same time, they tend to be homogeneous sequences of items, such as file names, words, database rows, tasks, etc. One item can be selected at a time based on its position, which is called *indexing*, or a subset can be selected, which is called *slicing*. Lists are mutable, which means items can be added, dropped, or replaced, and a list can grow or shrink in length (typically from the end of the list).

```
word_list = ['the', 'quick', 'brown', 'fox']
number_list = [0, 1, 1, 2, 3, 5, 8]
```

Tuples are similar to lists in that they are also ordered sequences, so they can be indexed and sliced, but they are immutable, like strings. They usually group together objects of different types, which are accessed via indexing. A good example would be representing a point in an x-y plane, such as **p1 = (0, 0)**. The first element represents the x position, and the second the y. They are accessed as **p1[0]** and **p1[1]**, respectively.

```
point = (0, 0)
also_a_point = 0, 0
a_3d_point = (0, 1, 2)
```

Dictionaries are similar to MATLAB® structures but allow for arbitrary keys as long as they are immutable—meaning numbers, strings, and tuples. That means you are not limited to strings. The values can be any objects.

```
meals = {'breakfast': 'sardines', 'lunch': 'salad', 'dinner': 'cake'}
```

Sets are not used very often but allow for standard “set operations,” such as intersection, union, or difference. They are also used for fast membership testing of the type “is value x in collection s.”

```
lights = {'red', 'yellow', 'green'}
choices = ['yes', 'no', 'yes', 'yes']
unique_choices = set(choices)
unique_choices is {'yes', 'no'}
```

Strings, lists, tuples, dictionaries, and sets are built-in types as are the numeric types (int, float, and complex). Other types can be imported from the standard library, or from third-party packages. The most important third-party type, when coming from MATLAB®, is probably the NumPy array, which is a set of homogeneous multidimensional arrays. Python programmers tend to use the fundamental data structures for most tasks that do not involve numerical computing.

To sum things up, if you are looking for matrices, use NumPy arrays; for structures, use dictionaries; and for cell arrays, use lists.

Organizing Code in Packages, not Toolboxes

In Python, collections of definitions (functions, classes) and statements, usually targeted toward solving a particular set of problems, are called packages. They are equivalent to toolboxes in MATLAB®. A single Python file is called a *module*, and a folder of Python files is a *package*. Python looks up modules and packages in a list of locations stored in the `sys.path` variable.

To find out what the path is for the current interpreter, execute the following two lines: **import sys**, and **print(sys.path)**.

This list is initialized with the directory containing the input script (or the current directory), the value of the PYTHONPATH environment variable, and installation-dependent default paths.

An important difference from MATLAB® is that Python developers tend to avoid modifying the path variables (both **PYTHONPATH** and **sys.path**), preferring instead to install packages in a standard location called *site-packages*. For cases where multiple versions of the same package would conflict with each other, the preferred solution is to create a stand-alone environment for each project, instead of modifying the paths in a project-specific manner. The environments are called *virtual environments*.

To learn more about environments, package management, and dependency resolution, refer to the Enthought Deployment Manager as well as the “Packages” section of the Python Packaging User Guide.

Syntax

Example 2.1 shows some key syntax similarities and differences between the two languages. It contains many common types and operations. In it, there are three sinusoids with different frequencies, saved to a PDF. It is shown below (Figure 2.1).

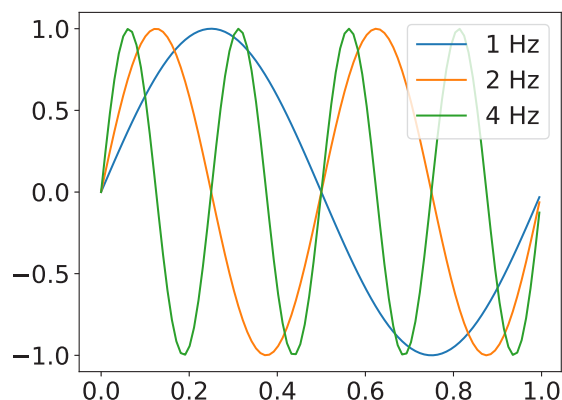


Figure 2.1: Figure generated by running the Python script in Example 2.1

Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 fs = [1, 2, 4]
5 all_time = np.linspace(0, 2, 200)
6 t = all_time[:100]
7
8 for f in fs:
9     y = np.sin(2 * np.pi * f * t)
10    plt.plot(t, y, label='{f} Hz'.format(f))
11
12 plt.legend()
13 plt.savefig('basics_python.pdf')
```

MATLAB®

```
1
2
3
4 fs = [1 2 4];
5 allTime = linspace(0, 2, 200);
6 t = allTime(1:100);
7 hold('on')
8 for f = fs
9     y = sin(2 * pi * f * t);
10    plot(t, y, 'DisplayName', sprintf('%d Hz', f));
11 end
12 legend('show');
13 saveas(gcf, 'basics_matlab.pdf');
```

Example 2.1: Plotting three sinusoids of different frequencies and saving the result as a PDF with Python and MATLAB®.

The semicolons (;) at the end of each line are no longer necessary. In Python, you have to explicitly *display* something, instead of explicitly silencing it. It is mainly done with the `print` function.

The first two lines in the Python example contain import statements. They define new names in the current namespace, **np** and **plt**, which can be referred to later. Both new names are aliases, the first to the NumPy package, and the second to the **pyplot** module of the matplotlib package. These imports are required to have access to the functions in those two packages, and the aliases are for convenience. The full name of a package or module does not need to be typed each time. This differs from MATLAB®, where any name is available as long as the file where the function is defined is on the “path.” Including those imports provides great benefits. For example, it avoids name conflicts that arise when two packages contain functions with the same name. Also, because imports usually appear at the very top of the file, you can easily see the requirements for the script.

Line 4 defines a *list* of frequencies in the Python example and a matrix in the MATLAB® example. Items always have to be separated by commas in Python. On line 5, there is an array of 200 points, between 0 and 2 inclusively. The array creation function in the Python example is part of NumPy, therefore we use the **np.** prefix to tell Python to look for the **linspace** function under the NumPy namespace. Generally speaking, the period means “look up the name on the right, under the namespace on the left.”

On line 6, only the first hundred points are selected. The **[:100]** syntax means “select a subset of the data, starting at the beginning (the lower bound is omitted), and up to position 100, non-inclusively.” The subset selection is called *slicing*. In Python, slicing (and indexing) is done using square brackets. Regular parentheses, on the other hand, are used in two ways. The first way is to group things together. For example, in a mathematical equation to force a certain order of operation, or to visually group the elements of a tuple. The second way is to *call* a function or a class. This is always the case when parentheses are “attached” to a name, such as `print('Hello world')`, or `np.arange(10)`. In the MATLAB® example, square brackets are only used to define a matrix, and the parentheses are used to both slice and call.

Line 7 turns on “hold” in the MATLAB® example, so that subsequent calls to plot all appear on the same figure. To “hold” is the default in Python’s matplotlib.

The for-loop on lines 8 to 11 iterates through the frequencies (line 8), calculates the y value (line 9), and plots y as a function of x , while at the same time labeling each line with the current frequency (line 10). The labels are later used in the legend. Notice that Python does not require the end keyword to delimit the end of the loop. The line indentation by four space characters is meaningful, and demarcates the body of the for-loop (and also of functions, classes, while-loops, etc.). In the Python example, on line 10, the plot function is part of the **pyplot** namespace. Still in the Python example, the optional label *keyword argument* is used to set the label of the line being plotted. Keyword arguments behave like mini-assignments, valid for the body of the function called. The keyword argument syntax can also be used to define default arguments when writing functions.

Finally, the legend is displayed on line 12, and the figure is saved as a PDF on line 13. In the Python example, matplotlib automatically saves the most recently active figure, the same way both matplotlib and MATLAB® know where a line should go when calling a plot. The section on plotting will cover matplotlib in more detail.

In summary, Python requires explicit imports for packages and modules; commas are necessary between items when declaring a list, array, tuple, etc.; and keyword arguments can be used to specify values for optional arguments.

Indexing and Slicing: Why Zero-Based Indexing

To select data, Python uses an index starting at zero, and defines intervals as closed on the left and open on the right, which means that the lower bound is included but the upper bound is not. Zero-based indexing was [popularized by Dijkstra](#), and here are two examples of the simplicity provided by this design decision.

In Python, we can slice a list into three parts, at (say) positions **low** and **high**, and add them back together with the following code. The `>>>` symbol is the Python prompt. IPython uses numbered prompts instead. They are functionally equivalent.

```
1 >>> a = [1, 2, 3, 4, 5, 6, 7]
2 >>> low, high = 2, 4
3 >>> a == a[0:low] + a[low:high] + a[high:7]
4 True
```

In fact, if the slice starts at zero then this would more often than not be written without the lower bound. If the slice goes until the end of the sequence, we would typically omit the upper bound:

```
1 >>> a == a[:low] + a[low:high] + a[high:]
```

Contrast this to MATLAB®, which requires two “+1”s:

```
1 >> a = [1, 2, 3, 4, 5, 6, 7];
2 >> low = 2;
3 >> high = 4;
4 >> all(a == [a(1:low), a(low+1:high), a(high+1:end)])
5
6 ans =
7
8 logical
9
10 1
```

In this scenario, most individuals would find themselves likely to forget the “+1”.

Here is another example: given a 2D image, **img**, stored in row-major order, we want to find the linear position in the array of the element at position (**x**, **y**). Using zero-based indexing, that linear position is **img[y * width + x]**, whereas with one-based indexing it is **img((y - 1) * width + x)**.

Here is another example where we repeatedly select **step** consecutive elements in a sequence of letters (see Example 2.2). On the MATLAB® side, we need a “-1” in the slice because of the inclusion of the upper bound.

Python

```
1 >>> letters = 'abcdef'
2 >>> step = 2
3 >>> for offset in range(0, len(letters), step):
4 ...     print(letters[offset:offset+step])
5 ...
6 ab
7 cd
8 ef
```

MATLAB®

```
1 >> letters = 'abcdef';
2 >> step = 2;
3 >> for offset = 1:step:length(letters)
4     fprintf('%s\n', letters(offset:offset+step-1));
5 end
6 ab
7 cd
8 ef
```

Example 2.2: The Python range function, in this case, expects (low, high, step).

Another indexing and slicing difference between Python and MATLAB® is in the way we refer to the last element of a sequence. Python uses negative indices. There is no special keyword. The last element of a sequence is index -1, -2. The one before the last element, -3, is the third element from the end, and so on. This syntax, combined with the fact that it is not necessary to specify the upper bound if the slice goes all the way to the end, results in a very compact notation to get the last *N* elements of a sequence:

```
1 >>> a = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
2 >>> a[-1]
3 'g'
4 >>> last_three = a[-3:]
5 >>> last_three
6 ['e', 'f', 'g']
```

NumPy Arrays Are Not Matrices

Most of the scientific Python stack is based on the homogeneous multidimensional arrays provided by the NumPy package. They are very similar to matrices in MATLAB®, but have a few important differences, which are highlighted here.

First, arrays can be one-dimensional. They do not have to be at least two-dimensional.

Each dimension is called an *axis*. The array type is `numpy.ndarray`, but they are usually displayed with the alias `array`. Arrays have useful attributes pertaining to their content. The main ones are:

`ndarray.ndim` The number of axes in the array (dimensions).

`ndarray.shape` The number of elements along each axis. It is always a tuple. The shape is equivalent to getting the size of a *matrix* A with `size(A)` in MATLAB®. For a 2D array with r rows and c columns, the shape is `(r, c)`. A one-dimensional array of length n has the shape `(n,)`.

`ndarray.size` The total number of elements in the array.

`ndarray.dtype` An object describing the type of the elements in the array. For example, it could be `int64` for 64-bit integers, `float32` for 32-bit floating point numbers, or `uint8` for unsigned 8-bit integers.

Operations between arrays are *always* element-wise unless otherwise stated using specific methods or notation. Therefore, there is no need for the `*` and `./` operators used in MATLAB®. Instead, performing a matrix product requires the `@` operator (available in Python 3 only) or explicitly calling the `dot` method, as shown in Example 2.3.

Data Ordering of Arrays: Row-Major vs Column-Major

NumPy is part of a group of languages and libraries defined as using a *row-major order*, with other notable members being C/C++ and Mathematica. The order affects how multidimensional arrays are stored in linear memory. In a row-major language, contiguous elements in a row are saved next to each other. MATLAB®, in contrast, uses *column-major order* where contiguous elements of a column are stored next to each other. Another way to express the difference is that the fastest varying index in a NumPy array is the last one, whereas it is the first one in MATLAB®. Given the two-dimensional array A :

$$A = \begin{pmatrix} 10 & 11 & 12 \\ 20 & 21 & 22 \end{pmatrix},$$

The indexing patterns required to select the contiguous elements in row-major (left) and column-major data structures (right) is shown in Table 2.1.

```
1 >>> import numpy as np
2 >>> a = np.arange(9).reshape(3, 3)
3 >>> a
4 array([[0, 1, 2],
5        [3, 4, 5],
6        [6, 7, 8]])
7 >>> e = np.eye(3)
8 >>> e
9 array([[ 1.,  0.,  0.],
10        [ 0.,  1.,  0.]])
```

```

11         [ 0., 0., 1.])
12 >>> a * e
13 array([[ 0., 0., 0.],
14        [ 0., 4., 0.],
15        [ 0., 0., 8.]])
16 >>> a @ e # or a.dot(e) in Python 2
17 array([[ 0., 1., 2.],
18        [ 3., 4., 5.],
19        [ 6., 7., 8.]])

```

Example 2.3: NumPy performs element-wise operations on arrays by default. To perform matrix multiplication, use the @ operator or the dot method.

Python (0-indexed)			MATLAB® (1-indexed)		
Address	Access	Value	Address	Access	Value
0	A[0,0]	10	0	A[1,1]	10
1	A[0,1]	11	1	A[2,1]	20
2	A[0,2]	12	2	A[1,2]	11
3	A[1,0]	20	3	A[2,2]	21
4	A[1,1]	21	4	A[1,3]	12
5	A[1,2]	22	5	A[2,3]	22

Table 2.1: Selecting contiguous elements of a row-major (left) and column-major array (right). The index base (zero or one) does not impact the order in which the elements are stored.

In both languages, an $R \times C$ array has R rows and C columns, even though Python stores R rows of C elements and MATLAB® stores C columns of R elements. The shape similarity breaks down when using arrays of three dimensions or more. A three-dimensional NumPy array has the shape (D, R, C) , where D is the “depth” of the cube. An equivalent MATLAB® matrix would have the shape (or size) $R \times C \times D$. Example 2.4 shows the creation of an array of 24 integers reshaped to have 3 rows, 4 columns, and a depth of 2.

Another way to think about the difference between row- and column-major order is that in row-major order (NumPy) new dimensions are prepended to the shape, as illustrated in Figure 2.2, whereas they are appended in column-major order (MATLAB®).

Python

```
1 >>> a = np.arange(24).reshape(2, 3, 4)
2 >>> a
3 array([[[ 0,  1,  2,  3],
4         [ 4,  5,  6,  7],
5         [ 8,  9, 10, 11]],
6        [[12, 13, 14, 15],
7         [16, 17, 18, 19],
8         [20, 21, 22, 23]]])
```

MATLAB®

```
1 >> a = reshape(0:23, 3, 4, 2)
2
3 a(:,:,1) =
4
5 0 3 6 9
6 1 4 7 10
7 2 5 8 11
8
9
10 a(:,:,2) =
11
12 12 15 18 21
13 13 16 19 22
14 14 17 20 23
```

Example 2.4: In Python, the depth of the 3D cube corresponds to the first dimension, whereas it is the last dimension in MATLAB®.

Here are some examples of how to store different types of multidimensional data using NumPy.

- Multichannel signals with C channels sampled at T points often have the shape **(C, T)**.
- The shape of images depends on the access patterns. The scikit-image library uses the following shapes, with **p_{ln}** being a “plane” and **ch** a “channel”, such as red, green and blue.
 - 2D grayscale images have the shape **(row, col)**;
 - 2D multichannel images (e.g., RGB) have the shape **(row, col, ch)**;
 - 3D grayscale images have the shape **(p_{ln}, row, col)**;
 - 3D multichannel images have the shape **(p_{ln}, row, col, ch)**;
 - Time-varying 3D images have the shape **(t, p_{ln}, row, col, ch)**.

The combination memory layout and access pattern on an array can have a significant impact on performance. Because of how modern processors pre-fetch data, it is worthwhile to operate on items that are next to each other in memory. If you have to loop over elements of an array, it is better to iterate along the innermost dimension. That would be the last one in Python and the first one in MATLAB®.

The example at the end of this report has some Python code to profile the difference between contiguous and non-contiguous processing. In some cases, the speed gain can be more than an order of magnitude. But remember, in NumPy, just as in MATLAB®, looping over elements of an array should be used only as a last resort. Vectorized operations are faster and more concise.

In summary, it is possible to do with NumPy arrays anything possible with MATLAB® matrices, even though it requires a small adjustment regarding data arrangements, and how dimensions are used.

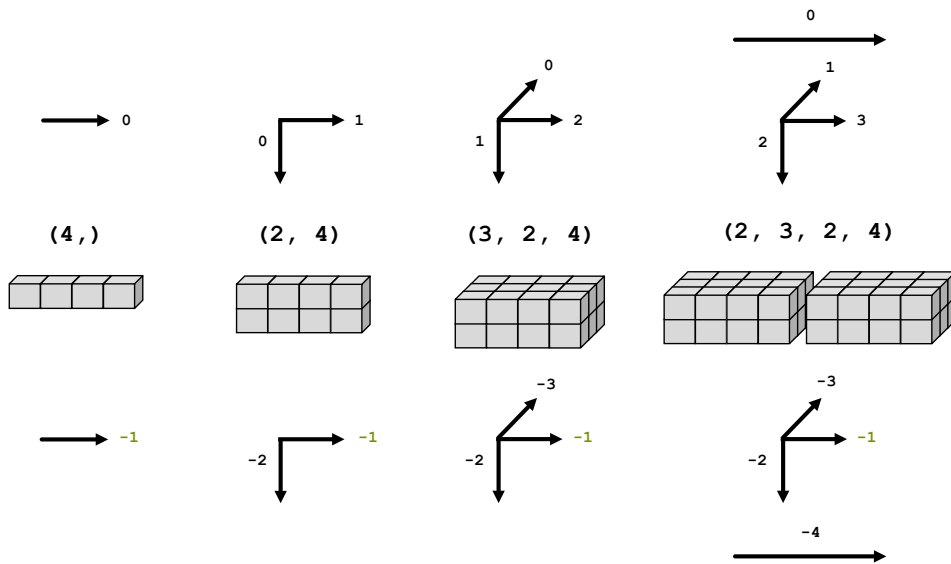


Figure 2.2: Visualization of one-, two-, three-, and four-dimensional NumPy arrays. The arrows on the top point in the direction of each axis, with zero always being the first element of the shape tuple. The arrows at the bottom illustrate how the last axis (-1) always corresponds to the columns.

Programming Paradigm: Object-Oriented vs. Procedural

Both Python and MATLAB® are “multi-paradigm” languages. A paradigm, in the context of programming languages, is a way to classify a language in terms of how code is executed and organized. For example, a language classified as *imperative* allows side effects, whereas a *functional* language does not. Both Python and MATLAB® are mainly imperative. On the code organization side, two important categories (but not the only ones) are *procedural*, where code is organized in functions (or *procedures*), and *object-oriented* (OO), where data and code are grouped together. Although both languages support both paradigms, they each lean toward a different side. Python tends to be more object-oriented, and MATLAB® tends to be more procedural.

In Python, everything is an object, from numbers and characters all the way to arrays, classes, and even functions. An object contains data, named attributes, and functions that “belong” to the object, called methods. Each object has a type and an identity. Multiple objects can have the same type, but each has its own identity. The type defines what the object is capable of. For example, numbers can be added to other numbers, strings can be formatted, arrays can be indexed and sliced, etc.

A common practice when using object-oriented programming is to have methods act on the data stored in that object, instead of passing the object to a function. This means methods sometimes do not take any argument because they “know” where to find the data. In contrast, a function usually requires data to be passed to it so it can act on it.

Example 2.5 compares the OO approach (on the top) and the procedural approach (on the bottom) when defining a NumPy array, reshaping it, and then calculating its overall and per-column maxima. In the OO example, the array is not passed to each method. In the procedural example, all the functions are under the NumPy namespace (`np.`) and the arrays `a` and `b` are passed to each call.

In the case of NumPy, no syntax is preferred to the other.

Object-Oriented

```
1 >>> a = np.arange(6)
2 >>> b = a.reshape(2, 3)
3 >>> b
4 array([[0, 1, 2],
5        [3, 4, 5]])
6 >>> b.max()
7 5
8 >>> b.max(axis=0)
9 array([3, 4, 5])
```

Procedural

```
1 >>> a = np.arange(6)
2 >>> b = np.reshape(a, (2, 3))
3 >>> b
4 array([[0, 1, 2],
5        [3, 4, 5]])
6 >>> np.max(b)
7 5
8 >>> np.max(b, axis=0)
9 array([3, 4, 5])
```

Example 2.5: Comparison of the object-oriented and procedural interfaces in NumPy.

Another common pattern allowed by the OO approach is to chain methods rather than nesting functions. Example 2.6 shows two comparisons between method chaining and function nesting. In the first, trailing whitespace in a string is removed. It is then converted to uppercase, and the spaces are replaced with underscores. In the second, an array is defined and reshaped, then the maximum of each column is found. The top shows how it is done in Python with chained method calls, and the bottom shows the MATLAB® version using nested functions.

Python

```
1 >>> sentence = "the quick brown fox "
2 >>> sentence.strip().upper().replace(' ', '_')
3 'THE_QUICK_BROWN_FOX'
4
5
6
7
8 >>> # Now with an array
9 >>> a = np.arange(12)
10 >>> a.reshape(3, 4).max(axis=0)
11 array([ 8, 9, 10, 11])
12
```

MATLAB®

```
1 >> sentence = "the quick brown fox ";
2 >> replace(upper(strip(sentence)), " ", "_")
3
4 ans =
5
6 "THE_QUICK_BROWN_FOX"
7
8 >> % Now with an array
9 >> a = 0:11;
10 >> max(reshape(a, 3, 4), [], 1)
11
12 ans =
13
14      2  5  8 11
```

Example 2.6: Transforming a string and an array using chained methods in Python (top) and nested functions in MATLAB® (bottom). The maxima are not the same on both sides because NumPy reshapes the array one row at a time and MATLAB® one column at a time.

In IPython, method discovery for a given type can be done interactively using “tab completion.” Press the “tab” key after entering the variable named followed by a period, as seen in Example 2.7.

```
1 In [1]: words = ['Hello', 'World!']
2 In [2]: words.<TAB>
3 words.append words.count words.insert words.reverse
4 words.clear words.extend words.pop words.sort
5 words.copy words.index words.remove
```

Example 2.7: Using tab-completion to discover methods available on an object. <TAB> represents the “tab” key on the keyboard. Lines 3-5 are the methods available on lists.

It is entirely possible to use Python in a procedural way, using only the built-in types. Yet, it is easy to define your own types if need be. Python’s extensive data model also makes it simple to make your own types behave like native ones. You will probably find that using an OO approach makes large projects easier to manage.

Anti-Patterns

What we refer to as “anti-patterns” are a handful of habits that many MATLAB® developers have and take with them when writing Python code. Knowing about them can help quickly ensure that your code looks more pythonic.

1. Do not use the **pylab** mode of IPython, nor run **from matplotlib.pylab import ***, nor use any other form of **from package import ***, where the star means “everything”. **pylab** is a deprecated “feature” of IPython that would import into the current namespace all functions from NumPy, as well as many from matplotlib, and would activate *interactive* mode. It seems convenient, and it replicates the MATLAB® experience, but this practice can actually cause very subtle bugs. To activate the interactive mode, instead use either the **%matplotlib** IPython magic command or start IPython with the **--matplotlib** option.
2. Do not iterate over objects using an index, such as in this code snippet:

```
1 words = ['quick', 'brown', 'fox']
2 for i_word in range(len(words)):
3     print(words[i_word])
```

Python has a rich *iterator* protocol that allows you to iterate over the elements of any sequence directly. The following is much more pythonic, as well as readable:

```
1 words = ['quick', 'brown', 'fox']
2 for word in words:
3     print(word)
```

3. Do not “clear all” or “close all” within your scripts. Scripts run in their own namespace when you run them in an IPython session.
4. Do not prefix function names. Instead of calling related functions **moleculecompute**, **moleculelistelements**, and **moleculeview**, put them in a **molecule** module and name them **computemass**, **listelements** and **view**. In fact, in this case, defining a **Molecule** class with those three methods would probably be a better idea.

5. Do not squeeze all code together, and do not mix code styles. Instead, use white space and follow a style guide. Example 2.8 does not use any white space and mixes *camelCase* with *alllowercase* naming. Example 2.9 makes ample use of white space, including splitting arguments on multiple lines for better readability. It follows the Python style guide, which is defined in the Python Enhancement Proposal 8, more commonly referred to as PEP 8. Most of the Python community follows this style guide. There are even tools, such as `autopep8` and `Black`, that can automatically format code to conform to PEP 8.

```
1 import numpy as np
2 from scipy.signal import correlate
3 cleanSignal=np.repeat([0.,1.,0.,0.,1.,1.,0.,1.],128)
4 noisysignal=cleanSignal+np.random.randn(cleanSignal.shape)
5 corr=correlate(noisysignal,np.ones(128),mode='same',method='fft')/128
```

Example 2.8: Busy example without any white space.

```
1 import numpy as np
2 from scipy.signal import correlate
3
4 clean_signal = np.repeat([0., 1., 0., 0., 1., 1., 0., 1.], 128)
5 noisy_signal = clean_signal + np.random.randn(clean_signal.shape)
6 corr = correlate(noisy_signal,
7                 np.ones(128),
8                 mode='same',
9                 method='fft') / 128
```

Example 2.9: Example 2.8 reformatted to follow the PEP 8 style guide for better readability.

How Do I?

This section is loosely structured around a common workflow when working with data, which is to load data, clean and process it, do some modeling or analysis, generate some kind of report with text and figures, and save the results. This section introduces the common packages used to perform each of the tasks, shows a small example, and makes some recommendations about things to keep in mind. Let's start from the beginning, with loading data.

Load Data

Even though Python's standard library has many modules to read and write various data types, it is recommended that you use third-party packages to do so. They are usually more feature complete, and are better integrated with the scientific Python ecosystem. Table 3.1 presents a few common data types, together with the recommended packages. The italicized packages are part of the standard library.

Data type	Packages
Audio	<code>scipy.io.wavfile.read</code>
Binary data	<code>numpy.fromfile</code>
CSV	Pandas, <i>csv</i>
Excel	Pandas
HDF5	<i>h5py</i> , <i>pytables</i>
HTML	Beautiful Soup, Pandas (if data in HTML tables)
Images	<code>scipy.ndimage</code> , Scikit-Image, Pillow
JSON	<i>json</i> , <i>simplejson</i>
MATLAB MAT	<code>scipy.io.loadmat/savemat</code>
NetCDF4	<code>netcdf4-python</code>
Tabular data	Pandas
Web APIs	Requests

Table 3.1: Common data types and the recommended packages to read and write them.
Modules that are part of the standard library are italicized.

The SciPy package has built-in support to read and write MAT files with the functions `loadmat` and `savemat`, respectively. They are located in the `scipy.io` module. The `loadmat` function loads the content of the MAT file as a dictionary. It takes care of the type conversion between the two languages. For example, MATLAB® matrices are loaded as NumPy arrays. In Example 3.1, three variables are created in a MATLAB® script and saved to the MAT file named `my_data.mat`.

```
1 clear;
2 my_scalar = 1;
3 my_1d_array = [1.2 3.4 5.6];
4 my_2d_array = [1 3 5;
5               2 4 6];
6 save('my_data.mat')
```

Example 3.1: Create some data in MATLAB®, which is loaded in Python in Examples 3.2 and 3.3.

Then in Example 3.2, the data is read using the `loadmat` function. The three matrices are loaded with the same number of dimensions they had in MATLAB®. They are all two-dimensional.

```
1 >>> from scipy.io import loadmat
2 >>> data = loadmat('my_data.mat')
3 >>> data['my_scalar']
4 array([[1]], dtype=uint8)
5 >>> data['my_1d_array']
6 array([[1.2, 3.4, 5.6]])
7 >>> data['my_2d_array']
8 array([[1, 3, 5],
9        [2, 4, 6]], dtype=uint8)
```

Example 3.2: Load data from a MAT file in Python using the `scipy.io.loadmat` function. MATLAB® matrices are loaded as NumPy arrays with at least two dimensions. The `dtype` of `my_1d_array` is not shown because it has the default value of `float64`, which is the same as `double`.

In Example 3.3, the data is loaded with the `squeeze_me` option set to `True`, which means that all unit dimensions are “squeezed out.” The results are a *floating point* number, a 1D array, and a 2D array. This option should be used with care since it changes the number of dimensions, but it is useful when converting code to Python because it yields arrays similar in shape to what NumPy would create.

```
1 >>> data = loadmat('my_data.mat', squeeze_me=True)
2 >>> data['my_scalar']
3 1
4 >>> data['my_1d_array']
5 array([1.2, 3.4, 5.6])
6 >>> data['my_2d_array']
7 array([[1, 3, 5],
8        [2, 4, 6]], dtype=uint8)
```

Example 3.3: Load data from a MAT file in Python using the `loadmat` function with the `squeeze_me` argument set to `True`. It has the effect of “squeezing” out unit matrix dimensions. It is equivalent to calling the function `np.squeeze` on all the inputs.

The Pandas package implements the *DataFrame* data structure, which is similar to the *table* type in MATLAB®. It is a tabular data structure with labeled rows and columns, which happens to fit a wide range of real-world data. DataFrames allow for database-style queries and operations, such as join and merge, as well as elegant data selection and subsetting because of the labeled dimensions. Pandas is a powerful tool for data analysis.

All of the functions to read data in Pandas start with the prefix **read_**. There are currently 18 different functions (Pandas 0.25). To read data from text files, the main function is **read_csv**. Given a URL, the **read_html** function downloads the page's contents, identifies and parses all tables, and returns a list of DataFrames. There is no need to write a custom HTML parser. The **read_sql** function can execute an SQL query on the most common database types, and load the result as a DataFrame. Finally, **read_excel** can read Excel XLS and XLSX files (even if they contain functions), and return either a single sheet as a DataFrame or the complete workbook as a dictionary of DataFrames.

Signal Processing

SciPy contains most of the functionality required for signal processing. The **scipy.signal** sub-package contains functions for convolution, filter design and filtering, window functions, peak finding, spectral analysis, and more. The **scipy.fftpack** provides bindings to the FFTPACK Fortran library for fast Fourier transforms (FFTs). If your work involves a large number of FFTs, you will benefit greatly from using the ones provided by NumPy, in the **numpy.fft** module. The NumPy version provided in the Enthought Python distribution is linked against the Intel® Math Kernel Library (MKL), which can be orders of magnitude faster than FFTPACK for non-power-of-two signal lengths.

Linear Algebra

NumPy does provide a **numpy.matrix** type, which is a specialized 2D array, but you should not use it. Arrays are the default type, and support all matrix operations with methods and functions as well as the **@** operator for multiplication.

Both NumPy and SciPy provide functionality for linear algebra, in **numpy.linalg** and **scipy.linalg**, respectively. The SciPy implementation contains a superset of NumPy's functions. It is also often faster because it is built against optimized libraries such as Intel® MKL, BLAS, and LAPACK. Therefore, it is recommended to use the function for linear algebra from SciPy if you have it installed.

The Python syntax does not support the two “solve” functions implemented with the **/** and **** operators in MATLAB®. In order to solve the equation $ax=b$ for the unknown x , use **linalg.solve(a, b)** (if **a** is square) or **linalg.lstsquare(a, b)**, instead of $a \setminus b$ in MATLAB®.

The NumPy for MATLAB® Users section of the online documentation has a useful section on linear algebra equivalents. Many of the function names are the same, except that they live under the **scipy.linalg** namespace.

Machine Learning

Using Python gives you access to the scikit-learn machine learning package, which provides fast, efficient implementations of the most common machine learning algorithms for data mining and data analysis. It can perform classification, regression, and clustering, as well as dimensionality reduction, model selection, and various kinds of preprocessing. Scikit-learn depends only on NumPy, SciPy, and matplotlib.

The best feature of scikit-learn may well be its very elegant application programming interface (API), which has inspired a lot of other packages and libraries. All the *estimators* (models) follow a similar interface. First, you *fit* the estimator to training data, and then you *predict* the class of new data. Each estimator has the methods **fit(X, y)** and **predict(T)**, with *X* being the training data, *y* the class corresponding to each training sample, and *T* the testing data. The training data *X* must be a 2D NumPy array of the shape **(n_samples, n_features)**, and the training targets *y* should have the shape **(n_samples,)**. The test data can have a different number of samples but must have the same number of features as the training data.

Example 3.4 shows how to predict handwritten digits loaded from the **sklearn.datasets** submodule (lines 1-6). The Support Vector Classifier estimator is instantiated on line 9, fitted on line 10, and used for prediction on Line 11. Figure 3.1 shows the first of the predicted digits (line 18), which is correctly predicted.

Trying a different estimator would only require importing a different module on line 8, and using a different estimator on line 10.

```
1 >>> from sklearn import datasets
2 >>> digits = datasets.load_digits()
3 >>> digits.data.shape
4 (1797, 64)
5 >>> digits.target.shape
6 (1797,)
7
8 >>> from sklearn import svm
9 >>> est = svm.SVC(gamma=0.0001, C=100.)
10 >>> est.fit(digits.data[:-5], digits.target[:-5])
11 >>> est.predict(digits.data[-5:])
12 array([9, 0, 8, 9, 8])
13 >>> digits.target[-5:]
14 array([9, 0, 8, 9, 8])
```

Example 3.4 Recognizing handwritten digits with scikit-learn.

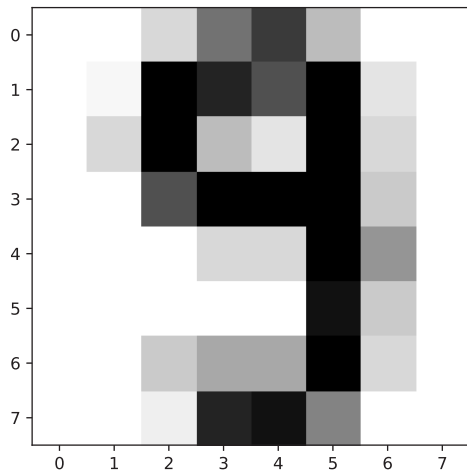


Figure 3.1: First digit of the test sequence.

Statistical Analysis

The main packages for doing statistical analysis are:

- The **scipy.stats** submodule of SciPy, which provides a large number of probability distributions as well as many statistical functions.
- statsmodels, which provides functionality for the estimation of different statistical models, such as linear regression models, discrete choice models, generalized linear models, and state space models, as well as for nonparametric statistics, time series analysis, and ANOVA. It plays well with the Pandas package and can use R-style formulas.
- PyMC3 for Bayesian modeling and probabilistic machine learning with a focus on Markov chain Monte Carlo and variational fitting algorithms.

Image Processing and Computer Vision

For image processing and computer vision, the most commonly used packages are:

- The **scipy.ndimage** submodule of SciPy which has native functions for multidimensional image processing, including filtering, interpolation, measurements, and morphology analysis.
- scikit-image, which extends the **scipy.ndimage** submodule, with more filtering capabilities, color-space transformations, restoration, segmentation, various transformations, and more.
- OpenCV, the computer vision and machine learning library, which has native Python bindings and requires only the opencv package.

Optimization

The main packages for solving optimization problems are:

- **scipy.optimize** provides functions for local and global optimization, root finding, curve fitting, and linear programming.
- For large and complex optimization problems, check out the mystic framework for “highly-constrained non-convex optimization and uncertainty quantification.”

Natural Language Processing

Python has a vibrant ecosystem of packages for doing natural language processing (NLP). Here are the main ones:

- NLTK is the most full-featured package for NLP. It supports many languages and comes with multiple corpora and lexical resources. It is a great tool for teaching and research.
- spaCy is designed to “get things done.” It implements a subset of NLTK’s features but is more readily usable. It is also faster.
- gensim is a package for topic modeling. It is designed for streaming processing of large datasets.
- scikit-learn also offers some text-processing functionality, mainly for bag-of-words text classification.

Data Visualization

The main package for data visualization is called matplotlib. It was designed to emulate MATLAB®’s plotting interface. Therefore, function names – and most concepts – should feel very familiar. The MATLAB®-like interface is part of a submodule called pyplot, which is usually imported as **plt** using **import matplotlib.pyplot as plt**. Under the **plt** namespace, you will find all the plotting functions that exist in MATLAB®, such as **plot**, **imshow**, **figure**, **xlabel**, **title**, and many others. The pyplot interface behaves like plotting in MATLAB®. It keeps track of the current figure and plotting area, so that subsequent function calls affecting the plots are directed to the current axes. In matplotlib and MATLAB®, an axes is another name for a subplot. It is the region of the image with the data representation.

In addition to the “state-machine,” which keeps track of the most recently active axes, matplotlib offers an object-oriented interface. It is similar to manipulating figure and axes handles in MATLAB®. The pyplot module is still used to create the figure, and possibly the axes as well, but plotting and annotating is done directly on the figure and axes objects. Example 3.5 reuses the content of Example 2.1, but this time compares the object-oriented approach of using matplotlib on top of the state-machine approach on the right. The important change is the call to the **plt.subplots** function on line 8, which is used to create the figure object, **fig**, and one axes object, **ax**. Plotting is done

directly on the `ax` object on line 11. Showing the legend is also done via the axes, on line 13. Saving the figure is a method on the figure itself, not the axes, because a figure can have multiple axes (or subplots).

Object-oriented approach

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 fs = [1, 2, 4]
5 all_time = np.linspace(0, 2, 200)
6 t = all_time[:100]
7
8 fig, ax = plt.subplots()
9 for f in fs:
10     y = np.sin(2 * np.pi * f * t)
11     ax.plot(t, y, label='{0} Hz'.format(f))
12
13 ax.legend()
14 fig.savefig('basics_python.pdf')
```

State-machine approach

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 fs = [1, 2, 4]
5 all_time = np.linspace(0, 2, 200)
6 t = all_time[:100]
7
8
9 for f in fs:
10     y = np.sin(2 * np.pi * f * t)
11     plt.plot(t, y, label='{0} Hz'.format(f))
12
13 plt.legend()
14 plt.savefig('basics_python.pdf')
```

Example 3.5: Comparing the object-oriented approach of using `matplotlib` (top), and the state-machine approach (bottom).

There are two significant differences from MATLAB® worth keeping in mind. First, the “hold” state is on by default, and since version 2.0, the `hold` function is deprecated, or about to be removed. Instead, the developers recommend manually clearing the axes with `ax.clear()`. Second, `matplotlib`, like MATLAB®, plots one line per column when plotting a 2-dimensional array. Therefore, don’t forget to transpose your data before plotting something such as a 2-channel signal.

For more information, and an introduction to the structure of the package and the terms used throughout the documentation, read the “Usage” section of the `matplotlib` FAQ.

Seaborn is another plotting library worth knowing about. It uses `matplotlib` to render the figures and provides a high-level interface for plotting statistical graphics. A non-exhaustive list of features includes visualizing univariate and bivariate distributions, fitting and visualizing linear regressions, and comparing distributions across subsets of data. Example 3.6 is a “one liner” (split over lines 4-6) showing the survival rate of passengers on the Titanic. Table 3.2 shows the first five rows of data. It is loaded as a Pandas `DataFrame` on line 3, and passed as the data argument to the `factorplot` function. The rest of the arguments to `factorplot` declare how the different columns are encoded visually.

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 titanic = sns.load_dataset("titanic")
4 grid = sns.factorplot(x='class', y='survived', col='who',
5                       data=titanic, kind='bar', ci=None,
6                       order=['First', 'Second', 'Third'])
7 plt.savefig('seaborn_titanic.pdf')
```

Example 3.6: Using Seaborn to plot the survival rate of the Titanic passengers. Arguments to a function can be split over multiple lines without special continuation symbols. The *ci* argument is used to show confidence intervals. If confidence is 100%, setting it to *None* prevents the interval from being shown. This code produces Figure 3.2.

class	survived	who	age	fare
Third	0	man	22.0	7.2500
First	1	woman	38.0	71.2833
Third	1	woman	26.0	7.9250
First	1	woman	35.0	53.1000
Third	0	man	35.0	8.0500

Table 3.2: Subset of the titanic data used in Example 3.6.

Finally, when it comes to plotting, you are not limited to the built-in plotting libraries. For example, *Plot.py18* creates interactive web-based visualizations using the D3.js library. *Bokeh* has a similar goal but uses its own open source library. *Altair* generates JSON data following the *VEGA-Lite* specification, which in turn generates interactive HTML5 Canvas or SVG visualizations. All three are worth a look, especially if you frequently work inside Jupyter Notebooks.

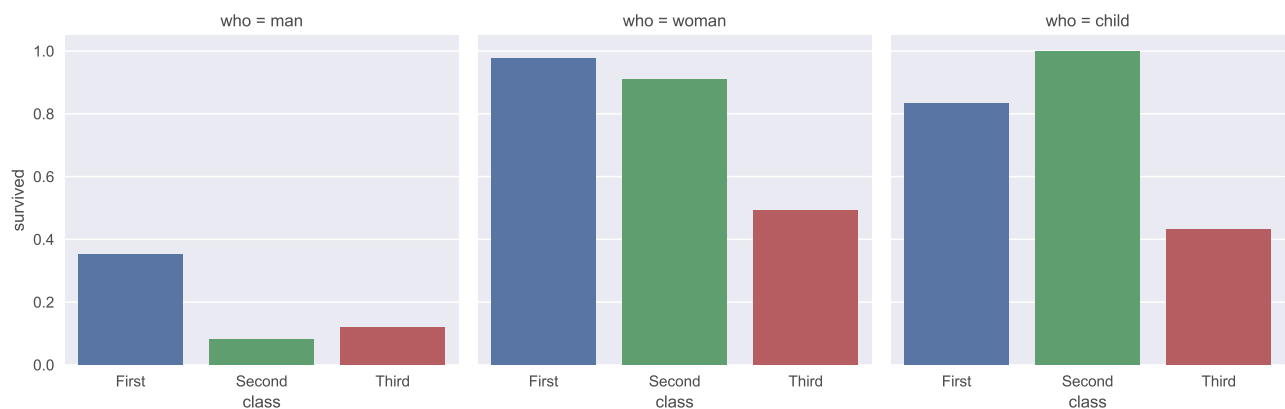


Figure 3.2: Survival proportions from the *titanic* data, as a function of class and whether the passenger was a man (male of 18 years of age or older), a woman or a child. The figure is generated by the code in Example 3.6.

Save Data

Python does not have a functional equivalent to “Save Workspace” in MATLAB®. The packages to read data mentioned in Table 3.1 all have the ability to write to their respective format. Here are some other options. Python has built-in support for serialization of arbitrary objects using the **pickle** package. It works well to serialize built-in types, but it is not very efficient when working with numerical data because it basically converts the arrays to text. NumPy provides the **np.save** function to save a single array to a file using the **.npy** format, and the **np.savez** function to save multiple arrays to a single file using the **.npz** format. The latter is basically a zipped archive of multiple **.npy** files. The **.npy** format is much more efficient than pickling when storing arrays because it can save the raw bytes, but it is not recommended for archival storage.

A great format for storing arbitrary data is the *Hierarchical Data Format*, version 5, more commonly called HDF5. It is a data format supported by a wide range of languages, including MATLAB®, C/C++, Fortran, Java, IDL, R, and Julia. In fact, MATLAB® **.mat** files version 7.3 and later use the HDF5 format. It is structured around two types of object: *groups*, which are basically folders, and *datasets*, which are basically files representing homogeneous multidimensional arrays. It is also possible to attach arbitrary metadata to any group or dataset using named attributes, and to automatically compress the data before writing it to disk.

There are two packages available in Python to interact with HDF5. They are h5py and PyTables. h5py exposes the HDF5 format using the interface of normal Python and NumPy objects, such as dictionaries and NumPy arrays. PyTables provides a higher level abstraction on top of HDF5 which allows database-like queries, computational kernels, and advanced indexing capabilities. Example 3.7 imports h5py and NumPy (lines 1-2), creates some data (lines 3-5), and opens the HDF5 file for writing (line 6). Writing data to the file involves assigning a value to a key in a dictionary. The key value corresponds to the “path” where the data should be saved. In this case, 'x' is stored as a dataset named 'x' in the 'inputs' group, and the outputs **y_sin** and **y_cos** are both saved in the 'outputs' group (lines 7-9). The File object must be closed (line 10).

```
1 >>> import h5py
2 >>> import numpy as np
3 >>> x = np.linspace(0, 2*np.pi, 100)
4 >>> y_sin= np.sin(x)
5 >>> y_cos= np.cos(x)
6 >>> f = h5py.File('my_data.h5', 'w')
7 >>> f['inputs/x'] = x
8 >>> f['outputs/y_sin'] = y_sin
9 >>> f['outputs/y_cos'] = y_cos
10 >>> f.close()
```

Example 3.7: Writing data to an HDF5 file.

If you decide to work with HDF5 files, consider installing the HDF Compass application. It is a file browser optimized for exploring HDF5 files.



What Else?

There is obviously much more to Python than what is mentioned here. Here are some tips for finding and evaluating other packages that may solve your problem.

1. Do a web search for the task you are trying to solve. It should probably include the words “python package.”
2. If the package you found is on GitHub, look at the date of the last commit, the version number, the number of stars, and the number of contributors. The more recent it is, and the larger the number of contributors, the more “trustworthy” a package is likely to be.
3. If it is on the Python Package Index (PyPI), look at when the most recent version was uploaded and follow links to the package’s home page or documentation page. Does it have documentation? Does it seem helpful?
4. Does the code look “Pythonic”? Does it follow the PEP-8 style guide? Does it require a lot of boilerplate code? (Most Python packages do not require a lot of boilerplate.)

Strategies for Converting to Python

The strategy you choose to convert a codebase to Python depends on multiple factors, including the code's size and the time available. We will explore two strategies. In the first, we reimplement a MATLAB® script in Python, one step at a time. It is likely to create the most downtime because you can use the Python implementation only once everything has been converted. However, it is the simplest. It is also the one that will provide the best performance because there will not be any data conversions between one language and the other.

The second strategy is to convert one function at a time, and to call the Python function from MATLAB®. This is possible thanks to MATLAB®'s ability to call arbitrary Python functions. The downside of this approach is that you will have to be very careful about type conversions. Not all types have a one-to-one mapping, nor are they all supported by the MATLAB® API. There will also likely be significant overhead in passing large arrays from one language to the other. Still, this option is appealing because it allows you to convert at your own rhythm, and to better manage risk.

From the Bottom Up: Converting One Function at a Time

Here is an overview of the process to reimplement a MATLAB® application in Python.

1. Refactor the MATLAB® code to have small(er), more testable functions.
2. Write tests for each MATLAB® function. Instead of generating the data in the scripts, save test data to disk. We will reuse it to test the Python implementation.
3. Pick a function, preferably the lowest level one. That's the one we'll start with. Write a Python test that uses the data you just saved.
4. Write the Python function to make the tests pass.
5. Repeat the process one function at a time, starting at item 2.
6. Extra: for functions that you "cannot" convert, consider calling MATLAB® from Python using the "MATLAB® Engine API for Python."

To understand the process, refer to this small script, which starts with three vectors (lines 1-2) that are rotated using a rotation matrix (lines 3-6). Then the slope of the vectors is calculated and printed before and after the rotation (lines 8-12), and both sets of vectors are plotted in the same figure (lines 14-26).

```
1 before = [[1 1 2];
2           [1 2 1]];
3 w = pi/2;
4 R = [cos(w) -sin(w);
5       sin(w) cos(w)];
6 after = R * before;
7
8 origin = zeros(size(before));
9 before_slope = (before(2, :) - origin(2, :)) ./ (before(1, :) - origin(1, :));
10 after_slope = (after(2, :) - origin(2, :)) ./ (after(1, :) - origin(1, :));
11 disp(['Before rotation: ', sprintf('%.2f', before_slope)]);
12 disp(['After rotation: ', sprintf('%.2f', after_slope)]);
13
14 hold on
15 p_origin = zeros(size(before));
16
17 x = [p_origin(1, :); before(1, :)];
18 y = [p_origin(2, :); before(2, :)];
19 plot(x, y)
20 text(before(1,:), before(2, :), num2cell(1:3))
21
22 x = [p_origin(1, :); after(1, :)];
23 y = [p_origin(2, :); after(2, :)];
24 plot(x, y)
25 text(after(1,:), after(2, :), num2cell(strcat(string(1:3), '')))
26 daspect([1 1 1])
```

The first step is to refactor the code into functions. Refactoring is the process of changing the internal structure of a program without changing its external behavior. Here there are three possible refactorings. The first is to put the vector rotation in its own function. The second is to create a slope function for the calculation of the slope. And the third is to create a **plotvectors** function to plot and annotate the vectors. If you want to learn more about refactoring, refer to the book *Refactoring* by Martin Fowler.

Here is the script after refactoring. The main code now contains only the top-level logic.

```
1 close all
2
3 before = [[1 1 2];
4           [1 2 1]];
5 w = pi/2;
6 after = rotate(before, w);
7
8 origin = zeros(size(before));
9 before_slope = slope(origin, before);
10 after_slope = slope(origin, after);
11 disp(['Before rotation: ', sprintf('%.2f', before_slope)])
12 disp(['After rotation: ', sprintf('%.2f', after_slope)]);
13 hold on
14 plotvectors(before, true)
15 plotvectors(after, false)
```

The repeated code has been refactored into the functions **rotate**, **slope**, and **plotvectors**.

```
1 function out = rotate(v, w)
2 % ROTATE Rotate matrix in Euclidean space.
3 % R = ROTATE(V, W) matrix V by W radians.
4
5 R = [cos(w) -sin(w);
6      sin(w) cos(w)];
7 out = R * v;

1 function s = slope(p1, p2)
2 % SLOPE Calculate the slope between two points in 2D space.
3 % S = SLOPE(P1, P2) calculates the slopes between P1 and P2, where both
4 % have size [2, nPoints].
5
6 s = (p2(2, :) - p1(2, :)) ./ (p2(1, :) - p1(1, :));

1 function plotvectors(v, isBefore)
2 % PLOTVECTORS Plot 2D vectors from origin.
3 % PLOTVECTORS(V, ISBEFORE) Plots set of vectors V, with one vector per
4 % column. ISBEFORE is a boolean. If false, the vector annotation will be
5 % appended with a quotation mark ("").
6
7 p_origin = zeros(size(v));
8 x = [p_origin(1, :); v(1, :)];
9 y = [p_origin(2, :); v(2, :)];
10 plot(x, y);
11 if isBefore
12     labels = num2cell(1:length(v));
13 else
14     labels = num2cell(strcat(string(1:length(v)), '"'));
15 end
16 text(v(1,:), v(2, :), labels);
```

The second step is to write tests for each MATLAB® function. In this case, the tests have a special purpose, to “map” the behavior of the function, not necessarily validate that it is “correct.” The test data will be used to validate that the new Python function behaves the same way as the MATLAB® one.

The interesting part of the script is how to save the inputs and outputs. For each function, one cell array is saved for the inputs and one for the outputs. Each set of inputs is itself placed in a cell array. This arrangement will allow you to easily loop through all the pairs of inputs and outputs when implementing the tests in Python. The data in this example is saved in **.mat** files but other formats, such as HDF5 or even CSV files, could have been chosen.

```
1 %% Generate data for rotate function
2 before = [[1 1 2];
3           [1 2 1]];
4 ws = [-pi, -pi/2, 0, 0.5, pi];
5 inputs = cell(size(ws));
6 outputs = cell(size(ws));
7 for i = 1:length(ws)
8     inputs{i} = {before, ws(i)};
9     R = rotate(before, ws(i));
10    outputs{i} = R;
11 end
12 save('rotate_data.mat', 'inputs', 'outputs')
13
14
15 %% Generate data for slope function
16 pls = [[1, 1, 2, 0, 1];
17        [1, 2, 1, 1, -1]];
18 p0s = zeros(size(pls));
19
20 inputs = cell(length(pls), 1);
```



```
21 outputs = cell(length(p1s), 1);
22 for i = 1:length(inputs)
23     inputs{i} = {p0s(:, i), p1s(:, i)};
24     s = slope(p0s(:, i), p1s(:, i));
25     outputs{i} = s;
26 end
27 save('slope_data.mat', 'inputs', 'outputs')
```

The third step is to implement the Python tests. MathWorks considers testing to be an “Advanced Software Development” technique, but it is quite easy to do in Python. To define a test in the `pytest` package, you must write a function whose name starts with `test_` in a file whose name also starts with `test_`. We'll start with the `test_rotate` function in the file `test_main.py`. The `rotate` function will be implemented in the `main.py` file, which does not exist yet. It is common to name the test file after the file being tested. In Example 4.1, we begin with importing the `testing` submodule of NumPy (line 1). It has helpful functions for testing the equality of *floating point* arrays. On line 2, we explicitly import the `loadmat` function, which will be used to read the mat files. On line 4, we import the first function that we want to test. Then comes the actual test function, which does not take any arguments. The following lines follow a pattern that will be reused. First we load the mat file, which returns a dictionary where the keys are the variable names (line 8). The `squeeze=True` argument to `loadmat` “squeezes out” unit dimensions. On line 9 and 10, we extract the inputs and outputs. On line 11, we iterate over the inputs and outputs at the same time. The `zip` function works like a zipper; it returns the first element of each sequence, in a tuple, which is then unpacked as two names, `inputs` and `outputs`. Finally, on line 12, we assert that the result of calling `rotate` with the inputs is numerically “close” to the outputs. An assertion continues silently if it evaluates to true and otherwise fails with an `AssertionError`, which will be caught by `pytest`. The `*inputs` syntax expands the elements of the `inputs` sequences.

Now that we have a test, we can run it within IPython with the command `!pytest`, as shown below. The exclamation mark tells IPython to execute the following text as a terminal command, not a Python function. The test fails with an `ImportError` because we have not yet written the `rotate` function.

```
1 In [1]: !pytest
2 ===== test session starts =====
3 platform darwin -- Python 3.5.2, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
4 rootdir: /Users/achabot/step1, inifile:
5 collected 0 items / 1 errors
6
7 ===== ERRORS =====
8 _____ ERROR collecting test_main1.py _____
9 ImportError while importing test module '/Users/achabot/step1/test_main1.py'.
10 Hint: make sure your test modules/packages have valid Python names.
11 Traceback:
12 test_main1.py:4: in <module>
13     from main import rotate
14 E ImportError: cannot import name 'rotate'
15 !!!!!!!!!!!!! Interrupted: 1 errors during collection
16 !!!!!!!!!!!!!
17 ===== 1 error in 0.97 seconds =====
```

```
1 import numpy.testing as npt
2 from scipy.io import loadmat
3
4 from main import rotate
5
6
7 def test_rotate():
8     data = loadmat('./rotate_data.mat', squeeze_me=True)
9     test_inputs = data['inputs']
10    test_outputs = data['outputs']
11    for inputs, outputs in zip(test_inputs, test_outputs):
12        npt.assert_allclose(rotate(*inputs), outputs)
```

Example 4.1: First test in `testmain.py` for the `rotate` function.

Step four is to write the code required to make the tests pass, which is the `rotate` function. It is very similar to the MATLAB® code, except for the `np.` prefix and the use of the `@` operator.

```
1 import numpy as np
2
3
4 def rotate(v, w):
5     """Rotate 2D matrix v by angle w in radians."""
6     R = np.array([[np.cos(w), -np.sin(w)],
7                  [np.sin(w), np.cos(w)]])
8     return R @ v
```

If the test is run again, it passes. Pytest shows a period for each test that passes.

```
1 In [2]: !pytest
2 ===== test session starts =====
3 platform darwin -- Python 3.5.2, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
4 rootdir: /Users/achabot/step1, inifile:
5 collected 1 items
6
7 test_main.py .
8
9 ===== 1 passed in 0.69 seconds =====
```

The fifth step is to go through the same process for each other function. The next test, for `test_slope`, has the same structure as `test_rotate`, except that data is loaded from a different file and we call the `slope` function. We also added the import from the `main.py` module.

```
1 from main import slope
2
3 def test_slope():
4     data = loadmat('./slope_data.mat', squeeze_me=True)
5     test_inputs = data['inputs']
6     test_outputs = data['outputs']
7     for inputs, outputs in zip(test_inputs, test_outputs):
8         npt.assert_allclose(slope(*inputs), outputs)
```

The next code listing has the code required to make `test_slope` pass. There are a few things that differ from the MATLAB® implementation. First, we’re using zero-based indexing. Second, we use only one index within the square brackets (`p0[0]`) instead of an index and a slice like in (`p1(1, :)`). That is because indexing the first dimension of a NumPy array returns all the elements below that dimension, instead of only one element like in MATLAB®.

```
1 def slope(p0, p1):
2     """Calculate slope between p0 and p1."""
3     return (p1[1] - p0[1]) / (p1[0] - p0[0])
```

Run the tests once more to make sure everything passes.

```
1 In [2]: !pytest
2 ===== test session starts =====
3 platform darwin -- Python 3.6.0, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
4 rootdir: /Users/achabot/step2, inifile:
5 collected 2 items
6
7 test_main.py ..
8 ===== 2 passed in 0.89 seconds =====
```

Testing the rest of the original script is harder, especially testing plotting functions. The two languages do not produce identical figures and automatic testing of “visual equality” is a difficult task, so manual testing is required. The MATLAB® code of `main.m` produces Figure 4.1. Function `plot_vectors` in `main.py` (below) was written to produce a figure that is as close to the original as possible. Notice that the function is renamed to follow the Python standard of *words_separated_by_underscores*. The `before` argument is also specified with a default value of `True`, so we don’t need to pass that value when not necessary (line 3). The matplotlib `annotate` function only adds one annotation at a time, so on line 7, we use the built-in function `enumerate` to automatically count how many vectors we have annotated. On lines 8 to 11, we use string formatting to append a quotation mark to “non-before” vector labels.

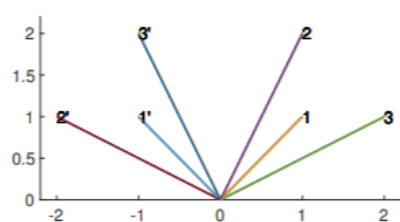


Figure 4.1: Figure generated when running `main.m`. The font size and line thickness have been adjusted to better render in this guide.

```
1 import matplotlib.pyplot as plt
2
3 def plot_vectors(v, before=True):
4     """Plot vectors v from origin."""
5     v_p = np.vstack((np.zeros_like(v), v))
6     plt.plot(v_p[:,2], v_p[1:,2])
7     for i, xy in enumerate(v.T):
8         if before:
9             label = '{}'.format(i)
10        else:
11            label= "{}'".format(i)
12        plt.annotate(label, xy)
```

We now have all the building blocks necessary to recreate the main script in Python. In the code below, we need to end the script with `plt.show()` to be sure that the figure is shown in a blocking manner. We removed the “hold” statement because it is the default in matplotlib.

```
1 before = np.array([[1, 1, 2],
2                   [1, 2, 1]])
3 after = rotate(before, np.pi/2)
4
5 origin = np.array((0, 0))
6 before_slope = slope(origin, before)
7 after_slope = slope(origin, after)
8 print("Before rotation: {}".format(before_slope))
9 print("After rotation: {}".format(after_slope))
10
11 plot_vectors(before)
12 plot_vectors(after, before=False)
13 plt.show()
```

There is one problem left. Running the tests brings up the user interface for the figure and blocks the execution of the tests. This is because running the tests actually executes the code in `main.py` as part of the import process. The solution is to put the “main” code inside a conditional statement that checks whether the file is being imported or run as a standalone script. This is done by comparing the value of the `'__name__'` variable to the string `'__main__'`. They will be equal if the script is run in a standalone manner. Otherwise, they will not be equal because `'__name__'` will have the value of the file name (`'main'` in this case). The final code produces Figure 4.2. The complete listing for the main code and the tests can be found in the appendix.

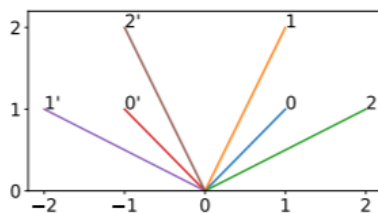


Figure 4.2: Figure generated when running `main.py`. The font size and line thickness have been adjusted to better render in this guide.

From the Top Down: Calling Python from MATLAB®

MATLAB® has built-in support for calling Python libraries. It is as easy as prefixing any call to a Python function with `py`, as you can see in the example below. The variable `a` is actually a Python object on the MATLAB® side:

```
1 >> a = py.numpy.arange(12).reshape(3, 4)
2 >> a.max()
3
4 ans =
5 11
```

MATLAB® can convert most of the built-in types between the two languages, but it does not allow you to easily pass arrays from one language to the other. In fact, it only supports converting 1D matrices to Python **array.array** and back. To simplify that task, we will use two functions, **mat2array** and **array2mat** (Examples 4.2 and 4.3).

mat2array takes a MATLAB® matrix and converts it to a NumPy array while preserving the shape. You can optionally specify a shape, which is useful when creating 1D NumPy arrays, and specifying a dtype. It should be reasonably efficient regarding its' memory usage.

The second function, **array2mat**, requires converting the NumPy array into a Python **array.array** type using the *typecode*, a one-character string representing the type of the array. This can be accessed directly on the NumPy array. The **array.array** type only accepts 1D arrays, so the array must be flattened before passing to the constructor (line 22). Finally, the **array.array** must be converted to a MATLAB® matrix with the correct type. We are taking advantage of the fact that NumPy dtypes and MATLAB® integer functions have the same name to create the typefunction with **str2func** (line 20). Floating point arrays are converted to double (line 14) and complex arrays are not supported, which is a limitation of **array.array**.

```

1 function a = mat2array(x, shape, dtype)
2 % MAT2ARRAY Convert MATLAB matrix to NumPy array.
3 % A = MAT2ARRAY(X) converts the matrix X to an array of the
4 % same shape (size) as X.
5 %
6 % A = MAT2ARRAY(X,SHAPE) reshapes the resulting array to have the shape
7 % SHAPE. The total number of elements cannot change. Useful when creating
8 % 1-D arrays.
9 %
10 % A = MAT2ARRAY(X,SHAPE,TYPE) or MAT2ARRAY(X,[],TYPE) specifies the NumPy
11 % dtype to use for the array. It will use NumPy's default if unspecified.
12 %
13 % See also ARRAY2MAT.
14
15 if nargin < 2 || isempty(shape)
16     shape = size(x);
17 end
18 if nargin < 3
19     dtype = py.None;
20 end
21
22 a = py.numpy.array(x(:)', dtype).reshape(shape);

```

Example 4.2: MATLAB® function *mat2array* to convert a matrix to a NumPy array while conserving the shape.

```

1 function m = array2mat(a)
2 % ARRAY2MAT(A) Convert NumPy array to MATLAB matrix.
3 % M = ARRAY2MAT(A) converts the array according to its type.
4 % Signed and unsigned integer arrays are converted using their
5 % specified precision. Float types are converted to "double".
6 % Complex arrays are not supported.
7 %
8 % See also MAT2ARRAY.
9
10 a = py.numpy.atleast_2d(a);
11 shape = cell2mat(cell(a.shape));
12 typename = char(a.dtype.name);
13
14 if startsWith(typename, 'float')
15     typefunction = @double;
16 elseif startsWith(typename, 'complex')
17     error('Complex types are not supported. The type was "%s."', ...

```

```
18         char(a.dtype.name))
19     else
20         typefunction = str2func(typename);
21     end
22
23     m = typefunction(py.array.array(a.dtype.char, a.ravel().tolist()));
24     m = reshape(m, shape);
```

Example 4.3: MATLAB® function *array2mat* to convert a NumPy array into a matrix while conserving the shape.

Given those two functions, here is a strategy for converting to Python, one function at a time.

1. Pick a function to convert to Python.
2. Write tests and generate test data for that function, if you do not already have any. Save the data to disk. (This is essentially the same process as in the previous section.)
3. If you do not have a test framework set up on the MATLAB® side, write a Python test for the function you are about to convert, using the data you just generated.
4. Implement the Python code required to make the tests pass.
5. Run the tests on the Python side.
6. Once the tests pass, replace the content of the MATLAB® function with a call to the Python function. You will likely need the functions **mat2array** and **array2mat** defined above to convert data both before and after calling your new Python function.
7. Run tests, on the MATLAB® side this time, if you have any.
8. Repeat until you have converted everything.

Here is an example to illustrate the process. Start with a function to multiply two things, called **multiplyby**.

```
1 function out = multiplyby(x, y)
2 out = x * y;
```

Then create some test data. The version below is simple, but your coverage should be more exhaustive. You should also implement proper tests with error reporting on the MATLAB® side.

```
1 % Generate test data
2 x = [0, 1, -1, 0.5, nan];
3 ys = [0, 1, -1, 0.5, nan];
4
5 inputs = cell(size(x));
6 outputs = cell(size(ys));
7 for i = 1:length(x)
8     inputs{i} = {x, ys(i)};
9     out = multiplyby(x, ys(i));
10    outputs{i} = out;
11 end
12 save('multiplyby_test_data.mat', 'inputs', 'outputs')
```

With that test data set, write a Python test function following the same iteration process through the inputs and outputs as described in the previous section. Notice how the function has been renamed according to Python standards.

```
1 import numpy.testing as npt
2 from scipy.io import loadmat
3
4 from main import multiply_by
5
6
7 def test_multiply_by():
8     data = loadmat('multiplyby_test_data.mat', squeeze_me=True)
9     test_inputs = data['inputs']
10    test_outputs = data['outputs']
11    for inputs, outputs in zip(test_inputs, test_outputs):
12        npt.assert_allclose(multiply_by(*inputs), outputs)
```

Then implement the Python function. The tests pass, though the output is not shown here.

```
1 def multiply_by(x, y):
2     return x * y
```

Finally, replace the body of the MATLAB® **multiplyby** function with a call to the Python implementation. Use the conversion functions to convert types between the two languages.

```
1 function out = multiplyby(x, y)
2 x_array = mat2array(x);
3 out_array = py.main.multiply_by(x_array, y);
4 out = array2mat(out_array);
```

The two strategies presented here should be sufficient for converting some—if not all—of your MATLAB® code to Python. There is a third strategy, which involves writing the “main” code in Python and calling MATLAB® using the MATLAB® Engine API for Python. The limitations are similar to calling Python from MATLAB®, only some data types are supported, and the memory overhead can be significant. Once again, NumPy arrays are not supported. Consequently, before choosing this option, review the Stack Overflow answer to the question “Improve performance of converting NumPy array to MATLAB double”. It suggests a modification to the **matlab** package to reduce the overhead of passing NumPy arrays to MATLAB®, and it is about 15 times faster than the default code when using a 1-million element array.

What Next?

For those looking to gain proficiency in Python as fast as possible, Enthought offers the Python for Scientists and Engineers course. It is five days of interactive training which provides a solid foundation on which to build high-quality software in terms of readability and performance. The first day covers the fundamental data types as well as control flow and code organization. The second day entails numeric data processing using NumPy, as well as data visualization with matplotlib. The third day is dedicated to data analysis of time series and tabular data using Pandas. The fourth day is split between best practices for writing good, readable, maintainable, and fast code, and how to create interfaces between Python and other languages such as C and C++. The week concludes with a one-day module on rapid development of scientific Graphical User Interfaces (GUIs).

To learn more, contact us at +1.512.536.1057 or visit our website.

Acknowledgments

Thanks to Matt Moelter and Christoph Scheidiger for their feedback on this document.

Appendix

Code Example: Profiling Contiguous Array Operations

Example for timing the difference between operating on contiguous v. non-contiguous memory. The benefit depends on the total number of elements and the number of elements along each dimension.

```
1 from timeit import timeit
2
3 setup = """
4 import numpy as np
5 a = np.arange(1000000000).reshape(10000, 10000)
6
7 def contiguous_sum(x):
8     for i in range(x.shape[0]):
9         x[i].sum()
10
11 def non_contiguous_sum(x):
12     for i in range(x.shape[-1]):
13         x[:, i].sum()
14 """
15
16 n=100
```



```
17 time_contiguous = timeit('contiguous_sum(a)', setup=setup, number=n) / n
18 time_non_contiguous = timeit('non_contiguous_sum(a)', setup=setup, number=n) / n
19 print("Contiguous: {:.4f}s per loop".format(time_contiguous))
20 print("None Contiguous: {:.4f}s per loop".format(time_non_contiguous))
21 print("Ratio: {:.3f}".format(time_non_contiguous / time_contiguous))
```

Complete Version of main.py, in Chapter 4

This is the complete content of `main.py`, as converted to Python in Chapter 4.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def rotate(v, w):
6     """Rotate 2D matrix v by angle w in radians."""
7     print(v, w)
8     R = np.array([[np.cos(w), -np.sin(w)],
9                  [np.sin(w), np.cos(w)]])
10    return R @ v
11
12
13 def plot_vectors(v, before=True):
14     """Plot vectors v from origin."""
15     v_p = np.vstack((np.zeros_like(v), v))
16     plt.plot(v_p[:,2], v_p[1:,2])
17     for i, xy in enumerate(v.T):
18         if before:
19             label = '{}'.format(i)
20         else:
21             label = "{}".format(i)
22     plt.annotate(label, xy)
23
24
25 def slope(p0, p1):
26     """Calculate slope between p0 and p1."""
27     return (p1[1] - p0[1]) / (p1[0] - p0[0])
28
29
30 if __name__ == '__main__':
31     before = np.array([[1, 1, 2],
32                       [1, 2, 1]])
33     after = rotate(before, np.pi/2)
34     origin = np.array((0, 0))
35
36     before_slope = slope(origin, before)
37     after_slope = slope(origin, after)
38     print("Before rotation: {}".format(before_slope))
39     print("After rotation: {}".format(after_slope))
40     plot_vectors(before)
41     plot_vectors(after, before=False)
42     plt.show()
```

This is the complete content of `testmain.py`.

```
1 import numpy.testing as npt
2 from scipy.io import loadmat
3
4 from main import rotate, slope
5
6
7 def test_rotate():
8     data = loadmat('./rotate_data.mat', squeeze_me=True)
9     test_inputs = data['inputs']
10    test_outputs = data['outputs']
11    for inputs, outputs in zip(test_inputs, test_outputs):
12        npt.assert_allclose(rotate(*inputs), outputs)
13
14
15 def test_slope():
```

```
16 data = loadmat('./slope_data.mat', squeeze_me=True)
17 test_inputs = data['inputs']
18 test_outputs = data['outputs']
19 for inputs, outputs in zip(test_inputs, test_outputs):
20     npt.assert_allclose(slope(*inputs), outputs)
```

References

Altied, Francesc, Ivan Vilata et al. (2002–). *PyTables: Hierarchical Datasets in Python*.

URL: <http://www.pytables.org>.

Bird, Steven, Ewan Klein, and Edward Loper (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc.

URL: <http://nltk.org>.

Bokeh Development Team (2017). *Bokeh: Python library for interactive visualization*.

URL: <https://bokeh.pydata.org>.

Bradski, G. (2000). “The OpenCV Library”. In: *Dr. Dobb's Journal of Software Tools*.

URL: <http://opencv.org>.

Collette, Andrew (2013). *Python and HDF5*. O'Reilly Media. ISBN: 978-1449367831.

URL: <http://www.h5py.org>.

Dijkstra, Edsger W (1982). *Why numbering should start at zero* (EWD 831).

URL: <http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF>.

Fowler, Martin and Kent Beck (1999). *Refactoring: improving the design of existing code*.

Addison-Wesley Professional. URL: <https://refactoring.com>.

Granger, Brian and Jake Vanderplas (2017). *Altair: Declarative statistical visualization library for Python*. URL: <https://github.com/altair-viz/altair>.

URL: <https://github.com/altair-viz/altair>.

Intel® (2017). *Intel® Math Kernel Library*. [Last accessed 2017-08-16].

URL: <https://software.intel.com/en-us/mkl>.

Kluyver, Thomas et al. (2016). “Jupyter Notebooks-a publishing format for reproducible computational workflows.” In: *ELPUB*, pp. 87–90.

DOI: <https://doi.org/10.3233/978-1-61499-649-1-87>. URL: <https://jupyter.org>.

Krekel, Holger and pytest-dev team (n.d.). *pytest: helps you write better programs*.

URL: <https://pytest.org>.

McKerns, Michael M. et al. (2012). “Building a Framework for Predictive Science”.

In: *CoRR* abs/1202.1056. <https://github.com/uqfoundation/mystic>.

URL: <http://arxiv.org/abs/1202.1056>.

Pedregosa, F. et al. (2011). “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12. <http://scikit-learn.org>, pp. 2825–2830.

URL: <http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf>.

Perez, Fernando and Bryan E. Granger (2007). “IPython: A System for Interactive Scientific Computing”. In: *Computing in Science Engineering* 9.3, pp. 21–29. ISSN: 1521-9615. DOI: <https://doi.org/10.1109/MCSE.2007.53>. URL: <http://ipython.org>.

Plotly (2017). URL: <https://plot.ly>.

Řehůřek, Radim and Petr Sojka (2010). “Software Framework for Topic Modelling with Large Corpora”. English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. <http://is.muni.cz/publication/884893/en>. Valletta, Malta: ELRA, pp. 45–50. URL: <https://radimrehurek.com/gensim/>.

Salvatier, John, Thomas V. Wiecki and Christopher Fonnesbeck (2016). “Probabilistic programming in Python using PyMC3”. In: *PeerJ Computer Science* 2, e55. DOI: <https://doi.org/10.7717/peerj-cs.55>. URL: <https://github.com/pymc-devs/pymc3>.

Seabold, Skipper and Josef Perktold (2010). “Statsmodels: Econometric and statistical modeling with python”. In: *9th Python in Science Conference*. <http://www.statsmodels.org>. URL: <http://conference.scipy.org/proceedings/scipy2010/pdfs/seabold.pdf>.

spaCy (2017). *spaCy: Industrial-Strength Natural Language Processing in Python*. URL: <https://spacy.io>.

Stack Overflow (visited 2020-05-01). “Improve performance of converting numpy array to MATLAB double.” URL: <https://stackoverflow.com/questions/45284124>

Swarztrauber, Paul N. (1985). “Vectorizing the FFTs”. In: *Parallel Computations*. Ed. by G. Rodrigue. Academic Press, pp. 51–83. ISBN: 978-0-12-592101-5. URL: <http://www.netlib.org/fftpack/>.

The MathWorks, Inc. (2017). MATLAB Engine API for Python. <https://www.mathworks.com/help/matlab/matlab-engine-for-python.html>. [Last accessed 2017-07-26].

van der Walt, Stéfan et al. (2014). “scikit-image: image processing in Python”. In: *PeerJ* 2, e453. ISSN: 2167-8359. DOI: <https://doi.org/10.7717/peerj.453>. URL: <http://scikit-image.org>.

Waskom, Michael et al. (2017). mwaskom/seaborn: v0.8.0. DOI: <https://doi.org/10.5281/zenodo.824567>. URL: <http://seaborn.pydata.org/>.