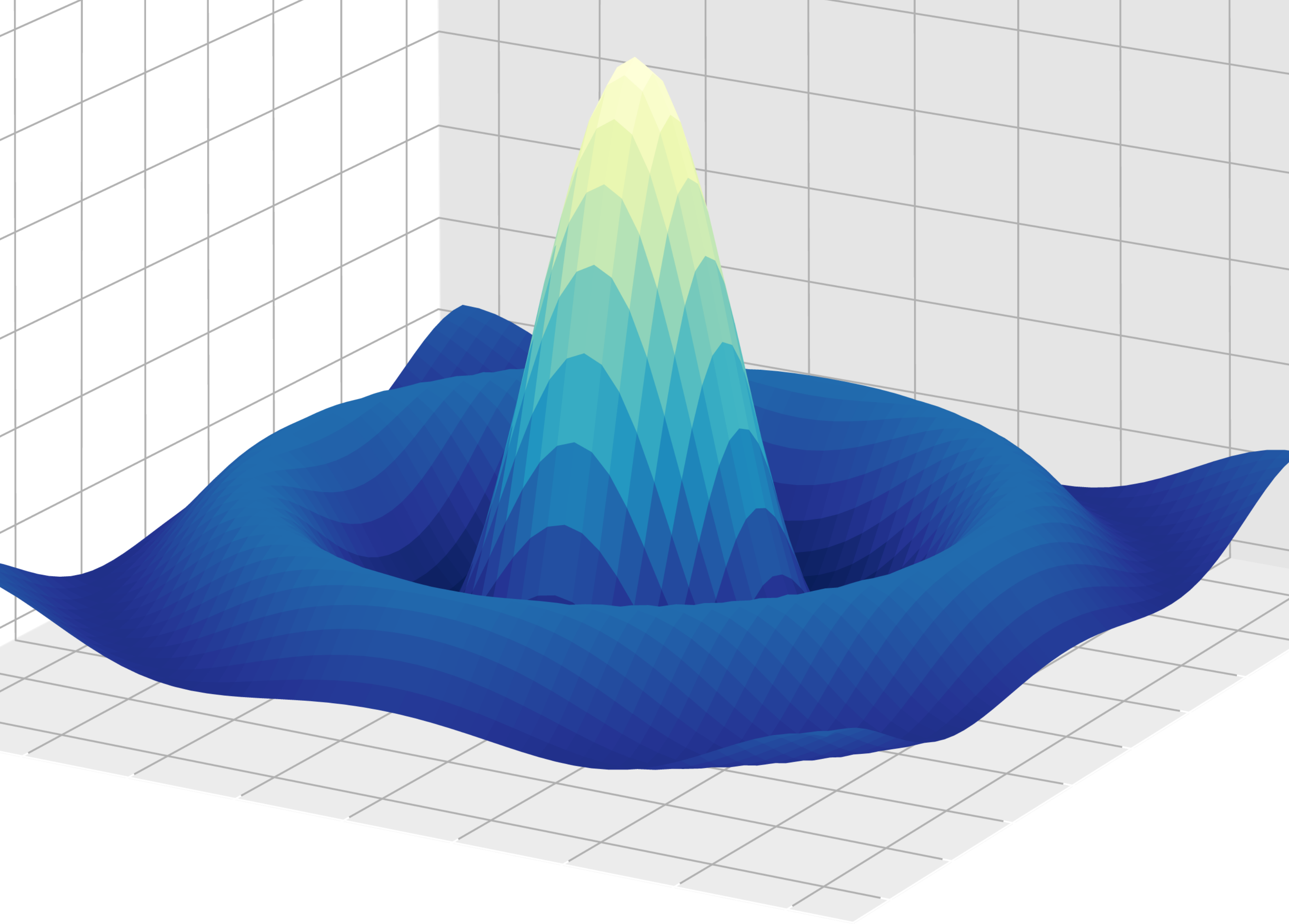

White Paper

MATLAB® to Python:

A Migration Guide

WRITTEN BY ALEXANDRE CHABOT-LECLERC

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

www.enthought.com

Version 1.0, September 2017

Contents

1 Introduction

This document will guide you through your transition from MATLAB® to Python. The first section presents some reasons why you would want to to do so as well as how to get started quickly by installing Enthought Canopy. The second section highlights some of the most important differences between the two languages, including 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.

Download Enthought Canopy at https://store.enthought.com/downloads/.

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 gradually transition to Python. Both rely on testing to validate that the new Python code works the same way (or is broken in the same way!) as your MATLAB® code. They approach the problem by either converting all function directly to Python or by calling Python from MATLAB®. You should use the one that is most convenient for your project.

Who Is This Guide For This guide is for long time MATLAB® users who want to migrate to Python, either partially or completely. I once was such a person.

Who Is This Document Not For If you rely heavily on the Simulink® graphical programming environment, you're out of luck. There is no good Simulink® equivalent in the Python ecosystem. If you have very special hardware integration needs, you *might* be able to find a package that works for you on the Python Package Index, but there is no guarantee that it will be actively maintained or that it will support all the features you need. However, there is nothing stopping you from implementing the features you need and sharing them with the world. Someone, somewhere,

For a good discussion about Simulink® alternatives, I recommend the article *Free and commercial alternatives to Simulink* on Mike Croucher's blog, Walking Randomly.

would surely be thankful for it. And finally, this is not the right document for you if you are new to programming.

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 3.6 but there is nothing in this guide that uses version-specific features.

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

If you are reading this guide, I assume that you do not need to be convinced too hard to transition to Python. I will nonetheless go through some of the reasons why someone might want to switch from MATLAB® to Python. At the very least, it might help you convince someone else. I split the reasons into five different categories: financial, freedom, technical, social, and curiosity.

FINANCIAL: Cost is often the first reason given for switching away from MATLAB®. It is certainly a good reason. Licensing fees add up quickly, especially if you rely on many toolboxes, and they might be a significant part of your budget if you are in a small organization. Python certainly has the appeal of being free, both as in “free beer”, or *gratis* and as in “free speech”. I’ll cover the free speech aspect in a moment. But to be honest, and to repeat the adage: “There is no such thing as a free lunch”. It is true that when using Python you do not have to pay a license fee from anyone, and that you have access to many free open source packages. However, be aware of the transition “fee”. One day you are a good MATLAB® developer, and the next you are a mediocre (for now, but I am sure you will improve!) Python developer. The gamble is that Python will allow you to be more agile and productive in the long term, which many individuals and companies have shown.

FREEDOM: Choosing Python, or any other open source language, lets you run your code “forever”. You are not 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 your colleagues, and people you don’t even know, can run your Python code without themselves buying a license. This can greatly improve the chances of survival of your project.

TECHNICAL: Python has the benefit of being first and foremost a *general purpose* programming language. It does happen to be a great language for scientific computing, and it even has some features specifically for that, but it's not *only* a scientific computing language. It can be used to do everything from building a file synchronization system (Dropbox), a photo-sharing service (Instagram), a 3-D modeling and video-editing application (Blender), a video hosting platform (YouTube), to discovering gravitational waves.[1]

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

[1] Python was used in most components of the Laser Interferometer Gravitational-Wave Observatory (LIGO) project. They have created a very interesting collection of tutorials.

SOCIAL: Maybe you have met a funny, clever and irresistible *Pythonista* at one of the many local meet-ups and thought you would check out their programming language of choice. The Python community is certainly a great reason to pick the language. There are the multiple PyCon conferences around the world, from the main one in North American, 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. I would be remiss if I failed to mention the PyLadies and Django Girls chapters around the world. Pick your location and your topic of choice and I am sure you will find a Python user group nearby. 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 the 800,000+ questions tagged with "Python" on Stack Overflow and the countless number of articles, books, and blog posts about Python.

Django is a high-level Web framework, and one of the most widely used Python packages. It is to Python what Rails is to Ruby, if that means something to you. More details at https://www.djangoproject.com.

You can find a calendar of upcoming Python events at https://www.python.org/events/.

MATLAB® has about 74,000 questions on Stack Overflow right now.

CURIOSITY: Finally, you might just be curious. There is nothing wrong with that.

Getting Started

The most straightforward way to start using Python, especially for numerical computing, is to install the free Enthought Canopy application. It most closely replicates the MATLAB® integrated development environment (IDE). Canopy is both an IDE and a Python distribution. The IDE integrates a code editor, an interactive IPython prompt, a variable browser, a graphical debugger, and a graphical package manager. The package manager provides simple, one-click install of more than 450 core scientific and analytic packages.

To install Canopy, first download the installer for your platform at https://store.enthought.com/downloads/. Since you are probably new to Python, I recommend selecting the most recent Python version available, which is Python 3.6 at the time of writing. Download the 2.7 version only if you plan on working on an older project that only supports Python 2. Code samples in this report should work in Python 2 but have only been tested using Python 3. Once Canopy is installed and you have opened the editor, you will be presented with the main screen shown in Figure 1.1.

The "Python pane" is an enhanced Python prompt called IPython.[2] It is a powerful interactive shell, with introspection, tab completion, and a rich media interface. The code editor at the top of the window offers syntax highlighting and syntax checking. The file browser on the left allows you to easily navigate through your file system. The status bar gives you quick access to the current syntax. In addition to the features shown in Figure 1.1, Canopy also provides a graphical debugger, a variable browser (similar to the Workspace viewer in MATLAB®) and a documentation browser. Canopy also integrates with the Jupyter project.[3] I will not be covering Jupyter in detail in this white paper but it is definitely worth your while to check it out. It allows you to create and share documents called *notebooks* that contain live code, equations, graphical visualizations, and text. It was originally developed for the Python language but can, in fact, be used with a wide variety of languages, including MATLAB®. To see representative Jupyter notebook examples, visit http://nbviewer.jupyter.org.

There are other ways of installing Python, including from the https://python.org website and from other scientific Python distributions. In my experience of teaching hundreds of scientists, analysts, engineers and data scientists, Canopy is the application and distribution that is easiest to get started with and stays the most out of your way.

[2] Fernando Perez 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

[3] Thomas Kluyver 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

If you want to try Jupyter with MATLAB®, you will need the `matlab_kernel` package. See https://github.com/calysto/matlab_kernel for details.

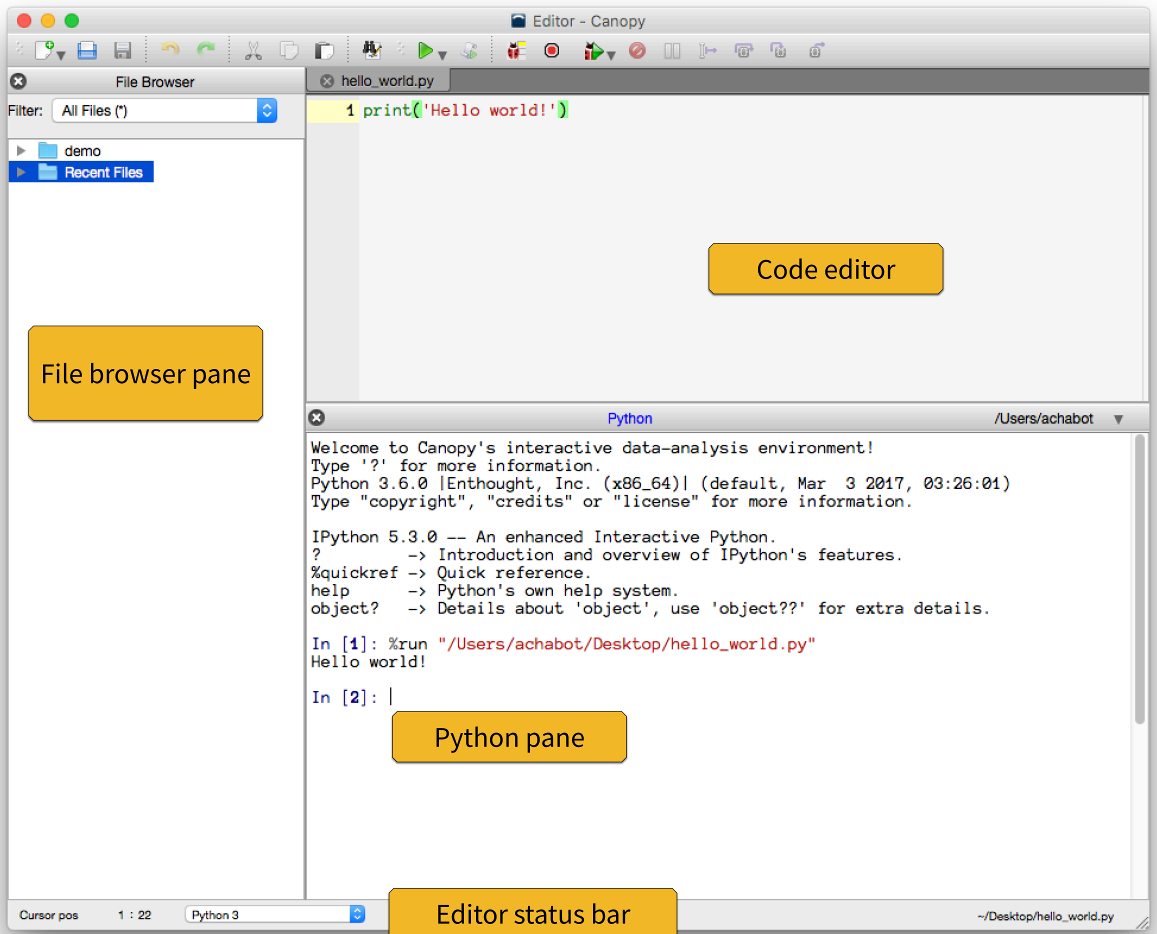

Figure 1.1: Overview of the Canopy Editor.

2
Differences Between Python and MATLAB®

As a MATLAB® user, you are in luck. In the grand scheme[1] of programming languages, Python and MATLAB® are not that different from each other. It's not as if you were coming from Lisp or Objective C. Still, "not that different" does not mean that they are "the same". In this section, I will highlight a few areas where they differ. Some are simple to assimilate, such as Python using zero-based indexing, or the meaning of the different types of brackets. Others might be a little harder to grasp, 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.

[1] Pun totally intended.

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. That need is addressed by the NumPy package[2], which provides multidimensional arrays. With NumPy, numerical computing will be as fast and as concise as in MATLAB®. I will cover NumPy in more detail on page 16. 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. The following list presents the types in more detail. The side notes next to each type show some code examples.

[2] I will cover this in more detail later, but a package is equivalent to a toolbox.

- 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.

```
real_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 you cannot modify them.You instead create new strings based on the contents of an existing one.

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

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.

- 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 items of different types at the same time, they tend to be homogeneous sequences of items, such as filenames, words, database rows, tasks, etc. You can select one item at a time based on its position, which is called *indexing*, or a subset, 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 lists' cousins. 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 to represent 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 your new favorite friends. They are similar to MATLAB® structures, but allow for arbitrary keys, as long as they are immutable.[3] That means you are not limited to strings. The values can be any objects.

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

[3] That's numbers, strings, and tuples if you are keeping track.

- Sets are not used very often but allow for expressive "set operations", such as intersection, union, difference, etc. 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'}
```

These are all the built-in types. Other types must be imported from the standard library or from third party packages. The most important third-party type, when coming from MATLAB®, is probably NumPy arrays. We will discuss more about them later, on page 16. For now, let's just say that they are 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 towards 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 folder of Python files is a *package*. Python looks up modules and packages in a list of locations stored in the `sys.path` variable.[4] 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 frown upon modifying the path variables (both `PYTHONPATH` and `sys.path`), preferring instead to install packages in a standard location called *site-packages*.[5] For cases where multiple versions of the same package would conflict with each other, the preferred solution is to create a standalone 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, I recommended reading the documentation for the Enthought Deployment Manager, which powers Canopy, as well as the "Installing Packages" section of the Python Packaging User Guide.

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

[5] Your package manager, such as edm or pip, usually installs packages there. To find where it is, execute the following: `import site; site.getsitepackages()`

Syntax

Example 2.1 shows some key syntax similarities and differences between the two languages. It contains many common types and operations. In it, I plot three sinusoids with different frequencies and save the figure to a PDF. It is shown on the right (Figure 2.1).

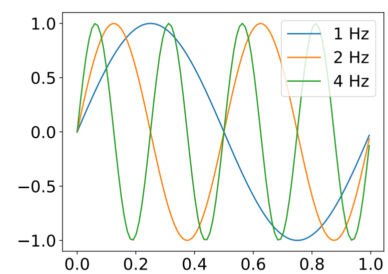

Figure 2.1: Figure generated by running the Python script in Example 2.1.

Python

```python
import numpy as np
import matplotlib.pyplot as plt

fs = [1, 2, 4]
all_time = np.linspace(0, 2, 200)
t = all_time[:100]

for f in fs:
    y = np.sin(2 * np.pi * f * t)
    plt.plot(t, y, label='{} Hz'.format(f))

plt.legend()
plt.savefig('basics_python.pdf')
```

MATLAB®

```matlab



fs = [1 2 4];
allTime = linspace(0, 2, 200);
t = allTime(1:100);
hold('on')
for f = fs
    y = sin(2 * pi * f * t);
    plot(t, y, 'DisplayName', sprintf('%d Hz', f));
end
legend('show')
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®.

Let's start off with the semicolons (;) at the end of each line on the MATLAB® side. You don't need them anymore. In Python, you have to explicitly *display* something, instead of explicitly silencing it.[6] No more unwanted deluge of scrolling data.

[6] It is mainly done with the `print` function.

In the first two lines on the Python side are the *import statements*. They define new names in the current namespace, `np` and `plt`, which I can then refer to later. Both new names are aliases, first to the NumPy package and then 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 — so I don't have to repeatedly type the full name of a package or module each time I use one of its functions. This is quite different from MATLAB®, where any name is available as long as the file where the function is defined is on the "path". Including those imports might seem annoying and repetitive at first, but it actually 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 on the Python side and an array on the MATLAB® side. Items always have to be separated by commas in Python. Then, on line 5, I create an array of 200 points, between 0 and 2 inclusively. The array creation function on the Python side is part of NumPy, therefore we use the `np.` prefix to tell Python to look for the `linspace` function under the NumPy namespace. Generallly speaking, the period means "look up the name on the right, under the namespace on the left". It does wonders for readability. On line 6, I select only the first hundred points. 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 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 is to *call* a function or a class. This is always the case when parenthesis are "attached" to a name, such as in `print('Hello world')`, or `np.arange(10)`. On the MATLAB® side, the square brackets are only used to define matrices, and the parentheses are used to both slice and call.

On line 7, I turn on "hold" on the MATLAB® side, 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 function. The line indentation by 4 space characters is meaningful and demarcates the body of the for-loop (and also of functions, classes, while-loops, etc.). On the Python side, on line 10, the `plot` function is part of the `pyplot` namespace. Still on the Python side, 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, I display the legend on line 12 and save the figure as a PDF on line 13. On the Python side, matplotlib knows to automatically save the most recently active figure, the same way both matplotlib and MATLAB® know where a line should go when calling `plot`. The section on plotting, on page 27, will cover matplotlib in more detail.

In summary, Python requires explicit imports for the required 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,[7] which means that the lower bound is included but the upper bound is not. You might think zero-based indexing is a terrible idea, in which case I would point you to a well-known paper by Dijkstra[8] about why indexing should start at `0`. If you are still not convinced after reading his argument, here are two examples of the simplicity provided by this design decision.

[7] That is $[\text{low}, \text{high})$ in mathematical terms.

[8] Edsger W Dijkstra (1982). *Why numbering should start at zero (EWD 831)*. URL: http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF
Dijkstra had a really neat hand writing. He was also quite proactive, he numbered his publications himself and did not wait for librarians to do so!

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.

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

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

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

Contrast that to MATLAB®, which requires two "+1"s:

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

ans =
```

```

logical

   1
```

Be honest, how likely are you to forget the +1?

Here is another example. Given a 2D image, `img`, stored in row-major order,[9] 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)`. Now there is a -1 in there!

Here is a last one[10] where we repeatedly select *step* consecutive elements in a sequence of letters. On the MATLAB® side, we need a -1 in the slice because of the inclusion of the upper bound.

The common joke is that "There are only two hard things in Computer Science: cache invalidation, naming things, and off-by-one errors."

[9] That means the array is stored as a sequence of rows. See p. 16 for more details about the differences between row-major and column-major order.

[10] Sorry, this is the third one. I am off by one.

Python

```python
>>> letters = 'abcdef'
>>> step = 2
>>> for offset in range(0, len(letters),
...                         step):
...     print(letters[offset:offset+step])
ab
cd
ef
```

MATLAB®

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

Example 2.2: The Python range function, in this case, expects `(low, high, step)`. On the Python side, the step size doesn't lie.

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 is the one-before-last element, -3 is the third element from the end, and so on and so forth. Combine this syntax with the fact that it is not necessary to specify upper bound if the slice goes all the way to the end and you get a very compact notation to get the last *N* elements of a sequence:

```python
>>> a = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> a[-1]
'g'
>>> last_three = a[-3:]
>>> last_three
['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 array's number of axes (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* elementwise unless specified otherwise using specific methods or notation. Therefore, there is no need for the `.*` and `./` operators used in MATLAB®. Instead, to perform a matrix product, you must use the `@` operator (available in Python 3 only) or explicitly call 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 dimension 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.

```
>>> import numpy as np
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> e = np.eye(3)
>>> e
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> a * e
array([[ 0.,  0.,  0.],
       [ 0.,  4.,  0.],
       [ 0.,  0.,  8.]])
>>> a @ e  # or a.dot(e) in Python 2
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 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, whereas they are *appended* in column-major order (MATLAB®), as illustrated in Figure 2.2

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

Python

```
>>> a = np.arange(24).reshape(2, 3, 4)
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

MATLAB®

```
>> a = reshape(0:23, 3, 4, 2)

a(:,:,1) =

     0     3     6     9
     1     4     7    10
     2     5     8    11


a(:,:,2) =

    12    15    18    21
    13    16    19    22
    14    17    20    23
```

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

- Multichannel signals with S signals of T samples often have the shape `(S, T)`.
- The shape of images depends on the access patterns. The scikit-image[11] library uses the following shapes, with `pln` being a "plane" and *ch* a channel, such as red, green and blue.
  - 2D grayscale images have the shape `(row, col)`;
  - 2D multichannel images (eg. RGB) have the shape `(row, col, ch)`;
  - 3D grayscale images have the shape `(pln, row, col)`;
  - 3D multichannel images have the shape `(pln, row, col, ch)`;
  - Time-varying 3D images have the shape `(t, pln, row, col, ch)`.

[11] Stéfan van der Walt 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

THE COMBINATION OF THE MEMORY LAYOUT and of the access pattern on an array can have a significant impact on performance. Because of how modern processors prefetch 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 inner most dimension. That would be the last one in Python and the first one in MATLAB®. Example 5, 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 be can 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 much faster as well as 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.

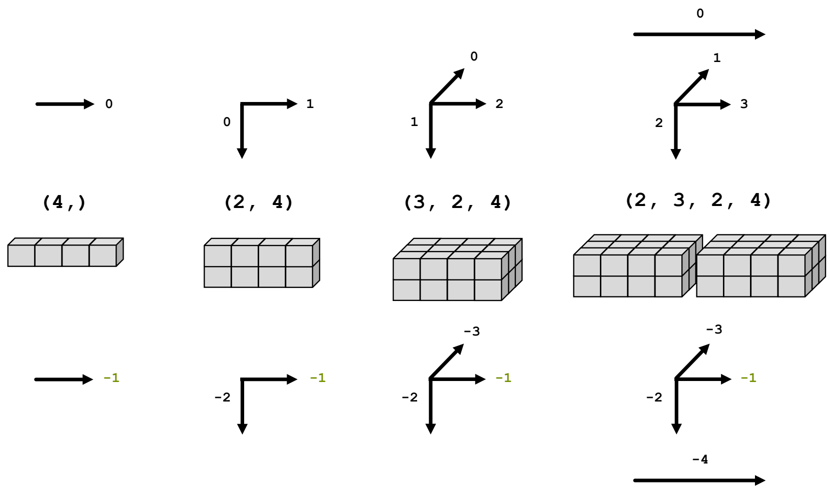

Figure 2.2: Visualization of a one-, two-, three-, and four-dimensional NumPy arrays. The arrows on the top point in the direction of each axis, with zero always being the first element of the shape tuple. The arrows at the bottom illustrate how the *last* axis (-1) always corresponds to the columns.

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 how it is 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 the code is organized in functions (or *procedures*, hence the name), and *object-oriented* (OO), where data and code are grouped together. Although both languages support both paradigms, they each lean towards a different side. Python tends to be more object-oriented and MATLAB® tends to be more procedural.

A function or expression has a "side effect" if it modifies some state outside of its scope, such as changing a global variable, or modifying data in place.

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. In the real world, you and I are of type `Human`, but we each have our own identity. The parallel is in fact quite apt; objects often model real-world objects. 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 idiom 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 them. Example 2.5 compares the OO approach (on the left) and the procedural approach (on the right) when defining a NumPy array, reshaping it, and then calculating its overall and per-column maxima. On the OO side, the array is not passed to each method. On the procedural side, all the functions are under the NumPy namespace (`np.`) and the arrays `a` and `b` are passed to each call.

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

Object-Oriented

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

Procedural

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

In the case of NumPy, no syntax is preferred to the other. I tend to use methods more often than functions but use whichever you are most comfortable with.

ANOTHER COMMON PATTERN allowed by the OO approach is to chain methods rather than nesting functions. Example 2.7 shows two comparisons between method chaining and function nesting. In the first, I remove trailing whitespace in a string, convert it to uppercase, and then replace spaces with underscores. In the second, I define an array, reshape it and calculate one maximum per column. The left-hand side shows how it is done in Python with chained method calls and the right-hand side shows the MATLAB® version using nested functions.

I would argue that method chaining is easier to read because of the flow of data from left to right. Conceptually it is like passing the data through a sequence of filters.

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 2.6.

Example 2.6: 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.

```
1 >>> words = ['Hello', 'World!']
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
```

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 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.

Python

```
>>> sentence = "the quick brown fox    "
>>> sentence.strip().upper().replace(' ', '_')
'THE_QUICK_BROWN_FOX'




>>> # Now with an array
>>> a = np.arange(12)
>>> a.reshape(3, 4).max(axis=0)
array([ 8,  9, 10, 11])
```

MATLAB®

```
>> sentence = "the quick brown fox    ";
>> replace(upper(strip(sentence)), " ", "_")

ans =

    "THE_QUICK_BROWN_FOX"

>> % Now with an array
>> a = 0:11;
>> max(reshape(a, 3, 4), [], 1)

ans =

     2     5     8    11
```

Example 2.7: Transforming a string and an array using chained methods in Python (left) and nested functions in MATLAB® (right). 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.

3
How Do I?

I know learning for the sake of learning is gratifying and fun, and learning a new programming language is particularly fun, but we (I include myself in there!) should not forget that we program to get some work done. 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 finally save the results. I am aware that things rarely happen in such a linear fashion, but you certainly get the point. 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, I typically recommend 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 packages in *italic* are part of the standard library.

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

I will not give an example for each of the data types mentioned above but will rather focus on the MAT file type, as well as the Pandas package used to read tabular data and Excel files. Section 3 will present the h5py package in more detail.

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, I create three variables in a MATLAB® script and save them to the MAT file named `my_data.mat`.

```
clear;
my_scalar = 1;
my_1d_array = [1, 2, 3];
my_2d_array = [1 3 5;
               2 4 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, I read the data using the `loadmat` function. The three matrices are loaded with the same number of dimensions they had in MATLAB®. They are all two-dimensional.

```
>>> from scipy.io import loadmat
>>> data = loadmat('my_data.mat')
>>> data['my_scalar']
array([[1]], dtype=uint8)
>>> data['my_1d_array']
array([[1, 2, 3]], dtype=uint8)
>>> data['my_2d_array']
array([[1, 3, 5],
       [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.

In Example 3.3, I load the data 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 I find that is actually useful when converting code to Python because it yields arrays of similar shape as what NumPy would create.

THE PANDAS PACKAGE implements the *DataFame* 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 = loadmat('my_data.mat', squeeze_me=True)
>>> data['my_scalar']
1
>>> data['my_1d_array']
array([1, 2, 3], dtype=uint8)
>>> data['my_2d_array']
array([[1, 3, 5],
       [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. If has the effect of "squeezing" out unit matrix dimensions. It is equivalent to calling the function `np.squeeze` on all the inputs.

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. You will definitely want to add it to your list of things to learn about. I will only focus here on Pandas' ability to read data from various sources.

All of the functions to read data in Pandas start with the prefix `read_`. There are currently (Pandas 0.20) 16 different functions. Here are five. To read data from text files, the main functions are `read_csv` and `read_table`. They are in fact the same function but use a different default separator, respectively the comma and the 'tab' character. 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[1] 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.

[1] Pandas relies on the SQLAlchemy package for connecting to the database and executing the query. All the databases supported by SQLALchemy are therefore also supported by Pandas. For more details, see http://docs.sqlalchemy.org/.

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, windows functions, peak finding, spectral analysis, and more. The `scipy.fftpack` provides bindings to the FFTPACK[2] 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 Enthought Canopy is linked against the Intel® Math Kernel Library (MKL)[3], which can be orders of magnitude fast than FFTPACK for non-power-of-two signal lengths.

[2] Paul N. Swarztrauber (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/

[3] Intel® (2017). *Intel® Math Kernel Library*. [Last accessed 2017-08-16]. URL: https://software.intel.com/en-us/mkl

Linear algebra

NumPy *does* provide a `numpy.matrix` type, which is a specialized 2-D array, but you should not use it.[4] 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 linear algebra functions from SciPy if you have installed it.

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)` otherwise, instead of `a \ 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.

[4] It is not me who says it, it is the NumPy developers. See https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

Machine learning

Using Python gives you access to the scikit-learn[5] machine learning package, which might by itself be a good enough reason to transition to Python. It provides fast and 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 `fix(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 2-D 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 15. Figure 3.1 shows the first of the predicted digits (line 18), which

[5] F. Pedregosa 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

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.

Example 3.4: Recognizing handwritten digits with scikit-learn.

```
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])
```

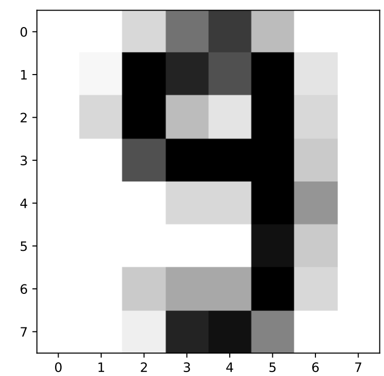

Figure 3.1: First digit of the test sequence.

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[6], 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[7] is a package for Bayesian modeling and probabilistic machine learning with a focus on Markov chain Monte Carlo and variational fitting algorithms.

[6] Skipper Seabold 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

[7] John Salvatier, 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

Image Processing and Computer Vision

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

- The `scipy.ndimage` submodule of SciPy, with functions for multidimensional image processing, including filtering, interpolation, measurements and morphology analysis.

- sckikit-image[8], which extends the `scipy.ndimage` submodule, with more filtering capabilities, color-space transformations, restoration, segmentation, various transformations and more.
- OpenCV[9], the computer vision and machine library, has native Python bindings and requires only the `opencv` package.

[8] Stéfan van der Walt 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

[9] G. Bradski (2000). "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools*. URL: http://opencv.org/

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[10] framework for "highly-constrained non-convex optimization and uncertainty quantification".

[10] Michael M. McKerns 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

Natural Language Processing

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

- NTLK[11] 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[12] is designed to "get things done". It implements a subset of NLTK's features but is more readily usable. It is also faster.
- gensim[13] 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.

[11] Steven Bird, 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

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

[13] Radim Řehůřek 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/

Data Visualization

The main package for data visualization is called matplotlib. It was designed[14] 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`,[15] which is usually imported as `plt` with `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

[14] matplotlib was created in 2002.

[15] You can find the "Pyplot Tutorial" here: http://matplotlib.org/users/pyplot_tutorial.html

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 then 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 the left to 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='{} 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='{} Hz'.format(f))
12
13 plt.legend()
14 plt.savefig('basics_python.pdf')
```

Example 3.5: Comparing the object-oriented approach of using matplotlib (left) and the state-machine approach (right).

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.[16] 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. It makes it easy to convert to Python, but it is rather strange given that Python and NumPy are row-major. NumPy users would expect one line per row, but it is not how it works. Therefore, don't forget to transpose your data before

[16] "Deprecated" means that it is about to be removed.

plotting something such as a 2-channel signal, otherwise, you will end up with a Jackson Pollock-like display of two-point lines!

A last note about matplotlib: I highly recommend reading the "Usage" section of the matplotlib FAQ. It has a great introduction to the structure of the package and to the terms used throughout the documentation.

Seaborn[17] 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 to 6...) showing the survival rate of passengers on the Titanic. Table 3.2 shows the first five rows of the 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.

[17] Michael Waskom et al. (2017). *mwaskom/seaborn: v0.8.0*. DOI: https://doi.org/10.5281/zenodo.824567. URL: http://seaborn.pydata.org/

```
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. Let's say I am 100% confident about the data here, so setting it to `None` prevents the interval to be 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, one great feature of Python when it comes to plotting is that you are not limited to the built-in plotting libraries. Yes, some libraries, such as Seaborn, use matplotlib under the covers, but many do not. For example, Plot.py[18] creates interactive web-based visualizations using the D3.js library. Bokeh[19] has a similar goal but uses its own open source library. Altair[20] generates JSON data following the VEGA-Lite specification,[21] which in turn generates interactive HTML5 Canvas or SVG visualizations. All three are worth a look, especially if you frequently work inside Jupyter Notebooks.

[18] Plotly (2017). URL: https://plot.ly

[19] Bokeh Development Team (2017). *Bokeh: Python library for interactive visualization*. URL: https://bokeh.pydata.org

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

[21] Learn more about the VEGA visualization grammar at https://vega.github.io.

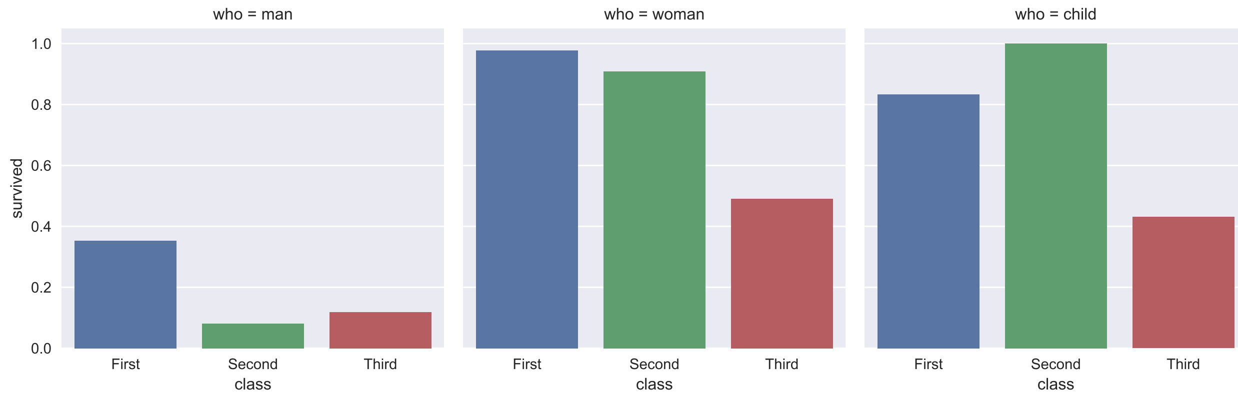

Figure 3.2: Survival proportions from the *titanic* data, as a function of class and whether the passenger was a man (male of 18 years of age or older), a woman or a child. The figure is generated by the code of Example 3.6.

Save Data

Python does not have a functional equivalent to "Save Workspace" in MATLAB®, which I would argue is a good thing. The Save Workspace functionality might seem convenient because it allows you to "pause" your current work session. However, it does not save the state of the code with it. This means that it is possible to be in an inconsistent state, with code that could never generate the data that has been reloaded in a later session. Nevertheless, I understand that it is useful to save a subset of variables for use cases such as caching, or when serializing data to communicate with another application or process.

Serialization is the process of transforming a data structure or object into a format that can be stored and reconstructed later. See Wikipedia: Serialization for more details.

The packages to read data mentioned in Table 3.1 (p. 22) 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.[22] 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` functions 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.

[22] It is for object *preservation*, get it?

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 objects: *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[23] and PyTables[24]. h5py exposes the HDF5 format using the interface of normal Python and NumPy objects, such as dictionaries and NumPy arrays. I will demonstrate it below. PyTables provides a higher level abstraction on top of HDF5 which allows database-like queries, computational kernels, and advanced indexing capabilities. In Example 3.7, I import h5py and NumPy (lines 1–2), create some data (lines 3–5) and open the HDF5 file for writing (line 6). Writing data to the file is as easy as 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).

[23] Andrew Collette (2013). *Python and HDF5*. O'Reilly Media. ISBN: 978-1449367831. URL: http://www.h5py.org

[24] Francesc Alted, Ivan Vilata, et al. (2002–). *PyTables: Hierarchical Datasets in Python*. URL: http://www.pytables.org/

Example 3.7: Writing data to an HDF5 file.

```
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()
```

If you decide to work with HDF5 files, I recommend installing the HDF Compass application.[25]. It is a file browser optimized for exploring HDF5 files.

[25] You will find installation instructions at this link: https://support.hdfgroup.org/projects/compass/index.html

What Else?

There is obviously much more to Python than what I have 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 it has, 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.

4. Does it have documentation? Does it seem helpful?

5. Does the code look "Pythonic"? This is something hard to judge at first, but here are a couple of hints. Does it follow the PEP-8 style guide? Does it require a lot of boilerplate code? Most Python packages do not.

4
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. I present here two approaches. In the first one, I reimplement a MATLAB® script in Python, one step at at time. It is likely to create the most downtime because you can use the Python implementation only once *everything* has been converted. However, I would say it is the simplest. It 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 the 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 (you already do that, right? Right?). 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”[1].

[1] 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]

I created a small script to illustrate the process. I start with three vectors (line 3), which I rotate using a rotation matrix (lines 5 and 6). Then I calculate and print the slope of the vectors before and after the rotation (lines 10–14), and finally, I plot both sets of vectors in the same figure (lines 16–27).

```
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. I identified 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, I highly recommend reading *Refactoring* by Martin Fowler[2]. It is a surprisingly pleasant read with many great ideas.

[2] Martin Fowler and Kent Beck (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional. URL: https://refactoring.com

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

```
close all

before = [[1 1 2];
          [1 2 1]];
w = pi/2;
after = rotate(before, w);

origin = zeros(size(before));
before_slope = slope(origin, before);
after_slope = slope(origin, after);
disp(['Before rotation: ', sprintf('%.2f, ', before_slope)])
disp(['After rotation: ', sprintf('%.2f, ', after_slope)]);
hold on
plotvectors(before, true)
plotvectors(after, false)
```

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

```
function out = rotate(v, w)
% ROTATE Rotate matrix in Euclidean space.
%   R = ROTATE(V, W) matrix V by W radians.

R = [cos(w) -sin(w);
     sin(w)  cos(w)];
out = R * v;
```

```
function s = slope(p1, p2)
% SLOPE Calculate the slope between two points in 2D space.
%   S = SLOPE(P1, P2) calculates the slopes between P1 and P2, where both
%   both are have size [2, nPoints].

s = (p2(2, :) - p1(2, :)) ./ (p2(1, :) - p1(1, :));
```

```
function plotvectors(v, isBefore)
% PLOTVECTORS Plot 2D vectors from origin.
%   PLOTVECTORS(V, ISBEFORE) Plots set of vectors V, with one vector per
%   column. ISBEFORE is a boolean. It false, the vector annotation will be
%   appended with a quotation mark ("'").

p_origin = zeros(size(v));
x = [p_origin(1, :); v(1, :)];
y = [p_origin(2, :); v(2, :);];
plot(x, y);
```

```
if isBefore
    labels = num2cell(1:length(v));
else
    labels = num2cell(strcat(string(1:length(v)), "'"));
end
text(v(1,:), v(2, :), labels);
```

THE SECOND STEP is to write tests for each MATLAB® function. Ideally, I would already have such tests to validate that each function is correct and deals properly with edge cases, but I do not have any test data right now, so I will create some. In this case, the tests have a special purpose. I want to "map" the behavior of the function, not necessarily validate that it is "correct". I will use that data to validate that the new Python function behaves the same way as the MATLAB® one.

I wrote some code to generate some of that test data. It could definitely be more exhaustive. The interesting part of the script is how to save the inputs and outputs. For each function, I save a cell array for the inputs and one for the outputs. Each set of inputs is itself placed in a cell array. This arrangement will allow me to easily loop through all the pairs of inputs and outputs when implementing the tests in Python. I am saving the data in `.mat` files because it is convenient and because Python can read that format via the `scipy.io.loadmat` function. I could have chosen other formats, such as HDF5 or even CSV files.

```
%% Generate data for rotate function
before = [[1 1 2];
          [1 2 1]];
ws = [-pi, -pi/2, 0, 0.5, pi];
inputs = cell(size(ws));
outputs = cell(size(ws));
for i = 1:length(ws)
    inputs{i} = {before, ws(i)};
    R = rotate(before, ws(i));
    outputs{i} = R;
end
save('rotate_data.mat', 'inputs', 'outputs')


%% Generate data for slope function
p1s = [[1, 1, 2, 0, 1];
       [1, 2, 1, 1, -1]];
p0s = zeros(size(p1s));

inputs = cell(length(p1s), 1);
```

```
outputs = cell(length(p1s), 1);
for i = 1:length(inputs)
    inputs{i} = {p0s(:, i), p1s(:, i)};
    s = slope(p0s(:, i), p1s(:, i));
    outputs{i} =  s;
end
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,[3] but it is quite easy to do in Python. I will use the pytest[4] package. To define a test, you must write a function whose name starts with `test_` in a file whose name also starts with `test_`. I will 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 the example, I start with importing the `testing` submodule of NumPy (line 1). It has helpful functions for testing the equality of floating point arrays. On line 2, I explicitly import the `loadmat` function, which I will use to read the mat files. On line 4, I import the first function that I want to test. Then comes the actual test function, which does not take any arguments. The following lines follow a pattern I will reuse. First I load the mat file, which returns a dictionary where the keys are the variable names (line 8). The `squeeze_me=True` argument to `loadmat` "squeezes out" unit dimensions. It should be used with caution, but it this case it is appropriate. On line 9 and 10 I extract the inputs and outputs. On line 11 I 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 I then unpack as two names, `inputs` and `outputs`. Finally, on line 12 I 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. It is as if I was calling the function with multiple arguments. It is very convenient here because I do not need to know ahead of time how many arguments `rotate` requires.

Now that I have one test, I must run it. I can do it within the Canopy Python prompt 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 I have not yet written the `rotate` function.

[3] The documentation can be found under the Testing Frameworks section.

[4] Holger Krekel and pytest-dev team (n.d.). *pytest: helps you write better programs*. URL: https://pytest.org

```
In [1]: !pytest
=============================== test session starts ================================
platform darwin -- Python 3.5.2, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
```

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

```
import numpy.testing as npt
from scipy.io import loadmat

from main import rotate


def test_rotate():
    data = loadmat('./rotate_data.mat', squeeze_me=True)
    test_inputs = data['inputs']
    test_outputs = data['outputs']
    for inputs, outputs in zip(test_inputs, test_outputs):
        npt.assert_allclose(rotate(*inputs), outputs)
```

```
rootdir: /Users/achabot/step1, inifile:
collected 0 items / 1 errors

====================================== ERRORS ======================================
__________________________ ERROR collecting test_main1.py __________________________
ImportError while importing test module '/Users/achabot/step1/test_main1.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
test_main1.py:4: in <module>
    from main import rotate
E   ImportError: cannot import name 'rotate'
!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 errors during collection !!!!!!!!!!!!!!!!!!!!!!
============================== 1 error in 0.97 seconds ==============================
```

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.

```
import numpy as np


def rotate(v, w):
    """Rotate 2D matrix v by angle w in radians."""
    R = np.array([[np.cos(w), -np.sin(w)],
                  [np.sin(w), np.cos(w)]])
    return R @ v
```

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

```
In [2]: !pytest
```

```
============================= test session starts ==============================
platform darwin -- Python 3.5.2, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
rootdir: /Users/achabot/step1, inifile:
collected 1 items

test_main.py .

=========================== 1 passed in 0.69 seconds ===========================
```

THE FIFTH STEP is to go through the same process for each other function. I wrote the next test, for `test_slope`. It has the same structure as `test_rotate`, except that I load data from a different file and call the `slope` function. I also added the import from the `main.py` module.

```
from main import slope

def test_slope():
    data = loadmat('./slope_data.mat', squeeze_me=True)
    test_inputs = data['inputs']
    test_outputs = data['outputs']
    for inputs, outputs in zip(test_inputs, test_outputs):
        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, I use zero-based indexing. Second, I use only one index within the square brackets (`p0[0]`) instead of an index and a slice like in MATLAB® (`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®. I *could* have ported the MATLAB® code more directly and used `p0[0, :]`, but I prefer the more compact version.

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

I run the tests once more to make sure everything passes:

```
In [2]: !pytest
============================= test session starts ==============================
platform darwin -- Python 3.6.0, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
rootdir: /Users/achabot/step2, inifile:
collected 2 items

test_main.py ..
=========================== 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. I will need to do some manual testing. The MATLAB® code of `main.m` produces Figure 4.1. I have written function `plot_vectors` in `main.py` (below) to produce a figure that is as close to the original as possible. Notice that I renamed the function to follow the Python standard of words_separated_by_underscores. I also specified the `before` argument with a default value of True, so I 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, I use the built-in function `enumerate` to automatically count how many vectors I have annotated. On lines 8 to 11, I use string formatting to append a quotation mark to "non-before" vector labels.

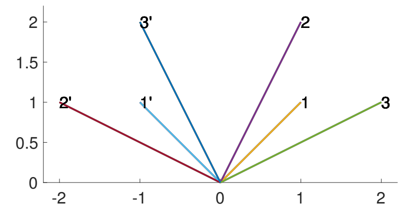

Figure 4.1: Figure generated when running `main.m`. The font size and line thickness have been adjusted to better render in this guide.

```
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)
```

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

```
before = np.array([[1, 1, 2],
                   [1, 2, 1]])
after = rotate(before, np.pi/2)

origin = np.array((0, 0))
before_slope = slope(origin, before)
after_slope = slope(origin, after)
print("Before rotation: {}".format(before_slope))
print("After rotation: {}".format(after_slope))

plot_vectors(before)
plot_vectors(after, before=False)
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 listings for the main code and the tests can be found in the appendix starting on page 47.

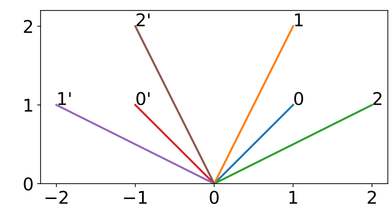

Figure 4.2: Figure generated when running `main.py`. The font size and line thickness have been adjusted to better render in this guide.

From the Top Down: Calling Python from MATLAB®

MATLAB® has built-int 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:

```
>> a = py.numpy.arange(12).reshape(3, 4)
>> a.max()

ans =
    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 1-D matrices to Python `array.array` and back, which is near useless. I wrote two functions, `mat2array` and `array2mat` to simplify that task (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 1-D NumPy arrays, and specifying a dtype. It should be reasonably efficient regarding its memory usage.

The second function, `array2mat` was a bit trickier to write. It required converting the NumPy array into a Python `array.array` type. For that, I needed the *typecode*, a one-character string representing the type of the array. This can conveniently be accessed directly on the NumPy array. The `array.array` type only accepts 1-D arrays, so I had to flatten the array before passing it to the constructor (line 22). Finally, the `array.array` must be converted to a MATLAB® matrix with the correct type. I am taking advantage of the fact that NumPy dtypes and MATLAB® integer functions have the same name to create the `typefunction` with `str2func` (line 19). Floating point arrays are automatically converted to double (line 14) and complex arrays are not supported, which is a limitation of `array.array`.

```
function a = mat2array(x, shape, dtype)
% MAT2ARRAY Convert MATLAB matrix to NumPy array.
%   A = MAT2ARRAY(X) converts the matrix X to an array of the
%   same shape (size) as X.
%
%   A = MAT2ARRAY(X,SHAPE) reshapes the resulting array to have the shape
%   SHAPE. The total number of elements cannot change. Useful when creating
%   1-D arrays.
%
%   A = MAT2ARRAY(X,SHAPE,TYPE) or MAT2ARRAY(X,[],TYPE) specifies the NumPy
%   dtype to use for the array. It will use NumPy's default if unspecified.
%
% See also ARRAY2MAT.

if nargin < 2 || isempty(shape)
    shape = size(x);
end
if nargin < 3
    dtype = py.None;
end

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.

```matlab
function m = array2mat(a)
% ARRAY2MAT(A) Convert NumPy array to MATLAB matrix.
%  M = ARRAY2MAT(A) converts the array according to its type.
%  Signed and unsigned integer arrays are converted using their
%  specified precision. Float types are converted to "double".
%  Complex arrays are not supported.
%
% See also MAT2ARRAY.

a = py.numpy.atleast_2d(a);
shape = cell2mat(cell(a.shape));
typename = char(a.dtype.name);

if startsWith(typename, 'float')
    typefunction = @double;
elseif startsWith(typename, 'complex')
    error('Complex types are not supported. The type was "%s."', ...
          char(a.dtype.name))
else
    typefunction = str2func(typename);
end

m = typefunction(py.array.array(a.dtype.char, a.ravel().tolist()));
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. Basically go through 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. Congrats, you've converted a MATLAB function to Python!
9. Repeat until you have converted everything!

Here is a simple example that should illustrate the process. I start with a function to multiply two things, called `multiplyby`.

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

I then create some test data. I am keeping it simple here but your coverage should be more exhaustive. You should also implement proper tests with error reporting on the MATLAB® side.

```
% Generate test data
x = [0, 1, -1, 0.5, nan];
ys = [0, 1, -1, 0.5, nan];

inputs = cell(size(x));
outputs = cell(size(ys));
for i = 1:length(x)
    inputs{i} = {x, ys(i)};
    out = multiplyby(x, ys(i));
    outputs{i} = out;
end
save('multiplyby_test_data.mat', 'inputs', 'outputs')
```

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

```python
import numpy.testing as npt
from scipy.io import loadmat

from main import multiply_by


def test_multiply_by():
    data = loadmat('multiplyby_test_data.mat', squeeze_me=True)
    test_inputs = data['inputs']
    test_outputs = data['outputs']
    for inputs, outputs in zip(test_inputs, test_outputs):
        npt.assert_allclose(multiply_by(*inputs), outputs)
```

I implement the Python function, which is straightforward. The tests pass, even though I do not show the output here.

```python
def multiply_by(x, y):
    return x * y
```

And finally, I replace the body of the MATLAB® `multiplyby` function with a call to the Python implementation. I also use the conversion functions to convert types between the two languages.

```matlab
function out = multiplyby(x, y)
x_array = mat2array(x);
out_array = py.main.multiply_by(x_array, y);
out = array2mat(out_array);
```

The two strategies presented here should get you started in converting some, if not all of your MATLAB® code to Python. There is a third strategy, which I did not mention. It is the opposite of the second strategy: write the "main" code in Python and call MATLAB® using the MATLAB® Engine API for Python. The limitations are similar as calling Python from MATLAB®. Only some data types are supported and the memory overhead can be significant. Once again, NumPy arrays are not supported. If you decide to take that route, I recommend having a look at this Stack Overflow answer by max9111 to the question "Improve performance of converting NumPy array to MATLAB double": https://stackoverflow.com/a/45290997/572506. It suggests a modification to the `matlab` package to reduce the overhead of passing NumPy arrays to MATLAB®. In my testing, it is about 15 times faster than the default code when using a 1-million element array.

5
What Next?

For people who want to speed up their transition and get proficient in Python as fast as possible, Enthought offers the "Python for Scientists & Engineers" course. It is five days of highly interactive training which will give you a rock-solid base to build high-quality software in terms of both readability and performance. The first day covers the fundamental data types as well as control flow and code organization. On the second day, we dive deeper into 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 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 wraps up on day five with a one-day module on rapid development of scientific Graphical User Interfaces (GUIs).

To learn more, contact us at 512.536.1057 or visit the course website at https://www.enthought.com/training/course/python-for-scientists-and-engineers/.

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.

```
from timeit import timeit

setup = """
import numpy as np
a = np.arange(100000000).reshape(10000, 10000)

def contiguous_sum(x):
    for i in range(x.shape[0]):
        x[i].sum()

def non_contiguous_sum(x):
    for i in range(x.shape[-1]):
        x[:, i].sum()
"""

n=100
time_contiguous = timeit('contiguous_sum(a)', setup=setup, number=n) / n
time_non_contiguous = timeit('non_contiguous_sum(a)', setup=setup, number=n) / n
print("Contiguous: {:.4f}s per loop".format(time_contiguous))
print("None Contiguous: {:.4f}s per loop".format(time_non_contiguous))
print("Ratio: {:.3f}".format(time_non_contiguous / time_contiguous))
```

Complete Version of `main.py`, in Chapter 4

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

```
import numpy as np
import matplotlib.pyplot as plt
```

```python


def rotate(v, w):
    """Rotate 2D matrix v by angle w in radians."""
    print(v, w)
    R = np.array([[np.cos(w), -np.sin(w)],
                  [np.sin(w), np.cos(w)]])
    return R @ v


def plot_vectors(v, before=True):
    """Plot vectors v from origin."""
    v_p = np.vstack((np.zeros_like(v), v))
    plt.plot(v_p[::2], v_p[1::2])
    for i, xy in enumerate(v.T):
        if before:
            label = '{}'.format(i)
        else:
            label= "{}'".format(i)
        plt.annotate(label, xy)


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


if __name__ == '__main__':
    before = np.array([[1, 1, 2],
                       [1, 2, 1]])
    after = rotate(before, np.pi/2)
    origin = np.array((0, 0))

    before_slope = slope(origin, before)
    after_slope = slope(origin, after)
    print("Before rotation: {}".format(before_slope))
    print("After rotation: {}".format(after_slope))
    plot_vectors(before)
    plot_vectors(after, before=False)
    plt.show()
```

This this the complete content of `test_main.py`.

```python
import numpy.testing as npt
from scipy.io import loadmat
```

```python

from main import rotate, slope


def test_rotate():
    data = loadmat('./rotate_data.mat', squeeze_me=True)
    test_inputs = data['inputs']
    test_outputs = data['outputs']
    for inputs, outputs in zip(test_inputs, test_outputs):
        npt.assert_allclose(rotate(*inputs), outputs)


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

Anti-Patterns

What I am calling here "anti-patterns" are a handful of "habits" that many MATLAB® developers have and take with them when writing Python code. Knowing about them is sure to quickly make your code look more *pythonic*.

1. Do not use the `pylab` mode, by which I mean running `from matplotlib.pylab import *`, or any other form of `from package import *`, where the star means "import everything". You will inevitably hear or read about them somewhere. `pylab` is a deprecated "feature" of matplotlib that would import all functions from NumPy and matplotlib, as well as many from SciPy into the current namespace. It *seems* to be convenient, and it replicated the MATLAB® experience, but this practice can actually cause very subtle bugs. Please do not do this.

2. Do not iterate over objects using an index, such as in this code snippet:

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

It is an anti-pattern. 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:

```python
words = ['quick', 'brown', 'fox']
for word in words:
```

```
    print(word)
```

3. Do not “clear all” or “close all” within your scripts. It is unnecessary. Scripts run in their own namespace when you run them in an IPython session.

References

Alted, 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.

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.

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/.