Why Python Rocks for Research

By HOYT KOEPKE

THE FOLLOWING IS an account of my own experience with Python. Because that experience was so positive, this is an unabashed attempt to promote the use of Python for general scientific research and development. About four years ago, I dropped MATLAB in favor of Python as my primary language for coding research projects. This article is a personal account of how rewarding I found that experience to be.

As I describe in the next sections, the variety and quality of Python's features have spoiled me. Even in small scripts, I now rely on Python's numerous data structures, classes, nested functions, iterators, the flexible function calling syntax, an extensive kitchen-sink-included standard library, great scientific libraries, and outstanding documentation.

To clarify, I am not advocating *just* Python as the perfect scientific programming environment; I am advocating Python plus a handful of mature 3rd-party open source libraries, namely Numpy/Scipy for numerical operations, Cython for low-level optimization, IPython for interactive work, and MatPlotLib for plotting. Later, I describe these and others in more detail, but I introduce these four here so I can weave discussion of them throughout this article.

Given these libraries, many features in MATLAB that enable one to quickly write code for machine learning and artificial intelligence – my primary area of research – are essentially a small subset of those found in Python. After a day learning Python, I was able to still use most of the matrix tricks I had learned in MATLAB, but also utilize more powerful data structures and design patterns when needed.

Holistic Language Design

I once believed that the perfect language for research was one that allowed concise and direct translation from notepad scriblings to code. On the surface, this is reasonable. The more barriers between generating ideas and trying them out, the slower research progresses. In other words, the less one has to think about the actual coding, the better. I now believe, however, that this attitude is misguided.

MATLAB's language design is focused on matrix and linear algebra operations; for turning such equations into one-liners, it is pretty much unsurpassed. However, move beyond these operations and it often becomes an exercise in frustration. R is beautiful for interactive data analysis, and its open library of statistical packages is amazing. However, the language design can be unnatural, and even maddening, for larger development projects. While Mathematica is perfect for interactive work with pure math, it is not intended for general purpose coding.

The problem with the "perfect match" approach is that you lose generalizability very quickly. When the criteria for language design is too narrow, you inevitably choose excellence for one application over greatness for many. This is why universities have graduate programs in computer language design — navigating the pros and cons of various design decisions is extremely difficult to get right. The extensive use of Python in everything from system administration and website design to numerical number-crunching shows that it has, indeed, hit the sweet spot. In fact, I've anecdotally observed that becoming better at R leads to skill at interacting with data, becoming better at MATLAB leads to skill at quick-and-dirty scripting, but becoming better at Python leads to genuine programming skill.

Practically, in my line of work, the downside is that some matrix operators that are expressable using syntactical constructs in MATLAB become function calls (e.g. `x = solve(A, y)` instead of `x = A \ y`). In exchange for this extra verbosity — which I have not found problematic — one gains incredible flexibility and a language that is natural for everything from automating system processes to scientific research. The coder doesn't have to switch to another language when writing non-scientific code, and allows one to easily leverage other libraries (e.g. databases) for scientific research.

Furthermore, Python allows one to easily leverage object oriented and functional design patterns. Just as different problems call for different ways of thinking, so also different problems call for different programming paradigms. There is no doubt that a linear, procedural style is natural for many scientific problems. However, an object oriented style that builds on classes having internal functionality and external behavior is a perfect design pattern for others. For this, classes in Python are full-featured and practical. Functional programming, which builds on the power of iterators and functions-as-variables, makes many programming solutions concise and intuitive. Brilliantly, in Python, everything can be passed around as an object, including functions, class definitions, and modules. Iterators are a key language component and Python comes with a full-featured iterator library. While it doesn't go as far in any of these categories as flagship paradigm languages such as Java or Haskell, it does allow one to use some very practical tools from these paradigms. These features combine to make the language very flexible for problem solving, one key reason for its popularity.

Readability

To reiterate a recurrent point, Python's syntax is *very* well thought out. Unlike many scripting languages (e.g. Perl), readability was a primary consideration when Python's syntax was designed. In fact, the joke is that turning pseudocode into correct Python code is a matter of correct indentation.

This readability has a number of beneficial effects. Guido van Rossum, Python's original author, writes:

> *This emphasis on readability is no accident. As an object-oriented language, Python aims to encourage the creation of reusable code. Even if we all wrote perfect documentation all of the time, code can hardly be considered reusable if it's not readable. Many of Python's features, in addition to its use of indentation, conspire to make Python code highly readable.*

In addition, I've found it encourages collaboration, and not just by lowering the barrier to contributing to an open source Python project. If you can easily discuss your code with others in your office, the result can be better code and better coders.

As two examples of this, consider the following code snippet:

```
def classify(values, boundary=0):
  "Classifies values as being below (False) or above (True)
a boundary."
  return [(True if v > boundary else False) for v in
values]

# Call the above function
classify(my_values, boundary=0.5)
```

Let me list three aspects of this code. First, it is a small, self-contained function that only requires three lines to define, including documentation (the string following the function). Second, a default argument for the boundary is specified in a way that is instantly readable (and yes, that does show up when using Sphinx for automatic documentation). Third, the list processing syntax is designed to be readable. Even if you are not used to reading Python code, it is easy to parse this code — a new list is defined and returned from the list `values` using `True` if a particular value `v` is above `boundary` and `False` otherwise. Finally, when calling functions, Python allows named arguments — this universally promotes clarity and reduces stupid bookkeeping bugs, particularly with functions requiring more than one or two arguments.

Permit me to contrast these features with MATLAB. With MATLAB, globally available functions are put in separate files, discouraging the use of smaller functions and — in practice — often promotes cut-and-paste programming, the bane of debugging. Default arguments are a pain, requiring conditional coding to set unspecified arguments. Finally, specifying arguments by name when calling is not an option, though one popular but artificial construct — alternating names and values in an argument list — allows this to some extent.

Balance of High Level and Low Level Programming

The ease of balancing high-level programming with low-level optimization is a particular strong point of Python code. Python code is meant to be as high level as reasonable — I've heard that in writing similar algorithms, on average you would write six lines of C/C++ code for every line of Python. However, as with most high-level languages, you often sacrifice code speed for programming speed.

One sensible approach around this is to deal with higher level objects — such as matrices and arrays — and optimize operations on these objects to make the program acceptably fast. This is MATLAB's approach and is one of the keys to its success; it is also natural with Python. In this context, speeding code up means vectorizing your algorithm to work with arrays of numbers instead of with single numbers, thus reducing the overhead of the language when array operations are optimized.

Abstractions such as these are absolutely essential for good scientific coding. Focusing on higher-level operations over higher-level data types generally leads to massive gains in coding speed and coding accuracy. Python's extension type system seamlessly allows libraries to be designed around this idea. Numpy's array type is a great example.

However, existing abstractions are not always enough when you're developing new algorithms or coding up new ideas. For example, vectorizing code through the use of arrays is powerful but limited. In many cases, operations really need loops, recursion, or other coding structures that are extremely efficient in optimized, compiled machine code but are not in most interpreted languages. As variables in many interpreted languages are not statically typed, the code can't easily be compiled into optimized machine code. In the scientific context, Cython provides the perfect balance between the two by allowing either.

Cython works by first translating Python code into equivalent C code that runs the Python interpreted through the Python C API. It then uses a C compiler to create a shared library that can be loaded as a Python module. Generally, this module is functionally equivalent to the original Python module and usually runs marginally faster. The advantage, however, is that Cython allows one to statically type variables — e.g. `cdef int i` declares `i` to be an integer. This gives massive speedups, as typed variables are now treated using low-level types rather than Python variables. With these annotations, your "Python" code can be as fast as C — while requiring very little actual knowledge of C.

Practically, a few type declarations can give you incredible speedups. For example, suppose you have the following code:

```
def foo(A):
  for i in range(A.shape[0]):
    for j in range(A.shape[1]):
      A[i,j] += i*j
```

where `A` is a 2d NumPy array. This code uses interpreted loops and thus runs fairly slowly. However, add type information and use Cython:

```
def cyfoo(ndarray[double, ndim=2] A):
  cdef size_t i, j

  for i in range(A.shape[0]):
    for j in range(A.shape[1]):
      A[i,j] += i*j
```

Cython translates necessary Python operations into calls to the Python C-API, but the looping and array indexing operations are turned into low level C code. For a 1000 x 1000 array, on my 2.4 GHz laptop, the Python version takes 1.67 seconds, while the Cython version takes only 3.67 milliseconds (a vectorized version of the above using an outer product took 15.1 ms).

A general rule of thumb is that your program spends 80% of its time running 20% of the code. Thus a good strategy for efficient coding is to write everything, profile your code, and optimize the parts that need it. Python's profilers are great, and Cython allows you to do the latter step with *minimal* effort.

Language Interoperability

As a side affect of its universality, Python excels at gluing other languages together. One can call MATLAB functions from Python (through the MATLAB engine) using MLabWrap, easing transitions from MATLAB to Python. Need to use that linear regression package in R? RPy puts it at your fingertips. Have fast FORTRAN code for a particular numerical algorithm? F2py will effortless generate a wrapper. Have general C or C++ libraries you want to call? Ctypes, Cython, or SWIG are three ways to easily interface to it (my favorite is Cython). Now, if only all these were two way streets...

Documentation System

Brilliantly, Python incorporates module, class, function, and method documentation directly into the language itself. In essence, there are two levels of comments — programming level comments (start with #) that are ignored by the compiler, and documentation comments that are specified by a doc string after the function or method name. These documentation strings add tags to the methods which are accessible by anyone using an interactive Python shell or by automatic documentation generators.

The beauty of Python's system becomes apparent when using Sphinx, a documentation generation system originally built for Python language documentation. To allow sufficient presentation flexibility, it allows reStructuredText directives, a simple, readable markup language that is becoming widely used in code documentation. Sphinx works easily with embedded doc-strings, but it is useful beyond documentation — for example, my personal website, my course webpages when I teach, my code documentation sites, and, of course, Python's main website are generated using Sphinx.

One helpful feature for scientific programming is the ability to put LaTeX equations and plots directly in code documentation. For example, if you write:

```
.. math:: \Gamma(z) = \int_0^\infty x^{z-1}e^x\,dx
```

in the doc string, it is rendered in the webpage as

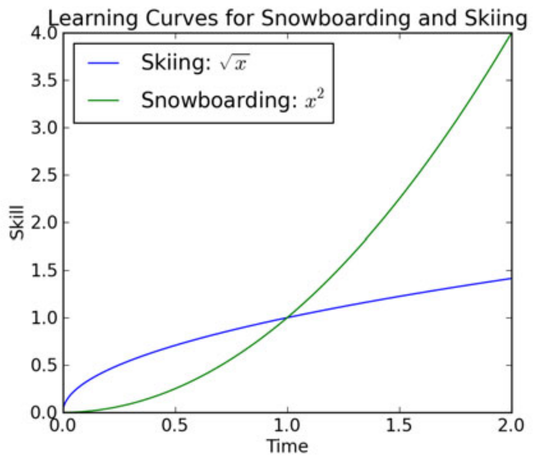
$$\Gamma(z) = \int_0^\infty x^{z-1}e^x\,dx$$

Including plots is easy. The following doc-string code:

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

  x = np.linspace(0,2, 1000)
  plt.figure() plt.plot(x, np.sqrt(x), label = r"Skiing: $\
  sqrt{x}$")
  plt.plot(x, x**2, label = r"Snowboarding: $x^2$")
  plt.title("Learning Curves for Snowboarding and Skiing")
  plt.xlabel("Time") ;
  plt.ylabel("Skill") plt.legend(loc='upper left')
  plt.show()
```

gives



In essence, this enables not only comments about the code, but also comments about the science and research behind your code, to be interwoven into the coding file.

Hierarchical Module System

Python uses modular programming, a popular system that naturally organizes functions and classes into hierarchical namespaces. Each Python file defines a module. Classes, functions, or variables that are defined in or imported into that file show up in that module's namespace. Importing a module either creates a local dictionary holding that module's objects, pulls some of the module's objects into the local namespace. For example, `import hashlib` binds `hashlib.md5` to hashlib's md5 checksum function; alternately, `from hashlib import md5` binds `md5` to this function. This helps programming namespaces to follow a hierarchical organization.

On the coding end, a Python file defines a module. Similarly, a directory containing an `__init__.py` Python file is treated the same way, files in that directory can define submodules, and so on. Thus the code is arranged in a hierarchical structure for both the programmer and the user.

Permit me a short rant about MATLAB to help illustrate why this is a great feature. In MATLAB, all functions are declared in the global namespace, with names determined by filenames in the current path variable. However, this discourages code reusability by making the programmer do extra work keeping disparate program components separate. In other words, without a hierarchical structure to the program, it's difficult to extract and reuse specific functionality. Second, programmers must either give their functions long names, essentially doing what a decent hierarchical system inherently does, or risk namespace conflicts which can be difficult to resolve and result in subtle errors. While this may help one to throw something together quickly, it is a horrible system from a programming language perspective.

Data Structures

Good programming requires having and using the correct data structures for your algorithm. This is almost universally underemphasized in research-oriented coding. While proof-of-concept code often doesn't need optimal data structures, such code causes problems when used in production. This often — though it's rarely stated or known explicitly — limits the scalability of a lot of existing code. Furthermore, when such features are not natural in a language's design, coders often avoid them and fail to learn and use good design patterns.

Python has lists, tuples, sets, dictionaries, strings, thread-safe queues, and many other types built-in. Lists hold arbitrary data objects and can be sliced, indexed, joined, split, and used as stacks. Sets hold unordered, unique items. Dictionaries map from a unique key to anything and form the real basis of the language. Heaps are available as operations on top of lists (similar to the C++ STL heaps). Add in NumPy, and one has an n-dimensional array structure that supports optimized and flexible broadcasting and matrix operations. Add in SciPy, and you have sparse matrices, kd-trees, image objects, time-series, and more.

Available Libraries

Python has an *impressive* standard library packaged with the program. Its philosophy is "batteries-included", and a standard Python distribution comes with built-in database functionality, a variety of data persistence features, routines for interfacing with the operating system, website interfacing, email and networking tools, data compression support, cryptography, xml support, regular expressions, unit testing, multithreading, and much more. In short, if I want to take a break from writing a bunch of matrix manipulation code and automate an operating system task, I don't have to switch languages.

Numerous libraries provide the needed functionality for scientific . The following is a list of the ones I use regularly and find to be well-tested and mature:

- **NumPy/SciPy:** This pair of libraries provide array and matrix structures, linear algebra routines, numerical optimization, random number generation, statistics routines, differential equation modeling, Fourier transforms and signal processing, image processing, sparse and masked arrays, spatial computation, and numerous other mathematical routines. Together, they cover most of MATLAB's basic functionality and parts of many of the toolkits, and include support for reading and writing MATLAB files. Additionally, they now have great documentation (vastly improved from a few years ago) and a very active community.
- **IPython:** One of the best things in Python is IPython, an enhanced interactive Python shell that makes debugging, profiling code, interactive plotting. It supports tab completion on objects, integrated debugging, module finding, and more — essentially, it does almost everything you'd expect a command line programming interface to do. Additionally,
- **Cython:** Referenced earlier, Cython is a painless way of embedding compiled, optimized bits of code in a larger Python program.
- **SQLAlchemy:** SQLAlchemy makes leveraging the power of a database incredibly simple and intuitive. It is essentially a wrapper around an SQL database. You build queries using intuitive operators, then it generates the SQL, queries the database, and returns an iterator over the results. Combining it with sqlite — embedded in Python's standard library — allows one to leverage databases for scientific work with impressive ease. And, if you tell sqlite to build its database in memory, you've got another powerful data structure. To slightly plagiarize xkcd, SQLAlchemy makes databases fun again.
- **PyTables:** PyTables is a great way of managing large amounts of data in an organized, reliable, and efficient fashion. It optimizes resources, automatically transferring data between disk and memory as needed. It also supports on-the-fly (DE)compression and works seamlessly with NumPy arrays.
- **PyQt:** For writing user interfaces in C++, I recommend it is, in my experience, difficult to beat QT. PyQt brings the ease of QT to Python. And I do mean ease — using the interactive QT designer, I've build a reasonably complex GUI-driven scientific application with only a few dozen lines of custom GUI code. The

entire thing was done in a few days. The code is cross-platform over Linux, Mac OS X, and Windows. If you need to develop a front end to your data framework, and don't mind the license (GPL for PyQT, LGPL for QT), this is, in my experience, the easiest way to do so.

- **TreeDict:** Without proper foresight and planning, larger research projects are particularly prone to the second law of thermodynamics: over time, the organization of parameters, options, data, and results becomes increasingly random. TreeDict is a Python data structure I designed to fight this. It stores hierarchical collections of parameters, variables, or data, and supports splicing, joining, copying, hashing, and other operations over tree structures. The hierarchical structure promotes organization that naturally tracks the conceptual divisions in the program — for example, a single file can define all parameters while reflecting the structure of the rest of the code.
- **Sage:** Sage doesn't really fit on this list as it packages many of the above packages into a single framework for mathematical research. It aims to be a complete solution to scientific programming, and it incorporates over a hundred open source scientific libraries. It builds on these with a great notebook concept that can really streamline the thought process and help organize general research. As an added bonus, it has an online interface for trying it out. As a complete package, I recommend newcomers to scientific Python programming try Sage first; it does a great job of unifying available tools in a consistent presentation.
- **Enthought Python Distribution:** Also packaging these many libraries into a complete package for scientific computing, the Enthought Python Distribution is distributed by a company that contributes heavily to developing and maintaining these libraries. While there are commercial support options, it is free for academic use.

Testing Framework

I do not feel comfortable releasing code without an accompanying suite of tests. This attitude, of course, reflects practical programmer wisdom; code that is guaranteed to function a certain way — as encapsulated in these unit tests — is reusable and dependable. While packaging test code without does not always equate with code quality, there is a strong correlation. Unfortunately, the research community does not often emphasize writing proper test code, due partly to that emphasis being directed, understandably, towards technique, theory, and publication. But this is exactly why a no-boilerplate, practical and solid testing framework and simple testing constructs like assert statements are so important. Python provides a built-in, low barrier-to-entry testing framework that encourages good test coverage by making the fastest workflow, including debugging time, involve writing test cases. In this way, Python again distinguishes itself from its competitors for scientific code.

Downsides

No persuasive essay is complete without an honest presentation of the counterpoints, and indeed several can be made here. In fact, many of my arguments invite a counterargument — with so many options available at every corner, where does one start? Having to make decisions at each turn could paralyze productivity. For most applications, wouldn't a language with a rigid but usually adequate style — like MATLAB — be better?

While one can certainly use a no-flair scripting style in Python, I agree with this argument, at least to a certain extent. However, the situation is not uniformly bad — rather, it's a bit like learning to ski versus learning to snowboard. The first day or two learning to snowboard is always horrid, while one can pick up basic skiing quite quickly. However, fast-forward a few weeks, and while the snowboarder is perfecting impressive tricks, the skier is still working on not doing the splits. An exaggerated analogy, perhaps, but the principle still holds: investment in Python yields impressive rewards, but be prepared for a small investment in learning to leverage its power.

The other downside with using Python for general scientific coding is the current landscape of conventions and available resources. Since MATLAB is so common in many fields, it is often conventional to publish open research code in MATLAB (except in some areas of mathematics, where Python is more common on account of Sage; or in statistics, where R is the lingua franca). While MLabWrap makes this fairly workable, it does means that a Python programmer may need to work with both languages and possess a MATLAB license. Anyone considering a switch should be aware of this potential inconvenience; however, there seems to be a strong movement within scientific research towards Python — largely for the reasons outlined here.

A Complete Programming Solution

In summary, and reiterating my point that Python is a complete programming solution, I mention three additional points, each of which would make a great final thought. First, it is open source and completely free, even for commercial use, as are many of the key scientific libraries. Second, it runs natively on Windows, Mac OS, linux, and others, as does its standard library and the third party libraries I've mentioned here. Third, it fits quick scripting and large development projects equally well. A quick perusal of some success stories on Python's website showcases the diversity of environments in which Python provides a scalable, well-supported, and complete programming solution for research and scientific coding. However, the best closing thought is due to Randall Monroe, the author of xkcd: "Programming is fun again!" ■

Hoyt Koepke is a PhD student in the Statistics Department at the University of Washington studying optimization, ranking models, probability theory, and machine learning/artificial intelligence. As a teen, he learned to program when his parents would only let him play computer games he wrote himself, and subsequently got a MSc in computer science from the University of British Columbia following a BA in physics at the University of Colorado. He can be contacted at *hoytak@stat.washington.edu* or visited online at *www.stat.washington.edu/~hoytak*.