Tutorial: Learning Python & Julia
(A User and Developer View)

Douglas D. Hodson, PhD
Air Force Institute of Technology
937-255-3636 x4719
About Me

• Education & experience
  – Bachelor's in Physics, Master’s in Electro-Optics, MBA, PhD Computer Engineering
  – Experience in the domain of Modeling and Simulation
    • Real-time & non-real-time
  – Experience in computer programming (strong C++ orientation)

• Teaching
  – Software engineering sequence
  – Experimental design - statistics as applied to experimentation

• Research interests
  – Modeling and Simulation
    • Real-time / highly interactive (human and/or hardware in the loop)
    • Non-real-time, so-called constructive or analytic simulation
    • Infrastructure concerns (e.g., architectures, frameworks)
  – Data processing & analytics
    • The analysis process or workflow from collected raw data to information
    • The processing of such data (languages, infrastructure, etc.)
Disclaimers

• There is no way the full breath of either Python and/or Julia ecosystems can be presented in 1.5 hours
  – But I can present a few things associated with data science and scientific computing

• Personal biases
  – Not a fan of the Python language, but impressed by extension packages
    • View it as a modern version of BASIC
    • The lack of annotated types drives me crazy
    • Believe it stumbled into the domains of data science and scientific computing mostly due to extension packages (e.g., NumPy, Matplotlib)
    • View the package management system a mess
  – Really like Julia
    • Even though I’m a somewhat diehard C++/statically-typed language proponent, the design of Julia impresses
General: Python

• Often discussed/presented from different perspectives
  – Language
    • Conceived in late 1980s as a teaching and scripting language
    • Python 2.x: 2000
    • **Python 3.x**: 2008
  – Interactive application (i.e., REPL)
    • E.g., idle, IPython
  – Interpreter/execution system
    • E.g., CPython, Numba, PyPy, IronPython
  – Distributions
    • E.g., Anaconda Python, Enthought Python, Python(x,y)
    • Basically a bundle that includes Python interpreter plus libraries
      – NumPy, SciPy, Matplotlib, pandas, etc...
General: Julia

- **Language**
  - First introduced 2012 as a language designed for technical computing
  - Julia 1.0 : 2018
- **Modern conveniences tend to be “baked in”**
  - Optimized code execution due to JIT compiler
  - Good default interactive REPL
  - Built-in package management
- **Community less fragmented**
  - One language, not two (i.e., Python 2 vs 3)
  - Tends to be more “technically” minded
    - Discussions tend towards technical and bleeding edge concepts (e.g., multiple dispatch, missing value representation)
Language in 15 Minutes

• Learn X in Y minutes
  – For the programmer (https://learnxinyminutes.com/)
  – Whirlwind tour of algorithms, data structures, languages and tools

• Python 3.x

• Julia 1.x
  – https://learnxinyminutes.com/docs/julia/
Language Features: Python 3

• Native datatypes
  – Booleans: either True or False
  – Numbers: integers (e.g., 2), floats (e.g., 1.1), fractions (e.g., ½), complex numbers (e.g., 2+3j)
  – Strings: sequences of Unicode characters
  – Bytes and byte arrays (e.g., a jpeg image)
  – Lists: order sequences of values
  – Tuples: ordered, immutable sequences of values
  – Sets: unordered bags of values
  – Dictionaries: unordered bags of key-value pairs
  – Everything is an object

• Additional types
  – module, function, class, method, file and even compiled code

• No such thing as constants (i.e., const)
Language Features: Julia

- Native datatypes
  - Rich hierarchies (example above)
  - **Arrays!!!**
    - Tuples
      - Ordered sequence of elements, like an array
      - Elements may be named, thus, a dictionary
- Has constants (i.e., “const”)
- In general, Julia positions itself as a language for numerical and scientific computing
Computation Strategies

• Ahead-of-Time (AOT) compilation
  – Compile code before executed
  – Example: C++

• Just-in-Time (JIT) compilation
  – Compile as executed
  – Usually compiled to an intermediate form, then executed (looks a bit like AOT combined with interpreter, but usually much smarter)
  – Example: Julia

• Interpreted
  – Code interpreted (line by line) and executed by the interpreter application
  – Example: Python – or more specifically, “CPython” - the C language interpreter reference implementation
Development Paradigms

• Explicit compilation followed by execution
  – An IDE facilitates your development by helping you edit, compile, link and debug your code
    • The end product is your final executable is often executed “outside” of that tool
  – Examples: C++, Rust, Go, FORTRAN, etc...

• REPL (Read-Evaluate-Print Loop)-based
  – Often for interpreted languages, the interpreter itself might provide conveniences (much like an IDE) to support development, but it also usually required to execute your code (as it provides the “runtime” system)
  – Can be an attractive way to develop code because of its interactive nature (immediate feedback) and can avoid sometimes annoying tedious issues associated with compiling, linking and possibly interfacing to third party packages/libraries
  – Third party packages or external libraries are often conveniently included as part of a larger so-called “distribution” (e.g., Anaconda Python)
  – Examples: Python, Julia, R, Perl, Tcl, etc...
Foundations
(High Performance Arrays of Data for Data Science & Computational Science)

Battle of the Venn Diagrams!
Static vs Dynamic Typing

- The execution performance/flexibility tradeoff
  - In a typical statically-typed language (e.g., “C”) variables of a particular type point to data
  - In a typical dynamically-typed language (e.g., “Python”) variables point to a structure that contains type information and data
By default, Python’s type system is dynamic (i.e., flexible, but inefficient)
NumPy introduces a whole new set of types (and functions) constrains this flexibility to significantly improve performance
  – NumPy-based packages build upon this
When this approach is used, Python is often viewed as a *glue* language
- A “user interface” of sorts
Modern languages (such as Julia) resolve this issue by turning functions and methods (the verbs of language), into first class citizens. As first class citizens, they are defined to “behave” the same way (just with different types of data). Notice how the verb “mean” was not escorted by a noun.
Python vs Julia

Types that are being provided by NumPy package resemble the default types already present in Julia

(In other words, Julia is already designed with this in mind)
Foundations
(Mathematics for Data Science & Computational Science)

(Consider Linear Algebra)
BLAS/LAPACK

• **BLAS (Basic Linear Algebra Subprograms)**
  – A collection of routines that provide standard building blocks for performing basic vector and matrix operations
  – Originated as FORTRAN library in 1979

• **LAPACK (Linear Algebra PACKage)**
  – A collection of routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems
  – LINPACK written in FORTRAN 77, developed in 1970s and 1980s
  – LAPACK now written in FORTRAN 90
BLAS/LAPACK Variations

• OpenBLAS
  – Implementation of BLAS library with many hand-crafted optimizations for specific processor types

• ATLAS (Automatically Tuned Linear Algebra Software)
  – Applies empirical techniques in order to provide portable performance

• MKL (Intel Math Kernel Library)
  – Routines that are hand-optimized specifically for Intel processors
  – Launched on 9 May 2003
  – Available for Windows, Linux and macOS
BLAS/LAPACK

• Many standard **vectorized** computations are calling subroutines from packages known as BLAS and LAPACK

• They can be called natively or “wrapped” so they can be called from languages such as MATLAB, Octave, Python (via NumPy), Julia, etc.
Query Configuration

```
>> version -blas
ans =
    'Intel(R) Math Kernel Library Version 2017.0.31 Product Build 20170606 for Intel(R) 64 architecture applications, CQR branch AVX
',

>> version -lapack
ans =
    'Intel(R) Math Kernel Library Version 2017.0.31 Product Build 20170606 for Intel(R) 64 architecture applications, CQR branch AVX
    Linear Algebra PACKage Version 3.7.0
',
```

```
octave:1> version -blas
ans = OpenBLAS (config: NO_LAPACK NO_LAPACKE DYNAMIC_ARCH NO_AFFINITY Sandybridge MAX_THREADS=8)
octave:2> version -lapack
ans = Linear Algebra PACKage Version 3.8.0
```

Octave 5.1.0  MATLAB 2018a  MKL
Octave Installation

```
octave:1> version -blas
ans = OpenBLAS (config: NO_LAPACK NO_LAPACKE DYNAMIC_ARCH NO_AFFINITY Sandybridge MAX_THREADS=8)
octave:2> version -lapack
ans = Linear Algebra PACKAGE Version 3.8.0
octave:3>
```
Query Configuration

BLAS and LAPACK are NOT part of Python, they are part of NumPy

```python
blas_info:
    libraries = ['blas']
    library_dirs = ['/usr/lib']
    language = f77
lapack_info:
    libraries = ['lapack']
    library_dirs = ['/usr/lib']
    language = f77
atlas_threads_info:
    NOT AVAILABLE
blas_opt_info:
    libraries = ['blas']
    library_dirs = ['/usr/lib']
    language = f77
    define_macros = [('', 1)]
atlas_blas_threads_info:
    NOT AVAILABLE
openblas_info:
    NOT AVAILABLE
lapack_opt_info:
    libraries = ['lapack', 'blas']
    library_dirs = ['/usr/lib']
    language = f77
    define_macros = [('', 1)]
```

Linux Platform

```python
mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/apps/anaconda3/Library/lib']
    define_macros = ...
    include_dirs = ...
blas_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/apps/anaconda3/Library/lib']
    define_macros = ...
    include_dirs = ...
blas_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/apps/anaconda3/Library/lib']
    define_macros = ...
    include_dirs = ...
lapack_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/apps/anaconda3/Library/lib']
    define_macros = ...
    include_dirs = ...
lapack_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/apps/anaconda3/Library/lib']
    define_macros = ...
    include_dirs = ...
```

Windows Platform

Python - numpy.show_config()
Query Configuration

BLAS and LAPACK are included as part of Julia’s standard library

```
julia> versioninfo()
Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, sandybridge)
```

```
julia> LinearAlgebra.versioninfo()
BLAS: libopenblas (USE64BITINT DYNAMIC_ARCH NO_AFFINITY Sandybridge MAX_THREADS=16)
LAPACK: libopenblas64_`

Julia 1.1.0
Some Benchmarks
(Execution Performance)
Benchmark: Recursive Fibonacci

- Benchmark is used to compare relative performance of a statically compiled language (C++) to an interpreted (Python) and a JIT compiled one (Julia).

- Benchmark not written to achieve maximal performance
  - Written to test the performance of identical algorithms and code pattern in each language.
Benchmark: Recursive Fibonacci

C++ Implementation

```
// compile program
// execute program

#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int fib(int n)
{
  if (n<=1) return n;
  return fib(n-1)+fib(n-2);
}

int main()
{
  high_resolution_clock::time_point t1{high_resolution_clock::now());
  const int x{fib(40)};
  high_resolution_clock::time_point t2{high_resolution_clock::now());
  const auto elapsed_time = duration_cast<milliseconds>(t2-t1).count();
  cout << "x : " << x << endl;
  cout << "elapsed_time : " << elapsed_time << " millisecs" << endl;
}
```

Calculate the 40th value in sequence and print out result and how much time it took to compute

Elapsed time: 1.6 secs

Note the requirement to explicitly state data types being used... (i.e., statically compiled)
Benchmark: Recursive Fibonacci

Note: Same algorithm and code pattern – no attempt to optimize

Julia 1.1.0
Implementation

Calculate the 40\textsuperscript{th} value in sequence and print out result and how much time it took to compute

Elapsed time: \textbf{4.9 secs}

Python 3.7.1
Implementation

Calculate the 40\textsuperscript{th} value in sequence and print out result and how much time it took to compute

Elapsed time: \textbf{129.2 secs}
Quick (Very Unscientific) Results

<table>
<thead>
<tr>
<th>Language</th>
<th>Elapsed Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ (C++17) optimized</td>
<td>1.6</td>
</tr>
<tr>
<td>Julia (1.1.0)</td>
<td>4.9</td>
</tr>
<tr>
<td>Python (3.7.1)</td>
<td>129.2</td>
</tr>
</tbody>
</table>

Relative performance Python/C++ : \( \frac{129.2}{1.6} = 80.8 \) (base 10 log difference ≈ 1.9 orders of magnitude slower)

(Caveats: not considering memory usage, JIT optimizations, only one run - no replications, runtime environment was the same, how elapsed time was ad-hoc, etc.)

(The “effect size” was LARGE compared to variation - an indication that a difference exists)
Published Benchmarks

Recursion Fibonacci

My estimate ($\approx 0.5$)

My estimate ($\approx 1.9$)
Published Benchmarks

Matrix Multiply
(Benchmarks written to use BLAS, if “conveniently” available)
Published Benchmarks (Reference)

The benchmark data shown above were computed with Julia v1.0.0, SciLua v1.0.0-b12, Rust 1.27.0, Go 1.9, Java 1.8.0_17, Javascript V8 6.2.414.54, Matlab R2018a, Anaconda Python 3.6.3, R 3.5.0, and Octave 4.2.2. C and Fortran are compiled with gcc 7.3.1, taking the best timing from all optimization levels (-O0 through -O3). C, Fortran, Go, Julia, Lua, Python, and Octave use OpenBLAS v0.2.20 for matrix operations; Mathematica uses Intel(R) MKL. The Python implementations of matrix_statistics and matrix_multiply use NumPy v1.14.0 and OpenBLAS v0.2.20 functions; the rest are pure Python implementations. Raw benchmark numbers in CSV format are available here and the benchmark source code for each language can be found in the perf. files listed here. The plot is generated using this Julia benchmarks notebook.

These micro-benchmark results were obtained on a single core (serial execution) on an Intel(R) Core(TM) i7-3960X 3.30GHz CPU with 64GB of 1600MHz DDR3 RAM, running openSUSE LEAP 15.0 Linux.

Link: https://julialang.org/benchmarks
Published Benchmarks

Python roughly 1-2 orders of magnitude slower than C/C++ and/or Julia
Desire (SW Engineering Perspective)

• Write code in a language that is accessible to the part-time or casual programmer (maybe beginner), yet retain the performance of statically typed, compiled languages
  – Or, more specifically, make it easy for “scientists and engineers to build extensible scientific software without having to get a degree in software engineering” (ref: SWIG manual)

• The challenge
  – The explicit specification of types to perform a specific computational tasks can be off-putting to the casual programmer
  – But, the explicit specification of types provides strong clues to a compiler so the code to be executed can be optimized for runtime performance
    • The role of data types is a big topic in computer science!
  – A double edged sword of sorts – optional typing?
A Typical Solution

• To address this challenge, many programmers view the solution in terms of using different languages for different tasks (perfectly reasonable)
  – This is often called “The Two Language Problem”
  – Prototype in one (e.g., Python), develop final product (i.e., production code) in another (e.g., C++)
    • Two different kinds of people (users & developers)

• Wrapping code
  – The “connecting” the two languages – sometimes called an integration or wrapping of one language (e.g., C/C++) to be used or called from another (e.g., Python)
  – Lots and lots of wrapper tools exist!
  – Concept is to reengineer or refactor performance-oriented software into accessible scripting language components (i.e., modules, packages, libraries, etc....)
    • NumPy is a great example (C & FORTRAN code “wrapped” so that it can be accessed via Python)
Illustrating “Wrapping” Approach

This function executes very slowly when written in Python!

```
int fib(int n)
{
    if (n<=1) return n;
    return fib(n-1)+fib(n-2);
}
```

Function to “Wrap”

```
swig -python fib.i
gcc -c -O3 -fpic ..;/fib.c fib_wrap.c -I/home/me/apps/anaconda3/include/python3.7m
gcc -shared -O3 fib.o fib_wrap.o -o _fib.so
```

Steps to compile C function to Make Accessible to Python (SWIG)

```
import timeit
from fib import *

t1 = timeit.default_timer()
x = fib(40)
t2 = timeit.default_timer()
elapsed_time = t2 - t1
print(x)
print(elapsed_time)
```

Python 3.7.1 Implementation

Calculate the 40th value in sequence and print out result and how much time it took to compute (Python now accessing “wrapped” optimized C code)

Elapsed time: 3.7 secs (≈ 1.5 orders of magnitude +)
Lots of Approaches for Python

- **Cython**
  - Add type annotations to existing python code; transpile to “C” code; compile it to a module; use

- **Numba**
  - Compile Python code to machine code

- **PyPy**
  - Replace CPython interpreter (written in C) with a Just-in-Time-based compiler

- IMO – because native Python code executes slowly, maybe it’s best to view a Python distribution as a collection of tools & libraries “glued” together and accessed through this language
  - From this perspective, the Python language & interpreter is an interactive, programmable (i.e., flexible) user interface
A Different Approach

- **Julia**
  - Multiple approaches “baked in”
  - Optional type annotation
    - Optional typing is part of the language without the ceremony of creating new modules
    - Similar to Cython
  - Backend is a Just-in-Time (JIT) compiler
    - Based on LLVM (modern compiler infrastructure)
    - Similar to PyPy
  - Users can become package developers, package developers are users
Gap Between Users & Developers

• Python
  – Users become consumers of well written packages (e.g., NumPy, Matplotlib) often written in other statically compiled languages (e.g., C, C++, FORTRAN) by experienced developers
    • Large gap – developing external Python packages is no small task
    • Python serves as “glue”

  ![User](User) ![Developer](Developer)

• Julia
  – Most packages in Julia are written in Julia
    • Smaller gap - users can become developers, developers are users

  ![User](User) ![Developer](Developer)
Performance & Optimization of Execution Time is Not Always Everything!

(How about productivity as measured from a different perspective?)
Ecosystem Capabilities
(Packages, Libraries, Modules, etc.)
Major Packages: Python

SciPy (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:

- **NumPy**: Base N-dimensional array package
- **SciPy library**: Fundamental library for scientific computing
- **Matplotlib**: Comprehensive 2D plotting
- **IPython**: Enhanced Interactive Console
- **Sympy**: Symbolic mathematics
- **pandas**: Data structures & analysis
### Ecosystem

#### Data Visualization and Plotting

Data visualization has a complicated history. Plotting software makes trade-offs between features and simplicity, speed and beauty, and a static and dynamic interface. Some packages make a display and never change it, while others make updates in real-time.

**Plots.jl** is a visualization interface and toolset. It provides a common API across various backends, like GR.jl, PyPlot.jl, and PlotlyJS.jl. Users who prefer a more grammar of graphics style API might like the pure Julia Gadfly.jl plotting package. VegaLite.jl provides the Vega-Lite grammar of interactive graphics interface as a Julia package. For those who do not wish to leave the comfort of the terminal, there is also UnicodePlots.jl.
Major Packages: Julia

Interact with your Data

The Julia data ecosystem lets you load multidimensional datasets quickly, perform aggregations, joins and preprocessing operations in parallel, and save them to disk in efficient formats. You can also perform online computations on streaming data with OnlineStats.jl. Whether you're looking for the convenient and familiar DataFrames, or a new approach with JuliaDB, Julia provides you a rich variety of tools. The Queryverse provides query, file IO and visualization functionality. In addition to working with tabular data, the JuliaGraphs packages make it easy to work with combinatorial data.

Julia can work with almost all databases using JDBC.jl and ODBC.jl drivers. In addition, it also integrates with the Hadoop ecosystem using Spark.jl, HDFS.jl, and Hive.jl.
**Major Packages: Julia**

### Rich Ecosystem for Scientific Computing

Julia is designed from the ground up to be very good at numerical and scientific computing. This can be seen in the abundance of scientific tooling written in Julia, such as the state-of-the-art differential equations ecosystem (DifferentialEquations.jl), optimization tools (JuMP.jl and Optim.jl), iterative linear solvers (IterativeSolvers.jl), a robust framework for Fourier transforms (AbstractFFTs.jl), a general purpose quantum simulation framework (Yao.jl), and many more, that can drive all your simulations.

Julia also offers a number of domain-specific ecosystems, such as in biology (BioJulia), operations research (JuliaOpt), image processing (JuliaImages), quantum physics (QuantumBFS, QuantumOptics), nonlinear dynamics (JuliaDynamics), quantitative economics (QuantEcon), astronomy (JuliaAstro) and ecology (EcoJulia). With a set of highly enthusiastic developers and maintainers from various parts of the scientific community, this ecosystem will only continue to get bigger and bigger.
Statistical Graphics
(Lots of Solutions)
Grammar of Graphics Based

  – (Julia) Gadfly
  – (Python) ggplot
Examples: Grammar of Graphics

Geoms
- geom_abline
- geom_area
- geom_bar
- geom_density
- geom_histogram
- geom_hline
- geom_jitter
- geom_line
- geom_matrix
- geom_point
- geom_rect
- geom_step
- geom_text
- geom_tile
- geom_vline

Stats
- stat_bind
- stat_function
- stat_smooth

Facets
- facet_grid
- facet_wrap

Themes
- theme_bw
- theme_gray
- theme_venn
- theme_xkcd
- theme_bw

Aesthetics
- scale_color_brewer
- scale_color_gradient
- scale_color_gradient2
- scale_color_hue
- scale_color_hue2

Scales
- scale_x_continuous
- scale_x_log
- scale_y_continuous
- scale_y_log
- scale_y_log10

Other
- ggplot
Language Specific Solutions

Julia Specific

StatPlots

- Types:
  - DataFrames
  - Distributions
- Recipes:
  - histogram/histogram2d
  - boxplot
  - violin
  - marginalhist
  - corplot/cornerplot

Initialize:

```text
#pkg.clone("git@github.com:JuliaPlots/StatPlots.jl.git")
using StatPlots
gr(size=(400,300))
```

Python Specific

Scripting Layer

pyplot

Backend Layer

Artist Layer

Artist Implementations

Backend Implementations

User Interface Backends

Hardcopy Backends

- GTK
- wxWidgets
- Tk
- MacOS X

- Raster
- AGG
- GDK
- NPBAD
- Cairo

- Vector
- PS
- PDF
- SVG
- Cairo

RecipesBase.jl
Base package for defining transformation recipes on user types for Plots.jl
Julia: Other Considerations

JuliaPy
Software that connects the Julia and Python languages.

Pandas.jl
A Julia front-end to Python’s Pandas package.

PyPlot.jl
Plotting for Julia based on matplotlib.pyplot
• Python 2 & 3
  – Python 2 & Python 3 are two different packages - don’t let the singular name “Python” add confusion
  – Both are easy to use, general purpose languages that are more powerful than typical shell scripting languages (e.g., bash)
  – Modern replacement for the BASIC programming language with similar goals – a language that is designed for the casual or part time programmer, or a person that is learning programming concepts
  – Relatively easy to extend
  – Because of the extension packages (NumPy in particular), Python can now manipulate data with good performance (i.e., an important characteristic in the domain of data science and scientific computing)
  – It’s package management system is a mess, hence the need for “distributions”
    • And “virtual environments”
  – Useful for small scripts / prototyping; avoid use in larger applications (i.e., “production code”) – in other words, leverage strengths, minimize weaknesses
IMO: Julia Comments

- Julia 1.x
  - A language designed for technical computing, an explicit goal
    - Native support for high performance arrays
  - Promotes modern programming paradigms (verbs are center of attention, not nouns)
  - Ecosystem includes a standard package management system
  - JIT executed and built on LLVM – a modern compiler infrastructure
  - Packages exist to tap existing R and Python packages, if needed
  - First class documentation – you can learn much about the computing landscape by reading it
  - Designed to avoid the two-language problem (e.g., optional typing)
    - Promotes users becoming developers, and developers becoming users (avoid learning two or more languages)
  - Active community that engages in deep technical discussions concerning improving the performance of technical computing
Further Reading

- **Python**
  - Python Data Science Handbook does an excellent job of describing performance issues with Python and how SciPy packages are organized to optimize it for specific use cases

- **Julia**
  - Online docs
  - YouTube: “Josh Day Julia for Modern Data Analysis” – 30 minutes

- **General**
  - “Execution in the Kingdom of Nouns”, Stevey’s Blog Rants, 2006
EOF