UPPSALA
UNIVERSITET

Best Practices II: Testing, Documenting And Packaging Of Code

2018-02-08

Advanced Scientific Programming with Python

SciPy

- The scipy package contains various toolboxes dedicated to common issues in scientific computing.
- Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.
- scipy can be compared to other standard scientific-computing libraries, such as the GSL (GNU Scientific Library for C and C++), or Matlab's toolboxes.
- scipy is the core package for scientific routines in Python; it is meant to operate efficiently on numpy arrays, so that numpy and scipy work hand in hand.

Why SciPy?

- Before implementing a routine, it is worth checking if the desired data processing is not already implemented in SciPy.
- As non-professional programmers, scientists often tend to re-invent the wheel, which leads to buggy, non-optimal, difficult-to-share and unmaintainable code.
- By contrast, SciPy's routines are optimized and tested, and should therefore be used when possible.

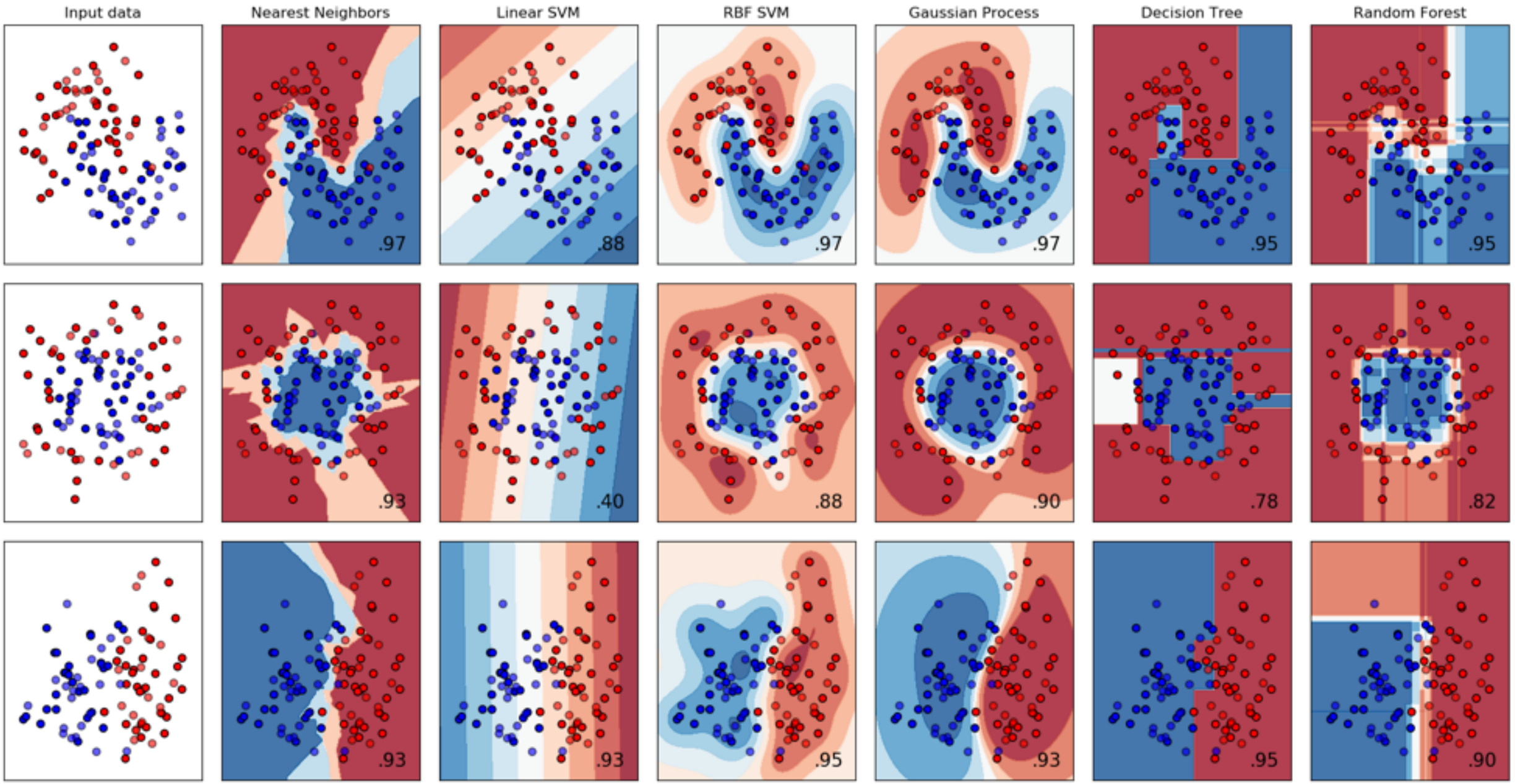
What's In SciPy?

- Clustering package (scipy.cluster)
- Constants (scipy.constants)
- Discrete Fourier transforms (scipy.fftpack)
- Integration and ODEs (scipy.integrate)
- Interpolation (scipy.interpolate)
- Input and output (scipy.io)
- Linear algebra (scipy.linalg)
- Miscellaneous routines (scipy.misc)
- Multi-dimensional image processing (scipy.ndimage)
- Orthogonal distance regression (scipy.odr)
- Optimization and root finding (scipy.optimize)
- Signal processing (scipy.signal)
- Sparse matrices (scipy.sparse)
- Sparse linear algebra (scipy.sparse.linalg)
- Compressed Sparse Graph Routines (scipy.sparse.csgraph)
- Spatial algorithms and data structures (scipy.spatial)
- Special functions (scipy.special)
- Statistical functions (scipy.stats)
- Statistical functions for masked arrays (scipy.stats.mstats)
- Low-level callback functions

SciPy Notebook

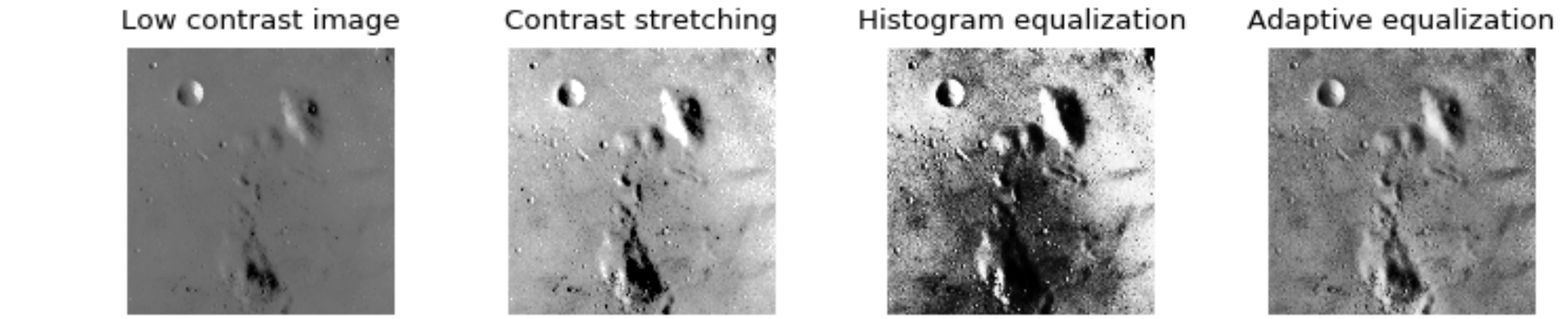
Beyond Numpy And Scipy

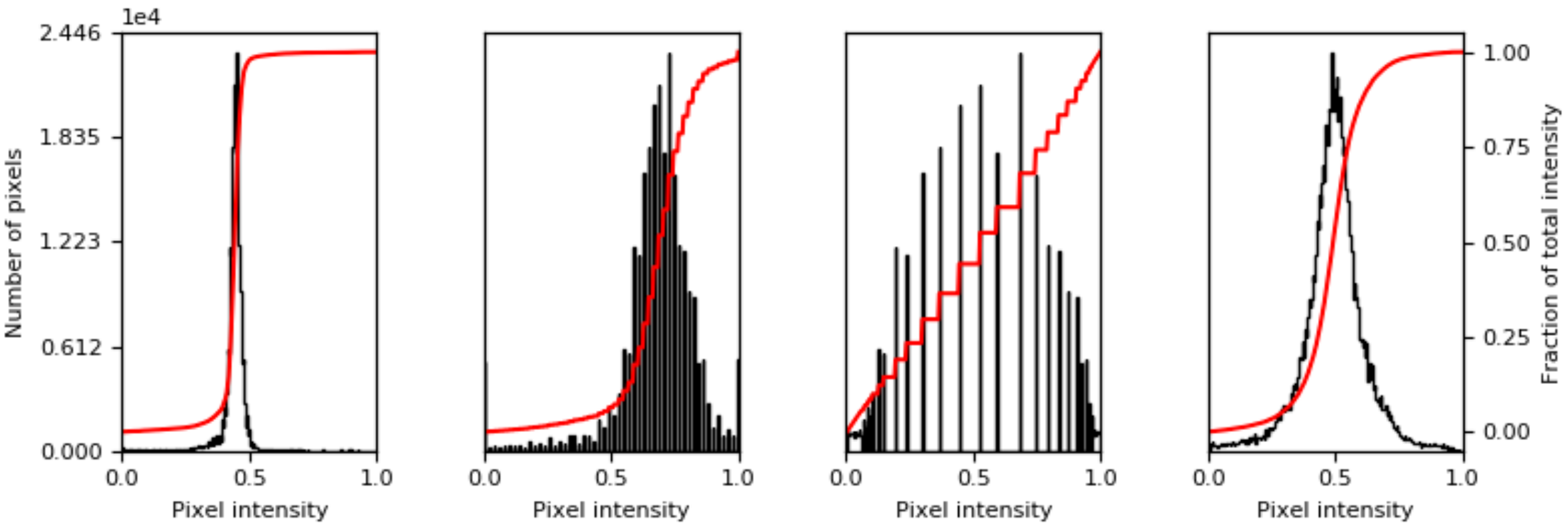
- scikit-learn - Simple Machine Learning in Python



Beyond Numpy And Scipy

- scikit-image - Image Processing in Python





Basics Of Cython

- The fundamental nature of Cython can be summed up as follows: Cython is Python with C data types.
- Almost any piece of Python code is also valid Cython code. The Cython compiler will convert it into C code which makes equivalent calls to the Python/C API.
- But Cython is much more than that, because parameters and variables can be declared to have C data types.
- Code which manipulates Python values and C values can be freely intermixed, with conversions occurring automatically wherever possible.
- Reference count maintenance and error checking of Python operations is also automatic, and the full power of Python's exception handling facilities, including the try-except and try-finally statements, is available to you – even in the midst of manipulating C data.

Cython Hello World

- As Cython can accept almost any valid python source file, one of the hardest things in getting started is just figuring out how to compile your extension.
- So lets start with the canonical python hello world. We'll save it under **helloworld.pyx**:

```
print("Hello World")
```

- Now we need to create a **setup.py** to compile this:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("helloworld.pyx")
)
```

- To now build your Cython file do:

```
$ python setup.py build_ext --inplace
```

- Which will leave a file in your local directory called helloworld.so in unix or helloworld.pyd in Windows.

Cython Hello World

- Now to use this file: start the python interpreter and simply import it as if it was a regular python module:

```
>>> import helloworld
Hello World
```

- Congratulations! You now know how to build a Cython extension.

- But so far this example doesn't really give a feeling why one would ever want to use Cython, so lets create a more realistic example.

Cython Primes

primes.py

```
import numpy

"""
 Calculates first kmax primes
"""
def primes(kmax):
    p = numpy.zeros((1000),dtype=numpy.int)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

cy_primes.pyx

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Cython Primes

```
In [1]: import primes

In [2]: import cy_primes

In [3]: %timeit primes.primes(2000)
1 loop, best of 3: 704 ms per loop

In [4]: %timeit cy_primes.primes(2000)
100 loops, best of 3: 1.92 ms per loop
```

For certain code Cython give massive performance gains!

Adapted from:
http://materials.jeremybejarano.com/MPIwithPython/introMPI.html
https://mpi4py.readthedocs.io/en/stable/intro.html#what-is-mpi

MPI

- MPI, the Message Passing Interface, is a standardized and portable message-passing system designed to function on a wide variety of parallel computers.
- The standard defines the syntax and semantics of library routines and allows users to write portable programs in the main scientific programming languages (Fortran, C, or C++).
- Since its release, the MPI specification has become the leading standard for message-passing libraries for parallel computers.
- Implementations are available from vendors of high-performance computers and from well known open source projects like MPICH, Open MPI or LAM.

Introduction to MPI

- As tradition has it, we will introduce you to MPI programming using a variation on the standard hello world program.
- Our first MPI python program will be the Hello World program for multiple processes. The source code is as follows:

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print "hello world from process ", rank
```

- After saving this text as hello.py, it is executed using the following command-line syntax, run from the file's directory:

```
$ mpiexec  -n 5 python  hello.py
```

- The above command will execute five python processes which can all communicate with each other.

Introduction to MPI

- When each program runs, it will print hello, and tell you its rank:

```
hello world from process 0
hello world from process 1
hello world from process 3
hello world from process 2
hello world from process 4
```

- Notice that when you try this on your own, they do not necessarily print in order. This is because 5 separate processes are running on different processors, and we cannot know beforehand which one will execute its print statement first.
- If the processes are being scheduled on the same processor instead of multiple processors, then it is up to the operating system to schedule the processes, and it has no preference of any one of our processes over any other process of ours. In essence, each process executes autonomously.

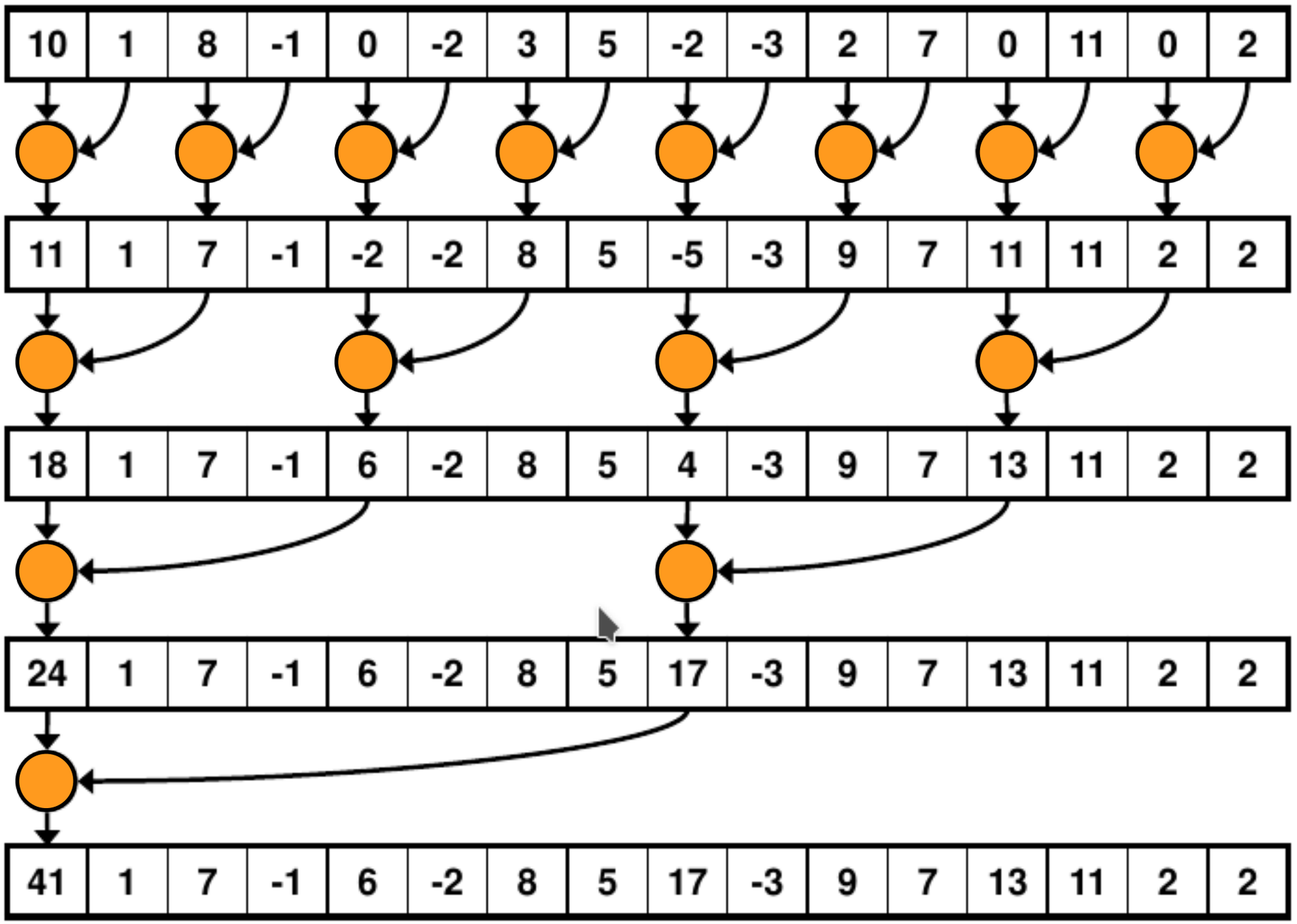
Point To Point Communication

- The simplest message passing involves two processes: a sender and a receiver.
- Let's make two processes. One will draw a random number and then send it to the other.
- We will do this using the routines Comm.Send and Comm.Recv:

```
#passRandomDraw.py
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print "Process", rank, "drew the number", randNum[0]
        comm.Send(randNum, dest=0)

if rank == 0:
        print "Process", rank, "before receiving has the number", randNum[0]
        comm.Recv(randNum, source=1)
        print "Process", rank, "received the number", randNum[0]
```

Collective Communication

- Suppose we have eight processes, each with a number to be summed.
- What's the best way to do the sum while minimising communication?



Collective Communication

```
#collective.py
#example to run: mpiexec -n 4 python collective.py 10000
import numpy
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

n = int(sys.argv[1])
x = numpy.linspace(start=float(rank)/size, stop=float(1+rank)/size, num=n/size,
endpoint=False)
cosx = numpy.cos(x)

#initializing variables. mpi4py requires that we pass numpy objects.
integral = numpy.zeros(1)
total = numpy.zeros(1)

# perform local computation. Each process integrates its own interval
integral[0] = numpy.sum(cosx)*1.0/n
print("Estimate of integral of cos(x) from %f to %f is %f" % (float(rank)/size,
float(1+rank)/size, integral))
```

Collective Communication

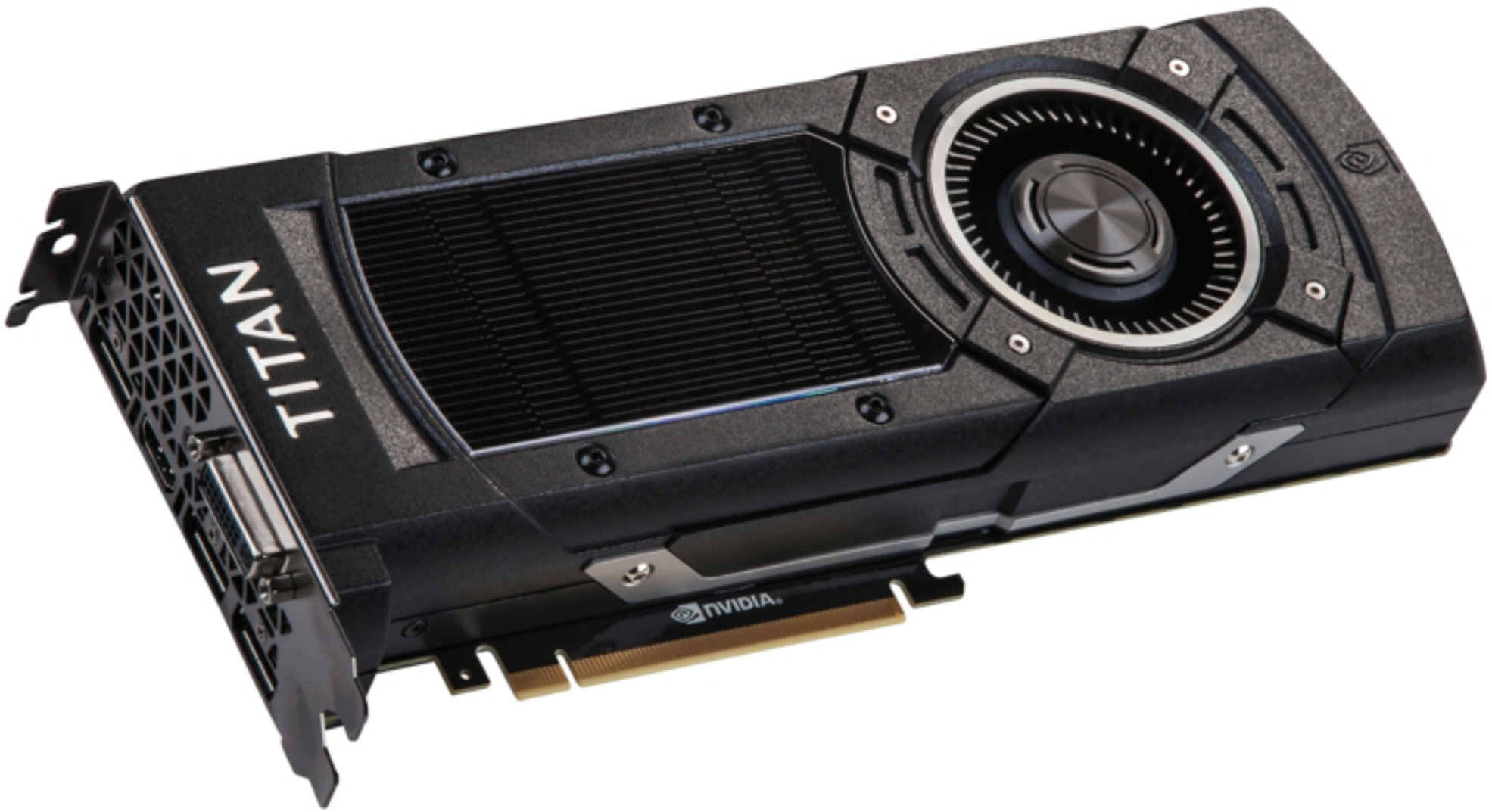
```
# communication
# root node receives results with a collective "reduce"
comm.Reduce(integral, total, op=MPI.SUM, root=0)

# root process prints results
if comm.rank == 0:
    print("With n=%d our estimate of the integral from 0 to 1 of cos(x)
is %f" % (n, total))
    print("Exact integral (sin(1)) is %f" % (numpy.sin(1.0)))
```

More on MPI

- There is a wealth of resources about MPI and the subject is too vast for one lecture
- Just be aware that you ca use it from Python with mpi4py
- For more information about MPI in general check:

  http://mpitutorial.com/
- And for mpi4py in particular check the documentation at:

  https://mpi4py.readthedocs.io/en/stable/index.html

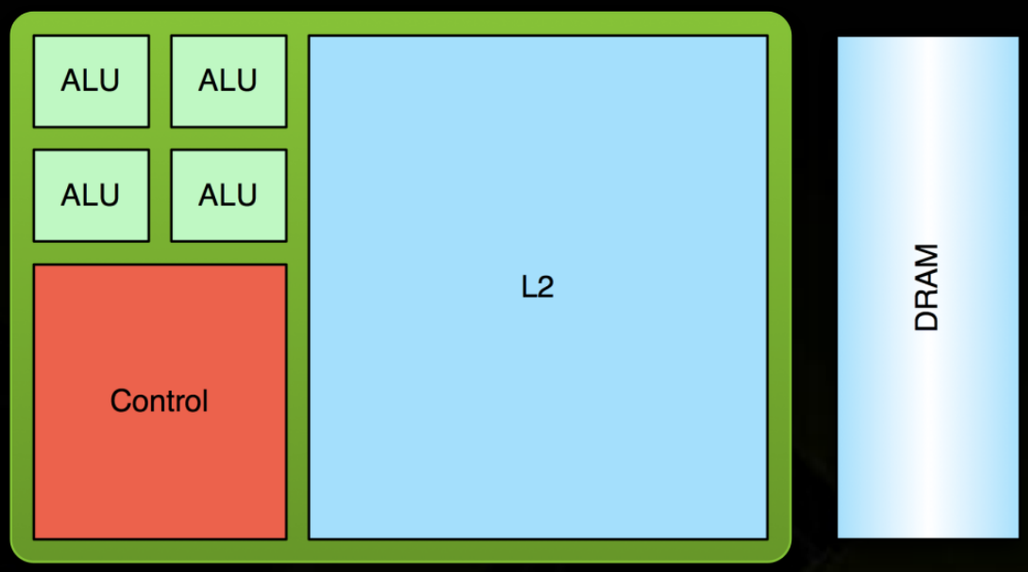


Adapted from:
http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf
https://documen.tician.de/pycuda/tutorial.html

GPU Acceleration

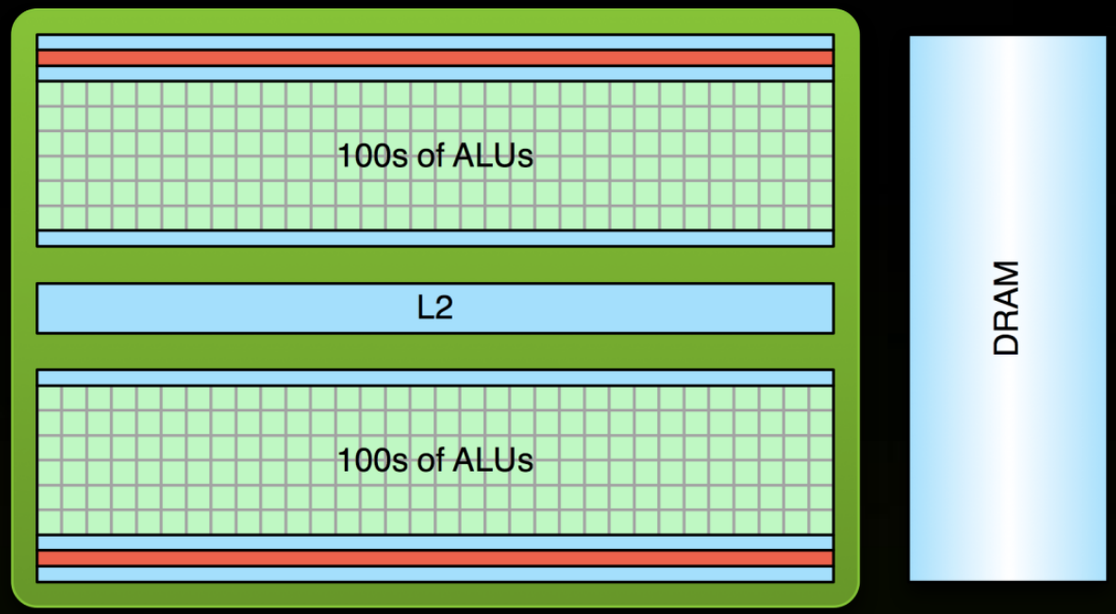
- Another way to obtain large gains in performance is to use GPUs to carry out a large fraction of the calculations
- GPUs have enormous computing power. For example an NVIDIA Titan XP is capable of 11 TFLOPS in single precision for $1500.
- For comparison an Intel Xeon E5-2699 v4 (a top of the line $4000 CPU) reaches only 1.8 TFLOPS
- The two main languages to program GPUs are OpenCL and CUDA
- You can use them from Python using the packages `PyCuda` and `PyOpenCL`, but this requires knowledge CUDA and OpenCL respectively.
- Another way to take advantage of them is to use libraries to do their computations in GPUs.
- An example is `arrayfire-python` which provides a range of numerical algorithms.

Low Latency or High Throughput?



CPU

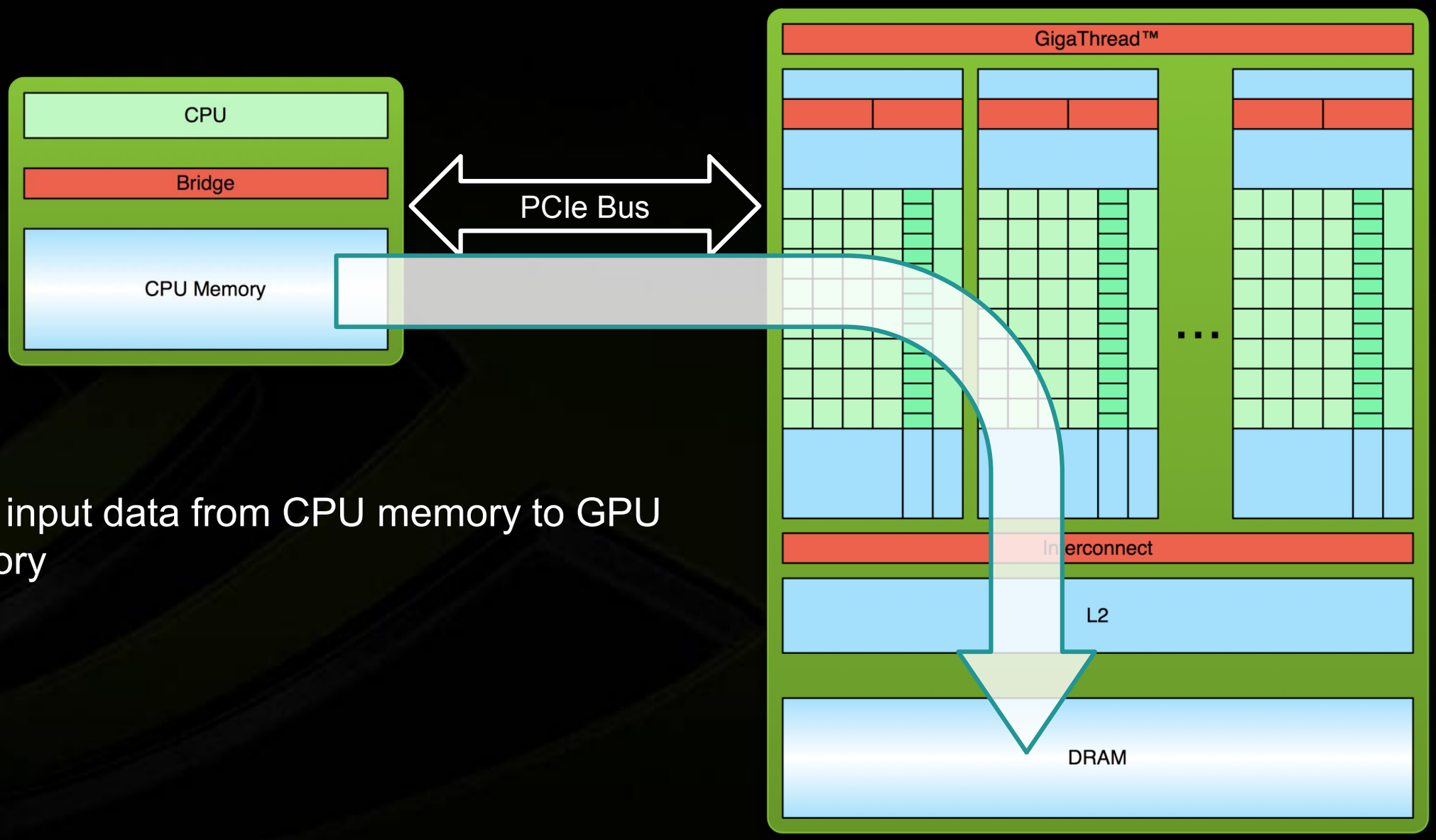
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution



GPU

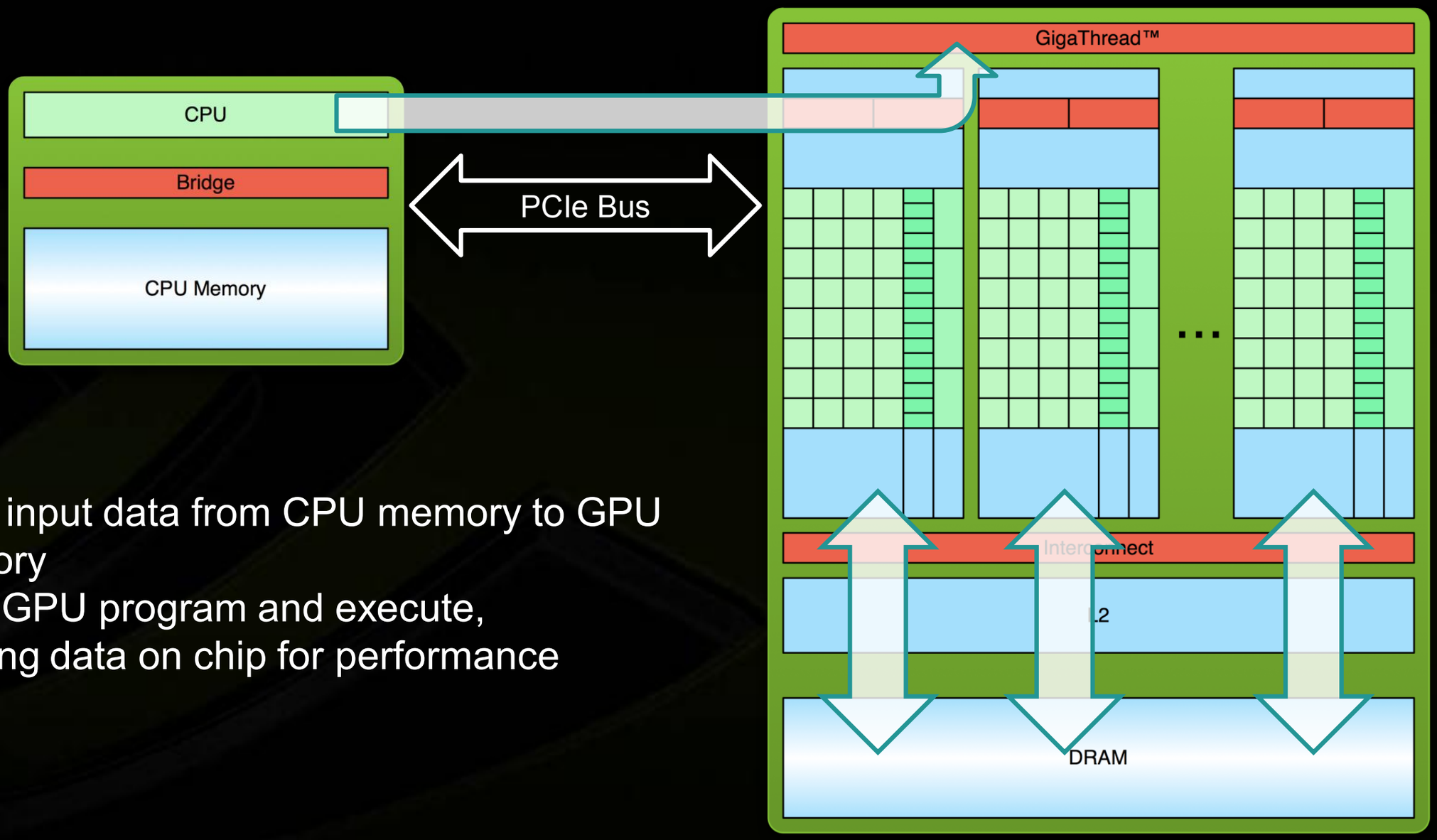
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Processing Flow



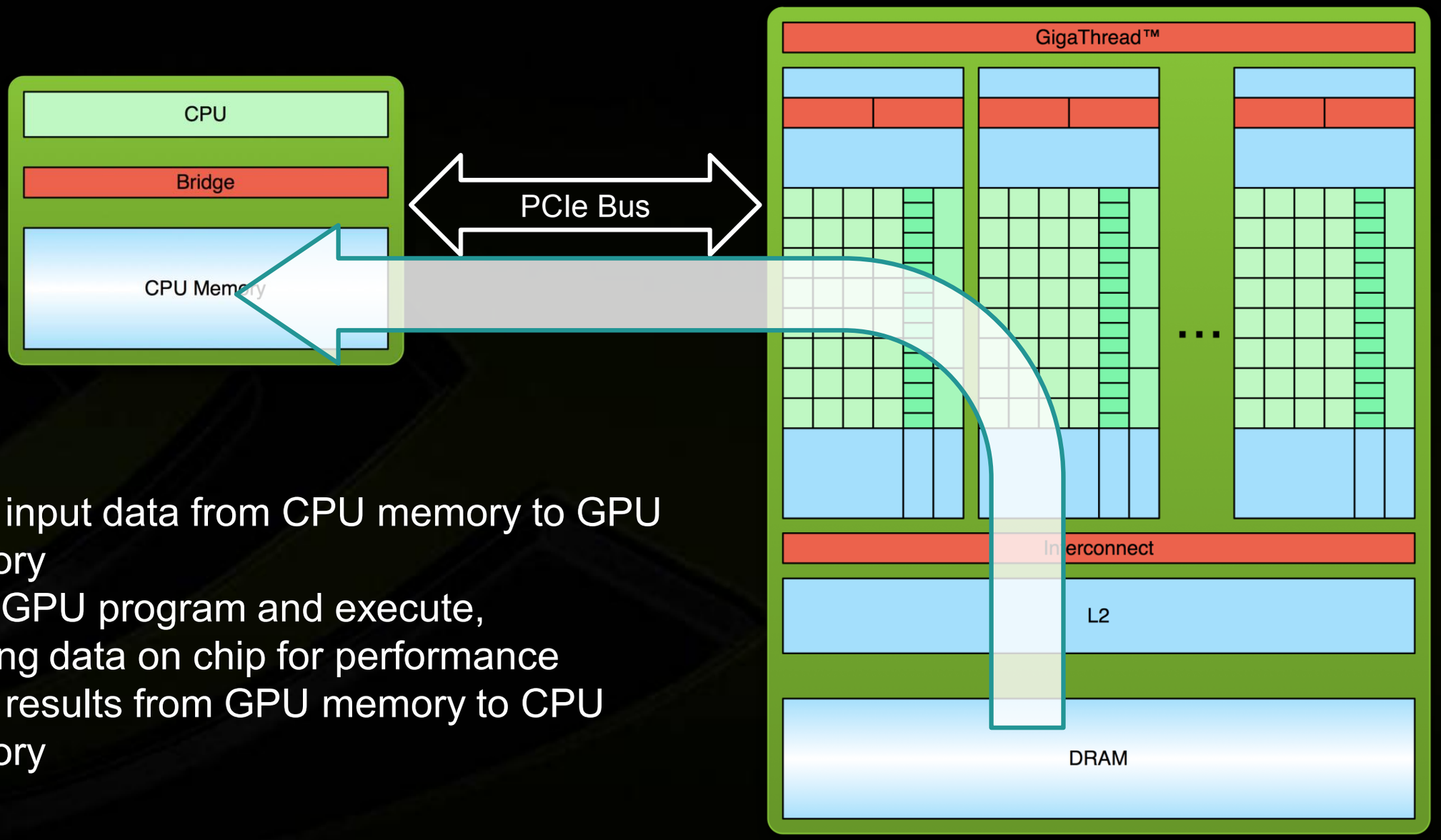
1. Copy input data from CPU memory to GPU memory

Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

PyCUDA example

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule

import numpy
# Let's make a 4x4 array of random numbers
a = numpy.random.randn(4,4)

# But wait, a consists of double precision numbers,
# but most nVidia devices only support single precision:
a = a.astype(numpy.float32)

# Finally, we need somewhere to transfer data to,
# so we need to allocate memory on the device
a_gpu = cuda.mem_alloc(a.nbytes)

# As a last step, we need to transfer the data to the GPU:
cuda.memcpy_htod(a_gpu, a)

# Kernel to double each entry in an array
# which is compiled by SourceModule
mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)
```

PyCUDA example

```
# Grab the compiled function
func = mod.get_function("doublify")

# Execute the function on the GPU
# Using one block with 4x4 threads
func(a_gpu, block=(4,4,1))

# Finally, we fetch the data back from the GPU and
# display it, together with the original a

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print(a_doubled)
print(a)
```

```
[[ 0.51360393  1.40589952  2.25009012  3.02563429]
 [-0.75841576 -1.18757617  2.72269917  3.12156057]
 [ 0.28826082 -2.92448163  1.21624792  2.86353827]
 [ 1.57651746  0.63500965  2.21570683 -0.44537592]]
[[ 0.25680196  0.70294976  1.12504506  1.51281714]
 [-0.37920788 -0.59378809  1.36134958  1.56078029]
 [ 0.14413041 -1.46224082  0.60812396  1.43176913]
 [ 0.78825873  0.31750482  1.10785341 -0.22268796]]
```

PyCUDA example

- Now lets do it the easy way!
- PyCUDA has some support for NumPy arrays so you don't need to write kernels for many things!

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

- For more information check the **pycuda.gpuarray.GPUArray** class

- ArrayFire is a general-purpose library that simplifies the process of developing software that targets parallel and massively-parallel architectures including CPUs, GPUs, and other hardware acceleration devices.
- Developers write code which performs operations on ArrayFire arrays which, in turn, are automatically translated into near-optimal kernels that execute on the computational device.
- ArrayFire runs on CPUs from all major vendors (Intel, AMD, ARM), GPUs from the prominent manufacturers (NVIDIA, AMD, and Qualcomm), as well as a variety of other accelerator devices on Windows, Mac, and Linux.
- And **most importantly** it's Open Source and available on GitHub! https://github.com/arrayfire/arrayfire

GitHub

arrayfire-python example

Arrayfire pi.py

```
import arrayfire as af

# Monte Carlo estimation of pi
def calc_pi_device(samples):
    x = af.randu(samples)
    y = af.randu(samples)
    within_unit_circle = (x * x + y * y) < 1
    return 4 * af.count(within_unit_circle) / samples
```

NumPy np_pi.py

```
import numpy as np

# Monte Carlo estimation of pi
def calc_pi_device(samples):
    x = np.random.random(size=samples)
    y = np.random.random(size=samples)
    within_unit_circle = (x * x + y * y) < 1.0
    return 4.0 * sum(within_unit_circle) / samples
```

```
In [7]: %timeit np_pi.calc_pi_device(100000)
10 loops, best of 3: 90.2 ms per loop

In [8]: %timeit pi.calc_pi_device(100000)
The slowest run took 80.80 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000 loops, best of 3: 859 µs per loop
```

pytest

Adapted from:
https://jacobian.org/writing/getting-started-with-pytest/

Testing

- Testing is crucial to high quality software!
- It's important to make testing as automated as possible to make sure it's done!
- Python as several packages that provide **test frameworks**: `unittest`, `nose/nose2`, `py.test`
- There are also services that automatically build your code and run tests every time you `push` your code.
- A few of these **Continuous Integration** services are Travis CI (travis-ci.org), GitLab (gitlab.com), CircleCI (circleci.com) and CodeShip (codeship.com).
- In this course we'll focus on `py.test` and **Travis CI**.

py.test in action

```python
# fib.py
def fib(n):
    """Return the first Fibonacci number above n."""
    a = 0
    b = 1
    while b < n:
        a, b = b, a + b
    return b
```

- Lets test our code above. The file with the tests should start with **`test_`**:

```python
# test_fib.py
import fib

# the name of the testing function should
# also start with test_
def test_fib():
    assert fib(0) == 1
```

- Now to run the test we simply do:

```
$ py.test
```

py.test in action

```
$ py.test
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/filipe/Documents/Teaching/Advanced Scientific Programming with Python/
python-course/day4-bestpractices-2/code/pytest, inifile:
collected 1 items

test_fib.py F

=================================== FAILURES ===================================
_____________________________________ test_fib _____________________________________

    def test_fib():
>       assert fib(0) == 1
E       TypeError: 'module' object is not callable

test_fib.py:4: TypeError
=========================== 1 failed in 0.01 seconds ===========================
```

As you can see you can also have bugs in your tests!

py.test in action

- And after fixing the silly error:

```
$ py.test
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/filipe/Documents/Teaching/Advanced Scientific Programming with Python/
python-course/day4-bestpractices-2/code/pytest, inifile:
collected 1 items

test_fib.py .

=========================== 1 passed in 0.01 seconds ===========================
```

- This is the most important feature of **`py.test`**, but it can do much more!
- Check pytest.org and py.test --help for more information.
- Lets do a slightly more advanced example now…

py.test in action

- Multiple test parameters with one function:

```
# test_fib_params.py
import fib
import pytest

# Fibonacci Sequence
# 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
@pytest.mark.parametrize("n, expected", [
    (0, 1),
    (1, 2),
    (3, 5)
])
def test_fib_parametrized(n, expected):
    assert fib.fib(n) == expected
```

- Lets try this again:

```
$ py.test
```

py.test in action

```
$ py.test
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/filipe/Documents/Teaching/Advanced Scientific Programming with Python/python-
course/day4-bestpractices-2/code/pytest, inifile:
collected 4 items

test_fib.py .
test_fib_params.py .FF

=================================== FAILURES ===================================
_____________________________ test_fib_parametrized[1-2] _____________________________

n = 1, expected = 2

    @pytest.mark.parametrize("n, expected", [
        (0, 1),
        (1, 2),
        (3, 5)
    ])
    def test_fib_parametrized(n, expected):
>       assert fib.fib(n) == expected
E       assert 1 == 2
E        +  where 1 = <function fib at 0x107876840>(1)
E        +    where <function fib at 0x107876840> = fib.fib

test_fib_params.py:11: AssertionError
```

py.test in action

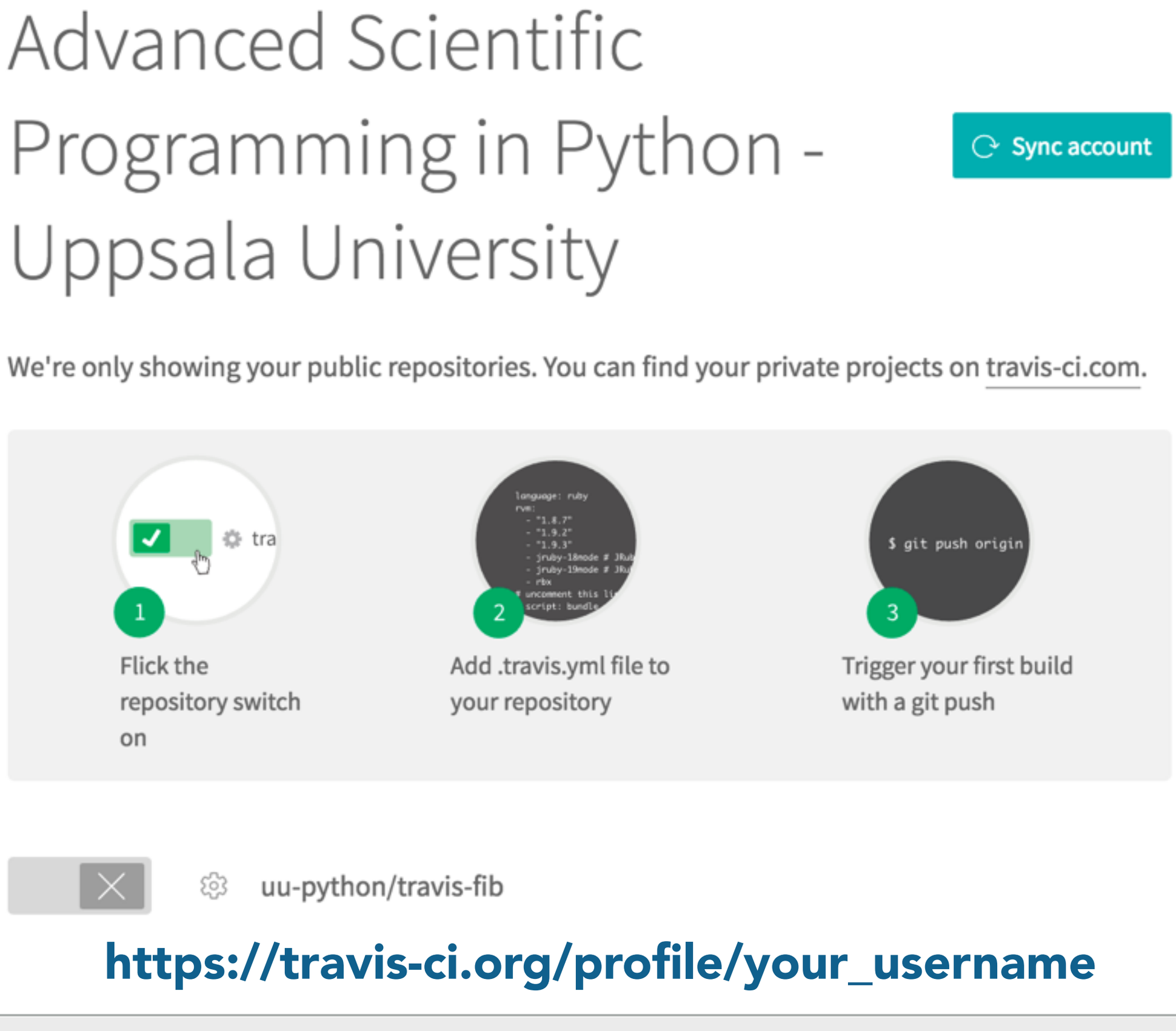
```
_____________________________ test_fib_parametrized[3-5] ______________________________

n = 3, expected = 5

    @pytest.mark.parametrize("n, expected", [
        (0, 1),
        (1, 2),
        (3, 5)
    ])
    def test_fib_parametrized(n, expected):
>       assert fib.fib(n) == expected
E       assert 3 == 5
E        +  where 3 = <function fib at 0x107876840>(3)
E        +    where <function fib at 0x107876840> = fib.fib

test_fib_params.py:11: AssertionError
====================== 2 failed, 2 passed in 0.02 seconds ======================
```

Conclusion: Our “simple” function does not do what it claims!

What does it really do...?


Travis CI

Enabling Travis CI for the repository



https://travis-ci.org/profile/your_username

Using Travis CI

- Travis CI has a strong integration with **`git`**
- I've placed our Fibonacci repository in GitHub
- Travis CI is controlled by a **`.travis.yml`** file in the repository. Here's a simple one:

```
# This project has python as the main language
language: python

# Run things with both these version of Python
python:
  - "2.7"
  - "3.4"

# Don't send email notifications for everything...
notifications:
  email: false

# We get a bare virtual machine
# We need to install our dependencies
install:
  - pip install numpy
  - pip install pytest

# Run the tests
script:
  - py.test
```

Travis CI build results

uu-python / travis-fib build passing

Current Branches Build History Pull Requests Build #2 Job #2.1 More options

master Use script not after_success for tests

Commit c67c5fd
Compare daee7ec..c67c5fd
Branch master

Filipe Maia authored and committed

#2.1 passed

Ran for 1 min 42 sec
about a minute ago

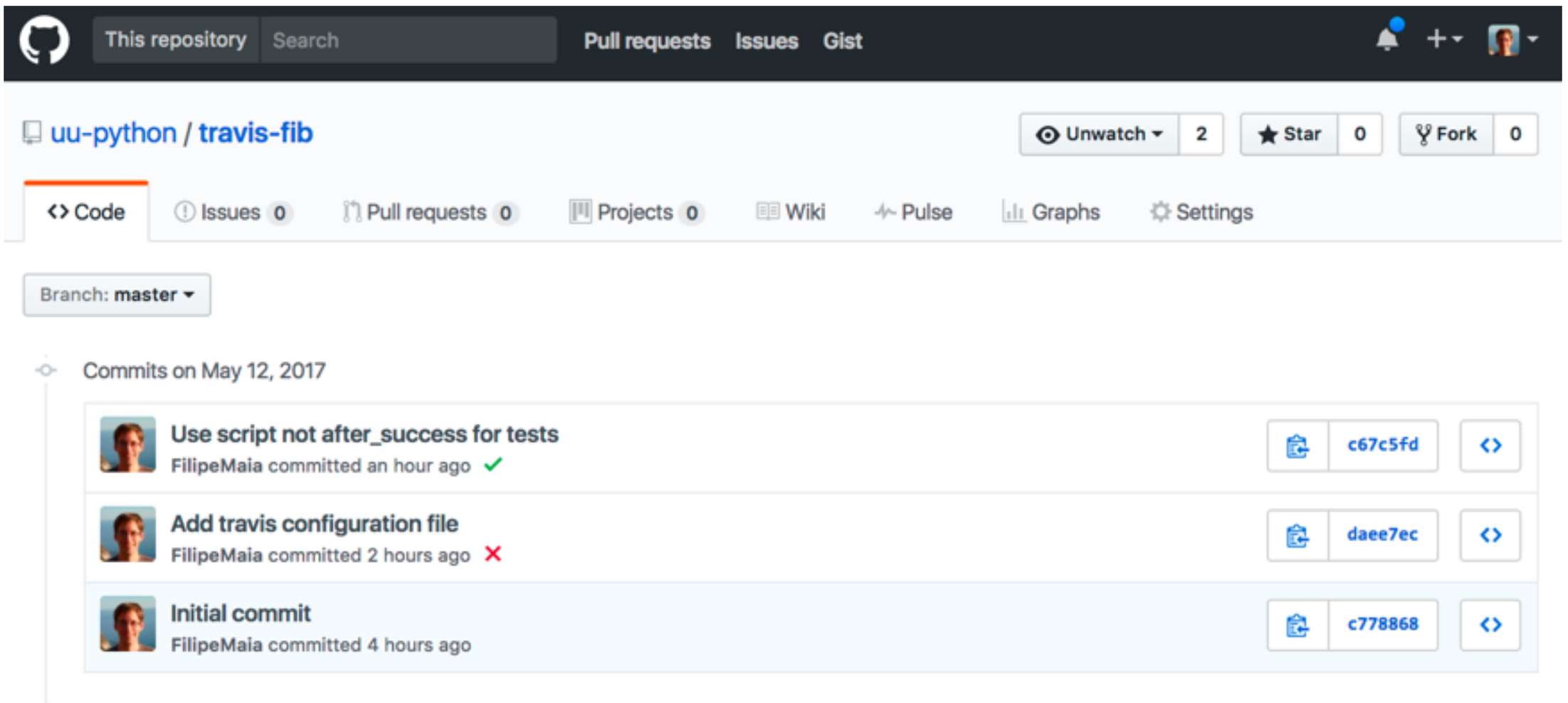
Restart job

Job log View config

Remove log Raw log

```
1  Worker information                                                    worker_info
6  Build system information                                              system_info
73
74  $ export DEBIAN_FRONTEND=noninteractive                              fix.CVE-2015-7547
110 $ sudo apt-get install libssl1.0.0                                   update_libssl1.0.0
134 $ git clone --depth=50 --branch=master https://github.com/uu-python/travis-fib.git uu-python/travis-fib   git.checkout   1.89s
144 $ source ~/virtualenv/python2.7/bin/activate                                                               0.00s
145
146 $ python --version
147 Python 2.7.9
148 $ pip --version
149 pip 6.0.7 from /home/travis/virtualenv/python2.7.9/lib/python2.7/site-packages (python 2.7)
150 $ pip install numpy                                                  install.1      0.23s
153 $ pip install pytest                                                 install.2      0.21s
157 $ py.test                                                                          0.27s
158 ============================= test session starts ==============================
159 platform linux2 -- Python 2.7.9 -- py-1.4.26 -- pytest-2.6.4
160 collected 4 items
161
162 test_fib.py .
163 test_fib_params.py ...
164
165 ========================== 4 passed in 0.02 seconds ===========================
166
167
168 The command "py.test" exited with 0.
169
170 Done. Your build exited with 0.
                                                                                                               Top
```

Travis CI Integration with GitHub



A More Complex Example

```
language: python

python:
  - "2.7"
  - "3.4"

notifications:
  email: false

sudo: false

cache:
  directories:
    - $HOME/.cache/pip
    - $HOME/arrayfire
    - $HOME/arrayfire-python
    - $HOME/local
```

A More Complex Example

```
addons:
  apt:
    sources:
      - ubuntu-toolchain-r-test
      - kubuntu-backports
    packages:
      - libatlas-base-dev
      - libfftw3-dev
      - gcc-4.9
      - g++-4.9
      - cmake
      - gdb
      - apport
```

A More Complex Example

```
before_install:
  - pip install codecov
  - cd $HOME
  - if [ ! -d "$HOME/arrayfire/.git" ]; then git clone  https://github.com/
arrayfire/arrayfire; else echo 'Using arrayfire from cached directory'; fi
  - mkdir -p arrayfire/build && cd arrayfire/build
  - git pull
  - cmake -DCMAKE_CXX_COMPILER=/usr/bin/g++-4.9 -DBUILD_CPU=ON -DBUILD_CUDA=OFF
-DBUILD_GRAPHICS=OFF -DBUILD_OPENCL=OFF -DBUILD_TEST=OFF -DBUILD_EXAMPLES=OFF -
DBUILD_UNIFIED=ON -DBUILD_CPU_ASYNC=ON -DCMAKE_INSTALL_PREFIX=${HOME}/local ..
  - make -j 2
  - make install
  - export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${HOME}/local/lib
# Install arrayfire-python
  - cd $HOME
  - if [ ! -d "$HOME/arrayfire-python/.git" ]; then git clone  https://
github.com/arrayfire/arrayfire-python.git; else echo 'Using arrayfire-python
from cached directory'; fi
  - cd arrayfire-python
  - git pull
  - python setup.py install
```

A More Complex Example

```
install:
  - pip install numpy
  - pip install ipython
  - pip install pytest-cov
  - pip install pytest-benchmark
  - cd ${HOME}/build/FilipeMaia/afnumpy
  - python setup.py install

after_success:
  - codecov

before_script:
 # Set the core file limit to unlimited so a core file is generated upon
crash
 - ulimit -c unlimited -S
 - ulimit -c

script:
  - cd ${HOME}/build/FilipeMaia/afnumpy
  - coverage run --source afnumpy -m py.test --benchmark-skip  -v --color=yes
--showlocals --durations=10
  - python -m pytest -v --benchmark-only --benchmark-compare --benchmark-
autosave  --benchmark-group-by=fullname
  - for i in $(find ./ -maxdepth 1 -name 'core*' -print); do gdb python core*
-ex "thread apply all bt" -ex "set pagination 0" -batch; done;
```

Test Coverage

gh / FilipeMaia / afnumpy — Log in

Codecov
Code coverage done right.

Add tests showing shortcomings of dot(). See #24.

79.53% | ø | ø


FilipeMaia 8 months ago ✔ CI Passed

5fa55ab master 6b77b6f

Diff | Files | Build | Graphs

/ afnumpy

Files					Coverage
core	439	281	0	158	64.01%
lib	226	196	0	30	86.73%
linalg	95	47	0	48	49.47%
__init__.py	44	39	0	5	88.64%
decorators.py	60	58	0	2	96.67%
fft.py	77	74	0	3	96.10%
indexing.py	206	144	0	62	69.90%

Documentation

- Code is read more times that it's written
- To understand the code and make use of it you need **documentation**
- Python includes **docstrings** to help you document your code
- docstrings are string literal that occurs as the first statement in a module, function, class, or method definition
- All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings
- Public methods (including `__init__.py`) should also have docstrings
- A package may be documented in the module docstring of the `__init__.py` file in the package directory

Documentation

```
# fib.py
def fib(n):
    """Return the first Fibonacci number above n."""
    a = 0
    b = 1
    while b < n:
        a, b = b, a + b
    return b
```

← Here's the docstring

- You can write anything you want in a docstring but a consistent standard greatly improves their usefulness
- One such standard is **numpydoc**
- You can read about it in at:
- https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

Numpydoc

```
def source(object, output=sys.stdout):
    """
    Print or write to a file the source code for a Numpy object.

    The source code is only returned for objects written in Python. Many
    functions and classes are defined in C and will therefore not return
    useful information.

    Parameters
    ----------
    object : numpy object
        Input object. This can be any object (function, class, module,
        ...).
    output : file object, optional
        If `output` not supplied then source code is printed to screen
        (sys.stdout).  File object must be created with either write 'w' or
        append 'a' modes.

    See Also
    --------
    lookfor, info

    Examples
    --------
    >>> np.source(np.interp)                        #doctest: +SKIP
    In file: /usr/lib/python2.6/dist-packages/numpy/lib/function_base.py
    def interp(x, xp, fp, left=None, right=None):
        \"\"\".... (full docstring printed)\"\"\"
        if isinstance(x, (float, int, number)):
            return compiled_interp([x], xp, fp, left, right).item()
        else:
            return compiled_interp(x, xp, fp, left, right)

    The source code is only returned for objects written in Python.

    >>> np.source(np.array)                         #doctest: +SKIP
    Not available for this object.
    """
```

- Here's a typical numpydoc docstring (in this case for **`numpy.source`**)
- The section names and separators are standardised.
- As all well as the parameters section.

Sphinx

Python documentation generator

Adapted from:
https://codeandchaos.wordpress.com/2012/07/30/sphinx-autodoc-tutorial-for-dummies/
http://www.sphinx-doc.org/en/stable/tutorial.html

Sphinx

- Sphinx can be used to turn your docstrings into beautiful web pages
- It was originally created for the Python documentation, and it has excellent facilities for the documentation of software projects in a range of languages.
- First lets make proper numpydoc docstrings:

```
def fib(n):
    """
    Return the first Fibonacci number above n.

    Iteratively calculate Fibonacci numbers until it finds one
    greater than n, which it then returns.

    Parameters
    ----------
    n : integer
      The minimum threshold for the desired Fibonacci number.

    Returns
    -------
    b : integer
      The first Fibonacci number greater than the input, `n`.

    Examples
    --------
    >>> fib.fib(1)
    2
    >>> fib.fib(3)
    5
    """
```

- Now lets setup our Sphinx configuration:

```
$ mkdir docs
$ sphinx-quickstart
Welcome to the Sphinx 1.2.3 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Enter the root path for documentation.
> Root path for the documentation [.]: docs
…
```

- I took mostly the default options except I enabled autodoc and asked for a Makefile to be created
- Now we need to tell it where our code is. Edit the newly created **docs/conf.py** and add:

```
sys.path.insert(0,os.path.abspath('..'))
```

- We also need to add **numpydoc** to the list of extension:

```
extensions = [
    'sphinx.ext.autodoc',
    'numpydoc'
]
```

- And finally tell Sphinx the modules that should be documented. Edit **`index.rst`** to add the modules of interest:

```
Welcome to fibonacci's documentation!
=====================================

Contents:

.. toctree::
   :maxdepth: 2

.. automodule:: fib
   :members:

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

- To actually generate the documentation run:

```
$ make html
sphinx-build -b html -d _build/doctrees   . _build/html
Making output directory...
Running Sphinx v1.2.3
loading pickled environment... not yet created
building [html]: targets for 1 source files that are out of date
updating environment: 1 added, 0 changed, 0 removed
reading sources... [100%] index
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
writing additional files... genindex py-modindex search
copying static files... done
copying extra files... done
dumping search index... done
dumping object inventory... done
build succeeded.

Build finished. The HTML pages are in _build/html.

$ open _build/html/index.html
```

Sphinx

Quick search

Go

Enter search terms or a module, class or function name.

Welcome to fibonacci's documentation!

Contents:

`fib.`**`fib`**(*n*)

Return the first Fibonacci number above n.

Iteratively calculate Fibonacci numbers until it finds one greater than n, which it then returns.

Parameters:	**n** : integer The minimum threshold for the desired Fibonacci number.
Returns:	**b** : integer The first Fibonacci number greater than the input, *n*.

Examples

```
>>> fib.fib(1)
2
>>> fib.fib(3)
5
```

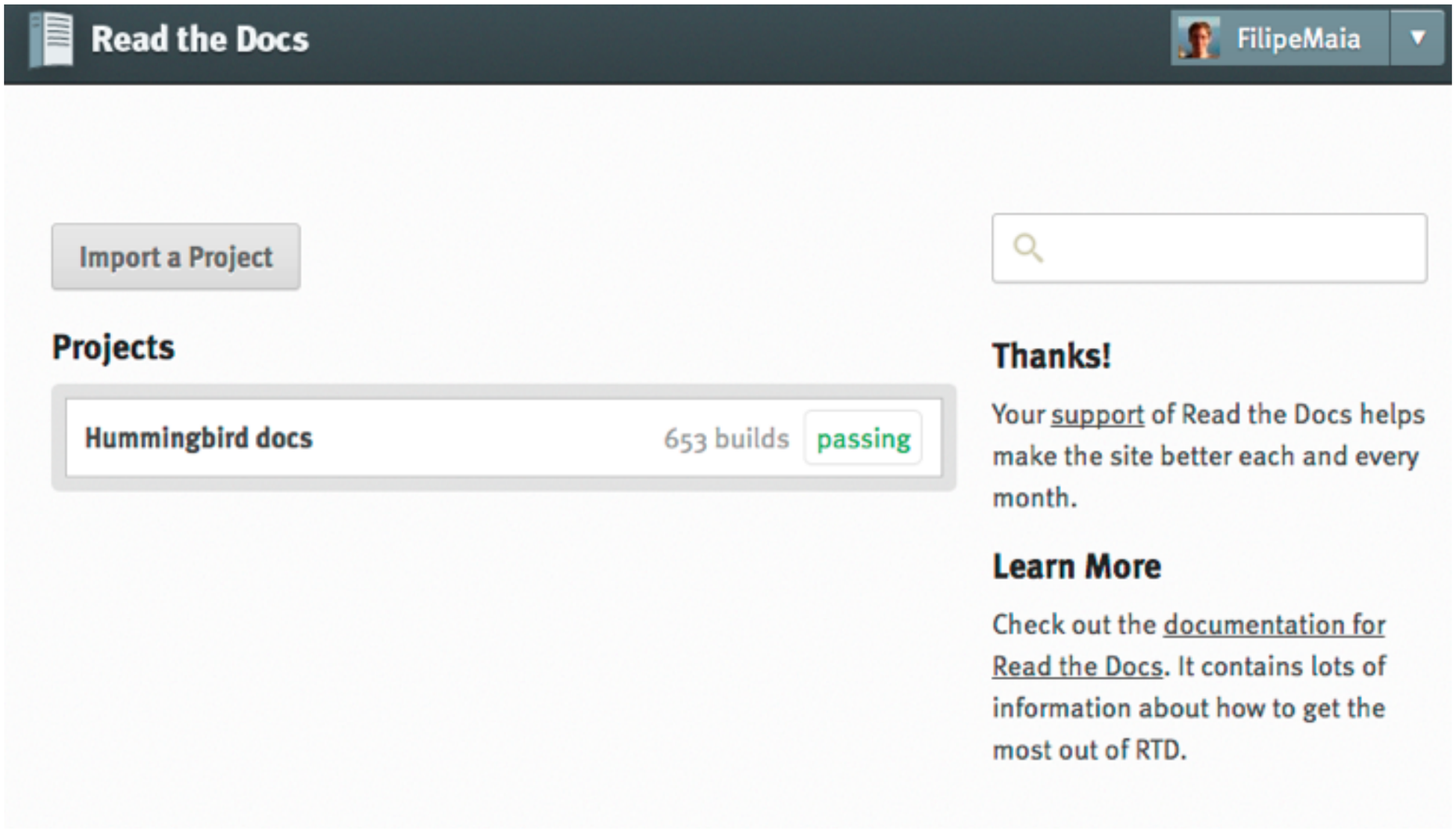
Indices and tables

- *Index*
- *Module Index*
- *Search Page*

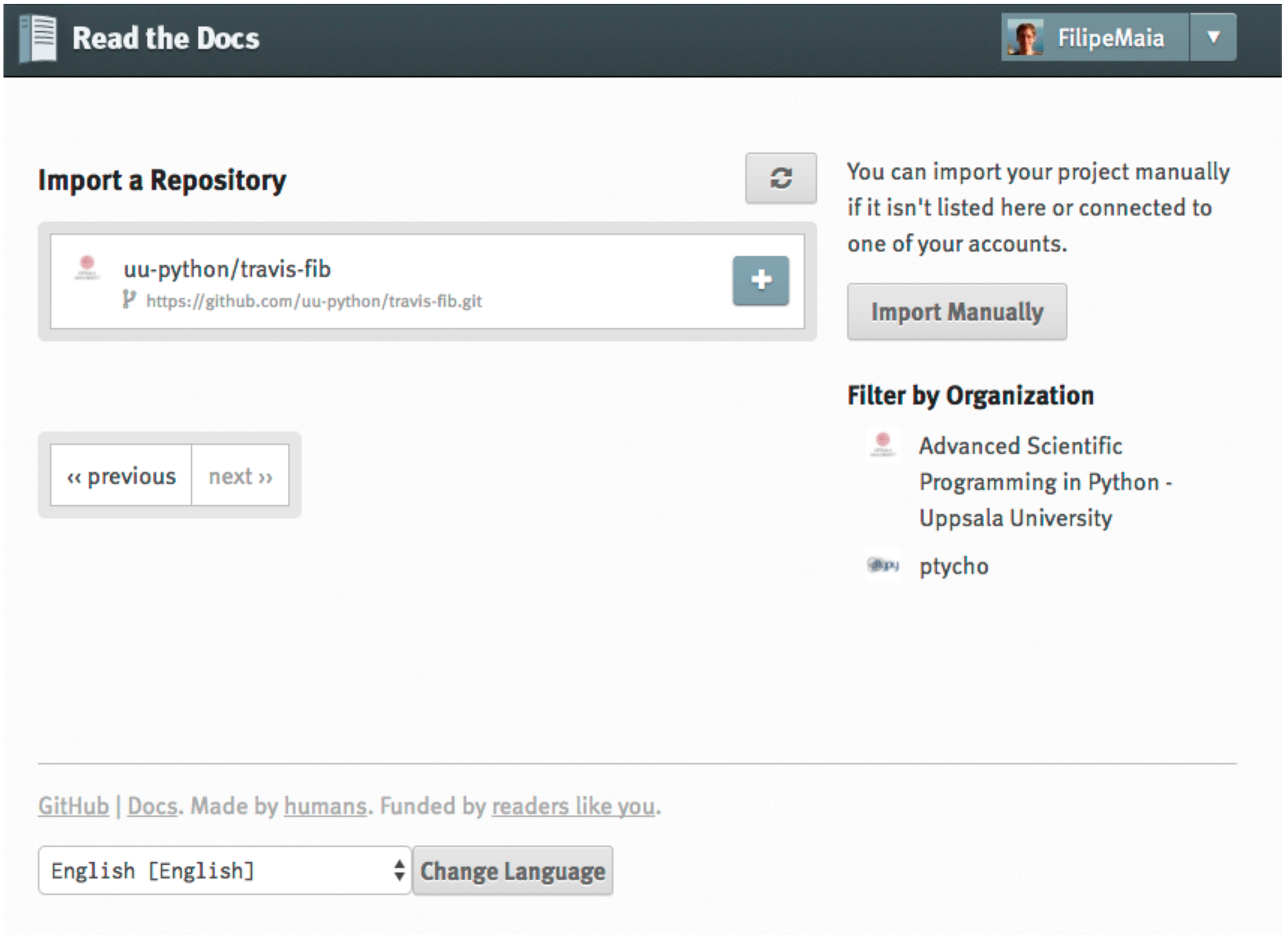
Adapted from:
http://docs.readthedocs.io/en/latest/getting_started.html

Read The Docs

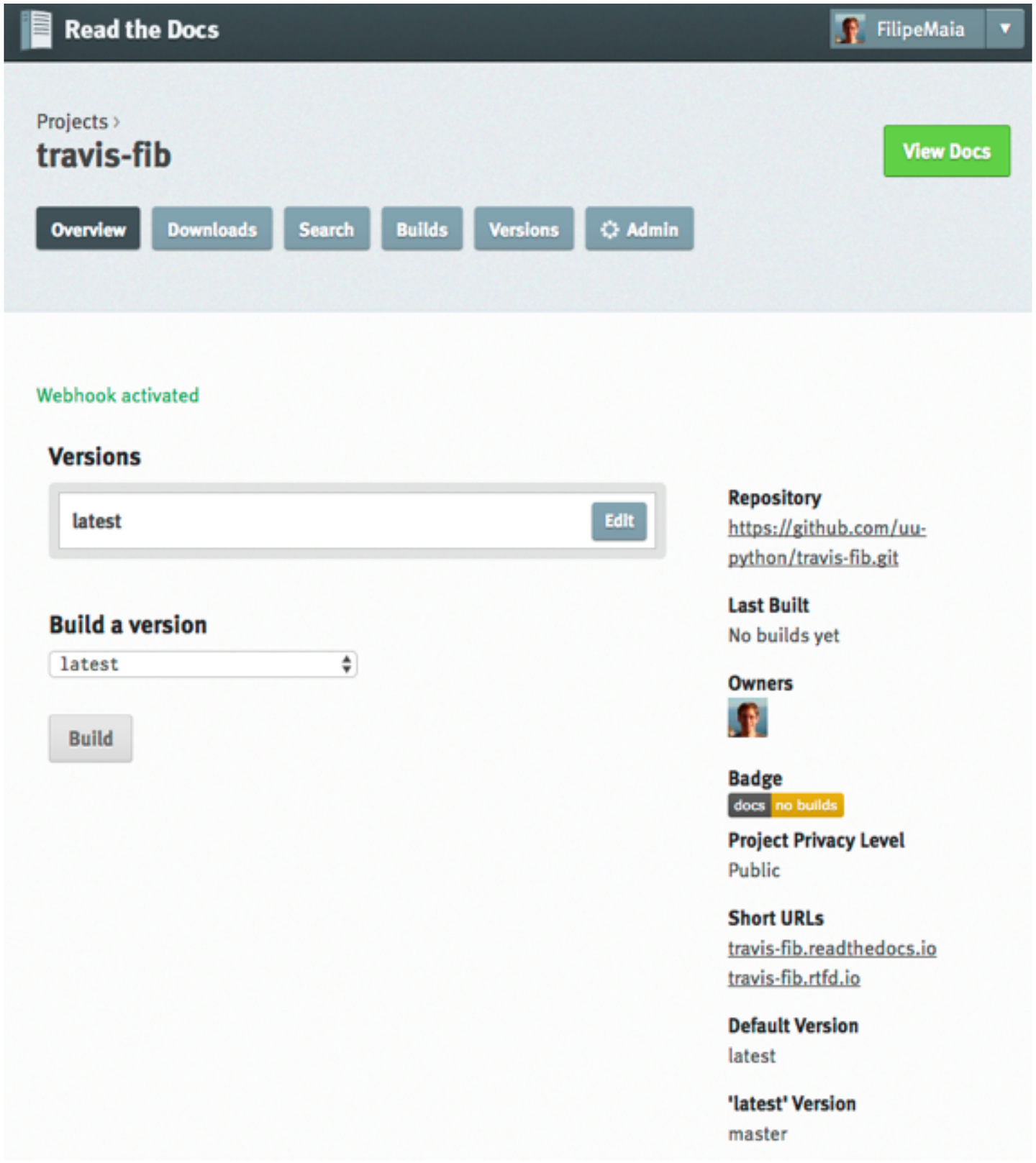
- Read The Docs (https://readthedocs.org) makes it easy to put your Sphinx documentation online.
- It's as simple as creating an account, connecting to your GitHub account and importing the project.



Read The Docs



Read The Docs



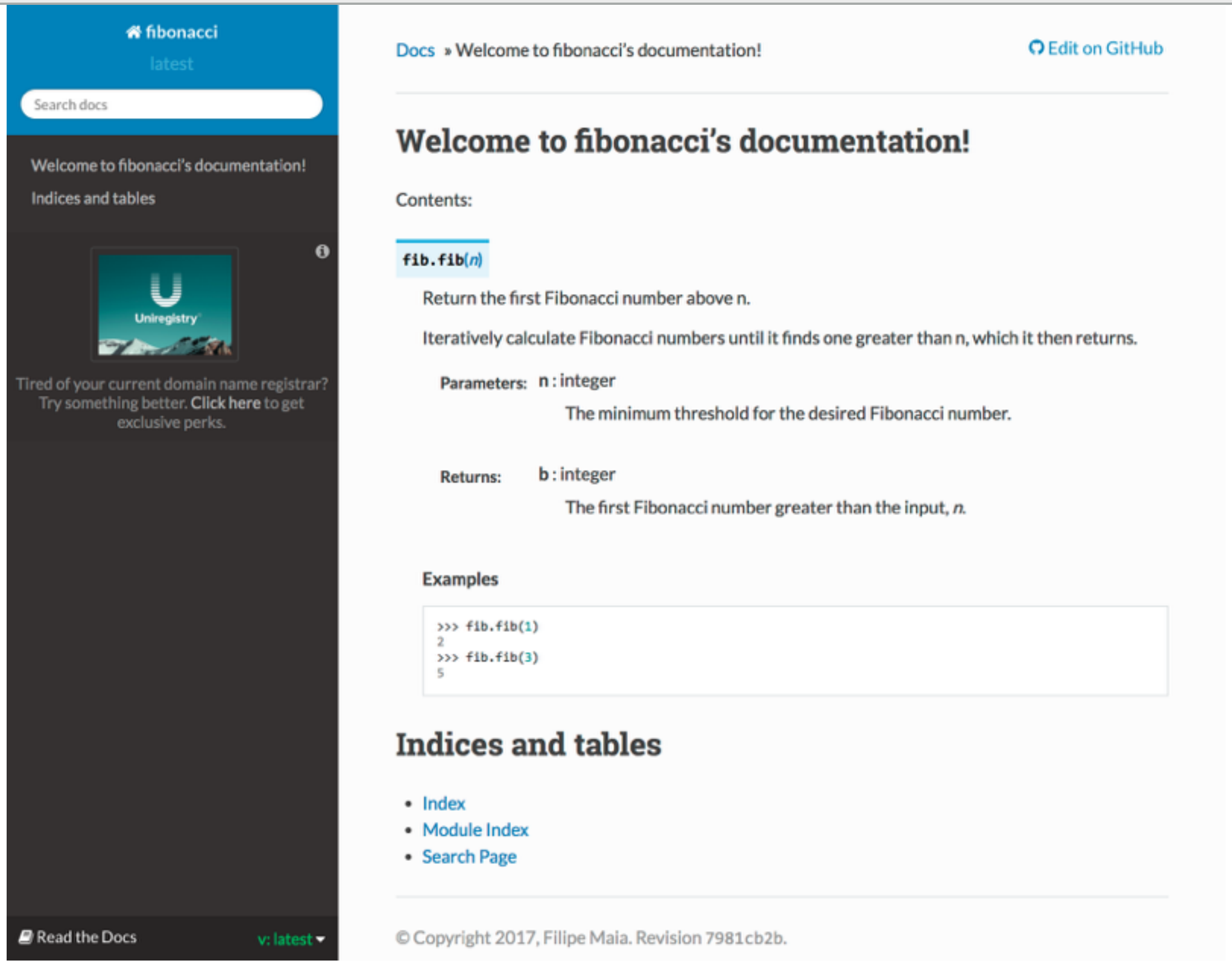
Read The Docs

- I needed to create a **`requirements.txt`** file in the root of the package with a single line saying "**`numpydoc`**".
- This lets Read The Docs know that it needs to install **`numpydoc`** before trying to generate the documentation otherwise I got:

```
Running Sphinx v1.5.3
making output directory...
...
ExtensionError: Could not import extension numpydoc (exception: No module named
numpydoc)

Extension error:
Could not import extension numpydoc (exception: No module named numpydoc)
Command time: 0s Return: 1
```

- Requirement files are an important of packaging, which we'll see in the next section.

Read The Docs



http://travis-fib.readthedocs.io/

Adapted from:
https://python-packaging.readthedocs.io/en/latest/minimal.html

Packaging

- Packaging provides an easy way for people to install your software
- PyPI (https://pypi.org/) the Python Package Index is a repository of software for the Python programming language.
- It currently contains 108 031 packages.
- Those are the packages that can be installed with `pip`.
- We'll now see how to get your small Fibonacci code into PyPI.

Picking A Name

- Python module/package names should generally follow the following constraints:

  1. All lowercase
  2. Unique on pypi, even if you don't want to make your package publicly available (you might want to specify it privately as a dependency later)

- I went with the uu-fibonacci.

- The initial file structure is as follows:

```
uu-fibonacci/
    docs/
    fib.py
    requirements.txt
```

- Now we have to create the `setup.py` file

setup.py

```
# setup.py
from setuptools import setup

setup(name='uu-fibonacci',
      version='1.0',
      description='Example package that calculates fibonacci numbers',
      url='https://github.com/uu-python/travis-fib',
      author='Filipe Maia',
      author_email='filipe.c.maia@gmail.com',
      license='BSD',
      py_modules=['fib'],
      packages=[])
```

- Now you can already install the package locally:

```
$ pip install . --user
Processing /Users/filipe/Documents/Teaching/Advanced Scientific Programming with
Python/python-course/day4-bestpractices-2/code/uu-fibonacci
Installing collected packages: uu-fibonacci
  Running setup.py install for uu-fibonacci ... done
Successfully installed uu-fibonacci-1.0
```

Publishing on PyPI

- First lets create a distribution file we can upload

```
$ python setup.py sdist
…
$ ls dist/
uu-fibonacci-1.0.tar.gz

$ twine register dist/uu-fibonacci-1.0.tar.gz
Registering package to https://pypi.python.org/pypi
Registering uu-fibonacci-1.0.tar.gz

$ twine upload dist/uu-fibonacci-1.0.tar.gz
Uploading distributions to https://pypi.python.org/pypi
Uploading uu-fibonacci-1.0.tar.gz
[==============================] 3592/3592 - 00:00:01
```

Testing our Package

```
$ pip uninstall uu-fibonacci
Uninstalling uu-fibonacci-1.0:
  /Users/filipe/Library/Python/2.7/lib/python/site-packages/fib.py
  /Users/filipe/Library/Python/2.7/lib/python/site-packages/fib.pyc
  /Users/filipe/Library/Python/2.7/lib/python/site-packages/uu_fibonacci-1.0-
py2.7.egg-info
Proceed (y/n)? y
  Successfully uninstalled uu-fibonacci-1.0

$ pip install uu-fibonacci --user
Collecting uu-fibonacci
  Downloading uu-fibonacci-1.0.tar.gz
Installing collected packages: uu-fibonacci
  Running setup.py install for uu-fibonacci ... done
Successfully installed uu-fibonacci-1.0
```

Testing our Package

```
$ ipython2
In [1]: import fib

In [2]: fib.fib(2)
Out[2]: 3

In [3]: fib.fib?
Signature: fib.fib(n)
Docstring:
Return the first Fibonacci number above n.

Iteratively calculate Fibonacci numbers until it finds one
greater than n, which it then returns.

Parameters
----------
n : integer
  The minimum threshold for the desired Fibonacci number.

Returns
-------
b : integer
  The first Fibonacci number greater than the input, `n`.

Examples
--------
>>> fib.fib(1)
2
>>> fib.fib(3)
5
File:      ~/Library/Python/2.7/lib/python/site-packages/fib.py
Type:      function
```

It works!

More About Packaging

- There's much more to learn about packaging
- I would suggest a thorough reading of :

`https://packaging.python.org/distributing/`