Some Mathematical Python

Robert Parini

University of York

7th of December 2016
Durham University

Why Python for Mathematics?

- A general purpose programming language with a readable, flexible syntax
- An ecosystem of scientific computing packages featuring wrappings to fast C/Fortran algorithms
- Can to connect to other languages (eg. C or C++ using Cython)
- Can be run in a Mathematica/Maple like notebook environment (Jupyter)
- Widely used, free and open source

Installation

- Easiest method is to use the Anaconda distribution for Python 3.5 from continuum.io/downloads
- Alternatively download Python 3.5 from python.org then use the included Python package index (pip) to download and install packages and their dependencies

```
pip3 install scipy numpy matplotlib sympy jupyter cython
```

Readability $\propto$ (number of $\{\}$)$^{-1}$

Python

```python
print('Hello World!')
```

C++

```cpp
#include <iostream>
int main()
{
  std::cout << "Hello World!";
}
```

Java

```java
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

A basic example: The Sieve of Eratosthenes

Let's find all the primes $\leq n$ using the Sieve of Eratosthenes

Some pseudocode from the wikipedia page on the Sieve of Eratosthenes:

Input: an integer $n > 1$
Let A be an **array of Boolean values**, indexed by **integers** 2 to n, initially all **set** to **true**.

for $i = 2, 3, 4, ...$, not exceeding $\sqrt{n}$:
  if A[i] **is true**:
    for j $= i^2$, $i^2 + i$, $i^2 + 2i$, $i^2 + 3i$, ..., not exceeding n:
      A[j] $:=$ **false**

Output: all i such that A[i] **is true**.

This crosses off all multiples of each prime i starting from i^2 since any smaller multiples will have another factor $< i$ and have therefore already been crossed off.

Input: an integer $n > 1$
Let A be an **array of Boolean values**, indexed by **integers** 2 to n, initially all **set** to **true**.

for $i = 2, 3, 4, \ldots$, not exceeding $\sqrt{n}$:
  if A[i] **is true**:
    for j $= i^2$, $i^2 + i$, $i^2 + 2i$, $i^2 + 3i$, ..., not exceeding n:
      A[j] := **false**

Output: all i such that A[i] **is true**.

```python
from math import sqrt
def sieve_of_eratosthenes(n):
    """ Returns a list of all primes not exceeding n """
    A = [True]*(n+1)
    for i in range(2, int(sqrt(n))+1):
        if A[i] == True:
            for j in range(i**2, n+1, i):
                A[j] = False

    return [i for i in range(2,n+1) if A[i] == True]
```

A basic example: The Sieve of Eratosthenes

```python
from math import sqrt

def sieve_of_eratosthenes(n):
    """ Returns a list of all primes not exceeding n """
    A = [True]*(n+1)

    for i in range(2, int(sqrt(n))+1):
        if A[i] == True:
            for j in range(i**2, n+1, i):
                A[j] = False

    return [i for i in range(2,n+1) if A[i] == True]

print(sieve_of_eratosthenes(47))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

A basic example: The Sieve of Eratosthenes

A minimal amount of work allows *n* to be a floating point number or an integer:

```
from math import sqrt

def sieve_of_eratosthenes(n):
    """ Returns a list of all primes not exceeding n """
    n = int(n) # cast n to an integer by truncation
    A = [True]*(n+1)

    for i in range(2, int(sqrt(n))+1):
        if A[i] == True:
            for j in range(i**2, n+1, i):
                A[j] = False

    return [i for i in range(2,n+1) if A[i] == True]

print(sieve_of_eratosthenes(47.6))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Built-in 'math' library

$$u(x,t) = 4\arctan\left(\exp\left[\cosh(\theta)x - \sinh(\theta)t\right]\right)$$

```
from math import exp, cosh, sinh
from math import atan as arctan

def soliton(x,t,theta):
    """
    One soliton solution of the sine-Gordon equation:
    u_{tt} - u_{xx} + sin(u) = 0
    with soliton rapidity theta
    """
    z = cosh(theta)*x - sinh(theta)*t
    return 4*arctan(exp(z))
```

Works for single values of x, t

```
print(soliton(1, 2, 0.2))
```

```
4.3046727972
```

NumPy

Pass a list of points and get a list in return

A loop in pure Python:

```
u = []
for x in [-1,-0.5,0,0.5,1]:
    u.append(soliton(x,2,0.2))
```

NumPy

Pass a list of points and get a list in return

A loop in pure Python:

```
u = []
for x in [-1,-0.5,0,0.5,1]:
    u.append(soliton(x,2,0.2))
```

Array broadcasting with NumPy:

```
import numpy as np
from numpy import exp, cosh, sinh, arctan
x = np.array([-1,-0.5,0,0.5,1], dtype='float64')
u = soliton(x,2,0.2)
```

- Many of NumPy's internal operations are implemented in C.
- NumPy takes advantage of array elements being homogeneous in type.

NumPy: Multidimensional Arrays

Create a multidimensional array from a function of the indices:

```
import numpy as np
def indexFunc(i,j,k):
        return k+(i+j)**2

A = np.fromfunction(indexFunc, shape=(2,3,4), dtype=int)
```

```
[[[ 0  1  2  3]   [[ 1  2  3  4]
  [ 1  2  3  4]    [ 4  5  6  7]
  [ 4  5  6  7]]   [ 9 10 11 12]]]
```

Slice arrays to get sub arrays

```
A[1]
```

```
[[ 1  2  3  4]
 [ 4  5  6  7]
 [ 9 10 11 12]]
```

```
A[1,:,1:3]
```

```
[[ 2  3]
 [ 5  6]
 [10 11]]
```

```
A[:,:,0]
```

```
[[0 1 4]
 [1 4 9]]
```

```
A[1,1:,0]
```

```
[4 9]
```

NumPy: Matrix operations

```
A = np.random.rand(3,3)     # create a random 3 by 3 matrix
```

```
array([[ 0.96480348,  0.57255811,  0.00281168],
       [ 0.23962651,  0.30004438,  0.16600863],
       [ 0.81472991,  0.56607295,  0.91356706]])
```

NumPy: Matrix operations

```
A = np.random.rand(3,3)     # create a random 3 by 3 matrix
```

```
array([[ 0.96480348,  0.57255811,  0.00281168],
       [ 0.23962651,  0.30004438,  0.16600863],
       [ 0.81472991,  0.56607295,  0.91356706]])
```

```
B = np.linalg.inv(A)        # invert
```

```
array([[ 1.43433403, -4.15223832,  0.75010861],
       [-0.66615854,  6.99994906, -1.26994394],
       [-0.86638468, -0.63435853,  1.21254918]])
```

NumPy: Matrix operations

```
A = np.random.rand(3,3)      # create a random 3 by 3 matrix
```

```
array([[ 0.96480348,  0.57255811,  0.00281168],
       [ 0.23962651,  0.30004438,  0.16600863],
       [ 0.81472991,  0.56607295,  0.91356706]])
```

```
B = np.linalg.inv(A)         # invert
```

```
array([[ 1.43433403, -4.15223832,  0.75010861],
       [-0.66615854,  6.99994906, -1.26994394],
       [-0.86638468, -0.63435853,  1.21254918]])
```

```
A.dot(B)                     # matrix multiplication
```

```
array([[  1.00000000e+00,   1.73038667e-16,  -1.11022302e-16],
       [ -5.55111512e-17,   1.00000000e+00,  -2.77555756e-17],
       [  0.00000000e+00,  -2.22044605e-16,   1.00000000e+00]])
```

NumPy: Matrix operations

```
A = np.random.rand(3,3)      # create a random 3 by 3 matrix
```

```
array([[ 0.96480348,  0.57255811,  0.00281168],
       [ 0.23962651,  0.30004438,  0.16600863],
       [ 0.81472991,  0.56607295,  0.91356706]])
```

```
B = np.linalg.inv(A)         # invert
```

```
array([[ 1.43433403, -4.15223832,  0.75010861],
       [-0.66615854,  6.99994906, -1.26994394],
       [-0.86638468, -0.63435853,  1.21254918]])
```

```
A.dot(B)                     # matrix multiplication
```

```
array([[  1.00000000e+00,   1.73038667e-16,  -1.11022302e-16],
       [ -5.55111512e-17,   1.00000000e+00,  -2.77555756e-17],
       [  0.00000000e+00,  -2.22044605e-16,   1.00000000e+00]])
```

Find eigenvalues and eigenvectors and check $(A - I\lambda)v = 0$

```
eigenvalues, eigenvectors = np.linalg.eig(A)
(A - np.identity(3)*eigenvalues[0]).dot(eigenvectors[:,0])
```

```
array([  1.27675648e-15,   1.44328993e-15,   5.55111512e-16])
```

Scipy

Contains a lot of general purpose mathematical routines including special functions, integration routines, optimisation, Fourier transforms, interpolation, ect.

```
import scipy.integrate as integrate
from scipy import exp, inf, pi, sqrt
def integrand(x):
    """ The Gaussian function """
    return exp(-x**2)
integral, err = integrate.quad(integrand, -inf, inf)
print(integral**2) # 3.141592653589 7927
```

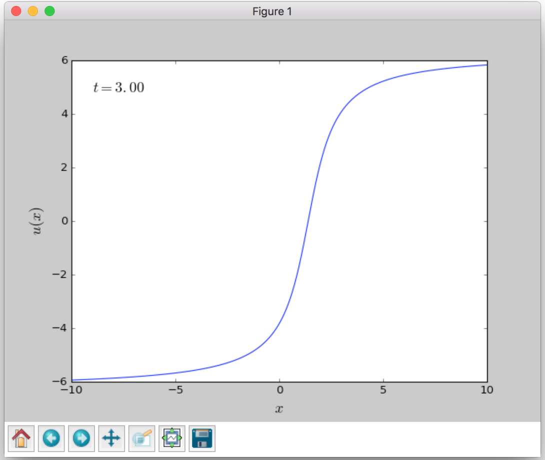
Matplotlib: Create plot

```
x  = np.linspace(-10,10,1001)
t  = 3
th = 0.2
u = soliton(x, t, th)

import matplotlib.pyplot as plt
plt.plot(x,u)
axis = plt.gca()
plt.xlabel('$x$', size=16)                    # works with LaTeX!
plt.ylabel('$u(x)$', size=16)
plt.text(0.05, 0.9, '$t= %.2f$' % t,
        transform=axis.transAxes, size=16)
```

Matplotlib: Show plot

Show plot directly:

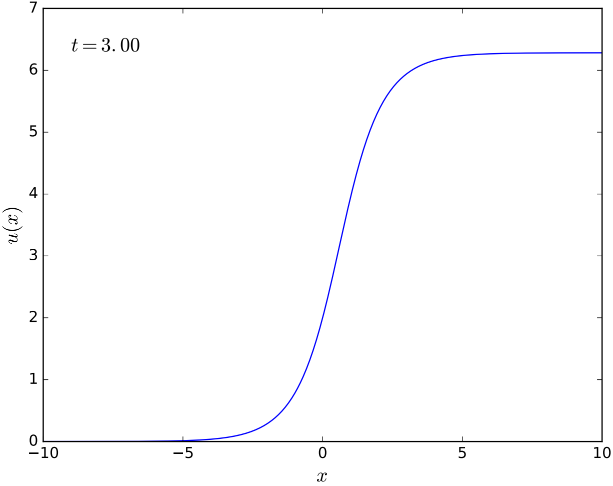
```
plt.show()
```



Matplotlib: Save plot

Or save to file:

```
plt.savefig('soliton.pdf')
```



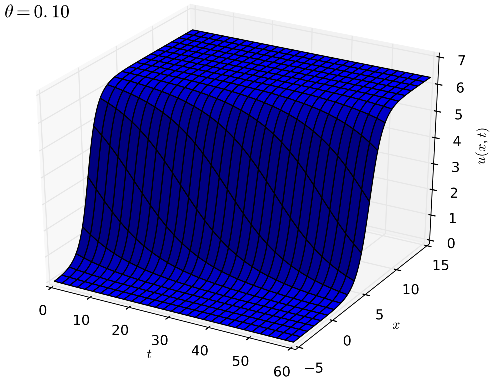
Matplotlib: Plot surface

```python
def soliton_surface_fig(x,t,th):
    """ Return figure with the soliton function plotted as
    a surface against the 1D arrays x and t """
    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D

    # set up the 3D axis
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.set_xlabel('$t$')
    ax.set_ylabel('$x$')
    ax.set_zlabel('$u(x,t)$')
    ax.text2D(0.05, 0.9, r'$\theta= %.2f$' % th,
        transform=ax.transAxes, size=16)

    # T[i,j] = t[i], X[i,j] = x[j]
    T, X = np.meshgrid(t,x, indexing = 'ij')
    ax.plot_surface(T, X, soliton(X,T,th))
    return fig
```

Matplotlib: Plot surface

As before, either use `plt.show()` to bring up an interactive plot or save directly

```
x = np.linspace(-5,15,201)
t = np.linspace(0,60,301)

fig = soliton_surface_fig(x, t, th=0.1)
fig.savefig('SolitonSurface.pdf')
```



Matplotlib: Animate

```
def soliton_animation(x, theta, t0, tFin=None, dt=1e-1):
        """
        Get animation function which plots a soliton with
        rapidity theta over 1D array x from time t0 to tFin
        with time step dt
        """
        from matplotlib import pyplot as plt
        from matplotlib import animation
        fig = plt.figure()
        ax = fig.gca()

        # set up the plot at t=t0
        plt.xlabel('$x$', size=16)
        plt.ylabel('$u(x)$', size=16)
        u0 = soliton(x,t0,theta)
        line, = plt.plot(x, u0)
        tLabel = plt.text(0.05, 0.9, '',
                transform=axis.transAxes, size=16)
        ...
```

Matplotlib: Animate

```python
def soliton_animation(x, theta, t0, tFin=None, dt=1e-1):

        ...

        # define how the plot updates every frame
        def update_plot(i):
                # update time and time label
                t = t0 + dt*i
                tLabel.set_text('$t= %.2f$' % t)

                # update line
                u = soliton(x,t,theta)
                line.set_data(x, u)

        ...
```

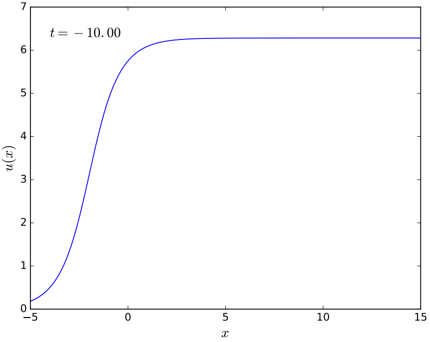
Matplotlib: Animate

```
def soliton_animation(x, theta, t0, tFin=None, dt=1e-1):

    ...

    # define the number of frames to be drawn
    if tFin is None:
        frames = None
    else:
        frames = int((tFin-t0)/dt)

    # call the animating function which will draw a new frame
    # every 'interval' milliseconds
    anim = animation.FuncAnimation(fig, update_plot, frames,
        interval=1)
    return anim
```

Matplotlib: Animate

Use `plt.show()` or save the animation to disk

```
x, theta, t0 = np.linspace(-5,15,1001), 0.2, -10

anim = soliton_animation(x, theta, t0, tFin=60)
writer = animation.FFMpegWriter(bitrate=1000, fps=60)
anim.save('solitonAnimation.mov', writer=writer)
```



Symbolic computation with SymPy and Jupyter

Open with: `jupyter notebook SymbolicEx.ipynb`

Interactive animation and generators

Conway's Game of Life

```python
import numpy as np

def life_step(state):
    """ Takes a step in Conway's game of life on a torus """
    # Compute the number of alive cells adjacent (including
    # diagonals) to each cell
    # Rolling matricies is periodic so this implements PBC
    neighbours = sum(np.roll(np.roll(state, i, axis=0), j, axis=1)
        for i in (-1,0,1) for j in (-1,0,1) if (i != 0 or j != 0))

    # Any live cell with fewer than two live neighbours dies
    state = np.where(neighbours < 2, 0, state)
    # Any live cell with more than three live neighbours dies
    state = np.where(neighbours > 3, 0, state)
    # Any dead cell with exactly three live neighbours is born
    state = np.where(neighbours == 3, 1, state)

    return state
```

Conway's Game of Life: Generators

```python
def life_generator(initialState):
    state = initialState
    while True:
        yield state
        state = life_step(state)

X = np.zeros((4,5))
X[2,1:4] = 1
life = life_generator(X)
```

```python
next(life) # 1st (=X)
```

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  1.  1.  1.]
 [ 0.  0.  0.  0.]]
```

```python
next(life) # 2nd
```

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]]
```

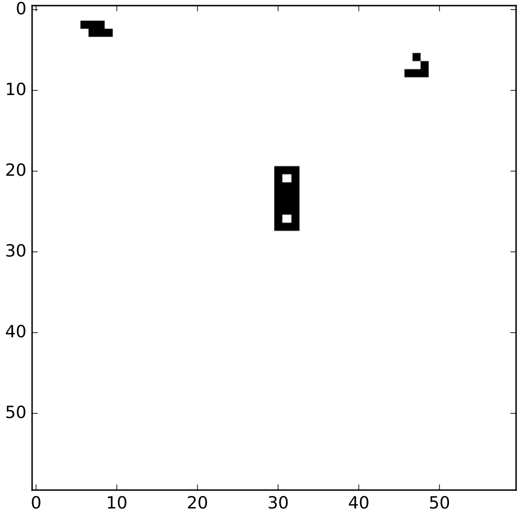
```python
next(life) # 3rd
```

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  1.  1.  1.]
 [ 0.  0.  0.  0.]]
```

Conway's Game of Life: Animating an image

```
from matplotlib import pyplot as plt
import matplotlib.animation as animation

def game_of_life(initialState):
    # cmap='Greys' maps values in [0,1] to colours
    # between ['white', 'black']
    image = plt.imshow(initialState, interpolation = 'nearest',
        cmap='Greys')
    life = life_generator(initialState)

    def update_image(state):
        image.set_array(state)

    fig = plt.gcf()
    # pass frames a generator which will in turn pass to
    # update_image each state
    ani = animation.FuncAnimation(fig, update_image, frames=life,
        interval=300)
    plt.show()
```

Conway's Game of Life: Animating an image



Conway's Game of Life: Passing values to generators

```
def life_generator(initialState):
    state = initialState
    while True:
        passed_state = yield state
        if passed_state is None:
            state = life_step(state)
        else:
            state = passed_state
```

```
life = life_generator(X)
Y = np.array([[0, 1, 0, 0],
              [0, 0, 1, 0],
              [1, 1, 1, 0],
              [0, 0, 0, 0]])
```

```
life.send(Y) # 1st (=Y)
```

```
[[0 1 0 0]
 [0 0 1 0]
 [1 1 1 0]
 [0 0 0 0]]
```

```
next(life) # 2nd
```

```
[[0 0 0 0]
 [1 0 1 1]
 [0 1 1 1]
 [1 0 1 0]]
```

Conway's Game of Life: Adding interactivity

```
def interactive_game_of_life(initialState):
    ...

    def on_click(event):
        if event.button == 1:
            # on left click
            x, y = event.xdata, event.ydata

            if x is not None and y is not None:
                x, y = int(round(x)), int(round(y))

                # flip selected tile
                state = image.get_array()
                state[y,x] = (state[y,x] + 1) % 2
                life.send(state)

    fig.canvas.mpl_connect('button_press_event', on_click)

    ...
```

Conway's Game of Life: Adding Pause/Unpause

```
def life_generator(initialState):
    state = initialState
    paused = False
    while True:
        passedVal = yield state
        if passedVal is None:
            if not paused:
                state = life_step(state)
        elif passedVal == 'toggle pause':
            paused = not paused
        else:
            state = passedVal
```

Now we can pause the generator with

```
life = life_generator(X)
life.send('toggle pause')
```

Conway's Game of Life: Adding Pause/Unpause

```
def interactive_game_of_life(initialState):
    ...

    pauseLabel = plt.text(0.42, 1.02, '',
        transform=plt.gca().transAxes, size=16)

    def on_keyPress(event):
        if event.key == ' ':
            # space pauses the animation
            life.send('toggle pause')
            if pauseLabel.get_text() == '':
                pauseLabel.set_text('Paused')
            else:
                pauseLabel.set_text('')

    fig.canvas.mpl_connect('key_press_event', on_keyPress)

    ...
```

Conway's Interactive Game of Life

Saving/loading NumPy arrays

```
def interactive_game_of_life(initialState)
    ...
    # disable some default matplotlib shortcuts
    plt.rcParams['keymap.save'] = ''
    plt.rcParams['keymap.yscale'] = ''

    def on_keyPress(event):
        ...
        saveFile = 'life.npy'
        if event.key == 's':
            print('Saving to %s' % saveFile)
            image.get_array().dump(saveFile)
            print('Saved')
        elif event.key == 'l':
            print('Loading %s' % saveFile)
            loadedState = np.load(saveFile)
            life.send(loadedState)
            print('Loaded')
        ...
    ...
```

Save/Load Game of Life

Speeding up Python

Cython: typing

```
### CyPrimes.pyx
from math import sqrt
def sieve_of_eratosthenes_inti( int n):
    cdef int i, j

    A = [True]*(n+1)
    for i in range(2, int(sqrt(n))+1):
        if A[i] == True:
            for j in range(i**2, n+1, i):
                A[j] = False

    return [i for i in range(2,n+1) if A[i] == True]
```

```
### SomePythonScript.py
import pyximport
pyximport.install()
from CyPrimes import sieve_of_eratosthenes_inti
sieve_of_eratosthenes_inti(47)
```

Cython: assigning memory

```
### CyPrimes.pyx
from cpython.mem cimport PyMem_Malloc, PyMem_Realloc, PyMem_Free
def sieve_of_eratosthenes_malloc(int n):
    # manually allocate (n+1)*sizeof(bint) bytes of memory
    cdef bint* A = <bint *>PyMem_Malloc((n+1) * sizeof(bint))
    cdef int i, j
    if not A:
        raise MemoryError()
    try:
        for i in range(2,n+1):
            A[i] = True
        for i in range(2, int(sqrt(n))+1):
            if A[i] == True:
                for j in range(i**2, n+1, i):
                    A[j] = False
        return [i for i in range(2,n+1) if A[i] == True]
    finally:
        # free allocated memory after use
        PyMem_Free(A)
```

Cython: importing from C++

```cpp
### CppPrimes.cpp
std::vector<int> sieve_of_eratosthenes(int n){
    std::vector<int> primes;
    primes.reserve(n/2);
    std::vector<bool> A(n+1, true);
    for (int i = 2; i <= (int)sqrt(n); i++) {
        if (A[i]) {
            // i is a prime so append it to primes
            primes.push_back(i);
            for (int j = i*i; j <= n; j += i) {
                A[j] = false;
    }}}
    // return indices from sqrt(n) to n that are primes
    // these were not already covered in the original loop
    for (int i = (int)sqrt(n)+1; i <= n; i++) {
        if (A[i]) {
            primes.push_back(i);
    }}
    return primes;
}
```

Cython: importing from C++

```
### CppPrimes_caller.pyx
from libcpp.vector cimport vector

# import the C++ function 'sieve_of_eratosthenes'
# from the file "CppPrimes.cpp"
cdef extern from "CppPrimes.cpp":
    vector[int] sieve_of_eratosthenes(int n)

# wrap the C++ function
def sieve_of_eratosthenes_cpp(int n):
    return sieve_of_eratosthenes(n)
```

Cython: importing from C++

```python
### CppPrimesSetup.py
# Compiles CppPrimes_caller.pyx -> CppPrimes_caller.so
# python CppPrimesSetup.py build_ext --inplace

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    name = 'CppPrimes',
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("CppPrimes_caller",
        ["CppPrimes_caller.pyx"],
        language = "c++"
        )]
)
```

Cython

```python
### TimingPrimes.py
# How long does it take to get all the primes up to 1 million?
import timeit
repeat = 1000
min(timeit.repeat(stmt='sieve_of_eratosthenes(1000000)',
    setup='from Primes import sieve_of_eratosthenes',
    number=1, repeat=repeat))
```

```
Original time       0.20030007600144017

Typing i,j time     0.10809241200331599 # 1.8x faster

Malloc time         0.05655720400682185 # 3.5x faster

C++ -> Python time  0.007263979001436382 5 # 28x faster
```

NumPy

```
from math import sqrt
import numpy as np
def sieve_of_eratosthenes_np(n):
    n = int(n)
    A = np.ones(n+1, dtype=bool)
    A[:2] = False # 0 and 1 are not primes

    for i in range(2, int(sqrt(n))+1):
        if A[i] == True:
            A[i**2:n+1:i] = False

    return np.nonzero(A)[0]
```

```
repeat = 1000
min(timeit.repeat(stmt='sieve_of_eratosthenes_np(1000000)',
    setup='from NumpyPrimes import sieve_of_eratosthenes_np',
    number=1, repeat=repeat))
```

```
NumPy time: 0.005090902006486431 # 39x faster than original!
```

A few things I didn't mention

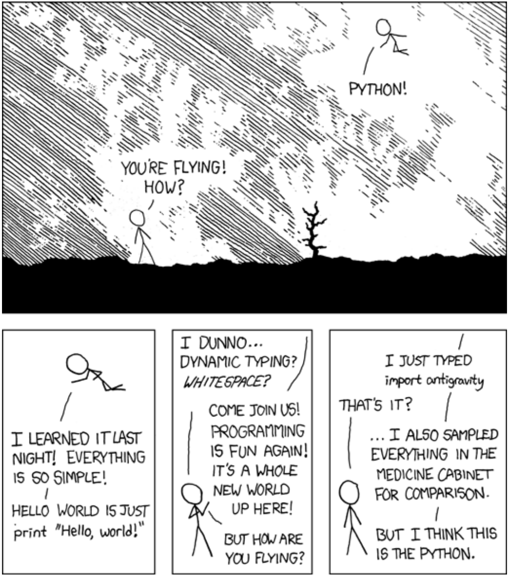
Aspects of Python

- Classes
- Multiprocessing

Other packages

- mpmath - arbitrary precision floating point arithmetic with many special functions and some common routines such as integration and rootfinding
- Pandas - data analysis
- SageMath - algebraic computing software (not really a package!)

Thank you!


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.