numpy and scipy (1/2)

Computing Methods for Experimental Physics and Data Analysis

L. Baldini
luca.baldini@pi.infn.it

Università and INFN–Pisa

Compiled on October 13, 2019

Introduction

- Python is notorious for coming *with batteries included*. . .
- . . . but as a data scientist numpy and and scipy will be your best friends!
- Among (many) other things, numpy offers:
  - a powerful n-dimensional array object
  - mathematical functions that interoperate natively with arrays
- And scipy provides:
  - integration
  - optimization (a.k.a. fitting)
  - interpolation
  - signal processing

numpy arrays

https://bitbucket.org/lbaldini/programming/src/tip/snippets/numpy_arrays.py

```
import numpy as np

# Initialization from a list
a1 = np.array([1., 2., 3])
print(a1)

# Zeros, ones, and fixed values
a2 = np.zeros(10)
a3 = np.ones((2, 2))
a4 = np.full(7, 3.)
print(a2)
print(a3)
print(a4)

# Grids
a5 = np.linspace(0., 10., 11)
a6 = np.logspace(0., 1., 11)
print(a5)
print(a6)

[Output]
[1. 2. 3.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[[1. 1.]
 [1. 1.]]
[3. 3. 3. 3. 3. 3. 3.]
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[ 1.          1.25892541  1.58489319  1.99526231  2.51188643  3.16227766
  3.98107171  5.01187234  6.30957344  7.94328235 10.        ]
```

https://bitbucket.org/lbaldini/programming/src/tip/snippets/numpy_arrays_vs_lists.py

```python
import numpy as np

# arrays and lists seem similar...
l = [1., 2., 3.]
a = np.array(l)
print(l)
print(a)

# ...but they support basic arithmetic in a different fashion
print(l + l)
print(a + a)

[Output]
[1.0, 2.0, 3.0]
[1. 2. 3.]
[1.0, 2.0, 3.0, 1.0, 2.0, 3.0]
[2. 4. 6.]
```

- arrays and lists are fundamentally different objects
    - different footprint in memory, operate at different speed
    - arrays are homogeneous, lists don't need to
    - arrays offer a much more powerful indexing/slicing
    - arrays interoperate with numpy mathematical functions

Broadcasting

https://bitbucket.org/lbaldini/programming/src/tip/snippets/numpy_arrays_broadcasting.py

```
import numpy as np

a1 = np.array([1., 2.])
a2 = np.array([[1., 2.], [3., 4.]])
c = np.pi

print(a1)
print(a2)
print(c)
print(a1 * a1)
print(a1 * c)
print(a1 * a2)

[Output]
[1. 2.]
[[1. 2.]
 [3. 4.]]
3.141592653589793
[1. 4.]
[3.14159265 6.28318531]
[[1. 4.]
 [3. 8.]]
```

- Under certain conditions, numpy can make operations on arrays of different shape
- This is extremely useful when vectorizing problems

Mathematical functions in numpy

https://bitbucket.org/lbaldini/programming/src/tip/snippets/numpy_functions.py

```
import numpy as np
import math

a = np.array([0.1, 1., 10.])

print(np.log10(a))
print(np.exp(a))
print(np.sin(a))

print(np.log10(0.1))

print(math.log10(a))

[Output]
[-1.  0.  1.]
[1.10517092e+00 2.71828183e+00 2.20264658e+04]
[ 0.09983342  0.84147098 -0.54402111]
-1.0
Traceback (most recent call last):
  File "snippets/numpy_functions.py", line 12, in <module>
    print(math.log10(a))
TypeError: only size-1 arrays can be converted to Python scalars
```

- numpy mathematical functions interoperate natively with arrays
  - (and work on plain old numbers, too)

Digression: pseudo-random number generators

```python
import random
x = random.random()
```

- Every programming language comes with a Pseudo Random Number Generator (PRNG)
  - Python is no exception: `https://docs.python.org/3/library/random.html`
  - Mersenne-Twister, 53-bit precision, period of $2^{19937} - 1$.
- PRNGs are an interesting (and fun) subject by themselves:
  - Donald E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd Edition
  - M. Matsumoto and T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.
- A PRNG produces random floats uniformly in [0.0, 1.0).

Vectorization

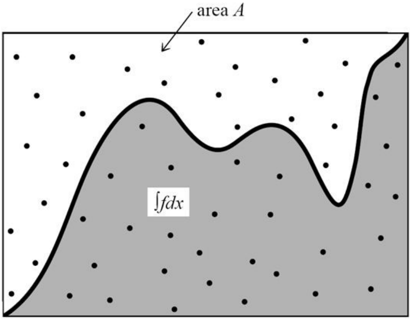
a.k.a. avoid explicit for loops in Python whenever you can

https://bitbucket.org/lbaldini/programming/src/tip/snippets/vectorization.py

```python
import random
import time
import numpy as np

# How many random numbers (uniformly distributed between 0 and 1) do you
# want to throw?
n = 1000000

# The slow way: explicit for loop in Python.
t0 = time.time()
x = []
for i in range(n):
    x.append(random.random())
dt = time.time() - t0
print('Elapsed time: %.3f s' % dt)

# The quick way: vectorizing in numpy
t0 = time.time()
x = np.random.random(size=n)
dt = time.time() - t0
print('Elapsed time: %.3f s' % dt)

[Output]
Elapsed time: 0.131 s
Elapsed time: 0.015 s
```

How does vectorizaion work?

- Python is know to be slooow
  - This is the price you pay for being so beautiful and flexible
- Does it matter? If depends. . .
  - If you are parsing a text file or fetching a web page probably not
  - If you are performing a CPU-intensive processing on a TB of data probably yes
- What's so magic in using numpy?
- numpy is written in C as a Python extension
  - Routines are highly optimized to cruch numbers
  - When you perform an array operation in Python you are actually executing optimized C code
- Basic message: avoid for loops in pure Python when crunching numbers



- Hit or miss, aka acceptance/rejection method:
  - Enclose your pdf in a rectangle
  - Throw a x and a y
  - Accept x if $y \leq f(x)$
- This is horrible—please don't use it!

- Probability density function (pdf)

$$p(x) \quad (\geq 0)$$

- Cumulative function (cf)

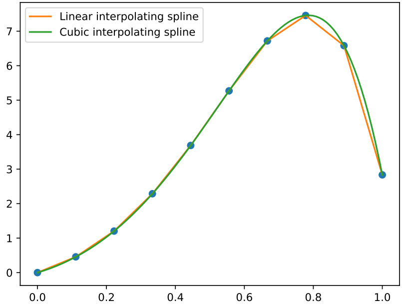
$$F(x) = \int_{-\infty}^{x} p(x')dx'$$

- Percent-point function (ppf)

$$x = F^{-1}(q)$$

- Awesome fact: if q is uniformly distributed in $[0, 1]$, then $x = F^{-1}(q)$ is distributed according to $p(x)$!

An interesting object: splines



- Defined piecewsise by polinomials of degree k ($k = 3$ fairly popular)
  - Interpolating: passing through a set of pre-defined points
  - First $k - 1$ derivatives continuos at the control points
- Superior to polynomial interpolation or curve fitting in many cases

Splines: construction and properties

https://bitbucket.org/lbaldini/programming/src/tip/snippets/spline.py

```python
import numpy as np
from scipy.interpolate import InterpolatedUnivariateSpline

x = np.linspace(0., 1., 10)
y = np.exp(3. * x) * np.sin(3. * x)

s1 = InterpolatedUnivariateSpline(x, y, k=1)
s3 = InterpolatedUnivariateSpline(x, y, k=3)

print(s1(0.234))
print(s1.integral(0.2, 0.8))

[Output]
1.3192110648078448
2.6659857771053925
```

- Evaluation is fairly inexpensive
  - If the input *x*-array is sorted can do a binary search in O(log(N)) complexity
- Derivatives and integrals are easy
  - Can be calculated *exactly* by means of elementary arithmetic operations

References

- ▷ https://numpy.org/
- ▷ https://www.scipy.org/
- ▷ https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html
- ▷ https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
- ▷ https://docs.scipy.org/doc/scipy/reference/interpolate.html