Introduction to Python Day 4

Verjinia Metodieva and Daniel Parthier

2025-03-04

NumPy

- Python library numerical data
- Very fast (*C/C++* and multithreaded)
- Vectorized (substitutes `for`-loops)
- Good short cut for a lot of things
- Will be your best friend!

NumPy Structures

- similar to what we know already

Basic

- Different data types (`dtype`) are valid
- `type` followed by `bit` (8, 16, 32, 64, 128)

  1. float64
  2. int64
  3. complex64
  4. bool_
  5. str_
  6. object_
  7. datetime64

Matrix

- Matrix for matrix calculations

```
import numpy as np
new_matrix = np.matrix([[1,2,3],[4,5,6]])
new_matrix
```

```
matrix([[1, 2, 3],
        [4, 5, 6]])
```

Array

- 1D, 2D and nD
  - Matrices are a kind of array with special functions
- Different ways to make them
- Have shape properties

Array

- Convert an existing list to an array

```
list_input = [1,2,3]
new_array = np.array(list_input)
new_array
```

```
array([1, 2, 3])
```

- Can also be reversed with *method*

```
new_array.tolist()
```

```
[1, 2, 3]
```

Make filled Arrays

- Array with zeros

```
np.zeros(5)
```

```
array([0., 0., 0., 0., 0.])
```

- Arrays with ones

```
np.ones(5)
```

```
array([1., 1., 1., 1., 1.])
```

- Arrays filled with what you want

```
np.full(5, "hello")
```

```
array(['hello', 'hello', 'hello', 'hello', 'hello'], dtype='<U5')
```

Array initiation short-cut

- Use the size/shape of another array

```
old_array = np.array([1,2,3,4])
np.zeros_like(old_array)
```

```
array([0, 0, 0, 0])
```

n-dimensional Arrays

- Every initation of Arrays can be multidimensional

2D

```
np.ones((3,5))
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

3D

```
np.ones((3,3,2))
```

```
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

Special cases for Arrays

- Arrays can also be "empty"

```
np.empty(10)
```

```
array([ 4.68366755e-310,  0.00000000e+000,  6.93340905e-310,
        1.27305173e-158,  2.37151510e-322,  2.37151510e-322,
        4.68366754e-310,  0.00000000e+000,  8.95114257e+245,
       -3.87685713e+045])
```

- Only useful in very specific cases (otherwise danger zone)

Array Sequences

- Generate sequences

```
np.arange(start=2, stop=10, step=2)
```

```
array([2, 4, 6, 8])
```

- Similar logic to iterators from day 1 (`2:10:2`)
- Array can also go in steps of floats e.g. `0.2`

Array Sequences

- Alternative `linspace` and `logspace`

```
np.linspace(start=2, stop=10, num=6)
np.logspace(start=2, stop=10, num=6)
```

```
array([ 2. ,  3.6,  5.2,  6.8,  8.4, 10. ])
```

```
array([1.00000000e+02, 3.98107171e+03, 1.58489319e+05, 6.30957344e+06,
       2.51188643e+08, 1.00000000e+10])
```

Array with random numbers

- Callback: `spike_simulation`
- Generate random numbers as arrays

```
np.random.seed(42)
np.random.normal(5, 2, 10)
```

```
array([5.99342831, 4.7234714 , 6.29537708, 8.04605971, 4.53169325,
       4.53172609, 8.15842563, 6.53486946, 4.06105123, 6.08512009])
```

Indexing

Special Indices

- Recall finding the maximum

```
# looping through data indices. find the max
B = [1, 4, 6, 7, 89, 54]
big_indx = 0
for i in range(len(B)):
    if B[i] > B[big_indx]:
        big_indx = i
print('The max value in B is', B[big_indx], 'found on position', big_indx)
```

```
The max value in B is 89 found on position 4
```

```
# looping through data indices. find the max
B = [1, 4, 6, 7, 89, 54]
big_indx = np.argmax(B)
print('The max value in B is', B[big_indx], 'found on position', big_indx)
```

```
The max value in B is 89 found on position 4
```

`for` loops can be often replaced using functions and make your code faster and easier to read. As you can see from the example we can also use a list as function input. Numpy will convert the list automatically, work with an array and return an array too.

Operations

- Lots of useful functions:
  - Mathematical functions
  - Linear algebra
  - Sorting and Counting
  - Statistics
  - Random number generation
  - Input/Output (I/O)
  - Memory mapping (mmap)

Mathematical function

- Vectorized functions
- Versions which handle `nan`

```
power = np.array([312, 271, 912, 851, 239, 715, np.nan])
np.sqrt(power)
```

```
array([17.66352173, 16.46207763, 30.19933774, 29.17190429, 15.45962483,
       26.73948391,         nan])
```

```
np.sum(power)
np.nansum(power)
```

```
np.float64(nan)
```

```
np.float64(3300.0)
```

Statistics

- Get some summary statistics

```
power = np.array([313, 271, 912, 851, 239, 715])
np.mean(power)
np.median(power)
np.std(power)
```

```
np.float64(550.1666666666666)
```

```
np.float64(514.0)
```

```
np.float64(282.72508240731355)
```

Functions in 2D

- Apply functions to different dimensions (axes)

```
power = np.array([[313, 271, 912, 851, 239, 715],
                  [469, 137, 312, 253, 532, 416],
                  [517, 246, 111, 321, 651, 219]])
np.mean(power, axis=0)
np.mean(power, axis=1)
```

```
array([433., 218., 445., 475., 474., 450.])
```

```
array([550.16666667, 353.16666667, 344.16666667])
```

```
np.std(power, axis=0)
np.std(power, axis=1)
```

```
array([ 87.08616423,  58.17788812, 340.26166402, 267.31753902,
       173.12615824, 203.9133803 ])
```

```
array([282.72508241, 134.04155657, 184.49247199])
```

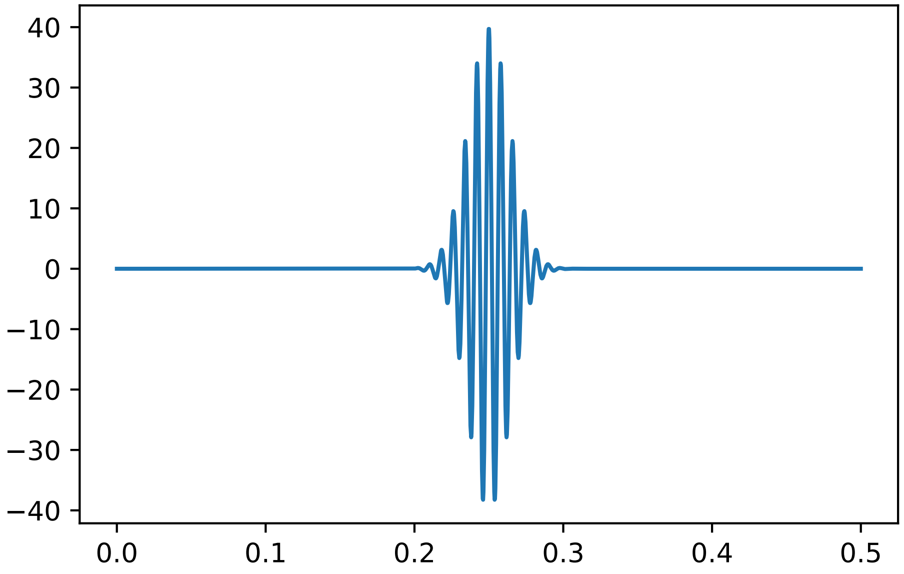
Putting things together

What could this be?

$sin(250x)\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

```
import matplotlib.pyplot as plt
time = np.linspace(start=0, stop=0.5, num=2000)
mu, sigma = 0.25, 0.01
sinewave = np.sin(time * 250 * np.pi)
gaussian = (1 / (np.sqrt(2 * np.pi * np.square(sigma))) *
            np.exp(-(np.square(time - mu) /np.square(2 * sigma))))

plt.plot(time, gaussian * sinewave)
plt.show()
```


40
30
20
10
0
−10
−20
−30
−40
0.0
0.1
0.2
0.3
0.4
0.5