Python for scientific research

Number crunching with `NumPy` and `SciPy`

Bram Kuijper

University of Exeter, Penryn Campus, UK

March 3, 2020

Speedy express

1. Declare variables using built-in data types and execute operations on them
2. Use flow control commands to dictate the order in which commands are run and when
3. Encapsulate programs into reusable functions, modules and packages
4. Use string manipulation and regex to work with textual data
5. Interact with the file system
6. **Next**: Number crunching using `NumPy/SciPy`

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers
- The `NumPy` (numeric python) package provides a comprehensive set of mathematical data structures (e.g vector, matrix) and functions (e.g trigonometry, random number generators)

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers
- The `NumPy` (numeric python) package provides a comprehensive set of mathematical data structures (e.g vector, matrix) and functions (e.g trigonometry, random number generators)
- The `SciPy` (scientific python) library builds on top of `NumPy` to provide a collection of numerical algorithms for:

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers
- The `NumPy` (numeric python) package provides a comprehensive set of mathematical data structures (e.g vector, matrix) and functions (e.g trigonometry, random number generators)
- The `SciPy` (scientific python) library builds on top of `NumPy` to provide a collection of numerical algorithms for:
    - Statistics (e.g correlation coefficients) using `scipy.stats`

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers
- The `NumPy` (numeric python) package provides a comprehensive set of mathematical data structures (e.g vector, matrix) and functions (e.g trigonometry, random number generators)
- The `SciPy` (scientific python) library builds on top of `NumPy` to provide a collection of numerical algorithms for:
  - Statistics (e.g correlation coefficients) using `scipy.stats`
  - Signal processing (e.g Fourier transform, filtering) using `scipy.fftpack` and `scipy.signal`

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers
- The `NumPy` (numeric python) package provides a comprehensive set of mathematical data structures (e.g vector, matrix) and functions (e.g trigonometry, random number generators)
- The `SciPy` (scientific python) library builds on top of `NumPy` to provide a collection of numerical algorithms for:
  - Statistics (e.g correlation coefficients) using `scipy.stats`
  - Signal processing (e.g Fourier transform, filtering) using `scipy.fftpack` and `scipy.signal`
  - Solving differential equations, using `scipy.integrate`

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers
- The `NumPy` (numeric python) package provides a comprehensive set of mathematical data structures (e.g vector, matrix) and functions (e.g trigonometry, random number generators)
- The `SciPy` (scientific python) library builds on top of `NumPy` to provide a collection of numerical algorithms for:
  - Statistics (e.g correlation coefficients) using `scipy.stats`
  - Signal processing (e.g Fourier transform, filtering) using `scipy.fftpack` and `scipy.signal`
  - Solving differential equations, using `scipy.integrate`
  - Optimisation using `scipy.optimize`

Motivation

- We are typically faced by numerical tasks, such as, computing Pearson's correlation coefficient or generating random numbers
- The `NumPy` (numeric python) package provides a comprehensive set of mathematical data structures (e.g vector, matrix) and functions (e.g trigonometry, random number generators)
- The `SciPy` (scientific python) library builds on top of `NumPy` to provide a collection of numerical algorithms for:
  - Statistics (e.g correlation coefficients) using `scipy.stats`
  - Signal processing (e.g Fourier transform, filtering) using `scipy.fftpack` and `scipy.signal`
  - Solving differential equations, using `scipy.integrate`
  - Optimisation using `scipy.optimize`
  - ...

A taste of NumPy: vectors

- Example of a NumPy vector

```python
import numpy as np

# 1D array/vector
x = np.array([1, 3, 4, 2])
x.min() # return min of array
x.max() # return max of array
x.sum() # sum all elements in array
```

A taste of NumPy: vectors

- Example of a NumPy vector

```
import numpy as np

# 1D array/vector
x = np.array([1, 3, 4, 2])
x.min() # return min of array
x.max() # return max of array
x.sum() # sum all elements in array
```

- Here, `np.array()` returns a `np.ndarray` object.

A taste of NumPy: vectors

- Example of a NumPy vector

```
import numpy as np

# 1D array/vector
x = np.array([1, 3, 4, 2])
x.min() # return min of array
x.max() # return max of array
x.sum() # sum all elements in array
```

- Here, `np.array()` returns a `np.ndarray` object.
  - For example, operations like `list.reverse()` do not work

A taste of NumPy: vectors

- Example of a `NumPy` vector

```python
import numpy as np

# 1D array/vector
x = np.array([1, 3, 4, 2])
x.min() # return min of array
x.max() # return max of array
x.sum() # sum all elements in array
```

- Here, `np.array()` returns a `np.ndarray` object.
  - For example, operations like `list.reverse()` do not work
- Easy, however, to cast `np.ndarray` to a list

A taste of NumPy: vectors

- Example of a NumPy vector

```python
import numpy as np

# 1D array/vector
x = np.array([1, 3, 4, 2])
x.min() # return min of array
x.max() # return max of array
x.sum() # sum all elements in array
```

- Here, `np.array()` returns a `np.ndarray` object.
  - For example, operations like `list.reverse()` do not work
- Easy, however, to cast `np.ndarray` to a list

```python
x_list = list(x) # [1, 3, 4, 2 ]
x_list.reverse() # now x_list is [2, 4, 3, 1]
```

A taste of NumPy: matrices and linear algebra

- Specifying the matrix:

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

A taste of NumPy: matrices and linear algebra

- Specifying the matrix:

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
import numpy as np
# 2D array (i.e., a matrix)
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

A taste of NumPy: matrices and linear algebra

- Specifying the matrix:

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
import numpy as np
# 2D array (i.e., a matrix)
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
x*x # element-by-element multiplication (Kronecker
    product)
```

- Specifying the matrix:

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
import numpy as np
# 2D array (i.e., a matrix)
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
x*x # element-by-element multiplication (Kronecker
    product)
x.shape # return dimensions of matrix
```

A taste of NumPy: matrices and linear algebra

- Specifying the matrix:

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
import numpy as np
# 2D array (i.e., a matrix)
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
x*x # element-by-element multiplication (Kronecker
    product)
x.shape # return dimensions of matrix
np.dot(x, x) # matrix multiplication (dot product)
```

A taste of NumPy: linear algebra

- Calculate eigenvalues and eigenvecturs

```
import numpy as np

# declare a 2x2 matrix
x = np.array([[1, 2], [3, 4]])
```

A taste of NumPy: linear algebra

- Calculate eigenvalues and eigenvecturs

```
import numpy as np

# declare a 2x2 matrix
x = np.array([[1, 2], [3, 4]])

# calculate eigenvalues
eival = np.linalg.eigvals(x) # [-0.37228132
    5.37228132]
```

A taste of NumPy: linear algebra

- Calculate eigenvalues and eigenvecturs

```
import numpy as np

# declare a 2x2 matrix
x = np.array([[1, 2], [3, 4]])

# calculate eigenvalues
eival = np.linalg.eigvals(x) # [-0.37228132
    5.37228132]

# calculate eigenvalues and right eigenvectors
eival, eivec = np.linalg.eig(x)
# eigenvalues (eival):
# [-0.37228132  5.37228132]
# [-0.37228132  5.37228132]
#
# eigenvectors (eivec):
# [-0.82456484 -0.41597356]
# [ 0.56576746 -0.90937671]]
```

Numpy: number sequences

- Functions to create evenly spaced sequences of numbers:

```
# 0 to 1 (but not including 1) in steps of 0.1
np.arange(0, 1, 0.1)

# 100 evenly spaced values between 0 and 1 (including
    1)
np.linspace(0, 1, 100)

# 10 evenly spaced values between 10^0 and 10^1 (log
    scale)
np.logspace(0, 1, 10)
```

Numpy: random numbers

- Random numbers using `numpy.random.default_rng`:

```python
# import the default random number generator function
from numpy.random import default_rng
```

Numpy: random numbers

- Random numbers using `numpy.random.default_rng`:

```
# import the default random number generator function
from numpy.random import default_rng

# initialize random number generator object
# optionally one can provide it with a seed:
# same random sequence is then 'replayed' each time
rng_obj = default_rng(seed=3434)
```

Numpy: random numbers

- Random numbers using `numpy.random.default_rng`:

```python
# import the default random number generator function
from numpy.random import default_rng

# initialize random number generator object
# optionally one can provide it with a seed:
# same random sequence is then 'replayed' each time
rng_obj = default_rng(seed=3434)

# uniform distribution
rng_obj.uniform(size=5)

# normal distribution
rng_obj.normal(size=10)

# random integers from 0 to 9 (in this case: returns a
    5x5 matrix)
rng_obj.integers(low=0, high = 10, size=(5,5))
```

A taste of SciPy

- Basic statistics provided by `scipy.stats`
- For more advanced statistical tests (e.g., GLM, GLMM), check out the `statsmodels` package

```
import scipy.stats as sp

# Create two random arrays
x1 = rng_obj.normal(size=30)
x2 = rng_obj.normal(size=30)

# Correlation coefficientss
sp.pearsonr(x1, x2) # pearson correlation
sp.spearmanr(x1, x2) # spearman correlation
sp.kendalltau(x1, x2) # kendall correlation

# Statistical tests
sp.ttest_ind(x1, x2) # independent t-test
sp.mannwhitneyu(x1, x2) #  Mann-Whitney rank test
sp.wilcoxon(x1, x2) # Wilcoxon signed-rank test

# Least-squares regression
sp.linregress(x1, x2)
```

Predator prey equations (Lotka Volterra)

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \alpha u - \beta uv$$

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\gamma v + \delta uv$$

Where:

- u: is the number of prey (e.g rabbits)
- v: is the number of predators (e.g foxes)
- α: prey growth rate in the absence of predators
- β: dying rate of prey due to predation
- γ: dying rate of predators in the absence of prey
- δ: predator growth rate when consuming prey

Predator prey equations in Python

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \alpha u - \beta uv$$

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\gamma v + \delta uv$$

```
def predator_prey(x, t):
    """
    Predator prey model (Lotka Volterra)
    """
    # Constants
    alpha = 1
    beta = 0.1
    gamma = 1.5
    delta = 0.075

    # x = [u, v] describes prey and predator populations
    u, v = x

    # Define differential equation (u = x[0], v = x[1])
    du = alpha*u - beta*u*v
    dv = -gamma*v + delta*u*v

    return du, dv
```

Solve differential equations

```
from scipy.integrate import odeint

time = np.linspace(0, 35, 1000) # time vector
init = [10, 5] # initial condition: 10 prey, 5 predators
x = odeint(predator_prey, init, time) # solve
```

Solve differential equations

```
from scipy.integrate import odeint

time = np.linspace(0, 35, 1000) # time vector
init = [10, 5] # initial condition: 10 prey, 5 predators
x = odeint(predator_prey, init, time) # solve
# returns 2-dimensional np.array():
#[[10.           5.          ]
# [10.17902694  4.87146722]
# [10.36582394  4.74852012]
# ...
# [36.19731528  5.37939637]
# [36.77291203  5.61756101]
# [37.32525798  5.87497465]]
```

Solve differential equations

```
from scipy.integrate import odeint

time = np.linspace(0, 35, 1000) # time vector
init = [10, 5] # initial condition: 10 prey, 5 predators
x = odeint(predator_prey, init, time) # solve
# returns 2-dimensional np.array():
#[[10.           5.         ]
# [10.17902694  4.87146722]
# [10.36582394  4.74852012]
# ...
# [36.19731528  5.37939637]
# [36.77291203  5.61756101]
# [37.32525798  5.87497465]]

# open a matplotlib figure
import matplotlib.pyplot as plt
```

Solve differential equations

```
from scipy.integrate import odeint

time = np.linspace(0, 35, 1000) # time vector
init = [10, 5] # initial condition: 10 prey, 5 predators
x = odeint(predator_prey, init, time) # solve
# returns 2-dimensional np.array():
#[[10.            5.          ]
# [10.17902694   4.87146722]
# [10.36582394   4.74852012]
# ...
# [36.19731528   5.37939637]
# [36.77291203   5.61756101]
# [37.32525798   5.87497465]]

# open a matplotlib figure
import matplotlib.pyplot as plt

fig = plt.figure()
```

Solve differential equations

```
from scipy.integrate import odeint

time = np.linspace(0, 35, 1000) # time vector
init = [10, 5] # initial condition: 10 prey, 5 predators
x = odeint(predator_prey, init, time) # solve
# returns 2-dimensional np.array():
#[[10.          5.        ]
# [10.17902694  4.87146722]
# [10.36582394  4.74852012]
# ...
# [36.19731528  5.37939637]
# [36.77291203  5.61756101]
# [37.32525798  5.87497465]]

# open a matplotlib figure
import matplotlib.pyplot as plt

fig = plt.figure()
plt.plot(time, x[:,0],"r", time, x[:,1], "b")
```

Solve differential equations

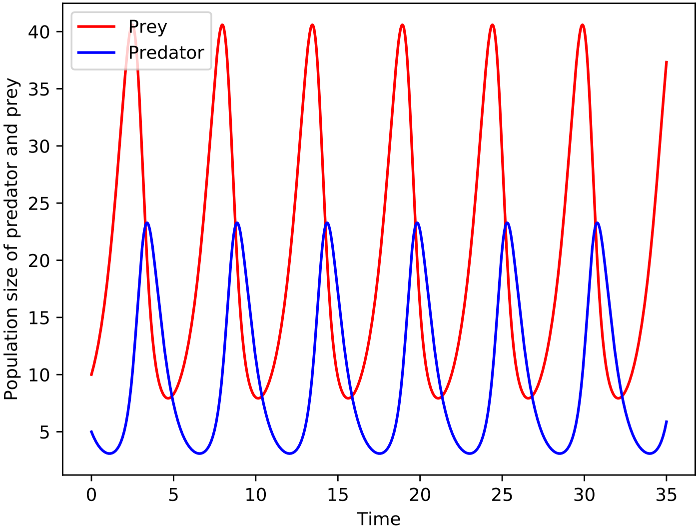
```
from scipy.integrate import odeint

time = np.linspace(0, 35, 1000) # time vector
init = [10, 5] # initial condition: 10 prey, 5 predators
x = odeint(predator_prey, init, time) # solve
# returns 2-dimensional np.array():
#[[10.           5.          ]
# [10.17902694   4.87146722]
# [10.36582394   4.74852012]
# ...
# [36.19731528   5.37939637]
# [36.77291203   5.61756101]
# [37.32525798   5.87497465]]

# open a matplotlib figure
import matplotlib.pyplot as plt

fig = plt.figure()
plt.plot(time, x[:,0],"r", time, x[:,1], "b")
plt.xlabel("Time")
plt.ylabel("Population size of predator and prey")
fig.savefig("graph.pdf")
```

Solve differential equations: resulting graph



Fourier transform

```
from scipy.fftpack import fftfreq, fft

# Create frequency vector
N = len(time)
freq = fftfreq(N, np.mean(np.diff(time)))
freq = freq[range(int(N/2))]

# Compute Fast Fourier Transform
y = fft(x[:, 0])/N # compute and normalise fft
y = y[range(int(N/2))] # keep only positive frequencies
```

Fourier transform