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

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

A taste of NumPy: vectors

- Example of a `NumPy` vector

```
1 import numpy as np
2
3 # 1D array/vector
4 x = np.array([1, 3, 4, 2])
5 x.min() # return min of array
6 x.max() # return max of array
7 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

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

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

A taste of NumPy: vectors

- Example of a `NumPy` vector

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

- Here, `np.array()` returns a `np.ndarray` object.
  - Not a list, hence 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

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

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

```
1 x_list = list(x) # [1, 3, 4, 2 ]
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:

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

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

A taste of NumPy: matrices and linear algebra

- Specifying the matrix:

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

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

A taste of NumPy: matrices and linear algebra

- Specifying the matrix:

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

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

A taste of NumPy: matrices and linear algebra

- Specifying the matrix:

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

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

A taste of NumPy: linear algebra

- Calculate eigenvalues and eigenvecturs

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

A taste of NumPy: linear algebra

- Calculate eigenvalues and eigenvecturs

```
1 import numpy as np
2
3 # declare a 2x2 matrix
4 x = np.array([[1, 2], [3, 4]])
5
6 # calculate eigenvalues
7 eival = np.linalg.eigvals(x) # [-0.37228132
      5.37228132]
```

A taste of NumPy: linear algebra

- Calculate eigenvalues and eigenvecturs

```
1  import numpy as np
2
3  # declare a 2x2 matrix
4  x = np.array([[1, 2], [3, 4]])
5
6  # calculate eigenvalues
7  eival = np.linalg.eigvals(x) # [-0.37228132
       5.37228132]
8
9  # calculate eigenvalues and right eigenvectors
10 eival, eivec = np.linalg.eig(x)
11 # eigenvalues (eival):
12 # [-0.37228132  5.37228132]
13 # [-0.37228132  5.37228132]
14 #
15 # eigenvectors (eivec):
16 # [-0.82456484 -0.41597356]
17 # [ 0.56576746 -0.90937671]]
```

Numpy: number sequences

- Functions to create evenly spaced sequences of numbers:

```
1 # 0 to 1 (but not including 1) in steps of 0.1
2 np.arange(0, 1, 0.1)
3
4 # 100 evenly spaced values between 0 and 1 (including
      1)
5 np.linspace(0, 1, 100)
6
7 # 10 evenly spaced values between 10^0 and 10^1 (log
      scale)
8 np.logspace(0, 1, 10)
```

Numpy: random numbers

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

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

Numpy: random numbers

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

```
1 # import the default random number generator function
2 from numpy.random import default_rng
3
4 # initialize random number generator object
5 # optionally one can provide it with a seed:
6 # same random sequence is then 'replayed' each time
7 rng_obj = default_rng(seed=3434)
```

Numpy: random numbers

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

```
1  # import the default random number generator function
2  from numpy.random import default_rng
3
4  # initialize random number generator object
5  # optionally one can provide it with a seed:
6  # same random sequence is then 'replayed' each time
7  rng_obj = default_rng(seed=3434)
8
9  # uniform distribution
10 rng_obj.uniform(size=5)
11
12 # normal distribution
13 rng_obj.normal(size=10)
14
15 # random integers from 0 to 9 (in this case: returns a
       5x5 matrix)
16 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

```
1  import scipy.stats as sp
2
3  # Create two random arrays
4  x1 = rng_obj.normal(size=30)
5  x2 = rng_obj.normal(size=30)
6
7  # Correlation coefficientss
8  sp.pearsonr(x1, x2) # pearson correlation
9  sp.spearmanr(x1, x2) # spearman correlation
10 sp.kendalltau(x1, x2) # kendall correlation
11
12 # Statistical tests
13 sp.ttest_ind(x1, x2) # independent t-test
14 sp.mannwhitneyu(x1, x2) #  Mann-Whitney rank test
15 sp.wilcoxon(x1, x2) # Wilcoxon signed-rank test
16
17 # Least-squares regression
18 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

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

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

```python
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

```
1 from scipy.integrate import odeint
2
3 time = np.linspace(0, 35, 1000) # time vector
4 init = [10, 5] # initial condition: 10 prey, 5 predators
5 x = odeint(predator_prey, init, time) # solve
```

Solve differential equations

```
1  from scipy.integrate import odeint
2
3  time = np.linspace(0, 35, 1000) # time vector
4  init = [10, 5] # initial condition: 10 prey, 5 predators
5  x = odeint(predator_prey, init, time) # solve
6  # returns 2-dimensional np.array():
7  #[[10.          5.        ]
8  # [10.17902694  4.87146722]
9  # [10.36582394  4.74852012]
10 # ...
11 # [36.19731528  5.37939637]
12 # [36.77291203  5.61756101]
13 # [37.32525798  5.87497465]]
```

Solve differential equations

```
1  from scipy.integrate import odeint
2
3  time = np.linspace(0, 35, 1000) # time vector
4  init = [10, 5] # initial condition: 10 prey, 5 predators
5  x = odeint(predator_prey, init, time) # solve
6  # returns 2-dimensional np.array():
7  #[[10.          5.        ]
8  # [10.17902694  4.87146722]
9  # [10.36582394  4.74852012]
10 # ...
11 # [36.19731528  5.37939637]
12 # [36.77291203  5.61756101]
13 # [37.32525798  5.87497465]]
14
15 # open a matplotlib figure
16 import matplotlib.pyplot as plt
```

Solve differential equations

```
1  from scipy.integrate import odeint
2
3  time = np.linspace(0, 35, 1000) # time vector
4  init = [10, 5] # initial condition: 10 prey, 5 predators
5  x = odeint(predator_prey, init, time) # solve
6  # returns 2-dimensional np.array():
7  #[[10.          5.        ]
8  # [10.17902694  4.87146722]
9  # [10.36582394  4.74852012]
10 # ...
11 # [36.19731528  5.37939637]
12 # [36.77291203  5.61756101]
13 # [37.32525798  5.87497465]]
14
15 # open a matplotlib figure
16 import matplotlib.pyplot as plt
17
18 fig = plt.figure()
```

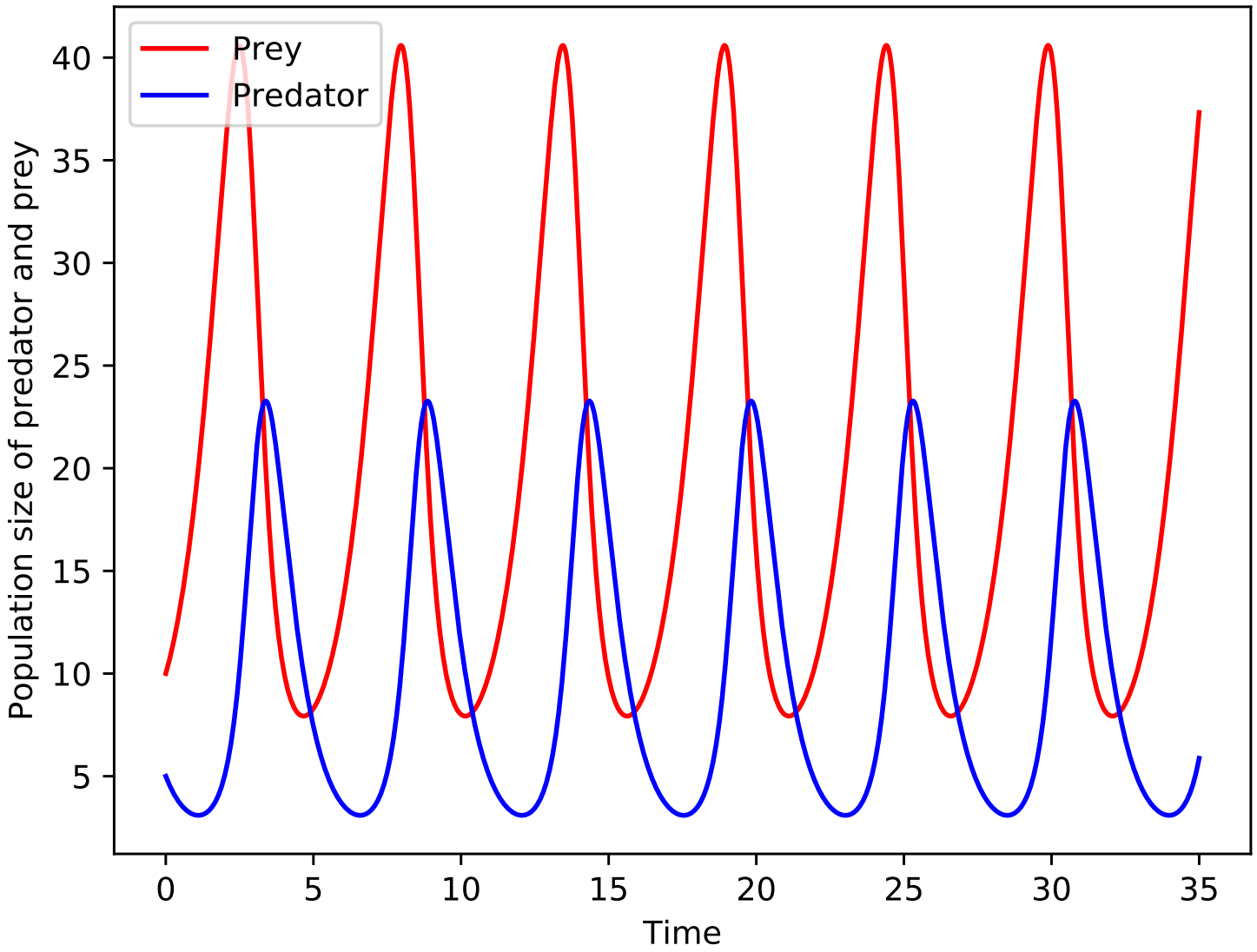
Solve differential equations

```
1  from scipy.integrate import odeint
2
3  time = np.linspace(0, 35, 1000) # time vector
4  init = [10, 5] # initial condition: 10 prey, 5 predators
5  x = odeint(predator_prey, init, time) # solve
6  # returns 2-dimensional np.array():
7  #[[10.          5.        ]
8  # [10.17902694  4.87146722]
9  # [10.36582394  4.74852012]
10 # ...
11 # [36.19731528  5.37939637]
12 # [36.77291203  5.61756101]
13 # [37.32525798  5.87497465]]
14
15 # open a matplotlib figure
16 import matplotlib.pyplot as plt
17
18 fig = plt.figure()
19 plt.plot(time, x[:,0],"r", time, x[:,1], "b")
```

Solve differential equations

```
1  from scipy.integrate import odeint
2
3  time = np.linspace(0, 35, 1000) # time vector
4  init = [10, 5] # initial condition: 10 prey, 5 predators
5  x = odeint(predator_prey, init, time) # solve
6  # returns 2-dimensional np.array():
7  #[[10.          5.        ]
8  # [10.17902694  4.87146722]
9  # [10.36582394  4.74852012]
10 # ...
11 # [36.19731528  5.37939637]
12 # [36.77291203  5.61756101]
13 # [37.32525798  5.87497465]]
14
15 # open a matplotlib figure
16 import matplotlib.pyplot as plt
17
18 fig = plt.figure()
19 plt.plot(time, x[:,0],"r", time, x[:,1], "b")
20 plt.xlabel("Time")
21 plt.ylabel("Population size of predator and prey")
22 fig.savefig("graph.pdf")
```

Solve differential equations: resulting graph



Fourier transform

```
1  from scipy.fftpack import fftfreq, fft
2
3  # Create frequency vector
4  N = len(time)
5  freq = fftfreq(N, np.mean(np.diff(time)))
6  freq = freq[range(int(N/2))]
7
8  # Compute Fast Fourier Transform
9  y = fft(x[:, 0])/N # compute and normalise fft
10 y = y[range(int(N/2))] # keep only positive frequencies
```

Fourier transform