Getting Started with NumPy

- Introduction to Python data science
- NumPy arrays
- Manipulating array elements
- Manipulating array shape

Introduction to Python data science

- Overview
- Python libraries for data science
- Getting the data science libraries

Overview

Python is a popular choice for data science and machine learning

Attractive characteristics of Python:

- Dynamic language, so it’s good for rapid exploratory coding
- Relatively simple syntax, so it’s easier to become proficient
- Popular in schools and universities, so the skills are out there!

Python libraries for data science

NumPy is a numeric processing API for Python

- Fast mathematical computation of numeric arrays and matrices

Pandas provides additional features based on NumPy

- Additional support for indexing, reading/writing CSV/Excel, etc.

Matplotlib is a graphical plotting API for Python

- Similar to Matlab, allows you to plot graphs, charts, etc.

Scikit-Learn is a machine learning library for Python

- Implements many supervised/unsupervised learning algorithms

Getting the data science libraries

If you're using Anaconda, these are already downloaded for you.

If you're using a standalone Python distribution, you'll need to install the libraries you need.

```
pip install numpy

pip install openpyxl
pip install xlrd
pip install matplotlib

pip install pandas
```

NumPy arrays

- Getting Started with NumPy arrays
- Techniques for creating NumPy arrays
- Reading CSV data
- Visualizing data

Getting Started with NumPy arrays (1/2)

NumPy holds data in N-dimensional arrays

- An array is an instance of the numpy.ndarray class
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html

All the data in a NumPy array is the same type

- This allows NumPy to store and process the data efficiently
- This is a very important point!

Why are NumPy arrays more efficient than Python lists?

- Python is dynamically typed, so every object contains metadata that identifies the type at run time
- In a Python list, every item contains this metadata - eek!
- In a NumPy array, only the array itself contains the metadata

Getting Started with NumPy arrays (2/2)

Example

- Creates a NumPy array from a Python list
- Gets the shape of the array, via the shape property
- Gets the data type of array elements, via the dtype property

```
# Import the NumPy module.
import numpy as np

# Create a 1D NumPy array from a Python list.
a = np.array([1, 2, 3])
print('Data values in a\n', a)
print('Shape of a:', a.shape)
print('Data type in a:', a.dtype)

# Create a 2D Numpy array from a Python list of lists.
b = np.array([[1, 2, 3], [4, 5, 6]])
print('\nData values in b\n', b)
print('Shape of b:', b.shape)
print('Data type in b:', b.dtype)
```

For a full list of NumPy standard types, see:

Techniques for creating NumPy arrays (1/2)

There are lots of ways to create a NumPy array

```
import numpy as np

# Create array with mixed types - NumPy converts elements "upwards".
a = np.array([1, 2, 3.14])

# Create array with a specified type.
b = np.array([1, 2, 3], dtype='float64')

# Create array from a numeric range.
c = np.arange(0, 20, 2)

# Create array of elements, linear spaced.
d = np.linspace(0.0, 1.0, 11)

# Create array of zeros.
# You can specify multi-dim arrays too.
e = np.zeros(5)

# Create array of ones.
f = np.ones(5)

# Create array of elements, with specified value.
```

Techniques for creating NumPy arrays (2/2)

You can also create random arrays, which can be handy

```
# Import the NumPy module.
import numpy as np

# Create random values in range [0.0, 1.0).
a = np.random.random(10)

# Create normally-distributed random values.
b = np.random.normal(5, 2, 10)

# Create random integers in range [0, 101).
c = np.random.randint(0, 101, 10)
```

Reading CSV data

A common requirement is to read data from a CSV file

- The easiest way to do this is via the Pandas read_csv() function

Pandas reads values into a multi-column DataFrame

- You can then extract a column into a NumPy array

```
import numpy as np
import pandas as pd

# Read a csv file, get a Pandas DataFrame back.
dataframe = pd.read_csv('WorldCupWinners.csv')

# Get the 'Teams' column.
teams = np.array(dataframe['Team'])
print(teams)
```

We'll not have time to dive into Pandas on this course but it's a topic worth exploring!

Visualizing data (1/2)

Visualization is an important aid to help you understand the shape and meaning of data

You can use the MatPlotLib library to visualize data in lots of different ways

- Line graphs
- Scatter graphs
- Bar-charts
- Pie-charts
- Histograms
- Etc.

Visualizing data (2/2)

Here's a simple example of how to visualize data using MatPlotLib - we'll see more plotting features later

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = np.array([1, 19, 76, 45, 34, 42, 30, 5, 77, 54, 89])

plt.xlabel('Element in array')
plt.ylabel('Value')
plt.plot(data)
plt.show()
```

Manipulating array elements

- Indexing into an array
- Slicing an array
- Accessing a specific column or row
- Aside: Views vs. copies

Indexing into an array

Indexing into a NumPy array is quite intuitive

- [i] Access element from start, first element is at [0]
- [-i] Access element from end, last element is at [-1]
- [r,c] Access element in 2-D array (etc. for higher dimensions)

```
1   import numpy as np
2
3   # Create a 1-D array, index into it, and modify elements.
4   a = np.array([0, 10, 20, 30, 40, 50, 60, 70])
5   print(a)
6   print(a[1])        # 10
7   print(a[-1])       # 70
8   a[1] = 111
9   print(a)
10
11  # Create a 2-D array, index into it, and modify elements.
12  b = np.array([[0, 10, 20, 40], [50, 60, 70, 80]])
13  print(b)
14  print(b[0, 1])     # 10
15  print(b[0, -1])    # 40
16  print(b[-1, 1])    # 60
17  print(b[-1, -1])   # 80
```

Slicing an array

You can slice into an array using a [start:stop:step] index

- start Default start is 0
- stop Default stop is the size of the dimension
- step Default step is 1

```
import numpy as np

# Create a 1-D array, and get various slices.
a = np.array([0, 10, 20, 30, 40, 50, 60, 70, 80])
print(a[3:])         # [30 40 50 60 70 80]
print(a[:3])         # [ 0 10 20]
print(a[3:7])        # [30 40 50 60]
print(a[3:7:2])      # [30 50]
print(a[3::2])       # [30 50 70]
print(a[::2])        # [ 0 20 40 60 80]

# Create a 2-D array, and get various slices in each dimension.
b = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]])
print(b[1:, 1:])     # [ [40 50] [70 80] ]
print(b[:2, :2])     # [ [ 0 10] [30 40] ]
print(b[::2, ::2])   # [ [ 0 20] [60 80] ]
```

Accessing a specific column or row

To get a specific column or row in a multidimension array:

- Use an empty slice to skip a dimension
- E.g. in a 2D array, [:,1] gets column 1
- E.g. in a 2D array, [1,:] gets row 1

```
import numpy as np

a = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]])

# To access a specific column ...
print(a[:, 0])   # [ 0 30 60]
print(a[:, 1])   # [10 40 70]
print(a[:, 2])   # [20 50 80]

# To access a specific row ...
print(a[0, :])   # [ 0 10 20]
print(a[1, :])   # [30 40 50]
print(a[2, :])   # [60 70 80]

# To access a specific row, simpler syntax ...
print(a[0])      # [ 0 10 20]
```

Aside: Views vs. copies

When you get an array slice/row/column, you get a view on the data

- If you make any changes, it will change the actual data

If you want to get a copy of the data:

- Call copy() on the slice/row/column

```
1   import numpy as np
2
3   # Demonstrate views.
4   a = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]])
5   col0View = a[:, 0]
6   col0View[2] = 600
7   print(col0View)
8   print(a)   # [[ 0  10  20] [ 30  40  50] [600  70  80]]
9
10  # Demonstrate copies.
11  b = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]])
12  col0Copy = a[:, 0].copy()
13  col0Copy[2] = 600
14  print(col0Copy)
15  print(b)   # [[ 0  10  20] [ 30  40  50] [60  70  80]]
```

Manipulating array shape

- Reshaping an array
- Creating new axes
- Concatenating arrays
- Stacking arrays vertically or horizontally
- Splitting an array

Reshaping an array

Reshaping is a simple and common technique for creating multidimensional arrays

- Create a 1D array initially (typically)
- Reshape it to a multidimensional array (must be compatible shape)
- The multidimensional array is a view onto the original 1D array

```
import numpy as np

# Create 1D array initially, for simplicity.
a = np.arange(9)
print(a)               # [0 1 2 3 4 5 6 7 8]
print(a.shape)         # (9,)

# Reshape as 2D array (view on a).
b = a.reshape((3,3))
print(b)               # [[0 1 2] [3 4 5] [6 7 8]]
print(b.shape)         # (3, 3)

# Changing items in b will change values in underlying a.
b[0,0] = 99
print(a)               # [99  1  2  3  4  5  6  7  8]
print(b)               # [[99  1  2] [ 3  4  5] [ 6  7  8]]
```

Creating new axes

Another useful technique is create new axes for an array

- Create a 1D array initially (typically)
- Create a new column or row, using np.newaxis

```
1   import numpy as np
2
3   # Create 1D array initially, for simplicity.
4   a = np.arange(5)
5   print(a)               # [0 1 2 3 4]
6   print(a.shape)         # (5,)
7
8   # Create 2D array with 1 row, 5 columns.
9   b = a[np.newaxis, :]
10  print(b)               # [[0 1 2 3 4]]
11  print(b.shape)         # (1, 5)
12
13  # Create 2D array with 5 rows, 1 column.
14  c = a[:, np.newaxis]
15  print(c)               # [[0] [1] [2] [3] [4]]
16  print(c.shape)         # (5, 1)
```

Concatenating arrays (1/2)

You can concatenate same-size arrays together

- np.concatenate() - you can specify the axis to concatenate on

Here's a simple example that concatenates 1D arrays

```
import numpy as np

# Create some 1D arrays.
a = np.array([ 0,  1])
b = np.array([10, 11])
c = np.array([20, 21])

# Concatenate the 1D arrays.
result = np.concatenate([a, b, c])
print(result)         # [0 1 10 11 20 21]
print(result.shape)   # (6,)
```

Concatenating arrays (2/2)

Here's an example that concatenates 2D arrays

- Note the optional axis parameter (default is 0)

```
import numpy as np

# Create some 2D arrays.
a = np.array([[ 0,  1], [10, 11]])
b = np.array([[20, 21], [30, 31]])
c = np.array([[40, 41], [50, 51]])

# Concatenate on axis 0 (this is the default, so can omit axis parameter).
result1 = np.concatenate([a, b, c], axis=0)
print(result1)         # [[0 1] [10 11] [20 21] [30 31] [40 41] [50 51]]
print(result1.shape)   # (6, 2)

# Concatenate on axis 1.
result2 = np.concatenate([a, b, c], axis=1)
print(result2)         # [[0 1 20 21 40 41] [10 11 30 31 50 51]]
print(result2.shape)   # (2, 6)
```

Stacking arrays vertically or horizontally

You can stack different-size arrays together

- np.vstack() - stack vertically (must have same no. of cols)
- np.hstack() - stack horizontally (must have same no. of rows)

```
import numpy as np

# Create some arrays with same number of columns (2), and stack vertically.
a = np.array([10, 11])
b = np.array([[20, 21], [30, 31]])
result1 = np.vstack([a, b])
print(result1)         # [[10 11] [20 21] [30 31]]
print(result1.shape)   # (3, 2)

# Create some arrays with same number of rows (2), and stack horizontally.
c = np.array([[40, 41], [50, 51]])
d = np.array([[60], [61]])
result2 = np.hstack([c, d])
print(result2)         # [[40 41 60] [50 51 61]]
print(result2.shape)   # (2, 3)
```

Splitting an array

You can split an array into subarrays

- np.split()
- np.vsplit()
- np.hsplit()

```
import numpy as np

# Split a 1D array.
a = np.arange(16)
a1, a2, a3, a4 = np.split(a, [2, 5, 9])
print('\na1\n', a1)   # [0 1]
print('\na2\n', a2)   # [2 3 4]
print('\na3\n', a3)   # [5 6 7 8]
print('\na4\n', a4)   # [9 10 11 12 13 14 15]

# Split a 2D vertically.
b = np.arange(16).reshape((4, 4))
b1, b2 = np.vsplit(b, [3])
print('\ntop\n', b1)     # [[0 1 2 3] [4 5 6 7] [8 9 10 11]]
print('\nbottom\n', b2)  # [[12 13 14 15]]

# Split a 2D horizontally.
```

Any questions?