Python: Numpy

NumPy is a package for scientific computing in Python. It implements an object to represent N-dimensional arrays (vectors, matrices and higher dimensional matrices). It also has linear algebra functions and random number generators.

Why do we need Numpy? Python is a dynamically typed language, it infers the type of a variable at runtime. This means that when Python store variables in memory, it not only stores the variable's value, but also its type. Then, when we perform a computation, like adding two variables (`x+y`), Python looks up the type of `x` and the type of `y`, and applies the definition of `+` to those types if it makes sense to do so. If those types are integers, for example, then Python performs an integer addition. However, if the type of `x` is integer, but `y` is a list, then the operation is not defined and Python will raise an exception (error). This type of checking is called a runtime type check.

The runtime type check makes programming in Python a pleasure, since you do not have to worry about types all the time. However, it does lead to inefficiencies when we start performing operations on large datasets. Numpy solves those inefficiencies.

Numpy introduces a new type of list, called a Numpy array. In an array, all elements have the same type. This has two benefits. First, the type is only stored once, so performing operations with numpy is much faster than with Python lists. Second, since all elements have the same type, their position in the memory is easy to compute. This means that accessing random elements in a numpy array is quick.

Numpy also provides a set of operations for numpy array, all implemented in C. Arrays are not required to be 1 dimensional, so the class can also deal with matrices and higher dimensional arrays. Numpy is the base of other higher-level packages, like `pandas` and `scipy`.

1 Installing and Importing

Numpy comes pre-installed with Anaconda, so all we need to do is import it into Python. In the jupyter notebook, we can import the package by executing:

```
import numpy
```

The `import` will look for the package named `numpy`, will find it and load all of its contents. The methods, variables and objects defined in the Numpy package will be available in the object `numpy`.

You will often see the following command being used:

```
import numpy as np
```

The code above defines `np` as an alias for `numpy`. An alias is often used because it requires less typing to use numpy.

2 Basics

To create a matrix with numpy we run:

```
matrix = np.array([[0, 1, 2, 3],
                   [4, 5, 6, 7]])
```

This creates a new matrix with dimensions 2 by 4. Notice that `np.array` takes as input a list of lists, where each sublist represents a row of a matrix and each element in the sublist is a different column.

Basic properties of `np.ndarray`:

```
type(matrix)
print(f'Shape of matrix = {matrix.shape}')
print(f'Number of axes = {matrix.ndim}')
print(f'Total number of elements = {matrix.size}')
```

The elements of the matrix can be accessed by using its indices. Since the matrix has 2 dimensions, we need to give it two indices: an index for the row and an index for the column. Remember that in Python the indexing starts at 0.

```
print(matrix[0, 0])                    # element at row 0 and column 0
print(matrix[1, 3])                    # last element of the 2nd row
print(matrix[1, -1])                   # last element of the 2nd row
print(matrix[-1, -1])                  # element at last row and last
    column
```

We can use `:` to slice the matrix and obtain all values of a certain row or of a certain column:

```
print(matrix[0, :])              # first row, all columns
print(matrix[:, 0])              # all rows, first column
print(matrix[0, 1:3])            # first row, columns 1 and 2
print(matrix[:, 1:3])            # all rows, columns 1 and 2
```

We can also select specific columns or rows:

```
print(matrix[:,[0, 3]])          # all rows, first column and last column
```

Like the built-in function `range`, numpy also provides a way to generate a sequence of numbers with the function `np.arange`:

```
vector = np.arange(20)
print(vector)
```

We can reshape this vector into a matrix:

```
matrix = vector.reshape(4,5)
print(matrix)
```

Notice that when converting a vector into a matrix we need to decide whether to put the values first into columns, or first into rows. For example, if we have a vector $[1, 2, 3, 4]$ and want to reshape it into a $2x2$ matrix, there are 2 natural possibilities: create the new matrix by filling each row first, or create the matrix by filling each column first.

$$\text{Fill Rows: } \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ vs. Fill Columns: } \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

The default in Python is to fill each row before moving to the next (first matrix above). The behavior is controlled by a keyword argument `order` given to the reshape method.

```
a = np.arange(4)
print(a)
# Different ways of reshaping
# fill each row, row by row
a.reshape((2, 2), order='C') # this is also the default
# fill each column, column by column
a.reshape((2, 2), order='F')
```

We can also get the type of the values stored in the matrix:

```
print(matrix.dtype)
```

On many cases we will want to create an empty array and populate as the code is executed. We can create arrays pre-filled with ones or zeros via:

```
ones = np.ones((4, 5))
zeros = np.zeros((10, 3))
```

The functions `np.ones` and `np.zeros` take a tuple as input, which defines the shape of the matrix to be created.

3 Data Types

The data type of values stored in a matrix are usually inferred when the values are assigned:

```
matrixOfIntegers = np.array([[0, 1],
                             [3, 4]])
print(matrixOfIntegers.dtype)
matrixOfFloats = np.array([[1.0, 2.3],
                           [-5.1, 9.8234]])
print(matrixOfFloats.dtype)
```

The first matrix has the type `int64` and the second matrix has the type `float64`. Notice that if we try to change a value in the first matrix to a float, it will be implicitly converted to an integer:

```
print(matrixOfIntegers[0, 0])
matrixOfIntegers[0, 0] = 0.34
print(matrixOfIntegers[0, 0])
```

We can specify the type of data when we first create the matrix:

```
matrix = np.array([[0, 1], [2, 3]], dtype='float64')
matrix[0, 0] = 3.2
print(matrix)
```

The matrix is initialized with integers, but we specify the type to be a float, so the values are implicitly converted to floats.

It is also possible to store strings in numpy arrays (matrices):

```
names = np.array(['A', 'B', 'C', 'D'], dtype='str_')
print(names.dtype)
```

Notice that the type is actually "<U1". The "U" stands for "unicode string" and "1" is the number of characters. Since all values passed to the `np.array` constructor have just 1 character, numpy assumes all data that we will store on this matrix will have a single character. This might not be the case, and if we try to store a bigger string:

```
names[0] = 'Guilherme'
print(names[0])
```

Instead of storing the entire string, only the first character was stored. We can let numpy know that we need more space in memory by defining how many characters we need:

```
names = np.array(['A', 'B', 'C', 'D'], dtype='<U100')
names[0] = 'Guilherme'
print(names[0])
```

Now we can store 100 characters in each element of the matrix.

The complete documentation on specifying data types (`dtype`) can be found on the data type objects reference page. Arrays can also hold more than one type, details on how to hanlde multiple types can be found on the structure arrays reference page.

4 Operations

Arithmetic operations with numpy arrays are always element wise.

```
a = np.arange(10)
b = np.ones(10)                    # float
print(f'Type a = {a.dtype}, b = {b.dtype}')
c = a + b
print(c)
print(f'Type c = {c.dtype}')     # implicit conversion of a to float
print(c*10)
print(c - 10)
print(c**2)
```

Notice that `**` is the same as `pow(c,2)`.

Comparisons are also done element wise:

```
print(c > 5)
```

Booleans can be used to recover elements of an array:

```
c[c>5]
```

You can recover the indices that satisfy a condition with the `np.where` function:

```
indices = np.where(c <= 2)
print(indices)
print(c[indices])
```

It is possible to manipulate all elements of an array and accumulate the values:

```
# some random values to stand in as returns
returns = np.random.random((78, 5))
returns /= 100                          # divides everything by 100
RV = np.sum(returns**2, axis=0)
annRV = 100*np.sqrt(252*RV)
print(f'RV = {RV}\n'
      f'annualized RV = {annRV}')
```

The `np.random.random` is a function that generates random values in the interval $[0, 1)$. The `np.sum` function accumulates over elements in an array by summing them. Calling `np.sum` without the `axis` argument will sum all of the elements in a matrix, resulting in

a single value. The keyword argument `axis=0` instructs the sum to occur along the rows. Numpy provides many universal mathematical functions, such as `np.sqrt` for computing the square root of a number (remember, element wise).

We can find the minimum and maximum values in an array:

```
print(f'Min RV = {np.min(annRV)}\n'
      f'Max RV = {np.max(annRV)}')
print(f'Index of Min = {np.argmin(annRV)}\n'
      f'Index of Max = {np.argmax(annRV)}')
```

Numpy arrays also implement a multitude of different methods:

```
x = np.array([4,3,2,1]);
# in-place sort
x.sort()
print(x)
# common operations
print(x.sum(), x.mean(), x.max(), x.min())
# equivalent to
print(np.sum(x), np.mean(x), np.max(x), np.min(x))
# index of extremum
print(x.argmin(), x.argmax(), np.argmin(x), np.argmax(x))
# variance
x.var()
x.std()
x.std() == x.var()**0.5
```

Numpy also implements matrix operations:

```
# Matrix multiplication
# create identity matrix
a = np.eye(2)
b = np.random.random((2, 2))
print(np.matmul(a, b))
# The symbol @ is overloaded for matrix multiplication
print(a @ b)
print(a @ b == np.matmul(a,b))
# Matrix transposition
print(np.transpose(b))
print(b.T)
# Matrix inversion
c = np.array([[1, 2],
              [3, 4]])
print(np.linalg.inv(c))
```

Numpy also implements other functions, like singular value decomposition (`np.linalg.svd`), eigenvalues (`np.linalg.eig`), Moore-Penrose inverse (`np.linalg.pinv`), Kronecker product (`np.kron`) and others. For more information on linear algebra with Numpy check the linear algebra reference page.

5 Mutability and Copying Arrays

Create a new array:

```
x = np.array([4,3,2,1], dtype='float_')
# arrays are mutable
x[0] = 10.32
```

```
print(x)
```

What happens when we assign a new name to x?

```
a = x
a[1] = -2.34
# What is printed?
print(a, a[1], x[1])
```

Notice that `x` is a name to an array, and `a` is a reference to `x`. Thus, `a` is also a name to the exact same array.

```
a == x
```

This is a sensible behavior for memory efficiency. If you do need to make a copy of an array, then:

```
a = np.copy(a)                    # deep copy
a[0] = -0
print(a, x)
```

6 Additional Functionality

Numpy has several functions that can be applied element-wise:

```
x = np.array([1, 2, 3, 4])
# vectorized functions (element-wise)
print(np.sin(x))                  # implicit type conversion
print(np.log(x), np.exp(x))
```

These functions are referred to as universal functions (`ufunc`). They are universal in the sense that they work with arrays (work with many elements).

We can compose these operations:

```
# +, -, /, * and ** also work element-wise
print(np.log(x + 1)*3/np.exp(x)**2)
```

If a function is not universal (also known as vectorized), applying it on an array will lead to error:

```
def f(x):
    return 1 if x > 0 else 0
f(x)
```

The issue here is that `bool(x)` is not defined for numpy arrays. While `x > 0` works, the part that says `if x > 0` is equivalent to `if bool(x > 0)`, which is undefined for arrays. We can vectorize a function:

```
f = np.vectorize(f)
f(x)
```

The function `np.vectorize` is basically creating a for-loop around `f`:

```
res = np.zeros(len(x))
for i in np.arange(len(x)):
    res[i] = f(x[i])
print(res)
```

We can accomplish the result of the function `f` using np.where:

```
help(np.where)
# returns 1s where x > 0, and 0 otherwise
np.where(x > 0, 1, 0)
```

Remember that comparisons are element-wise and can be used for slicing:

```
# comparisons are element-wise
a = np.array([-1, 0, 1])
b = np.array([1.2, -2.3, 3])
a > b                          # implicit type conversion for a
a > 0.2
# slice based on comparisons
a[a > 0.2]
a[a > b]
```

The Numpy subpackage `np.random` implements several random number generators. You can get more information about them with `help`:

```
help(np.random)
```

The Numpy subpackage `np.linalg` has other linear algebra tools. You can also use `help` to learn more about them:

```
# numpy subpackage np.linalg implements linear algebra functions
help(np.linalg)
```

A table that compares commands in Matlab to commands in Numpy is available here. A good overview of many numpy features is available here.

7 Implementing the OLS Estimator for Linear Regressions

We now know everything we need to know about Python to estimate parameters in a linear regression:

```
def linreg_ols(x, y):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    return beta
```

Generate some test data to test `linreg_ols`:

```
# 1000 data points with a constant + 3 explanatory variables
x = np.hstack((np.ones((1000, 1)), np.random.random((1000, 3))))
print(x)
beta = np.random.random((4, 1))
epsilon = np.random.random((1000, 1))
y = x @ beta + epsilon
print(y)
```

Estimate beta and compare the estimate to the true value:

```
beta_hat = linreg_ols(x, y)
print(np.hstack((beta, beta_hat)))
```

Let's update `linreg_ols` so that adding a constant to `x` is automatic:

```
def linreg_ols(x, y):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    return beta
```

Test it again:

```
x = x[:, 1:]
print(x)
print(x.shape, y.shape)
beta_hat = linreg_ols(x, y)
print(beta_hat)
```

Update `linreg_ols` so that adding a constant can be optionally specified by the user:

```
def linreg_ols(x, y, intercept=True):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    if intercept:
        x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    return beta


# if no intercept is supplied, than a column of ones is added
print(linreg_ols(x, y))
print(linreg_ols(x, y, True))
print(linreg_ols(x, y, 1))
print(linreg_ols(x, y, False))
```

Always remember to add documentation to your functions (your future self will thank you):

```
def linreg_ols(x, y, intercept=True):
    """Estimates linear regression coefficients with OLS.

    Estimates beta in a linear regression: y = x @ beta + epsilon.
    Uses the ordinary least squares estimator.

    Args:
        x (np.array): A nxK matrix of explanatory variables
        y (np.array): A nx1 matrix of dependent variables
        intercept (bool): Specifies whether to estimate intercept
    coefficient

    Returns:
        beta (np.array): A Kx1 vector with the beta estimates

    Raises:
        AssertionError: x and y have different number of rows
        LinAlgError: if x is a singular matrix

    Example:
        linreg_ols(np.array([[1, -2], [0.5, -3.1]]), np.array([1, 2.3]))
    """
    assert x.shape[0] == y.shape[0], "Different number of rows"
    if intercept:
        x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
```

```
    return beta


help(linreg_ols)
```

8 Save and Load Results

After running linear regressions, we might want to save the results. We can use `np.save` for that. It saves the results in a file for later use. The function `np.save` saves in a binary file, which is faster for loading compared to `.csv` files (but it is not human-readable).

```
beta = linreg_ols(np.random.random((1000, 3)),
                  np.random.random((1000, 1)))
print(beta)

help(np.save)
np.save('regression_estimates', beta, allow_pickle=False)  # npy extension
```

If you are only using the data only on your computer (not sharing), then you can set `allow_pickle=True` to speed things up.

To load the data, use `np.load`:

```
# load the data
del beta
print(beta)                        # not defined
beta = np.load('regression_estimates.npy', allow_pickle=False)
print(beta)
```

We can save the results in a `.csv` file, which is easier to share and inspect. To do so, we use `np.savetxt`:

```
np.savetxt('regression_estimates.csv', beta, delimiter=',')
# load from .csv
del beta
print(beta)
beta = np.loadtxt('regression_estimates.csv', delimiter=',')
print(beta)
```

We can also use `np.loadtxt` to load data created outside Python.