Core python scientific libraries: numpy, pandas, scipy

Array package: numpy

The python standard library does not include higher dimensional data structures such as 2-D arrays/matrices (although, as we have seen, a list of lists can be used as a form of 2D data structure).

Numpy adds to python arbitrarily high dimension array data structures, which are an essential component of scientific programming, to allow direct expression of linear algebra operations (matrices and vectors) as well as processing of 2D data such as images, experimental results, and higher dimensional arrays/tensors in deep neural networks etc.

Other scientific, engineering and statistical programming languages such as Matlab, R, IDL have such array data structures built-in.

A key advantage is that numpy arrays allow elementwise operations, which allows sophisticated vectorized expressions that perform complex operations without requiring explicit loops.

Numpy arrays are also much faster and memory efficient compared to the built-in python list, as they are homogeneous arrays with elements stored consecutively in memory, unlike builtin lists which allow a heterogeneous mix of element types and stores references (pointers) to objects- this flexibility leads to increased memory usage and decreased performance.
Numpy (and scipy) also calls lower-level C libraries that are faster than python and are numerically sophisticated.

See code examples and the numpy tutorial at https://numpy.org/doc/stable/user/absolute_beginners.html for an overview.

Statistical packages: pandas

There are many statistical tools. Some programming languages, such as R, are even designed for use in statistical analysis.

One of the most common statistical packages in Python is `pandas`, which builds on numpy arrays and implements the data frame data structure based on the R syntax.
For further details see:
https://pandas.pydata.org/docs/user_guide/10min.html#min
https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

To get started with `pandas`, you will need to get comfortable with its two data structures: *Series* and *DataFrame*.

1.1 Import module

```
In [1]:from pandas import Series, DataFrame
```

1.2 Series

A *Series* is a one-dimensional vector capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.) and an associated array of data labels, called its *index*. If no index is passed, one will be created automatically starting from 0.

```
In [1]:s =Series([4,7,-5,3])

In [2]:s

Out[2]:0    4
       1    7
       2   -5
       3    3
      dtype: int64


In [3]:s.values

Out[3]:array([ 4,  7, -5,  3])

In [4]:s = Series(range(5), index=list('abcde'))

In [5]:s

Out[5]:a    0
       b    1
       c    2
       d    3
       e    4
      dtype: int64
```

```
In [6]:s.mean()

Out[6]:2.0

In [7]:s+s

Out[7]:a    0
       b    2
       c    4
       d    6
       e    8
      dtype: int64
```

1.3 DataFrame

A *DataFrame* represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type. Consider this example of analysis comparing 2 genes in 3 different experiments:

```
In [8]:df = DataFrame([[20,20],[21,30],[19,40]],columns =

['gene_experimental','gene_control'])

In [9]:df

Out[9]:    gene_experimental gene_control
         0     20     20
         1     21     30
         2     19     40
```

Calculate the averages for each gene

```
In [10]:df.mean()

Out[10]:gene_experimental 20
        gene_control 30
```

So what if we just want the mean of `gene1`?

```
In [11]:df['gene1'].mean()

Out[11]:20
```

What about the variance?

```
In [12]:df.var()

Out[12]:gene_experimental 1
        Gene_control 100
```

There are higher-level stats functions in scipy:

```
In [13]: from scipy import stats
In [19]: stats.ttest_ind(df.gene1, df.gene2)

Out[13]: Ttest_indResult(statistic=-1.7234549688642784,
pvalue=0.15990222143539265)
```

1.4 Missing data

Missing data is common in most data analysis applications. While doing it by hand is always an option, `dropna` can be very helpful. On a *Series*, it returns the *Series* with only the non-null data and index values:

```
In [13]:s2 = s[1:]+s[:-1]

In [14]:s2

Out[14]:a    NaN
        b     2
        c     4
        d     6
        e    NaN
       dtype: int64


In [15]:s2.dropna()

Out[15]:b    2
        c    4
        d    6
```

Rather than filtering out missing data, you may want to fill in the 'holes' in any number of ways (known as data imputation). For most purposes, the `fillna` method is the workhorse function to use.

```
In [16]:s2.fillna(s2.mean())

Out[16]:a    4
        b    2
        c    4
        d    6
        e    4
       dtype: int64
```