Statistics Package in Python

Python provides very easy use of basic statistical functions via the statistics package, which can be imported as follows:

```
import statistics
```

Each of the following functions takes in a list of values (the data in question) and returns the appropriate statistic:

Here is a chart from python's website (https://docs.python.org/3/library/statistics.html) of the statistics functions in Python 3.8 dealing with measures of central location:

`mean()`	Arithmetic mean ("average") of data.
`fmean()`	Fast, floating point arithmetic mean.
`geometric_mean()`	Geometric mean of data.
`harmonic_mean()`	Harmonic mean of data.
`median()`	Median (middle value) of data.
`median_low()`	Low median of data.
`median_high()`	High median of data.
`median_grouped()`	Median, or 50th percentile, of grouped data.
`mode()`	Single mode (most common value) of discrete or nominal data.
`multimode()`	List of modes (most common values) of discrete or nomimal data.
`quantiles()`	Divide data into intervals with equal probability.

Here are the functions dealing with the spread of the data:

`pstdev()`	Population standard deviation of data.
`pvariance()`	Population variance of data.
`stdev()`	Sample standard deviation of data.
`variance()`	Sample variance of data.

The python documentation is good for most of these functions. The ones that one would use the most are:

```
mean(data)
median(data)
mode(data)
pstdev(data)
```

These calculate the mean, median, mode and standard deviation of the list of values passed to them.

Here is a short example where data is hard-coded into a list and each function call is made:

```
import statistics

data = [65, 68, 67, 72, 74, 71, 69, 61, 63, 64, 65, 69, 72]

avg = statistics.mean(data)
middle = statistics.median(data)
mostfreq = statistics.mode(data)
mystdev = statistics.pstdev(data)

print("Average of the data =", avg)
print("Median of the data =", middle)
print("Most common value =", mostfreq)
print("Standard Deviation =", mystdev)
```

Use of package numpy

Numpy is a package you have to download separately:

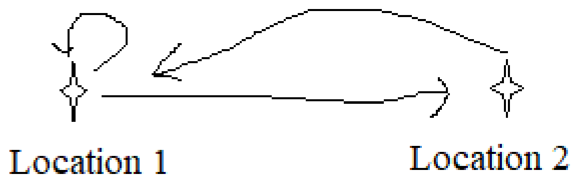
https://numpy.org/

This is a very rich package that works with several other packages to aid scientific computation and visualization.

One object that numpy makes easy to work with is matrices. A matrix is grid of numbers. Here is an example of a matrix with 2 rows and 2 columns:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

For example, this matrix could represent that there is one path from location 1 to location 1, one path from location 1 to location 2, and one path from location 2 to location 1, with no path from location 2 to location 2. Here is a small picture:



Here is how to create this array in numpy:

```
import numpy

mat = numpy.array([[1,1],[1,0]])
```

One question we could ask is: how many paths are there of length k from location i to location j? It turns out that the answer to all questions of this form can be calculated by taking the matrix above and raising it to the kth power. Matrix multiplication is defined in a non-intuitive way and is beyond the scope of what is presented in these notes. But, to multiply two matrices using numpy, just use the @ operator:

```
matsq = mat @ mat
```

Now, let's raise this matrix to the 10th power:

```
mat10 = mat

for i in range(9):
    mat10 = mat10 @ mat
```

```
print(mat10)
```

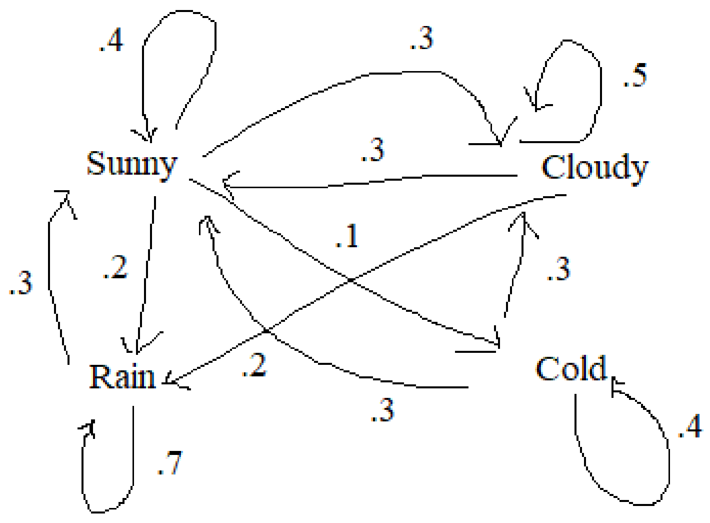
An even easier way to do this is to use the matrix_power function in numpy.linalg:

```
mat10 = numpy.linalg.matrix_power(mat,10)
```

What is kind of fun is that when you print this matrix, each entry is a Fibonacci number. So the number of paths of length 10 in the picture above from location 1 to location 2 is 55, or the 10th Fibonacci number. In fact, all the numbers in this matrix exponentiation are Fibonacci numbers!

Markov Chain Example

Another powerful application of matrix exponentiation is Markov Chains. A Markov Chain is a diagram that relates a set of states via conditional probabilities. For example, imagine that we have four possible states of weather: "Sunny", "Cloudy", "Rainy" and "Cold." Also, assume that given that the weather on a particular day is one of these four, we have the probability of what the next day will bring. Here is a sample diagram with some made up probabilities filled in:



$$\begin{array}{cccc} .4 & .3 & .2 & .1 \\ .3 & .5 & .2 & 0 \\ .3 & 0 & .7 & 0 \\ .3 & .3 & 0 & .4 \end{array}$$

Below the picture you see the matrix that represents the data. If we raise this matrix to a high power, we will reveal the steady state probabilities of each state. More generally, when this matrix is raised to the kth power, the entry in row i, column j represents the probability that, if we are in state i, then in k days, we'll end up in state j. Here is a small program that calculates these steady state probabilities:

```
import numpy

weather = 
numpy.array([[.4,.3,.2,.1],[.3,.5,.2,0],[.3,0,.7,0],[.3,.3,0,.4]
])

weatherexp = weather

for i in range(100):
    weatherexp = weatherexp @ weather

conditions = ["Sunny","Cloudy","Rainy", "Cold"]
print("List of probabilities of going from one condition to 
another in 101 days.")

for i in range(len(conditions)):
    for j in range(len(conditions)):
        print(conditions[i],"to",conditions[j],weatherexp[i][j])
```