More on Statistics, List Comprehensions and New Zealand Earthquake Data

Monte Carlo Methods

More on Statistics

A **statistic** is any measureable function of the data: $T(x) : \mathbf{X} \rightarrow \mathbf{T}$.

Thus, a statistic T is also an RV that takes values in the space $\mathbf{T}$.

When $x \in \mathbf{X}$ is the observed data, $T(x) = t$ is the observed statistic of the observed data x.

Example 1: New Zealand Lotto and 1114 IID $deMoirve$ RVs

Let's go back to our New Zealand lotto data.

We showed that for New Zealand lotto (40 balls in the machine, numbered $1, 2, \ldots, 40$), the number on the first ball out of the machine can be modelled as a $deMoivre(\frac{1}{40}, \frac{1}{40}, \ldots, \frac{1}{40})$ RV.

We have the New Zealand Lotto results for 1114 draws, from 1 August 1987 to 10 November 2008 (retrieved from the NZ lotto web site: http://lotto.nzpages.co.nz/previousresults.html).

We can think of this data as x, the realisation of a random vector $X = (X_1, X_2, \ldots, X_{1114})$ where $X_1, X_2, \ldots, X_{1114} \overset{IID}{\sim} deMoivre(\frac{1}{40}, \frac{1}{40}, \ldots, \frac{1}{40})$

The data space is every possible sequence of ball numbers that we could have got in these 1114 draws. $\mathbf{X} = \{1, 2, \ldots, 40\}^{1114}$. There are 40^{1114} possible sequences and our data is just one of these 40^{1114} possible points in the data space.

We will use our hidden function that enables us to get the ball one data in a list. Evaluate the cell below to get the data and confirm that we have data for 1114 draws.

jsMath

```
listBallOne = getLottoBallOneData()
len(listBallOne)
```

```

```

Some statistics

Sample mean

From a given sequence of RVs $X_1, X_2, \ldots, X_n$ (or a random vector $X = (X_1, X_2, \ldots, X_n)$ we can obtain another RV called the **sample mean** (technically, the n-sample mean):

$$T_n((X_1, X_2, \ldots, X_n)) = \bar{X}_n((X_1, X_2, \ldots, X_n)) := \frac{1}{n}\sum_{i=1}^{n} X_i$$

We write $\bar{X}_n((X_1, X_2, \ldots, X_n))$ as $\bar{X}$,

and its realisation $\bar{X}_n((X_1, X_2, \ldots, X_n))$ as $\bar{x_n}$.

By the properties of expectations that we have seen before,

$$E(\bar{X}_n) = E\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n}E\left(\sum_{i=1}^{n} X_i\right) = \frac{1}{n}\sum_{i=1}^{n} E(X_i)$$

And because every X_i is identically distributed with the same expectation, say $E(X_1)$, then

$$E(\bar{X}_n) = \frac{1}{n}\sum_{i=1}^{n} E(X_i) = \frac{1}{n} \times n \times E(X_1) = E(X_1)$$

We have already met the Sage pylab module which provides arrays and some useful statistical capabilities for our Lotto data. To be able to use these capabilities, it is easiest to convert our Lotto Data into a `pylab.array`.

```
from pylab import array, mean, var, std # make pylab stuff we need
accessible in Sage
listBallOne = getLottoBallOneData()    # make sure we have our list
arrayBallOne = pylab.array(listBallOne) # make the array out of the list
arrayBallOne     # disclose the array
```

Now we can find the sample mean of our Lotto data using the `mean` method of pylab arrays.

```
arrayBallOne.mean()        # sample mean statistic
```

```
pylab.mean(arrayBallOne)   # sample mean statistic; another way
```

If we go back to our definition of the sample mean $\bar{X}_n$

$$T_n((X_1, X_2, \ldots, X_n)) = \bar{X}_n((X_1, X_2, \ldots, X_n)) := \frac{1}{n}\sum_{i=1}^{n} X_i$$

we can see that we calculate it by summing all the data values X_i and then dividing the sum by the number of data values n. Instead of using the pylab magic to give us the mean, we could explicitly calculate it for ourselves using jsMath

Sage functions `sum()` and `len()`.

We can use `sum()` to add up (find the summation or sum of) the values in the list (this will give us $\sum_{i=1}^{n} X_i$), and we can use `len()` to find the length of the list which is the number of data values n.

```
sum(listBallOne)
```

```
# calculating the sample mean as a Sage rational the long way
sampleMean = sum(listBallOne)/len(listBallOne)
sampleMean
```

What has happened here?

If you ask Sage to divide one Sage.rings.integer by another, you will get the answer as a Sage.rings.rational.

```
type(sampleMean)
```

To convert from a Sage.rings.rational to a floating point real number, we can use the conversion function RR().

```
# calculating the sample mean as a Sage rational the long way
sampleMean = RR(sum(listBallOne)/len(listBallOne))
sampleMean
```

```
type(sampleMean)
```

```

```

Sample variance

From a given sequence of RVs $X_1, X_2, \ldots, X_n$ (or a random vector $X = (X_1, X_2, \ldots, X_n)$ we can obtain another RV called the **sample variance**.

$$T_n((X_1, X_2, \ldots, X_n)) = S_n^2((X_1, X_2, \ldots, X_n)) := \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2$$

Sometimes, we divide by $n-1$ instead of n.

The sample variance is a measure of spread from the sample.

We write $S_n^2((X_1, X_2, \ldots, X_n))$ as S_n^2,

and its realisation $S_n^2((x_1, x_2, \ldots, x_n))$ as s_n^2

Suppose that every X_i is identically distributed with the same variance, say $V(X_1)$. Then, using the definition of variance and the properties of expectations summarized by the following equalities:

- $V(X_1) = E(X_1^2) - (E(X_1))^2$
- $V(\bar{X}_n) = E(\bar{X}_n^2) - (E(\bar{X}_n))^2$
- Since $X_1, X_2, \ldots, X_n \overset{IID}{\sim} X_1$, $V(\bar{X}_n) = \frac{1}{n}V(X_1)$ and $E(\bar{X}_n) = E(X_1)$

jsMath

try to show that

$$E(S_n^2) = \frac{n-1}{n} V(X_1)$$

Here we use the `var` method of the pylab array to find the variance of the Lotto data.

```
arrayBallOne.var()          # sample variance statistic
```

```
pylab.var(arrayBallOne)     # sample variance by another way
```

```

```

Sample standard deviation

Similarly, we can look at a **sample standard deviation** = $\sqrt{\text{sample variance}}$

```
arrayBallOne.std()          # sample standard deviation statistic
```

```
pylab.std(arrayBallOne)     # sample standard deviation by another way
```

```

```

Double check that the sample standard deviation is indeed the square root of the sample variance

```
sqrt(arrayBallOne.var())    # just checking
```

```

```

Order statistics

Suppose that $X_1, X_2, \ldots X_n$ is a sequence of RVs.

Then, the n-sample order statistic $X_{([n])}$ is:

$$X_{([n])}((X_1, X_2, \ldots, X_n)) := (X_{(1)}, X_{(2)}, \ldots, X_{(n)})$$

such that

$$X_{(1)} \leq X_{(2)} \leq \ldots \leq X_{(n)}$$

For brevity, we write $X_{([n])}((X_1, X_2, \ldots, X_n))$ as $X_{[n]}$

and its realisation $X_{([n])}((x_1, x_2, \ldots, x_n))$ as $x_{([n])} = (x_{(1)}, x_{(2)}, \ldots, x_{(n)})$.

Thus, we simply sort the data to get the order statistic.

jsMath

For example, if we have the outcome of 3 $Bernoulli$ trials $x = (0, 1, 0)$,

then $x_{([3])} = (0, 0, 1)$.

For the Lotto data, we could easily sort our data using the `sort` method of a list, but watch out! This will sort our list *in-place*, which means that listBallOne will now be in the new, sorted, order. Very often, we want to keep our data in the original order. What if we wanted to know the result of the first draw, or the last draw? If we only have the sorted list we have lost this original ordering.

We have can do two things. First, we can use the `sorted()` function. This will return a new, sorted, copy of the list.

```
listBallOne = getLottoBallOneData()        # make sure we have our list
sortedListBallOne = sorted(listBallOne)
sortedListBallOne
```

We suggest that you use `sorted()` to make a sorted copy of a list. It is quicker than the copy-and-sort two-step process and you are less likely to make a mistake and sort the original list instead of the copy.

```
anotherSortedListBallOne = listBallOne[:]         # copy the list
anotherSortedListBallOne.sort()                   # sort the copy
anotherSortedListBallOne[1113]    # have a look at the last (maximum) value
in the sorted list
```

The kth order statistic of a sample is the kth smallest value in the sample. Order statistics that may be of particular interest include the smallest (minimum) and largest (maximum) values in the sample.

The pylab.array version of our data provides us with an easy way to get at the sample minimum and sample maximum statistics.

```
arrayBallOne.min()          # sample minimum statistic
```

```
arrayBallOne.max()          # sample maximum statistic
```

Frequencies

With lots of discrete data like this, we may be interested in the frequency of each value, or the number of times we get a particular value. This can be formalized as the following process of adding indicator functions of the data points $(X_1, X_2, \ldots, X_n)$:

$$\sum_{i=1}^{n} \mathbf{1}_{\{X_i\}}(x)$$

We have seen how we can use the Sage dictionary to give us a mapping from ball numbers in the list to the count of the number of times each ball number comes up. In the last worksheet we wrote a function to do this. Let us fully understand this function now.

jsMath

```
%auto
def makeFreqDict(myDataList):
    '''Make a frequency mapping out of a list of data.

    Param myDataList, a list of data.
    Return a dictionary mapping each unique data value to its frequency
count.'''

    freqDict = {} # start with an empty dictionary

    for res in myDataList:

        if res in freqDict: # the data value already exists as a key
                freqDict[res] = freqDict[res] + 1 # add 1 to the count
        else: # the data value does not exist as a key value
            freqDict[res] = 1 # add a new key-value pair for this new data
value, frequency 1

    return freqDict # return the dictionary created
```

Let us flex our Sage muscles by implementing another way to do the frequencies from a data list. We have learned about sorting a list and counting how many times a particular value occurs in a list (remember `count`: if we use `myList.count(0)` we will get the number of times 0 occurs in `myList`). We can now write a new implementation of the function to make a dictionary associating ball numbers with their frequencies. This function, unlike the one above, is not quite returning the $\sum_{i=1}^{n} \mathbf{1}_{\{X_i\}}(x)$.

```
%auto
def makeNewFreqDict(myDataList):
    '''Make a frequency mapping out of a list of data.

    Param myDataList, a list of data.
    Return a dictionary mapping each data value from min to max in steps of
1 to its frequency count.'''

    freqDict = {}             # start with an empty dictionary
    sortedMyDataList = sorted(myDataList)
    minData = sortedMyDataList[0]    # the minimum is the first value in the
sorted list
    maxData = sortedMyDataList[len(myDataList) - 1] # the maximum is the
last value in the sorted list
    dictKeys = range(minData, maxData+1, 1)  # make a list of values from
minData to maxData in steps of 1

    for k in dictKeys:
        freqDict[k] = myDataList.count(k)

    return freqDict # return the dictionary created
```

What do you need to note here?

- The function has a docstring (documentation string) that tells us what it does.
- The function parameter is called myDataList. This means that inside the function, this is the variable name th

jsMath

function is using for the list it is operating on
- We use `sorted()` inside our function to find the minimum and maximum values in the list, and then we use `range()` to create a list of values [min, min+1, ... max]
- This list of values will be the keys to our dictionary: the possible ball numbers that we want to find the counts for.
- We use a **for** loop to go through the list of key numbers one by one.
- Within the loop, we use `count()` to count the occurences in myDataList of the value assigned to k in that iteration of the loop
- And at the same time we can add a pair (key k, count of occurrences of k in list myDataList) to the dictionary.
- The resulting dictionary could be different to the way that we did it before: with this new method, if there is a value between min and max that is not in the list, we will have it as a key in the dictionary and explicitly give it a count of 0.

```
listBallOne = getLottoBallOneData()    # make sure we have our list
ballOneFreqs = makeFreqDict(listBallOne) # call the function
```

```
{1: 29, 2: 28, 3: 31, 4: 24, 5: 27, 6: 30, 7: 31, 8: 32, 9: 22, 10: 27,
11: 25, 12: 28, 13: 29, 14: 22, 15: 22, 16: 27, 17: 20, 18: 28, 19: 28,
20: 32, 21: 30, 22: 30, 23: 27, 24: 36, 25: 37, 26: 22, 27: 26, 28: 24,
29: 24, 30: 33, 31: 28, 32: 27, 33: 28, 34: 27, 35: 34, 36: 24, 37: 26,
38: 30, 39: 22, 40: 37}
```

```
listBallOne = getLottoBallOneData()    # make sure we have our list
ballOneFreqsNew = makeNewFreqDict(listBallOne) # call the function
```

```
{1: 29, 2: 28, 3: 31, 4: 24, 5: 27, 6: 30, 7: 31, 8: 32, 9: 22, 10: 27,
11: 25, 12: 28, 13: 29, 14: 22, 15: 22, 16: 27, 17: 20, 18: 28, 19: 28,
20: 32, 21: 30, 22: 30, 23: 27, 24: 36, 25: 37, 26: 22, 27: 26, 28: 24,
29: 24, 30: 33, 31: 28, 32: 27, 33: 28, 34: 27, 35: 34, 36: 24, 37: 26,
38: 30, 39: 22, 40: 37}
```

Empirical Mass Function

The **empirical mass function** (EMF) of a sequence of observed data $x_1, x_2, \ldots x_n$ is the sum of the following indicator functions:

$$EMF((x_1, x_2, \ldots, x_n)) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{x_i\}}(x)$$

What is this doing? This is adding normalized indicator functions of the data points $(X_1, X_2, \ldots, X_n)$:

$$\frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{X_i\}}(x)$$

The normalization part is the division by n, the total number of data points.

For example if we have x as the outcome of three $Bernoulli$ trials (like tossing a fair coin three times to see if we get heads), say $x = (1, 0, 1)$, then

$$EMF((x_1, x_2, \ldots, x_n)) = \frac{1}{3}(\mathbf{1}_{\{1\}}(x) + \mathbf{1}_{\{0\}}(x) + \mathbf{1}_{\{1\}}(x))$$

We can plot this by showing the height of a point as the *relative frequency* of occurences of that value. The relative frequency for a number is the count (frequency) for that number divided by the sample size (the sample size is the total number of trials). The relative frequency of some data value x_i appears in our EMF formula as $\frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{X_i\}}(x)$

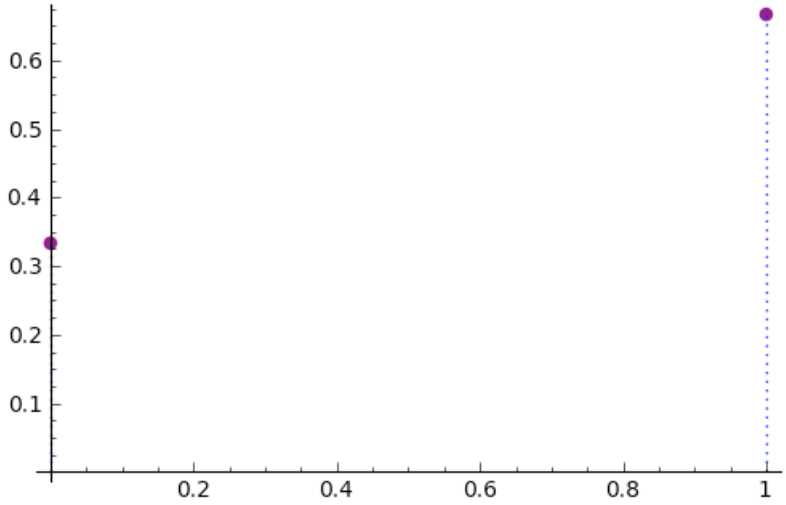
jsMath

Using our prepackaged and hidden makeEMF() function we can do this plot for this $Bernoulli$ trial data.

```
bernoulliOutcomes = (1,0,1)    # some data x

bernoulliRelFreqPairs = makeEMF(bernoulliOutcomes) # make a list of unique
values and relative frequencies
```

```
[(0, 0.333333333333333), (1, 0.666666666666667)]
```

```
bernoulliPlotEMF = point(bernoulliRelFreqPairs, rgbcolor = "purple",
pointsize="30")
for k in bernoulliRelFreqPairs:    # for each tuple in the list
    kkey, kheight = k     # unpack tuple
    bernoulliPlotEMF += line([(kkey, 0),(kkey, kheight)], rgbcolor="blue",
linestyle=":")
show(bernoulliPlotEMF)
```

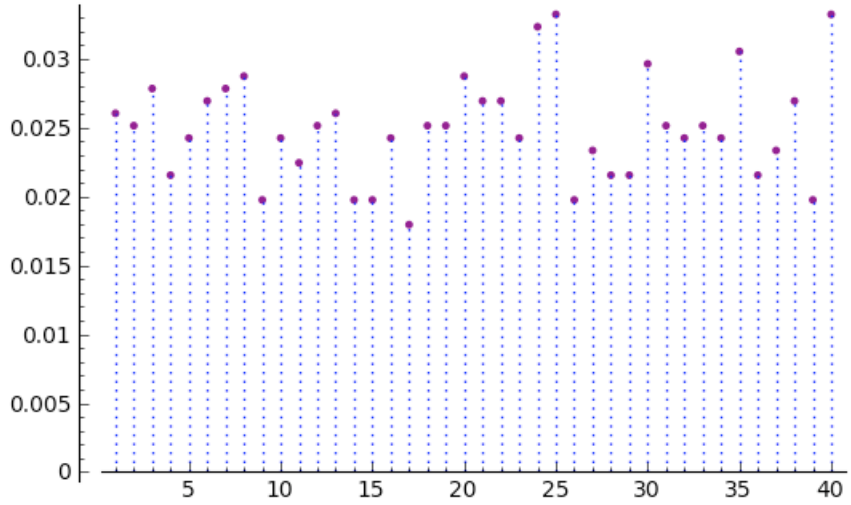


The dotted blue lines just help us to see what x value is associated with each point in the plot.

In the last worksheet, we showed an empirical mass function for the Lotto data.

```
listBallOne = getLottoBallOneData()    # make sure we have our list

# next we make a list of unique data values and their relative frequencies
numRelFreqPairs = makeEMF(listBallOne)
lottoPlotEMF = point(numRelFreqPairs, rgbcolor = "purple")
for k in numRelFreqPairs:    # for each tuple in the list
    kkey, kheight = k     # unpack tuple
    lottoPlotEMF += line([(kkey, 0),(kkey, kheight)], rgbcolor="blue",
linestyle=":")
```

jsMath



Empirical distribution function

Another extremely important statistics of the observed data is called the **empirical distribution function** (EDF). The EDF is the empirical or data-based distribution function (DF) just like the empirical mass function (EMF) is the empirical or data-based probability mass function (PMF). This can be formalized as the following process of adding indicator functions of the half-lines beginning at the data points $[X_1, +\infty), [X_2, +\infty), \ldots, [X_n, +\infty)$:

$$\widehat{F}_n(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{[X_i, +\infty)}(x)$$

This formula is saying that we are looking at the *cumulative* relative frequency for each of the values: not just how many data points had that value (normalized by n) but how many data points had that value or a smaller one (normalized by n).

Let's have a look at this for the simple of $Bernoulli$ example. We have bundled up the techniques we use in the hidden function called makeEDF so that you can just concentrate on the data for now.

```
bernoulliOutcomes = (1,0,1)    # some data x
bernoulliCumFreqPairs = makeEDF(bernoulliOutcomes) # make a list of unique
values and cumulative frequencies
bernoulliCumFreqPairs
```

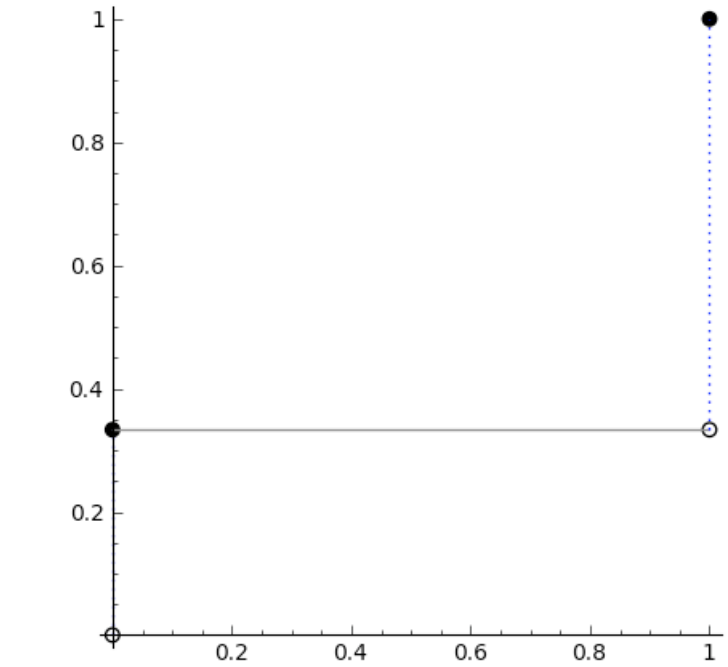
```
[(0, 0.33333333333333331), (1, 1.0)]
```

The two different values a $Bernoulli$ RV can take (in order from smallest to largest) are 0, 1. What is the cumulative relative frequency of the number of times we got a 0? It is $\frac{1}{3}$: we got 0 once and we had three datapoints in total, and there are no values that the RV can take smaller than 0. What is the cumulative relative frequency of the number of times we got a 1? We got a 1 twice so its relative frequency is $\frac{2}{3}$, which is what we showed with the EMF. However, for the cumulative relative frequency we add need to add to this the relative frequency of all values less than 1 (only zero in this case) so the **cumulative relative frequency** of 1 is $\frac{1}{3} + \frac{2}{3} = 1$.

We can plot this.

jsMath

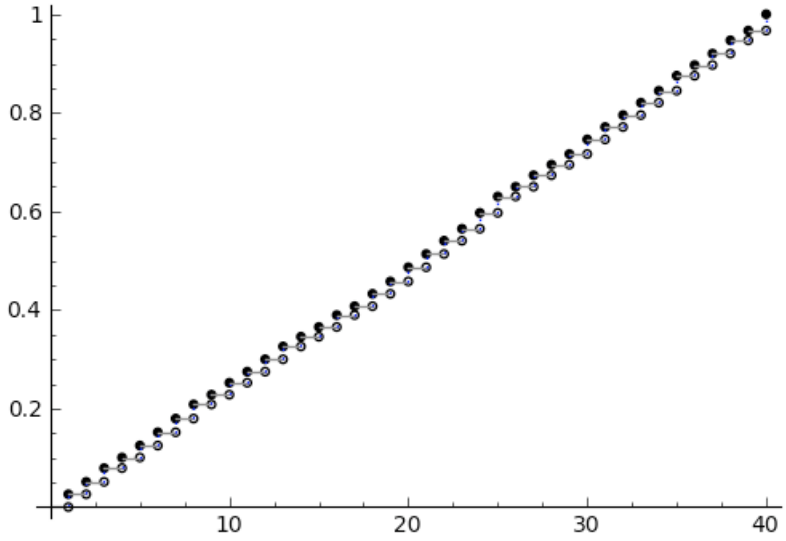
```
bernoulliOutcomes = (1,0,1)    # some data x

# next we make a list of unique values and cumulative frequencies
bernoulliCumFreqPairs = makeEDF(bernoulliOutcomes)
bernoulliPlotEDF = point(bernoulliCumFreqPairs, rgbcolor = "black", faceted
= true, pointsize="30")
for k in range(len(bernoulliCumFreqPairs)):
    x, kheight = bernoulliCumFreqPairs[k]     # unpack tuple
    previous_x = 0
    previous_height = 0
    if k > 0:
        previous_x, previous_height = bernoulliCumFreqPairs[k-1] # unpack
previous tuple
    bernoulliPlotEDF += line([(previous_x, previous_height),(x,
previous_height)], rgbcolor="grey")
    bernoulliPlotEDF += points((x, previous_height),rgbcolor = "white",
faceted = true, pointsize="30")
    bernoulliPlotEDF += line([(x, previous_height),(x, kheight)],
rgbcolor="blue", linestyle=":")
```



We showed last week how we can do the same for the Lotto data. This EDF plot has many more points on it because the first ball can have 40 possible numbers on it (1, 2, 3, . . ., 40)

jsMath

```
listBallOne = getLottoBallOneData()    # make sure we have our list
numCumFreqPairs = makeEDF(listBallOne)
lottoPlotEDF = points(numCumFreqPairs, rgbcolor = "black", faceted = true)
for k in range(len(numCumFreqPairs)):
    x, kheight = numCumFreqPairs[k]     # unpack tuple
    previous_x = 0
    previous_height = 0
    if k > 0:
        previous_x, previous_height = numCumFreqPairs[k-1] # unpack previous
tuple
    lottoPlotEDF += line([(previous_x, previous_height),(x,
previous_height)], rgbcolor="grey")
    lottoPlotEDF += points((x, previous_height),rgbcolor = "white", faceted
= true)
    lottoPlotEDF += line([(x, previous_height),(x, kheight)],
rgbcolor="blue", linestyle=":")
show(lottoPlotEDF, figsize=(6,4))
```



You try

List comprehensions

The **list comprehension** is a programming concept that allows us to represent lists (or sets, or tuples or other sequential containers) that are obtained by transforming each element of some given list and selecting a subset of them.

We have already talked about for loops. Sometimes when you are working with a list and a for loop seems to be the obvious way to create another list from the first list, a *list comprehension* is the easiest way to do it.

You can think of a list comprehension as a short specialised syntax which does a for loop on a list and creates another list.

We will illustrate this by staring with a simple list.

```
firstList = range(10)
firstList
```

Suppose that we want a list that contains all the elements in firstList multiplied by 2. We can use a **list comprehension**

to do this.

Remember that a list comprehension can be thought of as a short specialised syntax which does a for loop on a list and creates another list. The syntax is:

`[ some_expression_applied_to_each_element_in_list for element in list].`

Note the [ ] parentheses around the whole statement -- this is what makes the result into a list.

We will use a list comprehension to take each value in `firstList`, transform it (multiply by 2 in this case) and put the transformed value into a new list, `secondList`.

```
secondList = [2 * x for x in firstList]
secondList
```

Note also that although you might intuitively think that you could get a list containing every element in firstList multiplied by 2 with an expression like `2 * firstList`, you don't. Doing this will *replicate* the elements in firstList.

```
2 * firstList
```

We don't even have to explicitly create the first list - we can wrap making the list we are doing the list comprehension on into the list comprehension itself. Here we are creating a list of the powers of 2, $[2^0, 2^1, 2^4, \ldots, 2^{11}]$

```
anotherList = [2^x for x in range(11)]
anotherList
```

We can be choosy about what goes into the new list, selecting elements which elements to transform and put into the new list, by adding an **if** statement at the end. For instance, we can get the set of even numbers in the set {0,1,...,9} as follows.

```
range(0,10,1)
```

```
10%2==0
```

```
[x for x in range(0,10,1) if x%2==0]
```

```
evensList = [x for x in range(0,10,1) if x%2==0]
evensList
```

Let us do a more interesting comprehension where we transform each element x in the list [0, 2, ..., 8] by 2^x and then select the transformation of only those elements whose square, i.e., x^2, is divisible by 3. We see this list comprehension step-by-step below.

```
range(0,10,2)
```

```
[2^x for x in range(0,10,2)]
```

jsMath

```
yetAnotherList = [2^x for x in range(0,10,2) if (x^2)%3==0]
yetAnotherList
```

We used our hidden function, makeEMF, to help us to associate each of the numbers that appeared on Lotto ball one with the relative frequencies of its occurence(i.e., the number of times that number appeared divided by the total number of observations, 1114 in this case). We used a list comprehension to do this. First we made the dictionary that associated numbers with their counts. Then we summed up the counts to tell us how many data points there were in total.

```
listBallOne = getLottoBallOneData()    # make sure we have our list
freqs = makeFreqDict(listBallOne)     # make the frequency counts mapping
totalCounts = sum(freqs.values())    # total number of data points
```

Then we used a list comprehension to make a list of relative frequencies: for each frequency we divided it by the total frequency. The reason that we used `RR(totalCounts)` is to make sure that Sage evaluates the result as a floating point real. The RR() function converts from other numerical types to the Sage floating point real type.

```
relFreqs = [fr/RR(totalCounts) for fr in freqs.values()] # use a list
comprehension
numRelFreqPairs = zip(freqs.keys(), relFreqs) # zip the keys and relative
frequencies together
numRelFreqPairs
```

Try doing a list comprehension to get a list $[2^3,\ 4^3, 6^3,\ 8^3]$

```

```

```

```

```

```

You try

Example 2: New Zealand earthquakes

Now we are ready to play with some more real data - earthquakes. First of all we will use a file of data from before the earthquake. This is already attached to this worksheet (you will learn more about attaching data shortly...). For the moment, just evaluate this cell so that you have assigned the filename to the variable named `myFilename`.

```
myFilename = 'earthquakes_1July2009_19Mar2010.csv'
```

Both Python and Sage provide lots of different ways of reading data from files. In this case we are going to use the same function to get the data as we used for the Lotto data. This is one we wrote ourselves. It is a little bit more complicated than you need for this course so again we have hidden it away so that you can just concentrate on the data itself.

jsMath

```
# using the function
# we have already set the variable myFilename to our data file
myData = getData(myFilename,headerlines=1,sep=',')
```

```
help(getData)
```

docs-0.html

How many rows and columns have we got?

```
pylab.shape(myData)
```

What does the array look like? When Sage is asked to display large arrays, it will automatically truncate the output - what you see here is the first and last three columns of the first and last three rows. The data that is not shown is indicated with the . . . characters.

```
myData
```

The getData(...) function was a general function, designed just to get data from a file. It did not have to know anything about the format of the earthquake data file. When we want to process the array we have made, however, we do have to know about our data. If we want to extract the magnitude data, for instance, we have to know which column it is in. We made a function to take the column containing the magnitude data and return us a list of the magnitude data omitting all the `nan` (not a number) values. You can look at this function in the Optional Exploration part of the worksheet if you want to, but it is not essential for this course to understand it.

First have a look at the documentation for our function:

```
help(makeMagList)
```

docs-0.html

And now get the data:

```
listMags = makeMagList(myData) # use our makeMagList function to make a list
of magnitudes
```

When you evaluate the cell below, you will probably find that the length of the list is less than the number of rows in the array. This is because the makeMagList function misses out any `nan` (not a number) values in the array. If you look at your data in Excel you will see that many rows do not have magnitude data. Note that '*no data*' is very different to '*magnitude = 0*'!

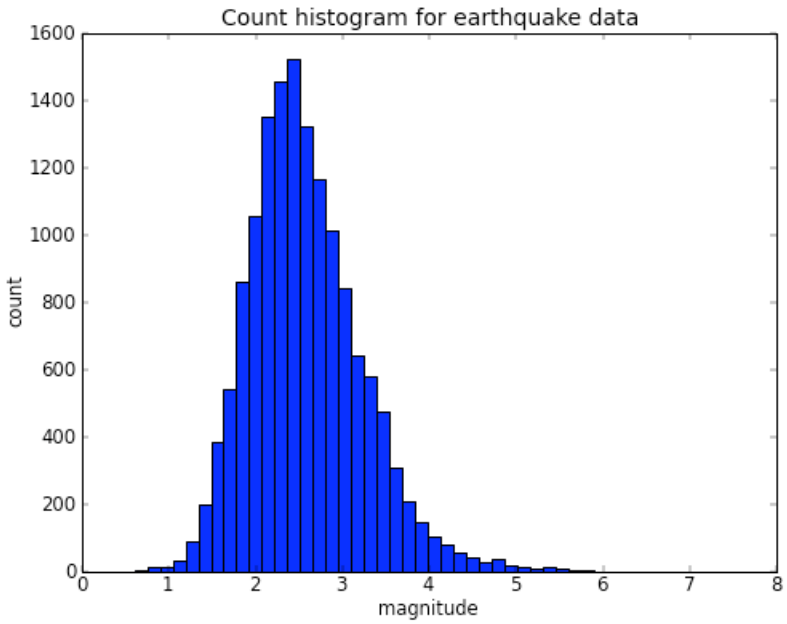
```
len(listMags)
```

We can use the pylab.hist function to make a histogram of the data. We will be doing more on histograms later. For the moment, just evaluate and admire your earthquake magntitudes histogram.

jsMath

```
pylab.clf() # clear current figure
n, bins, patches = pylab.hist(listMags, 50) # make the histogram (don't have
to have n, bins, patches = ...)
pylab.xlabel('magnitude') # use pyplot methods to set labels, titles etc
similar to as in matlab
pylab.ylabel('count')
pylab.title('Count histogram for earthquake data')
pylab.savefig('myHist',dpi=(70)) # save figure (dpi) to display the figure
pylab.show() # and finally show it
```

evaluate



We can also do things like finding the maximum magnitude in the data set we brought in.

```
max(listMags)
```

And the minimum.

```
min(listMags)
```

If we were interested in finding out more about the row of data that had the largest magnitude, we could do have a look at the whole row from the complete array. You don't need to know how to do this for this course, but if you are interested, this code extracts all rows where the value in the 12th column is greater than 7.0

```
myData[:,:][myData[:,11]>7.0]
```

```

```

We also made a more complicated function to get the latitude and longitude coordinates for each earthquake so that we can plot them on a scatter plot. Again, you can have a look at it in the Optional Exploration section if you really want to, but you do not need to be able to write it for this course!

jsMath

One thing important thing though is that we learned last year that some of the latitude and longitude data shows that earthquakes are well outside New Zealand's land or oceanic territory. Therefore we included a way to specify that we only want to get data that is within certain latitude and longitude ranges. Here we ask for data with latitude between -50 and -30, and longitude between 150 and 200 (inclusive) . We also ask only for data with magnitude $\geq$ 2.0

Use the function on our array `myData`.

```
listCoords = makeCoordList(myData, -50, -30, 150, 200, 2.0) # use our
makeCoordList function to make a list of coordinates (and magnitudes)

len(listCoords[0]) #how much data have we got
```

```
11957
```

The variable named `listCoords` has now been assigned to a list of three lists: one list for latitudes, one for longitudes, and one for magnitudes.

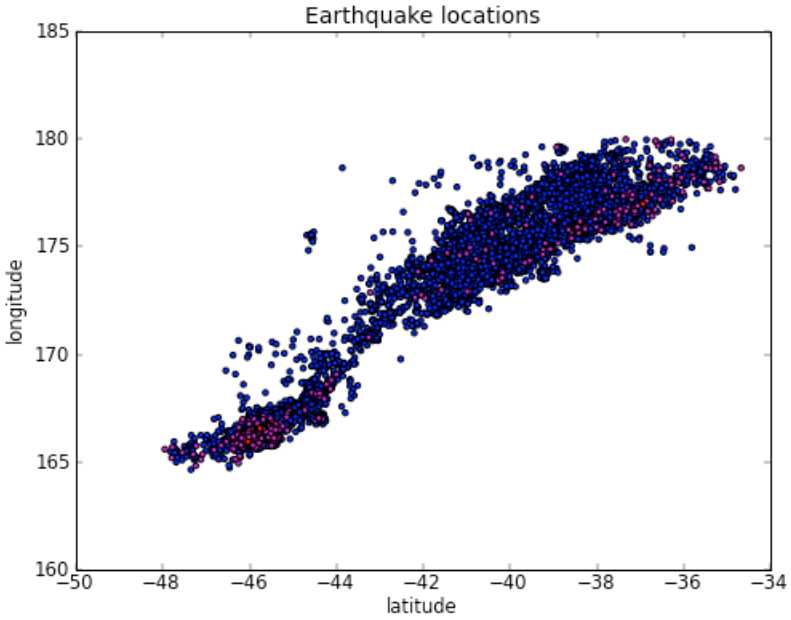
Now we are going to use the magnitudes to make a list of colours. The `elif` keyword means 'else-if' and is a way of making a more complex conditional statement checking multiple conditions (it is not essential material for this course, but is useful here).

```
clist = [] # make an empty list to hold colours depending on the magnitudes
for m in listCoords[2]: # go through the third list in listCoords, ie the
magnitudes
    if m == None:
        clist.append('k') # 'k' is black
    elif m > 6:                                  #elif is short for else-if
        clist.append('r') # 'r' is red
    elif m > 4:
        clist.append('m') # 'm' is magenta
    elif m > 2:
        clist.append('b') # 'b' is blue
    else:
        clist.append('y') # 'y' is yellow
```

Now we can plot the latitude/longitude coordinates as a scatter plot, with the colour of the point obtained from the list of colours we just created. We will do more on plotting later in the course.

```
pylab.clf() # clear current figure
pylab.scatter(listCoords[0],listCoords[1],s=10,c=clist) # make a scatter plot
pylab.xlabel('latitude') # use pyplot methods to set labels, titles etc
similar to as in matlab
pylab.ylabel('longitude')
pylab.title('Earthquake locations')
pylab.savefig('myScatter',dpi=(70)) # save figure (dpi) to display the figure
pylab.show() # and finally show it
```

jsMath



Try changing the minimum magnitude we want to show to see what effect this has on your plot (the minimum magnitude is the last argument passed to the `makeCoordList` function, for example if we use `makeCoordList(myData, -50, -30, 150, 200, 3.0)` then 3.0 is the minimum magnitude we will get.

```
help(pylab.scatter) # evaluate the cell to get a sense for the capabilities
of the scatter plot function in pylab
```

You try

Example 2 continued: Recent New Zealand earthquakes

So far, we have made actually getting the data very easy for you. For this, you are going to do a little more work.

First we are going to download some New Zealand earthquake data from the GeoNet Quake Search catalog site. Please follow the instructions below carefully:

1. Make sure that this will allow you to specify where you want to save files to (in Mozilla, go to the Tools menu, select Options, then Main, then select 'Always ask me where to save files' and click OK)
2. Go to the site http://magma.geonet.org.nz/resources/quakesearch/ (this will open in a new browser window or tab so that you can still use this worksheet)
3. Put in a date range. We suggest from 1 January 2011 to the latest date available. There are lots of readings each day so if you ask for any more you'll be swamped in data. This applies in particular if you ask for data from around 4 September 2010 - if you do that then only get data from say 1 Sept 2010 to 30 September 2010.
4. You can leave the filters by location, magnitude and depth blank, or get magnitudes greater than say 2.
5. Keep the default export format (CSV, Time, location, z, mag). CSV stands for Comma Separated Variable and is a useful format for data files.
6. Click Submit
7. When you are asked if you want to open the file in Excel or save to disk, we suggest that you save the file to somewhere on your P: drive, somewhere like your STAT221 folder. Keep it as CSV format and give it an

informative filename (for example our filename for our earlier data was `earthquakes_1Jul2009_19Mar2010.csv`).

Look at the data in Excel. Where it is practical to do so, it is always useful to be able to actually have a look at your datafile before you start trying to process it. Sometimes, if the file is huge, this may not be easy, but somehow you need to know things like how many header lines there are at the top (the lines before the data starts), what is the layout of the file, how tidy it all is ...

Now you need to actually upload your data file to the SAGE Notebook server so that you can access it from your worksheet. Please follow the instructions to do this carefully:

1. At the top left of the worksheet, click on the dropdown box Data...
2. Select the option 'Upload or create file'
3. In the next screen, click Browse to browse for the data file you just created and saved on your P: drive.
4. When you have got the file location, click the Upload File button. Your data will be uploaded to the Sage server and will be available for your worksheet.
5. You will get a screen which shows your data. Read the information at the top.
6. It is useful here to take a copy of the text that tells you where your file is. It will look like `DATA+'earthquakes_1July2009_19Mar2010.csv'` (copy it by selecting the text with your mouse and then copying).
7. Click on the 'Worksheet' button in the row of buttons at the top right of the worksheet. This will take you back to your worksheet.
8. In the cell below, replace the text `'earthquakes_1July2009_19Mar2010.csv'` with the text you copied just now (including the two quote marks ' ')

Now if you click evaluate for the cell below, the datafile will be assigned to a variable named `newFilename`.

```
newFilename = 'earthquakes_1July2009_19Mar2010.csv'
# using the function
# we have already set the variable newFilename to our data file
newData = getData(newFilename,headerlines=1,sep=',')
pylab.shape(newData) # check the shape (how many rows and columns)
```

Make the list of magnitudes, as we did before

```
newMags = makeMagList(newData) # use our makeMagList function to make a list
of magnitudes
```

You could check the minimum and maximums, like we did before:

```

```

```

```

Make a histogram:

```

```

jsMath

```
pylab.clf() # clear current figure
n, bins, patches = pylab.hist(newMags, 50) # make the histogram (don't have
to have n, bins, patches = ...)
pylab.xlabel('magnitude') # use pyplot methods to set labels, titles etc
similar to as in matlab
pylab.ylabel('count')
pylab.title('Count histogram for earthquake data')
pylab.savefig('myHist') # seem to need to have this to be able to actually
display the figure
pylab.show() # and finally show it
```

Make the coordinates lists, like we did before (again, excluding data which is well outside New Zealand and setting the minimum magnitude to 2)

```
newCoords = makeCoordList(newData, -50, -30, 150, 200, 2.0) # use our
makeCoordList function to make a list of coordinates (and magnitudes)

#how much data have we got?
```

```
colList = [] # make an empty list to hold colours depending on the magnitudes
for m in newCoords[2]: # go through the third list in newCoords, ie the
magnitudes
    if m == None:
        colList.append('k') # 'k' is black
    elif m > 6:                                #elif is short for else-if
        colList.append('r') # 'r' is red
    elif m > 4:
        colList.append('m') # 'm' is magenta
    elif m > 2:
        colList.append('b') # 'b' is blue
    else:
        colList.append('y') # 'y' is yellow
```

```
pylab.clf() # clear current figure
pylab.scatter(newCoords[0],newCoords[1],s=10,c=colList) # make a scatter plot
pylab.xlabel('latitude') # use pyplot methods to set labels, titles etc
similar to as in matlab
pylab.ylabel('longitude')
pylab.title('Earthquake locations')
pylab.savefig('myScatter') # seem to need to have this to be able to
actually display the figure
pylab.show() # and finally show it
```

```

```

```

```

You try

A revision on functions

jsMath

We are going to be using functions quite a lot from now on, so we will get a bit more formal about how we write them.

You can think of functions as **reusable** procedures for doing useful things that you might want to have available in lots of other, more specialised, sections of code. The beauty of a function is that it is a way of **generalising** your program instructions: When you want to use a function you pass it the value or values you want it to use and (usually - see below) get back an answer which has been calculated using those values.

In Sage we can define our own functions. We can write our own reusable, general procedures to do useful, frequently needed, things for us.

Think of writing some code to sort out who out of your classmates can join your basketball team. It is a very politically incorrect heightist team, unfortunately, so only people 1.5m tall or more can join, and also they have to be available on Tuesday nights. You've got this information in the form of records for each person (note the use of **tuples** for the records).

In the cell below we create records for 12 classmates.

```
%auto
# the auto command just makes sure the cell is automatically evaluated

# general classmate record format is (name, height, availability on
Tuesdays)
myClassmate1 = ("Fred", 1.2, true)
myClassmate2 = ("Freda", 1.7, false)
myClassmate3 = ("Mary", 1.5, true)
myClassmate4 = ("Xin", 1.2, true)
myClassmate5 = ("Andy", 1.6, false)
myClassmate6 = ("Hana", 1.7, true)
myClassmate7 = ("Mark", 1.6, true)
myClassmate8 = ("John", 1.7, true)
myClassmate9 = ("Jiao-nu", 1.5, true)
myClassmate10 = ("Sancho", 1.7, true)
```

Each classmate has a record encoded as a tuple containing 3 pieces of information (name, height in metres, and availability on Tuesday evenings) about the classmate it represents.

Now we want to find out who should be in our team. We can index into the tuple (remember the indexing operator `[ ]`) to get at the values in the first, second and third positions:

- `myClassmate1[0]` gets the value at the first position of the tuple `myClassmate1`, which is the name,
- `myClassmate1[1]` gets the value at the first position of the tuple `myClassmate1`, which is the height,
- `myClassmate1[2]` gets the value at the first position of the tuple `myClassmate1`, which is the availability on Tuesday night as a boolean value (true or false),

and we could 'hard-code' the whole thing:

jsMath

```
# first one
if (myClassmate1[1] >= 1.5) & myClassmate1[2]:
    print myClassmate1[0], "is in the team"
else:
     print myClassmate1[0], "cannot be in the team"

# second one
if (myClassmate2[1] >= 1.5) & myClassmate2[2]:
    print myClassmate2[0], "is in the team"
else:
     print myClassmate2[0], "cannot be in the team"

# third one
if (myClassmate3[1] >= 1.5) & myClassmate3[2]:
    print myClassmate3[0], "is in the team"
else:
     print myClassmate3[0], "cannot be in the team"
```

As you can see, that is **repetitive to code, boring,** and **very inflexible**. In fact, whenever you see repeated lines in your code that seem to be doing the same basic thing with different values, it is a good signal that it is time to write a function.

Here is an expanded version of the information we gave you in an earlier worksheet about defining a function:

1. The function name is preceded by the keyword `def` for definition.
2. It is recommended to make sure that the function name gives some very concise indication of what it does.
3. After the function name you put the function parameters within a pair of parentheses `( )`.

   The function **parameters** are variable names that are used within the body of the function, doing with them whatever it is you want the function to do. We'll come back to them in a minute....
4. After we have defined the function by its name followed by its parameters in parentheses, we end the line with a colon `:` before continuing to the next line.

   ```
   def myFunctionWithAGoodName(parameters):
   ```
5. Now, we are ready to write the content of the function. This must be indented and it is customary to leave 4 white spaces (as we have with all our other indentations). SAGE does this for you automatically if you have ended the line above with a colon `:`.
6. It is a matter of courteous programming practice to have a docstring for your function, i.e., comments on what the function does. The whole docstring is enclosed in a set of three single quotes (e.g., '''dostring'''). The docstring is returned when we ask SAGE for help on the function. The general format is to have one single line, ending in a full stop (period) which gives a concise description of the function, followed by a blank line, followed by more detail.

   ```
   def myFunctionWithAGoodName(parameters):
       '''This is the consise summary.

       Followed by more detail ...'''
   ```
7. Inside the body of the function we do whatever it is we want to do. Most of the time, but not always, we want the function to tell us the result of these calculations, i.e. we want the function to *return* something when it has finished. We do this with the keyword `return` followed by the expression to be returned.
8. In the SAGE Notebook you can define a function in one cell, evaluaate it, and then use the function in other cells further down. More generally, if you want to put more code in the same cell, below the body of the function (i.e., code that is not part of the function body) you have to move the start of the line backwards from the indentation that indicated the function body, so that it aligns with the `def` keyword that started the function definition.

Here is an example function to print out a line which states whether a classmate is in the team by testing whether the height in the classmate record is $\geq$ 1.5m and the availability on Tuesday nights in the classmate record is true.

```
def printInTeamFirstAttempt(classmate):
    '''A function to take a classmate tuple and print out if they are in the
team.

    Param classmate is the classmate to evaluate.

    Test classmate height against minimum height of 1.5m and print statement
to say if classmate is in team.'''

    if (classmate[1] >= 1.5) & classmate[2]:    # remember that classmate[2]
is availability on Tuesday night
        print classmate[0], "is in the team"
    else:
        print classmate[0], "cannot be in the team"
```

That is a good start but... What if we used our function for all our classmates and found that we did not get enough for a team. Maybe then we'd have to lower our minimum height, say to 1.4m? But then we would have to write a new function, with the minimum height of 1.4m embedded itside it. More repeated code - arrghghg. Instead, we could rewrite the function above to generalise it by making the minimum height to test against a **parameter** of the function. This means that the function itself does not have the minimum height to use embedded inside it, but instead you can tell the function what minimum height to use.

```
def printInTeamSecondAttempt(classmate, minHeight):
    '''A function to take a classmate tuple and print out if they are in the
team.

    Param classmate is the classmate to evaluate.
    Param minHeight is the minimum height to test against.

    Test classmate height against minHeight and prints statement to say if
classmate is in team.'''

    if (classmate[1] >= minHeight) & classmate[2]:
        print classmate[0], "is in the team"
    else:
        print classmate[0], "cannot be in the team"
```

Now you need to really think about what is going on here. The function has **parameters** `classmate` and `minHeight`. These are named in the parentheses following the `def` keyword and the function name `printInTeamSecondAttempt`. The code inside the body of the function uses these names in its calculations.

When we define the parameters, we are defining some variable names that the function knows about and can use. These are called **local variables**: they are local to (specific to) the function. This is what makes the function an efficient way of dealing with different classmates and different values for minHeight.

When our code says `printInTeamSecondAttempt(myClassmate1, mh)` we are **calling** the function and **passing** it the **arguments** `myClassmate1` and `mh`. Inside the function it will assign the record `myClassmate1` to a local variable `classmate` (its first parameter name) and it will assign the value assigned to `mh` to a local variable `minHeight` (its second parameter name).

On the next line, when our code says `printInTeamSecondAttempt(myClassmate2, mh)` we are **calling** the function again and this time **passing** it the **arguments** `myClassmate2` and `mh`. Inside the function it will now assign the record `myClassmate2` to a local variable `classmate` and it will assign the value assigned to `mh` to a local variable `minHeight` (its second parameter name) to .

jsMath

```
mh = 1.5 # minimum height
printInTeamSecondAttempt(myClassmate1, mh)
printInTeamSecondAttempt(myClassmate2, mh)
printInTeamSecondAttempt(myClassmate3, mh)
# etc etc etc
```

Here is an example where we are passing a different value for the minimum height to the function:

```
mh = 1.0 # minimum height
printInTeamSecondAttempt(myClassmate1, mh)
printInTeamSecondAttempt(myClassmate2, mh)
printInTeamSecondAttempt(myClassmate3, mh)
# etc etc etc
```

You'll notice that the `printInTeamSecondAttempt(...)` function does not actually return anything, it just prints out a line. It has given us a bit more flexibility, but what if we don't want to print things out? We might want to use this function in the middle of a huge program that deals with lots and lots of classmate tuples to make lots of different teams. Thousands of lines of output are not going to be helpful.

What about changing the function to **return** something? We could **return** a boolean value, ie `true` or `false`, to say whether the classmate tuple passed as the argument to the function meets our criteria.

```
%auto
# the auto command just makes sure the cell is automatically evaluated

def isInTeam(classmate, minHeight):
    '''A function to take a classmate tuple and checks if they can be in a
team.

    Param classmate is the classmate to evaluate.
    Param minHeight is the minimum height to test against.
    Return true if classmate tuple meets criteria, false otherwise.'''

    retValue = false            # a default value for the return value
    if (classmate[1] >= minHeight) & classmate[2]:
        retValue = true         # change the default return if classmate
```

Lets look at the this alternative. Functions that return boolean values are often given a name that starts with `is`... It's just a kind of convention and it means that you, as a programmer, can just see a function name like `isInTeam(...)` and, without even looking at the docstring, be pretty certain that it is going to return a boolean value, i.e. true or false.

Here we can see the keyword **`return`**, followed by the expression for the value we want to return. In this case we are returning the value the local variable named `retValue` is assigned to.

Note that this function uses a kind of shortcut to do the `if.. else` calculation that `printInTeam()` did. It specifies a default value for what it is going to return (we have assigned this to the variable name `retValue`). The default is `false`. The function body only changes default to `true` if the team membership criteria are satisfied. Doing this is a good way of programming for two reasons: First, in this case it saves writing the 'else' block and, in general, shorter code is faster and easier to understand. Secondly, it makes sure that we return a safe default value no matter how complex our function then gets.

In the next cell we call the function `isInTeam` and pass it the arguments `myClassmate1` and `mh`.

Try changing the value that the variable `mh` is assigned to.

jsMath

```
mh = 1.5 # use our original minimum height again
isInTeam(myClassmate1, mh)
```

`isInTeam` is a function.

```
type(isInTeam)
```

See what has happened to your lovely documentation string - it is now the 'help' file for the function!

```
help(isInTeam)
```

docs-0.html

When we call `isInTeam` we will get back a value. The function call gives a value whose type is the type of the value returned by the function (in this case, it is a boolean type).

```
type(isInTeam(myClassmate1, mh))
```

The function call is **returning** a value (`true` or `false`).

Our new function also does not give us the names, so why is it better than printInTeam()? Because, by returning a boolean value, it is more flexible than a function that just prints something out.

Lets create the kind of situation we might have if we had lots of classmate record tuples to deal with. We'd have them in some kind of *collection*, say a list (maybe an array, but we'll use list here):

```
%auto
# this just makes sure the cell is automatically evaluated
classmateList = [myClassmate1, myClassmate2, myClassmate3, myClassmate4,
myClassmate5, myClassmate6, myClassmate7, myClassmate8, myClassmate9,
myClassmate10, myClassmate11, myClassmate12]
```

`classmateList` is a list of tuples, each tuple containing 3 pieces of information (name, height in metres, and availability on Tuesday evenings) about the classmate it represents. Now say we wanted the same kind of print out that we had before. We can use a for loop on the list, like this:

```
mh = 1.5 # use our original minimum height again
for cm in classmateList:
    if isInTeam(cm, mh):
        print cm[0], "is in the team"
    else:
        print cm[0], "cannot be in the team"
```

The if clause of the for loop is *calling* the function `isInTeam(...)` and using the boolean value that is returned to decide whether to execute the if block code (`print cm[0], "is in the team"`) or the else block code (`print cm[0], "cannot be in the team"`).

`cm` is the name we have used for the loop variable, and each value (i.e., tuple) in the list is assigned to the variable `cm` in turn. `cm[0]` indexes into the tuple the loop variable `cm` is currently assigned to and gets the name of the person that tuple represents.

jsMath

But we can also use the *same* function in lots of different ways. What about using it to actually make our team?

```
myTeam=[]
mh = 1.5 # use our original minimum height again
for cm in classmateList:
    if isInTeam(cm, mh):
        myTeam.append(cm[0])
```

We can then ask Sage to tell us what our team is:>/span>

```
myTeam
```

You may also have seen that we could make this code shorter by just using a **list comprehension**:

```
mh = 1.5 # use our original minimum height again
myTeam = [cm[0] for cm in classmateList if isInTeam(cm,mh)]
myTeam
```

What if we wanted to make lots of different teams? Write a function to do it!

```
def makeTeam(clist, minValue):
    '''Return a basketball team from a list of classmate records and a
minimum height.

    Param clist is a list of classmate records in the form (name, height in
metres, availability).
    Param minValue is the minimum measurement criteria for the team.
    Return a list of names of classmates in the team.'''

    return [cm[0] for cm in classmateList if isInTeam(cm,minValue)]
```

The function `makeTeam(...)` has two parameters, `clist` and `minValue`. It uses a list comprehension to make the team list. The list comprehension calls the function `isInTeam(...)`, as described above for the for loop.

Note that because the list comprehension allows us to make the team list with a single expression, the function body only has to have the `return` keyword and the list comprehension expression that makes the list to return (if you used the for loop method the function body would have to first make the list using the for loop and then return the list). Most functions do have more than one line in the function body, but occasionally a one-line function is all you need.

Try out the function.

```
mh = 1.5 # use our original minimum height again
myTeam = makeTeam(classmateList, mh)
myTeam
```

Example 3: Functions for earthquake data

Lets think about processing file of earthquake data. There are things we want to do to the file (open it, read the data, turn the data into some kind of array, return the array). When we say "read the data" and "turn the data into some kind of

array", what do we mean by "the data"? Well, the data in the file consists of lines, one for each earthquake record. At the top of the file is a line which is the column headings.

```
fn = 'earthquakes_1July2009_19Mar2010.csv'
myFile = open(DATA+fn) # open the file - myFile is now a file object
for c in range(6):    # for loop counting 0 to 5
    myFile.readline()    # read successive lines of the file
```

```
myFile.close()   # don't have to but good practice
```

The file consists of separate lines (the '\n' character is a new line character, signifying the end of a line). The first line is the column headings. The lines after that are the records for each earthquake. You can see that although Excel will display the file to you in rows and columns, in fact what we really have is lines with commas to split up the separate components of the line.

So to "read the data" we want to take each line after the header line and turn it into a row in our 2-dimensional array of data. To do this we need to split up the line, and the signal for where to split is the comma. We also want to strip off the \n\ newline character at the end of each line. We are going to do this for each line (after the header line) in the file. This is a situation where we might first write a function to process a single line of data. In the cell below we have written a simple version of the way that we could process a line from our earthquakes file. Note how general this function is: we could use it on almost any file provided that it had lines that needed to be split up.

```
def processLineSimple(line, sep,):
    '''Process a line of data by splitting it up according to supplied
separator.

    Param line is the line to be split up.
    Param sep is the separator on which to split the line.
    Return a list of the numbers in the line, using nan for not a number,
        or None if there is an error processing the line.'''

    rlist=[] # default list
    # split is a method that allows you to split a string up into elements
separated by sep
    # rstrip removes trailing newline character from end of line
    line = line.rstrip()
    # we can also use it to strip off specific characters at the end of the
line
    line = line.rstrip(',')
    listStrs = line.split(sep)
    # need to use a try block to deal with data that cannot be turned into
floats
    # this includes empty strings and an other non-numeric data
    try:
        rlist=[safeFloat(x) for x in listStrs] # list comprehension
        #print 'Accepted', rlist
    except (ValueError, TypeError), diag:
        #print 'Omitted', line
        rlist = None

    return rlist;
```

Now when we want to deal with the whole file can use our `processLineSimple` function. You are not expected to write a function like this, but it will help your understanding of functions if you look at how it uses the `processLinesSimple` function we have just written to process each record in the file. In words, this function does the

following:

- Make a return value variable with default value of None (a special Python value)
- Open the file
- Read one line, the header line, and count how many elements there are in it. We expect each line to have at least this number of elements.
- Set up a temporary list to hold lists representing each record.
- For each of the remaining lines in the file:
  Use `processLineSimple` to process it;
  Check that we got back a list of record data and if so append it to the temporary list of record lists
- Turn the temporary list of lists into an array.
- Return the array.

```
import pylab # make sure we know about pylab

def getDataSimple(filename, sep, maxRecordsToGet):
    '''Open the file filename and return a pylab.array of the contents as
floats.

       Param filename is the filename to get the data from.
       When used in the SAGE notebook interface, the data file must be
uploaded
               to the worksheet.
       Param sep is the separator for data.
       Param maxRecordsToGet is the maximum records to get.

       getDataSimple only deals with 2-d numeric data.
       getDataSimple assumes there is one header line.
       There are no checks on file being openable or readable in this simple
version.
       getDataSimple will use nan for non-a-number values in the data.

       Returns a pylab.array of the data as floats.'''

    retArray = None # a default return value

    myFile = open(DATA+filename) # open the file - myFile is now a file
object

    headers = myFile.readline() # read the first line, assumed to be column
headers

    tempList = [] # start with an empty list

    lineCounter = 0 # variable for counting lines
    for line in myFile:  # this for loop reads each line of the file one by
one
        dataRow = processLineSimple(line, sep)
        if dataRow != None:    # check that the processing has not returned
None
            tempList.append(dataRow) # all okay, add to the templist

        lineCounter = lineCounter + 1
        if lineCounter > maxRecordsToGet: # if we have got enough lines
            break     # break out of the for loop
```

Now you can use the getDataSimple function to turn some of the earthquake data into an array. We are restricting the data to the first five records so that you see all of those records. Notice that some values are `nan`. That stands for not a number and indicates that those value were missing in the original data file (you can confirm that by looking at the jsMath

Excel).

```
recordsToGet = 5
myFn = 'earthquakes_1July2009_19Mar2010.csv'
shortData = getDataSimple(myFn, ',', recordsToGet)
shortData
```

Just to demonstrate how flexible these function are, here is another file of data. It is a record of an informal experiment a family did with their five cats to try to see which of two cat foods their cats liked. They timed how long it took each cat to each a meal of each of the two brands of cat food and recorded the results in the file CatFoodTime.txt. Have a look at the file.

```
myFn = 'CatFoodTime.txt'
myFile = open(DATA+myFn) # open the file - myFile is now a file object

for c in range(6):    # for loop counting 0 to 5
    myFile.readline()    # read successive lines of the file
```

```
myFile.close()
```

Now use exactly the same `getDataSimple` function to get the cat food data:

```
recordsToGet = 6
myFn = 'CatFoodTime.txt'
shortData = getDataSimple(myFn, None, recordsToGet)
shortData
```

```

```

You should now understand a bit better what we said about functions. They allow us to write flexible, reusable code, i.e. to avoid repetition and build larger blocks from smaller blocks. There is one other use for functions even if you don't think that you'll want to use the function in lots of different circumstances. This is just being able to **break down your code** into manageble blocks (functions), each of which fulfils a clear, self-contained purpose. This is described as **modular programming**. This kind of code is much easier to read and understand and maintain. You will also almost inevitably find that you reuse key blocks of code more than you thought you would when you first wrote it, so it's useful to make nice functions right from the start. Function **parameters** allow us to pass arguments to functions, and the function usually passes back, or *returns*, the result of its calculations. You can nest function calls within functions - that is what building up from your blocks is all about.

Think about how you could use a similar function to take lots of tuple records on people and calculate statistics like the proportion of a class who are over a certain height. Then think a bit wider - if you have some sort of data, you can use a function to do some analysis on the data and then summarise the results.

Optional exploration

Only do this if you have done all the You Try sections above and are happy that you understand everything. This section is optional.

Using files

jsMath

If you are interested in using files with Sage (or Python), have a look at the Python documentation about reading and writing files.

More on functions for reading earthquake data

When Sage reads in a csv file, as we read in the earthquake data, it treats all the values as strings. This is good in some ways -- our data could be a mixture of string columns and number columns and it gives us the chance to decide how to deal with the data that we expect. Our `getData` function was written just to handle numeric values - we did not expect anything that could not be converted to a number. Sage provides easy ways of converting from strings of numerical characters to number types, so we can just convert the columns of strings we know should represent numbers into number types. But, what if something has gone wrong in the file and something we expect to be a number contains an non-numeric character? This can happen easily, for lots of reasons. One of the most obvious is that there was simply no data in the original file. This comes through as an empty string or '' (that is two single quote marks with nothing between them). Therefore, we really want to use a safe way of converting from a string to a number -- one which will fail 'gracefully' rather than just fall over and die.... If you have spent half an hour of computer processing time reading in a huge file you are not going to be pleased when the whole thing fails at this stage because of one missing value or corrupted number!

An example of the kind of function we need to safely convert from strings to float types is shown below. Again, you don't have to worry about the details. What you do have to think about is that real-world computational statistics is a lot more than just pressing buttons with a computer package to analyse perfect data with no effort: you have to think about your data and and you have to think about how to deal with problems from basics.

```
%auto
def safeFloat(obj):
    '''make a float out of a given object if possible.

    param obj is the object to try to turn into a float
    returns the value of the object as a float or nan.

    the float(...) function only works with strings containing
    characters that can be part of a float, such as '123.45',
    so we have to check that the string we pass in can be made
    into a float, otherwise we will get an error when we try make
    a float out of a alpha-character string such as 'AK' or ''.

    We use exception handling to do this '''

    try:
        retval = float(obj)
    except (ValueError, TypeError), diag:
        retval = float('nan') # turn any non-number into nan (not a number)
```

This function gets magnitude data from the array, using a list comprehension to skip not-a-number (`nan`) values. `nan`'s typically arise when the original data was just missing, ie that line of the file did not have a value for magnitude. Note that it assumes a knowledge of the structure of the data (which column the magnitudes are in). This is not very flexible or robust and could have been done much better - can you think how?

jsMath

```
%auto
def makeMagList(myArray):
    '''Return a list of magnitudes from an array of earthquake data.

       Param myArray is a pylab.array of data.
       Return a list of the magnitudes from the array, omitting the nan
values
       Magnitudes are assumed to be in the 12 column of the file '''

    mags = myArray[:,11] # take the magnitude column
    retValue = [] # default return value
    try:
       for x in mags:
           if not math.isnan(x):  # math.isnan returns true if x is a nan
               retValue.append(x)
    except (ValueError, TypeError), diag:
        #print 'Ooops, trouble here'
        retValue = []
    return retValue
```

Here is a more complicated function to get the latitude and longitude coordinates for each earthquake so that we can plot them on a scatter plot. You very definitely don't have to worry about what the following function is doing, although we have provided more comments than would be usual in normal programming. Note again that the function assumes a knowledge of the data structure (which columns the latitudes and longitudes are in). Note also that we want to test the data to make sure that we have both latitude and longitude. This function could be implemented much more efficiently but it does it's job ...

jsMath

```
%auto
def makeCoordList(myArray, minLat, maxLat, minLng, maxLng, minMag):
    '''Return a list of lists of earthquake latitude and longitude data and
magnitudes.

    Param myArray is a pylab.array of the data.
    Param minLat, maxLat are the minimum and maximum latitudes to include in
the results.
    Param minLng, maxLng are the minimum and maximum longitudes to include
in the results.
    Parm minMag is the minimum magnitude we want to include.
    Latitudes are expected to be in the second column of the array
    Longitudes are expected to be in the third column of the array
    Magnitudes are expected to be in the 12th column of the array
    makeCoordList returns a list of three list:
        [0] is the first list in the list, the list of latitudes
        [1] is the second list in the list, the list of longitudes
        [2] is the third list in the list, the list of magnitudes
    ([0][i],[1][i]) would form the tuple of coordinates for the (i+1) valid
row in myArray
    All the component lists are of the same length:
        if a row in myArray is missing either latitude or longitude it is
excluded entirely
        if a row in myArray has latitude and longitude but no magnitude, [2]
will have Nan in that position.
    The lists are sorted so that magnitudes go from lowest to highest
        (latitude and longitude lists are sorted so the latitude and
longitude
        corresponding to magnitude.
    '''

    try: # use try ... except ... to make sure we can do what we want to do
        latStrs = myArray[:,1] # take the latitude column
        lngStrs = myArray[:,2] # take the longitude column
        magStrs = myArray[:,11] # take the mags column
        latlist = [] # start with some empty lists
        lnglist = []
        maglist = []
        n = len(latStrs) # count how many rows we have
        indexes = range(n) # make a list of indexes into the lists
        for i in indexes: # for loop over indexes
            lat = latStrs[i] # index into each list of strings
            lng = lngStrs[i]
            mag = magStrs[i]
            # check we have valid values for latitude and longitude
            if ((not math.isnan(lat)) & (not math.isnan(lng))):
                # add to the list if the values are in within our min and
max ranges
                if (lat >= minLat) & (lat <= maxLat) & (lng >= minLng) &
(lng <= maxLng) & (mag >= minMag):
                    latlist.append(lat) # append a lat at the end of the list
                    lnglist.append(lng) # append a lng at the end of the list
                    maglist.append(mag)

            # else: # omit and report on any lines with nan latitude or
longitude
                #print 'Ignored row' , (i+1), ' error diagnosis lat ', lat,
'long ', lng
        coordtuples = zip(maglist,latlist,lnglist) # temporary tuples
        coordtuples.sort() # which we can sort (lowest to highest)

        # make a list of three lists using list comprehensions
```

jsMath

```
%hide
```

```
%hide
```

```
%hide
```

```
%hide
```

```
%hide
```

jsMath