The Sage Notebook
Version 4.6.2

raazesh.sainudiin Toggle | Home | Published | Log | Settings | Help | Report a Problem | Sign out

STAT221Week02

last edited on April 04, 2011 02:29 PM by raazesh.sainudiin

File... Action... Data... sage Typeset

Print Worksheet Edit Text Undo Share Publish Save Save & quit Discard & quit

Map, Function, Collection, and Probability

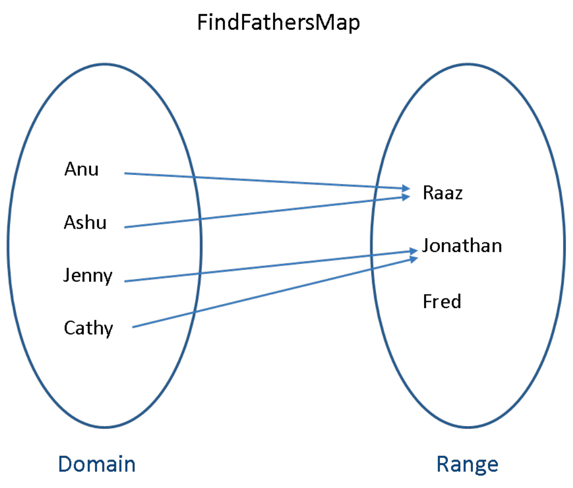
Monte Carlo Methods

Maps and Functions

In the last worksheet we talked about sets. A map or function is a specific kind of **relation** between two sets. The two sets are traditionally called the **domain** and the **range**.

The map or function associates **each element in the domain** with **exactly one** element in the **range**. So, more than one distinct element in the domain can be associated with the same element in the range, and not every element in the range needs to be mapped (i.e, not everything in the range needs to have something in the domain associated with it).

Here is a map for some family relationships: Raaz has two daughters named Anu and Ashu, Jenny has a sister called Cathy and the father of Jenny and Cathy is Jonathan. We can map these daughters to fathers with the FindFathersMap below: The daughters are in the domain and the fathers are in the range. Each daughter maps to a father. More than one daughter can map to the same father.



The notation for this mapping would be **FindFathersMap: Daughters $\rightarrow$ Fathers**

The domain is the set Daughters = {Anu, Ashu, Jenny, Cathy} and the range is the set Fathers = {Raaz, Jonathan, Fred}

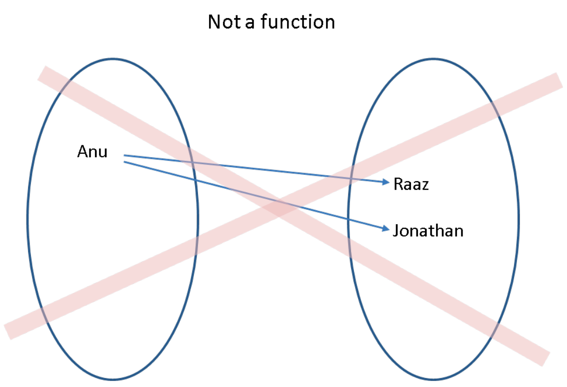
jsMath

The element in the range that an element in the domain maps to is called the **image** of that domain element. For example, Raaz is the image of Anu in the Find Fathers Map. The notation for this is **FindFathersMap(Anu) = Raaz**. Note that what we have just written is a **function**, just like the more familiar format of $f(x) = y$

In computer science lingo each element in the domain is called a **key** and the images in the range are called **values**. The map or function associates each **key** with a **value**.

The keys for the map are unique since the domain is a set, i.e., a collection of distinct elements. Lots of keys can map to the same image value (Jenny and Cathy have the same father, Anu and Ashu have the same father), but the idea is that we can uniquely identify the value we want if we know the key. This means that we can't have multiple identical keys. This makes sense when you look at how the map works. We use the map to find values from the key, as we did when we found Anu's father above. If we had more than one 'Anu' in the keys, we could not uniquely identify the value that maps to the key 'Anu'.

We do not allow maps (functions) where one element in the domain maps to more than one element in the range. In computer science lingo, each key can have only one value associated with it.



Formalising this, a **function** $f : \mathbf{X} \rightarrow \mathbf{Y}$ is equivalent to the set $\{(x, f(x)) : x \in \mathbf{X}, f(x) \in \mathbf{Y}\}$.

$(x, f(x))$ is an **ordered pair**.

The **pre-image** or inverse image of a function $f : \mathbf{X} \rightarrow \mathbf{Y}$ is $f^{[-1]}$.

The inverse image takes subsets in $\mathbf{Y}$ and returns subsets of $\mathbf{X}$.

The pre-image or inverse image of y is $f^{[-1]}(y) = \{x \in \mathbf{X} : f(x) = y\} \subset \mathbf{X}$.

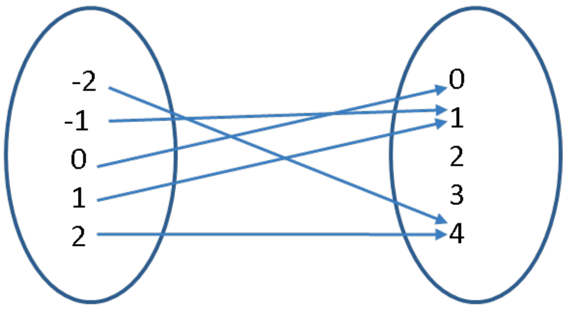
For example, for our FindFathersMap, FindFathersMap$^{[-1]}$(Raaz) = {Anu, Ashu} and FindFathersMap$^{[-1]}$(Jonathan) = {Jenny, Cathy}.

Now lets take a more 'mathy' looking function $f(x) = x^2$.

Version 1 of this function is going to have a *finite domain* (only five elements).

The domain is the set $\{-2, -1, 0, 1, 2\}$ and the range is the set $\{0, 1, 2, 3, 4\}$.

The mapping for version 1 is $f(x) = x^2 \ : \ \{-2, -1, 0, 1, 2\} \rightarrow \{0, 1, 2, 3, 4\}$



jsMath

We can also represent this mapping as the set of ordered pairs $\{(-2,4),(-1,1),(0,0),(1,1),(2,4)\}$

Note that the values 2 and 3 in the range have no pre-image in the domain. This is okay because not every element in the range needs to be mapped.

Having a domain with only five elements in it is a bit restrictive: how about an *infinite domain*?

Version 2 $f(x)=x^2 : \{\ldots,-2,-1,0,1,2,\ldots\} \to \{0,1,2,3,4,\ldots\}$

As ordered pairs, we now have $\{\ldots,(-2,4),(-1,1),(0,0),(1,1),(2,4),\ldots\}$, but it is impossible to write them all down since there are infinitely many elements in the domain.

What if we wanted to use the function for the whole of $\mathbb{R}$, the *real line*?

Version 3 $f(x)=x^2 : \mathbb{R} \to \mathbb{R}$

Example 1: Maps and Functions are encoded as (i) sets of ordered pairs or as (ii) procedures.

In Sage we can encode a map as **a set of ordered pairs** as follows.

```
findFathersMap = {'Jenny': 'Jonathan', 'Cathy': 'Jonathan', 'Anu': 'Raaz',
'Ashu': 'Raaz'}
findFathersMap
```

Given a key (child) we can use the map to find the father.

```
print "The father of Anu is", findFathersMap['Anu']
```

We could also encode our **inverse map** of the findFathersMap as findDaughtersMap. We map from one father key (e.g. Raaz) to multiple values for daughters (for Raaz, Anu and Ashu).

```
findDaughtersMap = {'Jonathan': ['Jenny', 'Cathy'], 'Raaz': ['Anu', 'Ashu']}

# don't worry about the way we have done the printing below, just check the
output makes sense to you!
for fatherKey in findDaughtersMap:
    print fatherKey, "has daughters"
    for child in findDaughtersMap[fatherKey]:
```

We can also use a map for a simple function y = x^2 + 2: {-2 ,-1 ,0, 1 ,2} → {2, 3, 4, 5, 6} by encoding this function as **a set of ordered pairs** {(-2,6), (-1,3), (0,2), (1,3), (2,6)} in SAGE as follows.

```
myFunctionMap = {-2: 6, -1: 3,  0: 2, 1: 3, 2: 6}
myFunctionMap
```

Looking up the value or image of the keys in our domain {-2,-1,0,1,2} works. Note the KeyError message when the key is outside the domain.

```
myFunctionMap[-2]
```

```
myFunctionMap[-20] # KeyError: -20 since -20 is not in {-2,-1,0,1,2}
```

jsMath

But it is clearly not a good way to try to specify a function mapping that can deal with lots of different x's: in fact it would be impossible to try to specify a mapping like this if we want the *domain* of the function to have infinitely many elements (for example, if the domain is the set of all integers, or all rational numbers, or a segment of the real line, for example $f(x) = x^2 + 2 : \mathbb{R} \rightarrow \mathbb{R}$.

Instead, in SAGE we can just define our own function as a **procedure** that can evaluate our image value 'on-the-fly' for any x we want. We'll be doing more on functions in later labs so for the moment just think about how defining the function can be seen as a much more flexible way of specifying a mapping from the *domain* of the function to the *image*.

Some basics about **encoding a function as a procedure**:

1. The function named `myFunc` we want to define is preceded by the keyword `def` for definition.
2. The function name `myFunc` is succeeded by any input argument(s) to it within a pair of parentheses, for e.g., `(x)` in this case since there is only one argument, namely `x`.
3. After we have defined the function by its name `myFunc` followed by its input argument `(x)` we end the line with a colon `:` before continuing to the next line.
4. Now, we are ready to write the body of the function. It is a customary to leave 4 white spaces. The number of spaces before a line or the indentation is used to delineate a block of code in SAGE.
5. It is a matter of courteous programming practice to enclose the Docstring, i.e., comments on what the function does, inside triple quotes. The Docstring is returned when we ask SAGE for help on the function.
6. Finally, we output the image of our function with the keyword `return` and the expression `x^2+2`.

```
def myFunc(x):                                 # starting the definition
    '''A function to return x^2 + 2'''         # the docstring
    return x^2+2                               # the function body (only one
line in this example
```

```
myFunc?
```

```
myFunc(0.1)   # use the function to calculate 0.1^2 + 2
```

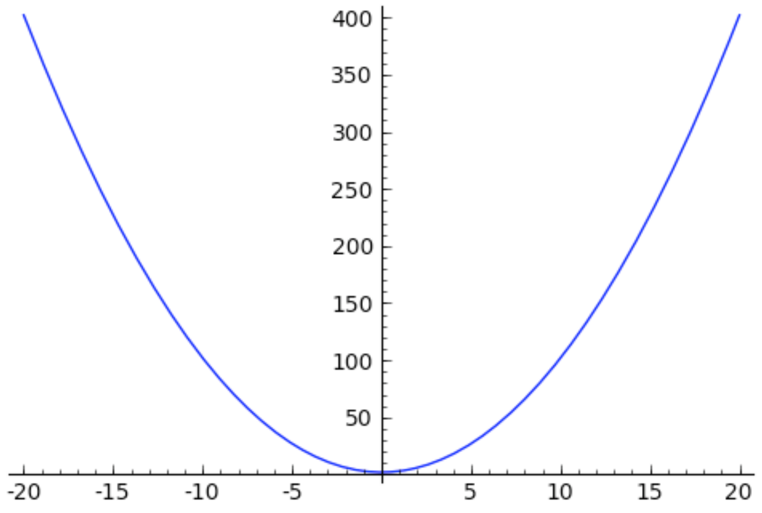
When you evaluate the cell below, SAGE will complain about an indentation error that you can easily fix.

```
def myFunc(x):
    '''A function to return x^2 + 2'''
     return x^2+2
```

The command to plot is pretty simple. The four arguments to plot are the function to be plotted, the input argument that is varying along the x-axis, the lower-bound and upper-bound of the input argument.

```
plot(myFunc(x),x, -20, 20)
```

jsMath



The simple plot command hides what is going on under the hood. Before we understand the fundamentals of plotting, let us get a better appreciation for the ordered pairs $(x, f(x))$ that make up the curve in this plot.

We can use SAGE to plot functions and add a way for you to interact with the plot. When you have evaluated this cell you'll see a plot of our function between $x = -20$ and $x = 20$. The point on the curve where $x = 3$ is indicated in red. You can alter the position of this point by putting a new value into the box at the top of the display (you can put in real number, eg 4.45, but if your number is outside the range -20 to 20 it won't be shown. Just try changing the value of x to plot as a point on the curve and don't worry about the way the code looks - you aren't expected to do this yourselves!.

```
@interact
def _(my_x=3):
    myPt = (my_x, myFunc(my_x))
    myLabel = "(%.1f, %.1f)" % myPt
    p = plot(myFunc, (x,-20,20))
    if (my_x >= -20 and my_x <= 20):
        p += point(myPt,rgbcolor='red', pointsize=20)
        p += text(myLabel, (my_x+4,myFunc(my_x)), rgbcolor='red')
    p.show()
```

You try

Define a more complicated function with four input arguments next. The rules for defining such a function are as before with the additional caveat of declaring all four input arguments inside the pair of parenthesis following the name of the function. In the cell below you will have to **uncomment one line** and then evaluate the cell to define the function (remember that comments begin with the # character).

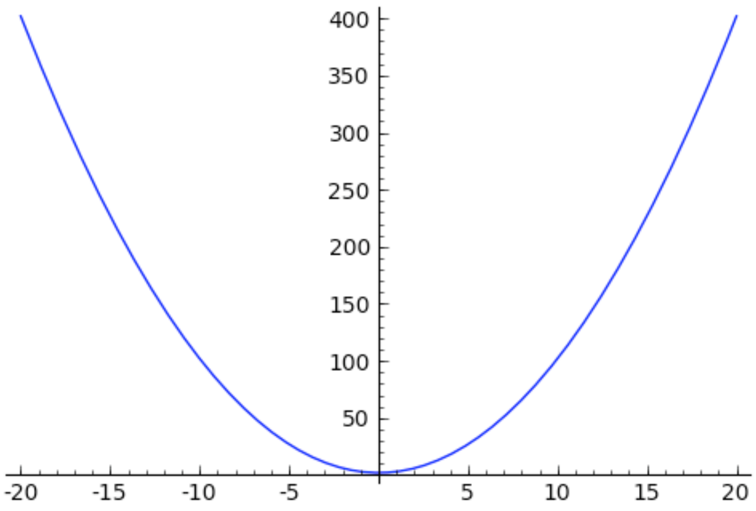
```
# Here is a quadratic function of x with three additional parameters a,b,c
def myQuadFunc(a,b,c,x):
    '''A function to return a*x^2 + b*x + c'''
    #return a*x^2 + b*x + c
```

Now try writing an expression to find out what `myQuadFunc` is for some values of x and coefficients a=1, b=0, c=2. We have put the expression that uses these coefficients and x=10 into the cell below for you. Can you see how Sage interprets the expression using the order in which we specified a, b, c, x in the definition above? Try changing the expression to evaluate the function with same coefficients (a, b, c) but different values of x.

```
myQuadFunc(1, 0, 2, 10) # a = 1, b = 0, c = 2, x = 10
```

jsMath

```
# we can make the same plot as before by letting a=1, b=0, c=2
plot(myQuadFunc(1,0,2,x),x, -20, 20)
```



You could try copying and pasting and then changing the values for the coefficients to make your own plot.

Example 2: Mathematical functions

Many familiar mathematical functions such as sin, cos, log, exp are also available directly in Sage as 'built-in' functions. They can be evaluated using parenthesis (brackets).

```
print 'sin(pi) prints as', sin(pi)
```

```
print 'cos(1/2) prints as', cos(1/2)
print 'cos(1/2).n(digits=5) prints as', cos(1/2).n(digits=5)
print 'float(cos(1/2)) prints as', float(cos(1/2))
print 'cos(0.5) prints as', cos(0.5)

print 'exp(2*pi*e) prints as', exp(2*pi*e)

print 'log(10) prints as', log(10)
print 'log(10).n(digits=5) prints as', log(10).n(digits=5)
```

You try

You can find out what built-in functions are available for a particular variable by typing the variable name followed by a '.' and then pressing the TAB key.

jsMath

```
x=-2
```

```
# type x. and press TAB
```

Try the abs function that evaluates the absolute value of x.

```
abs(x)
```

If you want to know what a built-in function does, type the function name followed by a '?'.

```
abs?
```

Collections in Sage

We have already talked about the Sage number types and a little about the string type. You have also met sets in Sage. A set in Sage is an example of a *collection* type. Collections are a useful idea: grouping or *collecting together* some data (or variables) so that we can refer to it and use it *collectively*.

Sage provides quite a few collections. One that we will meet very often is the **list**.

Example 1: Lists

Technically, a list in SAGE is a sequence type. This basically means that the order of the things in the list is imporant and we can use concepts related to ordering when we work with a list. Don't worry about that unless you are interested in the details, but look at the worksheet cells below to see how useful and flexible lists are.

When you want a list in SAGE you put the things in the list inside [ ] called square brackets. You can put almost anything in a list (including having lists of lists, as we'll see later).

```
L1 = []      # an empty list
L1           # display L1
```

```
L2 = ['orange', 'apple', 'lemon']   # a list of strings - remember that
strings are in quote marks
L2                                  # display L2
```

```
L2.append('banana')                 # append something to the end of a list
L2                                  # display L2
```

```
L3 = [10, 11, 12 ,13]               # a list of integers
```

jsMath

```
type(L3)                             # type of L3 is
```

There are various functions we can use with lists.   A very useful one is *len*, which gives us the length of the list, i.e., the number of elements in the list.

```
len(L3)
```

What about getting at the elements in a list once we have put them in?   This is done by *indexing* into the list, or *list indexing* .   Slightly confusingly, once you have a list, you *index* into it by using the [ ] brackets again.

```
L3[0]                                # the first position in the list is
```

Note that in SAGE the first position in the list is at position 0, or index [0] *not at index [1]*.   In the list L3, which has 4 elements (`len(L3) = 4`), the indices are [0], [1], [2], [3].   In the cell below you can check what you get when you ask for the element at the fourth position in the list (i.e., index [3] since the indexes start from [0]).

```
L3[3]
```

You will get an error message if the index you use is *out of range* (which means that you are trying to refer to something outside the range of the list).   SAGE knows that the list only has 4 elements, so asking for the element in the fifth position (index [4]) makes no sense.

```
L3[4]
```

We can also get at more than one element in the list with the indexing operator [ ], by using ':' to indicate all elements *from* a position *to* another position.   This is hard to explain in words but easy to see in action.

```
L3[0:2]                         # elements in positions 0 to 2 in list L3 are
```

If you leave out the starting and ending positions and just use [:] you'll get the whole list.   This is useful for making copies of whole lists.

```
L4 = L3[:]
L4            # disclose L4
```

SAGE also provides some helpful ways to make lists quickly.   The one you'll use most often is **range**.   Used in its most simple form, `range(n)` gives you a list of n integers from 0 to n-1.

```
L5 = range(10)                       # a quick way to make a list of 10
numbers starting from 0
L5
```

Note that the numbers you get start from 0 and the last one is 9.

You'll see that we can get even cleverer and use *range* to get a list that starts and stops at specified numbers with

jsMath

specified 'step' size between the numbers.   Let's try this to get numbers in steps of 5 from 100 to 195, in the cell below.

```
L6 = range(100, 200, 5)   # get a list with a specified start, stop and step
L6
```

Notice that again we don't go right up to the 'stop' number (200) but to the last one below it taking into account our step size (5), ie 195.

When we just asked for `range(10)` SAGE assumed a *default* start of 0 and a *default* step of 1. `range(10)` is equivalent to `range(0, 10, 1).`

You try

Find out more about list and range by evaluating the cells below and looking at the documentation.

```
help(list)
```

docs-0.html

```
help(range)
```

docs-0.html

Make yourself a list with some elements in - you can choose how many elements to have and what type they are. Assign the list to a variable named **myList**.

Add a new element to **myList**.

Use the nice way of copying we showed you above (remember, [:]) to copy everything in myList to a new list called **myNewList**. Use the len function to check the length of your new list.

Use the indexing operator [ ] to find out what the first element in **myList** is (remember that the index of the first element will be 0).

Use the indexing opertor [ ] to change the first element in **myNewList** to some different value.

jsMath

Disclose the original list, **myList**, to check that nothing in that has changed.

Use range to make a list of the integer numbers between 4 and 16, going up in steps of 4. Assign this list to a variable named **rangeList**.

Disclose your list **rangeList** to check the contents. You should have the values 4, 8, 12, 16. If you don't, check what you asked for in the cell above and fix it up to give the values that you wanted.

Example 2: Sets

We also declared sets and operated them earlier. Sets are another Sage collection. Remember that in a set, each element has to be unique. We can specify a set directly or make one out of a list.

```
L6 = range(100, 200, 5)  # make sure we have a list L6
S = set(L6)     # make the set S from the list L6
S              # display the set s
```

Sets are *unordered collections*. This makes sense when we think about what we know about sets: what matters about a set is the unique elements in it.

The set {1, 2, 3} is the same as the set {1, 3, 2} is the same as the set {2, 3, 1} etc, etc.

This means that it makes no sense to ask Sage what's at some particular position in a set as we could with lists. Lists are sequnces and order matters. Sets are unordered - order makes no sense for a set.

We cannot use the indexing operator [ ] with a set.

```
S[0] # will give an error message since sets are unordered collections
```

Example 3: Dictionaries

When we created our maps at the start of this worksheet, we used **dictionaries**: A Sage *dictionary* gives you a way of mapping from a *key* to a *value*. As we said earlier, the keys have to be unique (only one of each key) but more than one key can map to the same value. Remember the father's map? Now we know that is a dictionary or simply `dict`.

```
findFathersMap = {'Jenny': 'Jonathan', 'Cathy': 'Jonathan', 'Anu': 'Raaz',
'Ashu': 'Raaz'}
findFathersMap
```

```
    {'Cathy': 'Jonathan', 'Anu': 'Raaz', 'Jenny': 'Jonathan', 'Ashu':
    'Raaz'}
```

```
type(findFathersMap)
```

jsMath

```
    <type 'dict'>
```

When we make a dictionary, we tell Sage that is is a dictionary by using the curly brackets **{ }** and by giving each key value pair in the format *key: value*.

A dictionary has something like the indexing operator we used for lists, but instead of specifying the position we want (like [0]) we specify the *key* we want, and Sage returns the value that key maps to.

```
findFathersMap['Anu']            # who is Anu's father
```

```
    'Raaz'
```

You try

In the cell below we have the start of a simple dictionary for phone numbers.

```
myPhoneDict = {'Ben': 8888, 'Raaz': 3333}
myPhoneDict     # disclose the contents of the dictionary
```

```
    {'Raaz': 3333, 'Ben': 8888}
```

You try

In the cell below let us add 'susy' with phone number 78987 to our dictionary.

```
myPhoneDict['susy']=78987
```

```
myPhoneDict
```

```
    {'susy': 78987, 'Raaz': 3333, 'Ben': 8888}
```

Try adding to what we have to put in two more people, Fred, whose phone number is 1234, and Mary whose phone number is 7777.   Remember that for Sage, the names Fred and Mary are strings and you must put them in quote marks, like the names that are already there.

Now try asking SAGE for Ben's phone number (ie, the phone number *value* associated with the *key* 'Ben').

Probability

The origins of probability can be traced back to the 17th century. It arose out of the study of gambling and games of chance. Many well-known names associated with probability worked on problems to do with gambling: people like Bernoulli and Pascal did quite a lot of work in this area ... even Newton was persuaded to set down some of his thoughts about a game involving dice (in a letter to a Samuel Pepys).

Probability has a language of its own. We are going to introduce you to some of the essential terms:

An **experiment** is an activity or procedure that produces distinct or well-defined **outcomes**. The set of such outcomes is called the **sample space** of the experiment. The sample space is usually denoted with the symbol Ω. Lets look at some examples of experiments.

Roll a dice experiment

jsMath

If our experiment is to roll a dice wth faces painted red, green, yellow, pink, blue and black and find what colour the top face is at the end of each roll, then the sample space $\Omega =$ {red, green, yellow, pink, blue, black}.

Flip a coin experiment

If our experiment is to flip a coin where the two faces of the coin can be identified as 'heads' (H) and 'tails' (T), and we are interested in what face lands uppermost, then the sample space is $\Omega = \{H, T\}$.

Draw a fruit from a fruit bowl

Suppose we have a well-mixed fruit bowl that contains:

- 2 oranges
- 3 apples
- 1 lemon

If our experiment is to take a single fruit from the bowl and the outcome is the type of fruit we take then what is the sample space for this experiment?

Recall that the sample space is the set of all possible outcomes of the experiment. If we take a single fruit we could get only one of the three fruits in each draw: an orange, *or* an apple *or* a lemon. The sample space $\Omega = \{orange, apple, lemon\}$.

An **event** is a subset of the sample space. For example, we could take the event {orange, lemon} $\subset \Omega$ in the fruit bowl experiment.

Probability maps a set of events to a set of numbers. Abstractly, probability is a function that assigns numbers in the range 0 to 1 to events

$$P : \text{set of events } \rightarrow [0, 1]$$

which satisfies the following **axioms**:

1. For any event A, $0 \leq P(A) \leq 1$.
2. If Ω is the sample space, $P(\Omega) = 1$.
3. If A and B are disjoint (i.e., $A \cap B = \emptyset$), then $P(A \cup B) = P(A) + P(B)$.
4. If $A_1, A_2, \ldots$ is an infinite sequence of pair-wise disjoint events (i.e., $A_i \cap A_j = \emptyset$ when $i = j$), then

$$\begin{aligned} \underbrace{P\left(\bigcup_{i=1}^{\infty} A_i\right)}_{A_1 \cup A_2 \cup A_3 \ldots} &= \underbrace{\sum_{i=1}^{\infty} P\left(A_i\right)}_{P(A_1) + P(A_2) + P(A_3) + \ldots} \end{aligned}$$

These axioms or assumptions are motivated by the **frequency interpretation of probability**. The frequency interpretation of probability says that if we repeat an experiment a very large number of times then the fraction of times that the event A occurs will be close to $P(A)$.

More precisely,

$$\begin{array}{rcl} \text{let} \quad N(A,n) &=& \text{the number of times } A \text{ occurs in the first } n \text{ trials,} \\ \text{then} \quad P(A) &=& \lim_{n\rightarrow\infty} \frac{N(A,n)}{n} \end{array}$$

To think about this, consider what $\lim_{n\rightarrow\infty} \frac{N(A,n)}{n}$ is:

$$\frac{N(A,1)}{1}, \frac{N(A,2)}{2}, \frac{N(A,3)}{3}, \ldots \text{where is this fraction going?}$$

Let's look at axioms 1, 2, and 3 above more closely.

1. For any event A, $0 \leq P(A) \leq 1$. Well, clearly $0 \leq \frac{N(A,n)}{n} \leq 1$.
2. If Ω is the sample space, $P(\Omega) = 1$. This essentially says "something must happen". $P(\Omega) = \frac{N(\Omega,n)}{n} = \frac{n}{n} = 1$.

3. If A and B are disjoint (i.e., $A \cap B = \emptyset$), then $N(A \cup B, n) = N(A,n) + N(B,n)$ since $A \cup B$ occurs if either A or B occurs but we know that it is impossible for both A and B to happen (the intersection is the empty set). This extends to infinitely many disjoint events.

Axiom 4 is a bit more controversial, but here we assume it as part of our axiomatic definitiion of probability (without it the maths is much harder!).

Lets do some probability examples.

Example 1: Tossing a fair coin

$\Omega = \{H, T\}$, $P(H) = P(T) = \frac{1}{2}$

Check that all our axioms are satisfied:

1. $0 \leq P(H) = P(T) = \frac{1}{2} \leq 1$ and $0 \leq P(\Omega) = 1 \leq 1$.
2. $P(\Omega) = P(\{H, T\} = P(\{H\}) + P(\{T\}) = \frac{1}{2} + \frac{1}{2} = 1$.
3. $P(\{H, T\} = P(\{H\}) + P(\{T\})$.
4. is okay too!

Example 2: New Zealand Lotto

In New Zealand Lotto, the balls drawn are numbered 1 to 40. The number on a ball is an outcome.

$\Omega = \{1, 2, \ldots, 40\}$, $P(\omega) = \frac{1}{40}$ for each $\omega \in \Omega$ (i.e., $P(1) = P(2) = \ldots = P(40) = \frac{1}{40}$)

Now, consider the event that the first ball is even? What is the probability of this event, $P(\{2, 4, 6, \ldots, 38, 40\})$?

$$\begin{array}{rcll} P(\{2,4,\ldots,38,40\}) & = & P\left(\{2\} \cup \{4\} \cup \cdots \cup \{38\} \cup \{40\}\right) & \text{(defn. of set union)} \\ & = & P(\{2\}) + P(\{4\}) + \cdots + P(\{38\}) + P(\{40\}) & \text{(extn. Axiom 3)} \\ & = & \sum_{i \in \{2,4,\ldots,40\}} P(\{i\}) & \\ & = & 20 \times \frac{1}{40} & \\ & = & \frac{1}{2} & \end{array}$$

Similarly for the probability of an odd ball:

$$\begin{array}{rcl} P(\{1,3,5,\ldots,37,39\}) & = & P\left(\{1\} \cup \{3\} \cup \cdots \cup \{37\} \cup \{39\}\right) \\ & = & P(\{1\}) + P(\{3\}) + \cdots + P(\{37\}) + P(\{39\}) \\ & = & \sum_{i \in \{1,3,\ldots,37,39\}} P(\{i\}) \\ & = & 20 \times \frac{1}{40} \\ & = & \frac{1}{2} \end{array}$$

Example 3: The Waimakariri Urn

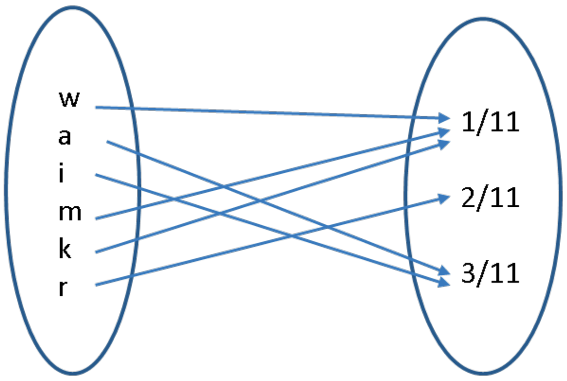
The waimakariri urn of letters is an urn with all the letters from "waimakariri" in it.

What is the sample space Ω and what probabilities should be assigned to the outcomes ω?

$\Omega = \{\omega_1 = \mathrm{w},\ \omega_2 = \mathrm{a},\ \omega_3 = \mathrm{i},\ \omega_4 = \mathrm{m}, \omega_5 = \mathrm{k}, \omega_6 = \mathrm{r}\}$

$P(\{\mathrm{w}\}) = \frac{1}{11}$, $P(\{\mathrm{a}\}) = \frac{3}{11}$, $P(\{\mathrm{i}\}) = \frac{3}{11}$, $P(\{\mathrm{m}\}) = \frac{1}{11}$, $P(\{\mathrm{k}\}) = \frac{1}{11}$, $P(\{\mathrm{r}\}) = \frac{2}{11}$

jsMath



Consider the event that the letter we draw is one of 'w', 'a', or 'r', i.e. the event $\{\omega_1 = \text{w}, \omega_2 = \text{a}, \ \omega_6 = \text{r}\}$

$$\begin{aligned} P(\{\text{w},\text{a},\text{r}\}) &= \underbrace{P(\{\text{w}\} \cup \{\text{a}\} \cup \{\text{r}\})}_{\text{these are pairwise disjoint events}} && \text{(definition of set unions)} \\ &= P(\{\text{w}\}) + P(\{\text{a}\}) + P(\{\text{r}\}) && \text{(by axiom 3 extended to 3 events)} \\ &= \tfrac{1}{11} + \tfrac{3}{11} + \tfrac{2}{11} \\ &= \tfrac{6}{11} \end{aligned}$$

Aside: The set of all possible sets of Ω is called the **power set** and is denoted by 2^{Ω}. The power set is a σ**-algebra** or σ**-field**. More on this later!

Now, having introduced a number of definitions, we will derive some basic logical consequences, (i.e., **properties**) of probabilities (axiomatically defined).

Property 1

$P(A) = 1 - P(A^c)$, where $A^c = \Omega \setminus A$

Proof

$A \cap A^c = \emptyset$ and $A \cup A^c = \Omega$

Recall that axiom 3 says that if $A_1 \cap A_2 = \emptyset$ then $P(A_1 \cup A_2) = P(A_1) + P(A_2)$

So this implies that $P(A) + P(A^c) = P(\Omega) = 1$, by axiom 2 which says that $P(\Omega) = 1$

Subtracting $P(A^c)$ from both sides, we get $P(A) + P(A^c) - P(A^c) = 1 - P(A^c)$

Cancelling out the two $P(A^c)$ terms on the right hand side, we get $P(A) = 1 - P(A^c)$ QED

Example 4

In the coin tossing experiment, $\Omega = \{H, T\}$

$P(H) = 1 - P(H^c) = 1 - P(\Omega \setminus H) = 1 - P(T)$

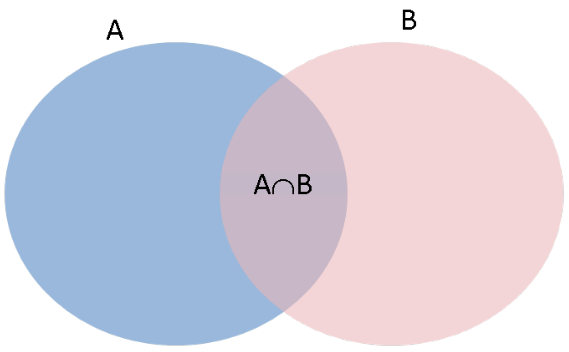
Property 2

For any two events A, B,

$P(A \cup B) = P(A) + P(B) - P(A \cap B)$

Proof

jsMath



This is an informal proof using the picture above. If we just add the probabilities of A and B we will double count the probabilties of the outcomes which are in both A and B. We adjust for this double counting by subtracting $P(A \cap B)$.

Note that if $A \cap B = \emptyset$ then $P(A \cap B) = 0$ and $P(A \cup B) = P(A) + P(B)$.

Example 5: Experiments, outcomes, sample spaces, events, and the probability of events

Let's go back to the well-mixed fruit bowl experiment. The fruit bowl contains:

- 2 oranges
- 3 apples
- 1 lemon

The experiment is to take one piece of fruit from the bowl and the outcome is the type of fruit we get.

The sample space is $\Omega = \{orange, apple, lemon\}$

We can use the Sage **list** to create this sample space (a list is a bit easier to use than a set, but using a list means that we are responsible for making sure that each element contained in it is unique).

```
# sample space is the set of distinct type of fruits in the bowl
samplespace = ['orange', 'apple', 'lemon']
```

We can also use a list to specify what the probability of each outcome in the sample space is. The probabilities can be calculated by knowing how many fruits of each kind are there in the bowl. We say that the fruit bowl is 'well-stirred', which essentially means that when we pick a fruit it really is a 'random sample' from the bowl (for example, we have not carefully put all the apples on the top so that someone will almost certainly get an apple when they take a fruit). More on this later in the course! Note that the probabilities encoded by a list named `probabilities` are in the same order as the outcomes in the `samplespace` list.

```
# probabilities take into account the number of each type of fruit in the
"well-stirred" fruit bowl
probabilities = [2/6, 3/6, 1/6]
```

```
probMapFruitbowl = ProbyMap(sspace = samplespace, probs=probabilities) #
make our probability map
probMapFruitbowl         # disclose our probability map
```

So we have probabilities associated with events -- that sounds like a good use for a dictionary? That's basically what are going to use, but we 'wrap' up the dictionary in some extra code that does things like check that the probabilities add to 1, and that each element in the list of outcomes we have given is unique. If you are interested in programming, we have coded our own type, or *class*, called ProbyMap (it's hidden, and you can ignore all the details completely!). Once the class is coded, we create a ProbyMap by providing the sample space and the probabilities. You don't have to worry

jsMath

about how the **class** is implemented, but note that in computational statistics you may often want to use the computer to create a **computerised representation of a concept**. Once the concept (a discrete probability map in our case), then we can use the computer to automate the mundane tasks of large-scale computations.

We can use our probability map to find the probability of a single outcome like this:

```
# Find the probability of outcome 'lemon'
probMapFruitbowl.P(['lemon'])
```

We can also use our probability map to find the probability of an event (set of outcomes).

```
# Find the probability of the event {lemon, orange}
probMapFruitbowl.P(['lemon','orange'])
```

Basically, the probability map is essentially a map or dictionary.

Next we will obtain the set of all events (the largest σ-algebra or σ-field in math lingo) from the outcomes in our sample space via the `Subset` function and find the probability of each event using our `ProbyMap` in a `for` loop.

```
# make the set of all possible events from the set of outcomes
setOfAllEvents = Subsets(samplespace)  # Subsets(A) returns the set of all
subsets of A
list(setOfAllEvents)    # disclose the set of all events
```

We have not done loops yet, but we will soon. Just as a foretaste, here we use a for loop to print out the computed probabilities for each event in the setOfAllEvents

```
# loop through the set of all events and print the computed probability
for event in setOfAllEvents:
    print "P(", event, ") = ", probMapFruitbowl.P(event)
```

You try

Try working through Example 6 below for yourself in the tutorial

Example 6: Experiments with the English language

In English language text there are 26 letters in the alphabet. Someone, somewhere, for some reason, has calculated the relative frequencies with which each letter appears:

E	13.0%	H	3.5%	W	1.6%
T	9.3%	L	3.5%	V	1.3%
N	7.8%	C	3.0%	B	0.9%
R	7.7%	F	2.8%	X	0.5%
O	7.4%	P	2.7%	K	0.3%
I	7.4%	U	2.7%	Q	0.3%
A	7.3%	M	2.5%	J	0.2%
S	6.3%	Y	1.9%	Z	0.1%
D	4.4%	G	1.6%		

Using these relative frequencies as probabilities we can create a probability map for the letters in the English alphabet.

jsMath

We start by defining the sample space and the probabilities.

```
alphaspace =
['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S',
'T','U','V','W','X','Y','Z']
alphaRelFreqs = [73/1000,9/1000,30/1000,44/1000,130/1000,28/1000,16/1000,35
/1000,74/1000,2/1000,3/1000,35/1000, 25/1000,78/1000,74/1000,27/1000,3
```

Then we create the probability map, represented by a ProbyMap object.

```
probMapLetters = ProbyMap(sspace = alphaspace, probs=alphaRelFreqs) # make
our probability map
probMapLetters         # disclose our probability map
```

Please do not try to list the set of all events of the 26 alphabet set: there are over 67 million events and the computer will probably crash! You can see how large a number we are talking about by evaluating the next cell which calculates 2^26 for you.

```
2^26
```

Instead of asking for the probability of each event (over 67 million of them to exhaustively march through!) we define some events of interest, say the vowels in the alphabet or the set of letters that make up a name.

```
vowels = ['A', 'E', 'I', 'O', 'U']
```

And we can get the probability that a letter drawn from a 'well-stirred' jumble of English letters is a vowel.

```
probMapLetters.P(vowels)
```

We can make ourselves another set of letters and find probabilities for that too. In the cell below, we go straight from a string to a set. The reason that we can do this is that a string is in fact another collection type!

```
NameOfRaaz=set("RAAZESHSAINUDIIN")        # make a set from a string
NameOfRaaz                                # disclose the set NameOfRaaz you
have built
```

```
probMapLetters.P(NameOfRaaz)
```

Try either adapting what we have above, or doing the same thing in some new cells, to find the probabilities of other sets of letters yourself.

jsMath

You try

If you have time, you can look at the further material about plotting below.

Example 7: Plotting a list of points

Remember the function we defined earier called `myFunc`? Let us take a closer look at what is happening under the hood when we called `plot` earlier on `myFunc`.

```
# redefine myFunc, just to to make sure...
def myFunc(x):                                # starting the definition
    '''A function to return x^2 + 2'''       # the docstring
    return x^2+2                              # the function body (only one
line in this example
```

Read the documentation about plot to give you an idea about what is going on.

```
help(plot)
```

docs-0.html

Lets use exactly 10 points to make a plot of `myFunc`.

```
# we are using exactly 10 points to make this plot
myPlot=plot(myFunc,(-2,2), plot_points=10,marker='.',linestyle=' ',
randomize=False, adaptive_recursion=0)
```

What do you get when you evaluate the next cell? Sage can tell you about the object `myPlot` that you just made.

```
myPlot[0]
```

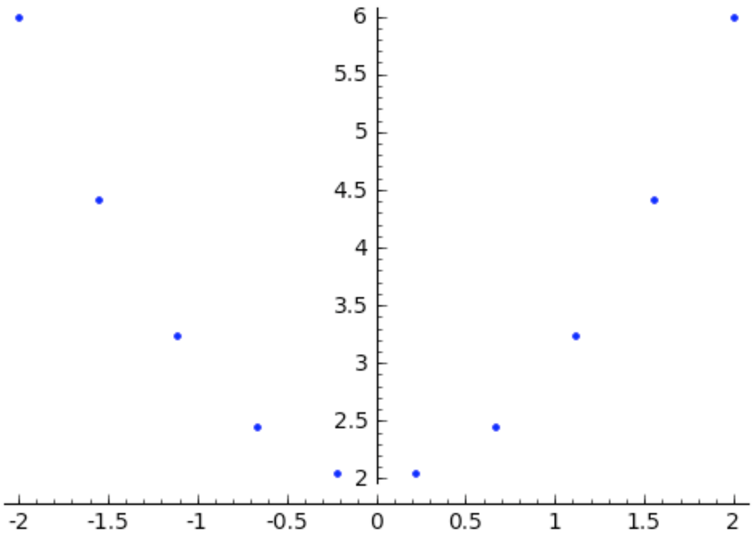
We can also get the actual list of ten 2-dimensional points (ordered pairs) that make up our plot `myPlot`.

```
# L7 is a list of ten 2-dimensional points or ordered pairs that make up our
plot MyPlot
L7=list(myPlot[0])
L7
```

To show the plot we use the **`show`** function. myPlot only has the 10 points to show.

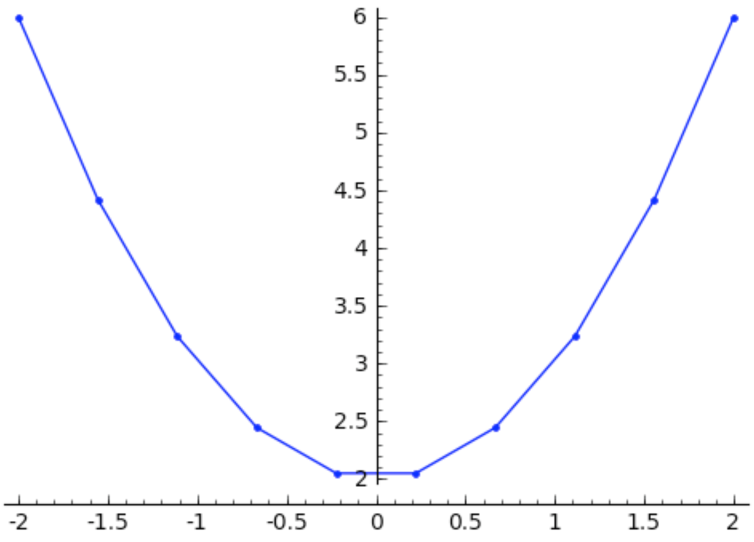
```
# plot only the 10 equally spaced points
show(myPlot)
```

jsMath



But, we could have asked for the line, which is drawn by linearly interpolating between the points. Note the different option we have used here for `linestyle`.

```
# the lines are drawn by linearly interpolating between the consecutive 2D
points
plot(myFunc,(-2,2), plot_points=10,marker='.',linestyle='-',
randomize=False, adaptive_recursion=0)
```



There is a specific function for plotting points, called `point`.

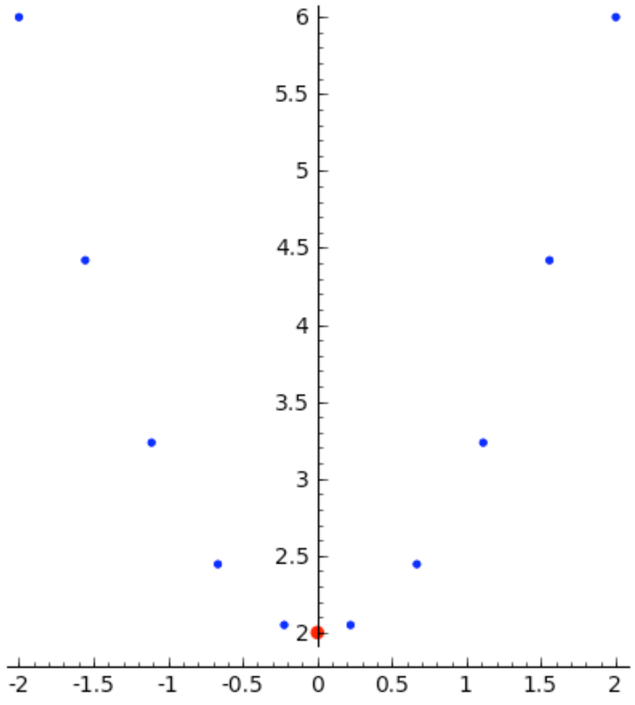
```
# make a Graphics 2D point (0,2) colored red with size 30
g = point((0,2),rgbcolor=(1,0,0), pointsize=30)
type(g)                 # the type of g is 'sage.plot.plot.Graphics'
```

We could have made a similar plot to `myPlot` 'by hand', adding points to a Graphics plot (again, this uses a for loop, which we will cover properly soon):

jsMath

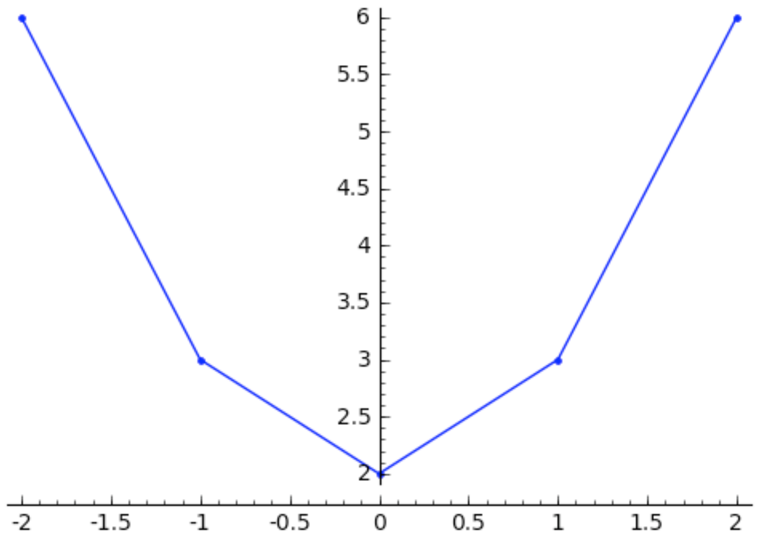
```
for pt in L7:     # with the for loop we go through each pt in our L7 list
    g += point(pt)            # we make a graphic point out of pt and add it
to our plot.Graphics g

# show the figure we have drawn and compare with show(myPlot)
g.show(figsize=[5,5],xmin=-2,xmax=2,ymin=2,ymax=6) # uncomment for more
```



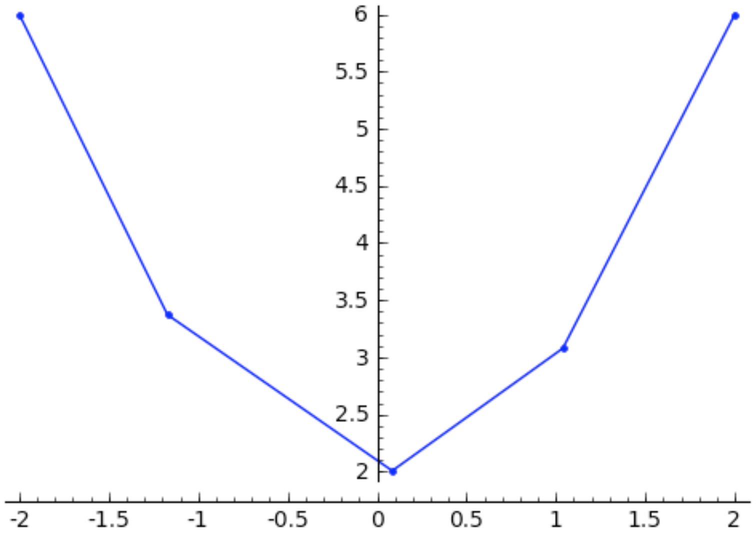
If we only plot 5 points the curve is not nearly as smooth:

```
# plot only the 5 equally spaced points with linearly interpolating line
plot(myFunc,(-2,2), plot_points=5,marker='.',linestyle='-', randomize=False,
adaptive_recursion=0)
```



Use a random selection of 5 points?

```
# plot 5 randomized points
plot(myFunc,(-2,2), plot_points=5,marker='.',linestyle='-', randomize=True,
adaptive_recursion=0)
```

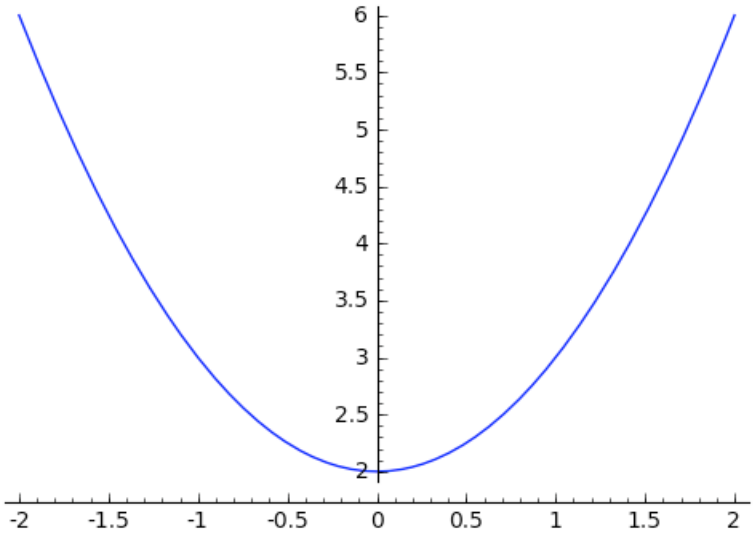


Try 200 equally spaced points?

```
# plot the 200 equally spaced points with interpolating line
plot(myFunc,(-2,2), plot_points=200,marker='.',linestyle='-',
randomize=False, adaptive_recursion=0)
```

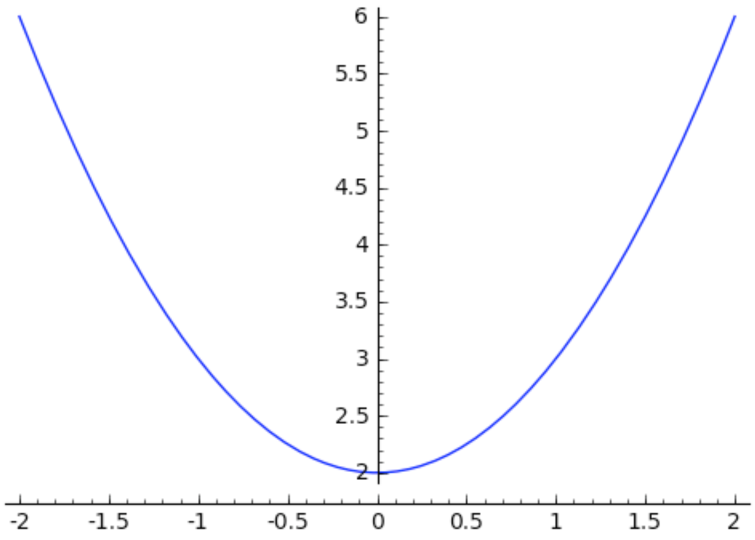
If you don't say what you want, the default is to plot with 200 randomly chosen points (199 line segments). The following plot is equivalent to what you would get if you did not specify the number of points or the randomise option. Note that the plot only looks smooth to the eye - close up, it is still made up of straight lines between points.

```
# DEFAULT plot plots the 199 interpolating lines randomly chosen, etc...
NOTE: the curve only looks smooth to the eye
plot(myFunc,(-2,2), plot_points=200,linestyle='-', randomize=True)
```



```
#use the default options
plot(myFunc,(-2,2))
```

jsMath



```
%hide
```

```
%hide
```

jsMath