STAT221Week01

1. Introduction, Numbers, Strings, Booleans and Sets

Monte Carlo Methods

Introduction

This is a course about **computational statistical experiments** with Monte Carlo methods.

Official Description: This course is about the generation of random numbers and their uses, including computer simulations to mimic and contrast random real-world phenomena. It will provide an intuitive and practical understanding of the basic methods in computational statistics, and show how to implement statistical algorithms to manipulate, visualise and comprehend various aspects of real-world data.

Who does computational statistical experiments?
A ***statistical experimenter*** is a person who conducts a ***statistical experiment***. Roughly, an experiment is an action with an empirically observable outcome (data) that cannot necessarily be predicted with certainty (in the sense that a repetition of the experiment may result in a different outcome). An experimenter attempts to learn about a phenomenon through the outcome of an experiment. An experimenter is often a decision-maker, scientist or engineer.

A simple example of an experiment is the Quincunx and a more popular one is the NZ Lotto or the British Lotto (See British Lottery animation). Other experiments close to home we may see in this course include *earth quakes in NZ*, *sea shells along New Brighton beach*, etc.

Recent technological advances are facilitating computationally intensive statistical experiments based on possibly massive amounts of empirical observations, in a manner that was not viable a decade ago. Hence, a successful decision-maker, scientist or engineer in most specialisations today is a **computational statistical experimenter**.

A computational statistical experimenter has to *tell a machine what to do* with the data, i.e. *program the machine*. In addition, statistical experimenters use a mathematically formal way of thinking about their experiments. They use set theory, probability theory and other branches of pure and applied mathematics through established statistical theory to reach their administrative, scientific and engineering decisions from their data.

This course is designed to help you take the first steps along this path.

What is Sage and why are we using it?
We will be using Sage for our *hands-on* work in this course. Sage is a free open-source mathematics software system licensed under the GPL.
Sage can be used to study mathematics and statistics, including algebra, calculus, elementary to very advanced number theory, cryptography, commutative algebra, group theory, combinatorics, graph theory, exact linear algebra, optimization, interactive data visualization, randomized or Monte Carlo algorithms, scientific and statistical computing and much more. It combines various software packages into an integrative learning, teaching and research experience that is well suited for novice as well as professional researchers.

Sage is a set of software libraries built on top of Python, a widely used general purpose programming language. Sage greatly enhance Python's already mathematically friendly nature. It is one of the languages used at Google, US National Aeronautic and Space Administration (NASA), US Jet Propulsion Laboratory (JPL), Industrial Light and Magic, YouTube, and other leading entities in industry and public sectors (read more...). Scientists, engineers, and mathematicians often find it well suited for their work. Obtain a more thorough rationale for Sage from Why Sage? and Success Stories, Testimonials and News Articles. Jump start your motivation by taking a Sage Feature Tour right now!

For course history and archive from 2009, 2008 and 2007 please see STAT 218: Computational Methods in Statistics.

Interaction

This is an interactive SAGE worksheet from the SAGE Notebook and interactive means...

We will **embed relevant videos** in the SAGE Notebook. To wrap up the ideas so far, See the chief economist at Google talk about statistics being the dream job for 2010's, free virtual academic resources such as The Khan Academy and a rapid introduction to SAGE Notebook in the following three videos.

We will **formally present mathematical and statistical concepts** in the SAGE Notebook.

$$\sum_{i=1}^{5} i = 1+2+3+4+5 = 15, \qquad \prod_{i=3}^{6} i = 3\times 4\times 5\times 6 = 360$$

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}, \qquad \lim_{x\to\infty} \exp(-x) = 0$$

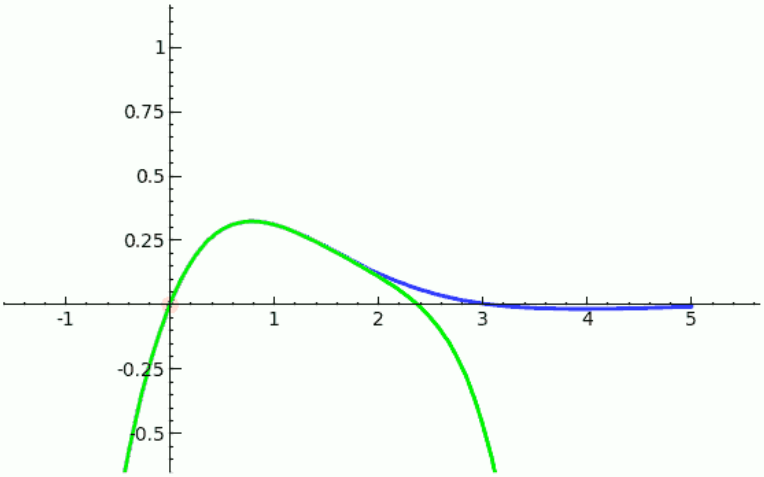
$$\{\alpha,\beta,\gamma,\delta,\epsilon,\zeta,\mu,\theta,\vartheta,\phi,\varphi,\omega,\sigma,\varsigma,\Gamma,\Delta,\Theta,\Phi,\Omega\}, \qquad \forall x\in X, \quad \exists y \leq \epsilon, \ldots$$

We will use interactive visualisations to convey concepts when possible. See the Taylor Series interact by Harald Schilly below.

jsMath

order

$$f(x) = e^{-x}\sin(x)$$
$$\hat{f}(x;0) = x - x^2 + \frac{x^3}{3} - \frac{x^5}{30} + \frac{x^6}{90} - \frac{x^7}{630} + \mathcal{O}(x^8)$$



However, there is no substitute for taking notes on paper with a pencil or pen. The exam is based on pencil-paper skills!

We will **write computer programs** within cells in the SAGE Notebook right after we learn the concepts. Thus, there is a significant overlap between lectures and labs in this course. We will **interactively visualise and analyse live data** with our computer programs in the SAGE Notebook.

Let us visualize the Stock Market data, fetched from Yahoo and Google by William Stein next by placing the cursor in the cell below and clicking the evaluate link at the bottom left of the purple rectangular outline of the cell.

```
%hide
```
evaluate

jsMath

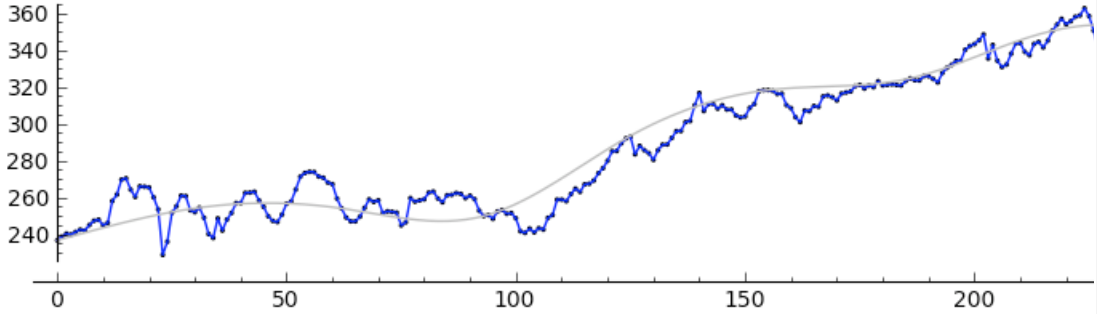
symbol aapl

other_symbol

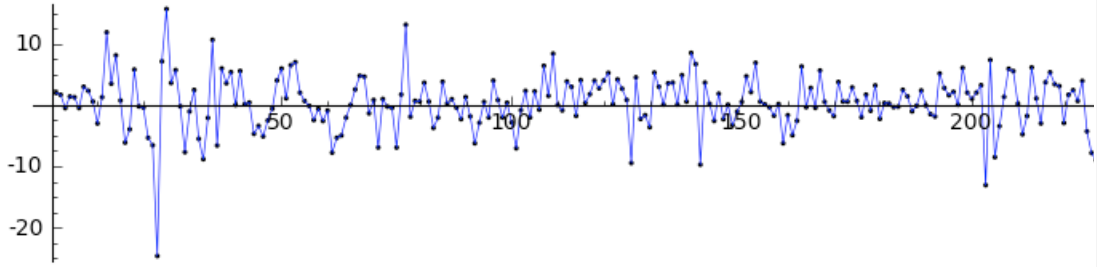
spline_samples 8

AAPL (339.72)

Price during last 52 weeks:
Grey line is a spline through 8 points (do not take seriously!):



Difference from previous day:



200day_moving_avg	323.794
50day_moving_avg	349.022
52_week_high	364.90
52_week_low	199.25
avg_daily_volume	18062400
book_value	59.353
change	-4.814
dividend_per_share	0.00
dividend_yield	N/A
earnings_per_share	17.923
ebitda	22.613B
market_cap	313.0B
price	339.746
price_book_ratio	5.81
price_earnings_growth_ratio	0.79
price_earnings_ratio	19.22
price_sales_ratio	4.16
short_ratio	0.60
stock_exchange	"NasdaqNM"
volume	12866317

jsMath

Let us fetch the CO2 data from US National Oceanic and Atmospheric Association and see it using the program by Marshall Hampton.

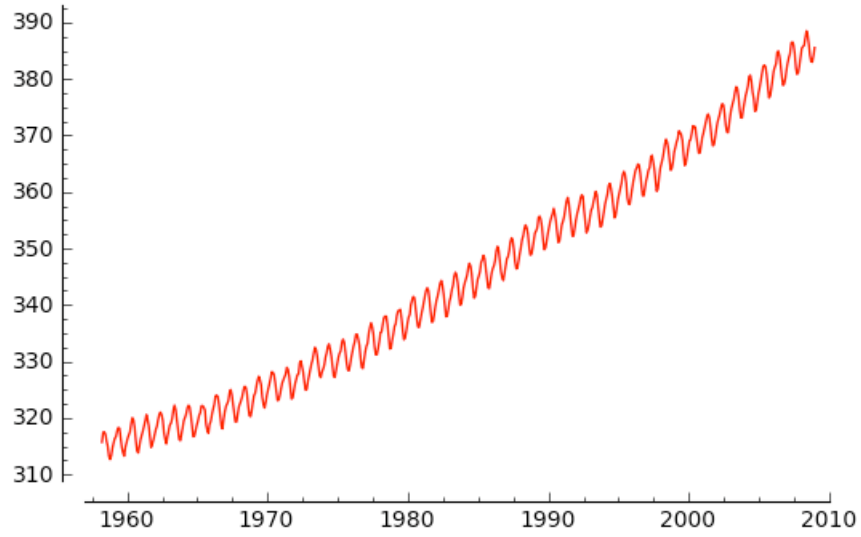
```
%hide
```

start_date 1958

end_date 2009

```
CO2 monthly averages at Mauna Loa (interpolated), from NOAA/ES

# File Creation:  Tue Mar  8 11:57:36 2011
```



Numbers, Strings, Booleans

This lab will start by showing you some of the basic numeric capabilities of SAGE.

1. A worksheet cell is the area enclosed by a gray rectangle.
2. You may type any expression you want to evaluate into a worksheet cell. We have already put some expressions into this worksheet.
3. When you are in a cell you can evaluate the expression in it by pressing <SHIFT><ENTER> or just by clicking the evaluate button below the cell.

To start with, we are going to be using SAGE like a hand-held calculator. Let's perform the basic arithmetic operations of addition, subtraction, multiplication, division, exponentiation, and remainder over the three standard number systems: Integers denoted by $\mathbb{Z}$, Rational Numbers denoted by $\mathbb{Q}$ and Real Numbers denoted by $\mathbb{R}$. Let us recall the real number line and the basics of number systems next.

Let us get our fingers dirty with some numerical operations in SAGE. Note that anything after a '#' symbol is a comment - comments are ignored by SAGE but help programmers to know what's going on.

Example 1: Integer Arithmetic

Try evaluating the cell containing 1+2 below by placing the cursor in the cell and pressing <SHIFT><ENTER>. Next, modify the expression and evaluate it again. Try 3+4, for instance.

```
1+2 # one is being added to 2
```

jsMath

```
3
```

```
3-4 # subtracting 4 from 3
```

```
-1
```

The multiplication operator is '*', the division operator is '/'.

```
2*6 # multiplying 2 by 6
```

```
12
```

```
15/5 # dividing 15 by 5
```

The exponentiation operator is '^'.

```
2^3 # exponentiating 2 by 3, i.e., raising 2 to the third power
```

Being able to finding the remainder after a division is surprisingly useful in computer programming.

```
11%3 # remainder after 11 is divided by 3; i.e., 11=3*3+2
```

Another way of referring to this is 11 *modulus* 3, which evaluates to 2. '%' is the modulus operator.

You try

Try typing in and evaluating some expressions of your own. You can get new cells above or below an existing cell by placing the cursor just above or below an existing cell and clicking after you see a purple bar.

What happens if you put space between the characters in your expression, like "`1 + 2`" instead of "`1+2`"?.

Example 2: Operator Precedence for Evaluating Arithmetic Expressions

Sometimes we want to perform more than one arithmetic operation with some given integers. Suppose, we want to "divide 12 by 4 then add the product of 2 and 3 and finally subtract 1." Perhaps this can be achieved by evaluating the expression "12/4+2*3-1"?

But could that also be interpreted as "divide 12 by the sum of 4 and 2 and multiply the result by the difference of 3 and 1"?

In programming, there are rules for the order in which arithmetic operations are carried out. This is called the *order of precedence*.

jsMath

The basic arithmetic operations are: `+, -, *, %, /, ^`. The order in which operations are evaluated are as follows:

1. ^ Exponents are evaluated right to left
2. *, %, / Then multiplication, remainder and division operations are evaluated left to right
3. +, - Finally, addition and subtraction are evaluated left to right

When operators are at the same *level* in the list above, what matters is the evaluation order (right to left, or left to right).

Operator precedence can be forced using parenthesis.

```
(12/4) + (2*3) - 1 # divide 12 by 4 then add the product of 2 and 3 and
finally subtract 1
```

```
12/4+2*3-1 # due to operator precedence this expression evaluates
identically to the parenthesized expression above
```

Operator precedence can be forced using nested parentheses. When our expression has nested parenthesis, i.e., one pair of parentheses inside another pair, the expression inside the inner-most pair of parentheses is evaluated first.

```
(12/(4+2)) * (3-1)  # divide 12 by the sum of 4 and 2 and multiply the
result by the difference of 3 and 1
```

You try

Try writing an expression which will subtract 3 from 5 and then raise the result to the power of 3.

Find out for yourself what we mean by the precedence for exponentiation (^) being from right to left: What do you think the expression `3^3^2` would evaluate to? Is it the same as `(3^3)^2` (27 squared) or `3^(3^2)` (3 raised to the power 9)? Try typing in the different expressions to find out:

Find an expression which will add the squares of four numbers together and then divide that sum of squares by 4.

Find what the precedence is for the modulus operator % that we discussed above: try looking at the difference between the results for `10%2^2` and `10%2*2` (or `10^2+2)`. Can you see how Sage is interpreting your expressions?

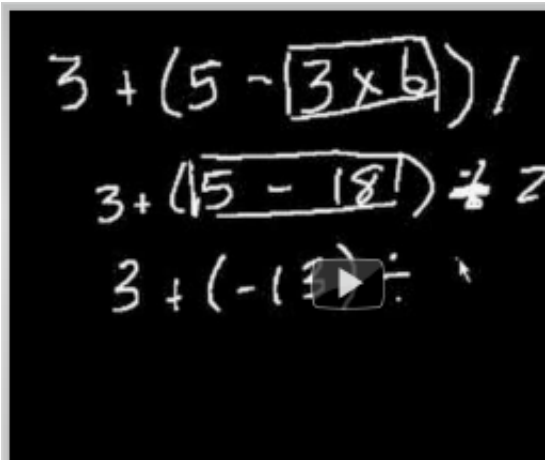
Note that when you have two operators at the same precedence level (like % and *), then what matters is the order - left to right or right to left. You will see this when you evaluate `10%2*2.`

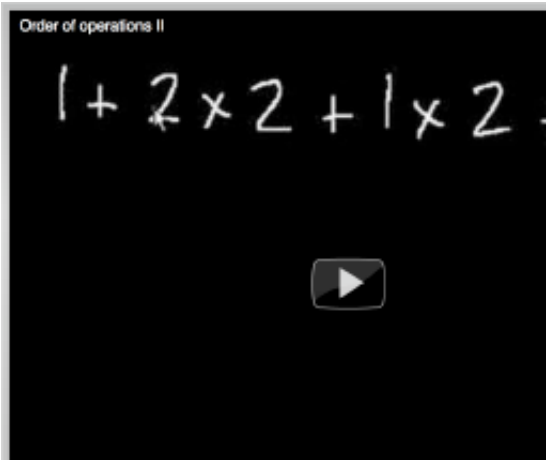
jsMath

Does putting spaces in your expression make any difference?

The lesson to learn is that it is always good to use the parentheses: you will make it clear to someone reading your code what you mean to happen as well as making sure that the computer actually does what you mean it to!

Try these videos to get some practice if you are rusty with order of operations.





Example 3: Rational Arithmetic

So far we have been dealing with integers. Integers are a *type* in SAGE. Algebraically speaking, integers, rational numbers and real numbers form a ring. This is something you will learn in a maths course in Group Theory or Abstract Algebra.

```
type(1) # find the data type of 1
```

However, life with only integers is a bit limited. What about values like '1/2'?

This brings us to the rational numbers.

```
type(1/2) # data type of 1/2 is a sage.rings.rational.Rational
```

Try evaluating the cell containing 1/2+2 below. Next, modify the expression and evaluate it again. Try 1/3+2/4, for instance.

```
1/2+2 # add one half to 2 or four halves to obtain the rational number 5/2
or five halves
```

jsMath

You can do arithmetic with rationals just as we did with integers.

```
3/4-1/4 # subtracting 3/4 from 1/4
```

```
1/2*1/2 # multiplying 1/2 by 1/2
```

```
(2/5) / (1/5) # dividing 2/5 by 1/5
```

```
(1/2)^3 # exponentiating 1/2 by 3, i.e., raising 1/2 to the third power
```

You try

Write an expression which evaluates to `1` using the rationals `1/3` and `1/12`, some integers, and some of the arithmetical operators - there are lots of different expressions you can choose, just try a few.

```

```

What does Sage do with something like `1/1/5`? Can you see how this is being interpreted? What should we do if we really want to evaluate 1 divided by 1/5?

```

```

```

```

Try adding some rationals and some integers together - what type is the result?

```

```

Example 4: Real Arithmetic (multi-precision floating-point arithmetic)

Recall that real numbers include integers, rational numbers irrational numbers like the square root of 2, pi and Euler's constant e. Real numbers can be thought of as all the numbers in the real line between negative infinity and positive infinity. Real numbers are represented in decimal format, for e.g. 234.4677878.

We can do arithmetic with real numbers and can combine them with integer and rational types in SAGE. Compare the results of evaluating the expressions below to the equivalent expressions using rationals above.

```
0.5+2 # one half as 0.5 is being added to 2 to obtain the real number
2.500..0 in SAGE
```

```
0.75-0.25 # subtracting 0.75 from 0.25 is the same as subtracting 0.75 from
```

jsMath

```
0.5*0.5 # multiplying 0.5 by 0.5 is the same as 1/2 * 1/2
```

```
(2/5.0) / 0.2 # dividing 2/5. by 0.2 is the same as (2/5) / (1/5)
```

```
0.5^3.0 # exponentiating 0.5 by 3.0 is the same as (1/2)^3
```

```
type(0.2) # data type of 0.2 is a sage.rings.real_mpfr.RealLiteral
```

Technical note: Computers can be made to exactly compute in integer and rational arithmetic. But, because computers with finite memory (all computers today!) cannot represent the uncountably infinitely many real numbers, they can only mimic or approximate arithmetic over real numbers using finitely many computer-representable floating-point numbers.

You try

Find the type of `1/2`.

```

```

Try a few different ways of getting the same result as typing (((1/5) / (1/10)) * (0.1 * 2/5) + 4/100))*5/(3/5) - this exact expression has already been put in for you in the cell below you could try something just using floating point numbers. Then see how important the parentheses are around rationals when you have an expression like this - try taking some of the out.

```
((((1/5) / (1/10)) * (0.1 * 2/5) + 4/100))*5/(3/5)
```

```

```

```

```

Example 5: Assignment and variables

In SAGE, the symbol = is the assignment operator. You can assign a numerical value to a variable in SAGE using the assignment operator. This is a good way to store values you want to use or modify later.

(If you have programmed before using a a language like C or C++ or Java, you'll see that SAGE is a bit different because in SAGE you don't have to say what type of value is going to be assigned to the variable.)

```
a = 1    # assign 1 to a variable named a
a        # disclose a
```

Just typing the name of a variable to get the value works in the Sage Notebook, but if you are writing a program and you want to output the value of a variable, you'll probably want to use something like the `print` command

```

```

jsMath

```
b = 2
c = 3
print a, b, c  # print out the values of a and b and c
```

Variables can be *strings* as well as numbers. Anything you put inside quote marks will be treated as a string by SAGE.

```
myStr = "this is a string"   # assign a string to the variable myStr
myStr                        # disclose myStr
```

```
type(myStr)                  # check the type for myStr
```

You can reassign different values to variable names. Using SAGE you can also change the type of the values assigned to the variable (not all programming languages allow you to do this)

```
a = 1
print "Right now, a =", a, "and is of type", type(a) # using , and strings
in double quotes print can be more flexible

a = 1/3 # reassign 1/3 to the variable a
```

```
f=(5-3)^(6/2)+3*(7-2) # assign the expression to f
f # disclose f
```

```
x=2^(1/2)
print x
print x.n()
print x.n(digits=30)
```

You can assign values to more than one variable on the same line, by separating the assignment expressions with a semicolon "`;`". However, it is usually best not to do this because it will make your code easier to read (it is hard to spot the other assignments on a single line after the first one).

```
s = 1; t = 2; u = 3;
print s + t + u
```

You try

Try assigning some values to some variables - you choose what values and you choose what variable names to use. See if you can print out the values you have assigned.

```

```

```

```

Assign the value `2` to a variable named `x`

jsMath

On the next line down in the same cell, assign the value **3** to a variable named **y**

Then (on a third line) put in an expression which will evaluate **x + y**

Now try reassigning a different value to **x** and re-evaluating **x + y**

Example 6: Truth statements and Boolean values

Consider statements like "Today is Friday" or "2 is greater than 1" or " 1 equals 1": statements which are either true or not true (i.e., false). SAGE has two values, `True` and `False` which you'll meet in this situation. These value are called Booleans values, or values of the type Boolean.

In SAGE, we can express statements like "2 is greater than 1" or " 1 equals 1" with *relational operators*, also known as *value comparison operators*. Have a look at the list below.

- < Less than
- > Greater than
- <= Less than or equal to
- >= Greater than or equal to
- == Equal to.
- != Not equal to

Lets try some really simple truth statements.

```
1 < 1          # 1 is less than 1
```

Let us evaluate the following statement.

```
1 <= 1  # 1 is less than or equal to 1
```

We can use these operators on variables as well as on values. Again, try assigning different values to x and y, or try using different operators, if you want to.

```
x = 1          # assign the value 1 to x
y = 2          # assign the value 2 to y
x == y         # evaluate the truth statement "x is equal to y"
```

Note that when we check if something *equals* something else, we use "==", a double equals sign. This is because '=', a single equals sign, is the *assignment* operator we talked about above. Therefore, to test if x equals y we can't write 'x = y' because this would assign y to x, instead we use the equality operator '==' and write 'x == y'.

We can also assign a Boolean value to a variable.

jsMath

```
# Using the same x and y as above
myBoolean = (x == y)   # assign the result of x == y to the variable
myBoolean
```

```
type(myBoolean)        # check the type of myBoolean
```

If we want to check if two things are *not* equal we use '!='. As we would expect, it gives us the opposite of testing for equality

```
x != y                 # evaluate the truth statement "x is not equal to y"
```

You try

Try assigning some values to two variables - you choose what values and you choose what variable names to use. Try some truth statements to check if they are equal, or one is less than the other.

Try some strings (we looked at strings briefly in Example 5 above). Can you check if two strings are equal? Can you check if one string is 'less than' (<) another string. How do you think that Sage is ordering strings (try comparing "fred" and "freddy", for example)?

Example 7: Mathematical constants

Sage has *reserved words* that are defined as common mathematical constants. For example, pi and e behave as you expect. Numerical approximations can be obtained using the `.n()` method, as before.

```
print pi, "~", pi.n()   # print a numerical approximation of the
mathematical constant pi
print e, "~", e.n()     # print a numerical approximation of the
mathematical constant e
```

Sets

jsMath

Let us learn elementary set theory. Sets are perhaps the most fundamental concept in mathematics.

***Set** is a collection of distinct elements*. We write a set by enclosing its elements with curly brackets. Let us see some example next.

1. The collection of $\star$ and $\circ$ is $\{\star, \circ\}$
2. We can name the set $\{\star, \circ\}$ by the letter A and write $A = \{\star, \circ\}$
3. Question: Is $\{\star, \star, \circ\}$ a set?
4. A set of letters and numbers that I like is $\{b, d, 6, p, q, 9\}$
5. The set of first five Greek alphabets is $\{\alpha, \beta, \gamma, \delta, \epsilon\}$

*The set that contains no elements is the **empty set***. It is denoted by

$$\boxed{\emptyset = \{\}}.$$

We say *an element **belongs to** or **does not belong** to a set* with the binary operators

$$\boxed{\in \text{ or } \notin}.$$

For example,

1. $\star \in \{\star, \circ\}$ but the element $\otimes \notin \{\star, \circ\}$
2. $b \in \{b, d, 6, p, q, 9\}$ but $8 \notin \{b, d, 6, p, q, 9\}$
3. Question: Is $9 \in \{3, 4, 1, 5, 2, 8, 6, 7\}$?

We say a set C is a **subset** of a set D and write

$$\boxed{C \subset D}$$

if every element of C is also an element of D. For example,

1. $\{\star\} \subset \{\star, \circ\}$
2. Question: Is $\{6, 9\} \subset \{b, d, 6, p, q, 9\}$?

Set Operations

We can add distinct new elements to an existing set by union operation denoted by $\cup$ symbol. For example

1. $\{\circ, \bullet\} \cup \{\star\} = \{\circ, \bullet, \star\}$
2. Question: $\{\circ, \bullet\} \cup \{\bullet\} =$?

More formally, we write the **union of two sets** A and B as

$$\boxed{A \cup B = \{x : x \in A \text{ or } x \in B\}}.$$

The symbols above are read as "A union B is equal to the set of all x suct that x belongs to A or x belongs to B" and simply means that A union B or $A \cup B$ is the set of elements that belong to A or B.

Similarly, the **intersection of two sets** A and B written as

$$\boxed{A \cap B = \{x : x \in A \text{ and } x \in B\}}$$

means A intersection B is the set of elements that belong to both A and B.

For example

1. $\{\circ, \bullet\} \cap \{\circ\} = \{\circ\}$
2. $\{\circ, \bullet\} \cap \{\bullet\} = \{\bullet\}$
3. $\{\circ\} \cap \{a, b, c, d\} = \emptyset$

The **set difference of two sets** A and B written as

$$\boxed{A \setminus B = \{x : x \in A \text{ and } x \notin B\}}$$

jsMath

means $A \setminus B$ is the set of elements that belong to A and not belong to B.

For example

1. $\{\circ, \bullet\} \setminus \{\circ\} = \{\bullet\}$
2. $\{\circ, \bullet\} \setminus \{\bullet\} = \{\circ\}$
3. $\{a, b, c, d\} \setminus \{a, b, c, d\} = \emptyset$

The **equality of two sets** A and B is defined in terms of subsets as follows:

$$\boxed{A = B \quad \text{if and only if} \quad A \subset B \text{ and } B \subset A}.$$

Two sets A anb B are said to be **disjoint** if

$$\boxed{A \cap B = \emptyset}.$$

Given a **universal set** Ω, we define the **complement** of a subset A of the universal set by

$$\boxed{A^c = \Omega \setminus A = \{x : x \in \Omega \text{ and } x \notin A\}}.$$

An Interactive Venn Diagram

Let us gain more intuition by seeing the unions and intersections of sets interactively. The following interact is from interact/misc page of Sage Wiki.

```
%hide
```

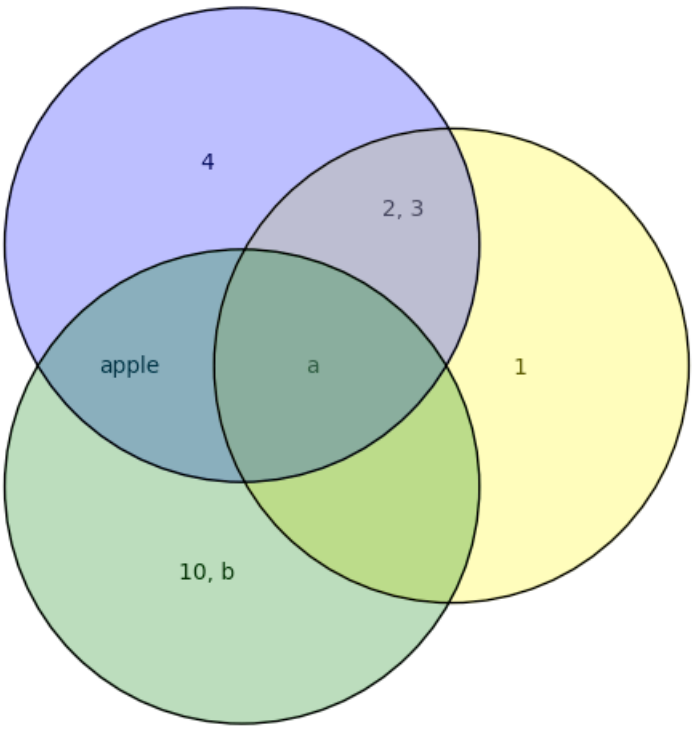
jsMath

X 1,2,3,a

Y 2,a,3,4,apple

Z a,b,10,apple

$$X \cap Y = \{2, 3, a\}$$
$$X \cap Z = \{a\}$$
$$Y \cap Z = \{a, apple\}$$
$$X \cap Y \cap Z = \{a\}$$



Now we'll look at how we can create and manipulate sets in SAGE.

Example 1: Making sets

In SAGE, you do have to specifically say that you want a set when you make it .

```
X = set([1, 2, 3, 4])  # make the set X={1,2,3,4}
X                      # disclose X
```

```
type(X)                # what is the type of X
```

```
4 in X                 # 'is 4 in X?'
```

jsMath

```
5 in X                  # 'is 5 in X?'
```

```

```

```
Y = set([1, 2])         # make the set Y={1,2}
Y                       # disclose Y
```

```
4 not in Y              # 'is 4 not in Y?'
```

```
1 not in Y              # 'is 1 not in Y?'
```

We can add new elements to a set.

```
X.add(5)     # add 5 to the set X
X
```

But remember from lectures that sets contain *distinct* elements.

```
X.add(1)     # try adding another 1 to the set X
X
```

You try

Try making the set Z={4,5,6,7} next. The instructions are in the two cells below

```
# Write in the expression to make set Z ={4, 5, 6, 7}
# (press ENTER at the end of this line to get a new line)
```

```
# Check if 4 is in Z
# (press ENTER at the end of this line to get a new line)
```

Make a set with the value `2/5` (as a rational) in it. Try adding `0.4` (as a floating point number) to the set. Does Sage do what you expect?

```

```

```

```

Example 2: Subsets

In lectures we talked about subsets of sets.

jsMath

Recall that Y is a subset of X if *all* the elements in Y are also in X.

```
print "X is", X
print "Y is", Y
print "Is Y a subset of X?"
Y <= X                          # 'is Y a subset of X?'
```

If you have time: We say Y is a *proper* subset of X if all the elements in Y are also in X but there is at least one element in X that it is *not* in Y. If X is a (proper) subset of Y, then we also say that Y is a (proper) superset of X.

```
X < X     # 'is X a proper subset of itself?'
```

```
X > Y     # 'is X a proper superset of Y?'
```

```
X > X     # 'is X a proper superset of itself?'
```

```
X >= Y     # 'is X a superset of Y?' is the same as 'is Y a subset of X?'
```

Example 3: More set operations

Now let's have a look at the other set operations we talked about in lectures: intersection, union, and difference.

Recall that the intersection of X and Y is the set of elements that are in *both* X and Y.

```
X & Y    # '&' is the intersection operator
```

The union of X and Y is the set of elements that are in *either* X *or* Y.

```
X | Y    # '|' is the union operator
```

The set difference between X and Y is the set of elements in X that are *not in* Y.

```
X - Y    # '-' is the set difference operator
```

You try

Try some more work with sets of strings below.

```
fruit = set(['orange', 'banana', 'apple'])
fruit
```

```
colours = set(['red', 'green', 'blue', 'orange'])
colours
```

jsMath

Fruit and colours are different to us as people, but to the computer, the string 'orange' is just the string 'orange' whether it is in a set called fruit or a set called colours.

```
print "fruit intersection colours is", fruit & colours
print "fruit union colours is", fruit | colours
print "fruit - colours is", fruit - colours
print "colours - fruit is", colours - fruit
```

Try a few other simple subset examples - make up your own sets and try some intersections, unions, and set difference operations. The best way to try new possible operations on a set such as X we just created is to type a period after X and hit <TAB> key. THis will bring up all the possible methods you can call on the set X.

```
mySet = set([1,2,3,4,5,6,7,8,9])
```

```
mySet.          # try placing the cursor after the dot and hit <TAB> key
```

```
mySet.add?      # you can get help on a method by adding a question mark
```

```

```

```

```

Infact, there are two ways to make sets in SAGE. We have so far used the python set to make a set. However we can use the SAGE Set to maka sets too. SAGE Set is more mathematically consisitent. If you are interested in the SAGE Set go to the source and work through the reference on sets. But, first let us appreciate the difference between set and Set!

```
X = set([1, 2, 3, 4])  # make the set X={1,2,3,4} with python set
X                      # disclose X
```

```
type(X)                # this is the set in python
```

```
anotherX = Set([1, 2, 3, 4])   # make the set anotherX={1,2,3,4} in SAGE Set
anotherX
```

```
type(anotherX)                 # this is the set in SAGE and is more mathy
```

```

```

```

```

```

```

jsMath

Post-Tutorial

After the tutorial, a few questions came up about arithmetical operators, numbers, and sets. Here are some short comments and demonstrations of things we did not cover in the tutorial.

Operator precedence

We gave a list for the precence of basic arithmetic operators:

1. ^ Exponents are evaluated right to left
2. *, %, / Then multiplication, remainder and division operations are evaluated left to right
3. +, - Finally, addition and subtraction are evaluated left to right

Note that When operators are at the same *level* in the list above, what matters is the evaluation order (right to left, or left to right).

This means that if we have the expression `10%2*2`, this is the equivalent of `(10%2)*2 = 0*2 = 0` because the modulus operator % is at the same precedence level as the multiplication operator * and the order of evaluation is **left** to **right**.

```
10%2*2
```

However, if we have `10%2^2` then this is the equivalent of `10%(2^2)` because the exponentiation operator ^ is at a higher level than the modulus operator %.

```
10%2^2
```

Sage real literals and Python floating point numbers

You don't need to know this but it might be interesting if you are into computing ...

We showed how you can find the type of a number value and we demonstrated that by default, Sage makes 'real' numbers like 3.1 into Sage real literals. This is an extra-fancy number type Sage developed because Sage is used extensively by mathematicians and others who want their number types to have very specific and 'mathy' properties. If you were just using Python (the programming language underlying most of Sage) then a value like 3.1 would be a floating point number or `float` type. Python has some interesting extra operators that you can use with Python floating point numbers, which also work with the Sage rings integer type but not with Sage real literals.

```
X = float(3.1) # convert the default Sage real literal 3.1 to a float 3.1
type (X)
```

```
3//2 # floor division
```

```
3.3//2.0 # this will give an error - floor division is not defined for Sage
real literals
```

```
float(3.5)//float(2.0)
```

```
type (3) # the default Sage rings integer type
```

jsMath

```
X = int(3) # conversion to a plain Python integer type
type(X)
```

```
3/2 # see the result you get when dividing one default Sage rings integer
type by another
```

One of the differences of Sage rings integers to plain Python integers is that result of dividing one Sage rings integer by another as a rational. This probably seems very sensible, but it is not what happens at the moment with Python integers.

```
int(3)/int(2) # division using python integers is "floor division" (but this
will change soon...)
```

In doing this, Python is similar to many other programming languages (like C, C++, Java ...) but it can be a bit unexpected unless you are used to it. Python is however changing (see this article for the gory details if you are really interested...). For the case when people actually want what they have called "floor division" (essentially getting the whole number part of the result of the division), then there is a special floor division operator `//`. Again, this works with Sage rings integers (any Python integers) and floats but not with Sage real literals.

```
3//2
```

```
float(3.5)//float(2.0)
```

```
3.5//2.0
```

Python also provides a rather nice function for getting both the whole number and the remainer parts of the division at once //. You guessed it .. this works with Sage rings intgers (any Python integers) and floats but not with Sage real literals.

```
divmod(3,2)
```

```
divmod(float(3.5), float(2.0))
```

```
divmod(3.5, 2.0)  # this will give an error because the divmod function is
not defined for Sage real literals
```

In the tutorial we showed the .n method. If X is some Sage real literal and we use `X.n(20)` we will be asking for 20 *bits* of precision, which is about how many bits in the computer's memory will be allocated to hold the number. If we ask for X.n(digits=20) will be asking for 20 *digits* of precision, which is not the same thing. Also note that 20 digits of precision does not mean showing the number to 20 decimal places, it means all the digits including those in front of the decimal point.

```
X=3.55555555
X.n(digits = 3)
```

jsMath

```
X.n(3) # this will use 3 bits of precision
```

If you want to actually round a number to a specific number of decimal places, you can also use the `round(...)` function.

```
round(X,3)
```

```
help(round)
```

Sets

In class we talked about sets and in the tutorial a few questions came up about the order in which Sage displays the elements in a set. The key is to remember that sets are *unordered*: a set {1, 2, 3} is the same as the set {2, 1, 3} is the same as the set {3, 1, 2} ... (soon we will talk about *ordered* collections like lists where order is important - for the moment just remember that a set is about collections of *unique* values or objects). Remember also that the (lower case s) set is a python type (in contrast to the 'more mathy' Sage Set with a capital S). For many 'non-mathy' computing purposes, what matters about a set is the speed of being able to find things in the set without making it expensive (in computer power) to add and remove things, and the best way of doing this (invisible to the programmer actually using the set) is to base the set on a hash table. Don't worry if you don't understand a word of that - you don't need to know it for this course - but I think that in practical terms what is does mean is that the ordering that Sage uses to actually display a set is related to the hash values it has given to the elements in the set which may be totally *unrelated* to what you or I would think of as any obvious ordering based on the magnitude of the values in the set, or the order in which we put them in, etc. Remember, you don't need to know anything about hash values and the 'under-the-hood' construction of sets for this course - this explanation is just to give a bit more background on something that some students commented on in the tutorial.

(However, if you are interested, you could have a look at this link for a bit more about the technical aspects of sets.)

```
aSet = set([2/5, 0.2, 1/7, 0.1])
aSet
```

```
aSet.add(0.3)
aSet
```

Python also provides something called a frozenset, which you can't change like an ordinary set

```
aFrozenSet = frozenset([2/5, 0.2, 1/7, 0.1])
aFrozenSet
```

```
aFrozenSet.add(0.3)
```

You can find more about the Python set and frozenset types in the Python built-in types documentation (the sets stuff is just over half way down that page).

```

```

jsMath