SAGEMATH

1 Purpose and Use

Microsot Word program, for example, cannot teach us to write but it can help us write. In the same way *SageMath* program cannot teach us math but it can help us to do math. Mathematical software of this kind liberates us from routine calculations. However, routine calculations cannot liberate us from understanding, as some people thinks. This kind of software is today what there were logarithmic tables and calculators in the past. And more than this - this kind of software a) performs more powerful numerical calculations, b) performs symbolic calculations, and c) help us tu visualise problems in order to better understand what happens and to gain qualitative answers.

From *Wiki*: *SageMath* (previously Sage or SAGE, "System for Algebra and Geometry Experimentation") is a computer algebra system with features covering many aspects of mathematics, including algebra, combinatorics, graph theory, numerical analysis, number theory, calculus and statistics.

From *Official Sage Tutorial*: *Sage* is free, open-source math software that supports research and teaching in algebra, geometry, number theory, cryptography, numerical computation, and related areas. Both the *Sage* development model and the technology in *Sage* itself are distinguished by an extremely strong emphasis on openness, community, cooperation, and collaboration: we are building the car, not reinventing the wheel. The overall goal of *Sage* is to create a viable, free, open-source alternative to *Maple*, *Mathematica*, *Magma*, and *MATLAB*.

You can use *SageMath* basically in two ways:

1. You can use *Sage Cell Server* on the web page *https://sagecell.sagemath.org (https://sagecell.sagemath.org)*, type a command (or commands) in the cell and execute it (them) by pressing `shift + enter` combination on keyboard. This is suitable for smaller tasks.
2. You can use a notebook, a file which consists of cells. For every cell you can determine its purpose: to be a text cell for writing text or to be an input code cell to write commands. When you execute an input code cell, the result will be put in corresponding output code cell. This document is such a file, and I am currently typing the text in the text cell. A notebook is suitable for presentations or for more complex tasks. I use this notebook for presentation of *SageMath*. You can create notebook files in a *SageMathCloud* by registering on *cocalc.com* (it is free) or in your computer by downloading *SageMath* software from http://www.sagemath.org (http://www.sagemath.org). If you want to use a notebook for *SageMath* tasks there is a lot of information on internet to learn how to use it. For example on *cocalc.com* there is `Help` where you can find instructions how to use notebooks and Markdown language for formatting text. When you need information about *SageMath* commands, the reference manual of each function, constant or command is accessed by writing the question mark "?" after its name. If you type "??" insted of "?", the corresponding source code will be displayed. If you write the first letters of the word and press the tab key, you will get all command names starting with these letters.

For users who use *Python* programing language: *SageMath* is an extension of Python language, so you can type Python code in code cells. For example, in code cells, commments begin with "#".

2 Basic Arithmetic

Arithmetic operations are typed as on a calculator. The sign for multiplication is "*", the sign for division is "/" and the sign for exponentiation is "^". Let's start with integers. The sign for integer division (division with reminder) is "//" for the result, and "%" for the reminder. There are no limits on the size of an integer.

```
In [1]:  1+2^2*13 #The hierarchy of operations is the standard one: if there is am
         biguity in an expression, first do exponentiation, then multiplication an
         d division, and then addition and subtraction.

Out[1]:  53
```

```
In [10]: (1+2^2*13)//10, (1+2^2*13)%10 #You can type several commands separated by
         comma. The results are displayed consecutively:

Out[10]: (5, 3)
```

If we put commands on different lines, *SageMath* will show in the output code cell only the result of the last command:

```
In [21]: (1+2^2*13)//10
         (1+2^2*13)%10

Out[21]: 3
```

If we want to see all results we must add command `print` :

```
In [22]: print((1+2^2*13)//10)
         (1+2^2*13)%10

         5

Out[22]: 3
```

We can use relations < (less), == (equal), != (not equal), <= (less or equal), and >= (greater or equall) which we can combine with connectives *or*, *and* and *not*. The sign of equality = has not standard meaning but programmer's meaning - it means assignment. For example, $x = 3$ means that variable *x* will get value 3.

```
In [2]:  1==2 or 2==2

Out[2]:  True
```

We can factorize:

```
In [2]: 48.factor() # Python users will recognise this syntax, the so called meth
        od notation, when the sign of operation is placed after the name of objec
        t on which it acts.

Out[2]: 2^4 * 3
```

Command `show` gives a nicer expression:

```
In [6]: show(48.factor())

Out[6]:
```

We can find the greatest common divisor (*gcd*) and the least common multiple (*lcm*) of two numbers:

```
In [11]: gcd(246,48), lcm(246,48)

Out[11]: (6, 1968)
```

Let's calculate with rational numbers:

```
In [7]: 2/3+1/6

Out[7]: 5/6

In [8]: show(2/3+1/6)

Out[8]:
```

Real numbers have standard decimal or scientific notation and we can calculate them up to a given precision. The command for numerical aproximation is *n*. For example, if we want to aproximate number π (denoted *pi* in SageMath), we can use the function notation `n(pi)` or the method notation `pi.n()` . With the optional parameter *digits* we can determine which number of digits we want to get in the approximation. Even better, the optional parameter *round* determines the precision of the approximation: how much decimal places after the period we want (the last digit is rounded).

In [3]:
```
print(pi) #SageMath will do nothing, because it doesn't know better notat
ion for pi

print(pi.n()) #the method notation

print(n(pi)) #the function notation

print(pi.n(digits=5))
print(pi.n(digits=30))
print (round(pi,4)) # there is no the method version for command "round".
print(1.2e3*2.1e2) # This is a scientific notation: 1.2e3 means 1.2*10^3
```

```
pi
3.14159265358979
3.14159265358979
3.1416
3.14159265358979323846264338328
3.1416
252000.000000000
```

We can translate (approximately) fractions to decimals and vice versa.

In [29]:
```
print(n(2/3))
print(Rational(n(2/3)))
print(Rational(n(pi)))
print(Rational(0.25))
```

```
0.666666666666667
2/3
245850922/78256779
1/4
```

For the square root we have command `sqrt` . For higher roots we must use exponential notation. If we want the exact result, *SageMath* will give us in some sense the best possible notation for the result:

In [30]:
```
print(sqrt(8))
print(8^(1/3))
print((-8)^(1/3))
```

```
2*sqrt(2)
2
2*(-1)^(1/3)
```

3 Basic Algebra

In *SageMath* we distinguish **constants** and **variables** in a mathematical sense. A constant is a name of a specific object, while a variable is a name of an intentionaly unspecified objects. In imperative programming terminology constants are called variables because we can change its value during the execution of a program, while variables in a mathematical sense are called symbolic variables. We will follow mathematical terminology.

By assignment statement we determine the value of a constant. For example, the assignment statement $a= 3$ determines the value of constant a to be 3.

Variables have to be declared by a declaration statement. For example, with statement `var('b')` we declare 'b' to be a variable. By defualt x is considered as a variable.

In [1]:
```
a=3
var('b')
print(a)
print(b)
print(a+b)
print(a+b+x)
```

```
3
b
b + 3
b + x + 3
```

Using variables, we can write open descriptions in *SageMath* (expressions which contain variables), insert values for variables, and simplify descriptions.

In [2]:
```
var('a,b') # We declared two variables.
print(a^2+a*b+3*b)
print((a^2+a*b+3*b).substitute(a = 2, b=3)) # After the substitution Sage
Math calculates the closed expression (expression without variables).
print((a^2+a*b+3*b).substitute( b=a)) # After the substitution SageMath s
implifies the open expression.
```

```
a^2 + a*b + 3*b
19
2*a^2 + 3*a
```

Command `expand` "multiplies brackets":

In [38]:
```
var('a,b') # We could inherit this declaration from the previous code cel
l but for the presentation it is better that every cell is maximally inde
npendent.
((a+b)^3).expand().show()
```

Out[38]:

We can factorise descriptions:

```
In [39]: var('a,b')
         (a^3 + 3*a^2*b + 3*a*b^2 + b^3).factor().show()
Out[39]:
```

We have at disposal command `simplify` in various forms:

```
In [40]: ((x+1)^2/(x*(x+1))).expand().show()
         ((x+1)^2/(x*(x+1))).simplify().show() # simplification
         ((x+1)^2/(x*(x+1))).simplify_rational().show() # Simplify_rational() tran
         sforms an expression into a quotient of two polinomials
         ((x+1)^2/(x*(x+1))).simplify_full().show()
Out[40]:
Out[40]:
Out[40]:
Out[40]:
```

```
In [42]: var('a,b')
         (a/(b*(a+b))-b/(a*(a+b))).expand().show()
         (a/(b*(a+b))-b/(a*(a+b))).expand_rational().show()
         (a/(b*(a+b))-b/(a*(a+b))).simplify().show()
         (a/(b*(a+b))-b/(a*(a+b))).simplify_rational().show()
         (a/(b*(a+b))-b/(a*(a+b))).simplify_full().show() # This is the best-.
Out[42]:
Out[42]:
Out[42]:
Out[42]:
Out[42]:
```

```
In [43]: var('a,b')
         (a/((a + b)*b) + b/((a + b)*a)).simplify().show()
         (a/((a + b)*b) + b/((a + b)*a)).simplify_rational().show()
         (a/((a + b)*b) + b/((a + b)*a)).simplify_full().show()
         (a/((a + b)*b) + b/((a + b)*a)).simplify_rational(algorithm='noexpand').s
         how()
Out[43]:
Out[43]:
Out[43]:
Out[43]:
```

Sometimes, command `collect()` is useful:

```
In [48]: (x^2*exp(x)+2*x*exp(x)).collect(exp(x)).show()
Out[48]:
```

We can group terms with common denominator with command `combine` :

```
In [45]: (1/x + (x+2)/x).combine()
Out[45]: (x + 3)/x
```

To simplify expressions with roots we use command `canonicalize_radical()`

```
In [47]: (sqrt(x^2+x)/sqrt(x)).canonicalize_radical().show()
Out[47]:
```

We can define functions (operations) in a standard mathematical way, by a formula which determines what to do with an input to the function to get the corresponding output. For example

```
In [4]: f(x)=x^2
        (f(x)).show()
        f.show()
        (f(2)).show()
        var('a,b')
        (f(a+b)).show()
Out[4]:
Out[4]:
Out[4]:
Out[4]:
```

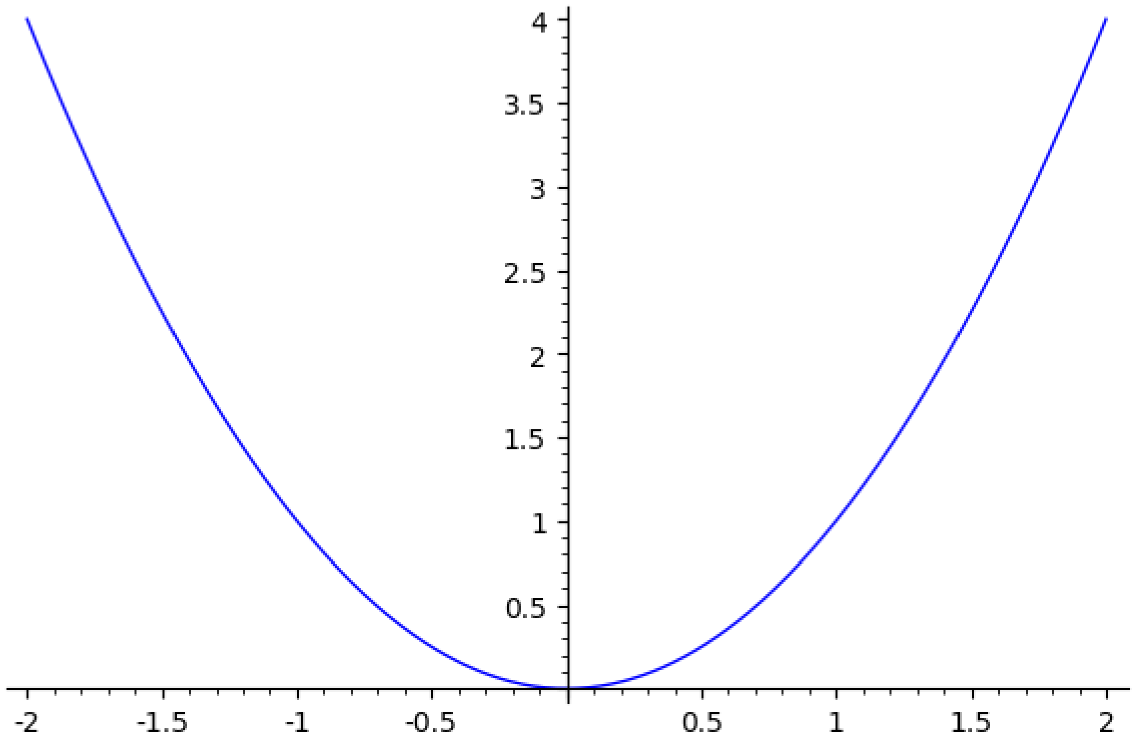
4 Basic Plotting

We can plot graphs of functions and equations with two uknowns in Cartesian coordinate system using command `plot` . We must describe what we want to plot and on what domain of numbers. Furthermore, we can specify optional arguments: the appareance of axes, axes marks. the graph title, color of plotted curves, etc.

For example, we will plot function $f(x) = x^2$ on interval [-2,2]:

```
In [5]: plot(x^2, (x,-2,2))
```

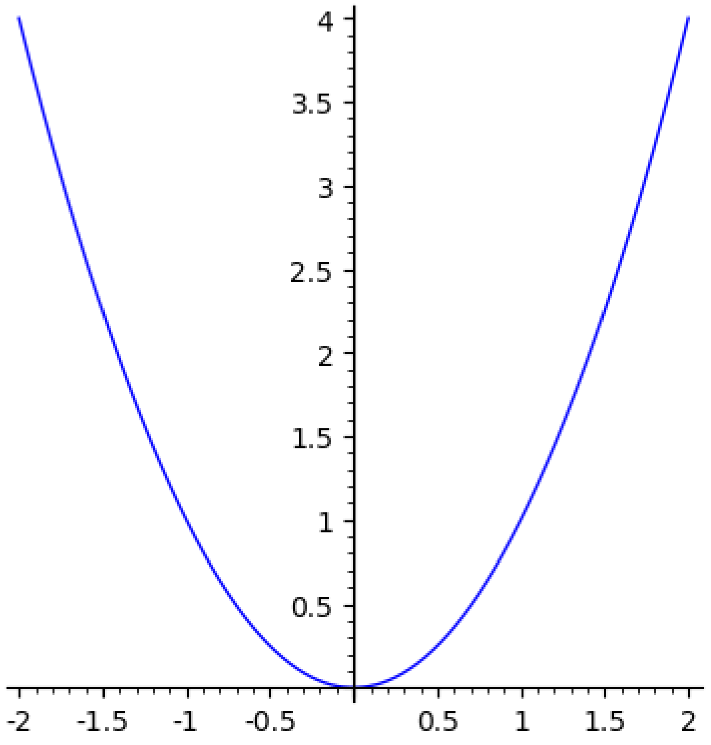
Out[5]:



If we want to have the same units on both axes, we must specify the aspect ratio:

```
In [6]: plot(x^2, (x,-2,2), aspect_ratio = 1)
```

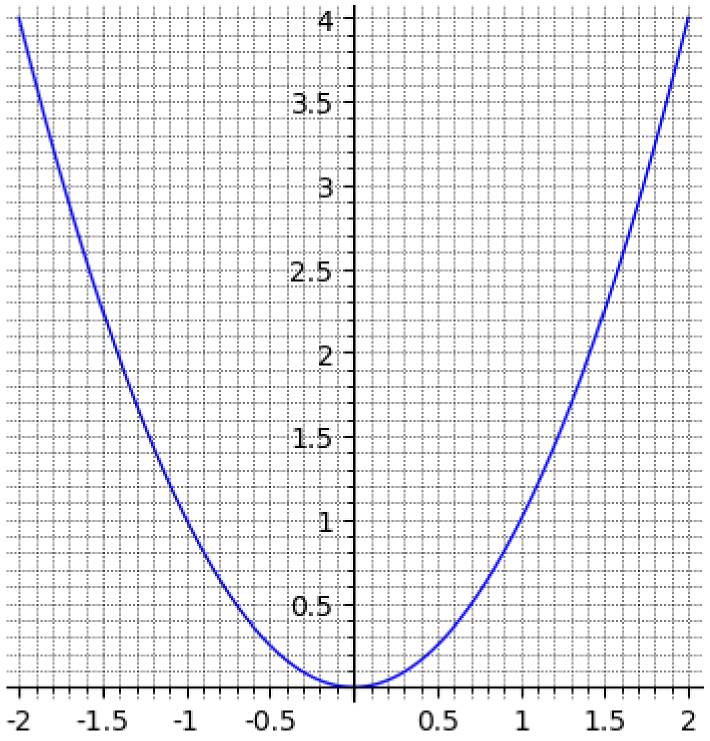
Out[6]:



If we want to see better the input - output pairs, we can plot a grid:

```
In [7]: plot(x^2, (x,-2,2), aspect_ratio = 1, gridlines = 'minor')
```

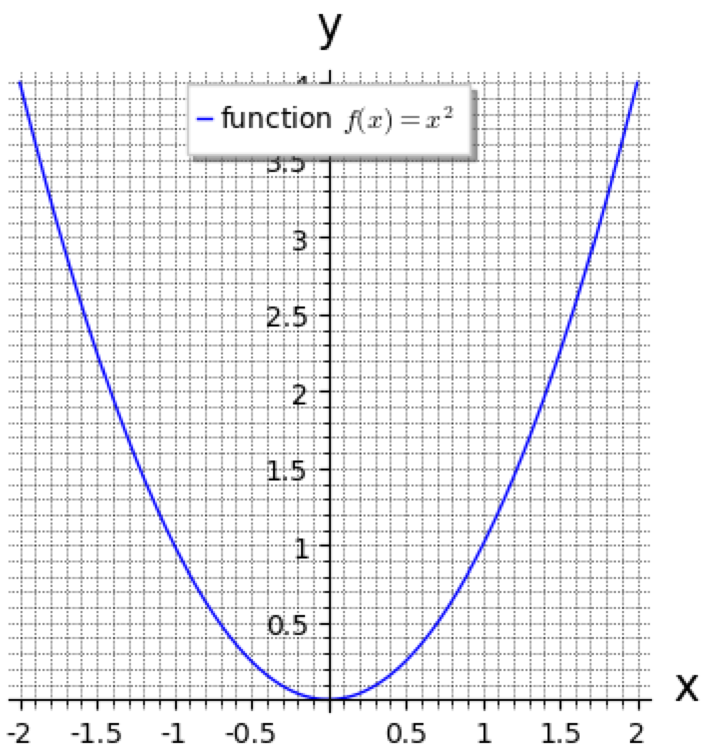
```
Out[7]:
```



And this is a more elaborated display:

In [8]:
```
plot(x^2, (x,-2,2), aspect_ratio = 1, gridlines = 'minor', axes_labels =
['x','y'], legend_label = 'function $f(x) = x^2$')
```

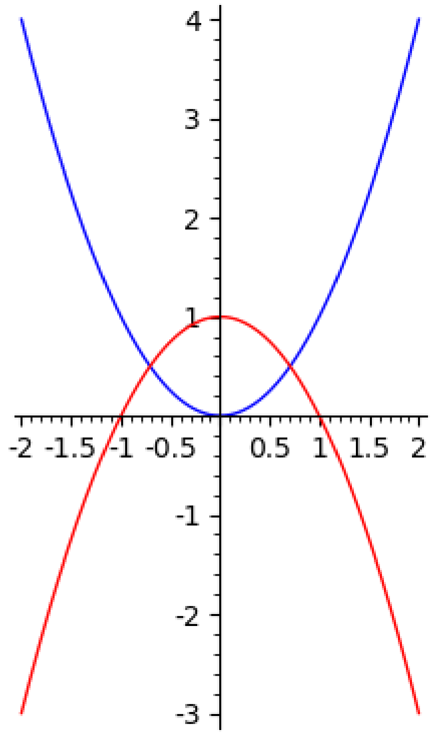
Out[8]:



We can plot several graphs:

In [9]:
```
plot(x^2, (x,-2,2), aspect_ratio = 1)+plot(1-x^2,(x,-2,2), color = 'red')
```

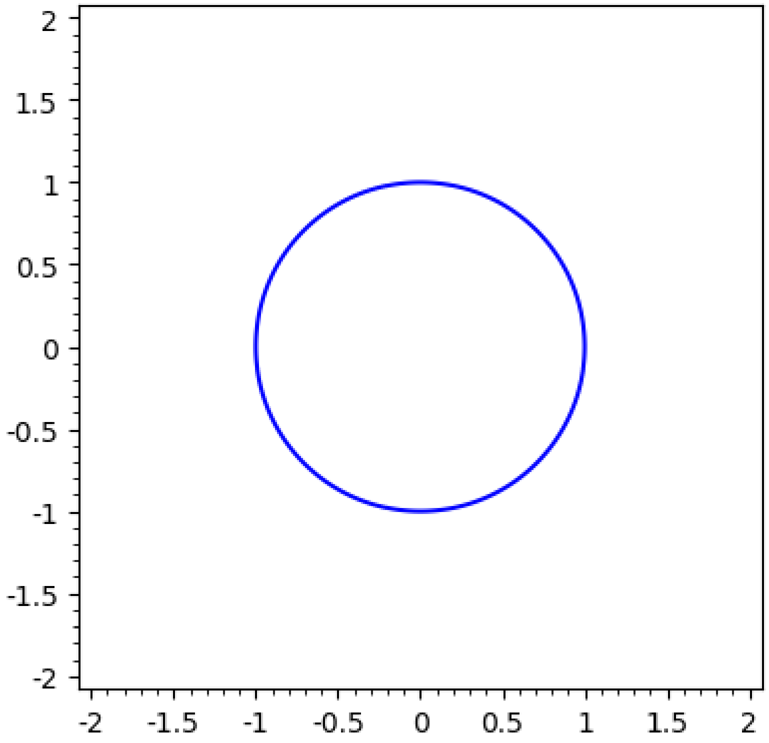
Out[9]:



Also, we can plot curves:

In [11]:
```
var('y')
implicit_plot(x^2+y^2==1,  (x,-2,2),(y,-2,2), aspect_ratio = 1)
```

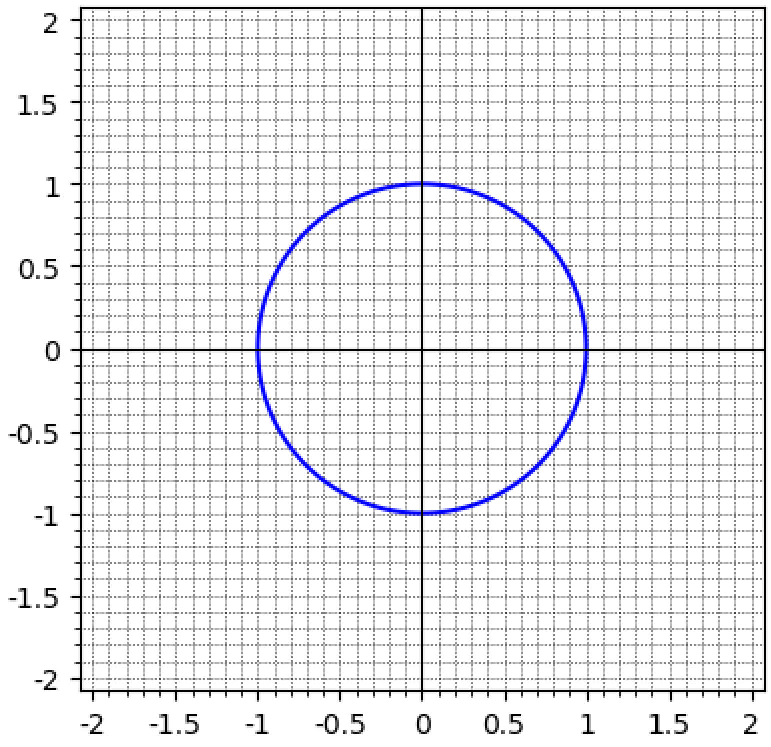
Out[11]:



We can add gridlines and axes:

In [13]:
```
var('y')
implicit_plot(x^2+y^2==1, (x,-2,2),(y,-2,2), aspect_ratio = 1, gridlines
='minor', axes = true)
```

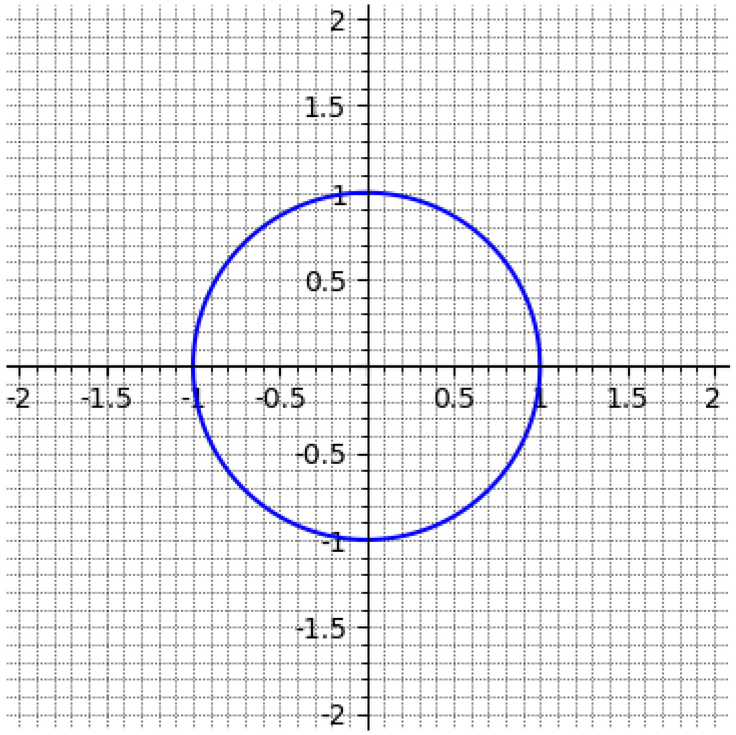
Out[13]:



and remove the frame:

In [14]:
```
var('y')
implicit_plot(x^2+y^2==1, (x,-2,2),(y,-2,2), aspect_ratio = 1, gridlines
='minor', axes = true,frame = false)
```

Out[14]:



5 Solving Equations

We use command `solve` for solving equations: we must specify which equation to solve and for which variable.

In [15]:
```
solve(x^2==2*x,x)
```

Out[15]:
```
[x == 0, x == 2]
```

In [16]:
```
var ('a,b')
solve(a*x==b,x) #This will not be completely correct because the case a =
0 is not considered separately.
```

Out[16]:
```
[x == b/a]
```

In [18]:
```
var ('a,b,c')
show(solve(a*x^2+b*x+c==0,x)) # a != 0
```

Out[18]:

Let's try to solve cubic equation:

In [20]:
```
var ('a,b,c,d')
show(solve(a*x^3+b*x^2+c*x+d==0,x))
```

Out[20]:

Oh, no. Let's solve a specific cubic equation:

In [21]:
```
show(solve(x^3+x^2+x-2==0,x))
```

Out[21]:

Oh, no (once again). Let us single out the third solution on the right hand side (rhs). *SageMath* counts from zero, so this solution has index 2.

In [23]:
```
show(solve(x^3+x^2+x-2==0,x)[2].rhs())
```

Out[23]:

It will be better to find a numerical approximation:

In [26]:
```
print((solve(x^3+x^2+x-2==0,x)[2].rhs()).n())
round(solve(x^3+x^2+x-2==0,x)[2].rhs(),6)
```

```
0.810535713766137
```

Out[26]:
```
0.810536
```

We could have done it more simply by naming the result of the solution:

In [27]:
```
lr=solve(x^3+x^2+x-2==0,x)
show(lr[2]) # it displays the third member of the solutions list.
show(lr[2].rhs()) # it displays the right hand side of  the third member
 of the solutions list.
show(round(lr[2].rhs(),6)) # the numerical aproximation of the right hand
side of the third member
```

Out[27]:

Out[27]:

Out[27]:

Let us solve the following equations:

```
In [3]: show(solve(x^5+x^4==x+1,x))
        show(solve(x^5+x^4==x-1,x))
```

```
Out[3]:

Out[3]:
```

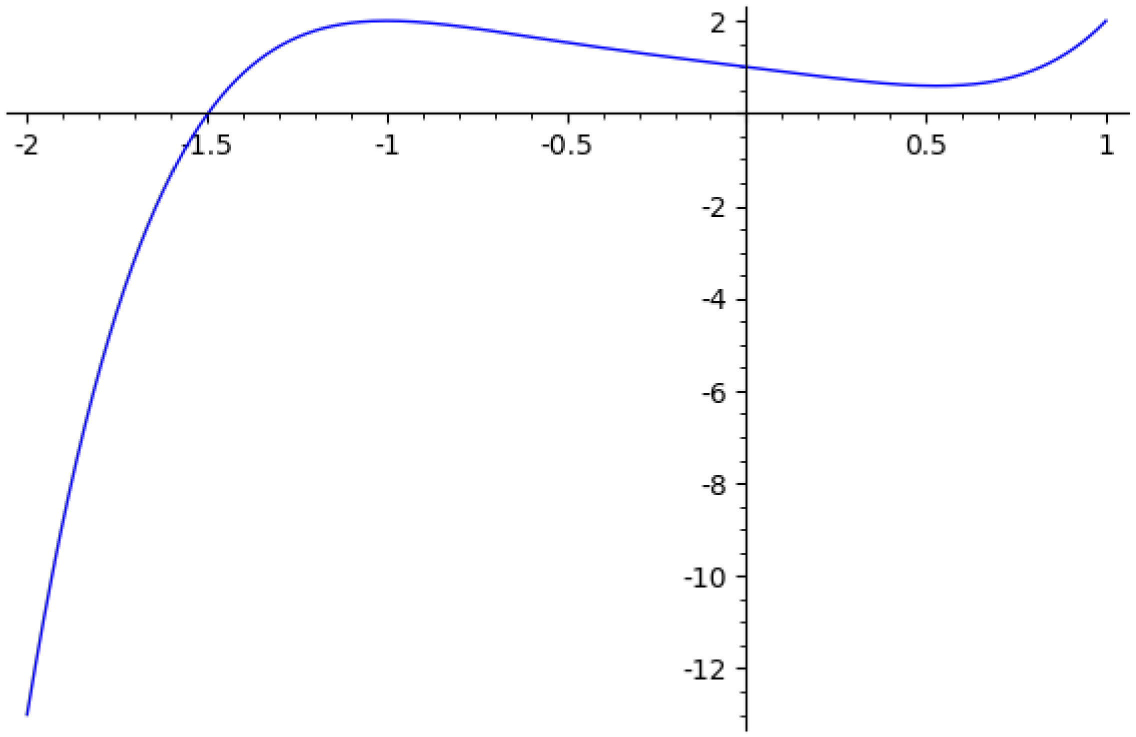
SageMath solved the first equation, while the second did not. In cases when the `solve` command fails to solve an equation analytically (it does not give us an exact solution), whether that equation cannot be solved analytically at all, or it can, but the `solve` command does not have an implemented method, then we must resort to numerical solution. But it is better to solve the equation qualitatively first, find out how many solutions there are and where they are approximately. For real solutions of the equation we have a simple procedure. Let's move all the members of the equation to one side of the equation:
$$x^{5}+x^{4}-x+1=0$$

The solutions of the equation are zeros of the function (any input for which the function returns zero) determined by the expression on the right side of the equation:
$$f(x)=x^5+x^4-x+1$$

The zeros of a function can be qualitatively determined on its graph - these are the common points of the graph of the function and the $ x $-axis:

```
In [3]: plot(x^5 + x^4 - x + 1, (x,-2,1))
```

Out[3]:



Examining the graph for lesser and greater values for $ x $ we can see that the function has only one zero which is approximately equal to $ -1.5 $. Using the `find_root` command, we can numerically approximate it to the desired accuracy:

```
In [1]: find_root(x^5+x^4==x-1,-2,-1)   #SageMath displays 17 reliable digits

Out[1]: -1.4970940487627966
```

In *SageMath* we can also solve multiple equations with multiple unknowns.

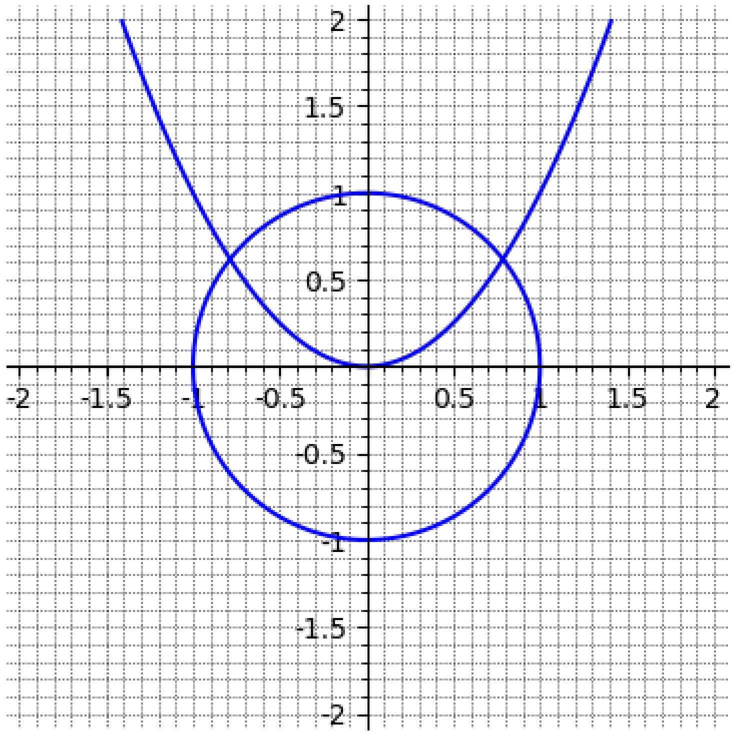
```
In [5]: var('y')
        show(solve([x^2+y^2==1,y==x^2],x,y))

Out[5]:
```

We will qualitatively examine the real solutions by drawing graphs of these equations. The solutions are the coordinates of the common points of these graphs.

```
In [7]: var('y')
        implicit_plot(x^2+y^2-1==0,(x,-2,2),(y,-2,2))+implicit_plot(y-x^2==0,(x,-
        2,2),(y,-2,2), gridlines='minor', axes =true, frame = false)

Out[7]:
```



We see that the solutions are approximately \$ (0.8,0.6) \$ and \$ (- 0.8,0.6) \$. We will now use the `findroot` command to specify the first solution to default SageMath precision (we can see from the symmetry that the second solution differs from the first only in the sign of the first number). However, the `findroot` command requires a little more preparation:

1) it is not part of the standard SageMath tool, so we must import it from the appropriate program module *mpmat*.

2) we must move all members of each equation to one side of the equation and define a list of functions assigned to those sides using the lambda notation (the notation is explained in the first chapter of the book UNDERSTANDING AND DOING MATH - CIRCLE 2).

In [5]:
```
from mpmath import findroot  # we import the command from the specified m
odule
findroot([lambda x, y: x^2+y^2-1, lambda x, y: y-x^2],(0.8,0.6)) # we imp
ort the list of corresponding functions and give the initial guess
```

Out[5]:
```
matrix(
[['0.786151377757423'],
 ['0.618033988749895']])
```