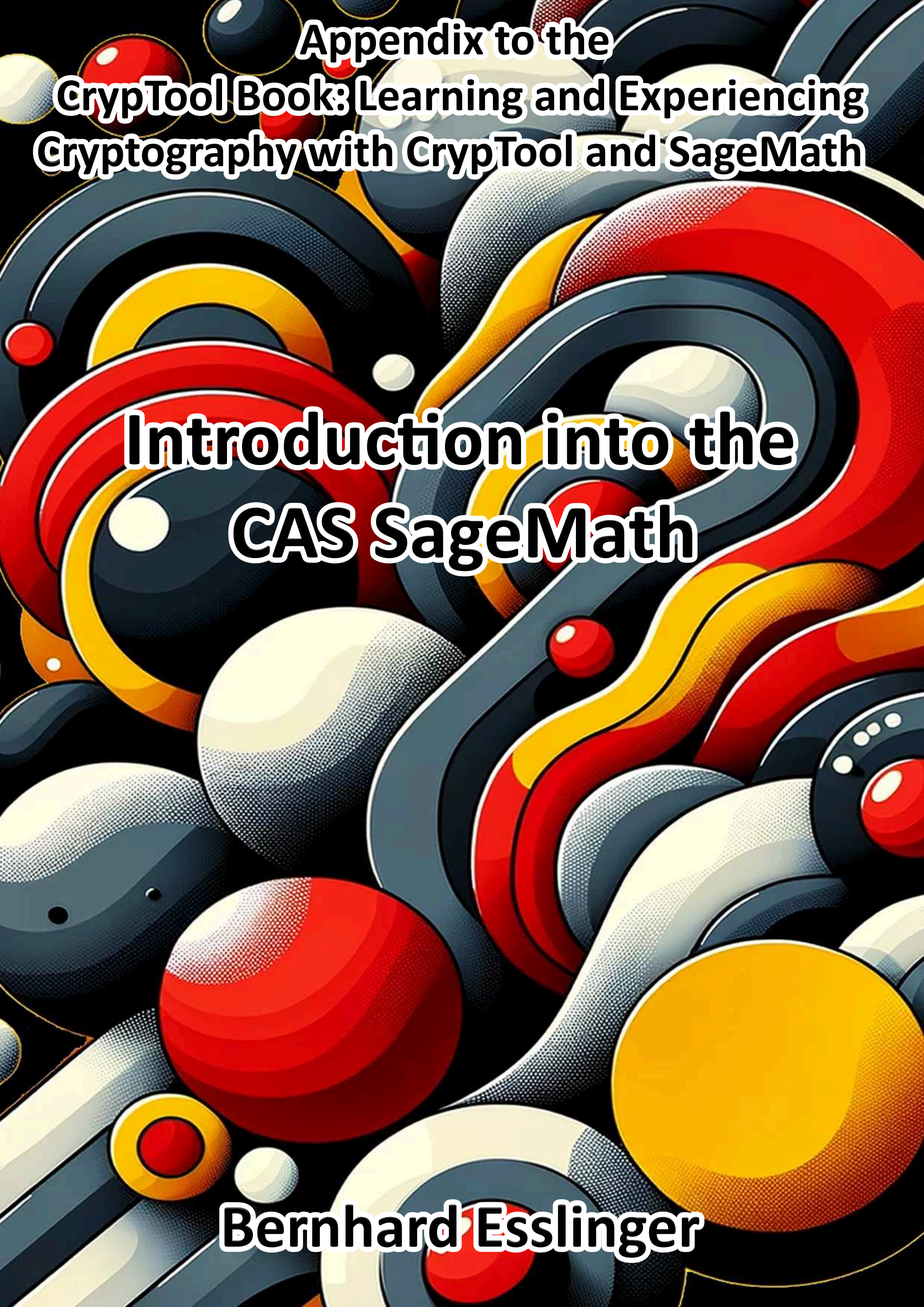
Appendix to the
CrypTool Book: Learning and Experiencing
Cryptography with CrypTool and SageMath

Introduction into the CAS SageMath

Bernhard Esslinger



Appendix to the „CrypTool Book“:
Learning and Experiencing Cryptography with CrypTool and SageMath

Introduction into the CAS SageMath

Bernhard Esslinger

https://www.cryptool.org

Friday 28th June, 2024 (13:15:00)

Suggestion for referencing with Bib(La)TEX:

```
@article{Esslinger:ctb_2024_app_sage_en,
  editor    = {Bernhard Esslinger},
  title     = {{A}ppendix to the {C}ryp{T}ool {B}ook:
               {L}earning and {E}xperiencing {C}ryptography
               with {C}ryp{T}ool and {S}age{M}ath:
               {I}ntroduction into the {CAS} {SageMath}},
  publisher = {CrypTool Project},
  year      = {2024}
}
```

Typesetting software: LATEX
Version control software: Git
Support with translations: DeepL Translator by DeepL.com and ChatGPT 4 by openai.com
Supporting the generation of Tikz diagram: Claude 3.5 Sonnet by claude.ai

Table of Contents

1 Introduction into the CAS SageMath

(Bernhard Esslinger. Last update: Jun 2024)

This chapter, which was designed as an appendix to the CrypTool book, mainly describes the ecosystem of SageMath, i. e. how to use SageMath in different environments. Beginners are often confused by the many variants – this chapter helps with examples and a systematic overview, which is summarized in the tables 1 and 2.

The CrypTool book contains numerous program examples created with SageMath. However, this appendix can also be read on its own as an introduction to SageMath. SageMath is a computer-algebra system (CAS) like Mathematica, Maple, and MATLAB. SageMath supports teaching, study, and research in mathematics. It combines many other high-quality open-source packages[1] and provides access to their functionalities via a common interface, based on the programming language Python.[2]

SageMath is free and can be downloaded from:

`https://www.sagemath.org`

SageMath can be used as a powerful desktop calculator, as a tool to help (undergraduate) students study mathematics, or as a programming environment for prototyping algorithms and research in algorithmic aspects of mathematics. Instead of programming mathematical tasks with SageMath, you can just as well use Python and libraries. The advantage of SageMath is that a number of mathematical constructs are already included and you can also do symbolic math.[3]

You can get a quick impression of SageMath e. g. with the references in Footnote 4.[4] Footnote 5 lists four examples of good documents about SageMath for further reading.[5]

With respect to studying cryptography, SageMath modules can be used to complement a first course in cryptography.[6] Introductions into cryptography using SageMath are in this footnote[7]. Section 2.8 of the CrypTool book contains SageMath samples for many classical cryptographic algorithms.

This introduction uses mainly examples to show what SageMath can do, but it also shows the environments and surroundings, i. e. how to use SageMath in the terminal and in the notebook; how the user controls provided by SageMath work together with those of Jupyter and Matplotlib; or how LaTeX and SageMath work together. This overall view was missing in many other introductions. It is supplemented with many annotated references that have proven to be useful in teaching and learning.

All SageMath examples of this introduction can be found on the CrypTool website:
`https://www.cryptool.org/en/ctbook/sagemath`

[1]Normally you only install the binaries of SageMath. However, if you compile it yourself, you get an impression of how big SageMath is: After downloading the source of SageMath 4.1, it took around 5 hours on an average Linux PC to compile the whole system including all libraries. The compiled version occupied 1.8 GB disk space. With SageMath 9.0 the bz2 file had around 2 GB, and uncompressed > 7 GB. It took me around 40 minutes to compile SageMath 10.3 under Ubuntu 22.04 – following the instructions at `https://sagemanifolds.obspm.fr/install_ubuntu.html`.

[2]There is also an interface to the C language, called Cython, which can speed up functions in SageMath. See `https://openwetware.org/wiki/Open_writing_projects/Sage_and_cython_a_brief_introduction`.
SageMath runs under the operating systems Linux, macOS, and Windows. From SageMath version 8, there was a native Windows installer; from SageMath 9 (released at Jan 1st, 2020), Python 3 (instead of Python 2) is used.

[3] `https://doc.sagemath.org/html/en/reference/calculus/sage/calculus/calculus.html`

[4]- "The SDSU Sage Tutorial", `https://mosullivan.sdsu.edu/sagetutorial/`
`https://mosullivan.sdsu.edu/sagetutorial/sagecalc.html`
- *Sage Quick Reference Cards*, `https://wiki.sagemath.org/quickref`

[5]- "Library": `https://www.sagemath.org/library.html`,
- "Documentation Project": `https://wiki.sagemath.org/DocumentationProject`,
- "Teaching": `https://wiki.sagemath.org/Teaching_with_SAGE`,
- "Calculating with Sage at high schools": `https://doc.sagemath.org/html/de/thematische_anleitungen/sage_gymnasium.html`.
To our knowledge, this is currently only there in German.

[6]- Overview, what crypto currently is in SageMath: `https://doc.sagemath.org/html/en/reference/cryptography/index.html`
- Teaching related aspects of developing crypto in SageMath: `https://groups.google.com/g/sage-devel/c/xVcsTY1C0IE`

[7]- David Kohel's course from 2015: `https://www.sagemath.org/files/kohel-book-2008.pdf`
- "Introduction to Cryptography with Open-Source Software", a very good book from Alasdair McAndrew, CRC, 2011

1.1 The three typical user interfaces of SageMath

Table 1 lists three ways in which SageMath is usually used: two ways each via a terminal and via a Jupyter notebook, and two samples via a website.[8]

1) Terminal based	a) Sage console	$ sage sage: factor(35) sage: ... sage: attach filename.sage	Text output in lines below the command. Graphics output into a file.
	b) Sage program	$ sage script.sage	
2) Jupyter notebook	a) run in browser	Starting the kernel: $ sage -n jupyter <filename.ipynb>	Text and graphics output in the cell's output. Graphics output also into a file.
	b) via VS Code		
3) External website	Via browser remote access on an external website	https://sagecell.sagemath.org/ or https://cocalc.com/	Output in browser

Tab. 1: Overview about the SageMath calling possibilities (user interfaces)

1. The first user interface is **command line** or **terminal** based, as shown in Fig. 1 on the next page. This is normally used when you installed SageMath locally.

   From the command line (terminal, shell) of the operating system, you have two options:[9]

   a) Either the interactive Sage console (`sage:`)[10] where text output appears below the entered command. When you generate graphical output, SageMath creates a file and asynchronously calls an image processing program to display the picture.

   b) As a second option in the terminal you can write SageMath programs (`scripts`) and execute them like a batch file (`$ sage script.sage`). This is used for longer calculations and typical tasks in programs.

2. Secondly, there is also a graphical user interface for SageMath (called a notebook). The most popular notebook today is the **Jupyter notebook** (see Fig. 2 on the following page).

   This option is well suited for interactive or didactic tasks, and is also frequently used by data analysts. Jupyter notebooks are similar to the REPL model (Read-Evaluate-Print-Loop). More about Jupyter can be found in appendix 1.7.2 on page 26 and in 1.4 on page 15.

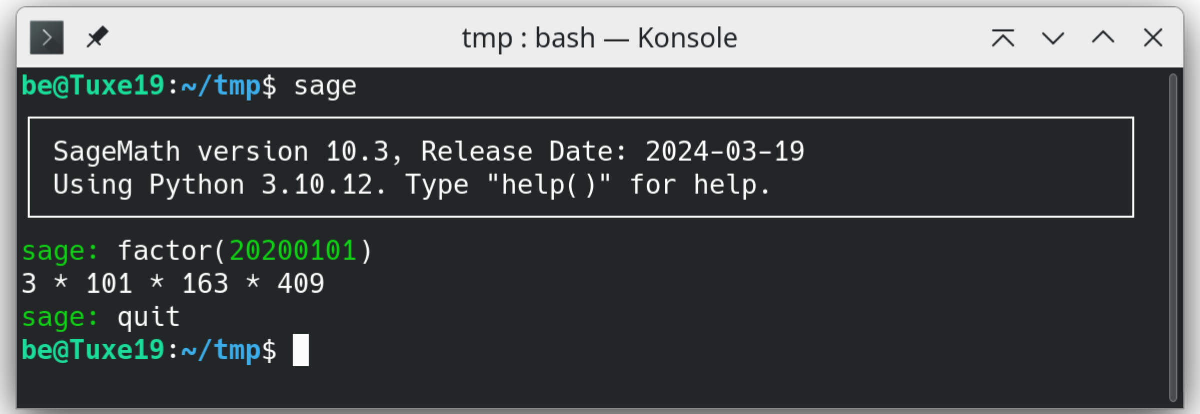
   The two most common ways to use a Jupyter notebook are:

   a) within a browser

   b) within Visual Studio Code (abbreviated as VS Code or VSC)

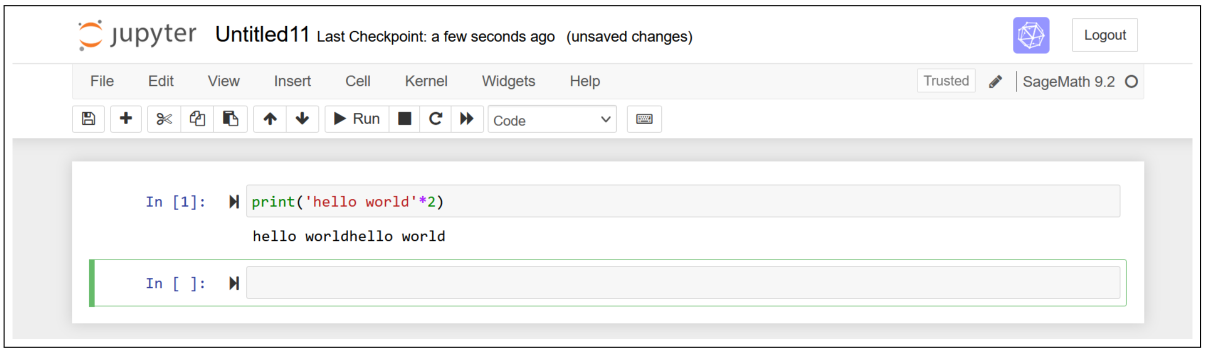
Remark 1: VS Code with a self-built SageMath installation

[8] See `https://doc.sagemath.org/html/en/installation/launching.html`.
[9] The differences are described in Section 5.12.5.4 of the CrypTool book.
[10] `https://doc.sagemath.org/html/en/tutorial/interactive_shell.html`



```
be@Tuxe19:~/tmp$ sage
┌────────────────────────────────────────────────────────────────────┐
│ SageMath version 10.3, Release Date: 2024-03-19                    │
│ Using Python 3.10.12. Type "help()" for help.                      │
└────────────────────────────────────────────────────────────────────┘
sage: factor(20200101)
3 * 101 * 163 * 409
sage: quit
be@Tuxe19:~/tmp$
```

Fig. 1: SageMath console from the command line (terminal)



```
In [1]:  print('hello world'*2)
         hello worldhello world

In [ ]:
```

Fig. 2: Creating a SageMath file with Jupyter in a browser

If you build your SageMath installation yourself, e. g. because the existing binaries in the distributions are not new enough[11] and still have an old SageMath version installed, it can happen that VS Code only displays the old Sage version as a potential kernel.

If the new kernel is not listed in VS Code, you can proceed as follows. Prerequisite to fix this: The kernel (Jupyter server) is started via `$ sage -n jupyter`. Click on the kernel icon (or click Ctrl+Shift+P and select "Select Kernel"). Then select one after the other:
`Select Another Kernel --> Existing Jupyter Server` and enter the URL of the running Jupyter server, e. g. `http://127.0.0.1:8888/tree?token=...` The current SageMath kernel (e. g. 10.3) is then also available.

Remark 2: Using SageMath functions within a Python file

If SageMath is installed locally, you can also use SageMath functions in a Python file. To do this, add `from sage.all import *` to the beginning of the Python file. However, the operators remain those of Python, so `**` causes exponentiation and `^` means XOR; while with SageMath `^` causes exponentiation. A minimal example (`test_sage-in-py-file.py`) can be found on the CrypTool website: `https://www.cryptool.org/en/ctbook/sagemath/`.

Remark 3: Possible future: SageMath running completely local in the browser

There are first attempts to create a WebAssembly-based version of SageMath so that SageMath can also run locally in the browser in the future (this mean, not only the frontend, but also the kernel): `https://github.com/sagemathinc/wasm-pari`.

1.2 Examples using the built-in mathematical functions in SageMath

Now that the basic options for installing and using SageMath have been shown, we will briefly describe what SageMath offers. SageMath examples 1.1 and 1.2 on the current page and on page 10 contain a few little examples – as an introduction to see what you can do with SageMath on the command line.[12]

SageMath example 1.1 contains calls for dealing with numbers (e. g. determining length), from number theory (e. g. finding prime numbers, coprimes and primitive roots), trigonometry and dealing with polynomials (analysis, symbolic expressions).

How SageMath treats symbolic expressions is interesting. If you define a mathematical function as a symbolic expression with symbolic variables, SageMath can distinguish between "arguments" as independent variables on the one hand, and parameters, that is, variables that are expected to have fixed values on the other hand.

SageMath Example 1.1: Small samples from different areas in mathematics (1)

```
print("\n# SageAppendix--SAMPLE 010: =========")

# Allgemein: Bestimmen der Länge der dezimalen und der binären Darstellung von 26!
# General: Determine the length of the decimal and the binary representation of 26!

print("14.digits():", 14.digits())  # Show the digits of an integer number

b=factorial(26)
print("Factorial b:", b)
print("Factorial b.n():", b.n())
print("Factorial b.n(prec=16):", b.n(prec=16))
print("Factorial b.ndigits():", b.ndigits())
print("Factorial b.ndigits(base=2):", b.ndigits(base=2))
print("Factorial b.nbits():", b.nbits())
```

[11] In May 2024, SageMath version 10.3 (the current release version then) was available on macOS via brew, but on Ubuntu the latest pre-built binary only had version 9.5. Also the Windows Subsystem for Linux WSL offered version 9.5 (the earlier pre-built binaries for Windows based on Cygwin are obsolete). See also `https://doc.sagemath.org/html/en/installation/`.

[12] The examples are mostly from a no more existing blog of Alasdair McAndrew.

ctd. SageMath Example 1.1

```
# Zahlentheorie / Number theory:
print("\n14.coprime_integers(max=16):", 14.coprime_integers(16))
i=randint(2^49,2^50); p=next_prime(i)
print("Next prime of %d: p=%d" % (i,p))
# p=1022095718672689 # for testing
r=primitive_root(p)
print("Primitive_root(p):", r)
pl=log(mod(10^15,p),r)
print("pl:", pl)
print("mod(r,p)^pl", mod(r,p)^pl)

# Trigonometrie (SageMaths Trigonometrie-Funktionen nutzen Bogenmaß statt Grad; pi=180°)
# Trigonometry (Sage's trigonometry functions use radians instead of degrees; pi=180°)
print("\nsin(pi/6):", sin(pi/6))  # 30° (Grad / degrees)

# Analysis (Infinitesimalrechnung) / Calculus // Symbolic expression:
# x=var('x')
var('a, x')  # symbolic variables (can be parameters or independent variables)
p1=a*exp(x^2)
print("\nExpression p1: ", p1)
p2=diff(p1,x,10) # compute tenth derivative
print("Expression p2: ", p2)
p(x)=diff(p1,x,10)*exp(-x^2)  # via "(x)" only x is considered independent variable
print("Expression p: ", p)
print("Polynom p simplified:", p.simplify_full())# now a polynomial since exp canceled out
print("Variables in p: {0};  Arguments in p: {1}".format(
    p.variables(), p.arguments()))  # an "argument" is an independent variable
print("Type of p: {0}".format(type(p)))

#------------------------------------
# SageAppendix--SAMPLE 010: =========
# 14.digits(): [4, 1]
# Factorial b: 403291461126605635584000000
# Factorial b.n(): 4.03291461126606e26
# Factorial b.n(prec=16): 4.033e26
# Factorial b.ndigits(): 27
# Factorial b.ndigits(base=2): 89
# Factorial b.nbits(): 89
#
# 14.coprime_integers(max=16): [1, 3, 5, 9, 11, 13, 15]
# Next prime of 816414536954577: p=816414536954609
# Primitive_root(p): 3
# pl: 171454160901578
# mod(r,p)^pl 183585463045391
#
# sin(pi/6): 1/2
#
# Expression p1:  a*e^(x^2)
# Expression p2:  1024*a*x^10*e^(x^2) + 23040*a*x^8*e^(x^2) + 161280*a*x^6*e^(x^2) + 403200*a*x^4*e^(x^2) +▶
  ▶ 302400*a*x^2*e^(x^2) + 30240*a*e^(x^2)
# Expression p:  x |--> 32*(32*a*x^10*e^(x^2) + 720*a*x^8*e^(x^2) + 5040*a*x^6*e^(x^2) + 12600*a*x^4*e^(x^2▶
  ▶) + 9450*a*x^2*e^(x^2) + 945*a*e^(x^2))*e^(-x^2)
# Polynom p simplified: 1024*a*x^10 + 23040*a*x^8 + 161280*a*x^6 + 403200*a*x^4 + 302400*a*x^2 + 30240*a
# Variables in p: (a, x);  Arguments in p: (x,)
# Type of p: <class 'sage.symbolic.expression.Expression'>
```

SageMath example 1.2 on the following page contains calls from linear algebra (LA) and functions for finite fields.

Vectors in linear algebra are sometimes represented as row vectors and sometimes as columns vectors. Both representations are equivalent and also both occur in mathematical literature. SageMath can handle both representations because it works with so-called “generic” vectors. If we define, say, the vector `v=vector([1,2])`

and the matrix `A=([[3,4],[5,6]])`, then we can multiply v with the matrix A from the right as well as from the left. The expression $A * v$ uses v as column vector while $v * A$ uses v as row vector. The vector just has to have the right length. To create an explicit row vector from a "generic" vector, just use the `row()` method. However, the result then is of the type `Matrix`. And the `transpose()` method can only be applied on the matrix type. Entering `v.transpose()` results in an error message – while entering `v.row().transpose()` works well.

SageMath Example 1.2: Small samples from different areas in mathematics (2)

```
print("\n# SageAppendix--SAMPLE 020: =========")

# Lineare Algebra / Linear algebra:

u = zero_vector(SR, 10)  # create a zeros vector (Nullvektor); SR = Symbolic Ring
print("Zero vector u:\n", u, sep="")

O = matrix.ones(SR, 2, 10)  # create 2 ones vectors. As there is no function like ones_vector(),
                            # define the vector as the first row of a matrix of ones, or
print("Ones matrix O:\n", O, sep="")

v = vector(SR, [1]*10)  # [1]*10 is the Python way to construct a list with 10 repeats of 1.
print("Ones vector v:\n", v, sep="")

M=matrix([[1,2,3],[4,5,6],[7,8,10]])
print("M echo:\n", M.echelon_form(), sep="")  # show the echelon basis matrix

c=random_matrix(ZZ,3,1)
print("Random vector c:\n", c, sep="")
print("Random vector c.transpose():\n", c.transpose(), sep="")
b=M*c
print("Matrix * vector: b = M*c:\n", b, sep="")
print("Using inverse matrix M^-1 * b:\n", M^-1*b, sep="")

A = matrix([[1,1,0],[0,2,0], [0,0,3],])  # 3*3
print("A:", type(A))
print("rows:", A.nrows()) # 3  # find out the number of rows of a matrix
print("cols:", A.ncols()) # 3  # find out the number of cols of a matrix
print("A echo:\n", A.echelon_form(), sep="")  # show the echelon basis matrix

v = A.column(1)  # (1, 2, 0)  # get the 2nd column (results in a 3*1 vector)
print("v:", type(v))
print("len(v):", v.length()) # Alternative: len(v)
print(v)

print("A*v:", A * v)  # (3, 4, 0) # results in a 3*1 vector
print("v*A:", v * A)  # (1, 5, 0) # results in a 1*3 vector

r = v.row()  # [1 2 0]   # REMARK: Vector.row() and .column() create a matrix type
print("r:", type(r))
print("r=v.row():", r)   # Remark: len(r) doesn't work, as r is a matrix (1*3)
c = r.transpose()  # [1] [2] [0]   # REMARK: r,c are type matrix (not vector).
print("c:", type(c))
print("c=transposed r:\n", c, sep='')

# vector multiplication
print("r*c:", r * c)  # [5]
print("c*r:\n", c * r, sep='')  # [1 2 0] [2 4 0] [0 0 0]

# multiplication of matrix and vector (explicit a row or col one)
# What does not work is:  A * r  and  c * A
print("r*A:", r * A)  # (1, 5, 0) # results in a 1*3 vector
print("A*c:\n", A * c, sep='')  # (3, 4, 0) # results in a 3*1 vector

# Create LaTeX command to typset a matrix equation
```

ctd. SageMath Example 1.2

```
# print('$$ A*c=b: %s * %s = %s $$'%(latex(A), latex(c), latex(A*c)))

# Endliche Körper (\url{http://de.wikipedia.org/wiki/Endlicher_K%C3%B6rper})
# Finite Fields (\url{http://en.wikipedia.org/wiki/Finite_field})
F.<x>=GF(2)[]
G.<a>=GF(2^4,name='a',modulus=x^4+x+1)
print("\nIn GF: a^2/(a^2+1): ", a^2/(a^2+1))
print("a^100:              ", a^100)
print("log(a^2,a^3+1):     ", log(a^2,a^3+1))
print("(a^3+1)^13:         ", (a^3+1)^13)

#------------------------------------
# SageAppendix--SAMPLE 020: =========
# Zero vector u:
# (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
# Ones matrix O:
# [1 1 1 1 1 1 1 1 1 1]
# [1 1 1 1 1 1 1 1 1 1]
# Ones vector v:
# (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
# M echo:
# [1 2 0]
# [0 3 0]
# [0 0 1]
# Random vector c:
# [-1]
# [ 0]
# [-5]
# Random vector c.transpose():
# [-1  0 -5]
# Matrix * vector: b = M*c:
# [-16]
# [-34]
# [-57]
# Using inverse matrix M^-1 * b:
# [-1]
# [ 0]
# [-5]
# A: <class 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
# rows: 3
# cols: 3
# A echo:
# [1 1 0]
# [0 2 0]
# [0 0 3]
# v: <class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
# len(v): 3
# (1, 2, 0)
# A*v: (3, 4, 0)
# v*A: (1, 5, 0)
# r: <class 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
# r=v.row(): [1 2 0]
# c: <class 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
# c=transposed r:
# [1]
# [2]
# [0]
# r*c: [5]
# c*r:
# [1 2 0]
# [2 4 0]
# [0 0 0]
# r*A: [1 5 0]
# A*c:
```

ctd. SageMath Example 1.2

```
# [3]
# [4]
# [0]
#
# In GF: a^2/(a^2+1):  a^3 + a
# a^100:               a^2 + a + 1
# log(a^2,a^3+1):      13
# (a^3+1)^13:          a^2
```

1.3 Getting help when using SageMath

Loading SageMath from the command line and entering `help()`, we get outputs like in SageMath example 1.3:

SageMath Example 1.3: SageMath help (Welcome)

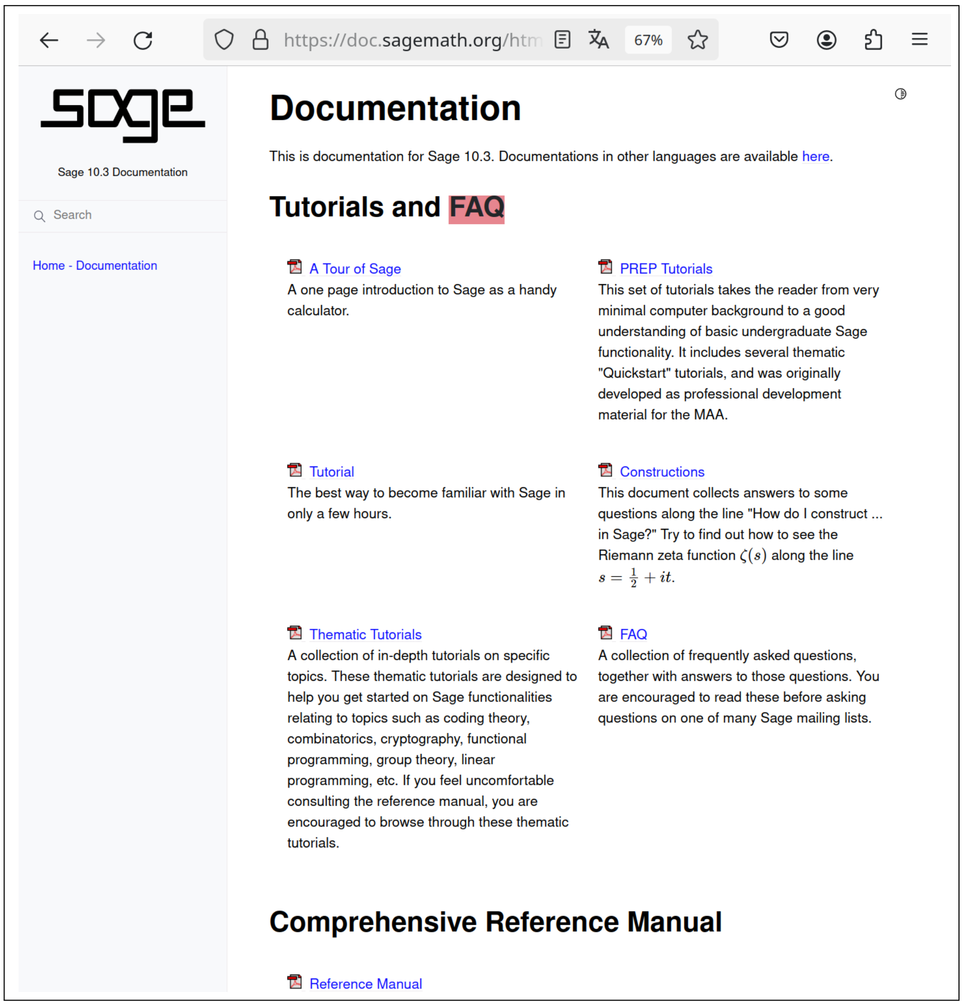
```
sage: help()
Welcome to Sage 10.3!

To view the Sage tutorial in your web browser, type "tutorial()", and
to view the (very detailed) Sage reference manual, type "manual()".
For help on any Sage function, for example "matrix_plot", type
"matrix_plot?" to see a help message, type "help(matrix_plot)" to see
a very similar message, type "browse_sage_doc(matrix_plot)" to view a
help message in a web browser, and type "matrix_plot??" to look at the
function's source code.

(When you type something like "matrix_plot?", "help(matrix_plot)", or
"matrix_plot??", Sage may start a paging program to display the
requested message. Type a space to scroll to the next page, type "h"
to get help on the paging program, and type "q" to quit it and return
to the "sage:" prompt.)

For license information for Sage and its components, read the file
"COPYING.txt" in the top-level directory of the Sage installation,
or type "license()".

To enter Python's interactive online help utility, type "python_help()".
To get help on a Python function, module or package, type "help(MODULE)" or
"python_help(MODULE)".
```

1.3.1 Getting help from websites

The official SageMath documentation is distributed with every release of SageMath (see Fig. 3). This includes the following documents:

- Tutorial – This tutorial is designed to help SageMath beginners become familiar with SageMath. It covers many features that beginners should be familiar with, and takes one to three hours to go through.
- Constructions – This document is in the style of a SageMath "cookbook". It is a collection of answers to questions about constructing various objects in SageMath.
- Developers' Guide – This guide is for developers who want to contribute to the development of SageMath. Among other issues, it covers coding style and conventions, modifying the core SageMath libraries, modifying the SageMath standard documentation, and code review and distribution.

- Reference Manual – This manual provides complete documentation on the major features of SageMath. The description of a class normally is accompanied by several code samples. All code samples in the reference manual are tested before a new SageMath version is released.
- Installation Guide – This guide explains how to install SageMath under various platforms.
- A Tour of Sage – This is a tour of SageMath that showcases various features of SageMath that are useful for beginners.
- Numerical Sage – This document introduces tools available under SageMath that are useful for numerical computation.
- Three Lectures about Explicit Methods in Number Theory Using Sage – This document is about using SageMath to perform computations in advanced number theory.



Documentation

This is documentation for Sage 10.3. Documentations in other languages are available here.

Tutorials and FAQ

A Tour of Sage
A one page introduction to Sage as a handy calculator.

PREP Tutorials
This set of tutorials takes the reader from very minimal computer background to a good understanding of basic undergraduate Sage functionality. It includes several thematic "Quickstart" tutorials, and was originally developed as professional development material for the MAA.

Tutorial
The best way to become familiar with Sage in only a few hours.

Constructions
This document collects answers to some questions along the line "How do I construct ... in Sage?" Try to find out how to see the Riemann zeta function $\zeta(s)$ along the line $s = \frac{1}{2} + it$.

Thematic Tutorials
A collection of in-depth tutorials on specific topics. These thematic tutorials are designed to help you get started on Sage functionalities relating to topics such as coding theory, combinatorics, cryptography, functional programming, group theory, linear programming, etc. If you feel uncomfortable consulting the reference manual, you are encouraged to browse through these thematic tutorials.

FAQ
A collection of frequently asked questions, together with answers to those questions. You are encouraged to read these before asking questions on one of many Sage mailing lists.

Comprehensive Reference Manual

Reference Manual

Fig. 3: Website for the SageMath standard documentation (`https://doc.sagemath.org/html/en/`)

Further assistance on specific problems is available

- in the archive of the `sage-support` mailing list at `https://groups.google.com/g/sage-support`
- at `https://ask.sagemath.org`
- at `https://math.stackexchange.com/questions/tagged/sagemath`
- at `https://doc.sagemath.org/html/en/reference/genindex.html`

1.3.2 Getting help via `help()`, `?`, `??`, or `search_src`

If we already know the exact name of a function, we can use `help` or the question mark "?" to obtain further information on that function. For example, the command `help(SubstitutionCryptosystem)` provides documentation on the according built-in class (see SageMath example 1.4). In both cases, inside the documentation mode you go to the next page by hitting the space bar, and you leave the documentation mode via "q". When using `help(fname)`, the listing is opened in a separate window, so the help listing does not clutter up your session, unlike the output of `fname?` sometimes does.

It is possible to place the question mark *in front of* the search like `?fname`. Besides, two question marks are also allowed like in `fname??` or `??fname`. In this case you also get the source code defining `fname`.

Finally, it is worth mentioning the various functions from here https://doc.sagemath.org/html/en/reference/misc/sage/misc/sagedoc.html, especially `search_src`, `my_getsource` or `search_def`. Those can be helpful if the regular help doesn't help.

SageMath Example 1.4: SageMath help (for single function)

```
sage: help(SubstitutionCryptosystem)

SubstitutionCryptosystem(...)
    Create a substitution cryptosystem.

    INPUT:
    - "S" - a string monoid over some alphabet

    OUTPUT:
    - A substitution cryptosystem over the alphabet "S".

    EXAMPLES::
        sage: M = AlphabeticStrings()
        sage: E = SubstitutionCryptosystem(M)
        sage: E
        Substitution cryptosystem on Free alphabetic string monoid on A-Z
        sage: K = M([ 25-i for i in range(26) ])
        sage: K
        ZYXWVUTSRQPONMLKJIHGFEDCBA
        sage: e = E(K)
        sage: m = M("THECATINTHEHAT")
        sage: e(m)
        GSVXZGRMGSVSZG

    TESTS::
        sage: M = AlphabeticStrings()
        sage: E = SubstitutionCryptosystem(M)
        sage: E == loads(dumps(E))
        True
```

1.3.3 Using tab completion

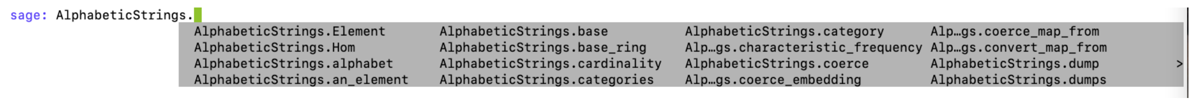
Here we present tab completion for searching two different topics: for finding the standard commands and for finding the methods of Sage objects.

Tab completion to find a command From within a SageMath session, we can obtain a list of (default) commands matching some pattern. To do so, we type the first few characters and then press the "Tab" key like in SageMath example 1.5 on the next page:

SageMath Example 1.5: SageMath help (expand with Tab key)

```
sage: Su[TAB]
   Subsets                  Subwords                 SuperPartitions
   SubstitutionCryptosystem Sudoku                   SupersingularModule
   SubwordComplex           SuperPartition           SuzukiGroup

sage: su[TAB]
        subfactorial     sum              supersingular_D
        subsets          sum_of_k_squares supersingular_j
        sudoku           super            surfaces
```

Tab completion to find all methods (member functions) of a Sage object To list all methods (member functions, attributes) of a Sage object `AA` just type `AA.`, then type the "Tab" key on your keyboard. A possible output is shown in Fig. 4.



```
sage: AlphabeticStrings.
                 AlphabeticStrings.Element      AlphabeticStrings.base         AlphabeticStrings.category          Alp…gs.coerce_map_from
                 AlphabeticStrings.Hom          AlphabeticStrings.base_ring    Alp…gs.characteristic_frequency     Alp…gs.convert_map_from
                 AlphabeticStrings.alphabet     AlphabeticStrings.cardinality  AlphabeticStrings.coerce            AlphabeticStrings.dump     >
                 AlphabeticStrings.an_element   AlphabeticStrings.categories   Alp…gs.coerce_embedding             AlphabeticStrings.dumps
```

Fig. 4: Output after dot with tab completion

List all methods of a Sage object via a program Instead of using the "Tab" key, you also can write a small program which prints all methods of a Sage object (here `AA`) – like in SageMath example 1.6.

SageMath Example 1.6: Program to print all member functions of a SageMath object

```
sage: AA = AlphabeticStrings()
sage: object_methods = [method_name for method_name in dir(AA)
....:                  if callable(getattr(AA, method_name))]; object_methods
['CartesianProduct',
 'Element',
 'Hom',
 '__bool__',
 ...
 'unrank',
 'unrank_range',
 'variable_name',
 'variable_names']
```

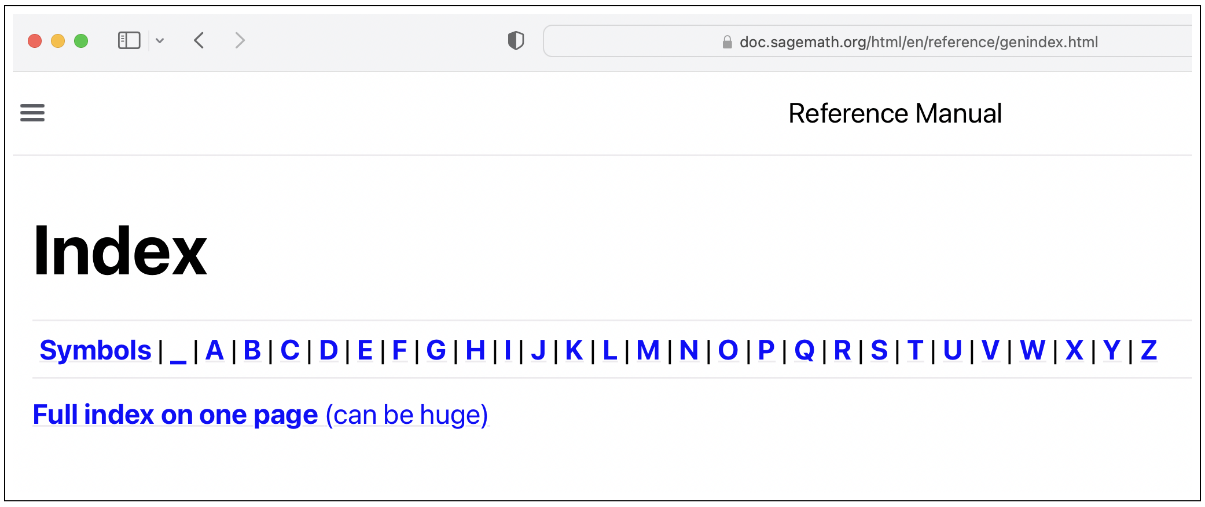
1.3.4 The comprehensive command list: The SageMath index

There is also a complete, alphabetically ordered list of the SageMath commands on the internet. Figure 5 shows this site, the so-called index. If you click on one of the letters at the top, all commands starting with that letter are shown and one can proceed searching like in Fig. 6. Alternatively, you can go to the full index, which is indeed quite huge.

1.4 Using the Jupyter notebook

So far we have shown the functions of SageMath in the terminal. Here we describe what a Jupyter notebook is and how to use it in the browser and in Visual Studio Code (VSC)..[13]

[13] If the Jupyter extension is installed, it is sufficient to load a file with the extension "ipynb" so that VS Code switches to the notebook view.


doc.sagemath.org/html/en/reference/genindex.html

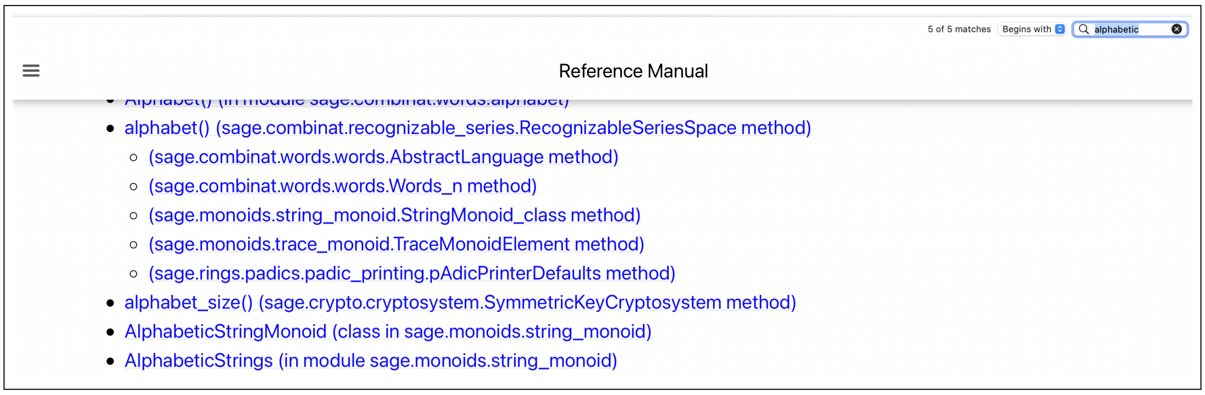
Reference Manual

Index

Symbols | _ | 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

Full index on one page (can be huge)

Fig. 5: Index


5 of 5 matches Begins with alphabetic

Reference Manual

- Alphabet() (in module sage.combinat.words.alphabet)
- alphabet() (sage.combinat.recognizable_series.RecognizableSeriesSpace method)
  - (sage.combinat.words.words.AbstractLanguage method)
  - (sage.combinat.words.words.Words_n method)
  - (sage.monoids.string_monoid.StringMonoid_class method)
  - (sage.monoids.trace_monoid.TraceMonoidElement method)
  - (sage.rings.padics.padic_printing.pAdicPrinterDefaults method)
- alphabet_size() (sage.crypto.cryptosystem.SymmetricKeyCryptosystem method)
- AlphabeticStringMonoid (class in sage.monoids.string_monoid)
- AlphabeticStrings (in module sage.monoids.string_monoid)

Fig. 6: Search among those commands that begin with the letter a.

A Jupyter notebook (formerly IPython Notebook) is a browser-based interactive computational environment for creating notebook documents. The file name usually ends with ".ipynb". IPython was originally developed as an enhanced Python terminal interpreter.[14]

Formally, a Jupyter notebook is a JSON file, with a specific structure containing an ordered list of cells – see Fig. 7.

There are three types of cells: code cells, markdown cells, and raw cells. Default initially is "Code". Raw cells are rarely used. Markdown cells (seldom called text cells) are used for documentation. For each cell, the cell type is displayed and can be changed there or via the menu.

Code cells consist of an input part and an output part. The output part (the cell's output) displays the result when the code in the cell's input is executed by the kernel (here SageMath).[15] More about the kernel in Section 1.5. So for example, a line of code that begins with a number sign (#) is formatted in italic green font, interpreted as a comment, and not executed by the notebook. The cell's output is shown directly below the code cell's input and can contain text and graphical output. The graphical output can also have interactive parts. See Figs. 8 and 9.

The editor for Jupyter notebooks has two modes: the edit mode and the command mode. You can use different keyboard shortcuts when in edit vs command mode. If you click inside a cell, you switch to edit mode.

When you run a cell, its code with all of its operations is executed. So, a cell can only be executed as a whole. To run it, click in the toolbar on "Run" or click the menu path "Cells > Run Cells". Alternatively, you can press **Ctrl+Enter**, which works in both, the edit and the command mode: Then the cell where the mouse pointer currently is located will be executed. This key combination is the preferred method for executing a cell compared to clicking the mouse:[16]

- **Ctrl+Enter** evaluates current cell and keeps focus in the current cell.
- **Shift+Enter** evaluates current cell and moves focus to the next cell.

Special key commands are available in the command mode. To enter command Mode press **Esc** or use the mouse to click outside a cell's editor area. Being in the command mode is indicated by a gray cell border with a blue left margin.

Some useful keyboard shortcuts in the command mode:

- **Enter+O** switch on/off whether the following output cell is shown or not.
- **a** insert a cell above the selected cell
- **b** insert a cell below the selected cell
- **d d** delete the selected cell(s)

When (re-)opening a Jupyter notebook (ipynb), it's recommended to first evaluate all cells by going to the first cell and click on "Execute cell and all below".

Remark: To initialize the variables of a cell there are 3 commands:

reset()
: Delete all user-defined variables and resets all global variables back to their default states.

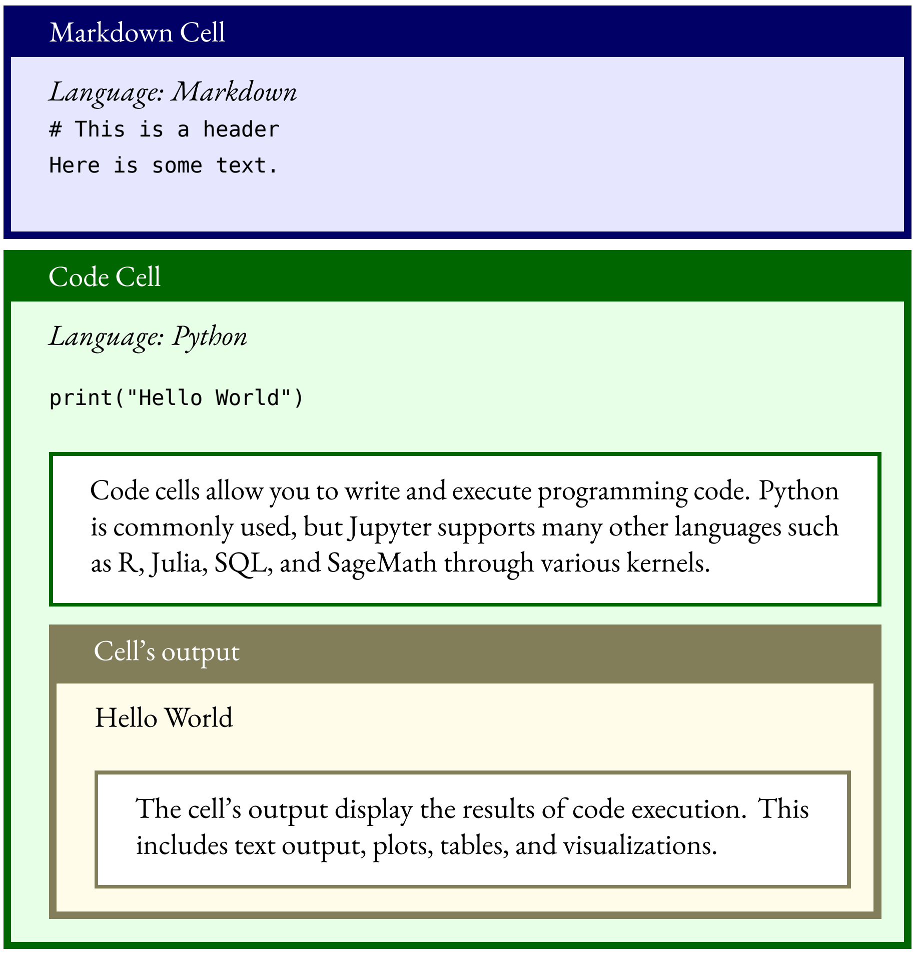
restore()
: Restore predefined global variables like QQ to their default values.

clear_vars()
: Delete all 1-letter symbolic variables predefined at startup of SageMath.

[14] `https://en.wikipedia.org/wiki/Project_Jupyter`, `https://jupyter.org/`, and `https://www.tutorialspoint.com/jupyter/ipython_introduction.htm`

[15] `https://jupyter-notebook.readthedocs.io/en/stable/notebook.html`

[16] There are many shortcuts making the usage easier. See `https://jupyter-tutorial.readthedocs.io/de/latest/workspace/jupyter/notebook/shortcuts.html`



Language: Markdown

```
# This is a header
Here is some text.
```

Language: Python

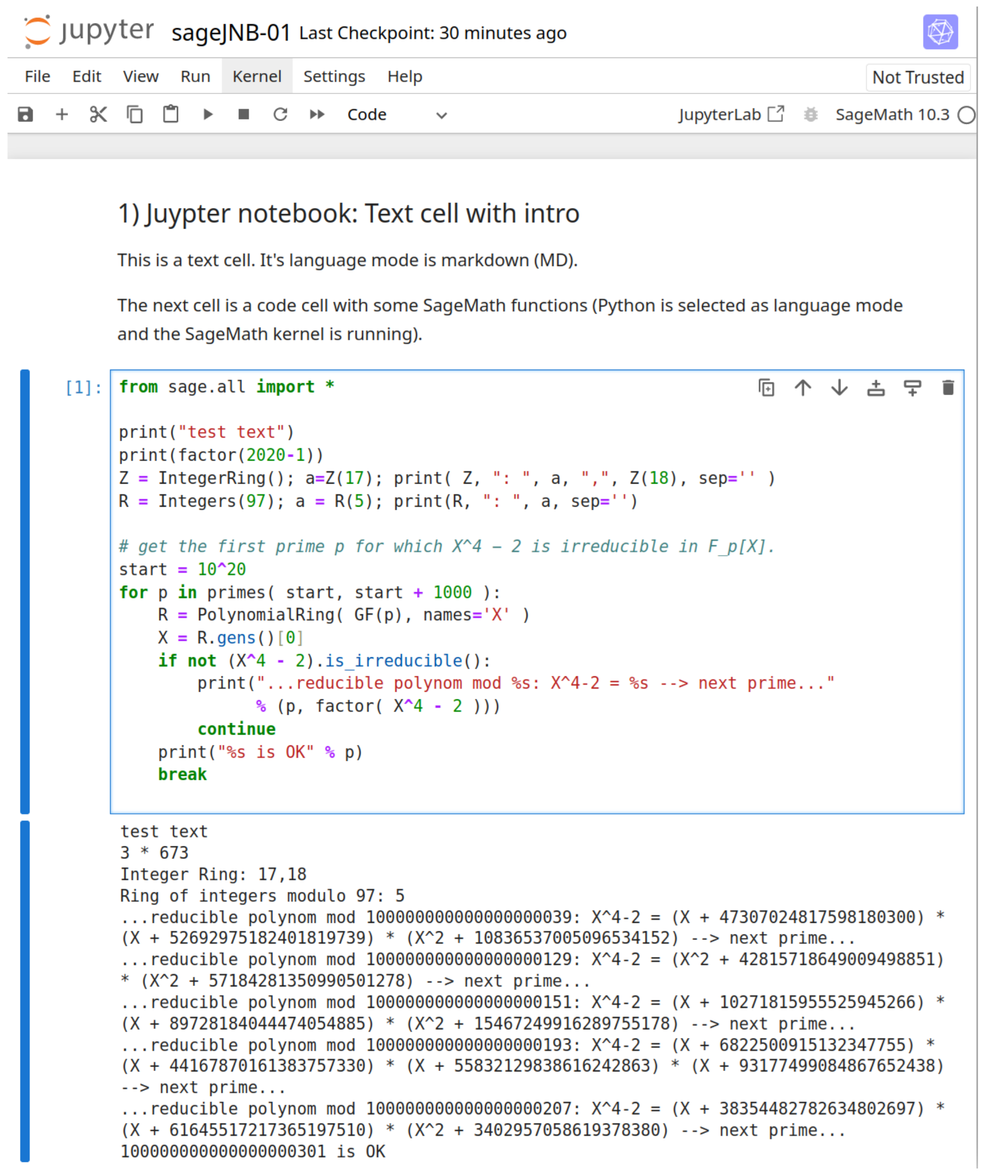
```
print("Hello World")
```

Code cells allow you to write and execute programming code. Python is commonly used, but Jupyter supports many other languages such as R, Julia, SQL, and SageMath through various kernels.

```
Hello World
```

The cell's output display the results of code execution. This includes text output, plots, tables, and visualizations.

Fig. 7: Typical generic sequence of cells within a Jupyter Notebook


jupyter sageJNB-01 Last Checkpoint: 30 minutes ago
File Edit View Run Kernel Settings Help
Not Trusted
Code
JupyterLab
SageMath 10.3

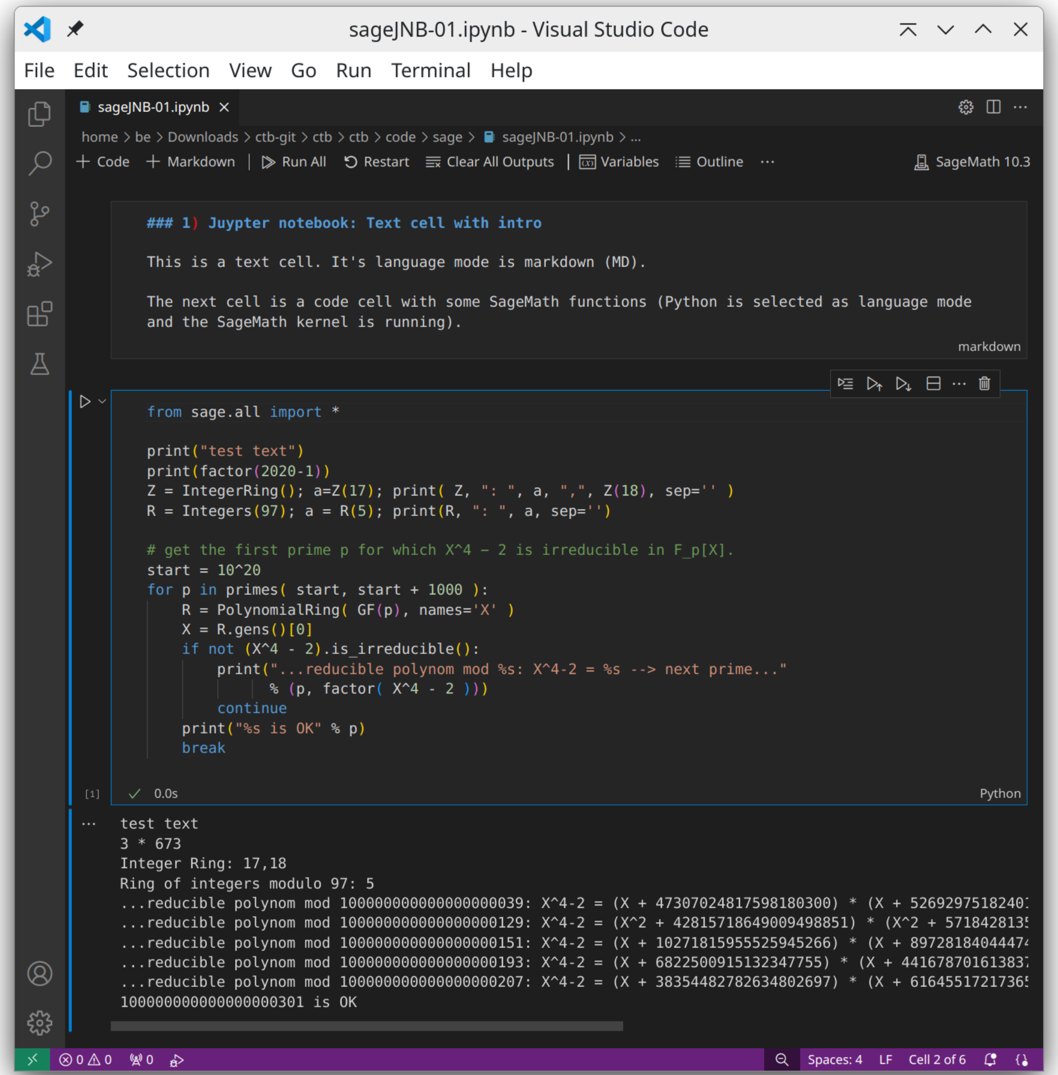
1) Juypter notebook: Text cell with intro

This is a text cell. It's language mode is markdown (MD).

The next cell is a code cell with some SageMath functions (Python is selected as language mode and the SageMath kernel is running).

```
[1]: from sage.all import *

     print("test text")
     print(factor(2020-1))
     Z = IntegerRing(); a=Z(17); print( Z, ": ", a, ",", Z(18), sep='' )
     R = Integers(97); a = R(5); print(R, ": ", a, sep='')

     # get the first prime p for which X^4 − 2 is irreducible in F_p[X].
     start = 10^20
     for p in primes( start, start + 1000 ):
         R = PolynomialRing( GF(p), names='X' )
         X = R.gens()[0]
         if not (X^4 - 2).is_irreducible():
             print("...reducible polynom mod %s: X^4-2 = %s --> next prime..."
                   % (p, factor( X^4 - 2 )))
             continue
         print("%s is OK" % p)
         break
```

```
test text
3 * 673
Integer Ring: 17,18
Ring of integers modulo 97: 5
...reducible polynom mod 100000000000000000039: X^4-2 = (X + 47307024817598180300) *
(X + 52692975182401819739) * (X^2 + 10836537005096534152) --> next prime...
...reducible polynom mod 100000000000000000129: X^4-2 = (X^2 + 42815718649009498851)
* (X^2 + 57184281350990501278) --> next prime...
...reducible polynom mod 100000000000000000151: X^4-2 = (X + 10271815955525945266) *
(X + 89728184044474054885) * (X^2 + 15467249916289755178) --> next prime...
...reducible polynom mod 100000000000000000193: X^4-2 = (X + 68225009151323477550) *
(X + 44167870161383757330) * (X + 55832129838616242863) * (X + 93177499084867652438)
--> next prime...
...reducible polynom mod 100000000000000000207: X^4-2 = (X + 38354482782634802697) *
(X + 61645517217365197510) * (X^2 + 34029570586193783800) --> next prime...
100000000000000000301 is OK
```

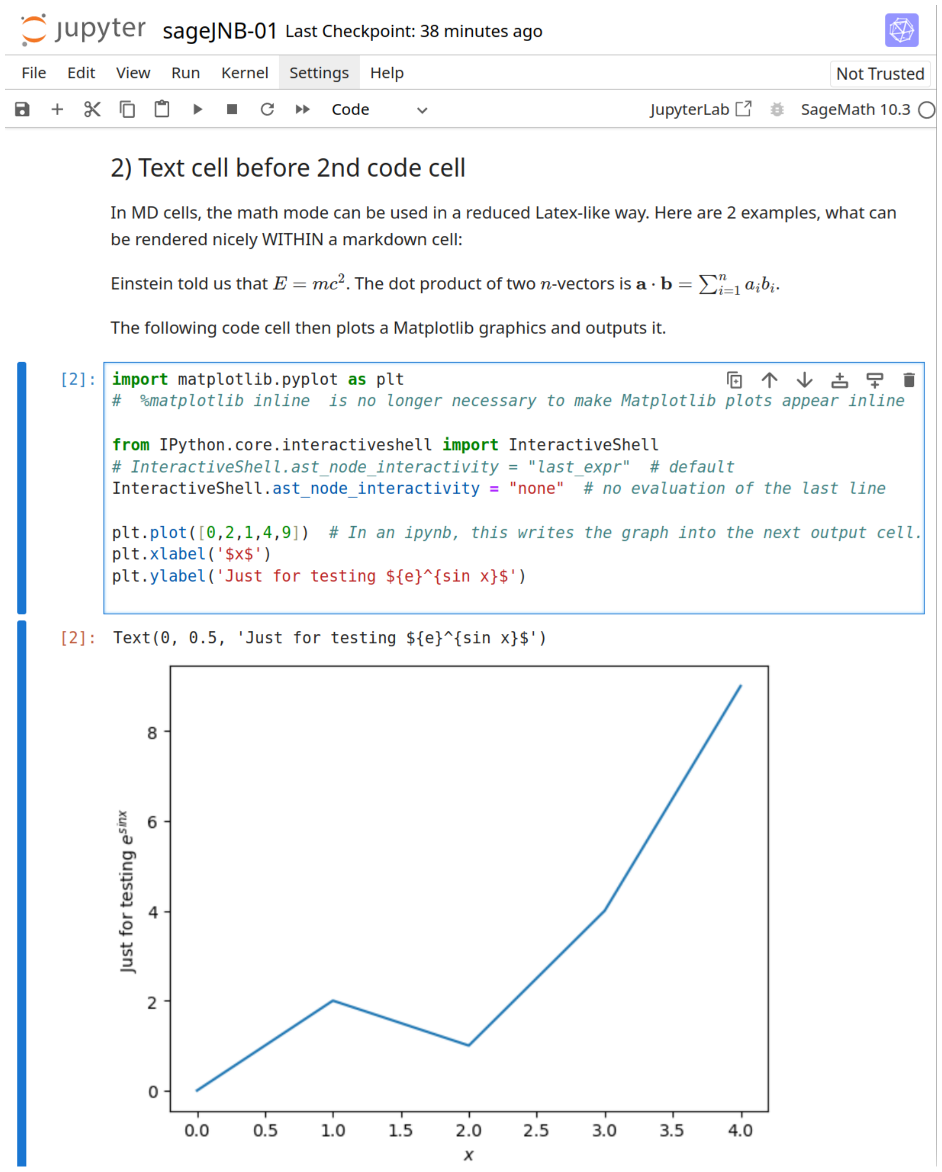

sageJNB-01.ipynb - Visual Studio Code
File Edit Selection View Go Run Terminal Help
sageJNB-01.ipynb
home > be > Downloads > ctb-git > ctb > ctb > code > sage > sageJNB-01.ipynb > ...
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ...
SageMath 10.3
markdown
Python
Spaces: 4 LF Cell 2 of 6

```
### 1) Juypter notebook: Text cell with intro

This is a text cell. It's language mode is markdown (MD).

The next cell is a code cell with some SageMath functions (Python is selected as language mode
and the SageMath kernel is running).
```

```
from sage.all import *

print("test text")
print(factor(2020-1))
Z = IntegerRing(); a=Z(17); print( Z, ": ", a, ",", Z(18), sep='' )
R = Integers(97); a = R(5); print(R, ": ", a, sep='')

# get the first prime p for which X^4 − 2 is irreducible in F_p[X].
start = 10^20
for p in primes( start, start + 1000 ):
    R = PolynomialRing( GF(p), names='X' )
    X = R.gens()[0]
    if not (X^4 - 2).is_irreducible():
        print("...reducible polynom mod %s: X^4-2 = %s --> next prime..."
              % (p, factor( X^4 - 2 )))
        continue
    print("%s is OK" % p)
    break
```

```
[1]  ✓ 0.0s

test text
3 * 673
Integer Ring: 17,18
Ring of integers modulo 97: 5
...reducible polynom mod 100000000000000000039: X^4-2 = (X + 47307024817598180300) * (X + 5269297518240
...reducible polynom mod 100000000000000000129: X^4-2 = (X^2 + 42815718649009498851) * (X^2 + 571842813
...reducible polynom mod 100000000000000000151: X^4-2 = (X + 10271815955525945266) * (X + 8972818404447
...reducible polynom mod 100000000000000000193: X^4-2 = (X + 68225009151323477550) * (X + 441678701613837
...reducible polynom mod 100000000000000000207: X^4-2 = (X + 38354482782634802697) * (X + 6164551721736
100000000000000000301 is OK
```

Fig. 8: Typical sequence of cells within a Jupyter Notebook for Sage code with text output within a browser and in VSC


jupyter sageJNB-01 Last Checkpoint: 38 minutes ago
File Edit View Run Kernel Settings Help
Not Trusted
Code
JupyterLab
SageMath 10.3

2) Text cell before 2nd code cell

In MD cells, the math mode can be used in a reduced Latex-like way. Here are 2 examples, what can be rendered nicely WITHIN a markdown cell:

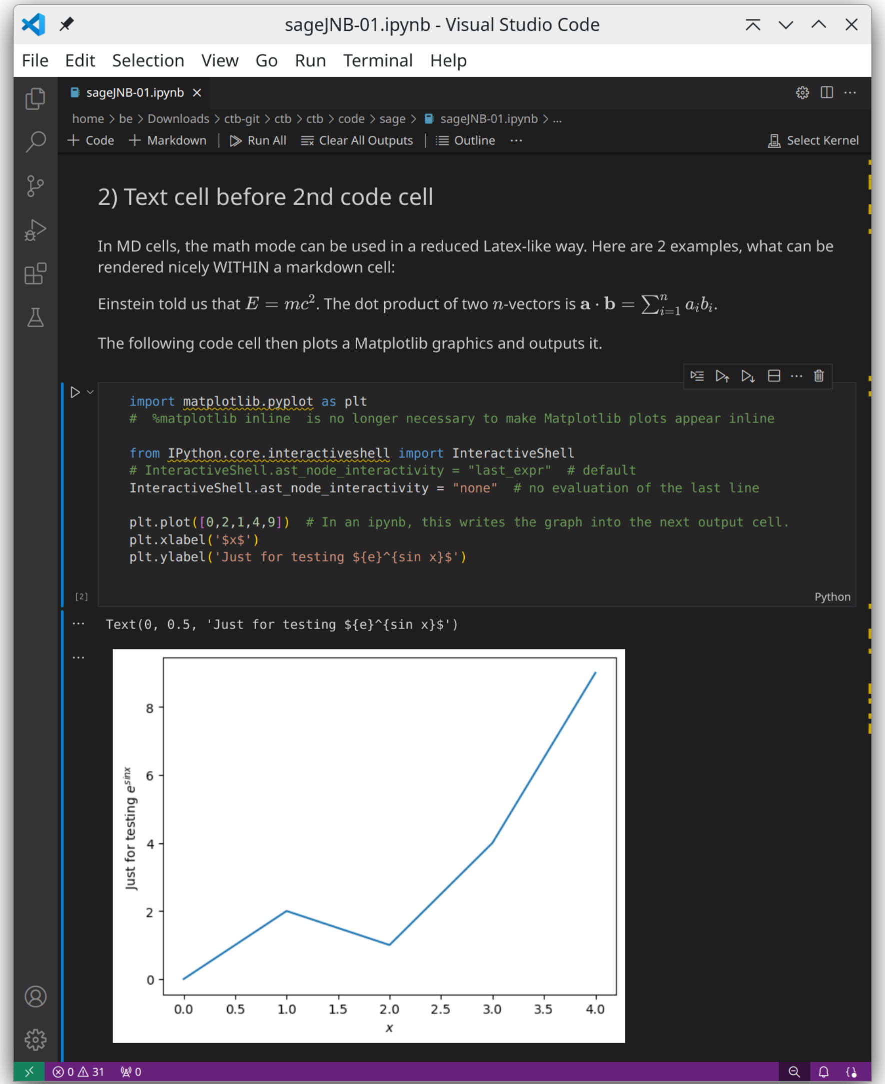
Einstein told us that $E = mc^2$. The dot product of two n-vectors is $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i$.

The following code cell then plots a Matplotlib graphics and outputs it.

```
[2]: import matplotlib.pyplot as plt
     #  %matplotlib inline  is no longer necessary to make Matplotlib plots appear inline

     from IPython.core.interactiveshell import InteractiveShell
     # InteractiveShell.ast_node_interactivity = "last_expr"  # default
     InteractiveShell.ast_node_interactivity = "none"  # no evaluation of the last line

     plt.plot([0,2,1,4,9])  # In an ipynb, this writes the graph into the next output cell.
     plt.xlabel('$x$')
     plt.ylabel('Just for testing ${e}^{sin x}$')
```

```
[2]: Text(0, 0.5, 'Just for testing ${e}^{sin x}$')
```

Just for testing e^{sinx}
x
0 2 4 6 8
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0


sageJNB-01.ipynb - Visual Studio Code
File Edit Selection View Go Run Terminal Help
sageJNB-01.ipynb
home > be > Downloads > ctb-git > ctb > ctb > code > sage > sageJNB-01.ipynb > ...
+ Code + Markdown | Run All | Clear All Outputs | Outline ...
Select Kernel
Python

2) Text cell before 2nd code cell

In MD cells, the math mode can be used in a reduced Latex-like way. Here are 2 examples, what can be rendered nicely WITHIN a markdown cell:

Einstein told us that $E = mc^2$. The dot product of two n-vectors is $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i$.

The following code cell then plots a Matplotlib graphics and outputs it.

```
import matplotlib.pyplot as plt
#  %matplotlib inline  is no longer necessary to make Matplotlib plots appear inline

from IPython.core.interactiveshell import InteractiveShell
# InteractiveShell.ast_node_interactivity = "last_expr"  # default
InteractiveShell.ast_node_interactivity = "none"  # no evaluation of the last line

plt.plot([0,2,1,4,9])  # In an ipynb, this writes the graph into the next output cell.
plt.xlabel('$x$')
plt.ylabel('Just for testing ${e}^{sin x}$')
```

[2]

```
Text(0, 0.5, 'Just for testing ${e}^{sin x}$')
```

Fig. 9: Typical sequence of cells within a Jupyter Notebook for Sage code with plot output within a browser and in VSC

1.5 The kernel of a Jupyter notebook

The Jupyter notebook, regardless of whether it is running in the browser or in VS Code, only displays the results of the so-called kernel, in this case the SageMath server. The kernel can be executed locally or on a remote server. More precisely, between the Jupyter notebook and the kernel there is the notebook server which can run either locally or remotely. In both cases, SageMath must be accessible to the Jupyter server as "kernel".[17] This is visualized in Fig. 10.

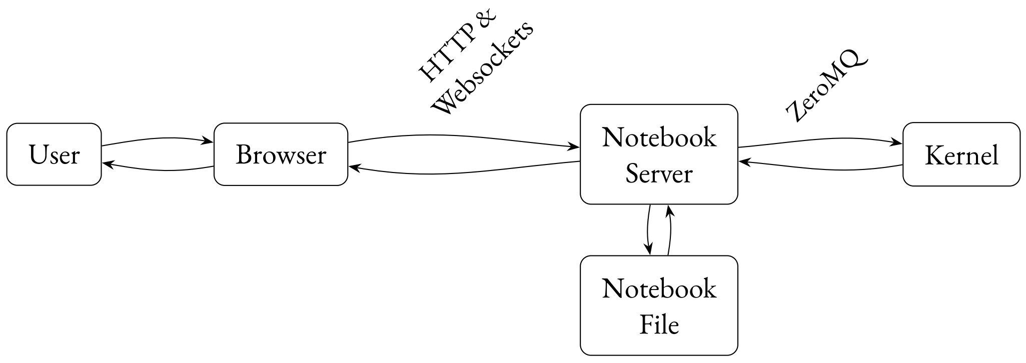


Fig. 10: Surroundings of a Jupyter Notebook[18]

As shown in Table 1 you can use SageMath remotely (without installing SageMath locally) :

`https://sagecell.sagemath.org/` or `https://cocalc.com/`

Two common methods to run the Jupyter server locally are:

- If you have installed Anaconda on your computer, first activate the SageMath environment and then in the same terminal start the Jupyter server, which then starts the specified notebook `Untitled.ipynb` in the browser:

  ```
  $ conda activate sage
  $ jupyter notebook Untitled.ipynb
  ```
  [19]

- Without Anaconda, but having a local SageMath installation, you can start the notebook under Linux normally this way from a terminal:

  ```
  $ sage -n jupyter
  ```
  or
  ```
  $ sage -n jupyterlab
  ```

Executing `$ sage -n jupyter` on the command line opens a browser window that shows all files in the current directory. If you double-click on an ipynb file, it then appears in another browser tab as a Jupyter notebook.

Sometimes the first click on the "Run" button results in the message "Error displaying widget: model not found". Pressing the button again usually solves the problem if the installation is correct.

[17] It may happen, that the notebook doesn't know a function like `interact`, then first check that it's a code cell, that SageMath is set as kernel, and then click on the icon "restart the kernel".

[18] With small changes taken from `https://mlsummit.ai/blog/jupyter-notebooks-fuer-lehre-und-entwicklung-alles-im-blick-notizbuch-fuer-entwickler/`

[19] In the browser you can then see as link something like `http://127.0.0.1:8888/notebooks/Untitled1.ipynb?kernel_name=sagemath`

1.6 Writing code with SageMath in the Sage console or via starting a Sage script in the terminal

When you start using a CAS (computer algebra system) you normally type in ready commands on the command line as in SageMath example 1.1 on page 8.[20]

If you develop your own functions, modify them and call them again, then it is much easier to do the development of the functions in an editor (easier compared to using the sage console only). For that three ways are common:

- Using the Jupyter notebook. This is discussed extensively in 1.8 and in 1.8.
- Executing of the edited script within the Sage console
- Executing of the edited script as Sage program from the terminal (e. g. in the bash shell)

The last two cases are discussed further within this subsection.

All three ways to develop code were applied in the CrypTool book Chapters 1.12 ("Appendix: Educational examples for symmetric ciphers using SageMath"), 2.8 ("Appendix: Examples using SageMath"), 4.15 ("Appendix: Examples using SageMath"), 5.20 ("Examples using SageMath"), and 9.4 ("Appendix: Boolean maps in SageMath").

To test and load the SageMath code stored in program file, there are two useful commands: `load()` and `attach()`.[21]

Suppose you have a function definition like this which has been saved to the file `primroots.sage`:

SageMath Example 1.7: Function definition in file with extension `.sage`

```
def somefunction(var1): # inside file with name e.g. primroots.sage
    r"""
    DocText.
    """
    ...
    return (L)
```

Loading The command `load()` can be used in all three cases: Sage console, Sage program, and Jupyter notebook – see Table 1 on page 6.

Within SageMath example 1.8, the above function from SageMath example 1.7 is loaded into SageMath with the command `load()`.

SageMath Example 1.8: load

```
sage: load("primroots.sage")
```

Then you can proceed to use on the command line any variable or function defined in that SageMath script `primroots.sage`.[22]

[20]When presenting code for the Sage console lines start with "sage:" and "...":

```
sage: m = 11
sage: for a in range(1, m):
....:     print( [power_mod(a, i, m) for i in range(1, m)] )
....:
```

Code in SageMath scripts and from Jupyter cells are presented in the same way as they appear there.

[21]See SageMath tutorial about programming, chapter "Loading and Attaching Sage files", https://doc.sagemath.org/html/en/tutorial/programming.html

[22]Notes/hints concerning SageMath scripts:

First, the `load` command executes all commands, regardless of whether it is called on the Sage command line or in a Sage script. After that the functions from the file loaded with `load` are still available. Unfortunately, unlike in Python, it is not possible to make just the functions of a SageMath file known. This functionality was implemented with the own function `my_import` (see Item (c) in Section 5.20.4 of the CrypTool book).

Attaching The command `attach()` can only be used on the Sage console (directly or within of another Sage script loaded from there), but it is not available in a Sage program called in a terminal or in a Jupyter notebook (there you have to use `load()`).

Normally we want to edit our own SageMath script and reload the content of the changed script into SageMath again. In order to automatically reload the file after every change, the command `attach()` is sufficient:[23]

SageMath Example 1.9: attach

```
sage: attach("primroots.sage")
```

Now edit and save again the SageMath script in a text editor, but don't exit SageMath. The changed function definition is reloaded into the running SageMath session after the next time you press Enter (and a syntax check is done at once). You can think of the command `attach()` as a way of telling SageMath to watch for all changes to a file, and reloading the file again once SageMath notices that there have been changes.

Figure 11 on the next page shows SageMath code in the editor VS Code with activated Python code highlighting. You can use other editors like gVim or Notepad++ just as well.

If you prefer to see the output of an attached file as if you entered the commands on the command line directly (not only what is shown via `print()`) then you could use the command `iload()`: Each line is loaded one at a time. To load the next line, you have to press the `Enter` key. You have to repeatedly press the `Enter` key until all lines of the SageMath script are loaded into the SageMath session.

SageMath Example 1.10: iload

```
sage: iload("primroots.sage")
```

1.7 SageMath and LaTeX

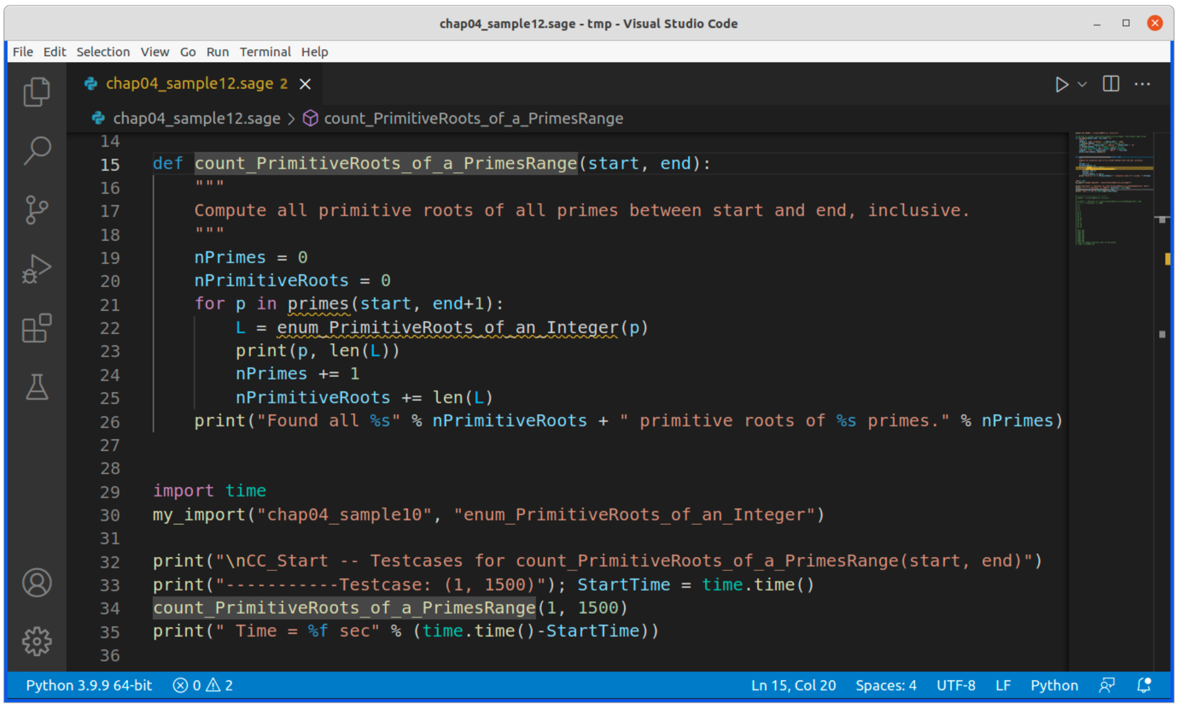
"Sage and the LaTeX dialect of TeX have an intensely synergistic relationship."[25] We will explain the relations most important for us using Table 2.

- Don't use white spaces in your file name.
- It's recommended that your SageMath script has the file extension ".sage" instead of ".py". For a SageMath script with the file extension ".sage" the default SageMath environment is also loaded when loading the file, and the syntax is checked.
- Instead of loading a script file from the SageMath prompt you can directly load it when starting SageMath; e. g. from a bash shell using `$ sage primroots.sage`. Here also, a syntax check is performed instantly and, if this check was successful, the script is executed.
- If you load your script as above, then SageMath first parses your script and copies it to another file with the extension ".py". SageMath then adds all necessary variables within "primroots.py" as well as all necessary import statements. That way, your SageMath script is executed as if you had entered the commands in your script directly to the SageMath command line.
- An important difference between commands within a script file and the commands entered manually behind the SageMath prompt is that all outputs need a `print()` statement. For example, instead of "m=11; m" you have to enter "m=11; print(m)".

[23] `attach()` can be applied directly after loading the script, even before having changed the script; and you can even omit `load()` and just call `attach()`, as `load()` is contained in `attach()`.

[24] The source code used in this editor can be found in SageMath example 5.20.11 of the CrypTool book. and in the file "chap05_sample120.sage" .

[25] Quote from the chapter "Sage, LaTeX and Friends" by Rob Beezer (2010-05-23), https://doc.sagemath.org/html/en/tutorial/latex.html


chap04_sample12.sage - tmp - Visual Studio Code
File Edit Selection View Go Run Terminal Help
chap04_sample12.sage 2
chap04_sample12.sage > count_PrimitiveRoots_of_a_PrimesRange
Python 3.9.9 64-bit ⊗ 0 ⚠ 2
Ln 15, Col 20 Spaces: 4 UTF-8 LF Python

```

def count_PrimitiveRoots_of_a_PrimesRange(start, end):
    """
    Compute all primitive roots of all primes between start and end, inclusive.
    """
    nPrimes = 0
    nPrimitiveRoots = 0
    for p in primes(start, end+1):
        L = enum_PrimitiveRoots_of_an_Integer(p)
        print(p, len(L))
        nPrimes += 1
        nPrimitiveRoots += len(L)
    print("Found all %s" % nPrimitiveRoots + " primitive roots of %s primes." % nPrimes)


import time
my_import("chap04_sample10", "enum_PrimitiveRoots_of_an_Integer")

print("\nCC_Start -- Testcases for count_PrimitiveRoots_of_a_PrimesRange(start, end)")
print("----------Testcase: (1, 1500)"); StartTime = time.time()
count_PrimitiveRoots_of_a_PrimesRange(1, 1500)
print(" Time = %f sec" % (time.time()-StartTime))

```

Fig. 11: SageMath example shown in an editor with code highlighting[24]

With SageMath	Sage console Sage program	latex()
	Jupyter notebook	Markdown cell: `$...$`

		Code cell:
		`latex(<<Latex code>>)`
		`Latex(r"""<<Latex code>>""")`
		`view oder show(<<Latex code>>)`

Within a LaTeX document	Use result of latex() from SageMath	e. g. `$ \frac{1}{5} \, z^{5} $`
	Use math mode and SageTeX macros	Call sequence: pdflatex → sage → pdflatex (see Fig. 18)

Tab. 2: Overview about the interconnection between SageMath and LaTeX

Remark: LaTeX separates the tasks of typesetting mathematics and typesetting normal text. This is achieved by the use of two operating modes, text and math mode. Text mode is the default mode for the document environment and does not need to be called explicitly. In LaTeX all mathematics needs to be inside some maths mode construction and many symbols, including `\circ` or `\sum_{n=1}^\infty`, are defined to only work in maths mode. The most common way to enter math mode is `$ ... $`, where the text within the dollar signs is in the math mode environment. Two further ways to enter math mode are `\begin{equation} ... \end{equation}` and `\[ ... \]` (also called double dollar or display style and used for offset formulas). Math mode ignores whitespace. When we wish to include text within mathematics, we must tell LaTeX that we are writing text, otherwise it will assume the word is actually a sequence of symbols and they are displayed in italics. There are two approaches to correctly formatting such text, the first using the `\mbox` command, and the second using the `\text{}` command which requires the `amsmath` package.

To get the LaTeX code of a Sage expression within SageMath code, we normally use the `latex()` function. This function expects the input to be in math mode like within `$ ... $`. Please note, that then the normal LaTeX text macros like `\texttt{}` don't work.

1.7.1 LaTeX and SageMath on the console

From within SageMath, the simplest way to exploit SageMath's support of LaTeX is to use the latex() function. The created strings can then be incorporated directly into standalone LaTeX documents. This works the same in a code cell of the Jupyter notebook and at the Sage command line. With `latex()` SageMath objects can be output (typesetting) in usual mathematics form in two ways: We can pass them to the command `latex()`, alternatively some SageMath objects provide the SageMath description by themselves when calling their own latex() method. The created text form (LaTeX code) can then be transformed via LaTeX display software in the usual mathematical form.

In other words: You pass a SageMath expression to the function latex() to get a LaTeX expression. So SageMath converts the LaTeX-relevant components directly into LaTeX syntax, and you can embed them 1:1 into a LaTeX document.

Example 1: Calling the command `latex(y)` (or sometimes `y.latex()`):[26]

```
sage: z=var('z'); latex(z^12)
z^{12}
sage: latex(integrate(z^4, z))
\frac{1}{5} \, z^{5}
sage: latex('This is a string')
\text{\texttt{This{ }is{ }a{ }string}}
sage: latex(QQ)
\Bold{Q}
```

Example 2: Every Object in SageMath has a LaTeXrepresentation. Calling `latex()` on `$y = (a*x^2)/5$` will result in an error, because an assignment (the term on the right side of $=$ is assigned to y) is not an object in SageMath. The right side alone is an object, and therefore we can call `latex()` on it:

```
sage: var('a,x,y')
(a, x, y)

sage: latex(y=(a*x^2)/5)
TypeError                                 Traceback (most recent call last)
<ipython-input-4-23375ccb8aef> in <module>
----> 1 latex(y=(a*x**Integer(2))/Integer(5))
TypeError: LatexCall.__call__() got an unexpected keyword argument 'y'

sage: latex((a*x^2)/5)
\frac{1}{5} \, a x^{2}

sage: y=(a*x^2)/5

sage: latex(y)
\frac{1}{5} \, a x^{2}
```

[26]The command `latex()` is not completely flawless. For instance, `\Bold` is not available in normal LaTeX, it is defined via `\newcommand{\Bold}` in `sage.misc.latex_macros`. So only use `\Bold` interactively or use it in a TeX file only after inserting the `\newcommand` definition. Easier would be `$\mathbb{Q}$` for $\mathbb{Q}$, where `\mathbb` requires the package `amsfonts`.

Example 3: The last command in the linear algebra part of SageMath example 1.2 on page 10 also contains `latex()` calls to **automatically** generate the code for a matrix equation.
See the following line commented out there:
```
# print('$$ A*c=b: %s * %s = %s $$'%(latex(A), latex(c), latex(A*c)))
```
This is the result:

$$A * c = b: \quad \begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix}$$

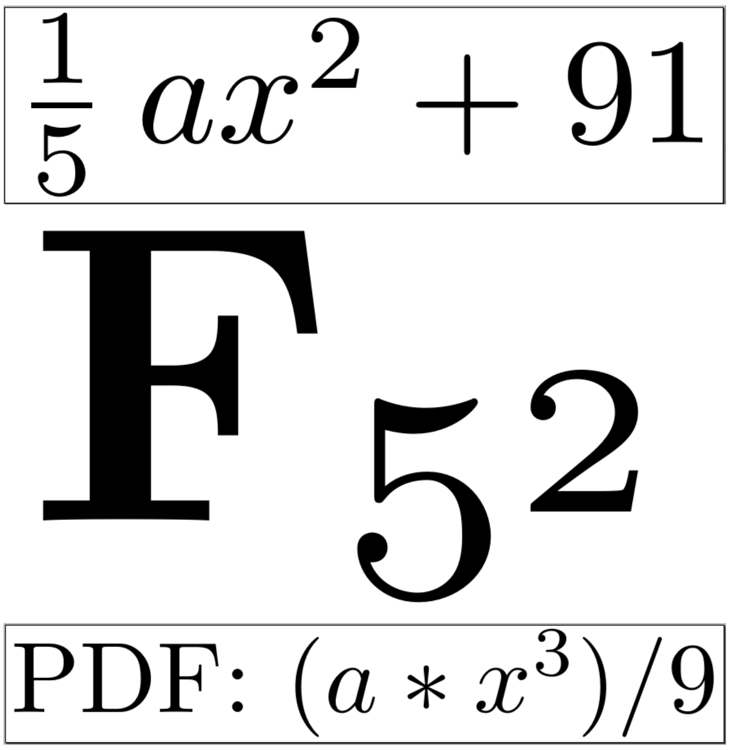
1.7.2 LaTeX and SageMath within a Jupyter notebook

The Figs. 8 and 9 on page 19 and on page 20 show the first two parts of the notebook `sageJNB-01.ipynb`. Each of these two parts contains a Markdown cell and a code cell (including its output). These two code cells with Sage code do not yet contain any “LaTeX commands”.

The third part of this notebook contains several variants of different “LaTeX commands” in the code cell – see Fig. 13 on the next page.

The Sage expressions rendered with `latex()` are output via `print()`, `show()`, and `view()` The differences can be seen in the cell’s output. You can also call `show()` without first calling `latex()` and receive output in the cell’s output as in a LaTeX document.

The three individual windows shown in Fig. 12 are created with the `view` command.

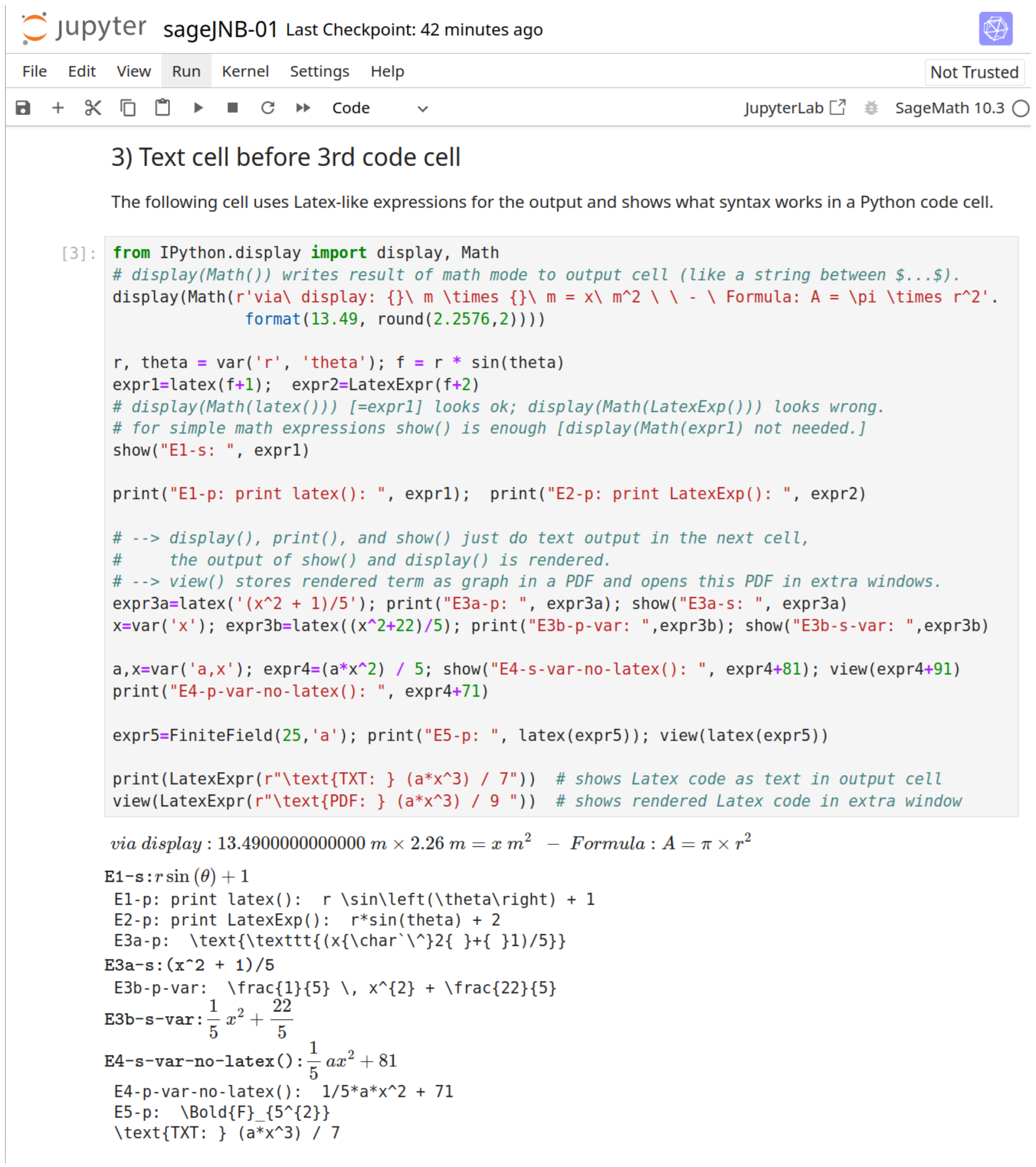


$$\frac{1}{5}\, a x^{2} + 91$$

$$\mathbf{F}_{5^2}$$

$$\text{PDF: } (a * x^3)/9$$

Fig. 12: Plot output from code in Fig. 13 in three extra windows


jupyter sageJNB-01 Last Checkpoint: 42 minutes ago
File Edit View Run Kernel Settings Help
Not Trusted
Code
JupyterLab
SageMath 10.3
[3]:

3) Text cell before 3rd code cell

The following cell uses Latex-like expressions for the output and shows what syntax works in a Python code cell.

```
from IPython.display import display, Math
# display(Math()) writes result of math mode to output cell (like a string between $...$).
display(Math(r'via\ display: {}\ m \times {}\ m = x\ m^2 \ \ - \ Formula: A = \pi \times r^2'.
           format(13.49, round(2.2576,2))))

r, theta = var('r', 'theta'); f = r * sin(theta)
expr1=latex(f+1);  expr2=LatexExpr(f+2)
# display(Math(latex())) [=expr1] looks ok; display(Math(LatexExp())) looks wrong.
# for simple math expressions show() is enough [display(Math(expr1) not needed.]
show("E1-s: ", expr1)

print("E1-p: print latex(): ", expr1);  print("E2-p: print LatexExp(): ", expr2)

# --> display(), print(), and show() just do text output in the next cell,
#     the output of show() and display() is rendered.
# --> view() stores rendered term as graph in a PDF and opens this PDF in extra windows.
expr3a=latex('(x^2 + 1)/5'); print("E3a-p: ", expr3a); show("E3a-s: ", expr3a)
x=var('x'); expr3b=latex((x^2+22)/5); print("E3b-p-var: ",expr3b); show("E3b-s-var: ",expr3b)

a,x=var('a,x'); expr4=(a*x^2) / 5; show("E4-s-var-no-latex(): ", expr4+81); view(expr4+91)
print("E4-p-var-no-latex(): ", expr4+71)

expr5=FiniteField(25,'a'); print("E5-p: ", latex(expr5)); view(latex(expr5))

print(LatexExpr(r"\text{TXT: } (a*x^3) / 7"))  # shows Latex code as text in output cell
view(LatexExpr(r"\text{PDF: } (a*x^3) / 9 "))  # shows rendered Latex code in extra window
```

$via\ display: 13.4900000000000\ m \times 2.26\ m = x\ m^2\ \ -\ Formula: A = \pi \times r^2$

E1-s: $r \sin\left(\theta\right) + 1$

```
 E1-p: print latex():  r \sin\left(\theta\right) + 1
 E2-p: print LatexExp():  r*sin(theta) + 2
 E3a-p:  \text{\texttt{(x{\char`\^}2{ }+{ }1)/5}}
```

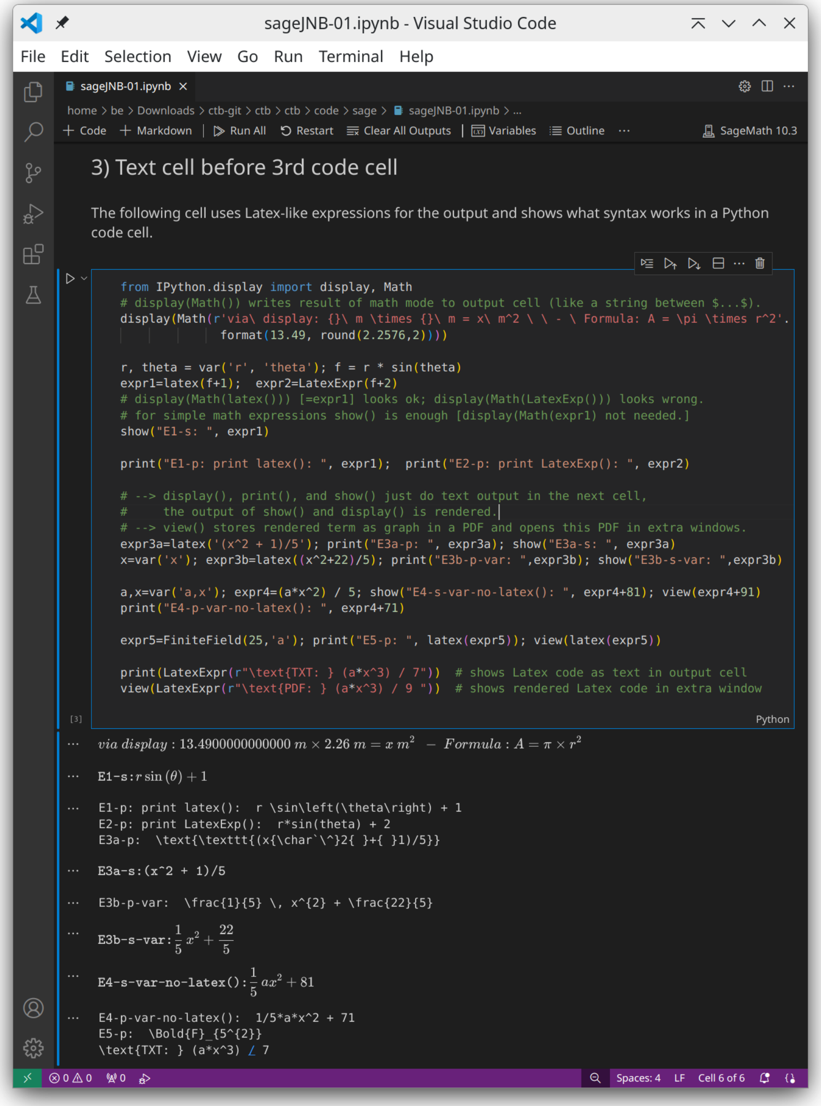
E3a-s: (x^2 + 1)/5

```
 E3b-p-var:  \frac{1}{5} \, x^{2} + \frac{22}{5}
```

E3b-s-var: $\frac{1}{5}\,x^2 + \frac{22}{5}$

E4-s-var-no-latex(): $\frac{1}{5}\,ax^2 + 81$

```
 E4-p-var-no-latex():  1/5*a*x^2 + 71
 E5-p:  \Bold{F}_{5^{2}}
 \text{TXT: } (a*x^3) / 7
```


sageJNB-01.ipynb - Visual Studio Code
File Edit Selection View Go Run Terminal Help
sageJNB-01.ipynb
home > be > Downloads > ctb-git > ctb > ctb > code > sage > sageJNB-01.ipynb > ...
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ...
SageMath 10.3
[3]
Python
Spaces: 4 LF Cell 6 of 6

3) Text cell before 3rd code cell

The following cell uses Latex-like expressions for the output and shows what syntax works in a Python code cell.

```
from IPython.display import display, Math
# display(Math()) writes result of math mode to output cell (like a string between $...$).
display(Math(r'via\ display: {}\ m \times {}\ m = x\ m^2 \ \ - \ Formula: A = \pi \times r^2'.
           format(13.49, round(2.2576,2))))

r, theta = var('r', 'theta'); f = r * sin(theta)
expr1=latex(f+1);  expr2=LatexExpr(f+2)
# display(Math(latex())) [=expr1] looks ok; display(Math(LatexExp())) looks wrong.
# for simple math expressions show() is enough [display(Math(expr1) not needed.]
show("E1-s: ", expr1)

print("E1-p: print latex(): ", expr1);  print("E2-p: print LatexExp(): ", expr2)

# --> display(), print(), and show() just do text output in the next cell,
#     the output of show() and display() is rendered.
# --> view() stores rendered term as graph in a PDF and opens this PDF in extra windows.
expr3a=latex('(x^2 + 1)/5'); print("E3a-p: ", expr3a); show("E3a-s: ", expr3a)
x=var('x'); expr3b=latex((x^2+22)/5); print("E3b-p-var: ",expr3b); show("E3b-s-var: ",expr3b)

a,x=var('a,x'); expr4=(a*x^2) / 5; show("E4-s-var-no-latex(): ", expr4+81); view(expr4+91)
print("E4-p-var-no-latex(): ", expr4+71)

expr5=FiniteField(25,'a'); print("E5-p: ", latex(expr5)); view(latex(expr5))

print(LatexExpr(r"\text{TXT: } (a*x^3) / 7"))  # shows Latex code as text in output cell
view(LatexExpr(r"\text{PDF: } (a*x^3) / 9 "))  # shows rendered Latex code in extra window
```

$via\ display: 13.4900000000000\ m \times 2.26\ m = x\ m^2\ \ -\ Formula: A = \pi \times r^2$

E1-s: $r \sin\left(\theta\right) + 1$

```
E1-p: print latex():  r \sin\left(\theta\right) + 1
E2-p: print LatexExp():  r*sin(theta) + 2
E3a-p:  \text{\texttt{(x{\char`\^}2{ }+{ }1)/5}}
```

E3a-s: (x^2 + 1)/5

```
E3b-p-var:  \frac{1}{5} \, x^{2} + \frac{22}{5}
```

E3b-s-var: $\frac{1}{5}\,x^2 + \frac{22}{5}$

E4-s-var-no-latex(): $\frac{1}{5}\,ax^2 + 81$

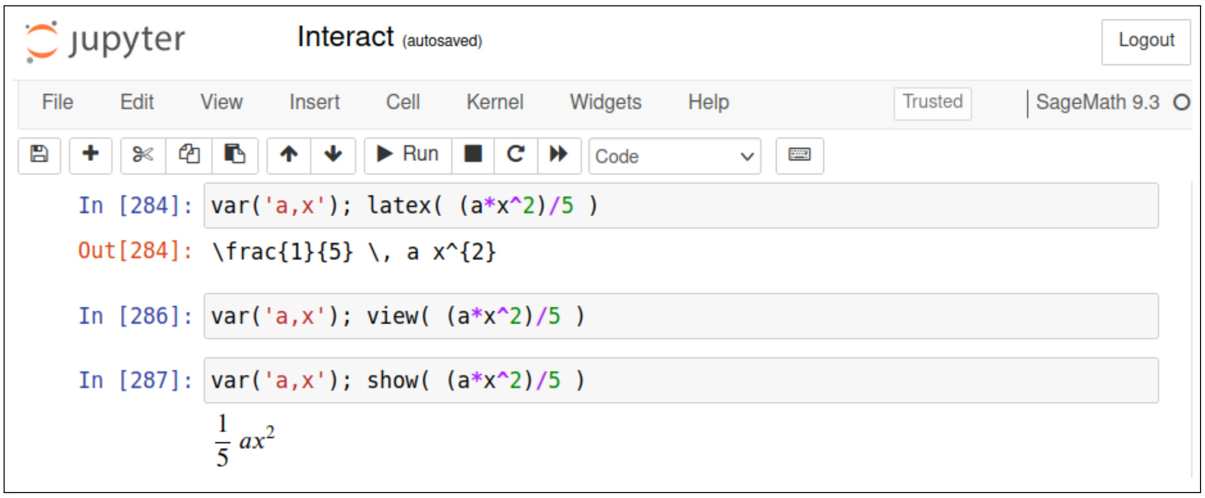
```
E4-p-var-no-latex():  1/5*a*x^2 + 71
E5-p:  \Bold{F}_{5^{2}}
\text{TXT: } (a*x^3) / 7
```

Fig. 13: Typical sequence of cells within a Jupyter Notebook for code Sage with LaTeX axpressions within a browser and in VSC

There are roughly three different ways of using LaTeX in a Jupyter notebook that execute SageMath code.

a) latex() The `latex()` command offers the same options here as on the Sage console – see Subsection 1.7.1 and Fig. 14.

b) view() or show() We use show() to create a LaTeX expression from the Sage expression and display it in nicely formatted output.[27,28] See Fig. 14.



```
In [284]: var('a,x'); latex( (a*x^2)/5 )
Out[284]: \frac{1}{5} \, a x^{2}

In [286]: var('a,x'); view( (a*x^2)/5 )

In [287]: var('a,x'); show( (a*x^2)/5 )
```

$\frac{1}{5}\,ax^2$

Fig. 14: latex() and show() in SageMath commands within a Jupyter notebook

c) Latex() If you already have prepared LaTeX code from a document, you can use it in the Sage code cell. The most stable is probably using the `Latex` command.[29]

Using `Latex()`, the LaTeX code between the delimiters `"""` is rendered by the LaTeX display in math mode. In contrast, `latex()` proceeds in such a way that the LaTeX code between the `"""` is only converted to LaTeX code in math mode.

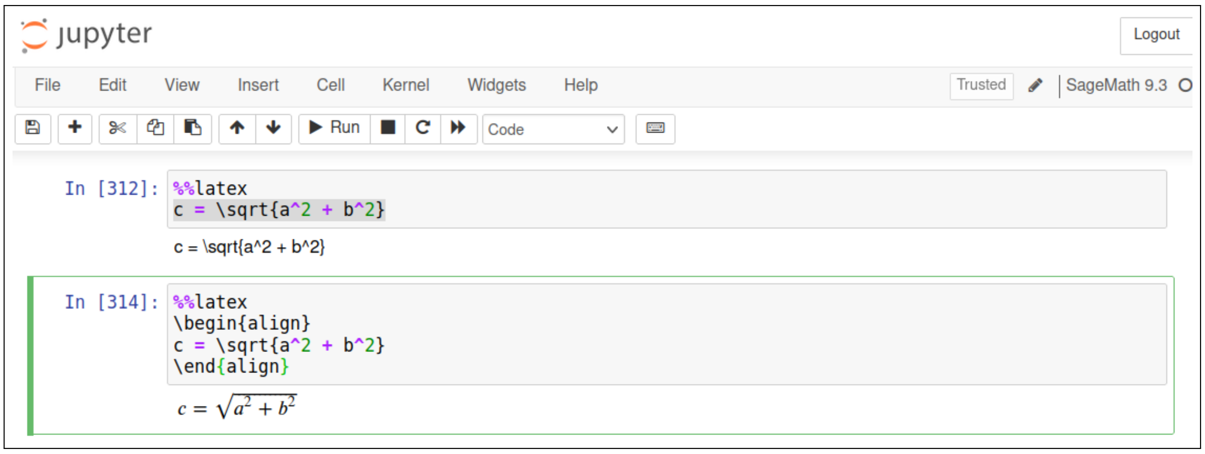
```
from IPython.display import Latex
Latex(r"""
\begin{align}
c = \sqrt{a^2 + b^2}
\end{align}
""")
```

Built-in magic commands for LaTeX in a Jupyter notebook This further variant has become rather uncommon. Here (instead of working with display()) the so-called "magic"[30] command `%%latex` is used. This command turns the entire cell into a LaTeX cell. See Fig. 15 on the following page.

[27] You can find further, not always completely up-to-date details under `https://doc.sagemath.org/html/en/tutorial/latex.html`.
[28] LaTeX must be installed to use the show() command.
[29] Less often used is the further display variant getting a raw LaTeX string for the math object.
[30] Jupyter notebooks have built-in magic commands ("magics") that start with `%`. For example, the IPython kernel extends the Python syntax this way. A distinction is made between two different types of magic:
- `Line magics` denoted by a single % prefix and run on a single input line
- `Cell magics` preceded by a double symbol `%%`. This allows you to run a single notebook cell in a subprocess of another interpreter (like bash, R, or latex) (rather than in the default kernel).

Whether and which extensions are available depends on the configured kernel (SageMath offers fewer magics than IPython).

```
%%latex
\begin{align}
c = \sqrt{a^2 + b^2}
\end{align}
```



```
In [312]: %%latex
          c = \sqrt{a^2 + b^2}

          c = \sqrt{a^2 + b^2}

In [314]: %%latex
          \begin{align}
          c = \sqrt{a^2 + b^2}
          \end{align}
```

$c = \sqrt{a^2 + b^2}$

Fig. 15: Usage of `%%latex()` in a Jupyter notebook

Two further examples:[31] Here text and Sage code are combined in different ways. See Fig. 16 on the next page.

1) Concatenate with `LatexExpr()`

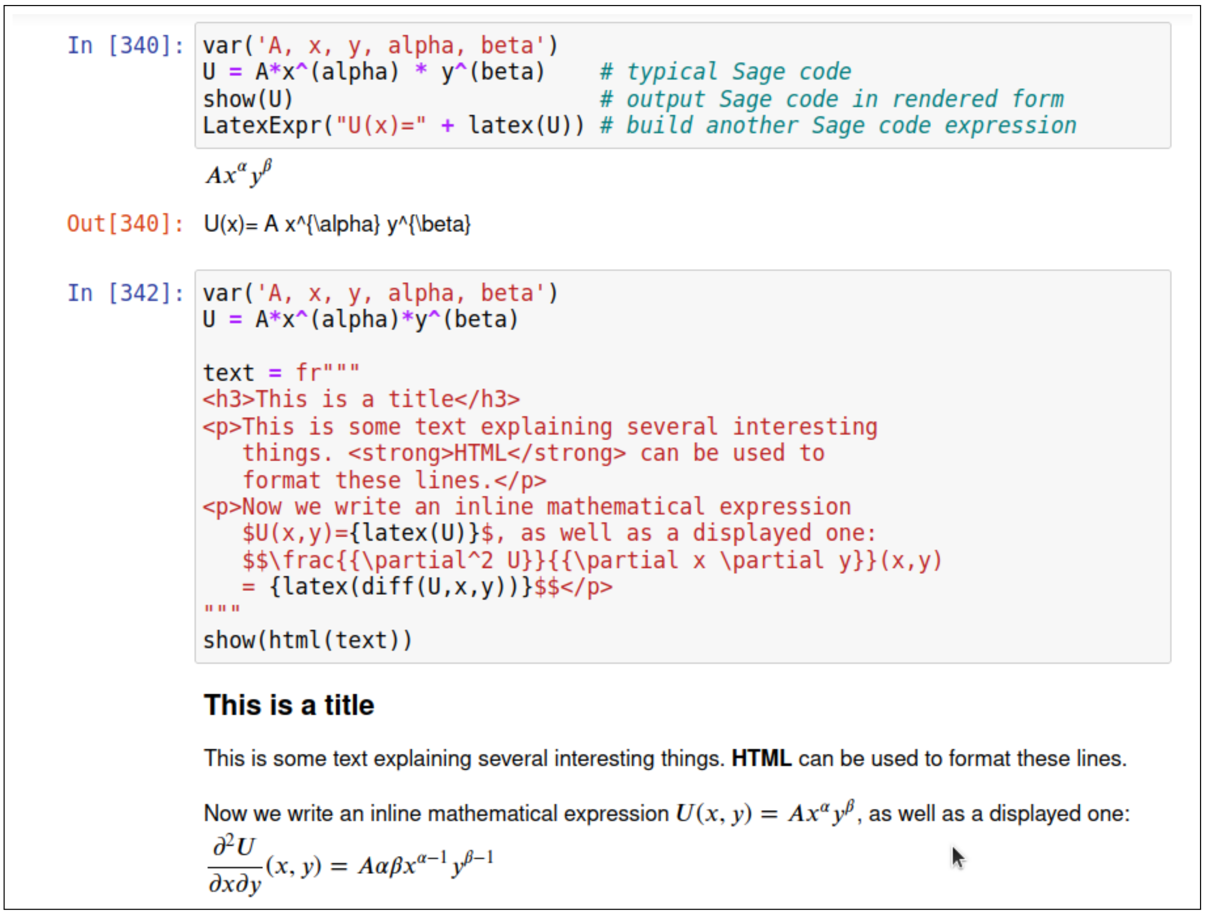
```
var('A, x, y, alpha, beta')
U = A*x^(alpha) * y^(beta)     # typical Sage code
show(U)                        # output Sage code in rendered form
LatexExpr("U(x)=" + latex(U)) # build another Sage code expression
```

2) Create html code containing Sage expressions and LaTeX commands

SageMath Example 1.11: Create html code containing Sage expressions and LaTeX commands

```
# SageJupyter_sample010.sage: SageMath within a Jupyter notebook
var('A, x, y, alpha, beta')
U = A*x^(alpha)*y^(beta)
text = fr"""
<h3>This is a title</h3>
<p>This is some text explaining several interesting
   things. <strong>HTML</strong> can be used to
   format these lines.</p>
<p>Now we write an inline mathematical expression
   $U(x,y)={latex(U)}$, as well as a displayed one:
   $$\frac{{\partial^2 U}}{{\partial x \partial y}}(x,y)
   = {latex(diff(U,x,y))}$$</p>
"""
show(html(text))
```

[31] From https://ask.sagemath.org/question/47978/add-a-text-in-latex-in-front-of-a-result/



```
In [340]: var('A, x, y, alpha, beta')
          U = A*x^(alpha) * y^(beta)     # typical Sage code
          show(U)                        # output Sage code in rendered form
          LatexExpr("U(x)=" + latex(U)) # build another Sage code expression
```

$A x^{\alpha} y^{\beta}$

```
Out[340]: U(x)= A x^{\alpha} y^{\beta}
```

```
In [342]: var('A, x, y, alpha, beta')
          U = A*x^(alpha)*y^(beta)

          text = fr"""
          <h3>This is a title</h3>
          <p>This is some text explaining several interesting
             things. <strong>HTML</strong> can be used to
             format these lines.</p>
          <p>Now we write an inline mathematical expression
             $U(x,y)={latex(U)}$, as well as a displayed one:
             $$\frac{{\partial^2 U}}{{\partial x \partial y}}(x,y)
             = {latex(diff(U,x,y))}$$</p>
          """
          show(html(text))
```

This is a title

This is some text explaining several interesting things. **HTML** can be used to format these lines.

Now we write an inline mathematical expression $U(x, y) = A x^{\alpha} y^{\beta}$, as well as a displayed one:

$$\frac{\partial^2 U}{\partial x \partial y}(x, y) = A \alpha \beta x^{\alpha-1} y^{\beta-1}$$

Fig. 16: Usage of text and code in a code cell of the Jupyter notebook

The Jupyter cell outputs are rendered using MathJax.[32] MathJax is a cross-browser JavaScript library that displays mathematical notation in browsers. The output can be generated in several formats, including HTML with CSS styling, or scalable vector graphics (SVG) images. MathJax can only map a subset of TeX and LaTeX.

Section 1.8 on page 34 (with Jupyter and interact) describes some more use cases where automated LaTeX typesetting is used via `$...$` e. g. for labels. This makes them look nicer in the output of the **Jupyter notebook**.

1.7.3 Within a LaTeX document use Sage commands generated by `latex()`

If you pass Sage objects to the `latex()` command in SageMath, you get LaTeX code. You can simply copy this into a LaTeX document (restriction see 1.7.1 on page 25).

In SageMath example 1.12 a matrix equation[33] is solved, and then the equation is output via show(). At the end, the LaTeX commands are generated, with which the solved equation can be output (see below). The according Jupyter notebook can be seen in Fig. 17 on the following page.

SageMath Example 1.12: Code to generate the LaTeX command of an equation

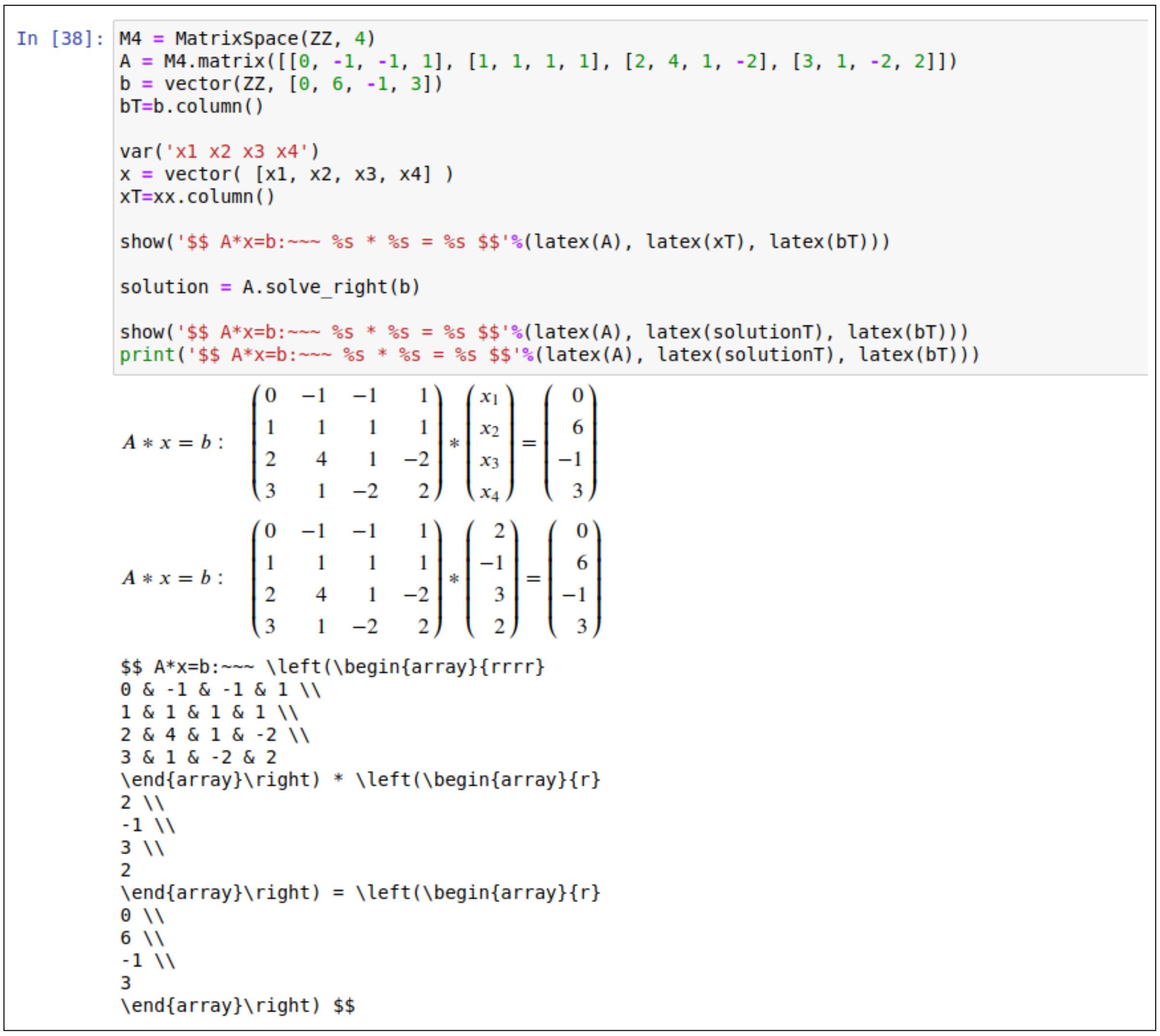
```
# SageJupyter_sample070.sage: SageMath within a Jupyter notebook
# Solve a matrix equation and generate LaTeX code to show the solved equation e.g. in a pdf.

M4= MatrixSpace(ZZ, 4)
A = M4.matrix([[0, -1, -1, 1], [1, 1, 1, 1], [2, 4, 1, -2], [3, 1, -2, 2]])
b = vector(ZZ, [0, 6, -1, 3])
```

[32] https://www.mathjax.org/

[33] The equation is from Craig Finch's book, p. 118. The code has been heavily customized.

ctd. SageMath Example 1.12

```
bT= b.column()

var('x1 x2 x3 x4')
x = vector( [x1, x2, x3, x4] )
xT= x.column()

print('$$ A*x=b:~~~ %s * %s = %s $$'%(latex(A), latex(xT), latex(bT)))

solution = A.solve_right(b)
solutionT= solution.column()

show('s:  $$A*x=b:~~~ %s * %s = %s $$'%(latex(A), latex(solutionT), latex(bT)))   # 'show' is an alias for ▶
  ▶pretty_print
        # which should choose the "best" output supported by the user interface; but showed only escaped ▶
          ▶text on console
print('p:  $$A*x=b:~~~ %s * %s = %s $$'%(latex(A), latex(solutionT), latex(bT)))  # 'print' shows text for ▶
  ▶LateX on console; 'view' had no effect.

import matplotlib.pyplot as plt
plt.plot([0,2,1,4,9])   # In a sage file (e.g. in VSC with sage extension), this has no effect.
```



```
In [38]: M4 = MatrixSpace(ZZ, 4)
         A = M4.matrix([[0, -1, -1, 1], [1, 1, 1, 1], [2, 4, 1, -2], [3, 1, -2, 2]])
         b = vector(ZZ, [0, 6, -1, 3])
         bT=b.column()

         var('x1 x2 x3 x4')
         x = vector( [x1, x2, x3, x4] )
         xT=xx.column()

         show('$$ A*x=b:~~~ %s * %s = %s $$'%(latex(A), latex(xT), latex(bT)))

         solution = A.solve_right(b)

         show('$$ A*x=b:~~~ %s * %s = %s $$'%(latex(A), latex(solutionT), latex(bT)))
         print('$$ A*x=b:~~~ %s * %s = %s $$'%(latex(A), latex(solutionT), latex(bT)))
```

$$A*x=b:~~~ \left(\begin{array}{rrrr} 0 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \\ 2 & 4 & 1 & -2 \\ 3 & 1 & -2 & 2 \end{array}\right) * \left(\begin{array}{r} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}\right) = \left(\begin{array}{r} 0 \\ 6 \\ -1 \\ 3 \end{array}\right)$$

$$A*x=b:~~~ \left(\begin{array}{rrrr} 0 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \\ 2 & 4 & 1 & -2 \\ 3 & 1 & -2 & 2 \end{array}\right) * \left(\begin{array}{r} 2 \\ -1 \\ 3 \\ 2 \end{array}\right) = \left(\begin{array}{r} 0 \\ 6 \\ -1 \\ 3 \end{array}\right)$$

```
$$ A*x=b:~~~ \left(\begin{array}{rrrr}
0 & -1 & -1 & 1 \\
1 & 1 & 1 & 1 \\
2 & 4 & 1 & -2 \\
3 & 1 & -2 & 2
\end{array}\right) * \left(\begin{array}{r}
2 \\
-1 \\
3 \\
2
\end{array}\right) = \left(\begin{array}{r}
0 \\
6 \\
-1 \\
3
\end{array}\right) $$
```

Fig. 17: Solve a matrix equation and create the LaTeX code of the equation

Here is the result of the LaTeX commands generated by `latex()`:

$$A * x = b : \begin{pmatrix} 0 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \\ 2 & 4 & 1 & -2 \\ 3 & 1 & -2 & 2 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 6 \\ -1 \\ 3 \end{pmatrix}$$

$$A * x = b : \begin{pmatrix} 0 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \\ 2 & 4 & 1 & -2 \\ 3 & 1 & -2 & 2 \end{pmatrix} * \begin{pmatrix} 2 \\ -1 \\ 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 6 \\ -1 \\ 3 \end{pmatrix}$$

1.7.4 Within a LaTeX document use Sage code handled by `SageTeX()`

This is – compared to Subsection 1.7.3 – the more exciting part.

Suppose you have a TEX distribution on your machine. Then you can enter SageMath code directly into LATEX documents, you can then run this code and embed the result automatically in the generated (usually: pdf) output.

For this, you need the package **SageTeX** which has to be included in the header of the TEX file via `\usepackage{sagetex}`. MikTEX brings `sagetex` with it, while in case you have TEXLive or MacTEX, you can get the package and the documentation here: `https://ctan.org/pkg/sagetex?lang=en`.

The TEX file you want to apply SageTeX to should be located in the same folder as the SageTeX files that you downloaded as a zip folder and then extracted. Alternatively, the SageMath installation also comes with this package, but you have to do some additional work to use it, see `https://doc.sagemath.org/html/en/reference/misc/sagetex.html`.

If you have a LATEX document that uses this package, the generation of the pdf is done in three steps as follows:

- `pdflatex file.tex` ⟶ the file `file.sagetex.sage` is created as well as the files `file.aux, file.log, file.pdf`.
- `sage file.sagetex.sage` ⟶ `sage` is run on the SageMath code that was generated in the last step and written to `file.sagetex.sage`. Here the three files `file.sagetex.[sage.py, scmd, sout]` are generated.
- Again call `pdflatex file.tex` ⟶ `file.pdf` is finished.

This workflow is summarized in Fig. 18.

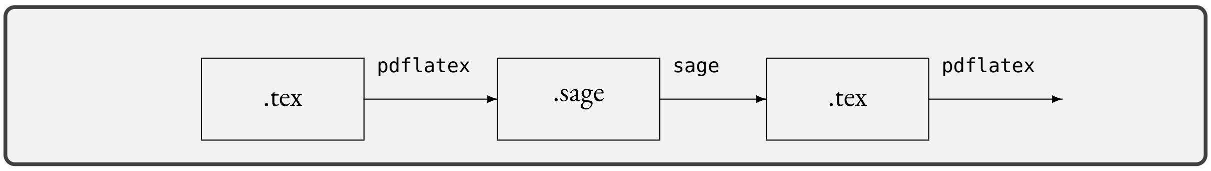


Fig. 18: Workflow with SageTeX

Remark: For experienced LATEX users and bigger LaTeX projects we recommend to use **arara**[34] to deal with

[34] arara by Paulo Cereda is a TeX automation tool based on rules written at the beginning of the main tex file of a LaTeX project. You have to write these rules about what arara should do into your LaTeX document by yourself. arara then determines its actions from these rules – rather than automatically generating rules from indirect resources, such as log file analysis. arara requires a Java virtual machine. See `https://gitlab.com/islandoftex/arara` and `https://islandoftex.gitlab.io/arara/`. Since 2024, arara is no longer part of the MacTeX delivery, but can be installed there later.

workflows like in Fig. 18. We want to introduce arara with a minimal working example (MWE) – see SageMath example 1.13. Try out entering `arara -w mwe.tex` in the terminal after saving the MWE example to a file named `mwe.tex`. The result of this command is, that the number 32 is printed in the PDF.

The prerequisite is that “pdflatex” has access to the file “sagetex.sty” from the installed SageMath version. If this is not the case, you can find instructions at https://doc.sagemath.org/html/en/tutorial/sagetex.html#sec-sagetex-install.

SageMath Example 1.13: Arara and sagetex

```
\documentclass{scrartcl}
% arara: pdflatex
% arara: sage: { files: [mwe.sagetex.sage]}
% arara: pdflatex
\usepackage{sagetex}
\begin{document}
$$\sage{2^5}$$
\end{document}
```

With SageTeX all the computational and LaTeX-formatting features of SageMath can be handled automatically. The SageTeX package is a collection of TeX macros[35], that allow a LaTeX document to include instructions to let SageMath compute and/or format various objects.

In the following sample, the LaTeX code for the matrix is created by the Sage code – the LaTeX file contains only the following four lines:[36]

```
\begin{sagesilent}
  var('x,y')
  M = matrix([[i*x+j*y for i in range(3)] for j in range(3)])
\end{sagesilent}
```

The generated matrix then looks like this:

$$M := \begin{pmatrix} 0 & x & 2\,x \\ y & x+y & 2\,x+y \\ 2\,y & x+2\,y & 2\,x+2\,y \end{pmatrix}$$

SageMath commands can also be used in interact samples (compare appendix 1.8.3 on page 36 and appendix 1.8.4 on page 37).

[35]Further details about SageTeX macros can be found at: https://ctan.org/pkg/sagetex, https://doc.sagemath.org/html/en/tutorial/latex.html and https://doc.sagemath.org/html/en/tutorial/sagetex.html.
Besides **SageTeX**, there are also Python packages which can automate things in a LaTeX document. The most well known one is **PythonTeX** (https://ctan.org/pkg/pythontex).
The almost 60-page document https://cryptool.org/download/ctb/PythonTex-by-Examples.pdf contains many examples and testcases. PythonTeX is often used together with SymPy, a LaTeX package that allows you to also embed the results from a symbolic maths package within a LaTeX document.
Alternative packages like `pyLaTeX` by Jelte Fennema, v1.3.4, 2020 or `hybrid-latex` by Leo Brewin, v0.1, 2018 are no more maintained.
[36]taken from https://phubert.github.io/sagetex-tutorial.pdf, page 8

1.8 SageMath with Jupyter and interact

With `interact`, SageMath programs become interactive – the impact of parameters on calculations can be visualized dynamically. This is done in a very general way using Python functionality. "Therefore, nearly every possible dependency could be shown."[37]

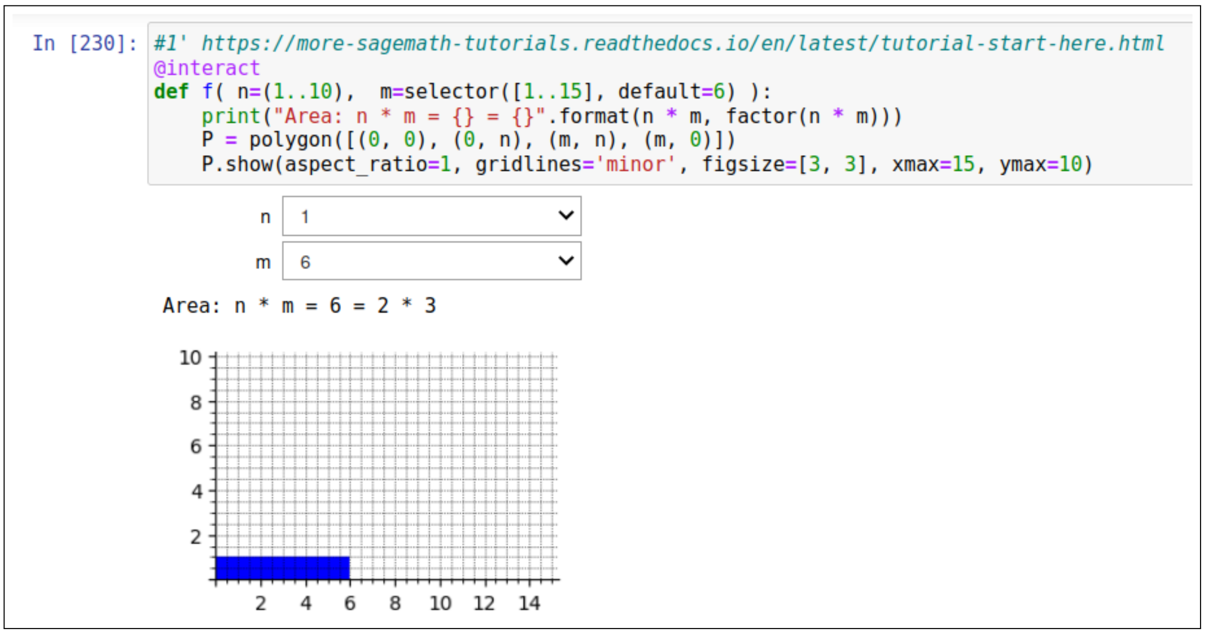
Till now, we showed the result of the calculations as text, plot or as (animated) gif. With interact, the user is able to manipulate predefined parameters and see the changed results instantaneously.

1.8.1 A typical sample with `interact()`

Figure 19 shows a slightly modified example from the SageMath tutorials[38]: In the two dropdown controls you can change the length of both sides `n` and `m` of a rectangle, which is then shown. Then its area size is calculated dynamically and the factors of the size are printed as text.

SageMath Example 1.14: Starting sample using `interact` in a Jupyter notebook

```
# SageJupyter_sample020.sage: SageMath within a Jupyter notebook
@interact
def f( n=(1..10),  m=selector([1..15],default=6) ):
    print("Area: n * m = {} = {}".format(n * m, factor(n * m)))
    P = polygon([(0, 0), (0, n), (m, n), (m, 0)])
    P.show(aspect_ratio=1, gridlines='minor', figsize=[3, 3], xmax=15, ymax=10)
```



```
In [230]: #1' https://more-sagemath-tutorials.readthedocs.io/en/latest/tutorial-start-here.html
          @interact
          def f( n=(1..10),  m=selector([1..15], default=6) ):
              print("Area: n * m = {} = {}".format(n * m, factor(n * m)))
              P = polygon([(0, 0), (0, n), (m, n), (m, 0)])
              P.show(aspect_ratio=1, gridlines='minor', figsize=[3, 3], xmax=15, ymax=10)
```

n 1

m 6

```
Area: n * m = 6 = 2 * 3
```

Fig. 19: A starting sample using `interact` in a Jupyter notebook

For the interaction with the user there are currently eight interactive Sage widget controls:

- boxes
- sliders
- range sliders
- checkboxes
- selectors (drop-down lists or buttons)
- grid of boxes

[37] https://www.sagemath.org/tour-graphics.html

[38] https://more-sagemath-tutorials.readthedocs.io/en/latest/tutorial-start-here.html

- color selectors
- plain text

In Fig. 19 on the previous page, two drop-down selectors are used. The first one for `n` is used implicitly, the second one for `m` is called explicitly in order to add an option for setting the initial value (6) used at the beginning (if no initial value is set, the first value in the list is used as default).

Many more details can be found on the following pages and at the links in the footnote.[39]

1.8.2 Technically – what are Decorators

Decorators are a construct in Python 3 that can be used to add a certain functionality to an existing function (or method). A decorator takes in a function, adds some functionality and returns it. So the decorator acts as a wrapper.[40]

The following two short examples are exactly the same –besides different notation. Because of better readability, only the `@` way is used. This is just a “syntactic sugar” to implement the decorator function.

```
def myplot(f=x^2):
    show(plot(f,(x,-3,3)))
myplot=interact(myplot)

@interact
def myplot(f=x^2):
    show(plot(f,(x,-3,3)))
```

The short examples are taken from the SageMath documentation.[41] The graph of a function `f` will be plotted, and the user can change the function `f`. We defined a new function `myplot` and made `f` a variable. To make a “control” allowing the user to interactively enter the function `f`, we just preface the function `myplot` with `@interact`. Note that we can still call the `myplot` function, even when we’ve used `@interact`. This is often useful in debugging it: `myplot(x^4)`.

It’s a convention to use the underscore “_” for throw-away names that we don’t care about like `myplot`. See the next example and Fig. 20 on the following page. In the figure, the option `aspect_ratio` is used: Values smaller than `1` squeeze the graph vertically which helps to reduce the needed vertical space. Then first `_(x^2)` is plotted and then `_(x^3)`.

```
@interact
def _(f=x^2):
    show(plot(f,(x,-3,3)))
_(x^3)
```

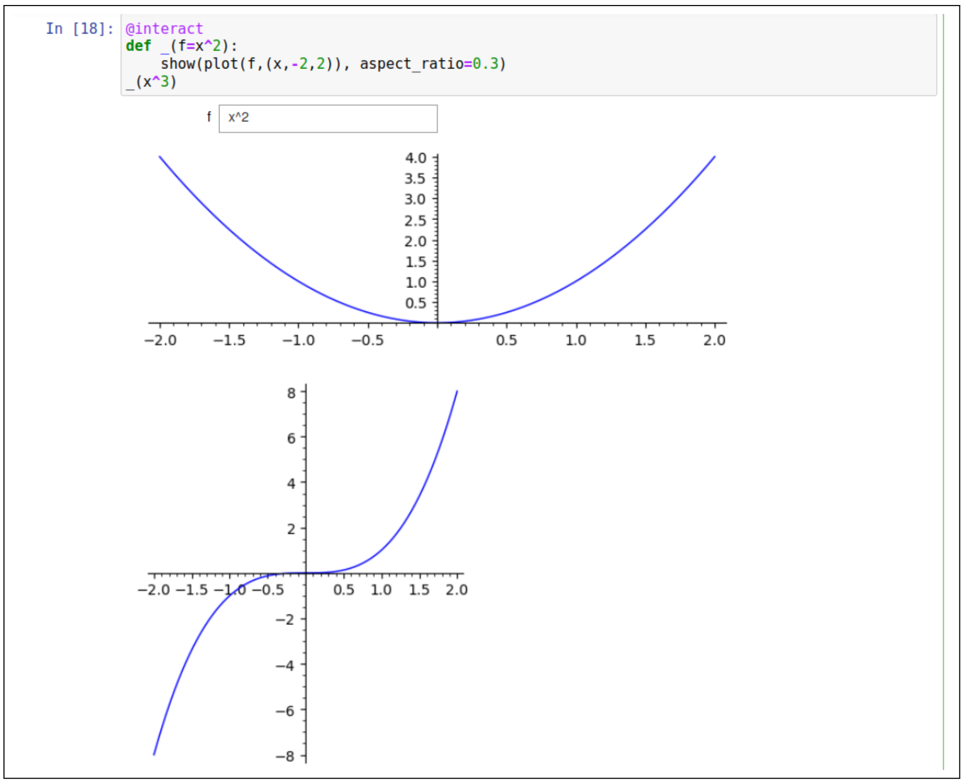
`@interact` “magically” offers in the next cell for each argument of the unnamed function a control, where the user can change the according argument. The result (calculation or plot) is adapted without having to click on “Run” again.

Using the interact method, basic widgets (graphical controls already included in SageMath) can be displayed automatically and linked to the parameters of user-defined functions. Widgets from Matplotlib or from the Jupyter notebook are more flexible and their usage is shown in SageMath example 1.17 on page 39.

[39] https://more-sagemath-tutorials.readthedocs.io/en/latest/tutorial-start-here.html,
https://doc.sagemath.org/html/en/prep/Quickstarts/Interact.html,
https://wiki.sagemath.org/interact/,
https://wiki.sagemath.org/interact/graphics

[40] The parameters of the decorator function are the same as the parameters of function it decorates. So any number of parameters can be used. In Python, this “magic” is done via function(*args,**kwargs), where args is the tuple of positional arguments and kwargs is the dictionary of keyword arguments. See https://docs.python.org/3/tutorial/controlflow.html#defining-functions.

[41] https://doc.sagemath.org/html/en/prep/Quickstarts/Interact.html



```
In [18]: @interact
         def _(f=x^2):
             show(plot(f,(x,-2,2)), aspect_ratio=0.3)
         _(x^3)
```

Fig. 20: Calling an unnamed function automatically via interact and then again directly

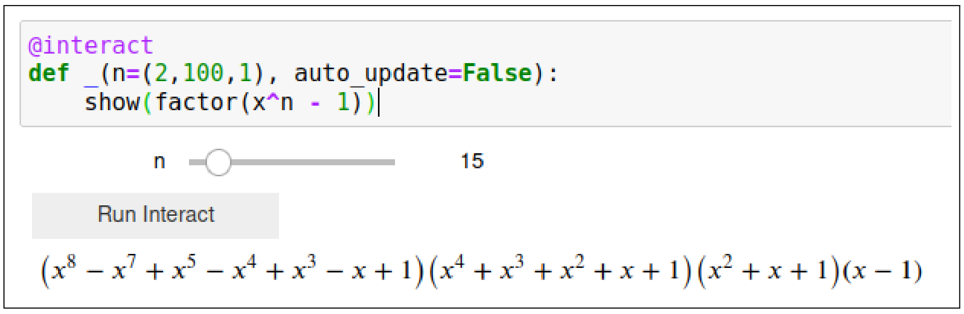
Widgets can be created either directly or through the interact function. Because it is convenient for quick use, mostly the interact method is used. Interact takes a function as its first argument, followed by the arguments of this function with their possible values. This creates widgets that allow to select those values, making a callback with the current value for every selection.

1.8.3 Interact samples without graphics

`@interact` is mainly used to show interactive graphs, but it also can be used for showing parameterized text output.

The following example[42] factors a symbolic expression depending on the exponent `n`.

```
@interact
def _(n=(2,100,1), auto_update=False):
    show(factor(x^n - 1))
```



```
@interact
def _(n=(2,100,1), auto_update=False):
    show(factor(x^n - 1))
```

$$\left(x^8 - x^7 + x^5 - x^4 + x^3 - x + 1\right)\left(x^4 + x^3 + x^2 + x + 1\right)\left(x^2 + x + 1\right)(x - 1)$$

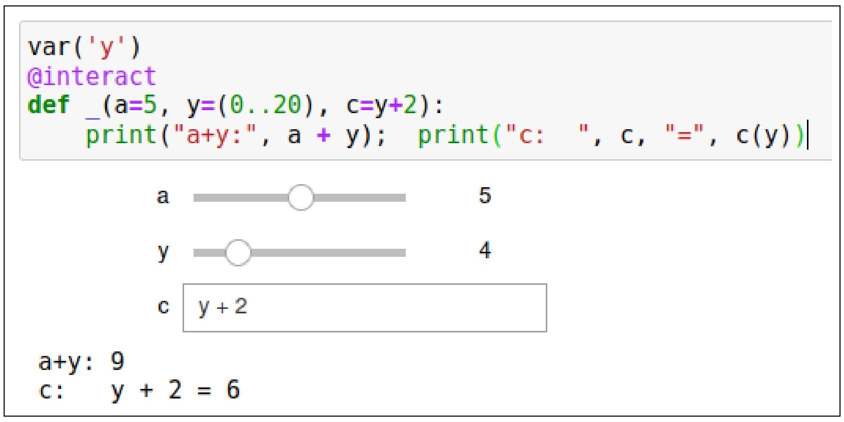
Fig. 21: Calling a function without a graph via interact (using auto_update)

[42]from http://fe.math.kobe-u.ac.jp/icms2010-dvd/SAGE/www.sagemath.org/doc/reference/sagenb/notebook/interact.html

Moving the slider in Fig. 21 on the preceding page changes the exponent. Here the recalculation and redraw is only performed after the user pushes the "Run Interact" button (via setting the option `auto_update` to False). Disabling this default makes sense for functions whose evaluation takes awhile or to prevent flickering because normally the recalculation is started at each point where the slider is moved over.

The sample in Fig. 22[43] builds an interact control set with three inputs: two slider inputs for `a` and `y` (running through the range of integers from -5 to 15 and from 0 to 20), and a text input for the expression `c`. Note that printing `c` shows its symbolic expression `y+2`, while `c(y)` evaluates its value.

```
var('y')
@interact
def _(a=5, y=(0..20), c=y+2):
    print("a+y:", a + y);  print("c:  ", c, "=", c(y))
```



```
var('y')
@interact
def _(a=5, y=(0..20), c=y+2):
    print("a+y:", a + y);  print("c:  ", c, "=", c(y))
```

```
a+y: 9
c:   y + 2 = 6
```

Fig. 22: Function with 2 inputs and evaluating a symbolic expression via interact

1.8.4 Interact samples with graphics

Now some more examples are shown, which use interact for manipulating graphics.[44]

The sample in Fig. 23 on the following page and in SageMath example 1.15 is adapted and enhanced – from the very good book [Fin11]. The curve $1/((x-a)^e) = (x-a)^{-e}$ has two interact parameters a, e. The graph shows the poles at $x = a$ for odd values of e and a flexible range of y values depending on the value of e. The labels of the axes are set manually in LaTeX in the code cell.

SageMath Example 1.15: Graph with vertical pole and parameters for different curves (see Fig. 23)

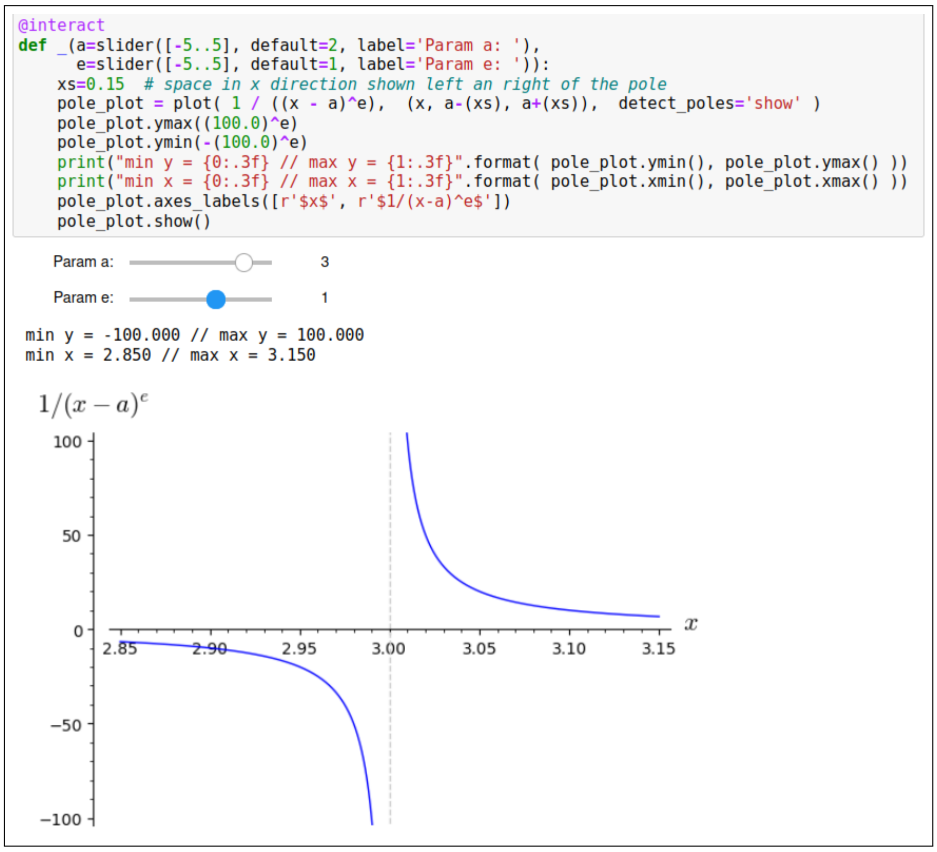
```
# SageJupyter_sample030.sage: SageMath within a Jupyter notebook
@interact
def _(a=slider([-5..5], default=2, label='Param a: '),
      e=slider([-5..5], default=1, label='Param e: ')):
    xs=0.15  # space in x direction shown left an right of the pole
    pole_plot = plot( 1 / ((x - a)^e),  (x, a-(xs), a+(xs)),  detect_poles='show' )
    pole_plot.ymax((100.0)^e)
    pole_plot.ymin(-(100.0)^e)
    print("min y = {0:.3f} // max y = {1:.3f}".format( pole_plot.ymin(), pole_plot.ymax() ))
    print("min x = {0:.3f} // max x = {1:.3f}".format( pole_plot.xmin(), pole_plot.xmax() ))
    pole_plot.axes_labels([r'$x$', r'$1/(x-a)^e$'])
```

[43] taken and modified from http://fe.math.kobe-u.ac.jp/icms2010-dvd/SAGE/www.sagemath.org/doc/reference/sagenb/notebook/interact.html

[44] Many more examples can be found at:
https://wiki.sagemath.org/interact/misc
https://wiki.sagemath.org/interact/games
https://wiki.sagemath.org/interact/graph_theory
https://www.sagemath.org/tour-graphics.html
https://wiki.sagemath.org/art

ctd. SageMath Example 1.15

```
    pole_plot.show()
```



```
@interact
def _(a=slider([-5..5], default=2, label='Param a: '),
      e=slider([-5..5], default=1, label='Param e: ')):
    xs=0.15  # space in x direction shown left an right of the pole
    pole_plot = plot( 1 / ((x - a)^e),  (x, a-(xs), a+(xs)),  detect_poles='show' )
    pole_plot.ymax((100.0)^e)
    pole_plot.ymin(-(100.0)^e)
    print("min y = {0:.3f} // max y = {1:.3f}".format( pole_plot.ymin(), pole_plot.ymax() ))
    print("min x = {0:.3f} // max x = {1:.3f}".format( pole_plot.xmin(), pole_plot.xmax() ))
    pole_plot.axes_labels([r'$x$', r'$1/(x-a)^e$'])
    pole_plot.show()
```

Param a: 3

Param e: 1

```
min y = -100.000 // max y = 100.000
min x = 2.850 // max x = 3.150
```

Fig. 23: Graph with vertical pole and parameters (compare Fig. 24)

Even a bit more flexible than SageMath example 1.15 on the preceding page is SageMath example 1.16 (without figure): It has a similar functionality, but the function $g()$ is defined **before** the unnamed function and so the function name g can be used in the unnamed function instead of using its expression several times. This works fine within `plot()` – and also within `axes_labels` the italic LaTeX presentation is shown for the evaluated g function: The trick here – learned from `ask.sagemath.org` – is to get the string for the label of the y-axis via concatenation: `'$'+latex(g(a,e,x))+'$'`.[45]

SageMath Example 1.16: Graph with vertical pole and parameters and generic function (without fig.)

```
# SageJupyter_sample040.sage: SageMath within a Jupyter notebook
var('a, x')
g(a,x) = a*x^3
@interact
def _(a=slider([-5..5], default=2, label='Param a: ')):
    p = plot(g(a,x),(x,-3,3), color='purple')  # make the plot line purple
    # p.axes_labels([ '$x$', '$a*x^3$' ])
    # p.axes_labels([ '$x$', eval('g()') ])
    p.axes_labels([ '$x$', '$'+latex(g())+'$' ])
    show(p)
    print("g: %s,   g()=%s,  g(a)=%s,  g(a,x)=%s,  g(a,1)=%d,  g(a,2)=%d" % ( g, g(), g(a), g(a,x), g(a,1),▸
     ▸ g(a,2) ))
#
# g: (a, x) |--> a*x^3,   g()=a*x^3,  g(a)=2*x^3,  g(a,x)=2*x^3,  g(a,1)=2,  g(a,2)=16
```

[45] Some parameters in the SageMath plot are still a bit unflexible. Using the underlying Mathplotlib directly can give more robustness.

1.9 SageMath with Jupyter and Matplotlib interactive_output

The graphics (plots) in SageMath are created with the Python package Matplotlib.[46] Matplotlib is the most common graphics library for Python in general.

The most commonly used functions of matplotlib are directly accessible through Sage functions. So far we have limited our examples to the graphics functions built into SageMath (see the Sage widgets in 1.8.1 on page 34).

Alternatively, you can either import the whole Mathplotlib package into SageMath (and then use its complete functionality) or use the graphics functions (widgets or controls) from the Jupyter notebook, which are also based on Mathplotlib. In the following example we use the Jupyter widgets and achieve a nicer layout than with the Sage widgets.

The previous sample in Fig. 23 and in SageMath example 1.15 on page 37 showed the controls below each other. This is practical for a quick result, but doesn't deliver a nice user interface. Now we reconstruct this example using Jupyter widgets (ipywidgets)[47].

The two ipywidgets.IntSlider act as input; ipywidgets.Label is used as output (instead of print). All four widgets are initially generated independently. Then the widgets are inserted into a grid consisting of $2 \cdot 3 = 6$ cells. The sliders take up the first two columns, the labels take the last column; Sliders are therefore twice as wide as the labels. Labels are not only available for coordinate axes (like in the previous examples), but can be placed anywhere. Label widgets display the value text but are not editable, which is intended here.

Not only g() but also myplot() is separately defined, so that the interactive function is relatively small: It calls myplot() and assigns the slider input controls aa and ee to the input variables a and e of the myplot() function.

With the help of interactive_output the graphic is not output immediately, but created in an output widget. As a result, the graphic could later be output several times per display.

Overall, the ipywidgets allow a much more flexible layout than the basic widgets integrated in SageMath. However, the code is longer 😃.

SageMath Example 1.17: Code for graph with widgets from ipywidgets (see Fig. 24)

```
# SageJupyter_sample060.sage: SageMath within a Jupyter notebook
# Widgets from ipywidgets and called via interactive_output

from ipywidgets import GridspecLayout, Layout, IntSlider, Label, interactive_output

var('a, e, x')
g(a,e,x) = 1 / ((x - a)^e)

# Create the controls
aa = IntSlider(min=-5, max=5, value=2, description='Param a: ', layout=Layout(width='auto', height='auto'))
ee = IntSlider(min=-5, max=5, value=1, description='Param e: ', layout=Layout(width='auto', height='auto'))

label1 = Label(description="for_x", layout=Layout(display='flex', justify_content="center", border='1px ▸
  ▸solid green'))
label2 = Label(description="for_y", layout=Layout(display='flex', justify_content="center", border='1px ▸
  ▸solid green'))

# Distribute the controls into a grid for better layout
grid = GridspecLayout(int(2), int(3))
grid[0,0:2]  = aa
grid[1,0:2]  = ee
grid[0,-1] = label1
grid[1,-1] = label2
```

[46] https://matplotlib.org/

[47] See https://ipywidgets.readthedocs.io/en/latest/examples/Using%20Interact.html. Two very nice introductions to Jupyter widgets can be found in https://www.elab2go.de/demo-py1/jupyter-notebook-widgets.php and https://kapernikov.com/ipywidgets-with-matplotlib/ (2020).

ctd. SageMath Example 1.17

```
def myplot(a, e):
    xs=0.15 # space in x direction shown to the left and right of the pole
    pole_plot = plot( g(a,e,x),  (x, a-(xs), a+(xs)),  detect_poles='show' )
    pole_plot.ymax((100.0)^e)
    pole_plot.ymin(-(100.0)^e)
    # Add spaces via the unicode character \xa0 ("nonbreaking space")
    skip = ':\xa0\xa0\xa0\xa0'
    s1 = 'x' + skip + "min = {0:.2f} \xa0 // \xa0 max = {1:.2f}".format( pole_plot.xmin(), pole_plot.xmax()▸
     ▸ )
    s2 = 'y' + skip + \
         ( "min = {0:.0f} \xa0 // \xa0 max = {1:.0f}".format(pole_plot.ymin(), pole_plot.ymax()) if (▸
          ▸pole_plot.ymin()>100 or pole_plot.ymin()<-100)
           else "min = {0:.2f} \xa0 // \xa0 max = {1:.2f}".format(pole_plot.ymin(), pole_plot.ymax()) )
    label1.value = s1
    label2.value = s2
    pole_plot.axes_labels([ '$x$', '$'+latex(g())+'$' ])
    pole_plot.show()

out = interactive_output( myplot, {'a': aa,  'e': ee} )
display(grid, out)
```

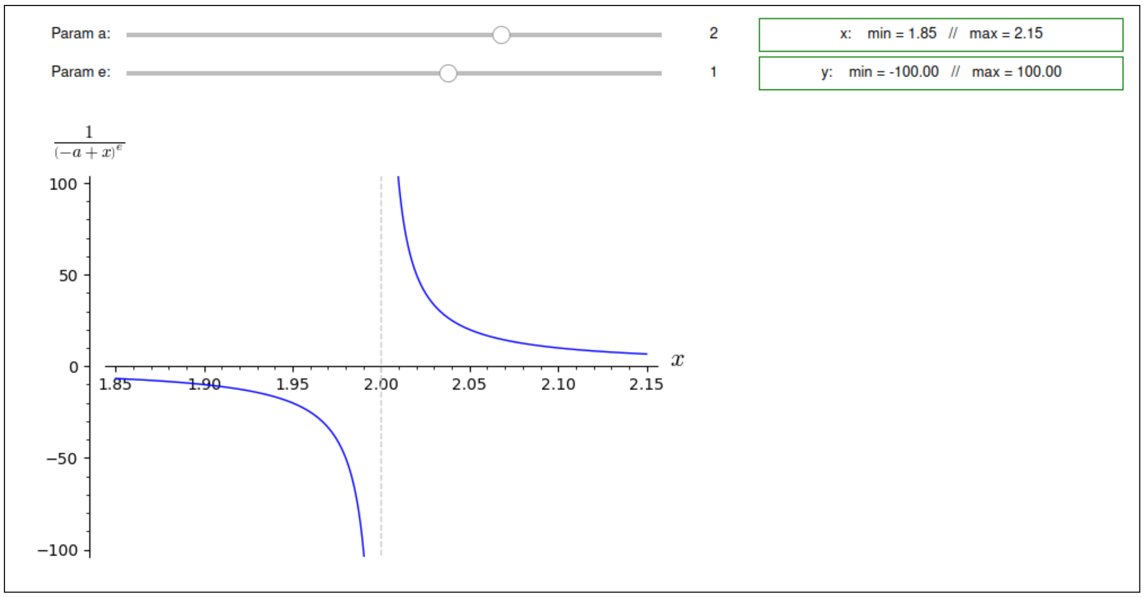

```
Param a:   2      x:   min = 1.85  //  max = 2.15
Param e:   1      y:   min = -100.00  //  max = 100.00
```

Fig. 24: Graph with widgets from ipywidgets (compare Fig. 23)

1.10 Further interact SageMath examples about cryptography

Besides the examples in this book, you can find at the link in Footnote 48 many simple examples, which present the commands from the area of cryptography in SageMath with interact:[48]

1. Shift Cipher (Encryption and Decryption)
2. Affine Cipher (Encryption and Decryption)
3. Substitution Cipher
4. Playfair Cipher (Encryption and Decryption)
5. Frequency Analysis Tools
   a) Letter Frequency Counter
   b) Frequency Analysis Decryption Guesser

[48] `https://wiki.sagemath.org/interact/cryptography` (created first at “Sage Days 103”, 7-9 August 2019)

6. Vigenère Cipher (Encryption and Decryption)
7. One-Time Pad (OTP)
8. Hill Cipher (Encryption and Decryption)
9. Modular Arithmetic (Preliminaries for RSA, Diffie-Hellman, ElGamal)
   a) Modular Arithmetic Multiplication Table
   b) Modular Exponentiation
   c) Discrete Log Problem (Solving for x and for b)
10. RSA
    a) RSA, from Alice's Perspective
    b) RSA, from Babette's Perspective
    c) RSA with Digital Signatures

1.11 The curve courses of MTW with SageMath

In the international MysteryTwister[49] (MTW, formerly MTC3) crypto puzzle competition, there are three levels in which the awarded points for a solved task are automatically calculated using a formula. The level specifies a base number of points, which the user gets at least and where the number of points converges to – the longer the task has been published. How fast this approximation takes place depends on the level: the lower the level, the faster (see `https://mysterytwister.org/challenges/points-calculation/`).

The according graph is visualized in Fig. 25 on the next page for the first 500 days. The graph is created with the SageMath example 1.18.

The 3 curves of the 3 levels can be seen in blue – normalized to a base value of 100. With the slider you can select the weight c in small steps to see in a red curve how quickly the "rate of fall" adjusts to the base value. This red curve always moves on top of the other curves; and hides the blue curve of level 1 in Fig. 25, as $c = 1.00$ in level 1.

The base value is 50 % of the maximum value (here in the graph it is always 200; in MTW it is 200, 2000, and 20000). Each of the three levels starts with its maximum value. After 10 days (L1/L2/L3) are each at (55/74/86) % of the maximum value. Level 1 (L1) reaches 50 % after just after 101 days. After 500 days (L2/L3) are each at (57/70) % of the maximum value.

SageMath Example 1.18: Code generating the graph for MTW points over time

```
# SageJupyter_sample050.sage: SageMath within a Jupyter notebook
# MTW formular: points for one challenge after d days

c,d  = var('c, d')

def f(c, d):
    r = 1 - ( 1 / (( 2^(1-c)) * ((d+1)^c)) )
    return ( r )

def g(c,d):
    100 / f(c,d)

@interact
def _( c = slider( 0, 2, step_size=0.05, default=1 ) ):
    g1 = 100 / f(1, d)
    p1 = plot(g1, (1, 500), thickness=1, color='blue' )

    g025 = 100 / f(0.25, d)
    p025 = plot(g025, (1, 500), thickness=1, color='blue' )

    g01 = 100 / f(0.1, d)
```

[49] `https://mysterytwister.org/`

ctd. SageMath Example 1.18

```
p01 = plot(g01, (1, 500), thickness=1, color='blue' )

g = 100 / f(c, d)  # 100 = 10*10^i, where i=1 (so all curves use level 1 base value)
# the x-axis variable is d and runs from day 1 to 500
p = plot(g, (1, 500), thickness=2, color='red' )

(p1+p025+p01+p).show(title='MTW curve courses for level L1, L2, and L3')
```

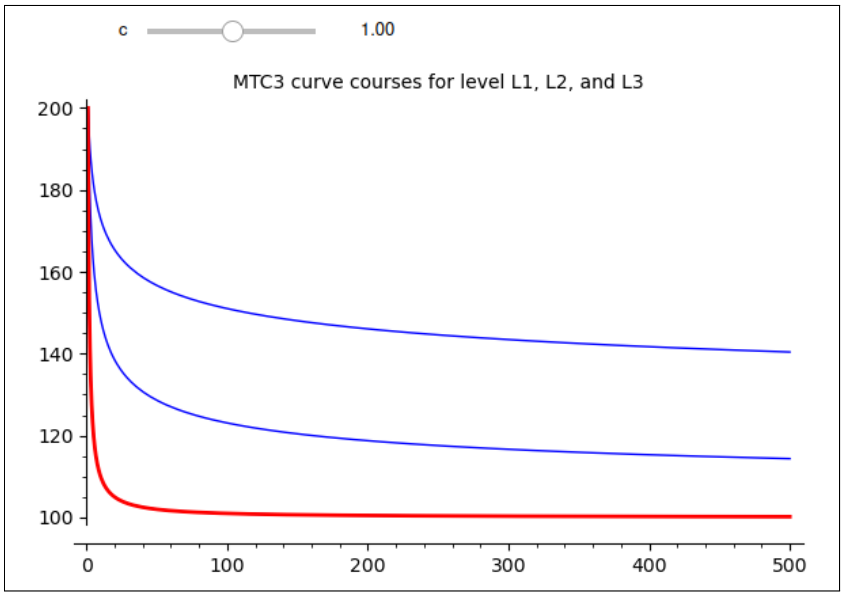


Fig. 25: Graph about MTW points over time

1.12 More professional Sage programs

In the literature and also here, you can often find short code examples for SageMath. But it is also possible to write very professional programs with Sage. This is mainly due to the fact that you can use in Sage programs all the possibilities of Python 3.

In SageMath example 2.8.19 of the CrypTool book you can see that to some extent. The analysis program

- has the typical structure of a Python program with a clear main function at the end.
- breaks down all tasks into manageable functions.
- parses the command line with `argparse`.
- contains test data (here in a `dictionary`).
- supports a verbose mode, which can be used to output internal processes (inner workings) and log information ad hoc.
- contains comments, usage info, and versioning.

1.13 Further hints for SageMath in this book

Some more hints:

- To get the version of your SageMath environment: `version()`
- To move quickly to the SageMath examples in this book,

  - either look in the index at `SageMath -> Code examples`,
  - or have a look at Chapter 2 on page 45
- The SageMath examples in this book can be found at the CrypTool website for download. Further details at the end of the overview in Chapter 2 on page 45.

All links of this appendix have been confirmed at June 5, 2024.

Acknowledgments

I would like to take this opportunity to thank Doris Behrendt for her thorough and constructive proofreading of this chapter.

2 Lists of Figures, Tables, Code Examples, etc.

2.1 List of Figures

2.2 List of Tables

2.3 List of SageMath Examples

[1]With small changes taken from `https://mlsummit.ai/blog/jupyter-notebooks-fuer-lehre-und-entwicklung-alles-im-blick-notizbuch-fuer-entwickler/`

All SageMath examples of this book can be found on the CrypTool website:
`https://www.cryptool.org/en/ctbook/sagemath`

The names of the SageMath script files contain first the chapter number and then the running number of the script within that chapter, e. g. `chap09_sample110.sage`.

All examples have been tested with the SageMath versions 9.3 (release date 2021-05-09) under Windows, 9.5 (release date 2022-01-30) under Linux, and 10.0 (release date 2023-05-23) under macOS.

3 Literature

[Fin11] Craig Finch. *Sage Beginner's Guide*. PACKT Publishing, 2011 (cit. on p. 37).

4 Index