Mathematics with SageMath

Learn math with open-source software

Mathematics with SageMath

Learn math with open-source software

Allaoua Boughrira, Hellen Colman, Samuel Lubliner
City Colleges of Chicago

December 27, 2024

Website: GitHub Repository[1]

[1] github.com/boughrira/template-math-with-sage
[2] creativecommons.org/licenses/by-sa/4.0

Preface

PLACEHOLDER

Hellen Colman
Chicago, January 2025

Acknowledgements

We would like to acknowledge the following peer-reviewers:

- [Name], [Affiliation]

We would like to acknowledge the following proof-readers:

- [Name], [Affiliation]

From the Student Authors

PLACEHOLDER
Allaoua Boughrira and Samuel Lubliner

Authors and Contributors

Allaoua Boughrira
Mathematics
Wright College
a.boughrira@gmail.com

Hellen Colman
Math Department
Wright College
hcolman@ccc.edu

Samuel Lubliner
Computer Science
Wright College
sage.oer@gmail.com

Contents

Chapter 1

Getting Started

Welcome to our introduction to SageMath (also referred to as Sage). This chapter is designed for learners of all backgrounds—whether you're new to programming or aiming to expand your mathematical toolkit. There are various options for running Sage, including the SageMathCell, CoCalc, and a local installation. The easiest way to get started is to use the SageMathCell embedded directly in this book. We will also cover how to use CoCalc, a cloud-based platform that provides a collaborative environment for running Sage code.

Sage is a free open-source mathematics software system that integrates various open-source mathematics software packages[1]. We will cover the basics, including SageMath's syntax, data types, variables, and debugging techniques. Our goal is to equip you with the foundational knowledge needed to explore mathematical problems and programming concepts in an accessible and straightforward manner.

Join us as we explore the capabilities of SageMath!

1.1 Intro to Sage

You can execute and modify Sage code directly within the SageMathCells embedded on this webpage. Cells on the same page share a common memory space. To ensure accurate results, run the cells in the sequence in which they appear. Running them out of order may cause unexpected outcomes due to dependencies between the cells.

1.1.1 Sage as a Calculator

Before we get started with discrete math, let's see how we can use Sage as a calculator. Here are the basic arithmetic operators:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Exponentiation: `**`, or `^`
- Division: `/`

[1] `doc.sagemath.org/html/en/reference/spkg/`

- Integer division: `//`
- Modulo: `%`

There are two ways to run the code within the cells:

- Click the Evaluate (Sage) button located under the cell.
- Use the keyboard shortcut Shift + Enter if your cursor is active in the cell.

```
# Lines that start with a pound sign are comments
# and ignored by Sage
1+1
```

```
100 - 1
```

```
3*4
```

```
# Sage uses two exponentiation operators
# ** is valid in Sage and Python
2**3
```

```
# Sage uses two exponentiation operators
# ^ is valid in Sage
2^3
```

```
# Returns a rational number
5/3
```

```
# Returns a floating point approximation
5/3.0
```

```
# Returns the quotient of the integer division
5//3
```

```
# Returns the remainder of the integer division
5 % 3
```

1.1.2 Variables and Names

We can assign the value of an expression to a variable. A variable is a name that refers to a value in the computer's memory. Use the assignment operator `=` to assign a value to a variable. The variable name is on the left side of the assignment operator, and the value is on the right side. Unlike the expressions above, the assignment statement does not display anything. To view the value of a variable, type the variable name and run the cell.

```
a = 1
b = 2
sum = a + b
sum
```

When choosing variable names, use valid identifiers.

- Identifiers cannot start with a digit.

- Identifiers are case-sensitive.
- Identifiers can include:
  - letters (`a - z`, `A - Z`)
  - digits (`0 - 9`)
  - underscore character `_`
- Do not use spaces, hyphens, punctuation, or special characters when naming identifiers.
- Do not use keywords as identifiers.

Below are some reserved keywords that you cannot use as variable names:

`False`, `None`, `True`, `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`.

Use the Python `keyword` module to check if a name is a keyword.

```
import keyword
keyword.iskeyword('if')
```

The output is `True` because `if` is a keyword. Try checking other names.

1.2 Display Values

Sage offers various ways to display values on the screen. The simplest way is to type the value into a cell, and Sage will display it. Sage also has functions that display values in different formats.

- `print()` displays the value of the expression inside the parentheses on the screen.
- `pretty_print()` displays rich text.
- `show()` is an alias for `pretty_print()`.
- `latex()` produces the raw LaTeX code for the expression inside the parentheses. You can paste this code into a LaTeX document to display the expression.
- `%display latex` renders the output of commands as LaTeX automatically.
- While Python string formatting is available, the output is unreliable for rendering rich text and LaTeX due to compatibility issues.

Sage will display the value of the last line of code in a cell.

```
"Hello, World!"
```

`print()` outputs a similar result without the quotes.

```
print("Hello, World!")
```

View mathematical notation with rich text.

```
show(sqrt(2) / log(3))
```

If we want to display values from multiple lines of code, we can use multiple functions to display the values.

```
a = x^2
b = pi
show(a)
show(b)
```

Obtain raw LaTeX code for an expression.

```
latex(sqrt(2) / log(3))
```

If you are working in a Jupyter notebook or SageMathCell, `%display latex` sets the display mode.

```
%display latex
# Notice we don't need the show() function
sqrt(2) / log(3)
```

The expressions will continue to render as LaTeX until you change the display mode. The display mode is still set from the previous cell.

```
ZZ
```

Revert to the default output with `%display plain`.

```
%display plain
sqrt(2) / log(3)
```

```
ZZ
```

1.3 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that models the world as a collection of interacting **objects**. More specifically, an object is an **instance** of a **class**. A class can represent almost anything.

Classes are like blueprints that define the structure and behavior of objects. A class defines the **attributes** and **methods** of an object. An attribute is a variable that stores information about the object. A method is a function that can interact with or modify the object. Although you can create custom classes, the open-source community has already defined classes for us. For example, there are specialized classes for working with integers, lists, strings, graphs, and more.

In Python and Sage, almost everything is an object. When assigning a value to a variable, the variable references an object. In this case, the object is a list of strings.

```
vowels = ['a', 'e', 'i', 'o', 'u']
type(vowels)
```

```
type('a')
```

The `type()` function confirms that `'a'` is an instance of the `string` class and `vowels` is an instance of the `list` class. We create a `list` object named `vowels` by assigning a series of characters within square brackets to a variable. This object, `vowels`, now represents a `list` of `string` elements, and we can interact with it using various methods.

Dot notation is a syntax used to access an object's attributes and call an object's methods. For example, the list class has an `append` method, allowing us to add elements to the list.

```
vowels.append('y')
vowels
```

A **parameter** is a variable passed to a method. In this case, the parameter is the string `'y'`. The `append` method adds the string `'y'` to the end of the list. The list class has many more methods that we can use to interact with the list object. While `list` is a built-in Python class, Sage offers many more classes specialized for mathematical applications. For example, we will learn about the Sage `Set` class in the next chapter. Objects instantiated from the `Set` class have methods and attributes useful for working with sets.

```
v = Set(vowels)
type(v)
```

While OOP might seem abstract at first, it will become clearer as we dive deeper into Sage. We will see how Sage utilizes OOP principles and built-in classes to offer a structured way to represent data and perform powerful mathematical operations.

1.4 Data Types

In computer science, **Data types** represent data based on properties of the data. Python and Sage use data types to implement these classes. Since Sage builds upon Python, it inherits all the built-in Python data types. Sage also provides classes that are well-suited for mathematical computations.

Let's ask Sage what type of object this is.

```
n = 2
print(n)
type(n)
```

The `type()` function reveals that `2` is an instance of the **Integer** class. Sage includes numerous classes for different types of mathematical objects.

In the following example, Sage does not evaluate an approximation of `sqrt(2) * log(3)`. Sage will retain the **symbolic** value.

```
sym = sqrt(2) / log(3)
show(sym)
type(sym)
```

String: a `str` is a sequence of characters used for text. You can use single or double quotes.

```
greeting = "Hello, World!"
print(greeting)
print(type(greeting))
```

Boolean: The type `bool` can be one of two values, `True` or `False`.

```
# Check if 5 is contained in the set of prime numbers
b = 5 in Primes()
```

```
print(f"{b} is {type(b)}")
```

List: A mutable collection of items within a pair of square brackets `[]`. If an object is mutable, you can change its value after creating it.

```
l = [1, 3, 3, 2]
print(l)
print(type(l))
```

Lists are indexed starting at `0`. Here, we access the first element of a list by asking for the value at index zero.

```
l[0]
```

Lists have many helpful methods.

```
# Find the number of elements in the list
len(l)
```

Tuple: An immutable collection within a pair of parenthesis `()`. If an object is immutable, you cannot change the value after creating it.

```
t = (1, 3, 3, 2)
print(t)
type(t)
```

set: A collection of items within a pair of curly braces `{}`. `set()` with lowercase `s` is built into Python. The items in a set are unique and unordered. After printing the set, we see there are no duplicate values.

```
s = {1, 3, 3, 2}
print(s)
type(s)
```

Set is a built-in Sage class. `Set` with a capital `S` has added functionality for mathematical operations.

```
S = Set([1, 3, 3, 2])
type(S)
```

We start by defining a `list` within square brackets `[]`. Then, the `Set()` function creates the Sage set object.

```
S = Set([5, 5, 1, 3, 5, 3, 2, 2, 3])
print(S)
```

Dictionary: A collection of key-value pairs.

```
d = {
    "title": "Discrete Math with SageMath",
    "institution": "City Colleges of Chicago",
    "topics_covered": [
        "Set Theory",
        "Combinations and Permutations",
        "Logic",
        "Quantifiers",
        "Relations",
```

```
        "Functions",
        "Recursion",
        "Graphs",
        "Trees",
        "Lattices",
        "Boolean Algebras",
        "Finite State Machines"
    ],
    "format": ["Web", "PDF"]
}
type(d)
```

Use the `pprint` module to print the dictionary in a more readable format.

```
import pprint
pprint.pprint(d)
```

1.5 Iteration

Iteration is a programming technique that allows us to efficiently write code by repeating instructions with minimal syntax. The `for` loop assigns a value from a sequence to the loop variable and executes the loop body once for each value.

```
# Print the numbers from 0 to 19
# Notice the loop is zero-indexed and excludes 20
for i in range(20):
    print(i)
```

```
# Here, the starting value of the range is included
for i in range(10, 20):
    print(i)
```

```
# We can also specify a step value
for i in range(30, 90, 9):
    print(i)
```

Here is an example of list comprehension, a concise way to create lists. Unlike Python's `range()`, the Sage range syntax for list comprehension includes the ending value.

```
# Create a list of the cubes of the numbers from 9 to 20
# The for loop is written inside the square brackets
[n**3 for n in [9..20]]
```

We can also specify a condition in list comprehension. Let's create a list that only contains even numbers.

```
[n**3 for n in [9..20] if n % 2 == 0]
```

1.6 Debugging

Error messages are an inevitable part of programming. When you make a syntax error and see a message, read it carefully for clues about the cause of the

error. While some messages are helpful and descriptive, others may seem cryptic or confusing. With practice, you will develop valuable skills for debugging your code and resolving errors. Not all errors will produce an error message. Logical errors occur when the syntax is correct, but the program does not produce the expected output. Remember, mistakes are learning opportunities, and everyone makes them. Here are some tips for debugging your code:

- Read the error message carefully for information to help you identify and fix the problem.
- Study the documentation.
- Google the error message. Someone else has likely encountered the same issue.
- Search for previous posts on Sage forums.
- Take a break and return with a fresh perspective.
- If you are still stuck after trying these steps, ask the Sage community.

Let's dive in and make some mistakes together!

```
# Run this cell and see what happens
1message = "Hello, World!"
print(1message)
```

Why didn't this print `Hello, World!` on the screen? The error message informed us of a `SyntaxError`. While the phrase `invalid decimal literal` may seem confusing, the key issue here is the invalid variable name. Valid identifiers must start with a letter or underscore. They cannot begin with a number or use any special characters. Let's correct the variable name by using a valid identifier.

```
message = "Hello, World!"
print(message)
```

Here is another error:

```
print(Hi)
```

In this case, we encounter a `NameError` because `Hi` is not defined. Sage assumes that `Hi` is a variable because there are no quotes. We can make `Hi` a string by enclosing it in quotes.

```
print("Hi")
```

Alternatively, if we intended `Hi` to be a variable, we can assign a value to it before printing.

```
Hi = "Hello, World!"
print(Hi)
```

Reading the documentation is essential to understanding how to use methods correctly. If we incorrectly use a method, we will likely get a `NameError`, `AttributeError`, `TypeError`, or `ValueError`, depending on the mistake.

Here is an example of a `NameError`:

```
l = [6, 1, 5, 2, 4, 3]
sort(l)
```

The `sort()` method must be called on the list object using dot notation.

```
l = [4, 1, 2, 3]
l.sort()
print(l)
```

Here is an example of an `AttributeError`:

```
l = [1, 2, 3]
l.len()
```

Here is the correct way to use the `len()` method:

```
l = [1, 2, 3]
len(l)
```

Here is an example of a `TypeError`:

```
l = [1, 2, 3]
l.append(4, 5)
```

The `append()` method only takes one argument. To add multiple elements to a list, use the `extend()` method.

```
l = [1, 2, 3]
l.extend([4, 5])
print(l)
```

Here is an example of a `ValueError`:

```
factorial(-5)
```

Although the resulting error message is lengthy, the last line informs us that the argument must be a non-negative integer.

```
factorial(5)
```

Finally, we will consider a logical error. If your task is to print the numbers from 1 to 10, you may mistakenly write the following code:

```
for i in range(10):
    print(i)
```

The output will be the numbers from 0 to 9. To include 10, we need to adjust the range because the start is inclusive and the stop is exclusive.

```
for i in range(1, 11):
    print(i)
```

For more information, read the CoCalc article about the top mathematical syntax errors in Sage[1]

[1] github.com/sagemathinc/cocalc/wiki/MathematicalSyntaxErrors

1.7 Defining Functions

Sage comes with many built-in functions. Math terminology is not always standard, so be sure to read the documentation to learn what these functions do and how to use them. You can also define custom functions yourself. You are welcome to use the custom functions we define in this book. However, since these custom functions are not part of the Sage source code, you will need to copy and paste the functions into your Sage environment. If you try to use a custom function without defining it, you will get a `NameError`.

To define a custom function in Sage, use the `def` keyword followed by the function name and the function's arguments. The function's body is indented. When you call the function, the `return` keyword returns a value from the function. The function definition is only stored in memory after you run the cell. You will not see any output when you run the cell that defines the function. You will see output only when you call the function. A green box under the cell indicates the successful execution of the cell. If the box is not green, you must run the cell to define the function.

You may have heard of Pascal's Triangle, a triangular array of numbers in which each number is the sum of the two numbers directly above it. Here is an example function that returns the n^{th} (0-indexed) row of Pascal's Triangle:

```
def pascal_row(n):
    return [binomial(n, i) for i in range(n + 1)]
```

Try calling the function for yourself. First, run the Sage cell with the function definition to define the function. If you try to call a function without defining it, you will get a `NameError`. After defining the function, you can use it in other cells. You won't see any output when you run the cell that defines the function. The Sage cells store the function definition memory. You will see output only when you call the function. After running the above cell, you can call the `pascal_row()` function.

```
pascal_row(5)
```

Input validation makes functions more robust. We may get some validation out of the box. For example, if we try to call the function using a string or decimal value as input, we will get a `TypeError`:

```
pascal_row("5")
```

However, if we try to call the function with a negative integer, the function will return an empty list without raising an error.

```
pascal_row(-5)
```

This lack of error handling is risky because it can go undetected and cause unexpected behavior. Let's add a `ValueError` to handle negative input:

```
def pascal_row(n):
    if n < 0:
        raise ValueError("`n` must be a non-negative
            integer")
    return [binomial(n, i) for i in range(n + 1)]
```

Running the above cell redefines the function. Try calling the function with a negative integer to see the input validation.

```
pascal_row(-5)
```

Functions can also include a `docstring` to provide documentation for the function. The `docstring` is a string that appears as the first statement in the function body. It describes what the function does and how to use it.

```
def pascal_row(n):
    r"""
    Return row `n` of Pascal's triangle.

    INPUT:

    - ``n`` -- non-negative integer; the row number of
    Pascal's triangle to return.
      The row index starts from 0, which corresponds to the
    top row.

    OUTPUT: list; row `n` of Pascal's triangle as a list of
    integers.

    EXAMPLES:

    This example illustrates how to get various rows of
    Pascal's triangle (0-indexed) ::

        sage: pascal_row(0)  # the top row
        [1]

        sage: pascal_row(4)
        [1, 4, 6, 4, 1]

    It is an error to provide a negative value for `n` ::

        sage: pascal_row(-1)
        Traceback (most recent call last):
        ...
        ValueError: `n` must be a non-negative integer

    .. NOTE::

        This function uses the `binomial` function to
    compute each
        element of the row.
    """
    if n < 0:
        raise ValueError("`n` must be a non-negative
            integer")

    return [binomial(n, i) for i in range(n + 1)]
```

After redefining the function and running the above cell, view the `docstring` by calling the `help()` function on the function name. You can also access the `docstring` with the `?` operator.

```
help(pascal_row)
# pascal_row? also displays the docstring
# pascal_row?? reveals the function's source code
```

For more information on code style conventions and writing documentation strings, refer to the General Conventions article from the Sage Developer's Guide.

1.8 Documentation

Sage can do many more mathematical operations. If you want an idea of what Sage can do, check out the Quick Reference Card[1] and the Reference Manual[2].

The tutorial[3] is an overview to become familiar with Sage.

The Sage documentation[4] can be found at this link. Right now, reading the documentation is optional. We will do our best to get you up and running with Sage with this text.

You can quickly reference Sage documentation with the `?` operator. You may also view the source code with the `??` operator.

```
Set?
```

```
Set??
```

```
factor?
```

```
factor??
```

1.9 Run Sage in the browser

The easiest way to get started is by running SageMath online. However, if you do not have reliable internet, you can also install the software locally on your own computer. Begin your journey with SageMath by following these steps:

1. Navigate to the SageMath website[1]
2. Click on Sage on CoCalc[2]
3. Create a CoCalc account[3]
4. Go to Your Projects[4] on CoCalc and create a new project.
5. Start your new project and create a new worksheet. Choose the SageMath Worksheet option.
6. Enter SageMath code into the worksheet. Try to evaluate a simple expression and use the worksheet like a calculator. Execute the code by clicking `Run` or using the shortcut `Shift + Enter`. We will learn more ways to run code in the next section.
7. Save your worksheet as a PDF for your records.

[1] wiki.sagemath.org/quickref
[2] doc.sagemath.org/html/en/reference/
[3] doc.sagemath.org/html/en/tutorial/
[4] doc.sagemath.org/html/en/index.html
[1] https://www.sagemath.org/
[2] https://cocalc.com/features/sage
[3] https://cocalc.com/auth/sign-up
[4] https://cocalc.com/projects

8. To learn more about SageMath worksheets, refer to the documentation[5]

9. Alternatively, you can run Sage code in a Jupyter Notebook[6] for some additional features.

10. If you are feeling adventurous, you can install Sage[7] and run it locally on your own computer. Keep in mind that a local install will be the most involved way to run Sage code.

[5] https://doc.cocalc.com/sagews.html
[6] doc.cocalc.com/jupyter-start.html
[7] doc.sagemath.org/html/en/installation/index.html

References

Most of the content in this book is based on the Mathematics lectures taught by Professor Hellen Colman at Wilbur Wright College. We focused our efforts on creating original work, and we drew inspiration from the following sources:

SageMath, the Sage Mathematics Software System (Version 10.2), The Sage Developers, 2024, https://www.sagemath.org.Beezer, Robert A., et al. The PreTeXt Guide. Pretextbook.org, 2024, https://pretextbook.org/doc/guide/html/guide-toc.html.Zimmermann, Paul. Computational Mathematics with SageMath. Society For Industrial And Applied Mathematics, 2019.

Colophon

PDF Download : This book was authored in PreTeXt and is available for free download in PDF format by clicking here[8].

[14] github.com/boughrira/template-math-with-sage/blob/release/output/print/manuscript.pdf

Index