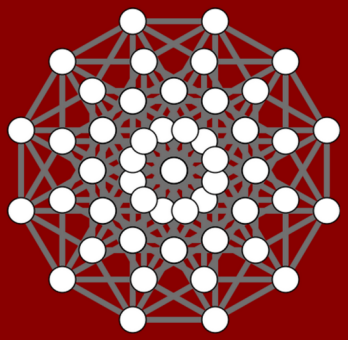

Zunaid Ahmed
Hellen Colman
Samuel Lubliner

Learn Math with open-source software

Discrete Math with SageMath

Discrete Math with SageMath

Learn math with open-source software

Discrete Math with SageMath

Learn math with open-source software

Zunaid Ahmed, Hellen Colman, Samuel Lubliner
City Colleges of Chicago

June 16, 2025

Website: GitHub Repository[1]

[1] github.com/SageMathOER-CCC/sage-discrete-math
[2] creativecommons.org/licenses/by/4.0

Preface

This book was written by undergraduate students at Wright College who were enrolled in my Math 299 class, Writing in the Sciences.

For many years, I have been teaching Discrete Math using the open source mathematical software SageMath. Despite the fabulous capabilities of this software, students were often frustrated by the lack of specific documentation geared towards beginning undergrad students in Discrete Math.

This book was born out of this frustration and the desire to make our own contribution to the Open Education movement, from which we have benefitted greatly. In the context of Open Pedagogy, my students and I ventured into a challenging learning experience based on the principles of freedom and responsibility. Each week, students wrote a chapter of this book. They found the topics and found their voice. We critically analyzed their writing, and they edited and edited again and again. They wrote code, tested it and polished it. In the process, we all learned so much about Sage, and we found some bugs in the software that are now in the process of being fixed thanks to its very active community of developers.

The result is the book we dreamed of having when we first attempted Discrete Math with Sage.

Our book is intended to provide concise and complete instructions on how to use different Sage functions to solve problems in Discrete Math. Our goal is to streamline the learning process, helping students focus more on mathematics and reducing the friction of learning how to code. Our textbook is interactive and designed for all math students, regardless of programming experience. Rooted in the open education philosophy, our textbook is, and always will be, free for all.

I am very proud of the work of my students and hope that this book will serve as inspiration for other students to take ownership of a commons-based education. Towards a future where higher education is equally accessible to all.

Hellen Colman
Chicago, May 2024

Acknowledgements

We would like to acknowledge the following peer-reviewers:

- Moty Katzman, University of Sheffield
- Ken Levasseur, University of Massachusetts Lowell
- Vartuyi Manoyan, Wright College

We would like to acknowledge the following proof-readers:

- Ted Jankowski, Wright College
- Justin Lowry, Wright College
- Yolanda Nieves, Wright College
- Fabio Re, Rosalind Franklin University
- Tineka Scalzo, Wright College

The making of this book is supported in part by the Wright College Open Educational Resource Expansion grant from the Secretary of State/Illinois State Library.

From the Student Authors

This textbook is a testament to our collaborative spirit and the Open Education movement, aiming to make higher education accessible to all by providing approachable resources for students to learn open-source mathematics software.

The creation of this textbook was a joint effort by a dedicated and inspirational team. The quality of our work reflects our collective contributions and enthusiasm.

We would like to acknowledge Hellen Colman, Professor of Mathematics at Wright College, our co-author, and our guiding star. Her mathematical expertise ensured the accuracy and relevance of our material. Inspired by her teaching and Discrete Math lectures at the City Colleges of Chicago-Wilbur Wright College, this project owes much to her mentorship and support. Her encouragement and trust have been invaluable, shaping our perspectives and approaches from our Discrete Math course to this OER development.

We extend our heartfelt gratitude to Ken Levasseur for his invaluable guidance and expertise in creating open-source textbooks. His contributions have significantly enhanced the quality and accessibility of our work.

A special thanks is due to Tineka Scalzo, Wright College librarian, for her valuable advice on publishing and copyright issues. Her insights have been instrumental in helping us navigate the complexities of academic publishing.

We also express our gratitude to the many talented developers and mathematicians within the open-source communities. The PreTeXt community played an essential role in our authoring process, while the SageMath community provided crucial subject matter expertise. We are very grateful to everyone who has worked to develop Sage and to the creators of PreTeXt.

We would also like to thank our peer reviewers and proofreaders, whose meticulous attention to detail ensured the clarity and quality of this textbook. Your contributions have been instrumental in bringing this project to fruition.

Finally, we express our deepest gratitude to all the contributors who made this project possible. Your dedication and collaborative spirit have made a lasting impact on this work and the field of open education.

Zunaid Ahmed and Samuel Lubliner

Authors and Contributors

Zunaid Ahmed
Computer Engineering
Truman College
zunaid.ahmed@hotmail.com

Hellen Colman
Math Department
Wright College
hcolman@ccc.edu

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

Allaoua Boughrira (Contrib.)
Mathematics
Wright College
a.boughrira@gmail.com

Michael Kattner (Contrib.)
Mathematics
Wright College
MDKattner@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 ways to run Sage, including SageMathCell, CoCalc, and a local installation. The simplest way to start is by using the SageMathCell embedded in this book. We will also cover how to use CoCalc, a cloud-based platform for running Sage in a collaborative environment.

Sage is a free, open-source mathematics software system that integrates various open-source math packages[1]. This chapter introduces Sage's syntax, data types, variables, and debugging techniques. Our goal is to equip you with the foundational knowledge to explore mathematical problems and programming concepts in an intuitive and practical manner.

Join us as we explore the capabilities of Sage!

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.

```
# This is an empty cell that you can use to type code
# and run Sage commands. These lines here are an example
# of comments for the reader and are ignored by Sage.
```

Note that these Sage cells allow the user to experiment freely with any of the Sage- supported commands. The content of the cells can be altered at runtime (on the live web version of the book) and executed in real-time on a remote Sage server. Users can modify the content of cells and execute any other Sage commands to explore various mathematical concepts interactively.

1.1.1 Sage as a Calculator

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

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

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `**` or `^` (Exponentiation)
- `/` (Division)
- `//` (Integer division)
- `%` (Modulo - remainder of division)

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

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

Try the following examples:

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

```
100 - 1
```

```
3*4
```

Sage supports two exponentiation operators:

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

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

Division in Sage:

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

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

Integer division and modulo:

```
5 // 3  # Returns the quotient
```

```
5 % 3   # Returns the remainder
```

1.1.2 Variables and Names

Variables store values in the computer's memory. This includes value of an expression to a variable. Use the assignment operator `=` to assign a value to a variable. The variable name is on the left side, and the value is on the right. Unlike the expressions above, an assignment does not display anything. To view a variable's value, type its name and run the cell.

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

When choosing variable names, follow these rules for valid identifiers:

- Identifiers cannot start with a digit.
- Identifiers are case-sensitive.
- - Letters (`a-z`, `A-Z`)
  - Digits (`0-9`)
  - Underscore (_)
- Do not use spaces, hyphens, punctuation, or special characters while naming your identifiers.
- Do not use reserved keywords as variable names.

Python has a set of reserved keywords that cannot be used 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`.

To check if a word is a reserved keyword, use the `keyword` module in Python.

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

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

1.2 Data Types

In computer science, **Data types** define the properties of data, and consequently the behavior of operations on these data types. Since Sage is built on Python, it naturally inherits all Python's built-in data types. Sage does also introduces new additional and custom classes and specific data types better-suited and optimized for mathematical computations.

Let's check the type of a simple integer in Sage.

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

The `type()` function confirms that `2` is an instance of Sage's **Integer** class.

Sage supports **symbolic** computation, where it retains and preserves the actual value of a math expression rather than evaluating them for approximated values.

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

Similarly, Sage uses to its own `rational` class when dividing two **Integers**.

```
k = 2/3
show(k)
type(k)
```

String: a `str` is a sequence of characters. Strings can be enclosed in single or double quotes.

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

Boolean: a (`bool`) data type represents one of only two possible values: `True` or `False`.

```
b = 5 in Primes()  # Check if 5 is a prime number
print(f"{b} is {type(b)}")
```

List: A mutable sequence or collection of items enclosed in square brackets `[]`. (an object is mutable if you can change its value after creating it).

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

Lists are indexed starting at `0`. The first element is at index zero, and can be accessed as follows:

```
l[0]
```

The `len()` function returns the number of elements in a list.

```
len(l)
```

Tuple: is an immutable sequence of items enclosed in parentheses `()`. (an object is immutable if you cannot change its value after creating it).

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

set (with a lowercase `s`): is a Python built-in type, which represents an unordered collection of unique items, enclosed in curly braces `{}`. The printout of the following example shows there are no duplicates.

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

In Sage, `Set` (with an uppercase `S`) extends the native Python's `set` with additional mathematical functionality and operations.

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

We start by defining a `list` using the square brackets `[]`. Then, Sage `Set()` function removes any duplicates and provides the mathematical set operations. Even though Sage supports Python sets, we will use Sage `Set` for the added features. Be sure to define `Set()` with an uppercase `S`.

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

A **Dictionary** is a collection of key-value pairs, enclosed in curly braces `{}`.

```
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 improve the dictionary readability.

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

1.3 Flow Control Structures

When writing programs, we want to control the flow of execution. Flow control structures allow your code to make decisions or repeat actions based on conditions. These structures are part of Python and work the same way in Sage. There are three primary types of flow control structures:

- **Assignment** statements store values in *variables*. These let us reuse results and build more complex expressions step by step. An assignment is performed using the `=` operator as discussed earlier (see Subsection 1.1.2). Note that Sage also supports compound assignment operators like `+=`, `-=`, `*=`, `/=`, and `%=` which combine assignment with basic arithmetic operations (addition, subtraction, multiplication, division and modulus).
- **Branching** uses *conditional statements* like `if`, `elif`, and `else` to execute different blocks of code based on logical tests.

- **Loops** such as `for` and `while` let us iterate over some data structures and also repeat blocks of code multiple times. This is useful when processing sequences, performing computations, or automating repetitive tasks.

These core concepts apply to almost every programming language and are fully supported in Sage through its Python foundation.

Notes. Sage uses the same control structures as Python, so most Python syntax for logic and repetition will work seamlessly in Sage.

1.3.1 Conditional Statements

The `if` statement lets your program execute a block of code only when a condition is true. You can add `else` and `elif` clauses to cover additional conditions.

```
x = 7
if x % 2 == 0:
    print("x is even")
elif x % 3 == 0:
    print("x is divisible by 3")
else:
    print("x is odd and not divisible by 3")
```

Use indentation to define blocks of code that belong to the `if`, `elif`, or `else` clauses. Just like in Python, the indentation is significant and is used to define code blocks.

1.3.2 Iteration

Iteration is a programming technique that allows us to efficiently repeat 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.

Here is a basic example of a `for` loop:

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

By default, `range(n)` starts at 0. To specify a different starting value, provide two arguments:

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

You can also define a step value to control the increment:

```
# Prints numbers from 30 to 90, stepping by 9
for i in range(30, 90, 9):
    print(i)
```

1.3.2.1 List Comprehension

List comprehension is a concise way to create lists. Unlike Python's `range()`, Sage's list comprehension syntax includes the ending value in a range.

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

You can also filter elements using a condition. Below, we create a list containing only the cubes of even numbers:

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

1.3.3 Other Flow Control Structures

In addition to `if` statements, Sage supports other common Python control structures:

- The `while` loops repeats a block of code while a condition remains true.
- The `break` statement terminates and exit a loop early.
- The `continue` statement skips the rest of the loop body and jump to the next iteration.
- The `pass` statement serves as a placeholder for future code to be added later, or to tell Sage do nothing (useful for example when we want to catch an exception so that the program does not crash, yet choose no to do anything with the exception object).

We will see examples on how to use these statements later on in the book. Here is a quick example of a `while` loop that prints out the numbers from 0 to 4:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

1.4 Defining Functions

Sage comes with many built-in functions. While Math terminology is not always standard, be sure to refer to the documentation to understand the exact functionality of these built-in functions and know how to use them. You can also define custom functions to suit your specific needs. 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. In this section, we'll explore how to define custom functions and use them.

To define a custom function in Sage, use the `def` keyword followed by the function name and the function's arguments. The body of the function is indented, and it should contain a `return` statement that outputs a value. Note that the function definition will only be stored in memory after executing the cell. You won't see any output when defining the function, but once it is defined, you can use it in other cells. If the cell is successfully executed, you

will see a green box underneath it. If the box is not green, run the cell again to define the function.

A simple example of defining a function is one that returns the n^{th} (0-indexed) row of Pascal's Triangle. Pascal's Triangle is a triangular array of numbers where each number is the sum of the two numbers directly above it.

Here's a function definition that computes a specific row of Pascal's Triangle. You need execute the cell to store the function in memory. You can only call the `pascal_row()` function once the definition has been executed. If you attempt to use the function without defining it first, you will receive a `NameError`.

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

After defining the function above, let's try calling it. :

```
pascal_row(5)
```

Sage functions can sometimes produce unexpected results if given improper input. For instance, passing a string or a decimal value into the function will raise a `TypeError`:

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

However, if you pass a negative integer, the function will silently return an empty list. This lack of error handling can lead to unnoticed errors or unexpected behaviors that are difficult to debug, so it is essential to incorporate input validation. Let's add a `ValueError` to handle negative input properly:

```
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)]
```

With the updated function definition above, try calling the function again with a negative integer. You will now receive an informative error message rather than an empty list:

```
pascal_row(-5)
```

Functions can also include a `docstring` in the function definition to describe its purpose, inputs, outputs, and any examples of usage. The `docstring` is a string that appears as the first statement in the function body. This documentation can be accessed using the `help()` function or the `?` operator.

```
def pascal_row(n):
    """
␣␣␣␣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, you can view the `docstring` by calling the `help()` function on the function name:

```
help(pascal_row)
```

Alternatively, you can access the source code using the `??` operator:

```
pascal_row??
```

To learn more on code style conventions and writing documentation strings, refer to the General Conventions article in the Sage Developer's Guide.

1.5 Object-Oriented Programming

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

Classes act as blueprints that define the structure and behavior of objects. A class specifies the **attributes** and **methods** of an object. - An **attribute** is a variable that stores information about the object. - A **method** is a function that interacts with or modifies the object. Although you can create custom classes, many useful classes are already available in Sage and Python, such as those for integers, lists, strings, and graphs.

1.5.1 Objects in Sage

In Python and Sage, almost everything is an object. When assigning a value to a variable, the variable references an object. The `type()` function allows us

to check an object's class.

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

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

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

1.5.2 Dot Notation and Methods

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

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

Here, `'y'` is passed as a **parameter** to the `append()` method, adding it to the end of the list. The `list` class provides many other methods for interacting with lists.

1.5.3 Sage's Set Class

While `list` is a built-in Python class, Sage provides specialized classes for mathematical objects. One such class is `Set`, which we will explore later on in the next chapter.

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

The `Set` class in Sage provides attributes and methods specifically designed for working with sets. While OOP might seem abstract at first, it will become clearer as we explore more and dive deeper into Sage features. Sage's built-in classes offer a structured way to represent data and perform powerful mathematical operations. In the next chapters, we'll see how Sage utilizes OOP principles and its built-in classes to perform mathematical operations.

1.6 Display Values

Sage provides multiple 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 offers functions to format and display output in different styles.

Sage automatically displays the value of the last line in a cell unless a specific function is used for output. Here are some key functions for displaying values:

- `print()` displays the value of the expression inside the parentheses as plain text.
- `pretty_print()` displays rich text as typeset LaTeX output.

- `show()` is an alias for `pretty_print()` and provides additional functionality for graphics.
- `latex()` returns the raw LaTeX code for the given expression, which then can be used in LaTeX documents.
- `%display latex` enables automatic rendering of all output in LaTeX format.
- While Python string formatting is available and can be used, it may not reliably render rich text or LaTeX expressions due to compatibility issues.

Let's explore these display methods in action.

Typing a string directly into a Sage cell displays it with quotes.

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

Using `print()` removes the quotes.

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

The `show()` function formats mathematical expressions for better readability.

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

To display multiple values in a single cell, use `show()` for each one.

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

The `latex()` function returns the raw LaTeX code for an expression.

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

In Jupyter notebooks or SageMathCell, you can set the display mode to LaTeX using `%display latex`.

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

Once set, all expressions onward will continue to be rendered in LaTeX format until the display mode is changed.

```
ZZ
```

To return to the default output format, use `%display plain`.

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

```
ZZ
```

1.7 Debugging

Error messages are an inevitable part of programming. When you encounter one, read it carefully for clues about the cause. Some messages are clear and descriptive, while others may seem cryptic. With practice, you will develop valuable skills debugging your code and resolving errors

Note that not all errors result in error messages. **Logical errors** occur when the syntax is correct, but the program does not produce the expected result. Usually, these are a bit harder to trace.

Remember, mistakes are learning opportunities —everyone makes them! Here are some useful debugging tips:

- Read the error message carefully —it often provides useful hints.
- Consult the documentation to understand the correct syntax and usage.
- Google-search the error message —it's likely that others have encountered the same issue.
- Check SageMath forums for previous discussions.
- Take a break and return with a fresh perspective.
- Ask the Sage community if you're still stuck after trying all the above steps.

Let's dive in and make some mistakes together!

A **SyntaxError** usually occurs when the code is not written according to the language rules.

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

Why didn't this print `Hello, World!` to the console? The error message indicates a `SyntaxError: invalid decimal literal`. The issue here is the invalid variable name. Valid *identifiers* must:

- Start with a letter or an underscore (never with a number).
- Avoid any special characters other than the underscores.

Let's correct the variable name:

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

A **NameError** occurs when a variable or function is referenced before being defined.

```
print(Hi)
```

Sage assumes `Hi` is a variable, but we have not defined it yet. There are two ways to fix this:

- Use quotes to indicate that `Hi` is a string.

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

- Alternatively, if we intended `Hi` to be a variable, then we must define it before first use.

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

Reading the documentation is essential to understanding the proper use of methods. If we incorrectly use a method, we will likely get a `NameError` (as seen above), an `AttributeError`, a `TypeError`, or `ValueError`, depending on the mistake.

Here is another 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)
```

An **AttributeError** occurs when an invalid method is called on an object.

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

The `len()` function must be used separately rather than as a method of a list.

```
len(l)
```

A **TypeError** occurs when an operation or function is applied to an *incorrect* data type.

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

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

```
l.extend([4, 5])
print(l)
```

A **ValueError** occurs when an operation receives an argument of the correct type but with an invalid value.

```
factorial(-5)
```

Although the resulting error message is lengthy, the last line informs us that Factorials are only defined for non-negative integers.

```
factorial(5)
```

A **Logical error** does not produce an error message but leads to incorrect results.

Here, assuming your task is to print the numbers from 1 to 10, and you mistakenly write the following code:

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

This instead will print the numbers 0 to 9 (because the start is inclusive but not the stop). If we want numbers 1 to 10, we need to adjust the range.

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

To learn more, check out the CoCalc article[1] about the top mathematical syntax errors in Sage.

1.8 Documentation

Sage offers a wide range of features. To explore what Sage can do, check out the Quick Reference Card[1] and the Reference Manual[2] for detailed information.

The tutorial[3] offers a useful overview for getting familiar with Sage and its functionalities.

You can find Sage documentation[4] at the official website. At this stage, reading the documentation is optional, but we will guide you through getting started with Sage in this book.

To quickly reference Sage documentation, use the `?` operator in Sage. This can be a useful way to get immediate help with functions or commands. You can also view the source code of functions using the `??` operator.

```
Set?
```

```
Set??
```

```
factor?
```

```
factor??
```

1.9 Miscellaneous Features

Sage is feature rich, and the following is a brief introduction of some of its miscellaneous features. Keep in mind that the primary goal of this book is to introduce Sage software and demonstrate how it can be used to experiment with discrete math concepts within Sage environment.

Sage is used here interactively, and mainly covering the basics. Having an understanding of any of the commands presented in this section would be crucial for working on a production-grade project with complex mathematical models (e.g. handling large datasets). In such cases, it would be more appropriate to use these commands within a standalone Sage environment. These commands are presented here just for the sake of completeness.

1.9.1 Reading and Writing Files in Sage

Sage provides various ways to handle input and output (I/O) operations.

This subsection explores writing data to files and importing data from files.

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

Sage allows reading from and writing to files using standard Python file-handling functions.

Writing to a file:

```
with open("output.txt", "w") as file:
    file.write("Hello, Sage!")
```

Reading from a file:

```
with open("output.txt", "r") as file:
    content = file.read()
    print(content)  # Output: Hello, Sage!
```

1.9.2 Executing Shell Commands in Sage

Sage allows executing shell commands directly using the `!` operator (prefix the shell command to be executed).

Listing the content of the current directory showing the file that we just created:

```
!ls -la
```

1.9.3 Importing and Exporting Data (CSV, JSON, TXT)

Sage supports structured data formats such as CSV and JSON.

Generating a CSV file using shell command:

```
!printf
    "Name,Age,Country\nAlice,25,USA\nBob,30,UK\nCharlie,28,Canada\n"
    > data.csv
```

Reading a CSV file in Sage:

```
import csv

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

1.9.4 Using External Libraries in Sage

Sage allows using external Python libraries to do advance calculation or access and communicate over a network (urllib.request library) .

Using NumPy for numerical computations:

```
import numpy as np
array = np.array([1, 2, 3])
print(array)  # Output: [1 2 3]
```

1.10 Run Sage in the browser

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

1. Navigate to Sage 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 Sage-Math Worksheet option.
6. Enter Sage 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.
8. To learn more about Sage worksheets, refer to the documentation[5].
9. Alternatively, you can run Sage code in a Jupyter Notebook[6] for 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. When using Sage locally, commands to display graphics will create and then open a temporary file, which can be saved permanently through the software used to view it.

[1] https://www.sagemath.org/
[2] https://cocalc.com/features/sage
[3] https://cocalc.com/auth/sign-up
[4] https://cocalc.com/projects
[5] https://doc.cocalc.com/sagews.html
[6] doc.cocalc.com/jupyter-start.html
[7] doc.sagemath.org/html/en/installation/index.html

Chapter 2

Set Theory

This chapter presents the study of set theory with Sage, starting with a description of the `Set()` function, its variations, and how to use it to calculate the basic set operations.

2.1 Creating Sets

2.1.1 Set Definitions

To construct a set, encase the elements within square brackets `[]`. Then, pass this `list` as an argument to the `Set()` function. It's important to note that the `S` in `Set()` should be uppercase to define a Sage set. In a set, each element is unique.

```
M = Set(["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
    "Aug", "Sep", "Oct", "Nov", "Dec"])
show(M)
```

Notice that the months in set M do not appear in the same order as when you created the set. Sets are unordered collections of elements.

We can ask Sage to compare two sets to see whether or not they are equal. We can use the `==` operator to compare two values. A single equal sign `=` and double equal sign `==` have different meanings.

The **equality operator** `==` is used to ask Sage if two values are equal. Sage compares the values on each side of the operator and returns the Boolean value. The `==` operator returns `True` if the sets are equal and `False` if they are not equal.

The **assignment operator** `=` assigns the value on the right side to the variable on the left side.

```
M = Set(["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
    "Aug", "Sep", "Oct", "Nov", "Dec"])
M_duplicates = Set(["Jan", "Jan", "Jan", "Feb", "Feb",
    "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
    "Oct", "Nov", "Dec"])

# The Set function eliminates duplicates
M == M_duplicates
```

2.1.2 Set Builder Notation

Instead of explicitly listing the elements of a set, we can use a set builder notation to define a set. The set builder notation is a way to define a set by describing the properties of its elements. Here, we use the Sage `srange` instead of the Python `range` function for increased flexibility and functionality.

```
# Create a set of even numbers between 1 and 10
A = Set([x for x in srange(1, 11) if x % 2 == 0])
A
```

Iteration is a way to repeat a block of code multiple times and can be used to automate repetitive tasks. We could have created the same set by typing `A = Set([2, 4, 6, 8, 10])`. Imagine if we wanted to create a set of even numbers between 1 and 100. It would be much easier to use iteration.

```
B = Set([x for x in srange(1, 101) if x % 2 == 0])
B
```

2.1.3 Subsets

To list all the subsets included in a set, we can use the `Subsets()` function and then use a `for` loop to display each subset.

```
W = Set(["Sun", "Cloud", "Rain", "Snow", "Tornado",
    "Hurricane"])
subsets_of_weather = Subsets(W)

subsets_of_weather.list()
```

2.1.4 Set Membership Check

Sage allows you to check whether an element belongs to a set. You can use the `in` operator to check membership, which returns `True` if the element is in the set and `False` otherwise.

```
"earthquake" in W
```

We can check if $Severe = \{Tornado, Hurricane\}$ is a subset of W by using the `issubset` method.

```
Severe = Set(["Tornado", "Hurricane"])
Severe.issubset(W)
```

When we evaluate `W.issubset(Severe)`, Sage returns `False` because W is not a subset of $Severe$.

```
W.issubset(Severe)
```

2.2 Cardinality

To find the cardinality of a set, we use the `cardinality()` function.

```
A = Set([1, 2, 3, 4, 5])
A.cardinality()
```

Alternatively, we can use the Python `len()` function. Instead of returning a Sage `Integer`, the `len()` function returns a Python `int`.

```
A = Set([1, 2, 3, 4, 5])
len(A)
```

In many cases, using Sage classes and functions will provide more functionality. In the following example, `cardinality()` gives us a valid output while `len()` does not.

```
P = Primes()
P.cardinality()
```

```
# This results in an error because the
# Python len() function is not defined for the primes class
P = Primes()
P.len()
```

2.3 Operations on Sets

2.3.1 Union of Sets

There are two distinct methods available in Sage for calculating unions.

Suppose $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6\}$. We can use the `union()` function to calculate $A \cup B$.

Notes. The union operation is relevant in real-world scenarios, such as merging two distinct music playlists into one. In this case, any song that appears in both playlists will only be listed once in the merged playlist.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A.union(B)
```

Alternatively, we can use the `|` operator to perform the union operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A | B
```

2.3.2 Intersection of Sets

Similar to union, there are two methods of using the intersection function in Sage.

Suppose $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6\}$. We can use the `intersection()` function to calculate $A \cap B$.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A.intersection(B)
```

Alternatively, we can use the `&` operator to perform the intersection operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A & B
```

2.3.3 Difference of Sets

Suppose $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6\}$. We can use the `difference()` function to calculate the difference between sets.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A.difference(B)
```

Alternatively, we can use the `-` operator to perform the difference operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
A - B
```

2.3.4 Multiple Sets

When performing operations involving multiple sets, we can repeat the operations to get our results. Here is an example:

Suppose $A = \{1, 2, 3, 4, 5\}$, $B = \{3, 4, 5, 6\}$ and $C = \{5, 6, 7\}$. To find the union of all three sets, we repeat the `union()` function.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
C = Set([5, 6, 7])
A.union(B).union(C)
```

Alternatively, we can repeat the `|` operator to perform the union operation.

```
A = Set([1, 2, 3, 4, 5])
B = Set([3, 4, 5, 6])
C = Set([5, 6, 7])
A | B | C
```

The `intersection()` and `difference()` functions can perform similar chained operations on multiple sets.

2.3.5 Complement of Sets

Let $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the universal set. Given the set $A = \{1, 2, 3, 4, 5\}$. We can use the `difference()` function to find the complement of A.

```
U = Set([1, 2, 3, 4, 5, 6, 7, 8, 9])
A = Set([1, 2, 3, 4, 5])
U.difference(A)
```

Alternatively, we can use the `-` operator.

```
U = Set([1, 2, 3, 4, 5, 6, 7, 8, 9])
A = Set([1, 2, 3, 4, 5])
```

```
U - A
```

2.3.6 Cartesian Product of Sets

Suppose $A = \{1, 2, 3, 4, 5\}$ and $D = \{x, y\}$. We can use the `cartesian_product()` and `Set()` functions to display the Cartesian product $A \times D$.

```
A = Set([1, 2, 3, 4, 5])
D = Set(['x', 'y'])
Set(cartesian_product([A, D]))
```

Alternatively, we can use the `.` notation to find the Cartesian product.

```
A = Set([1, 2, 3, 4, 5])
D = Set(['x', 'y'])
Set(A.cartesian_product(D))
```

2.3.7 Power Sets

The power set of the set V is the set of all subsets, including the empty set $\{\emptyset\}$ and the set V itself. Sage offers several ways to create a power set, including the `Subsets()` and `powerset()` functions. First, we will explore the `Subsets()` function. The `Subsets()` function is more user-friendly due to the built-in `Set` methods. Next, we will examine some limitations of the `Subsets()` function. We introduce the `powerset()` function as an alternative for working with advanced sets not supported by `Subsets()`.

The `Subsets()` function returns all subsets of a finite set in no particular order. Here, we find the power set of the set of vowels and view the subsets as a `list` where each element is a `Set`.

```
V = Set(["a", "e", "i", "o", "u"])
S = Subsets(V)
list(S)
```

We can confirm that the power set includes the empty set.

```
Set([]) in S
```

We can also confirm that the power set includes the original set.

```
V in S
```

The `cardinality()` method returns the total number of subsets.

```
S.cardinality()
```

There are limitations to the `Subsets()` function. For example, the `Subsets()` function does not support non-hashable objects.

About hashable objects:

- A hashable object has a hash value that never changes during its lifetime.
- A hashable object can be compared to other objects.
- Most of Python's immutable built-in objects are hashable.
- Mutable containers (lists or dictionaries) are not hashable.

- Immutable containers (tuples) are only hashable if their elements are hashable.

You will see an `unhashable type` error message when trying to create `Subsets` of a list containing a list. The `powerset()` function returns an iterator over the `list` of all subsets in no particular order. The `powerset()` function is ideal when working with non-hashable objects.

```
N = [1, [2, 3], 4]
list(powerset(N))
```

The `powerset()` function supports infinite sets. Let's generate the first 7 subsets from the power set of integers.

```
P = powerset(ZZ)

i = 0
for subset in P:
    print(subset)
    i += 1
    if i == 7:
        break
```

While the `Subsets()` function can represent infinite sets symbolically, it is not practical.

```
P = Subsets(ZZ)
P
```

Observe the `TypeError` message when trying to retrieve a random element from `Subsets(ZZ)`

```
P.random_element()
```

Pay close attention to the capitalization of function names. There is a difference between the functions `Subsets()` and `subsets()`. Notice the lowercase `s` in `subsets()`, which is an alias for `powerset()`.

2.3.8 Viewing Power Sets

Power sets can contain many elements. The powerset of the set R contains elements 128 elements.

```
R = Set(["red", "orange", "yellow", "green", "blue",
    "indigo", "violet"])
S = Subsets(R)
S.cardinality()
```

If we only want to view part of the power set, we can specify a range of elements with a technique called slicing. With slicing we convert the set into a list with order. For example, here are the first 5 elements of the list made from the elements of the power set.

```
S.list()[:5]
```

Now, let's retrieve the following 5 elements of the power set.

```
# Slicing to get elements from index 5 to 9
S.list()[5:10]
```

Chapter 3

Combinatorics

Counting techniques arise naturally in computer algebra as well as in basic applications in daily life. This chapter covers the treatment of the enumeration problem in Sage, including counting combinations, counting permutations, and listing them.

3.1 Combinatorics

3.1.1 Factorial Function

The factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n.

Compute the factorial of 5:

```
factorial(5)
```

3.1.2 Combinations

The combination (n, k) is an unordered selection of k objects from a set of n objects.

Notes. Use combinations when order does not matter, such as determining possible Poker hands. The order in which a player holds cards does not affect the kind of hand. For example, the following hand is a royal flush: 10, J, Q, K, A. The following hand is also a royal flush: A, K, J, 10, Q.

Calculate the number of ways to choose 3 elements from a set of 5:

```
Combinations(5, 3).cardinality()
```

List the combinations:

```
Combinations(5, 3).list()
```

Note that the function arranges the numbers from 0 to $n - 1$ for its listing of combinations.

The `binomial()` function provides an alternative method to compute the number of combinations.

```
binomial(5, 3)
```

3.1.3 Permutations

A permutation (n, k) is an ordered selection of k objects from a set of n objects.

Notes. Use permutations in situations where order does matter, such as when creating passwords. Longer passwords have more permutations, making them more challenging to guess by brute force.

To calculate the number of ways to choose 3 elements from a set of 5 when the order matters, use the `Permutations()` method.

```
Permutations(5, 3).cardinality()
```

List the permutations:

```
Permutations(5, 3).list()
```

Note that the function arranges the numbers from 1 to n for its listing of permutations.

When $n = k$, we can calculate permutations of n elements.

Calculate the number of permutations of a set with 3 elements:

```
Permutations(3).cardinality()
```

List the permutations:

```
Permutations(3).list()
```

The following is an example of permutations of specified elements:

```
A = Permutations(['a', 'b', 'c'])

A.list()
```

Choose 2:

```
A = Permutations(['a', 'b', 'c'], 2)

A.list()
```

Chapter 4

Logic

In this chapter, we introduce different ways to create Boolean formulas using the logical functions `not`, `and`, `or`, `if then`, and `iff`. Then, we show how to ask Sage to create a truth table from a formula and determine if an expression is a contradiction or a tautology.

4.1 Logical Operators

In Sage, the logical operators are AND `&`, OR `|`, NOT `~`, conditional `->`, and biconditional `<->`.

Name	Sage Operator	Mathematical Notation
AND	`&`	$\wedge$
OR	`|`	$\vee$
NOT	`~`	$\neg$
Conditional	`->`	$\rightarrow$
Biconditional	`<->`	$\leftrightarrow$

4.1.1 Boolean Formula

Sage's `propcalc.formula()` function allows for the creation of Boolean formulas using variables and logical operators. We can then use `show` function to display the mathematical notations.

```
A = propcalc.formula('(p␣&␣q)␣|␣(~p)')
show(A)
```

4.2 Truth Tables

The `truthtable()` function in Sage generates the truth table for a given logical expression.

Notes. Truth tables aid in the design of digital circuits.

```
A = propcalc.formula('p␣->␣q')
A.truthtable()
```

An alternative way to display the table with better separation and visuals would be to use `SymbolicLogic()`, `statement()`, `truthtable()` and the `print_table()` functions.

```
A = SymbolicLogic()
B = A.statement('p -> q')
C = A.truthtable(B)
A.print_table(C)
```

The command `SymbolicLogic()` creates an instance for handling symbolic logic operations, while `statement()` defines the given statement. The `truthtable()` method generates a truth table for this statement, and `print_table()` displays it.

Expanding on the concept of truth tables, we can analyze logical expressions involving three variables. This provides a deeper understanding of the interplay between multiple conditions. The `truthtable()` function supports expressions with a number of variables that is practical for computational purposes, if the list of variables becomes too lengthy (such as extending beyond the width of a LaTeX page), the truth table's columns may run off the screen. Additionally, the function's performance may degrade with a very large number of variables, potentially increasing the computation time.

```
B = propcalc.formula('(p & q) -> r')
B.truthtable()
```

4.3 Analyzing Logical Equivalences

4.3.1 Equivalent Statements

When working with Sage symbolic logic, the `==` operator compares semantic equivalence.

```
h = propcalc.formula("x | ~y")
s = propcalc.formula("x & y | x & ~y | ~x & ~y")
h == s
```

Do not attempt to compare equivalence of truth tables.

```
# Warning:
# Even though these truth tables look identical,
# the comparison will return False.
h.truthtable() == s.truthtable()
```

However, we can compare equivalence of truth table `lists`.

```
h_list = h.truthtable().get_table_list()
s_list = s.truthtable().get_table_list()
h_list == s_list
```

4.3.2 Tautologies

A tautology is a logical statement that is always true. The `is_tautology()` function checks whether a given logical expression is a tautology.

Notes. Tautologies are relevant in the field of cybersecurity. Attackers exploit vulnerabilities by injecting SQL code that turns a `WHERE` clause into a tautology, granting unintended access to the system.

```
a = propcalc.formula('p␣|␣~p')
a.is_tautology()
```

4.3.3 Contradictions

In contrast to tautologies, contradictions are statements that are always false.

```
A = propcalc.formula('p␣&␣~p')
A.is_contradiction()
```

Chapter 5

Relations

In this chapter, we will explore the relationships between elements in sets, building upon the concept of the Cartesian product introduced earlier. We will begin by learning how to visualize relations using Sage. Then, we will introduce some new functions that can help us determine whether these relations are equivalence or partial order relations.

5.1 Introduction to Relations

A **relation** R from set A to set B is any subset of the Cartesian product $A \times B$, indicating that $R \subseteq A \times B$. We can ask Sage to decide if R is a relation from A to B. First, construct the Cartesian product $C = A \times B$. Then, build the set S of all subsets of C. Finally, ask if R is a subset of S.

Recall the Cartesian product consists of all possible ordered pairs (a, b), where $a \in A$ and $b \in B$. Each pair combines an element from set A with an element from set B.

In this example, an element in the set A relates to an element in B if the element from A is twice the element in B.

```
A = Set([1, 2, 3, 4, 5, 6])
B = Set([1, 2, 7])

CP = Set(cartesian_product([A, B]))
S = Subsets(CP)

R = Set([(a, b) for a in A for b in B if a==2*b])

print("R␣=", R)
print("Is␣R␣a␣relation␣from␣set␣A␣to␣set␣B?", R in S)
```

5.1.1 Relation Composition

Let R be a relation from a set A to a set B and S a relation from B to a set C. The composite of R and S is the relation consisting of the ordered pairs (a, c) where $a \in A$ and $c \in C$, and for which there is a $b \in B$ such that $(a, b) \in R$ and $(b, c) \in S$. We denote the composite of R and S by $S \circ R$.

We can also use Sage to compose relations.

```
A = Set([1, 2, 3, 4, 5, 6])
B = Set([1, 2, 7])
R = Set([(a, b) for a in A for b in B if a==2*b])

C = Set([7,8,9,10])
S = Set([(a,c) for a in A for c in C if a + c == 10])

SoR = Set([(r[0], s[1]) for r in R for s in S if r[1] ==
    s[0]])
show(SoR)
```

5.1.2 Relations On a Set

When $A = B$ we refer to the relation as a relation **on** A.

Consider the set $A = \{2, 3, 4, 6, 8\}$. Let's define a relation R on A such that aRb if and only if $a|b$ (a divides b). The relation R can be represented by the set of ordered pairs where the first element divides the second:

```
A = Set([2, 3, 4, 6, 8])

# Define the relation R on A: aRb iff a divides b
R = Set([(a, b) for a in A for b in A if a.divides(b)])

show(R)
```

5.2 Digraphs

A digraph, or directed graph, is a visual representation of a relation R on the set A. Every element in set A is shown as a node (vertex). An arrow from the node a to the node b represents the pair (a, b) on the relation R.

A digraph must be made from a ***list*** and not a `Set`.

```
# Define the set A
A = Set([2, 3, 4, 6, 8])

# Define the relation R on A: aRb iff a divides b
R = [(a, b) for a in A for b in A if a.divides(b)]

DiGraph(R, loops=true)
```

The circles at the nodes are the same as arrows from the node to itself.

We can add a title to the digraph with the `name` parameter.

Notes. Digraphs come in handy when relationships have a clear direction, like who follows who on social media or how academic papers cite one another.

```
DiGraph(R, loops=true, name="Look␣at␣my␣digraph")
```

If the digraph does not contain a relation from a node to itself, we can omit the `loops=true` parameter. If we happen to forgot to include the parameter when we need to, Sage will give us a descriptive error message.

```
# Define the set A
A = Set([2, 3, 4, 6, 8])

# Define the relation R on A: aRb iff a < b
```

```
R = [(a, b) for a in A for b in A if a < b]
DiGraph(R)
```

We can also define the digraph using pair notion for relations.

```
DiGraph([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])
```

Alternatively, we can define the digraph directly. The element on the left of the : is a node. The node relates to the elements in the list on the right of the :.

```
# 1 relates to 2, 3, and 4
# 2 relates to 3 and 4
# 3 relates to 4
DiGraph({1: [2, 3, 4], 2: [3, 4], 3: [4]})
```

5.3 Properties

A relation on A may satisfy certain properties:

- **Reflexive**: $aRa \, \forall a \in A$
- **Symmetric**: If aRb then $bRa \, \forall a, b \in A$
- **Antisymmetric**: If aRb and bRa then $a = b \, \forall a, b \in A$
- **Transitive**: If aRb and bRc then $aRc \, \forall a, b, c \in A$

So far, we have learned about some of the built-in Sage methods that come out of the box, ready for us to use. Sometimes, we may need to define custom functions to meet specific requirements or check for particular properties. We define custom functions with the `def` keyword. If you want to reuse the custom functions defined in this book, copy and paste the function definitions into your own Sage worksheet and then call the function to use it.

5.3.1 Reflexive

A relation R is reflexive if a relates to a for all elements a in the set A. This means all the elements relate to themselves.

```
A = Set([1, 2, 3])
R = Set([(1, 1), (2, 2), (3, 3), (1, 2), (2, 3)])
show(R)
```

Let's define a function to check if the relation R on set A is reflexive. We will create a set of (a, a) pairs for each element a in A and check if this set is a subset of R. This will return `True` if the relation is reflexive and `False` otherwise.

```
def is_reflexive_set(A, R):
    reflexive_pairs = Set([(a, a) for a in A])
    return reflexive_pairs.issubset(R)

is_reflexive_set(A, R)
```

If we are working with `DiGraphs`, we can use the method `has_edge` to check if the graph has a loop for each vertex.

```
def is_reflexive_digraph(A, G):
    return all(G.has_edge(a, a) for a in A)

A = [1, 2, 3]
R = [(1, 1), (2, 2), (3, 3), (1, 2), (2, 3)]

G = DiGraph(R, loops=True)

is_reflexive_digraph(A, G)
```

5.3.2 Symmetric

A relation is symmetric if a relates to b, then b relates to a.

```
def is_symmetric_set(relation_R):
    inverse_R = Set([(b, a) for (a, b) in relation_R])
    return relation_R == inverse_R

A = Set([1, 2, 3])

R = Set([(1, 2), (2, 1), (3, 3)])

is_symmetric_set(R)
```

We can check if a `DiGraph` is symmetric by comparing the edges of the graph with the `reverse` edges. In our definition of symmetry, we are only interested in the relation of nodes, so we set edge `labels=False`.

```
def is_symmetric_digraph(digraph):
    return digraph.edges(labels=False) ==
        digraph.reverse().edges(labels=False)

relation_R = [(1, 2), (2, 1), (3, 3)]

G = DiGraph(relation_R, loops=true)

is_symmetric_digraph(G)
```

5.3.3 Antisymmetric

When a relation is antisymmetric, the only case that a relates to b and b relates to a is when a and b are equal.

```
def is_antisymmetric_set(relation):
    for a, b in relation:
        if (b, a) in relation and a != b:
            return False
    return True

relation = Set([(1, 2), (2, 3), (3, 4), (4, 1)])

is_antisymmetric_set(relation)
```

While Sage offers a built-in `antisymmetric()` method for `Graphs`, it checks for a more restricted property than the standard definition of antisymmetry. Specifically, it checks if the existence of a path from a vertex x to a vertex y implies that there is no path from y to x unless $x = y$. Observe that while

the standard antisymmetric property forbids the edges to be bidirectional, the Sage antisymmetric property forbids cycles.

```
# Example with the more restricted
# Sage built-in antisymmetric method
# Warning: returns False

relation = [(1, 2), (2, 3), (3, 4), (4, 1)]

DiGraph(relation).antisymmetric()
```

Let's define a function to check for the standard definition of antisymmetry in a `DiGraph`.

```
def is_antisymmetric_digraph(digraph):
    for edge in digraph.edges(labels=False):
        a, b = edge
        # Check if there is an edge in both directions (a to
            b and b to a) and a is not equal to b
        if digraph.has_edge(b, a) and a != b:
            return False
    return True

relation = DiGraph([(1, 2), (2, 3), (3, 4), (4, 1)])

is_antisymmetric_digraph(relation)
```

5.3.4 Transitive

A relation is transitive if a relates to b and b relates to c, then a relates to c.

Let's define a function to check for the transitive property in a `Set`:

```
def is_transitive_set(A, R):
    for a in A:
        for b in A:
            if (a, b) in R:
                for c in A:
                    if (b, c) in R and not (a, c) in R:
                        return False
    return True

A = Set([1, 2, 3])

R = Set([(1, 2), (2, 3), (1, 3)])

is_transitive_set(A, R)
```

You may be tempted to write a function with a nested loop because the logic is easy to follow. However, when working with larger sets, the time complexity of the function will not be efficient. This is because we are iterating through the set A three times. We can improve the time complexity by using a dictionary to store the relation R. Alternatively, we can use built-in Sage `DiGraph` methods.

```
D = DiGraph([(1, 2), (2, 3), (1, 3)], loops=True)

D.is_transitive()
```

5.4 Equivalence

A relation on a set is called an **equivalence relation** if it is reflexive, symmetric, and transitive. The **equivalence class** of an element a in a set A is the set of all elements in A that are related to a by this relation, denoted by:

$$[a] = \{x \in A \mid xRa\}$$

Here, $[a]$ represents the equivalence class of a, comprising all elements in A that are related to a through the relation R. This illustrates the grouping of elements into equivalence classes.

Consider a set A defined as:

$$A = \{x \mid x \text{ is a person living in a given building}\}$$

```
# Define the set of people
A = Set(['p_1', 'p_2', 'p_3', 'p_4', 'p_5', 'p_6', 'p_7',
    'p_8', 'p_9', 'p_10'])
A
```

Create sets for the people living on each floor of the building:

```
import pprint

# Define the floors as a dictionary, mapping floor names to
    sets of people
floors = {
    'first_floor': Set(['p_1', 'p_2', 'p_3', 'p_4']),
    'second_floor': Set(['p_5', 'p_6', 'p_7']),
    'third_floor': Set(['p_8', 'p_9', 'p_10'])
}

pprint.pprint(floors)
```

Let R be the relation on A described as follows:

xRy iff x and y live in the same floor of the building.

```
# Define the relation R based on living on the same floor
R = Set([(x, y) for x in A for y in A if any(x in
    floors[floor] and y in floors[floor] for floor in
    floors)])
R
```

This relation demonstrates the properties of an equivalence relation:

Reflexive: A person lives in the same floor as themselves.

```
def is_reflexive_set(A, R):
    reflexive_pairs = Set([(a, a) for a in A])
    return reflexive_pairs.issubset(R)

is_reflexive_set(A, R)
```

Symmetric: If person a lives in the same floor as person b, then person b lives in the same floor as person a.

```
def is_symmetric_set(relation_R):
    inverse_R = Set([(b, a) for (a, b) in relation_R])
    return relation_R == inverse_R

is_symmetric_set(R)
```

Transitive: If person a lives in the same floor as person b and person b lives in the same floor as person c, then person a lives in the same floor as person c.

```
G = DiGraph(list(R), loops=true)
G.is_transitive()
```

5.5 Partial Order

A relation R on a set is a Partial Order (PO) $\prec$ if it satisfies the reflexive, antisymmetric, and transitive properties. A poset is a set with a partial order relation. For example, the following set of numbers with a relation given by divisibility is a poset.

```
A = Set([1, 2, 3, 4, 5, 6, 8])

R = [(a, b) for a in A for b in A if a.divides(b)]

D = DiGraph(R, loops=True)

plot(D)
```

A Hasse diagram is a simplified visual representation of a poset. Unlike a digraph, the relative position of vertices has meaning: if x relates to y, then the vertex x appears lower in the drawing than the vertex y. Self-loops are assumed and not shown. Similarly, the diagram assumes the transitive property and does not explicitly display the edges that are implied by the transitive property.

Notes. Partial orders and Hasse diagrams help analyze task dependencies in scheduling applications.

If R is a partial order relation on A, then the function `Poset((A, R))` computes the Hasse diagram associated to R.

```
A = Set([1, 2, 3, 4, 5, 6, 8])

R = [(a, b) for a in A for b in A if a.divides(b)]

P = Poset((A, R))

plot(P)
```

Moreover, the `cover_relations()` function shows the pairs depicted in the Hasse diagram after the previous simplifications.

```
P.cover_relations()
```

The `.has_bottom()` function tests for a bottom element of a poset.

```
P.has_bottom()
```

The same applies for the `.has_top()` function but with a top element.

```
P.has_top()
```

The bottom element of a poset, if one exists, can be found with the `.bottom()` attribute.

```
P.bottom()
```

Similarly, the top element of a poset, if one exists, can be found with the `.top()` attribute.

```
P.top()
```

Notice that P does not have a top element, causing nothing to be returned by `P.top()`.

5.6 Relations in Action

Imagine you are creating both a light-mode and a dark-mode for a brand's website and have been asked to use complementary pairs of colors from the brand's color palette for the foreground of the website. Sage's `colors` dictionary and related attributes can be used to pick out colors for both modes.

Here is the `Set` of the brand's color palette.

```
C = Set(['paleturquoise', 'maroon', 'teal', 'green',
    'violet', 'firebrick'])
for c in C:
    print(c)
    circle((0,0),1, color=c,
        fill=true).show(figsize=[0.5,0.5], axes=False)
```

We can then check if all of these color names are keys in the `colors` dictionary by converting `colors` into a `Set`.

```
C.issubset(Set(colors))
```

Strings of color names in the `colors` dictionary correspond to RGB values, where each value is in the interval $[0, 1]$.

```
colors['violet']
```

5.6.1 Color Complements

Complementary colors are two colors whose hues are on opposite sides of the color wheel.

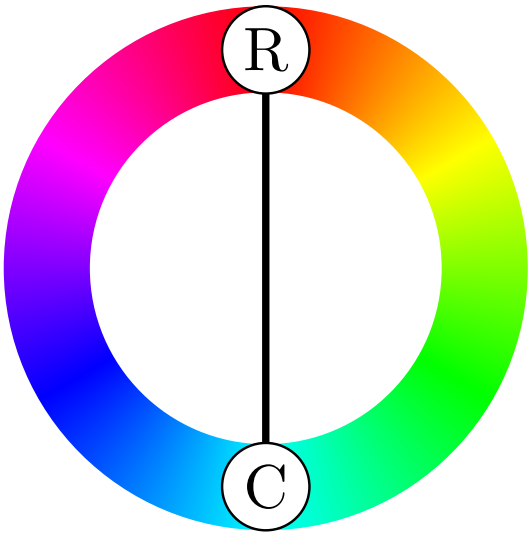

Figure 5.6.1 On RGB displays, red and cyan are complementary colors.

You can obtain a color's hue is by using the `.hsl()` attribute. The `.hsl()` attribute outputs in the format `(hue, saturation, lightness)` with all values being a `float` in the interval $[0, 1]$. This interval causes colors to be complementary if and only if their hues differ by $\frac{1}{2}$.

```
v_HSL = colors['violet'].hsl()
v_HSL
```

The hue value is the first entry in the tuple meaning the hue is at index `0`.

```
v_hue = colors['violet'].hsl()[0]
v_hue
```

The `float` class can cause issues when comparing values. For example, consider the complementary colors violet and green.

```
g_hue = colors['green'].hsl()[0]
print(v_hue)
print(g_hue)
print('Is␣complementary␣pair:', v_hue == g_hue + 0.5)
```

The imprecision caused by the `float` class caused a false negative when testing to see if two colors were complements. This can be avoided by converting the hue values to Sage's `rational` class.

```
g_hue = Rational(colors['green'].hsl()[0])
v_hue = Rational(colors['violet'].hsl()[0])
print(v_hue)
print(g_hue)
print('Is␣complementary␣pair:', v_hue == g_hue + 1/2)
```

Here is a relation, on C, defined as such: a color `c1` is related to a color `c2` if and only if the hues of `c1` and `c2` are complementary, where the test for complementation is the method used above.

```
def iscomplement(m, k):
    m_hue = Rational(colors[m].hsl()[0])
```

```
    k_hue = Rational(colors[k].hsl()[0])
    return m_hue == k_hue + 1/2

Comp = [(c1, c2) for c1 in C for c2 in C
    if iscomplement(c1, c2)]
Comp
```

5.6.2 Light and Dark Modes

Now that we know what pairs are complements, we will chose a darker foreground for the light background of the light-mode and a lighter foreground for the dark background of the dark-mode.

The lightness value is the third value in the tuple created by `.hsl()` meaning it is indexed at `2`.

```
v_light = colors['violet'].hsl()[2]
v_light
```

Instead of comparing the values of individual colors, we will now compare the lightness between two complementary pairs to create the light mode and the dark mode. This can be done by adding the lightness value of each color in the pair before comparing them.

```
vg = ('violet', 'green')
v_light = colors[vg[0]].hsl()[2]
g_light = colors[vg[1]].hsl()[2]
vg_total = v_light + g_light
vg_total
```

A partial order on `Comp` can be created, as relations are themselves sets. Here is a relation, on `Comp`, defined as such: `p1` relates to `p2` if and only if the sum of lightness values in `p1` is less than the sum in `p2`.

```
def isBrighterPair(m, k):
    m_total = colors[m[0]].hsl()[2] + colors[m[1]].hsl()[2]
    k_total = colors[k[0]].hsl()[2] + colors[k[1]].hsl()[2]
    return m_total < k_total

L_order = [(p1,p2) for p1 in Comp for p2 in Comp
    if isBrighterPair(p1,p2)]
L_order
```

Remember, `L_order` is a relation on a set containing pairs of colors, therefore it orders our complementary pairs. For example, the pair of teal and firebrick have a lower sum of lightness values than violet and green.

The `.top()` and `.bottom()` attributes can be used to find the brightest and darkest pairs.

```
P = Poset((Comp, L_order))
highest_pair = P.top()
lowest_pair = P.bottom()

print('The␣highest␣lightness␣pair␣is:', highest_pair)
print('The␣lowest␣lightness␣pair␣is:', lowest_pair)
```

Teal and maroon will be used for light-mode because they are the darkest, while pale-turquoise and firebrick will be used for dark-mode because they are the lightest.

The `.html_color()` attribute can be used to find these colors' corresponding hex codes for use in the website's code.

```
highest_hex = (colors[highest_pair[0]].html_color(),
    colors[highest_pair[1]].html_color())
lowest_hex = (colors[lowest_pair[0]].html_color(),
    colors[lowest_pair[1]].html_color())

print('Dark-mode␣hex␣codes␣are:', highest_hex)
(circle((0,0),1, color=highest_pair[0], fill=true) +
    circle((2.5,0),1, color=highest_pair[1],
    fill=true)).show(figsize=[3.5,1], axes=False)
print('Light-mode␣hex␣codes␣are:', lowest_hex)
(circle((0,0),1, color=lowest_pair[0], fill=true) +
    circle((2.5,0),1, color=lowest_pair[1],
    fill=true)).show(figsize=[3.5,1], axes=False)
```

Chapter 6

Functions

This chapter will briefly discuss the implementation of functions in Sage and will delve deeper into the sequences defined by recursion, including Fibonacci's. We will show how to solve a recurrence relation using Sage.

6.1 Functions

A function from a set A into a set B is a relation from A into B such that each element of A is related to exactly one element of the set B. The set A is called the domain of the function, and the set B is called the co-domain. Functions are fundamental in both mathematics and computer science for describing mathematical relationships and implementing computational logic.

In Sage, functions can be defined using direct definition.

For example, defining a function $f : \mathbb{R} \to \mathbb{R}$ to calculate the cube of a number, such as 3:

```
f(x) = x^3
show(f)
f(3)
```

6.1.1 Graphical Representations

Sage provides powerful tools for visualizing functions, enabling you to explore the graphical representations of mathematical relationships.

For example, to plot the function $f(x) = x^3$ over the interval $[-2, 2]$:

```
f(x) = x^3
plot(f(x), x, -2, 2)
```

6.2 Recursion

Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem. This approach is extensively used in mathematics and computer science, especially in the computation of binomial coefficients, the evaluation of polynomials, and the generation of sequences.

6.2.1 Recursion in Sequences

A recursive sequence is defined by one or more base cases and a recursive step that relates each term to its predecessors.

Notes. Use recursion to solve problems by breaking them down into similar steps. In programming, recursively defined functions often improve code readability.

Given a sequence defined by a recursive formula, we can ask Sage to find its closed form. Here, `s` is a function representing the sequence defined by recursion. The equation `eqn` defines the recursive relation $s_n = s_{n-1} + 2 \cdot s_{n-2}$. The `rsolve()` function is then used to find a closed-form solution to this recurrence, given the initial conditions $s_0 = 2$ and $s_1 = 7$. At last, we use the `SR()` function to convert from Python notation to mathematical notation.

```
from sympy import Function, rsolve
from sympy.abc import n
s = Function('s')
eqn = s(n) - s(n-1) - 2*s(n-2)
sol = rsolve(eqn, s(n), {s(0): 2, s(1): 7})
show(SR(sol))
```

We can use the `show()` function to make the output visually more pleasing; you can try removing it and see how the output looks.

Similarly, the Fibonacci sequence is another example of a recursive sequence, defined by the base cases $F_0 = 0$ and $F_1 = 1$, and the recursive relation $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. This sequence is a cornerstone example in the study of recursion.

```
from sympy import Function, rsolve
from sympy.abc import n
F = Function('F')
fib_eqn = F(n) - F(n-1) - F(n-2)
fib_sol = rsolve(fib_eqn, F(n), {F(0): 0, F(1): 1})
show(SR(fib_sol))
```

The `show()` function is again used here to present the solution in a more accessible mathematical notation, illustrating the power of recursive functions to describe complex sequences with simple rules.

We can also write a function `fib()` to compute the nth Fibonacci number by iterating and updating the values of two consecutive Fibonacci numbers in the sequence. Let's calculate the third Fibonacci number.

```
def fib(n):
    # assuming n is always an int
    if n < 2:
        return n
    else:
        # the initial terms F0 and F1
        U = 0; V = 1
        for k in range(2, n+1):
            W = U + V;
            U = V; V = W
        return V

# display the first 10 Fibonacci numbers
[fib(i) for i in range(10)]
```

We go back to the previous method where we calculated the closed form `fib_sol` and evaluate it now at $n = 3$.

```
from sympy import Function, rsolve, Symbol, simplify
n = Symbol('n')
F = Function('F')
fib_eqn = F(n) - F(n-1) - F(n-2)
fib_sol = rsolve(fib_eqn, F(n), {F(0): 0, F(1): 1})
# Evaluate the solution at n=3
fib3 = simplify(fib_sol.subs(n, 3))
show(SR(fib3))
```

As we can see, we obtain the same number either by evaluating the closed form at $n = 3$ or by finding the third Fibonacci number directly by iteration.

6.2.2 Recursion with Binomial Coefficients

Binomial coefficients, denoted as $\binom{n}{k}$, count the number of ways to choose k elements from an n-element set. They can be defined recursively. Sage can compute binomial coefficients using the `binomial(n, k)` function.

```
binomial(5, 3)
```

We can also explore the recursive nature of binomial coefficients by defining a function ourselves recursively.

```
def binomial_recursive(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return binomial_recursive(n-1, k-1) +
            binomial_recursive(n-1, k)
binomial_recursive(5, 3)
```

This function implements the recursive formula $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, with base cases $\binom{n}{0} = \binom{n}{n} = 1$.

Chapter 7

Graph Theory

Sage is extremely powerful for graph theory. This chapter presents the study of graph theory with Sage, starting with a description of the Graph class through the implementation of optimization algorithms. We also illustrate Sage's graphical capabilities for visualizing graphs.

7.1 Basics

7.1.1 Graph Definition

A **graph** $G = (V, E)$ consists of a set V of vertices and a set E of edges, where

$$E \subset \{\{u, v\} \mid u, v \in V\}$$

The set of edges is a set whose elements are subsets of two vertices.

Terminology:

- Vertices are synonymous with **nodes**.
- Edges are synonymous with **links** or **arcs**.
- In an **undirected graph** edges are **unordered** pairs of vertices.
- In a **directed graph** edges are **ordered** pairs of vertices.

There are several ways to define a graph in Sage. We can define a graph by listing the vertices and edges:

```
V = ['A', 'B', 'C', 'D', 'E']
E = [('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E',
    'A'), ('A', 'D'), ('C', 'E')]
G = Graph([V, E])
G.plot()
```

We can define a graph with an edge list. Each edge is a pair of vertices:

```
L = [('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E',
    'A'), ('A', 'C'), ('B', 'E')]
G = Graph(L)
G.plot()
```

We can define a graph with an edge dictionary like so: `{edge: [neighbor, neighbor, etc], edge: [ neighbor, etc], etc: [etc]}` Each dictionary key is a vertex. The dictionary values are the vertex neighbors.

```
E = {1: [2, 3, 4], 2: [1, 3, 4], 3: [1, 2, 4], 4: [1, 2, 3]}
G = Graph(E)
G.plot()
```

You can improve the readability of a dictionary by placing each item of the collection on a new line:

```
E = {
    1: [2, 3, 4],
    2: [1, 3, 4],
    3: [1, 2, 4],
    4: [1, 2, 3]
}
G = Graph(E)
G.plot()
```

Sage offers a collection of predefined graphs. Here are some examples:

```
graphs.PetersenGraph().show()
graphs.CompleteGraph(5).show()
graphs.TetrahedralGraph().show()
graphs.DodecahedralGraph().show()
graphs.HexahedralGraph().show()
```

Notes. Concepts from graph theory have practical applications related to social networks, computer networks, transportation, biology, chemistry, and more.

7.1.2 Weighted Graphs

A **weighted graph** has a weight, or number, associated with each edge. These weights can model anything including distances, costs, or other relevant quantities.

To create a weighted graph, add a third element to each pair of vertices.

```
E = [('A', 'B', 2), ('B', 'C', 3), ('C', 'D', 4), ('D', 'E',
    5), ('E', 'A', 1)]
G = Graph(E, weighted=True)
G.plot(edge_labels=True)
```

7.1.3 Graph Characteristics

Sage offers many built-in functions for analyzing graphs. Let's examine the following graph:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.show()
```

The `vertices()` method returns a list of the graph's vertices.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
```

```
G.vertices()
```

The `G.edges()` method returns triples representing the graph's vertices and edge labels.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.edges()
```

Return the edges as a tuple without the label by setting `labels=false`.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.edges(labels=false)
```

The **order** of $G = (V, E)$ is the number of vertices $|V|$.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.order()
```

The size of $G = (V, E)$ is the number of edges $|E|$.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.size()
```

The degree of the vertex v, $deg(v)$ is the number of edges incident with v.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.degree('A')
```

The degree sequence of $G = (V, E)$ is the list of degrees of its vertices.

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A'), ('A', 'C'), ('B', 'D')])
G.degree_sequence()
```

7.1.4 Graphs and Matrices

The *adjacency matrix* of a graph is a square matrix used to represent which vertices of the graph are adjacent to which other vertices. Each entry a_{ij} in the matrix is equal to 1 if there is an edge from vertex `i` to vertex `j`, and is equal to 0 otherwise.

```
G = Graph([('A', 'B'), ('A', 'E'), ('B', 'C'), ('C', 'D'),
    ('D', 'E')])
G.adjacency_matrix()
```

We can also define a graph with an adjacency matrix:

```
A = Matrix([
          [0, 1, 0, 0, 1],
          [1, 0, 1, 0, 0],
          [0, 1, 0, 1, 0],
          [0, 0, 1, 0, 1],
```

```
          [1, 0, 0, 1, 0]
])
G = Graph(A)
G.plot()
```

The *incidence matrix* is an alternative matrix representation of a graph, which describes the relationship between vertices and edges. In this matrix, rows correspond to vertices, and columns correspond to edges, with entries indicating whether a vertex is incident to an edge.

```
G = Graph([('A', 'B'), ('A', 'E'), ('B', 'C'), ('C', 'D'),
    ('D', 'E')])
G.incidence_matrix()
```

7.1.5 Manipulating Graphs in Sage

Add a vertex to a graph:

```
G = Graph([(1, 2), (2, 3), (3, 4), (4, 1)])
G.add_vertex(5)
G.show()
```

Add a list of vertices:

```
G.add_vertices([10, 11, 12])
G.show()
```

Remove a vertex from a graph:

```
G.delete_vertex(12)
G.show()
```

Remove a list of vertices from a graph:

```
G.delete_vertices([5,10,11])
G.show()
```

Add an edge between two vertices:

```
G.add_edge(1, 3)
G.show()
```

Delete an edge from a graph:

```
G.delete_edge(2, 3)
G.show()
```

Deleting a nonexistent vertex returns an error. Deleting a nonexistent edge leaves the graph unchanged. Adding a vertex or edge already in the graph, leaves the graph unchanged.

7.2 Plot Options

The `show()` method displays the graphics object immediately with default settings. The `plot()` method accepts options for customizing the presentation of the graphics object. You can import more features from Matplotlib or LaTeX

for fine-tuned customization options. Let's examine how the plotting options improve the presentation and help us discover insights into the structure and properties of a graph. The presentation of a Sage graphics object may differ depending on your environment.

7.2.1 Size

Here is a graph that models the primary colors of the RGB color wheel:

```
E = [
    ('r', 'g'),
    ('g', 'b'),
    ('b', 'r')
]

Graph(E).show()
```

Let's increase the `vertex_size`:

```
Graph(E).plot(vertex_size=1000).show()
```

Resolve the cropping by increasing the `figsize`. Specify a single number or a `(width, height)` tuple.

```
Graph(E).plot(vertex_size=1000, figsize=10).show()
```

Increasing the `figsize` works well in a notebook environment. However, in a SageCell, a large `figsize` introduces scrolling. Setting `graph_border=True` is an alternate way to resolve the cropping while maintaining the size of the graph.

```
Graph(E).plot(vertex_size=1000, graph_border=True).show()
```

7.2.2 Edge Labels

Let's add some edge labels. Within the `list` of edge `tuples`, the first two values are vertices, and the third value is the edge label.

```
E = [
    ('r', 'g', 'yellow'),
    ('g', 'b', 'cyan'),
    ('b', 'r', 'magenta')
]

G = Graph(E).plot(
    edge_labels=True,
)

G.show()
```

7.2.3 Color

There are various ways to specify `vertex_colors`, including hexadecimal, RGB, and color name. Hexadecimal and RGB offer greater flexibility because Sage does not have a name for every color. The color is the `dictionary` key, and the vertex is the `dictionary` value.

The following example specifies the color with RGB values. The values can range anywhere from `0` to `1`. Color the vertex `r` red by setting the first element in the RGB tuple to full intensity with a value of `1`. Next, ensure vertex `r` contains no green or blue light by setting the remaining tuple elements to `0`. Notice vertex `g` is darker because the green RGB value is `.65` instead of `1`.

```
set_vertex_colors = {
    (1,0,0): ['r'], # Color vertex `r` all red
    (0,.65,0): ['g'], # Color vertex `g` dark green
    (0,0,1): ['b'] # Color vertex `b` all blue
}

G = Graph(E).plot(
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

The following example specifies the color by name instead of RGB value. Sage will return an error if you use an undefined color name.

```
set_vertex_colors = {
    'red': ['r'],
    'green': ['g'],
    'blue': ['b']
}

G = Graph(E).plot(
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

Let's specify the `edge_colors` with RGB values. The edge from vertex `r` to vertex `g` is yellow because the RGB tuple sets red and green light to full intensity with no blue light. For darker shades, use values less than `1`.

```
set_edge_colors = {
    (1,1,0): [('r', 'g')],
    (0,1,1): [('g', 'b')],
    (1,0,1): [('b', 'r')]
}

G = Graph(E).plot(
    edge_colors=set_edge_colors,
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

This alternate method specifies the color by name instead:

```
set_edge_colors = {
    'yellow': [('r', 'g')],
```

```
    'cyan': [('g', 'b')],
    'magenta': [('b', 'r')]
}

G = Graph(E).plot(
    edge_colors=set_edge_colors,
    vertex_colors=set_vertex_colors,
    edge_labels=True,
)

G.show()
```

Consider accessibility when choosing colors on a graph. For example, the red and green on the above graph look indistinguishable to people with color blindness. Blue and red are usually a safe bet for contrasting two colors.

Here is a Sage Interact to help identify hexadecimal color values.

- First, click Evaluate (Sage) to define and load the interact. You are welcome to modify the interact definition to suit your needs.
- You may define a new edge list, vertex size, and graph border within an input box.
- After entering new values, press Enter on your keyboard to load the new graph.
- Click on the color selector square to change the color. The hexadecimal value appears to the right of the color square.
- After selecting a new color, the graph will update when you click outside the color selector.

```
@interact
def _(
    edges=input_box(default=[(1, 2), (2, 3), (3, 4), (4,
        1)], label="Graph", width=40),
    vertex_size=input_box(default=2000, label="Vertex␣Size",
        width=40),
    graph_border=input_box(default=True, label="Border",
        width=40),
    color=color_selector(widget='colorpicker', label="Click␣
        ->")
):
    g = Graph(edges)
    color_str = color.html_color()

    show(
        g.plot(
            vertex_size=vertex_size,
            graph_border=graph_border,
            vertex_colors=color_str
        )
    )
```

7.2.4 Layout

Let's define and examine the following graph. Evaluate this cell multiple times and notice the vertex positions are not consistent.