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

August 31, 2024

Website: GitHub Repository[1]

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

- Ken Levasseur, University of Massachusetts Lowell
- Moty Katzman, University of Sheffield
- Simon King, University of Jena∗
- Vartuyi Manoyan, Wright College

We would like to acknowledge the following proof-readers:

- Fabio Re, Rosalind Franklin University
- Justin Lowry, Wright College
- Soma Dey, Wright College∗
- Sydney Hart, Wright College∗
- Ted Jankowski, Wright College
- Tineka Scalzo, Wright College
- Yolanda Nieves, Wright College

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

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 your own custom classes, the open-source community has already defined thousands of classes for us to use. For example, there are specialized classes for working with integers, lists, strings, graphs, and more.

Let's use these new terms to describe the following example.

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

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.

```
type_element = type('a')
type_vowels = type(vowels)

print(f"The type of the element 'a' is: {type_element}")
print(f"The type of the variable vowels is: {type_vowels}")
```

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",
    "authors": ["Zunaid Ahmed", "Hellen Colman", "Samuel
        Lubliner"],
    "institution": "City Colleges of Chicago",
```

```
    "topics_covered": [
        "Set Theory",
        "Combinations and permutations",
        "Logic",
        "Quantifiers",
        "Relations",
        "Functions",
        "Recursion",
        "Graphs",
        "Trees",
        "Lattices",
        "Boolean algebras"
        ],
        "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)
```

Sage is capable of producing animations and pictures. To view more examples of Sage art, see gallery on the Sage Wiki[1]. If you want to use Sage to download a GIF of Fibonacci Tiles by Sébastien Labbé, use the `save()` method.

We will get an error if we supply the arguments out of order.

```
# Originally created by Sébastien Labbé.
# This code is adapted from the original available at:
    https://wiki.sagemath.org/art licensed under GPLv2. For
    more information, visit: https://moinmo.in/GPL
```

[1] `wiki.sagemath.org/art`

```
path_op = dict(rgbcolor='red', thickness=1)
fill_op = dict(rgbcolor='blue', alpha=0.3)
options = dict(pathoptions=path_op, filloptions=fill_op,
    endarrow=False, startpoint=False)
G = [words.fibonacci_tile(i).plot(**options) for i in
    range(7)]
a = animate(G)
a.save(delay=150, 'LabbeFibonacciTiles.gif')
```

In this example, `SyntaxError` occurs because the arguments in the `save()` method are in the wrong order. We can fix this by referring to the documentation and learning the correct argument positions. After running the following cell, the `save()` method saves the file to the cell. You can right-click on the file name and select the option to download the file locally.

```
# Originally created by Sébastien Labbé.
# Adapted from the original available at:
    https://wiki.sagemath.org/art licensed under GPLv2. For
    more information, visit: https://moinmo.in/GPL

path_op = dict(rgbcolor='red', thickness=1)
fill_op = dict(rgbcolor='blue', alpha=0.3)
options = dict(pathoptions=path_op, filloptions=fill_op,
    endarrow=False, startpoint=False)
G = [words.fibonacci_tile(i).plot(**options) for i in
    range(7)]
a = animate(G)
a.save('LabbeFibonacciTiles.gif', delay=150)
```

Sometimes, Sage offers flexibility when using a method with dot notation or as a function. Here is another valid way of using `save()`.

```
# Originally created by Sébastien Labbé.
# Adapted from the original available at:
    https://wiki.sagemath.org/art licensed under GPLv2. For
    more information, visit: https://moinmo.in/GPL

path_op = dict(rgbcolor='red', thickness=1)
fill_op = dict(rgbcolor='blue', alpha=0.3)
options = dict(pathoptions=path_op, filloptions=fill_op,
    endarrow=False, startpoint=False)
G = [words.fibonacci_tile(i).plot(**options) for i in
    range(7)]
a = animate(G)
save(a, 'LabbeFibonacciTiles.gif', delay=150)
```

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)
```

Remember Sage list comprehension behaves differently from `range()`. The range syntax for list comprehension includes the ending value.

For more information, read the CoCalc article about the top mathematical syntax errors in Sage[2]

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. In Sage, you can also define functions yourself. Defining a function in Sage is similar to defining a function in Python.

To define a function in Sage, use the `def` keyword followed by the function name and the function's arguments. The function's body is indented. The `return` keyword returns a value from 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 function definition is stored in memory, and 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)
```

You can also add input validation to your functions to make them 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)]
```

Try calling the function with a negative integer to see the input validation.

[2] `github.com/sagemathinc/cocalc/wiki/MathematicalSyntaxErrors`

```
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)]
```

We can 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 effective doc-

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

[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
[5] https://doc.cocalc.com/sagews.html

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.

[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"])
M
```

Use a `for` loop to view each element of the set on a new line.

```
for month in M:
    print(month)
```

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
```

If you have experience with Python, you may have used a Python `set`. Notice how the Python `set` begins with a lowercase `s`. Even though Sage supports Python sets, we will use Sage `Set` for the added features. Be sure to define `Set()` with an upper case `S`.

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)

for subset in subsets_of_weather:
    print(subset)
```

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.

Notes. Organizations such as clubs, gyms, or online subscription services can use set membership checks to validate if a user is a member or subscriber. Set membership checks can help control access to facilities or content.

```
"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.

Notes. In social networks like Facebook or Twitter, cardinality can represent the number of friends or followers a user has. Understanding the cardinality of social connections can help in personalization, targeted advertising, and studying social influence.

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

Alternatively, we can also use the `len()` function, built into Python. 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 $A\ B$.

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 $A\ B$.

Notes. A practical application of set intersection could be finding common members between two different social media groups. Members who belong to both groups represent the intersection of these groups.

```
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

We have two methods to solve for the difference as well.

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

Notes. In a real-life context, this can be seen as identifying items on a restaurant menu (set A) that you have not yet tried (set B), with A - B representing the new dishes to explore.

```
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 will 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.

Notes. This is analogous to having a list of all possible ice cream flavors (U) and identifying those flavors you have yet to try (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 $A \times D$.

Notes. The Cartesian product is relevant in situations like determining all possible combinations of two different sets, such as shirt colors and pants styles in a wardrobe.

```
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 Set of Sets

Notes. Exploring the power set of a set can be particularly useful in fields such as combinatorics, probability theory, and computer science, where understanding the relationships between subsets of a given set is crucial.

The power set of a set A is the set of all subsets of A, including the empty set and A itself. Sage offers several ways to create a power set including the

`Subsets()` and `powerset()` functions. We will demonstrate the use cases of both functions. While the `powerset()` function is more flexible, the `Subsets()` function is more user-friendly due to the built-in `Set` methods.

The `Subsets()` function returns all subsets of a finite set in no particular order. Here, we will find the power set of the set of colors of the rainbow.

```
rainbow = Set(["red", "orange", "yellow", "green", "blue",
    "indigo", "violet"])
color_combos = Subsets(rainbow)
color_combos
```

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

```
Set([]) in color_combos
```

```
rainbow in color_combos
```

We can find the total number of subsets with `cardinality()` method.

```
color_combos.cardinality()
```

We can view the subsets of a set as a `list` of `Sets`.

```
color_combos.list()
```

We can retrieve a random element from the power set with the `random_element()` method.

```
color_combos.random_element()
```

There are limitations to the `Subsets()` function. For example, the `Subsets()` function returns sets that often include non-hashable objects. Observe the error message when trying to create a power set of a list containing a list.

```
non_hashable_elements = [1, [2, 3], 4]
Subsets(non_hashable_elements).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.

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

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

```
power_set_integers = powerset(ZZ)

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

While the `Subsets()` function can represent infinite sets symbolically, it is not practical. Observe the error message when trying to retrieve a random element from `Subsets(ZZ)`

```
powerset_of_integers = Subsets(ZZ)
powerset_of_integers
```

```
powerset_of_integers.random_element()
```

Pay close attention to the capitalization of function names. Notice the lowercase `s` in `subsets()`, which is an alias for `powerset()`.

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. In Sage, the `factorial()` function computes this value, which is essential in permutations and combinations.

Notes. Factorials are widely used in probability problems, such as determining the number of possible ways a set of items can be arranged.

For example, to 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. Combinations are useful in scenarios like determining the number of possible committees that can be formed from a larger group.

For instance, to calculate the number of ways to choose 3 elements from a set of 5:

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

Another method would be using the `binomial()` function.

```
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. Understanding permutations can help solve problems like scheduling where the order of tasks or events matters.

For instance, to calculate the number of ways to choose 3 elements from a set of 5 when the order matters, we use the `Permutations()` and the `cardinality()` functions.

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

To display the set of permutations for 3 chosen elements from a set of 5 in order, we use the `Permutations()` function and the use the `Set()` function.

```
A = Permutations(5, 3)
Set(A)
```

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

To calculate the permutation of a set with 3 elements, we use the `Permutations()` and the `cardinality()` functions.

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

Similarly, we can also show the sets of permutations:

```
A = Permutations(3)
Set(A)
```

We can also specify the elements:

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

for f in Set(A):
    print(f)
```

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, logical operations such as AND `&`, OR `|`, NOT `~`, conditional `->`, and biconditional `<->` play crucial roles in constructing and evaluating logical expressions.

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

Notes. Logical operators are fundamental in digital circuit design. They are used to create complex circuits like ALUs (Arithmetic Logic Units) in computer processors, where operations such as addition, subtraction, and bit shifting rely on combinations of AND, OR, and NOT gates. Understanding these basics is essential for designing efficient electronic and software solutions.

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)
```

```
# Displays the Boolean formula (p AND q) OR (NOT p) in
    mathematical notation.
```

4.2 Truth Tables

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

Notes. Constructing truth tables is a practical method for debugging complex logical conditions in software development, ensuring all scenarios are accounted for.

```
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)
```

`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.

Notes. Truth tables for three variables are particularly useful in the study of digital circuits and Boolean algebra, where each variable represents a different input or condition.

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

4.3 Analyzing Logical Equivalences

4.3.1 Equivalent Statements

By comparing the truth tables, we can ascertain if two logical statements are equivalent, meaning they have identical truth values for all possible inputs.

Notes. This approach is invaluable in simplifying logical expressions in computer algorithms and understanding proofs in mathematical logic.

```
E1 = propcalc.formula('(p -> q) & (q -> r)')
E2 = propcalc.formula('p -> r')
T1 = E1.truthtable()
T2 = E2.truthtable()
```

```
T1 == T2
```

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. Understanding tautologies is essential in computer science, particularly in optimizing algorithms and validating logical propositions in software verification. It's also fundamental in mathematical proof strategies, such as proof by contradiction.

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

4.3.3 Contradictions

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

Notes. Analyzing the relationship between tautologies and contradictions can aid in the development of critical thinking skills, which are essential for debugging in programming, constructing mathematical proofs, and designing logical circuits.

```
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 "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.

Notes. In database management, relations model the connections between different entities (e.g., customers and orders). Each record in one table can relate to records in another, facilitating complex queries and data analytics essential for operational intelligence and decision making.

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 from B.

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

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

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)
```

Let's use relations to explore matching items of clothes. Let's define two sets, jackets and shirts, as examples:

Notes. In online shopping platforms, relations help personalize customer experiences. Products and user preferences can be modeled as sets, with relations

based on past purchases or browsing history to recommend new products effectively.

$$\text{jackets} = \{j_1, j_2, j_3\}$$

$$\text{shirts} = \{s_1, s_2, s_3, s_4\}$$

The Cartesian product of jackets and shirts includes all possible combinations of jackets with shirts.

```
# Define the sets of jackets and shirts
jackets = Set(['j1', 'j2', 'j3'])
shirts = Set(['s1', 's2', 's3', 's4'])

# View all the possible combinations of jackets and shirts
C = cartesian_product([jackets, shirts])

Set(C)
```

Notes. Recommendation systems in services like streaming and e-commerce use relations to connect users to products or media they are likely to enjoy, based on similar user preferences and behaviors, enhancing user engagement and satisfaction.

Since the Cartesian product returns all the possible combinations, some jackets and shirts will clash. Let's create a relation from jackets to shirts based on matching the items of clothing.

```
# Define a matching relation between jackets and shirts
R = Set([('j1', 's1'), ('j2', 's3'), ('j3', 's4'), ('j1',
    's2')])

print("Matching relation R =", R)
```

5.2 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 iff $a|b$ (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)
```

5.3 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.

Notes. Digraphs are essential in the modeling of computer networks where nodes represent computers or routers and directed edges represent the direction

of data flow. This aids in understanding and optimizing network traffic and routing protocols.

```
# 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)
```

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

Real-world Application: Social Network Analysis. In social network analysis, digraphs are used to represent relationships where direction matters, such as Twitter followers or citations in academic papers. This helps in identifying influential users or key research papers.

```
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)
```

Real-world Application: Systems Biology. Directed graphs are crucial in systems biology to model regulatory networks where nodes represent genes or proteins, and edges represent regulatory influences, helping to decode complex biological systems and interactions.

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]})
```

5.4 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.

5.4.1 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)
```

5.4.2 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)
```

5.4.3 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)
```

5.4.4 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()
```

5.5 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:

Notes. Equivalence relations are fundamental in data classification systems where objects need to be grouped based on shared characteristics. This is crucial in fields ranging from library science, where books are grouped by genres, to digital marketing, where consumers are segmented by purchasing behaviors.

```
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:

$$xRy \text{ iff } x \text{ and } y \text{ live in the same floor of the building.}$$

```
# 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
```

Notes. In social sciences, equivalence classes help in the study of social groups and structures, analyzing how individuals within the same social class share certain characteristics and behaviors, facilitating targeted sociological studies and policy-making.

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()
```

5.6 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.

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()
```

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.

Notes. Defining functions in Sage is not only useful for mathematical calculations but also for creating complex algorithms, modeling data, and simulating real-world scenarios in various fields of science and engineering.

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.

Notes. Utilizing graphical representations of functions can greatly aid in understanding complex concepts in calculus and analysis, such as differentiation and integration, by providing a visual context.

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.

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 us calculate the third Fibonacci number.

```
def fib(n):
    if n == 0 or n == 1: return n
    else:
        U = 0; V = 1  # the initial terms F0 and F1
        for k in range(2, n+1):
            W = U + V; U = V; V = W
        return V
fib(3)
```

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

In discrete mathematics, a graph is a fundamental concept used to model pairwise relations between objects. A graph is defined as an ordered pair $G = (V, E)$, where:

- V is a set of vertices (also called nodes).
- E is a set of edges (also called links or arcs), which are unordered pairs of vertices (in an undirected graph) or ordered pairs of vertices (in a directed graph).

The vertices represent the objects, and the edges represent the connections or relationships between these objects.

We can define a graph in Sage 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()
```

```
# A graphical representation of the graph with vertices A,
    B, C, D, E forming a pentagon and additional diagonals
    A-D and C-E.
```

Notes. Graph theory is pivotal in areas such as network routing and social networking. For example, in telecommunications, graphs are used to model and optimize routing of communications like telephone traffic or internet data packets through a network of nodes and links, ensuring efficient and fault-tolerant network design.

Another way would be specifying the edges only, where each edge is defined by a pair of vertices:

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

```
# A graphical representation of the graph with vertices A,
    B, C, D, E forming a pentagon and additional diagonals
    A-C and B-E.
```

Both methods allow for the creation and visualization of graphs in Sage, with the choice of method depending on the format of the data available and personal preference for clarity and organization.

Graphs can also be defined by specifying the neighbors of each vertex. In Sage, an undirected graph can be created using a dictionary to represent the adjacency list of each vertex, indicating the vertices that are connected by an edge.

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

```
# A graphical representation of the undirected graph. All
    vertices (1, 2, 3, and 4) are interconnected.
```

Sage offers a wide array of predefined graphs for experimentation and study. These graphs represent important concepts and structures in graph theory.

Examples of creating and visualizing some well-known predefined graphs:

```
P = graphs.PetersenGraph()
P.show()
K = graphs.CompleteGraph(5)
K.show()
T = graphs.TetrahedralGraph()
T.show()
D = graphs.DodecahedralGraph()
D.show()
H = graphs.HexahedralGraph()
H.show()
```

```
# Graphical representations of the Petersen graph, complete
    graph on 5 vertices, tetrahedral, dodecahedral, and
    hexahedral graphs.
```

These examples include the Petersen graph, a complete graph on 5 vertices (also known as the K5 graph), the tetrahedral graph, the dodecahedral graph, and the hexahedral (or cubic) graph, showcasing a variety of graph structures.

Defining a graph with data structures before creation allows for managing complex graphs efficiently. This approach is suitable for graphs with a large number of vertices or intricate connectivity.

An example using predefined data to define a graph:

```
graph_data = {'A': ['B', 'C', 'E'], 'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D', 'E'], 'D': ['B', 'C', 'E'], 'E':
    ['A', 'C', 'D']}
G = Graph(graph_data)
```

```
G.show()
```

```
# A graphical representation of the graph defined by the
    dictionary, showing a complex network of connections
    among five vertices.
```

This method provides a clear and organized way to define the relationships between vertices, especially for complex or large-scale graphs.

7.1.2 Weighted Graphs

Weighted graphs extend the concept of graphs by associating a weight, typically a numerical value, with each edge. These weights can represent distances, costs, capacities, or other quantities relevant to the problem being modeled. In Sage, weighted graphs are easily handled, allowing for the exploration of algorithms and properties specific to these types of graphs.

To create a weighted graph, you can add a third element to each pair of vertices representing the weight.

Here is an example of defining a weighted graph with five vertices, forming a pentagon where each edge has a unique weight:

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

```
# A graphical representation of the weighted graph, showing
    vertices A, B, C, D, E and edges with weights.
```

In the code above, the graph `G` is defined with vertices and edges, where each edge is a tuple containing two vertices and the weight of the edge. The `weighted=True` parameter indicates that the graph is weighted, and the `plot(edge_labels=True)` method visualizes the graph with edge weights displayed.

7.1.3 Graph Characteristics

Understanding the characteristics of graphs is fundamental to analyzing their properties and behaviors. These characteristics provide essential insights into the structure of the graph. Sage offers various functions to compute and analyze these characteristics, which are crucial for graph theory studies.

We start by plotting the following graph.

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

```
# Plot the graph.
```

To list all vertices of a graph, we use the `G.vertices()` method, which 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()
```

```
# List of vertices in the graph.
```

To list all edges of a graph, the `G.edges()` method is used. This returns a list of tuples representing the graph's edges.

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

```
# List of edges in the graph.
```

To remove empty labels, include `labels=false`.

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

```
# List of edges in the graph.
```

The order of a graph $G = (V, E)$ is the number of vertices $|V|$.
To find the order of the graph:

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

```
# The number of vertices in the graph G.
```

The size of a graph $G = (V, E)$ is the number of edges $|E|$.
To find the size of the graph:

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

```
# The number of edges in the graph G.
```

The degree of a vertex in G is the number of edges incident to the vertex.
To find the degree of vertex A:

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

```
# The degree of vertex 'A' in the graph.
```

The method `G.degree_sequence()` provides a list of degrees of all vertices in the graph in decreasing order.

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

```
# The degree sequence of the graph, listing the degrees of
    all vertices.
```

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.

To create a graph from 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()
```

```
# A graphical representation of the graph defined by the
    adjacency matrix.
```

In this example, the matrix `A` represents the adjacency matrix of a graph, and `Graph(A)` creates the graph based on that matrix. The plot shows the resulting graph.

The *incidence matrix* is another 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.

Besides creating graphs from matrices, you can also obtain these matrix representations from a given graph in Sage.

To get the adjacency matrix of a graph:

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

```
# The adjacency matrix of the graph G.
```

And to get the incidence matrix:

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

```
# The incidence matrix of the graph G.
```

7.1.5 Manipulating Graphs in Sage

Modifying graphs by adding or removing vertices and edges allows us to observe how these changes affect graph properties. Sage provides user-friendly functions for these modifications, facilitating dynamic exploration of graphs.

To add a vertex to a graph:

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

```
# A graphical representation of the graph with an
    additional vertex.
```

To remove a vertex from a graph:

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

```
# The graph with vertex 4 removed.
```

To add an edge between two vertices in a graph:

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

```
# The graph with a new edge between vertices 1 and 3.
```

To delete an edge from a graph:

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

```
# The graph with the edge between vertices 1 and 2 removed.
```

Observe that deleting a vertex that was not there, returns an error whilst deleting an edge that was not in the graph leaves the graph unchanged. Adding a vertex or edge that was already in the graph, leaves the graph unchanged.

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

```
# The graph with the edge between vertices 1 and 3 removed.
```

You can also add a list of vertices or edges.

```
G = Graph([(1, 2), (2, 3), (3, 4), (4, 1)])
G.add_vertices([10, 11, 12])
G.show()
```

```
# The graph with the edge between vertices 1 and 3 removed.
```

7.2 Plot Options

Visualizing graphs in Sage not only provides insights into the structure and properties of the graph but also offers flexibility through customization options. These options include adjusting colors, edge thickness, vertex size, and more, allowing for clearer representation and better understanding of complex graphs. This subsection will guide you through various customization options for graph visualization in Sage.

To begin with, you can customize the color of vertices and edges. This is particularly useful for highlighting specific parts of a graph, such as a path or a subgraph.

Here is an example of how to set different colors for vertices and edges:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A')])
G.plot(vertex_color='blue', edge_color='red')
```

```
# A graphical representation of the graph with blue
    vertices and red edges.
```

Adjusting the thickness of edges can also enhance the visualization, especially for weighted graphs or to emphasize certain edges over others.

To modify the edge thickness:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A')])
G.plot(edge_thickness=2.5)  # Adjusts the thickness of all
    edges
```

```
# A graphical representation of the graph with thicker
    edges.
```

The `edge_thickness` parameter allows you to specify the thickness of the edges in the plot.

Another useful customization is adjusting the size of the vertices, which can be helpful when dealing with graphs that have a large number of vertices or when you want to fit certain texts inside of the vertices.

To change the size of vertices:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A')])
G.plot(vertex_size=2000)  # Adjusts the size of all vertices
```

```
# A graphical representation of the graph with larger
    vertices.
```

In order to avoid the previous graph cropping, we can use

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A')])
G.plot(vertex_size=2000, graph_border=True)
```

```
# A graphical representation of the graph with larger
    vertices.
```

While customizing labels, Sage allows you to enable or disable labels for both vertices and edges, providing clarity and additional context to the graph. However, note that setting specific colors for edge labels directly through the `plot` method is not supported.

To customize labels and enable them for vertices and edges:

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

```
# A graphical representation of the graph with vertex and
    edge labels enabled.
```

Layouts in Sage determine how vertices are positioned on the plane. SageMath provides several standard layout algorithms, each offering a unique perspective on the graph's structure.

For instance, the `circular` layout places vertices in a circle, which can highlight the symmetry and regular structure of a graph.

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

```
# Visualizes the graph with vertices positioned in a
    circular layout.
```

The `spring` layout, also known as a force-directed layout, positions vertices to minimize edge intersections and edge length variance, resembling a system of springs. This layout often reveals the underlying structure of the graph by separating clusters of nodes.

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

```
# Displays the graph using a spring (force-directed) layout.
```

Another common layout is the `planar` layout, which is applied to graphs that can be drawn without any edge crossings.

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

```
# Attempts to display the graph with a planar layout, if
    possible. If the graph is not planar, an error message
    is printed.
```

After exploring the standard layouts, you might want to delve into more sophisticated layout algorithms offered by Graphviz. By setting the `layout` option to `graphviz`, you can access various Graphviz layout engines. For instance, `prog='dot'` is great for hierarchical graphs, `prog='neato'` for undirected graphs with spring model layouts, `prog='fdp'` for force-directed layouts, `prog='twopi'` for radial layouts, and `prog='circo'` for circular layouts. These layouts are particularly useful when the graph's structure needs to be visualized in a way that standard layouts cannot accommodate.

Here is an example using a Graphviz layout:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A')])
G.plot(layout='graphviz', prog='dot')
```

```
# The graph is rendered using the Graphviz 'neato' layout.
```

Changing the shape of the vertices can be useful for differentiating types of vertices or just for aesthetic purposes.

To set the vertex shape to a square:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A')])
# above from previous code
G.plot(vertex_shape='s')  # Changes the shape of vertices to
    squares
```

```
# A graphical representation of the graph with
    square-shaped vertices.
```

Customizing the style of the edges can help in distinguishing different types of relationships or interactions between vertices.

To change the edge style to dashed lines:

```
G = Graph([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'),
    ('E', 'A')])
# above from previous code
G.plot(edge_style='dashed')  # Changes the edges to dashed
    lines
```

```
# A graphical representation of the graph with dashed edges.
```

Below is an example how to apply multiple options to a plot:

```
# Creating the graph with its edges
G = Graph(
    [
        ('A', 'B'),
        ('B', 'C'),
        ('C', 'D'),
        ('D', 'E'),
        ('E', 'A')
    ]
)
G.plot(
    layout='circular',
    vertex_color='blue',
    edge_color='red',
    edge_thickness=2.5,
    vertex_size=2000,
    vertex_labels=True,
    edge_labels=True,
    vertex_shape='s',
    edge_style='dashed'
)
```

```
# A graphical representation of the graph incorporating
    multiple customization options for enhanced
    visualization.
```

7.3 Paths

A path between two vertices u and v is a sequence of consecutive edges starting at u and ending at v.

To get all paths between two vertices:

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

```
# Lists all paths from vertex 1 to vertex 4.
```

The length of a path is defined as the number of edges that make up the path.

Finding the shortest path between two vertices can be achieved using the `shortest_path()` function:

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

```
# Retrieves the shortest path from vertex 1 to vertex 4, if
    it exists.
```

Notes. Pathfinding in graphs is a key algorithm in logistics to determine the most efficient route between locations. It's used in GPS navigation systems to find the quickest route for vehicles, and in network design to optimize data flow between servers, enhancing the efficiency of network infrastructures.

A graph is said to be connected if there is a path between any two vertices in the graph.

To determine if a graph is connected, we can use the `is_connected()` function:

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

```
# Checks if the graph is connected, returning a boolean
    value.
```

A connected component of a graph G is a maximal connected subgraph of G. If the graph G is connected, then it has only one connected component.

For example, the following graph is not connected:

```
G = Graph({1: [2, 3], 2: [4], 5: [6]})
G.is_connected()
```

```
# Checks if the graph is connected, returning a boolean
    value.
```

To identify all connected components of a graph, the `connected_components()` function can be utilized:

```
G = Graph({1: [2, 3], 2: [4], 5: [6]})
G.connected_components()
```

```
# Returns a list of connected components, with each
    component represented as a list of vertices.
```

We can visualize the graph as a disjoint union of its connected components, by plotting it.

```
G = Graph({1: [2, 3], 2: [4], 5: [6, 7], 6: [7]})
G.show()
```

The diameter of a graph is the length of the longest shortest path between any two vertices.

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

```
# Calculates the diameter of the graph.
```

A graph is bipartite if its set of vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set:

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

```
# Checks if the graph is bipartite, returning a boolean
    value along with the partition sets if it is bipartite.
```

7.4 Isomorphism

Informally, we can say that an **isomorphism** is a relation of sameness between graphs. Let's say that the graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \to V'$ such that $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$.

This means there is a bijection between the set of vertices such that every time two vertices determine an edge in the first graph, the image of these vertices by the bijection also determines an edge in the second graph, and vice versa. Essentially, the two graphs have the same structure, but the vertices are labeled differently.

Notes. In chemistry, isomorphism is crucial in the study of molecules. Chemical isomers have the same molecular formula but different structures (or graphs), affecting their chemical properties and reactions. Understanding graph isomorphism helps chemists identify and differentiate these molecules efficiently.

```
C = Graph(
  {
    'a': ['b', 'c', 'g'],
    'b': ['a', 'd', 'h'],
    'c': ['a', 'd', 'e'],
    'd': ['b', 'c', 'f'],
    'e': ['c', 'f', 'g'],
    'f': ['d', 'e', 'h'],
    'g': ['a', 'e', 'h'],
    'h': ['b', 'f', 'g']
  }
)

D = Graph(
  {
    1: [2, 6, 8],
    2: [1, 3, 5],
    3: [2, 4, 8],
    4: [3, 5, 7],
    5: [2, 4, 6],
    6: [1, 5, 7],
    7: [4, 6, 8],
    8: [1, 3, 7]
  }
)

C.show()
D.show()
```

The sage `is_isomorphic()` method can be used to check if two graphs are isomorphic. The method returns `True` if the graphs are isomorphic and `False` if the graphs are not isomorphic.

```
C.is_isomorphic(D)
```

The **invariants under isomorphism** are conditions that can be checked to determine if two graphs are not isomorphic. If one of these fails then the graphs are not isomorphic. If all of these are true then the graph may or may not be isomorphic. The three conditions for invariants under isomorphism are:

$$G = (V, E) \text{ is connected iff } G' = (V', E') \text{ is connected}$$

$$|V| = |V'| \text{ and } |E| = |E'|$$

$$\text{degree sequence of } G = \text{ degree sequence of } G'$$

Notes. Graph isomorphism has implications in computer science, particularly in data mining and machine learning. Algorithms that detect isomorphisms can help identify similar data structures in databases, facilitating the classification and clustering of data based on structural similarity.

To summarize, if one graph is connected and the other is not, then the graphs are not isomorphic. If the number of vertices and edges are different, then the graphs are not isomorphic. If the degree sequences are different, then the graphs are not isomorphic. If all three invariants are satisfied, then the graphs may or may not be isomorphic.

Let's define a function to check if two graphs satisfy the invariants under isomorphism. Make sure you run the next cell to define the function before using the function.

```
def invariant_under_isomorphism(G1, G2):
    print("Are both graphs connected? ", end="")
    are_connected: bool = (
        G1.is_connected() == G2.is_connected()
    )
    print("Yes" if are_connected else "No")

    print(
        "Do both graphs have same number of "
        "vertices and edges? ", end=""
    )
    have_equal_vertex_and_edge_counts: bool = (
        G1.order() == G2.order() and
        G1.size() == G2.size()
    )
    print(
        "Yes" if have_equal_vertex_and_edge_counts else "No"
    )

    # Sort the degree-sequences because
    # the order of vertices doesn't matter.
    print(
        "Do both graphs have the same degree sequence? ",
        end=""
    )
    have_same_degree_sequence: bool = (
        sorted(G1.degree_sequence()) ==
        sorted(G2.degree_sequence())
```

```
    )
    print("Yes" if have_same_degree_sequence else "No")

    # All checks
    are_invariant_under_isomorphism =(
        are_connected and
        have_equal_vertex_and_edge_counts and
        have_same_degree_sequence
    )
    print(
        "\nTherefore, the graphs {0} isomorphic.".format(
            "may be" if are_invariant_under_isomorphism
            else "are not"
        )
    )
```

If we use `invariant_under_isomorphism` on the C and D, the output will let us know that the graphs may or may not be isomorphic. We can use the `is_isomorphic()` method to check if the graphs are definitively isomorphic.

```
invariant_under_isomorphism(C, D)
```

Notes. Network theory uses graph isomorphism to analyze and understand different networks, such as social networks or communication networks. By identifying isomorphic subgraphs, researchers can detect patterns and anomalies across different network systems.

Let's construct a different pair of graphs A and B defined as follow

```
A = Graph(
  [
    ('a', 'b'),
    ('b', 'c'),
    ('c', 'f'),
    ('f', 'd'),
    ('d', 'e'),
    ('e', 'a')
  ]
)

B = Graph(
  [
    (1, 5),
    (1, 9),
    (5, 9),
    (4, 6),
    (4, 7),
    (6, 7)
  ]
)

A.show()
B.show()
```

This time, if we apply `invariant_under_isomorphism` function on A and B, the output will show us that they are not isomorphic.

```
invariant_under_isomorphism(A, B)
```

7.5 Euler and Hamilton

7.5.1 Euler

An **Euler path** is a path that uses every edge of a graph exactly once. An Euler path that is a circuit is called an **Euler circuit**.

The idea of an Euler path emerged from the study of the **Königsberg bridges** problem. Leonhard Euler wanted to know if it was possible to walk through the city of Königsberg, crossing each of its seven bridges exactly once. This problem can be modeled as a graph, with the land masses as vertices and the bridges as edges.

```
konigsberg = [('A', 'B', 'b_1'),
              ('A', 'B', 'b_2'),
              ('A', 'C', 'b_3'),
              ('A', 'C', 'b_4'),
              ('D', 'A', 'b_5'),
              ('D', 'B', 'b_6'),
              ('D', 'C', 'b_7')]
G = Graph(konigsberg, multiedges=True)
G.show(edge_labels=True)
```

Notes. Eulerian circuits are not only academic but have practical applications in logistics and waste management. For example, garbage collection routes can be planned as Eulerian circuits to minimize travel and costs. Similarly, postmen use Eulerian paths to optimize mail delivery routes.

While exploring this problem, Euler discovered the following:

- A connected graph has an **Euler circuit** iff every vertex has an even degree.
- A connected graph has an **Euler path** iff there are at most two vertices with an odd degree.

We say that a graph is **Eulerian** if contains an Euler circuit.

We can use Sage to determine if a graph is Eulerian.

Notes. In computer networking, designing networks to ensure resilience and fault tolerance often involves constructing Eulerian graphs, where a network can remain operational even if a connection fails.

```
G.is_eulerian()
```

Since this returns `False`, we know that the graph is not Eulerian. Therefore, it is not possible to walk through the city of Königsberg, crossing each of its seven bridges exactly once.

We can use `path=True` to determine if a graph contains an Euler path. Sage will return the beginning and the end of the path.

```
G = Graph([(1, 2), (2, 3), (3, 4), (4, 1), (2, 4), (1, 3),
    (1, 4)], multiedges=True)
G.show()
G.is_eulerian(path=True)
```

If the graph is Eulerian, we can ask Sage to find an Euler circuit with the `eulerian_circuit` function. Let's take a look at the following graph.

```
G = Graph([(1, 2), (2, 3), (2, 3),(3, 4), (4, 1), (2, 4),
    (1, 3), (1, 4)], multiedges=True)
G.show()
G.eulerian_circuit()
```

If we are not interested in the edge labels, we can set `labels=False`. We can also set `return_vertices=True` to get a list of vertices for the path

```
G = graphs.CycleGraph(6)
G.eulerian_circuit(labels=False, return_vertices=True)
```

7.5.2 Hamilton

A **Hamilton path** is a path that uses every vertex of a graph exactly once. A Hamilton path that is a circuit is called a **Hamilton circuit**. If a graph contains a Hamilton circuit, we say that the graph is **Hamiltonian**.

Hamilton created the "Around the World" puzzle. The object of the puzzle was to start at a city and travel along the edges of the dodecahedron, visiting all of the other cities exactly once, and returning back to the starting city.

We can represent the dodecahedron as a graph and use Sage to determine if it is Hamiltonian. See for yourself if the dodecahedron is Hamiltonian.

Real-world Application. Hamiltonian circuits are integral to creating efficient routing algorithms used by services like Google Maps and UPS for finding the shortest path through multiple points, optimizing travel and delivery times.

```
graphs.DodecahedralGraph().show()
```

We can ask Sage to determine if the dodecahedron is Hamiltonian.

```
graphs.DodecahedralGraph().is_hamiltonian()
```

By running `Graph.is_hamiltonian??` we see that Sage uses the `traveling_salesman_problem()` function to determine if a graph is Hamiltonian.

The traveling salesperson problem is a classic optimization problem. Given a list of cities and the lengths between each pair of cities, what is the shortest possible route that visits each city and returns to the original city? This is one of the most difficult problems in computer science. It is **NP-hard**, meaning that no efficient algorithm is known to solve it. The complexity of the problem increases with the number of nodes. When working with many nodes, the algorithm can take a long time to run.

Let's explore the following graph:

Scientific Research. In scientific research, especially in genetics, Hamiltonian paths are used to solve the DNA sequencing problem by finding the shortest possible unique sequence that can represent a set of observed sequences.

```
G = Graph({1:{3:2, 2:1, 4:3, 5:1}, 2:{3:6, 4:3, 5:1},
    3:{4:5, 5:3}, 4:{5:5}})
G.show(edge_labels=True)
```

We can ask Sage if the graph contains a Hamiltonian cycle.

```
G.hamiltonian_cycle(algorithm='backtrack')
```

The function `hamiltonian_cycle` returns `True` and lists an example of a Hamiltonian cycle as the list of vertices `[1, 2, 3, 4, 5]`. This is just one of the many Hamiltonian cycles that exist in the graph. Now lets find the minimum Hamiltonian cycle.

```
h = G.traveling_salesman_problem(use_edge_labels=True,
    maximize=False)
h.show(edge_labels=True)
```

Now we have the plot of the minimum Hamiltonian cycle. The minimum Hamiltonian cycle is the shortest possible route that visits each city and returns to the original city. The minimum Hamiltonian cycle is the solution to the traveling salesperson problem. We can ask Sage for the sum of the weights of the edges in the minimum Hamiltonian cycle.

```
sumWeights = sum(h.edge_labels())
print(sumWeights)
```

If there is no Hamiltonian cycle, Sage will return `False`. If we use the `backtrack` algorithm, Sage will return a list that represents the longest path found.

```
G = Graph([(1, 2), (1, 3), (2, 3), (1,4), (4, 7), (3, 5),
    (5, 8), (8, 9), (2,6), (6, 9), (7, 9)])
G.show()
G.hamiltonian_cycle(algorithm='backtrack')
```

Chapter 8

Trees

This chapter completes the preceding one by explaining how to ask Sage to decide whether a given graph is a tree and then introduce further searching algorithms for trees.

8.1 Definitions and Theorems

Given a graph, a **cycle** is a circuit with no repeated edges. A **tree** is a connected graph with no cycles. A graph with no cycles and not necessarily connected is called a **forest**.

Let $G = (M, E)$ be a graph. The following are all equivalent:

- G is a tree.
- For each pair of distinct vertices, there exists a unique path between them.
- G is connected, and if $e \in E$ then the graph $(V, E - e)$ is disconnected.
- G contains no cycles, but by adding one edge, you create a cycle.
- G is connected and $|E| = |v| - 1$.

Let's explore the following graph:

```
data = {
  1: [4],
  2: [3, 4, 5],
  3: [2],
  4: [1, 2, 6, 7],
  5: [2, 8],
  6: [4, 9, 11],
  7: [4],
  8: [5, 10],
  9: [6],
  10: [8],
  11: [6]
  }

G = Graph(data)
G.show()
```

Notes. Trees play a crucial role in the design of network topologies, especially in ensuring that data is transferred efficiently without loops in network paths. Forests can represent disconnected networks, which are critical in scenarios where backup systems are designed to operate independently.

Let's ask Sage if this graph is a tree.

```
G.is_tree()
```

If we remove an edge, we can see that the graph is no longer a tree.

```
G_removed_edge = G.copy()
G_removed_edge.delete_edge((1, 4))
G_removed_edge.show()
G_removed_edge.is_tree()
```

However, we can see that the graph is still a forest.

```
G_removed_edge.is_forest()
```

If we add an edge, we can see that the graph contains a cycle and is no longer a tree.

```
G_added_edge = G.copy()
G_added_edge.add_edge((1, 2))
G_added_edge.show()
G_added_edge.is_tree()
```

8.2 Search Algorithms

The graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq \{\{u, v\} \in E \mid u, v \in V'\}$.

The subgraph $G' = (V', E')$ is a **spanning subgraph** of $G = (V, E)$ if $V' = V$.

A **spanning tree** for the graph G is a spanning subgraph of G that is a tree.

Given a graph, there are several algorithms to calculate a spanning tree. For instance the depth-first search algorithm and the breadth-first search algorithm are two of them.

Breadth-first search algorithm

1. Choose a vertex of the graph (root), arbitrarily.
2. Travel all the edges incident with the root vertex.
3. Give an order to this set of new vertices added.
4. Consider each of these vertices as a root, in order, and add all the unvisited incident edges that do not produce a cycle.
5. Repeat the method with the new set of vertices.
6. Follow the same procedure until all the vertices have been visited.

The output of this algorithm is a spanning tree.

The `breadth_first_search()` function provides a flexible method for traversing graphs. This function can be employed to explore a graph from a specified starting point (or points) and can be adapted for various applications such as

finding the shortest path, generating a spanning tree, or simply exploring the structure of a graph.

Let's consider the following graph:

```
G = Graph({0: [1, 2], 1: [2], 2: [3], 4: [1, 2], 3: [1, 2]})
G.show()
print(list(G.breadth_first_search(start=0,
    report_distance=True)))
```

In this example, the breadth-first search algorithm function is initiated at vertex 0, and it reports the length of each vertex from the starting point. Each element in the output is a tuple consisting of a vertex and its length from the starting vertex. This length is measured in terms of the number of edges that must be traversed to reach that vertex from the start.

Given a weighted graph, of all possible spanning trees that we can calculate, we may be interested in the minimal one. A **minimal spanning tree** is a spanning tree whose sum of wights is minimal. There are several algorithms to calculate a minimal spanning tree including Prim's Algorithm.

Prim's Algorithm: Keep two disjoint sets of vertices. One (L) contains vertices that are in the growing spanning tree and the other (R) that are not in the growing spanning tree.

1. Choose a vertex $u \in V$ arbitrarily. At this step, $L = \{u\}$ and $R = V - \{u\}$.
2. Select the cheapest vertex in R that is connected to the growing spanning tree L and add it into L
3. Follow the same procedure until all the vertices are in L

The output of this algorithm is a minimal spanning tree.

Let's explore the following graph:

```
G = Graph({1:{3:8, 5:1}, 2:{3:6, 4:4}, 3:{5:2}, 4:{1:5,
    3:4}})
G.show(edge_labels = True)
```

Notes. Spanning trees are crucial in network design, ensuring that all nodes are connected without any cycles, minimizing the overall layout cost while maintaining network connectivity. Prim's algorithm, in particular, is vital in constructing efficient road, electrical, and computer networks.

We can ask Sage for the minimal spanning tree of this graph. By running `Graph.min_spanning_tree??` We can see that `min_spanning_tree()` uses a variation of Prim's Algorithm by default. We can also use other algorithms such as Kruskal, Boruvka, or NetworkX.

```
G.min_spanning_tree(by_weight=True)
```

Let's visualize the minimal spanning tree.

```
h = Graph(G.min_spanning_tree(by_weight=True))
h.show(edge_labels = True)
```

Chapter 9

Lattices

This chapter builds on the partial order sets introduced earlier and explains how to ask Sage to decide whether a given poset is a lattice. Then, we show how to calculate the meet and join tables using built-in and customized Sage functions.

9.1 Lattices

9.1.1 Definition

A lattice is defined as a partially ordered set (poset) in which any two elements have a least upper bound (also known as join) and greatest lower bound (also known as meet).

In Sage, a lattice can be represented as a poset using the `Poset()` function. This function takes a tuple as its argument, where the first element is the set of elements in the poset, and the second element is a list of ordered pairs representing the partial order relations between those elements.

```
L = Poset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
L.plot()
```

```
 # This will plot the poset that forms a lattice structure.
```

Notes. Lattices play a critical role in computer science, particularly in structuring hierarchical access control in security systems and databases. They help in managing permissions through role-based access control, where roles and data can be organized in a lattice to determine the propagation of access rights.

We can also use `LatticePoset()` function to plot the lattice. The function `Poset()` can be used with any poset, even when the poset is not a lattice. The function `LatticePoset()` will give an error if the poset is not a lattice. The function is also more optimized and reduces the time needed when plotting complicated lattices.

```
L = LatticePoset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
```

```
L.plot()
```

```
# This will plot the latticeposet that forms a lattice
    structure.
```

The function `is_lattice()` can be used to determine whether the poset is a lattice.

```
L = Poset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
L.is_lattice()
```

```
# This will check if the poset is a lattice.
```

9.1.2 Join

The join of two elements in a lattice is the least upper bound of those elements.

To check if a poset is a join semi-lattice (every pair of elements has a least upper bound), we use `is_join_semilattice()` function.

```
L = Poset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
L.is_join_semilattice()
```

```
# Returns True if the lattice is a join semilattice;
    otherwise, False.
```

We can also find the join for individual pairs using the `join()` function.

```
L = Poset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
L.join('b', 'f')
```

9.1.3 Meet

The meet of two elements in a lattice is their greatest lower bound.

To check if a poset is a meet semi-lattice (every pair of elements has a greatest lower bound), we use `is_meet_semilattice()` function.

```
L = Poset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
L.is_meet_semilattice()
```

```
# Returns True if the lattice is a meet semilattice;
    otherwise, False.
```

We can also find the meet for individual pairs using the `meet()` function.

```
L = Poset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
```

```
L.meet('a', 'b')
```

9.2 Tables of Operations

This section delves into the representation of meet ( ) and join ( ) operations within lattices using operation tables. Such tables are pivotal for visualizing the algebraic structure of lattices and understanding how elements combine under each operation.

9.2.1 Meet Operation Table

The meet operation table illustrates the greatest lower bound, or meet, for every pair of elements in the lattice.

For outputting the table as a matrix, we need to specify that the poset is indeed a lattice, thus requiring us to use the function `LatticePoset()`. Then we can use the function `meet_matrix()` to process the table.

```
L = LatticePoset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
P = L.meet_matrix(); L
show(P)
```

```
# Displays the meet operation table as a matrix.
```

From the output matrix, we can see that each entry a_{ij} is not the actual value of the meet of the elements a_i and a_j but just its position in the lattice. Sage does not have a specific function to output the exact meet, but we can define our own function for this purpose.

```
L = LatticePoset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
           [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
            ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
P = L.meet_matrix()
elements = L.list()
K = [['' for _ in range(P.ncols())] for _ in
    range(P.nrows())]
for i in range(P.nrows()):
    for j in range(P.ncols()):
        K[i][j] = elements[P[i,j]]
for row in K:
    print(row)
```

We can further improve our function to show the output as a table instead of a matrix.

```
L = LatticePoset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
                  [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c',
                      'd'],
                   ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f',
                       'g']]))
P = L.meet_matrix()
elements = L.list()
```

```
def create_table(elements, P):
    column_widths = max(len(element) for element in
        elements) + 2
    header = " " * column_widths + "| " + "
        ".join(f"{elem:>{column_widths}}" for elem in
        elements)
    divider = "-" * len(header)
    table = [header, divider]
    for i, row_label in enumerate(elements):
        row = f"{row_label:>{column_widths}} | " + "
            ".join(f"{elements[P[i,j]]:>{column_widths}}"
            for j in range(len(elements)))
        table.append(row)
    return "\n".join(table)
print(create_table(elements, P))
```

9.2.2 Join Operation Table

Conversely, the join operation table presents the least upper bound, or join, for each pair of lattice elements.

Similarly, we can use the function `join_matrix()` to process the table.

```
L = LatticePoset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
          [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
           ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
P = L.join_matrix(); L
show(P)
```

```
# Displays the join operation table as a matrix.
```

We can also output the elements of the poset by slightly changing the function we previously defined.

```
L = LatticePoset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
          [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
           ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']]))
P = L.join_matrix()
elements = L.list()
K = [['' for _ in range(P.ncols())] for _ in
    range(P.nrows())]
for i in range(P.nrows()):
    for j in range(P.ncols()):
        K[i][j] = elements[P[i,j]]

for row in K:
    print(row)
```

We can also output the join table just as shown above for meet.

```
L = LatticePoset((['a', 'b', 'c', 'd', 'e', 'f', 'g'],
                  [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c',
                      'd'],
                   ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f',
                       'g']]))
P = L.join_matrix()
```

```
elements = L.list()
def create_table(elements, P):
    column_widths = max(len(element) for element in
        elements) + 2
    header = " " * column_widths + "| " + "
        ".join(f"{elem:>{column_widths}}" for elem in
        elements)
    divider = "-" * len(header)
    table = [header, divider]
    for i, row_label in enumerate(elements):
        row = f"{row_label:>{column_widths}} | " + "
            ".join(f"{elements[P[i,j]]:>{column_widths}}"
            for j in range(len(elements)))
        table.append(row)
    return "\n".join(table)
print(create_table(elements, P))
```

Chapter 10

Boolean Algebra

This chapter completes the preceding one by explaining how to ask Sage to decide whether a given lattice is a Boolean algebra. We also illustrate basic operations with Boolean functions.

10.1 Boolean Algebra

A Boolean algebra is a bounded lattice that is both complemented and distributive.

In computer software, Boolean algebra is used to construct and simplify expressions in programming languages that support logical decision-making processes. This simplification is critical in developing efficient algorithms and software that perform millions of calculations and logical operations per second.

```
p = Posets.DivisorLattice(20)
p.show()
```

```
# This will display the lattice structure for divisors of
    the number 20.
```

The following Sage cell checks whether the divisor lattice is distributive. A distributive lattice is one where the operations of join and meet distribute over each other.

```
p = Posets.DivisorLattice(20)
p.is_distributive()
```

```
# Returns True if the lattice is distributive, along with a
    certificate if applicable.
```

In digital electronics, the principles of Boolean algebra guide the design of circuits such as multiplexers, demultiplexers, encoders, and decoders, essential components in telecommunications and signal processing.

Another important property in Boolean algebra is the complemented lattice. A complemented lattice is one where every element has a complement in the lattice.

```
p = Posets.DivisorLattice(20)
p.is_complemented()
```

```
# Returns True if the lattice is complemented, along with a
    certificate if applicable.
```

10.2 Boolean functions

A Boolean function is a function that takes only values 0 or 1 and whose domain is the Cartesian product $\{0,1\}^n$.

In SageMath, Boolean functions can be manipulated through various built-in functions designed for handling Boolean variables and expressions.

Notes. Boolean algebra is crucial in digital circuit design. For example, simplifying the expression of a digital circuit can minimize the number of gates used, which reduces cost and power consumption. Techniques such as Karnaugh maps and Boolean simplification are common in the industry.

```
B = BooleanPolynomialRing(3, 'x')
f = B('x0 + x1 * x2')
show(f)
```

```
# Outputs the Boolean expression in polynomial form
```

References

We based most of this text on the Discrete Math lectures at Wilbur Wright College, taught by Professor Hellen Colman. We focused our efforts on creating original work, and we drew inspiration from the following sources:

Doerr, Al, and Ken Levasseur. ADS Applied Discrete Structures. https://discretemath.org, 21 May 2023.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

This book was authored in PreTeXt.

Index