DISCRETE MATH WITH SAGEMATH

Hellen Colman
City College of Chicago

Discrete Math with SageMath

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide and accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.

LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 08/21/2025

TABLE OF CONTENTS

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

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!

Footnote

1. https://doc.sagemath.org/html/en/reference/spkg/

1.1: Intro to Sage

You can execute and modify Sage code directly within the SageMath Cells 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.

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: `/`
- 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

Try the following examples:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
2
```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
100 - 1
```

```
99
```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
3*4
```

```
12
```

Sage supports two exponentiation operators:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
8
```

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

```
8
```

Division in Sage:


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

```
5/3
```



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

```
1.66666666666667
```

Integer division and modulo:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
1
```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

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

```
3
```

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.

Python has a set of reserved keywords that cannot be used as variable names:

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

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


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

```
True
```

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

1.2: Data Types

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
2
```

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[*/ \frac{\sqrt{2}}{\log(3)} /*]] > */ \tag{1.2.1}$$

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[*/ \frac{2}{3} /*]]> */ \tag{1.2.2}$$

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
Hello, World!
<class 'str'>
```

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

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

```
True is <class 'bool'>
```

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


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

```
[1, 3, 3, 2]
<class 'list'>
```

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

```
l[0]
```

```
1
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
len(l)
```

```
4
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
(1, 3, 3, 2)
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{1, 2, 3}
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
<class 'sage.sets.set.Set_object_enumerated_with_category'>
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

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

```
{1, 2, 3, 5}
```

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


```
d = {
    "title": "Discrete Math with SageMath",
    "institution": "City Colleges of Chicago",
    "topics_covered": [
        "Set Theory",
        "Combinations and Permutations",
        "Logic",
        "Quantifiers",
        "Relations",
        "Functions",
        "Recursion",
        "Graphs",
        "Trees",
        "Lattices",
        "Boolean Algebras",
        "Finite State Machines"
    ],
    "format": ["Web", "PDF"]
}
type(d)
```

```
<class 'dict'>
```

Use the `pprint` module to improve the dictionary readability.


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

```
{'format': ['Web', 'PDF'],
 'institution': 'City Colleges of Chicago',
 'title': 'Discrete Math with SageMath',
 'topics_covered': ['Set Theory',
                    'Combinations and Permutations',
                    'Logic',
                    'Quantifiers',
                    'Relations',
                    'Functions',
                    'Recursion',
                    'Graphs',
                    'Trees',
                    'Lattices',
                    'Boolean Algebras',
                    'Finite State Machines']}
```

1.3: Flow Control Structures

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

- ***Assignment*** statements store values in *variables*. These let us reuse results and build more complex expressions step by step. An assignment is performed using the `=` operator as discussed earlier (see Subsection 1.1.2). Note that Sage also supports compound assignment operators like `+=`, `-=`, `*=`, `/=`, and `%=` which combine assignment with basic arithmetic operations (addition, subtraction, multiplication, division and modulus).
- ***Branching*** uses *conditional statements* like `if`, `elif`, and `else` to execute different blocks of code based on logical tests.
- ***Loops*** such as `for` and `while` let us iterate over some data structures and also repeat blocks of code multiple times. This is useful when processing sequences, performing computations, or automating repetitive tasks.

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

Note

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

1.3.1 Conditional Statements

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
x is odd and not divisible by 3
```

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

1.3.2 Iteration

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

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


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

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

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

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

```
10
11
12
13
14
15
16
17
18
19
```

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


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

```
30
39
48
57
```

```
66
75
84
```

1.3.2.1 List Comprehension

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[729, 1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[1000, 1728, 2744, 4096, 5832, 8000]
```

1.3.3 Other Flow Control Structures

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

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
0
1
2
3
4
```

1.4: Defining Functions

Sage comes with many built-in functions. While Math terminology is not always standard, be sure to refer to the documentation to understand the exact functionality of these built-in functions and know how to use them. You can also define custom functions to suit your specific needs. You are welcome to use the custom functions we define in this book. However, since these custom functions are not part of the Sage source code, you will need to copy and paste the functions into your Sage environment. In this section, we'll explore how to define custom functions and use them.

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

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

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


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

```

```

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

```
pascal_row(5)
```

```
[1, 5, 10, 10, 5, 1]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 pascal_row("5")

Cell In[2], line 2, in pascal_row(n)
      1 def pascal_row(n):
----> 2     return [binomial(n, i) for i in range(n + Integer(1))]

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/rings/integer.pyx:1787
   1785         return y
   1786
-> 1787     return coercion_model.bin_op(left, right, operator.add)
   1788
   1789 cpdef _add_(self, right):

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/structure/coerce.pyx:1
   1246     # We should really include the underlying error.
   1247     # This causes so much headache.
-> 1248     raise bin_op_exception(op, x, y)
   1249
   1250 cpdef canonical_coercion(self, x, y):

TypeError: unsupported operand parent(s) for +: '<class 'str'>' and 'Integer Ring'
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
pascal_row(-5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 pascal_row(-Integer(5))

Cell In[5], line 3, in pascal_row(n)
      1 def pascal_row(n):
```

```
      2     if n < Integer(0):
----> 3         raise ValueError("`n` must be a non-negative integer")
      4     return [binomial(n, i) for i in range(n + Integer(1))]

ValueError: `n` must be a non-negative integer
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

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

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

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

        sage: pascal_row(0)  # the top row
        [1]
        sage: pascal_row(4)
        [1, 4, 6, 4, 1]

    It is an error to provide a negative value for `n`:
        sage: pascal_row(-1)
        Traceback (most recent call last):
        ...
        ValueError: `n` must be a non-negative integer

    .. NOTE::
```

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

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

```

```

After redefining the function, you can view the `docstring` by calling the `help()` function on the function name:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
help(pascal_row)
```

```
Help on function pascal_row in module __main__:

pascal_row(n)
    Return row `n` of Pascal's triangle.

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

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

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

        sage: pascal_row(0)  # the top row
        [1]
        sage: pascal_row(4)
        [1, 4, 6, 4, 1]

    It is an error to provide a negative value for `n`:
        sage: pascal_row(-1)
```

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

    .. NOTE::
        This function uses the `binomial` function to compute each
        element of the row.
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
pascal_row??
```

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

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

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

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

        sage: pascal_row(0)  # the top row
        [1]
        sage: pascal_row(4)
        [1, 4, 6, 4, 1]

    It is an error to provide a negative value for `n`:
        sage: pascal_row(-1)
        Traceback (most recent call last):
        ...
        ValueError: `n` must be a non-negative integer

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

    return [binomial(n, i) for i in range(n + Integer(1))]
File:      ~/<ipython-input-7-b46457fa0781>
Type:      function
```

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

1.5: Object-Oriented Programming

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

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

1.5.1 Objects in Sage

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
<class 'list'>
```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
<class 'str'>
```

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

1.5.2 Dot Notation and Methods

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
['a', 'e', 'i', 'o', 'u', 'y']
```

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

1.5.3 Sage's Set Class

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
<class 'sage.sets.set.Set_object_enumerated_with_category'>
```

The `Set` class in Sage provides attributes and methods specifically designed for working with sets.

While OOP might seem abstract at first, it will become clearer as we explore more and dive deeper into Sage features. Sage's built-in classes offer a structured way to represent data and perform powerful mathematical operations. In the next chapters, we'll see how Sage utilizes OOP principles and its built-in classes to perform mathematical operations.

1.6: Display Values

Sage provides multiple ways to display values on the screen. The simplest way is to type the value into a cell, and Sage will display it. Sage also offers functions to format and display output in different styles.

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

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

Let's explore these display methods in action.

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
"Hello, world!"
```

```
'Hello, world!'
```

Using `print()` removes the quotes.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
print('Hello world!')
```

```
Hello world!
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[*/ \frac{\sqrt{2}}{\log(3)} /*]]> */ \tag{1.6.1}$$

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[*/ x^2 /*]]> */ \tag{1.6.2}$$

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

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

```
\frac{\sqrt{2}}{\log\left(3\right)}
```

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


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

$$/* <![CDATA[*/ \frac{\sqrt{2}}{\log(3)} /*]]> */ \tag{1.6.3}$$

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

```
ZZ
```

```
Integer Ring
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
ZZ
```

```
Integer Ring
```

1.7: Debugging

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

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

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

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

Let's dive in and make some mistakes together!

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
  Cell In[1], line 2
    1message = "Hello, World!"
     ^
SyntaxError: invalid syntax
```

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

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

Let's correct the variable name:

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

```
Hello, World!
```

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


```
print(Hi)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 print(Hi)

NameError: name 'Hi' is not defined
```

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

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

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

```
Hi
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
Hello, World!
```

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

Here is another example of a `NameError`:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 2
      1 l = [Integer(6), Integer(1), Integer(5), Integer(2), Integer(4), Integer(3)]
----> 2 sort(l)

NameError: name 'sort' is not defined
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[1, 2, 3, 4]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

```
Cell In[7], line 2
      1 l = [Integer(1), Integer(2), Integer(3)]
----> 2 l.len()

AttributeError: 'list' object has no attribute 'len'
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
len(l)
```

```
3
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 2
      1 l = [Integer(1), Integer(2), Integer(3)]
----> 2 l.append(Integer(4), Integer(5))
```

```
TypeError: append() takes exactly one argument (2 given)
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[1, 2, 3, 4, 5]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
factorial(-5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 1
----> 1 factorial(-Integer(5))

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/symbolic/function.pyx::
   1177 res = self._evalf_try_(*args)
   1178 if res is None:
-> 1179     res = super(BuiltinFunction, self).__call__(
```

```
   1180             *args, coerce=coerce, hold=hold)
   1181

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/symbolic/function.pyx:(
    598     res = g_function_evalv(self._serial, vec, hold)
    599 elif self._nargs == 1:
--> 600     res = g_function_eval1(self._serial,
    601             (<Expression>args[0])._gobj, hold)
    602 elif self._nargs == 2:

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/functions/other.py:163
   1633 if isinstance(x, Integer):
   1634     try:
-> 1635         return x.factorial()
   1636     except OverflowError:
   1637         return

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/rings/integer.pyx:4530
   4528 """
   4529 if mpz_sgn(self.value) < 0:
-> 4530     raise ValueError("factorial only defined for non-negative integers")
   4531
   4532 if not mpz_fits_ulong_p(self.value):

ValueError: factorial only defined for non-negative integers
```

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


```
factorial(5)
```

```
120
```

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

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

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

```
0
1
2
3
4
5
6
7
8
9
```

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


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

```
1
2
3
```

```
4
5
6
7
8
9
10
```

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

1.8: Documentation

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

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

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

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


```
Set?
```

```
Signature:      Set(X=None)
Docstring:
   Create the underlying set of "X".

   If "X" is a list, tuple, Python set, or "X.is_finite()" is "True",
   this returns a wrapper around Python's enumerated immutable
   "frozenset" type with extra functionality.  Otherwise it returns a
   more formal wrapper.

   If you need the functionality of mutable sets, use Python's builtin
   set type.

   EXAMPLES:

      sage: X = Set(GF(9,'a'))
      sage: X
      {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2}
      sage: type(X)
      <class 'sage.sets.set.Set_object_enumerated_with_category'>
      sage: Y = X.union(Set(QQ))
      sage: Y
      Set-theoretic union of {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2} and Se
```

```
   sage: type(Y)
   <class 'sage.sets.set.Set_object_union_with_category'>

Usually sets can be used as dictionary keys.

   sage: d={Set([2*I,1+I]):10}
   sage: d                  # key is randomly ordered
   {{I + 1, 2*I}: 10}
   sage: d[Set([1+I,2*I])]
   10
   sage: d[Set((1+I,2*I))]
   10

The original object is often forgotten.

   sage: v = [1,2,3]
   sage: X = Set(v)
   sage: X
   {1, 2, 3}
   sage: v.append(5)
   sage: X
   {1, 2, 3}
   sage: 5 in X
   False

Set also accepts iterators, but be careful to only give *finite*
sets:

   sage: sorted(Set(range(1,6)))
   [1, 2, 3, 4, 5]
   sage: sorted(Set(list(range(1,6))))
   [1, 2, 3, 4, 5]
   sage: sorted(Set(iter(range(1,6))))
   [1, 2, 3, 4, 5]

We can also create sets from different types:

   sage: sorted(Set([Sequence([3,1], immutable=True), 5, QQ, Partition([3,1,1])]),
   [5, Rational Field, [3, 1, 1], [3, 1]]

Sets with unhashable objects work, but with less functionality:

   sage: A = Set([QQ, (3, 1), 5])  # hashable
   sage: sorted(A.list(), key=repr)
   [(3, 1), 5, Rational Field]
   sage: type(A)
   <class 'sage.sets.set.Set_object_enumerated_with_category'>
```

```
      sage: B = Set([QQ, [3, 1], 5])  # unhashable
      sage: sorted(B.list(), key=repr)
      Traceback (most recent call last):
      ...
      AttributeError: 'Set_object_with_category' object has no attribute 'list'
      sage: type(B)
      <class 'sage.sets.set.Set_object_with_category'>
Init docstring: Initialize self.  See help(type(self)) for accurate signature.
File:           /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/sets/set.py
Type:           function
```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
Set??
```

```
Signature: Set(X=None)
Docstring:
   Create the underlying set of "X".

   If "X" is a list, tuple, Python set, or "X.is_finite()" is "True",
   this returns a wrapper around Python's enumerated immutable
   "frozenset" type with extra functionality.  Otherwise it returns a
   more formal wrapper.

   If you need the functionality of mutable sets, use Python's builtin
   set type.

   EXAMPLES:

      sage: X = Set(GF(9,'a'))
      sage: X
      {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2}
      sage: type(X)
      <class 'sage.sets.set.Set_object_enumerated_with_category'>
      sage: Y = X.union(Set(QQ))
      sage: Y
```

```
   Set-theoretic union of {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2} and Se
   sage: type(Y)
   <class 'sage.sets.set.Set_object_union_with_category'>

Usually sets can be used as dictionary keys.

   sage: d={Set([2*I,1+I]):10}
   sage: d                  # key is randomly ordered
   {{I + 1, 2*I}: 10}
   sage: d[Set([1+I,2*I])]
   10
   sage: d[Set((1+I,2*I))]
   10

The original object is often forgotten.

   sage: v = [1,2,3]
   sage: X = Set(v)
   sage: X
   {1, 2, 3}
   sage: v.append(5)
   sage: X
   {1, 2, 3}
   sage: 5 in X
   False

Set also accepts iterators, but be careful to only give *finite*
sets:

   sage: sorted(Set(range(1,6)))
   [1, 2, 3, 4, 5]
   sage: sorted(Set(list(range(1,6))))
   [1, 2, 3, 4, 5]
   sage: sorted(Set(iter(range(1,6))))
   [1, 2, 3, 4, 5]

We can also create sets from different types:

   sage: sorted(Set([Sequence([3,1], immutable=True), 5, QQ, Partition([3,1,1])]),
   [5, Rational Field, [3, 1, 1], [3, 1]]

Sets with unhashable objects work, but with less functionality:

   sage: A = Set([QQ, (3, 1), 5])  # hashable
   sage: sorted(A.list(), key=repr)
   [(3, 1), 5, Rational Field]
   sage: type(A)
```

```
      <class 'sage.sets.set.Set_object_enumerated_with_category'>
      sage: B = Set([QQ, [3, 1], 5])  # unhashable
      sage: sorted(B.list(), key=repr)
      Traceback (most recent call last):
      ...
      AttributeError: 'Set_object_with_category' object has no attribute 'list'
      sage: type(B)
      <class 'sage.sets.set.Set_object_with_category'>
Source:
def Set(X=None):
    r"""
    Create the underlying set of ``X``.

    If ``X`` is a list, tuple, Python set, or ``X.is_finite()`` is
    ``True``, this returns a wrapper around Python's enumerated immutable
    ``frozenset`` type with extra functionality.  Otherwise it returns a
    more formal wrapper.

    If you need the functionality of mutable sets, use Python's
    builtin set type.

    EXAMPLES::

        sage: X = Set(GF(9,'a'))
        sage: X
        {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2}
        sage: type(X)
        <class 'sage.sets.set.Set_object_enumerated_with_category'>
        sage: Y = X.union(Set(QQ))
        sage: Y
        Set-theoretic union of {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2} and
        sage: type(Y)
        <class 'sage.sets.set.Set_object_union_with_category'>

    Usually sets can be used as dictionary keys.

    ::

        sage: d={Set([2*I,1+I]):10}
        sage: d                  # key is randomly ordered
        {{I + 1, 2*I}: 10}
        sage: d[Set([1+I,2*I])]
        10
        sage: d[Set((1+I,2*I))]
        10

    The original object is often forgotten.
```

```
::

    sage: v = [1,2,3]
    sage: X = Set(v)
    sage: X
    {1, 2, 3}
    sage: v.append(5)
    sage: X
    {1, 2, 3}
    sage: 5 in X
    False

Set also accepts iterators, but be careful to only give *finite*
sets::

    sage: sorted(Set(range(1,6)))
    [1, 2, 3, 4, 5]
    sage: sorted(Set(list(range(1,6))))
    [1, 2, 3, 4, 5]
    sage: sorted(Set(iter(range(1,6))))
    [1, 2, 3, 4, 5]

We can also create sets from different types::

    sage: sorted(Set([Sequence([3,1], immutable=True), 5, QQ, Partition([3,1,1])])
    [5, Rational Field, [3, 1, 1], [3, 1]]

Sets with unhashable objects work, but with less functionality::

    sage: A = Set([QQ, (3, 1), 5])  # hashable
    sage: sorted(A.list(), key=repr)
    [(3, 1), 5, Rational Field]
    sage: type(A)
    <class 'sage.sets.set.Set_object_enumerated_with_category'>
    sage: B = Set([QQ, [3, 1], 5])  # unhashable
    sage: sorted(B.list(), key=repr)
    Traceback (most recent call last):
    ...
    AttributeError: 'Set_object_with_category' object has no attribute 'list'
    sage: type(B)
    <class 'sage.sets.set.Set_object_with_category'>

TESTS::

    sage: Set(Primes())
    Set of all prime numbers: 2, 3, 5, 7, ...
```

```
        sage: Set(Subsets([1,2,3])).cardinality()
        8
        sage: S = Set(iter([1,2,3])); S
        {1, 2, 3}
        sage: type(S)
        <class 'sage.sets.set.Set_object_enumerated_with_category'>
        sage: S = Set([])
        sage: TestSuite(S).run()

    Check that :trac:`16090` is fixed::

        sage: Set()
        {}
    """
    if X is None:
        X = []
    elif isinstance(X, CategoryObject):
        if isinstance(X, Set_generic):
            return X
        elif X in Sets().Finite():
            return Set_object_enumerated(X)
        else:
            return Set_object(X)

    if isinstance(X, Element) and not isinstance(X, Set_base):
        raise TypeError("Element has no defined underlying set")

    try:
        X = frozenset(X)
    except TypeError:
        return Set_object(X)
    else:
        return Set_object_enumerated(X)
File:      /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/sets/set.py
Type:      function
```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
factor?
```

```
Signature:      factor(n, proof=None, int_=False, algorithm='pari', verbose=0, **kwds
Docstring:
   Return the factorization of "n".  The result depends on the type of
   "n".

   If "n" is an integer, returns the factorization as an object of
   type "Factorization".

   If n is not an integer, "n.factor(proof=proof, **kwds)" gets
   called. See "n.factor??" for more documentation in this case.

   Warning:

     This means that applying "factor" to an integer result of a
     symbolic computation will not factor the integer, because it is
     considered as an element of a larger symbolic ring.EXAMPLES:

        sage: f(n)=n^2
        sage: is_prime(f(3))
        False
        sage: factor(f(3))
        9

   INPUT:

   * "n" - an nonzero integer

   * "proof" - bool or None (default: None)

   * "int_" - bool (default: False) whether to return answers as
     Python ints

   * "algorithm" - string

     * "'pari'" - (default) use the PARI c library

     * "'kash'" - use KASH computer algebra system (requires that kash
       be installed)

     * "'magma'" - use Magma (requires magma be installed)

   * "verbose" - integer (default: 0); PARI's debug variable is set to
     this; e.g., set to 4 or 8 to see lots of output during
     factorization.
```

```
OUTPUT:

* factorization of n

The qsieve and ecm commands give access to highly optimized
implementations of algorithms for doing certain integer
factorization problems. These implementations are not used by the
generic factor command, which currently just calls PARI (note that
PARI also implements sieve and ecm algorithms, but they are not as
optimized). Thus you might consider using them instead for certain
numbers.

The factorization returned is an element of the class
"Factorization"; see Factorization?? for more details, and examples
below for usage. A Factorization contains both the unit factor (+1
or -1) and a sorted list of (prime, exponent) pairs.

The factorization displays in pretty-print format but it is easy to
obtain access to the (prime,exponent) pairs and the unit, to
recover the number from its factorization, and even to multiply two
factorizations. See examples below.

EXAMPLES:

   sage: factor(500)
   2^2 * 5^3
   sage: factor(-20)
   -1 * 2^2 * 5
   sage: f=factor(-20)
   sage: list(f)
   [(2, 2), (5, 1)]
   sage: f.unit()
   -1
   sage: f.value()
   -20
   sage: factor( -next_prime(10^2) * next_prime(10^7) )
   -1 * 101 * 10000019

   sage: factor(-500, algorithm='kash')      # optional - kash
   -1 * 2^2 * 5^3

   sage: factor(-500, algorithm='magma')     # optional - magma
   -1 * 2^2 * 5^3

   sage: factor(0)
   Traceback (most recent call last):
```

```
    ...
    ArithmeticError: factorization of 0 is not defined
    sage: factor(1)
    1
    sage: factor(-1)
    -1
    sage: factor(2^(2^7)+1)
    59649589127497217 * 5704689200685129054721

Sage calls PARI's factor, which has proof False by default. Sage
has a global proof flag, set to True by default (see
"sage.structure.proof.proof", or proof.[tab]). To override the
default, call this function with proof=False.

    sage: factor(3^89-1, proof=False)
    2 * 179 * 1611479891519807 * 504293943956596049162197

    sage: factor(2^197 + 1)  # long time (2s)
    3 * 197002597249 * 1348959352853811313 * 25195157386725301225914401084 3

Any object which has a factor method can be factored like this:

    sage: K.<i> = QuadraticField(-1)
    sage: factor(122 - 454*i)
    (-3*i - 2) * (-i - 2)^3 * (i + 1)^3 * (i + 4)

To access the data in a factorization:

    sage: f = factor(420); f
    2^2 * 3 * 5 * 7
    sage: [x for x in f]
    [(2, 2), (3, 1), (5, 1), (7, 1)]
    sage: [p for p,e in f]
    [2, 3, 5, 7]
    sage: [e for p,e in f]
    [2, 1, 1, 1]
    sage: [p^e for p,e in f]
    [4, 3, 5, 7]

We can factor Python, numpy and gmpy2 numbers:

    sage: factor(math.pi)
    3.141592653589793
    sage: import numpy
    sage: factor(numpy.int8(30))
    2 * 3 * 5
    sage: import gmpy2
```

```
      sage: factor(gmpy2.mpz(30))
      2 * 3 * 5
Init docstring: Initialize self.  See help(type(self)) for accurate signature.
File:           /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/arith/misc.
Type:           function
```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
factor??
```

```
Signature: factor(n, proof=None, int_=False, algorithm='pari', verbose=0, **kwds)
Docstring:
   Return the factorization of "n".  The result depends on the type of
   "n".

   If "n" is an integer, returns the factorization as an object of
   type "Factorization".

   If n is not an integer, "n.factor(proof=proof, **kwds)" gets
   called. See "n.factor??" for more documentation in this case.

   Warning:

     This means that applying "factor" to an integer result of a
     symbolic computation will not factor the integer, because it is
     considered as an element of a larger symbolic ring.EXAMPLES:

        sage: f(n)=n^2
        sage: is_prime(f(3))
        False
        sage: factor(f(3))
        9

   INPUT:

   * "n" - an nonzero integer
```

```
* "proof" - bool or None (default: None)

* "int_" - bool (default: False) whether to return answers as
  Python ints

* "algorithm" - string

  * "'pari'" - (default) use the PARI c library

  * "'kash'" - use KASH computer algebra system (requires that kash
    be installed)

  * "'magma'" - use Magma (requires magma be installed)

* "verbose" - integer (default: 0); PARI's debug variable is set to
  this; e.g., set to 4 or 8 to see lots of output during
  factorization.

OUTPUT:

* factorization of n

The qsieve and ecm commands give access to highly optimized
implementations of algorithms for doing certain integer
factorization problems. These implementations are not used by the
generic factor command, which currently just calls PARI (note that
PARI also implements sieve and ecm algorithms, but they are not as
optimized). Thus you might consider using them instead for certain
numbers.

The factorization returned is an element of the class
"Factorization"; see Factorization?? for more details, and examples
below for usage. A Factorization contains both the unit factor (+1
or -1) and a sorted list of (prime, exponent) pairs.

The factorization displays in pretty-print format but it is easy to
obtain access to the (prime,exponent) pairs and the unit, to
recover the number from its factorization, and even to multiply two
factorizations. See examples below.

EXAMPLES:

   sage: factor(500)
   2^2 * 5^3
   sage: factor(-20)
   -1 * 2^2 * 5
```

```
    sage: f=factor(-20)
    sage: list(f)
    [(2, 2), (5, 1)]
    sage: f.unit()
    -1
    sage: f.value()
    -20
    sage: factor( -next_prime(10^2) * next_prime(10^7) )
    -1 * 101 * 10000019

    sage: factor(-500, algorithm='kash')      # optional - kash
    -1 * 2^2 * 5^3

    sage: factor(-500, algorithm='magma')     # optional - magma
    -1 * 2^2 * 5^3

    sage: factor(0)
    Traceback (most recent call last):
    ...
    ArithmeticError: factorization of 0 is not defined
    sage: factor(1)
    1
    sage: factor(-1)
    -1
    sage: factor(2^(2^7)+1)
    59649589127497217 * 5704689200685129054721

Sage calls PARI's factor, which has proof False by default. Sage
has a global proof flag, set to True by default (see
"sage.structure.proof.proof", or proof.[tab]). To override the
default, call this function with proof=False.

    sage: factor(3^89-1, proof=False)
    2 * 179 * 1611479891519807 * 504293943956596049162197

    sage: factor(2^197 + 1)  # long time (2s)
    3 * 197002597249 * 1348959352853811313 * 251951573867253012259144010843

Any object which has a factor method can be factored like this:

    sage: K.<i> = QuadraticField(-1)
    sage: factor(122 - 454*i)
    (-3*i - 2) * (-i - 2)^3 * (i + 1)^3 * (i + 4)

To access the data in a factorization:

    sage: f = factor(420); f
```

```
      2^2 * 3 * 5 * 7
      sage: [x for x in f]
      [(2, 2), (3, 1), (5, 1), (7, 1)]
      sage: [p for p,e in f]
      [2, 3, 5, 7]
      sage: [e for p,e in f]
      [2, 1, 1, 1]
      sage: [p^e for p,e in f]
      [4, 3, 5, 7]

   We can factor Python, numpy and gmpy2 numbers:

      sage: factor(math.pi)
      3.141592653589793
      sage: import numpy
      sage: factor(numpy.int8(30))
      2 * 3 * 5
      sage: import gmpy2
      sage: factor(gmpy2.mpz(30))
      2 * 3 * 5
Source:
def factor(n, proof=None, int_=False, algorithm='pari', verbose=0, **kwds):
    """
    Return the factorization of ``n``.  The result depends on the
    type of ``n``.

    If ``n`` is an integer, returns the factorization as an object
    of type ``Factorization``.

    If n is not an integer, ``n.factor(proof=proof, **kwds)`` gets called.
    See ``n.factor??`` for more documentation in this case.

    .. warning::

       This means that applying ``factor`` to an integer result of
       a symbolic computation will not factor the integer, because it is
       considered as an element of a larger symbolic ring.

       EXAMPLES::

           sage: f(n)=n^2
           sage: is_prime(f(3))
           False
           sage: factor(f(3))
           9

    INPUT:
```

```
-  ``n`` - an nonzero integer

-  ``proof`` - bool or None (default: None)

-  ``int_`` - bool (default: False) whether to return
   answers as Python ints

-  ``algorithm`` - string

   - ```'pari'`` - (default) use the PARI c library

   - ``'kash'`` - use KASH computer algebra system (requires that
     kash be installed)

   - ```'magma'`` - use Magma (requires magma be installed)

-  ``verbose`` - integer (default: 0); PARI's debug
   variable is set to this; e.g., set to 4 or 8 to see lots of output
   during factorization.

OUTPUT:

-  factorization of n

The qsieve and ecm commands give access to highly optimized
implementations of algorithms for doing certain integer
factorization problems. These implementations are not used by the
generic factor command, which currently just calls PARI (note that
PARI also implements sieve and ecm algorithms, but they are not as
optimized). Thus you might consider using them instead for certain
numbers.

The factorization returned is an element of the class
:class:`~sage.structure.factorization.Factorization`; see Factorization??
for more details, and examples below for usage. A Factorization contains
both the unit factor (+1 or -1) and a sorted list of (prime, exponent)
pairs.

The factorization displays in pretty-print format but it is easy to
obtain access to the (prime,exponent) pairs and the unit, to
recover the number from its factorization, and even to multiply two
factorizations. See examples below.

EXAMPLES::

    sage: factor(500)
```

```
        2^2 * 5^3
        sage: factor(-20)
        -1 * 2^2 * 5
        sage: f=factor(-20)
        sage: list(f)
        [(2, 2), (5, 1)]
        sage: f.unit()
        -1
        sage: f.value()
        -20
        sage: factor( -next_prime(10^2) * next_prime(10^7) )
        -1 * 101 * 10000019

    ::

        sage: factor(-500, algorithm='kash')      # optional - kash
        -1 * 2^2 * 5^3

    ::

        sage: factor(-500, algorithm='magma')     # optional - magma
        -1 * 2^2 * 5^3

    ::

        sage: factor(0)
        Traceback (most recent call last):
        ...
        ArithmeticError: factorization of 0 is not defined
        sage: factor(1)
        1
        sage: factor(-1)
        -1
        sage: factor(2^(2^7)+1)
        59649589127497217 * 5704689200685129054721

    Sage calls PARI's factor, which has proof False by default.
    Sage has a global proof flag, set to True by default (see
    :mod:`sage.structure.proof.proof`, or proof.[tab]). To override
    the default, call this function with proof=False.

    ::

        sage: factor(3^89-1, proof=False)
        2 * 179 * 1611479891519807 * 504293943956596049162197

    ::
```

```
    sage: factor(2^197 + 1)  # long time (2s)
    3 * 197002597249 * 1348959352853811313 * 25195157386725301225914401084 3

Any object which has a factor method can be factored like this::

    sage: K.<i> = QuadraticField(-1)
    sage: factor(122 - 454*i)
    (-3*i - 2) * (-i - 2)^3 * (i + 1)^3 * (i + 4)

To access the data in a factorization::

    sage: f = factor(420); f
    2^2 * 3 * 5 * 7
    sage: [x for x in f]
    [(2, 2), (3, 1), (5, 1), (7, 1)]
    sage: [p for p,e in f]
    [2, 3, 5, 7]
    sage: [e for p,e in f]
    [2, 1, 1, 1]
    sage: [p^e for p,e in f]
    [4, 3, 5, 7]

We can factor Python, numpy and gmpy2 numbers::

    sage: factor(math.pi)
    3.141592653589793
    sage: import numpy
    sage: factor(numpy.int8(30))
    2 * 3 * 5
    sage: import gmpy2
    sage: factor(gmpy2.mpz(30))
    2 * 3 * 5

TESTS::

    sage: factor(Mod(4, 100))
    Traceback (most recent call last):
    ...
    TypeError: unable to factor 4
    sage: factor("xyz")
    Traceback (most recent call last):
    ...
    TypeError: unable to factor 'xyz'
"""
try:
    m = n.factor
```

```
    except AttributeError:
        # Maybe n is not a Sage Element, try to convert it
        e = py_scalar_to_element(n)
        if e is n:
            # Either n was a Sage Element without a factor() method
            # or we cannot it convert it to Sage
            raise TypeError("unable to factor {!r}".format(n))
        n = e
        m = n.factor

    if isinstance(n, Integer):
        return m(proof=proof, algorithm=algorithm, int_=int_,
                 verbose=verbose, **kwds)

    # Polynomial or other factorable object
    try:
        return m(proof=proof, **kwds)
    except TypeError:
        # Maybe the factor() method does not have a proof option
        return m(**kwds)
File:      /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/arith/misc.py
Type:      function
```

1.9: Miscellaneous Features

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

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

1.9.1 Reading and Writing Files in Sage

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

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

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

Writing to a file:


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

Reading from a file:

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

```
Hello, Sage!
```

1.9.2 Executing Shell Commands in Sage

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
!ls -la
```

```
total 96
drwxr-x---. 1 jovyan jovyan   115 May  6 22:12 .
drwxr-xr-x. 1 root   root      20 May  5 05:56 ..
-rw-r--r--. 1 jovyan jovyan  1961 May  5 05:40 apt.txt
-rw-r--r--. 1 jovyan jovyan    88 May  5 05:40 AUTHORS
-rw-r--r--. 1 jovyan jovyan   220 Jan  6  2022 .bash_logout
-rw-r--r--. 1 jovyan jovyan  3771 Jan  6  2022 .bashrc
drwxr-xr-x. 1 jovyan jovyan    18 May  5 06:02 .cache
drwxrwsr-x. 2 jovyan jovyan    30 May  5 05:49 .conda
drwx------. 3 jovyan jovyan    25 May  6 19:55 .config
-rw-r--r--. 1 jovyan jovyan  4707 May  5 05:40 environment.yml
drwxr-xr-x. 8 jovyan jovyan   180 May  5 05:40 .git
drwxr-xr-x. 1 jovyan jovyan    29 May  6 19:55 .ipython
drwxr-xr-x. 1 jovyan jovyan    22 May  5 06:02 .jupyter
-rw-r--r--. 1 jovyan jovyan 16263 May  6 22:12 .jupyter-server-log.txt
-rw-r--r--. 1 jovyan jovyan  1081 May  5 05:40 LICENSE
-rw-r--r--. 1 jovyan jovyan 22179 May  5 06:01 Manifest.toml
-rw-r--r--. 1 jovyan jovyan    12 May  6 22:12 output.txt
-rwxr-xr-x. 1 jovyan jovyan   849 May  5 05:40 postBuild
-rw-r--r--. 1 jovyan jovyan   807 Jan  6  2022 .profile
-rw-r--r--. 1 jovyan jovyan   414 May  5 05:40 Project.toml
-rw-r--r--. 1 jovyan jovyan 10496 May  5 05:40 README.md
```

```
drwxr-xr-x. 8 jovyan jovyan    95 May  6 20:06 .sage
drwxr-xr-x. 3 jovyan jovyan    17 May  5 06:03 .yarn
```

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

Sage supports structured data formats such as CSV and JSON.

Generating a CSV file using shell command:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

Reading a CSV file in Sage:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
import csv

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

```
['Name', 'Age', 'Country']
['Alice', '25', 'USA']
```

```
['Bob', '30', 'UK']
['Charlie', '28', 'Canada']
```

1.9.4 Using External Libraries in Sage

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

Using NumPy for numerical computations:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[1 2 3]
```

1.10: Run Sage in the browser

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

1. Navigate to Sage website.
2. Click on Sage on CoCalc.
3. Create a CoCalc account.
4. Go to Your Projects on CoCalc and create a new project.
5. Start your new project and create a new worksheet. Choose the SageMath Worksheet option.
6. Enter Sage code into the worksheet. Try to evaluate a simple expression and use the worksheet like a calculator. Execute the code by clicking `Run` or using the shortcut `Shift + Enter` . We will learn more ways to run code in the next section.
7. Save your worksheet as a PDF for your records.
8. To learn more about Sage worksheets, refer to the documentation.
9. Alternatively, you can run Sage code in a Jupyter Notebook for additional features.
10. If you are feeling adventurous, you can install Sage and run it locally on your own computer. Keep in mind that a local install will be the most involved way to run Sage code. When using Sage locally, commands to display graphics will create and then open a temporary file, which can be saved permanently through the software used to view it.

CHAPTER OVERVIEW

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

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[*/ \{\texttt{Feb}, \texttt{Jan}, \texttt{Aug}, \texttt{Apr}, \texttt{Jul}, \texttt{Dec}, \texttt{Mar}, \texttt{Nov}, \texttt{Sep}, \texttt{May}, \texttt{Jun}, \texttt{Oct}\} /*]]> */ \tag{2.1.1}$$

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

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

```
True
```

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{2, 4, 6, 8, 10}
```

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 
```

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

subsets_of_weather.list()
```

```
[{},
 {'Snow'},
 {'Hurricane'},
 {'Tornado'},
 {'Cloud'},
 {'Rain'},
 {'Sun'},
 {'Snow', 'Hurricane'},
 {'Snow', 'Tornado'},
 {'Snow', 'Cloud'},
 {'Snow', 'Rain'},
 {'Snow', 'Sun'},
 {'Hurricane', 'Tornado'},
 {'Cloud', 'Hurricane'},
 {'Hurricane', 'Rain'},
 {'Hurricane', 'Sun'},
 {'Cloud', 'Tornado'},
 {'Rain', 'Tornado'},
 {'Sun', 'Tornado'},
 {'Cloud', 'Rain'},
 {'Cloud', 'Sun'},
 {'Rain', 'Sun'},
 {'Snow', 'Hurricane', 'Tornado'},
```

```
{'Snow', 'Cloud', 'Hurricane'},
{'Snow', 'Hurricane', 'Rain'},
{'Snow', 'Hurricane', 'Sun'},
{'Snow', 'Cloud', 'Tornado'},
{'Snow', 'Rain', 'Tornado'},
{'Snow', 'Sun', 'Tornado'},
{'Snow', 'Cloud', 'Rain'},
{'Snow', 'Cloud', 'Sun'},
{'Snow', 'Rain', 'Sun'},
{'Cloud', 'Hurricane', 'Tornado'},
{'Hurricane', 'Rain', 'Tornado'},
{'Hurricane', 'Sun', 'Tornado'},
{'Cloud', 'Hurricane', 'Rain'},
{'Cloud', 'Hurricane', 'Sun'},
{'Hurricane', 'Rain', 'Sun'},
{'Cloud', 'Rain', 'Tornado'},
{'Cloud', 'Sun', 'Tornado'},
{'Rain', 'Sun', 'Tornado'},
{'Cloud', 'Rain', 'Sun'},
{'Snow', 'Cloud', 'Hurricane', 'Tornado'},
{'Snow', 'Hurricane', 'Rain', 'Tornado'},
{'Snow', 'Hurricane', 'Sun', 'Tornado'},
{'Snow', 'Cloud', 'Hurricane', 'Rain'},
{'Snow', 'Cloud', 'Hurricane', 'Sun'},
{'Snow', 'Hurricane', 'Rain', 'Sun'},
{'Snow', 'Cloud', 'Rain', 'Tornado'},
{'Snow', 'Cloud', 'Sun', 'Tornado'},
{'Snow', 'Rain', 'Sun', 'Tornado'},
{'Snow', 'Cloud', 'Rain', 'Sun'},
{'Cloud', 'Hurricane', 'Rain', 'Tornado'},
{'Cloud', 'Hurricane', 'Sun', 'Tornado'},
{'Hurricane', 'Rain', 'Sun', 'Tornado'},
{'Cloud', 'Hurricane', 'Rain', 'Sun'},
{'Cloud', 'Rain', 'Sun', 'Tornado'},
{'Snow', 'Hurricane', 'Rain', 'Tornado', 'Cloud'},
{'Snow', 'Hurricane', 'Tornado', 'Cloud', 'Sun'},
{'Snow', 'Hurricane', 'Rain', 'Tornado', 'Sun'},
{'Snow', 'Hurricane', 'Rain', 'Cloud', 'Sun'},
{'Snow', 'Rain', 'Tornado', 'Cloud', 'Sun'},
{'Hurricane', 'Rain', 'Tornado', 'Cloud', 'Sun'},
{'Snow', 'Hurricane', 'Rain', 'Tornado', 'Cloud', 'Sun'}]
```

Set Membership Check

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

```
"earthquake" in W
```

```
False
```

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


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

```
True
```

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

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

```
False
```

2.2: Cardinality

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
5
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
5
```

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()
```

```
+Infinity
```



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

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/structure/category_obj
    838 try:
--> 839     return self.__cached_methods[name]
    840 except KeyError:

KeyError: 'len'

During handling of the above exception, another exception occurred:

AttributeError                            Traceback (most recent call last)
Cell In[4], line 4
      1 # This results in an error because the
      2 # Python len() function is not defined for the primes class
      3 P = Primes()
```

```
----> 4 P.len()

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/structure/category_obj
    831         AttributeError: 'PrimeNumbers_with_category' object has no attribute
    832     """
--> 833     return self.getattr_from_category(name)
    834
    835 cdef getattr_from_category(self, name):

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/structure/category_obj
    846     cls = self._category.parent_class
    847
--> 848 attr = getattr_from_other_class(self, cls, name)
    849 self.__cached_methods[name] = attr
    850 return attr

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/cpython/getattr.pyx:36
    365     dummy_error_message.cls = type(self)
    366     dummy_error_message.name = name
--> 367     raise AttributeError(dummy_error_message)
    368 cdef PyObject* attr = instance_getattr(cls, name)
    369 if attr is NULL:

AttributeError: 'Primes_with_category' object has no attribute 'len'
```

2.3: Operations on Sets

Union of Sets

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

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

> **Note**
>
> 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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{1, 2, 3, 4, 5, 6}
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{1, 2, 3, 4, 5, 6}
```

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 $\boldsymbol{A}\cap\boldsymbol{B}$.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{3, 4, 5}
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{3, 4, 5}
```

Difference of Sets

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{1, 2}
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{1, 2}
```

Multiple Sets

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

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

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

```
{1, 2, 3, 4, 5, 6, 7}
```

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
```

```
{1, 2, 3, 4, 5, 6, 7}
```

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

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 $\boldsymbol{A}$.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{8, 9, 6, 7}
```

Alternatively, we can use the `-` operator.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
{8, 9, 6, 7}
```

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 $\boldsymbol{A \times D}$.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
Set of elements of The Cartesian product of ({1, 2, 3, 4, 5}, {'x', 'y'})
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
Set of elements of The Cartesian product of ({1, 2, 3, 4, 5}, {'x', 'y'})
```

Power Sets

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[{},
 {'e'},
 {'u'},
 {'a'},
 {'i'},
 {'o'},
 {'e', 'u'},
 {'a', 'e'},
 {'i', 'e'},
 {'e', 'o'},
 {'a', 'u'},
 {'i', 'u'},
 {'u', 'o'},
 {'a', 'i'},
 {'a', 'o'},
 {'i', 'o'},
 {'a', 'e', 'u'},
 {'i', 'e', 'u'},
 {'e', 'u', 'o'},
 {'a', 'i', 'e'},
 {'a', 'e', 'o'},
 {'i', 'e', 'o'},
 {'a', 'i', 'u'},
 {'a', 'u', 'o'},
 {'i', 'u', 'o'},
 {'a', 'i', 'o'},
 {'a', 'i', 'e', 'u'},
 {'a', 'e', 'u', 'o'},
 {'i', 'e', 'u', 'o'},
 {'a', 'i', 'e', 'o'},
 {'a', 'i', 'u', 'o'},
 {'e', 'u', 'a', 'i', 'o'}]
```

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

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

```
True
```

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


```
V in S
```

```
True
```

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


```
S.cardinality()
```

```
32
```

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

About hashable objects:

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[[], [1], [[2, 3]], [1, [2, 3]], [4], [1, 4], [[2, 3], 4], [1, [2, 3], 4]]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
P = powerset(ZZ)

i = 0
```

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

```
[]
[0]
[1]
[0, 1]
[-1]
[0, -1]
[1, -1]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
Subsets of Integer Ring
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
P.random_element()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/rings/integer.pyx:6524
   6523 try:
-> 6524     y = Integer(y)
   6525 except TypeError:

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/rings/integer.pyx:747,
    746
--> 747                 raise TypeError("unable to coerce %s to an integer" % type(x)
    748

TypeError: unable to coerce <class 'sage.rings.infinity.PlusInfinity'> to an integer

During handling of the above exception, another exception occurred:

TypeError                                 Traceback (most recent call last)
Cell In[22], line 1
----> 1 P.random_element()

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/combinat/subset.py:427
    406 def random_element(self):
    407     """
    408     Return a random element of the class of subsets of ``s`` (in other
    409     words, a random subset of ``s``).
   (...)
    425         True
    426     """
--> 427     k = ZZ.random_element(0, self.cardinality())
    428     return self.unrank(k)

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/combinat/subset.py:354
    329 def cardinality(self):
    330     r"""
    331     Return the number of subsets of the set ``s``.
    332
   (...)
    352         True
    353     """
--> 354     return Integer(1) << self._s.cardinality()

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/rings/integer.pyx:6587
   6585     if not isinstance(x, Integer):
   6586         return x << int(y)
```

```
-> 6587     return (<Integer>x)._shift_helper(y, 1)
   6588
   6589 def __rshift__(x, y):

File /srv/conda/envs/notebook/lib/python3.8/site-packages/sage/rings/integer.pyx:6526
   6524     y = Integer(y)
   6525 except TypeError:
-> 6526     raise TypeError("unsupported operands for %s: %s, %s"%(("<<" if sign == 1
   6527 except ValueError:
   6528     return coercion_model.bin_op(self, y, operator.lshift if sign == 1 else o

TypeError: unsupported operands for <<: 1, +Infinity
```

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

Viewing Power Sets

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


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

```
128
```

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

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

```
[{}, {'red'}, {'indigo'}, {'green'}, {'violet'}]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[{'orange'}, {'blue'}, {'yellow'}, {'red', 'indigo'}, {'red', 'green'}]
```

CHAPTER OVERVIEW

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

Factorial Function

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

Compute the factorial of :5:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
factorial(5)
```

```
120
```

Combinations

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

Note

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
10
```

List the combinations:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[[0, 1, 2],
 [0, 1, 3],
 [0, 1, 4],
 [0, 2, 3],
 [0, 2, 4],
 [0, 3, 4],
 [1, 2, 3],
 [1, 2, 4],
 [1, 3, 4],
 [2, 3, 4]]
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
10
```

3.1.3 Permutations

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

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
60
```

List the permutations:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 2],
 [1, 3, 4],
```

```
[1, 3, 5],
[1, 4, 2],
[1, 4, 3],
[1, 4, 5],
[1, 5, 2],
[1, 5, 3],
[1, 5, 4],
[2, 1, 3],
[2, 1, 4],
[2, 1, 5],
[2, 3, 1],
[2, 3, 4],
[2, 3, 5],
[2, 4, 1],
[2, 4, 3],
[2, 4, 5],
[2, 5, 1],
[2, 5, 3],
[2, 5, 4],
[3, 1, 2],
[3, 1, 4],
[3, 1, 5],
[3, 2, 1],
[3, 2, 4],
[3, 2, 5],
[3, 4, 1],
[3, 4, 2],
[3, 4, 5],
[3, 5, 1],
[3, 5, 2],
[3, 5, 4],
[4, 1, 2],
[4, 1, 3],
[4, 1, 5],
[4, 2, 1],
[4, 2, 3],
[4, 2, 5],
[4, 3, 1],
[4, 3, 2],
[4, 3, 5],
[4, 5, 1],
[4, 5, 2],
[4, 5, 3],
[5, 1, 2],
[5, 1, 3],
[5, 1, 4],
[5, 2, 1],
```

```
 [5, 2, 3],
 [5, 2, 4],
 [5, 3, 1],
 [5, 3, 2],
 [5, 3, 4],
 [5, 4, 1],
 [5, 4, 2],
 [5, 4, 3]]
```

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

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
6
```

List the permutations:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

The following is an example of permutations of specified elements:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

A.list()
```

```
[['a', 'b', 'c'],
 ['a', 'c', 'b'],
 ['b', 'a', 'c'],
 ['b', 'c', 'a'],
 ['c', 'a', 'b'],
 ['c', 'b', 'a']]
```

Choose 2:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

A.list()
```

```
[['a', 'b'], ['a', 'c'], ['b', 'a'], ['b', 'c'], ['c', 'a'], ['c', 'b']]
```

CHAPTER OVERVIEW

4: Logic

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

4.1: Logical Operators

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

Name	Sage Operator	Mathematical Notation
AND	`&`	∧
OR	`|`	∨
NOT	`~`	¬
Conditional	`->`	→
Biconditional	`<->`	↔

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[*/ (p \wedge q) \vee (\neg p) /*]]> */ \tag{4.1.1}$$

4.2: Truth Tables

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

> **Note**
>
> Truth tables aid in the design of digital circuits.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
p      q      value
False  False  True
False  True   True
True   False  False
True   True   True
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
p     | q     | value |
------------------------
False | False | True  |
False | True  | True  |
True  | False | False |
True  | True  | True  |
```

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

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


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

```
p      q      r      value
False  False  False  True
False  False  True   True
False  True   False  True
False  True   True   True
True   False  False  True
True   False  True   True
True   True   False  False
True   True   True   True
```

4.3: Analyzing Logical Equivalences

Equivalent Statements

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
True
```

Do not attempt to compare equivalence of truth tables.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
False
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
True
```

Tautologies

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

Note

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
True
```

Contradictions

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

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

```
True
```

CHAPTER OVERVIEW

5: Relations

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

5.1: Introduction to Relations

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

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

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

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

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

```
R = {(2, 1), (4, 2)}
Is R a relation from set A to set B? True
```

5.1.1 Relation Composition

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

We can also use Sage to compose relations.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

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

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

$$/* <![CDATA[/* \{(2,9),(4,8)\} /*]] > */ \tag{5.1.1}$$

5.1.2 Relations On a Set

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

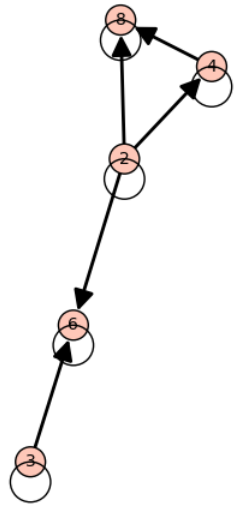
$$/* <![CDATA[/* \{(4,4),(8,8),(2,4),(3,3),(2,6),(3,6),(4,8),(2,2),(6,6),(2,8)\} /*]] > */ \tag{5.1.2}$$

5.2: Digraphs

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


```
# 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)
```

Looped digraph on 5 vertices



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

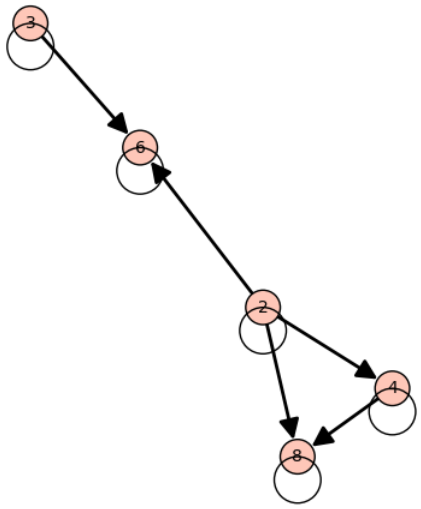
We can add a title to the digraph with the `name` parameter.

Note

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

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

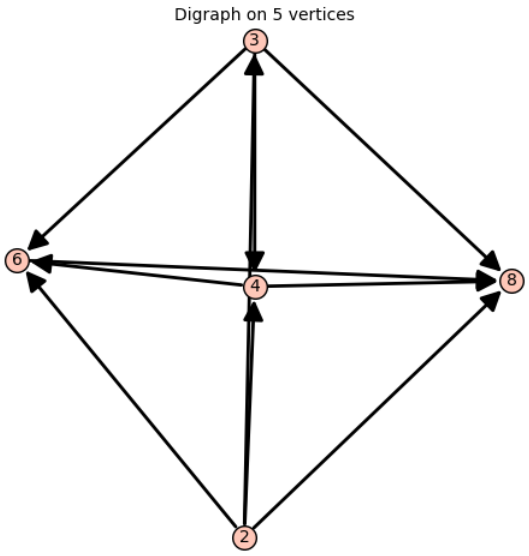
Look at my digraph: Looped digraph on 5 vertices



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

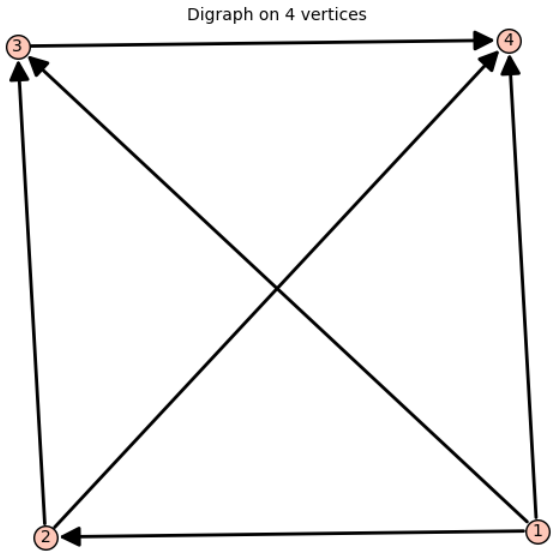

```
# Define the set A
A = Set([2, 3, 4, 6, 8])
```

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



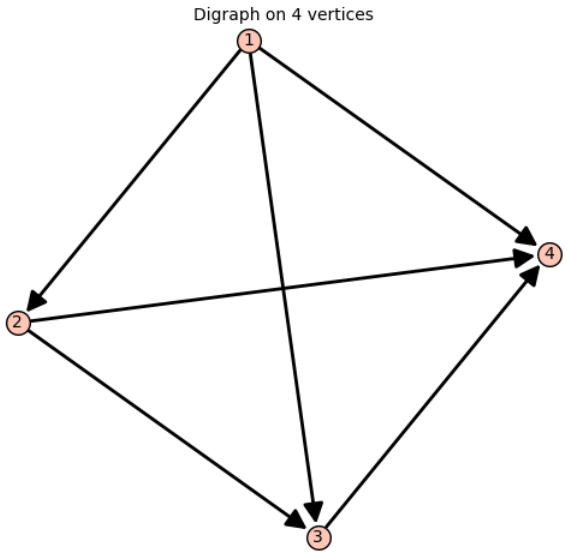
We can also define the digraph using pair notion for relations.


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



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


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



5.3: Properties

A relation on A may satisfy certain properties:

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

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

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[/* \{(2,3),(1,2),(3,3),(2,2),(1,1)\} /*]] > */ \tag{5.3.1}$$

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
True
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
True
```

Symmetric

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
True
```

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` .

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
True
```

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)
```

```
True
```

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()
```

```
False
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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 n
        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)
```

```
True
```

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` :

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
True
```

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

D.is_transitive()
```

```
True
```

5.4: Equivalence

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

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

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

Consider a set A defined as:

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
# 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
```

```
{'p_8', 'p_6', 'p_1', 'p_2', 'p_10', 'p_7', 'p_9', 'p_3', 'p_4', 'p_5'}
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
{'first_floor': {'p_3', 'p_4', 'p_1', 'p_2'},
 'second_floor': {'p_6', 'p_5', 'p_7'},
 'third_floor': {'p_10', 'p_8', 'p_9'}}
```

Let R be the relation on A described as follows:

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


```
# 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
R
```

```
{('p_3', 'p_1'), ('p_7', 'p_7'), ('p_4', 'p_1'), ('p_4', 'p_3'), ('p_6', 'p_5'), ('p_
```

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)
```

```
True
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
True
```

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

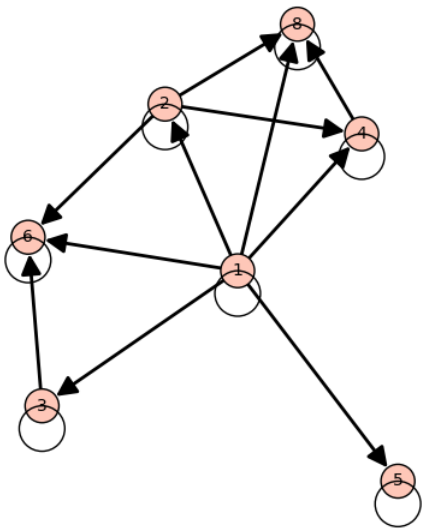
```
G = DiGraph(list(R), loops=true)
G.is_transitive()
```

```
True
```

5.5: Partial Order

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


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

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

D = DiGraph(R, loops=True)

plot(D)
```



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

Note

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

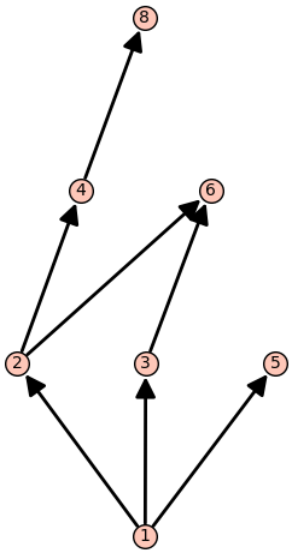
If R is a partial order relation on A, then the function `Poset((A, R))` computes the Hasse diagram associated to R.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

```
P.cover_relations()
```

```
[[1, 2], [1, 3], [1, 5], [2, 4], [2, 6], [4, 8], [3, 6]]
```

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


```
P.has_bottom()
```

```
True
```

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


```
P.has_top()
```

```
False
```

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

```
P.bottom()
```

```
1
```

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


```
P.top()
```

```

```

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

5.6: Relations in Action

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
firebrick
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
True
```

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

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

```
RGB color (0.9333333333333333, 0.5098039215686274, 0.9333333333333333)
```

5.6.1 Color Complements

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

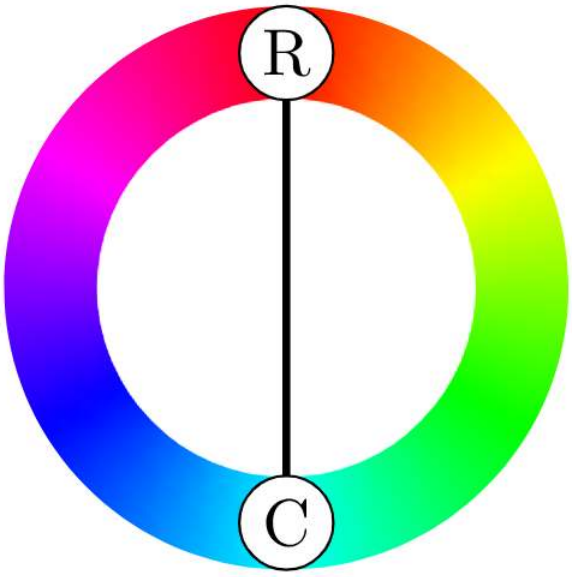


Figure **5.6.1.** On RGB displays, red and cyan are complementary colors.

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

Login

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

```
(0.8333333333333334, 0.7605633802816902, 0.7215686274509804)
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
0.8333333333333334
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
0.8333333333333334
0.3333333333333333
Is complementary pair: False
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
5/6
1/3
Is complementary pair: True
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
    return m_hue == k_hue + 1/2

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

```
[('paleturquoise', 'firebrick'),
 ('paleturquoise', 'maroon'),
 ('violet', 'green'),
 ('teal', 'firebrick'),
 ('teal', 'maroon')]
```

5.6.2 Light and Dark Modes

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
0.7215686274509804
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
0.9725490196078431
```

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


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

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

```
[(('paleturquoise', 'maroon'), ('paleturquoise', 'firebrick')),
 (('violet', 'green'), ('paleturquoise', 'firebrick')),
 (('violet', 'green'), ('paleturquoise', 'maroon')),
 (('teal', 'firebrick'), ('paleturquoise', 'firebrick')),
 (('teal', 'firebrick'), ('paleturquoise', 'maroon')),
 (('teal', 'firebrick'), ('violet', 'green')),
 (('teal', 'maroon'), ('paleturquoise', 'firebrick')),
 (('teal', 'maroon'), ('paleturquoise', 'maroon')),
 (('teal', 'maroon'), ('violet', 'green')),
 (('teal', 'maroon'), ('teal', 'firebrick'))]
```

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

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

```
The highest lightness pair is: ('paleturquoise', 'firebrick')
The lowest lightness pair is: ('teal', 'maroon')
```

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

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

print('Dark-mode hex codes are:', highest_hex)
(circle((0,0),1, color=highest_pair[0], fill=true) + circle((2.5,0),1, color=highest_
```

```
print('Light-mode hex codes are:', lowest_hex)
(circle((0,0),1, color=lowest_pair[0], fill=true) + circle((2.5,0),1, color=lowest_pa
```

```
Dark-mode hex codes are: ('#afeeee', '#b22222')
```

CHAPTER OVERVIEW

6: Functions

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

6.1: Functions

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

In Sage, functions can be defined using direct definition.

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

$$/* <![CDATA[*/ x \mapsto x^3 /*]]> */ \tag{6.1.1}$$

Graphical Representations

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

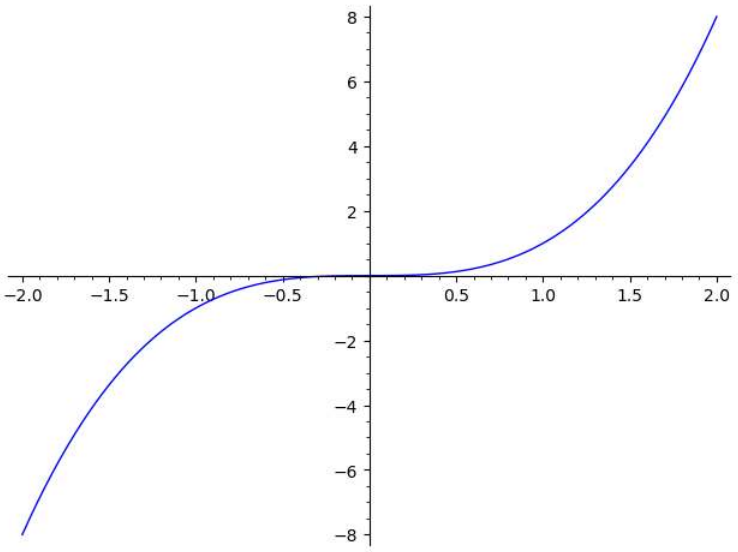
For example, to plot the function $f(x) = x^3$ over the interval :[−2,2]:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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



CHAPTER OVERVIEW

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

Graph Definition

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

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

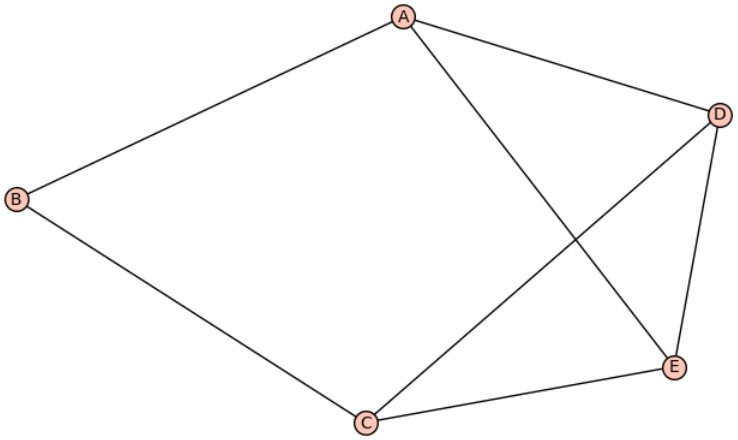
The set of edges is a set whose elements are subsets of two vertices.

Terminology:

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

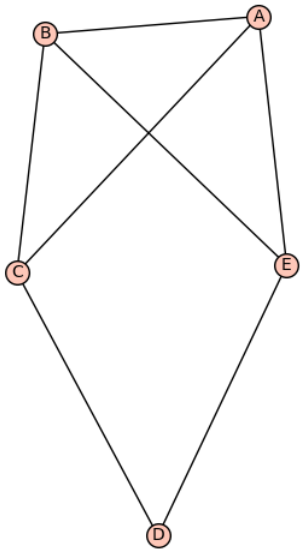
There are several ways to define a graph in Sage. We can define a graph by listing the vertices and edges:


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



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

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



We can define a graph with an edge dictionary like so: `{edge: [neighbor, neighbor, etc], edge: [ neighbor, etc], etc: [etc]}`

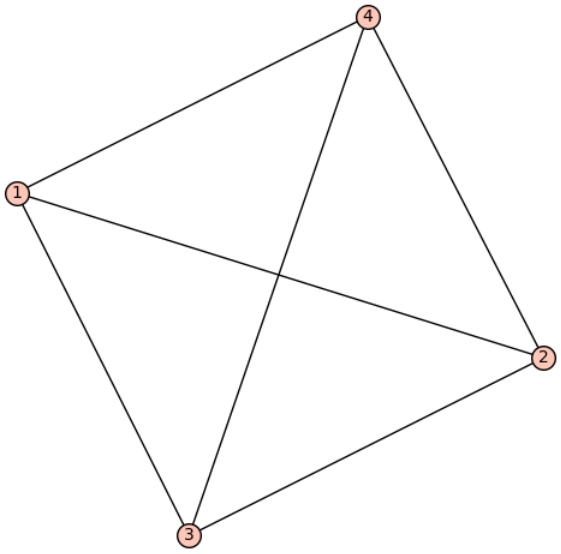
Each dictionary key is a vertex. The dictionary values are the vertex neighbors.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

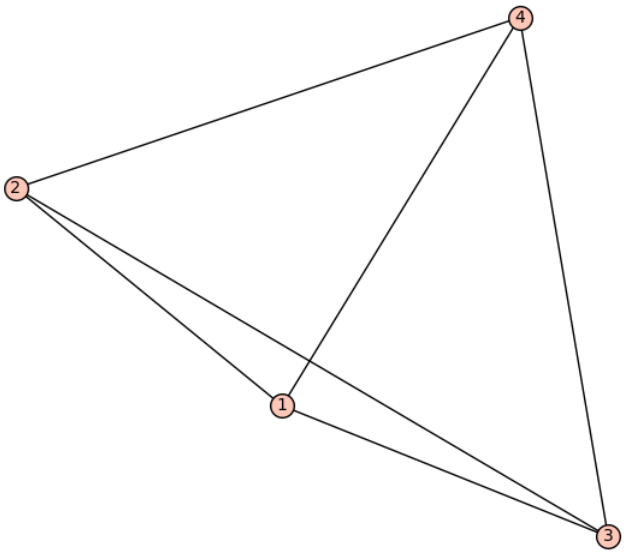
Login

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



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


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



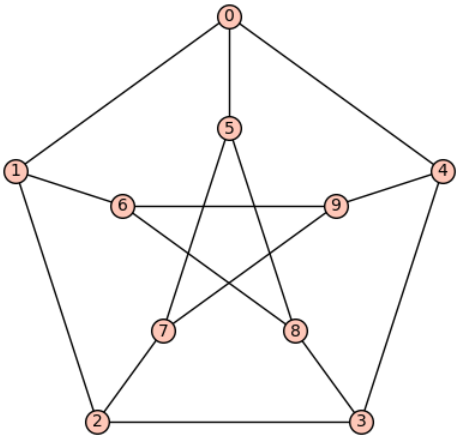
Sage offers a collection of predefined graphs. Here are some examples:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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



Note

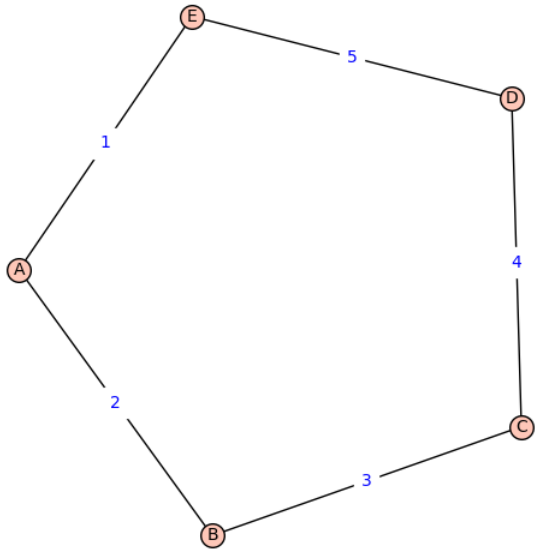
Concepts from graph theory have practical applications related to social networks, computer networks, transportation, biology, chemistry, and more.

Weighted Graphs

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

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

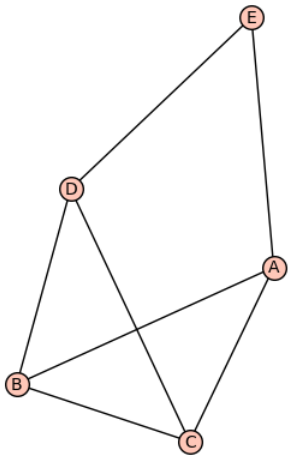

```
E = [('A', 'B', 2), ('B', 'C', 3), ('C', 'D', 4), ('D', 'E', 5), ('E', 'A', 1)]
G = Graph(E, weighted=True)
G.plot(edge_labels=True)
```



Graph Characteristics

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

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



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


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

```
['A', 'B', 'C', 'D', 'E']
```

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

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

```
[('A', 'B', None), ('A', 'C', None), ('A', 'E', None), ('B', 'C', None), ('B', 'D', N
```

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


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

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

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

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

```
5
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
7
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
3
```

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

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

```
[3, 3, 3, 3, 2]
```

Graphs and Matrices

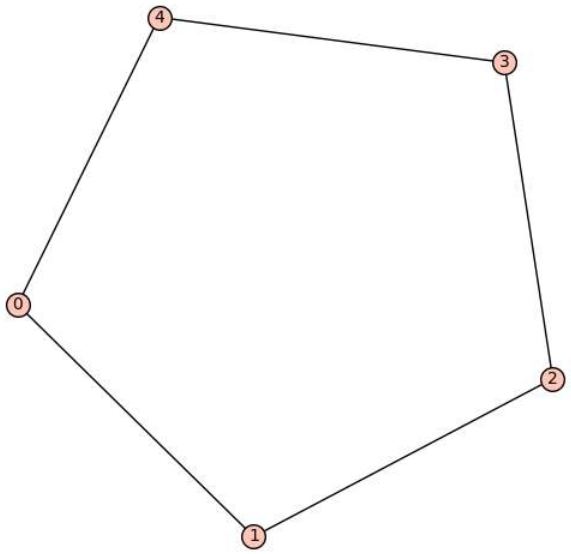
The *adjacency matrix* of a graph is a square matrix used to represent which vertices of the graph are adjacent to which other vertices. Each entry a_{ij} in the matrix is equal to 1 if there is an edge from vertex `i` to vertex `j`, and is equal to 0 otherwise.


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

```
[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]
```

We can also define a graph with an adjacency matrix:

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



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

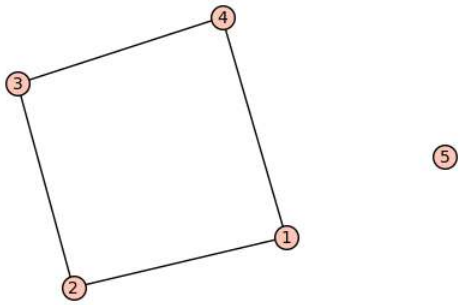
```
G = Graph([('A', 'B'), ('A', 'E'), ('B', 'C'), ('C', 'D'), ('D', 'E')])
G.incidence_matrix()
```

```
[1 1 0 0 0]
[1 0 1 0 0]
[0 0 1 0 1]
[0 0 0 1 1]
[0 1 0 1 0]
```

Manipulating Graphs in Sage

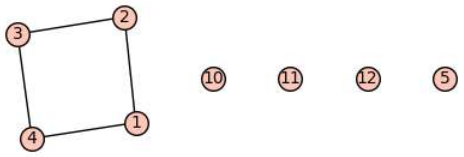
Add a vertex to a graph:


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



Add a list of vertices:

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



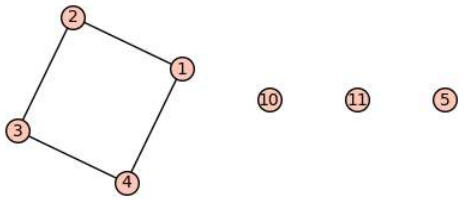
Remove a vertex from a graph:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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



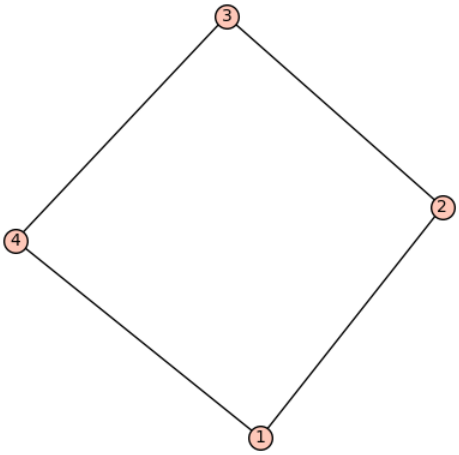
Remove a list of vertices from a graph:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

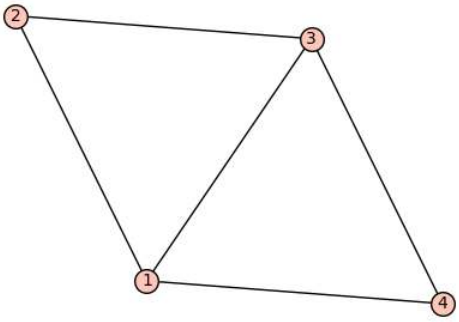
Login

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



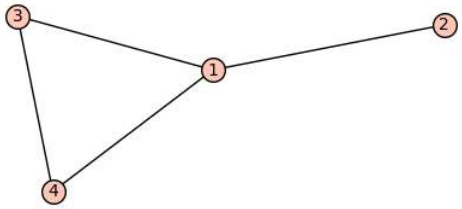
Add an edge between two vertices:


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



Delete an edge from a graph:

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



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

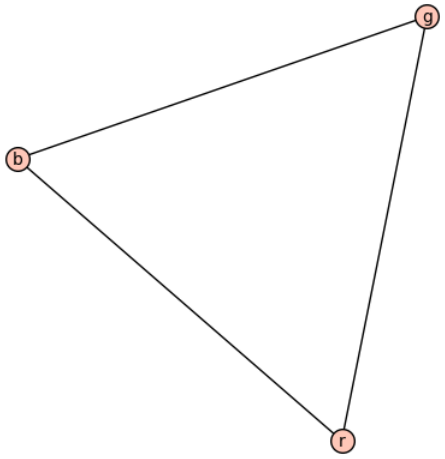
7.2: Plot Options

The `show()` method displays the graphics object immediately with default settings. The `plot()` method accepts options for customizing the presentation of the graphics object. You can import more features from Matplotlib or LaTeX for fine-tuned customization options. Let's examine how the plotting options improve the presentation and help us discover insights into the structure and properties of a graph. The presentation of a Sage graphics object may differ depending on your environment.

Size

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


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

Graph(E).show()
```



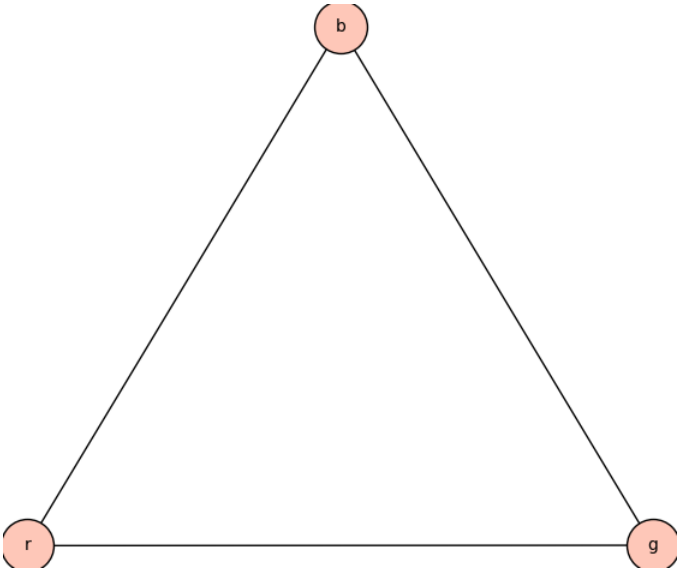
Let's increase the `vertex_size` :

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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



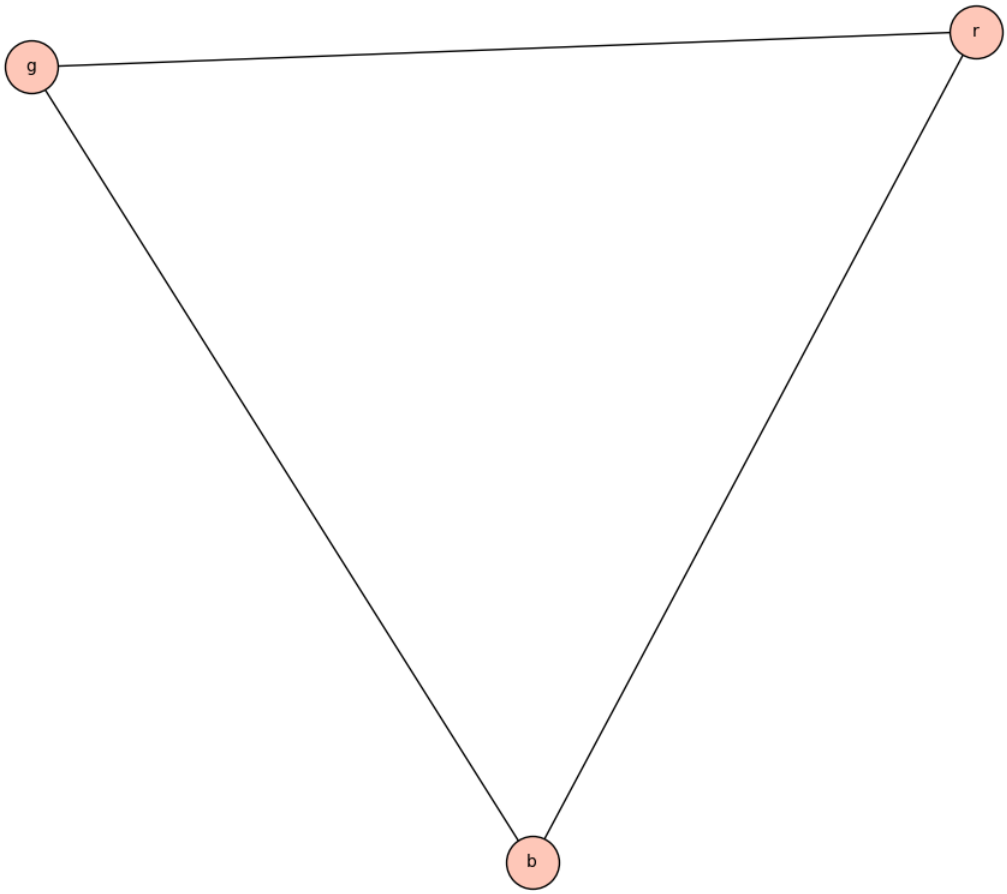
Resolve the cropping by increasing the `figsize` . Specify a single number or a `(width, height)` tuple.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

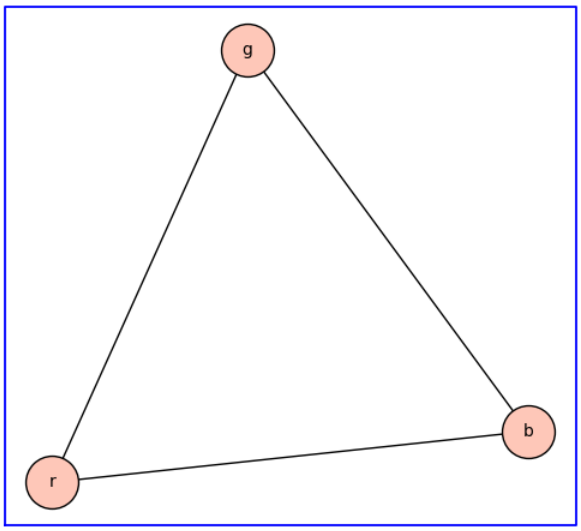
Login

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



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

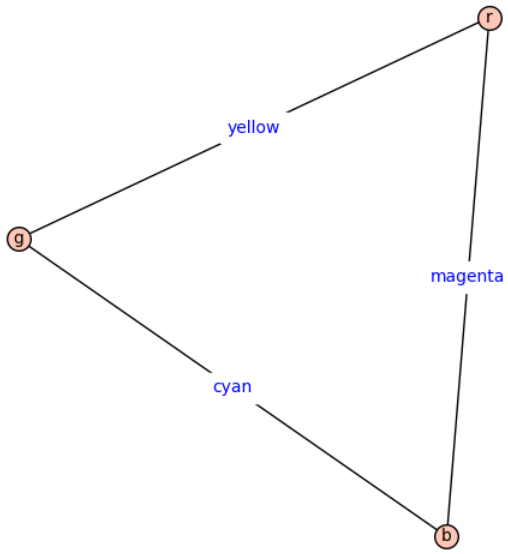

```
Graph(E).plot(vertex_size=1000, graph_border=True).show()
```



Edge Labels

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


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

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

G.show()
```



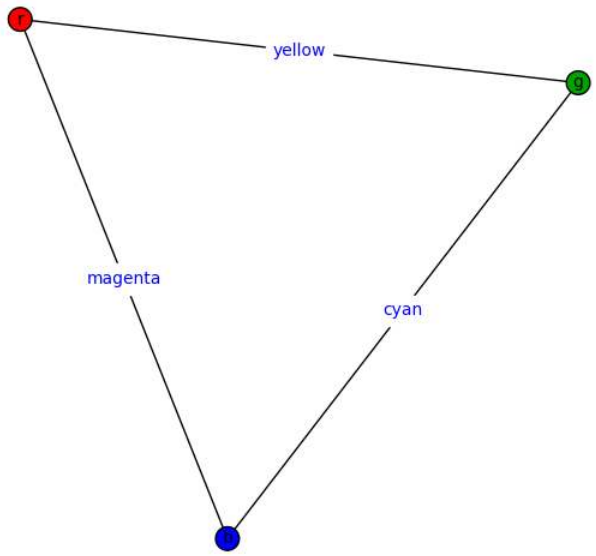
Color

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

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

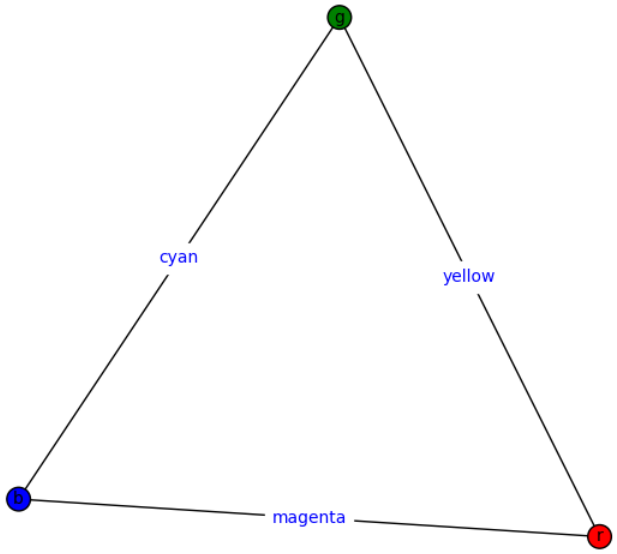

```
set_vertex_colors = {
    (1,0,0): ['r'], # Color vertex `r` all red
    (0,.65,0): ['g'], # Color vertex `g` dark green
    (0,0,1): ['b'] # Color vertex `b` all blue
}

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

```
)

G.show()
```



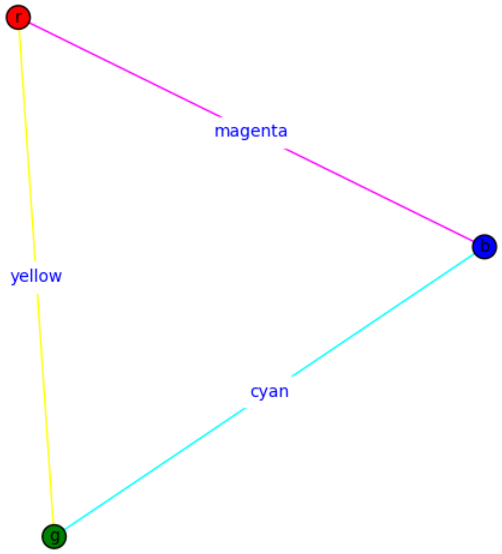
The following example specifies the color by name instead of RGB value. Sage will return an error if you use an undefined color name.


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

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

G.show()
```



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


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

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

G.show()
```



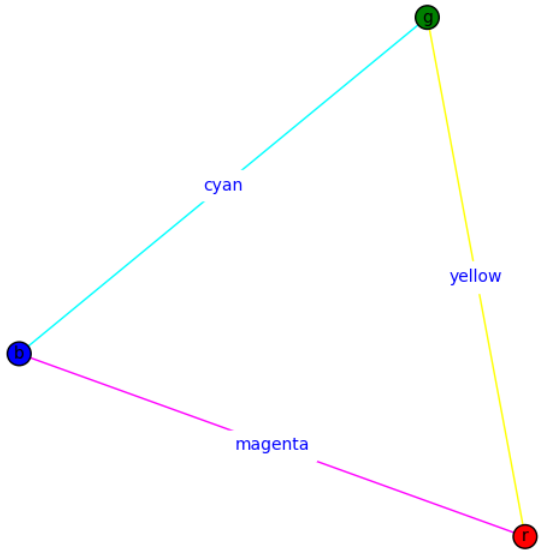
This alternate method specifies the color by name instead:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
set_edge_colors = {
    'yellow': [('r', 'g')],
    'cyan': [('g', 'b')],
    'magenta': [('b', 'r')]
}

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

G.show()
```



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

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

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


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

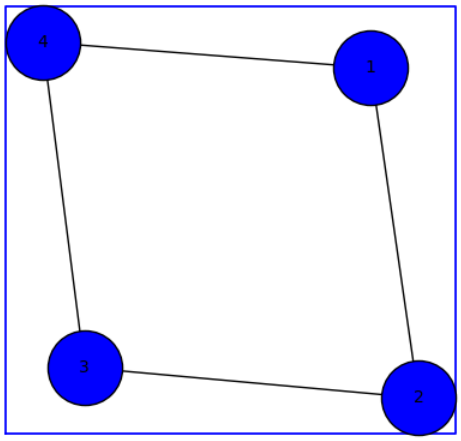
```
    show(
        g.plot(
            vertex_size=vertex_size,
            graph_border=graph_border,
            vertex_colors=color_str
        )
    )
```

Graph

Vertex Size

Border

Click ->



Layout

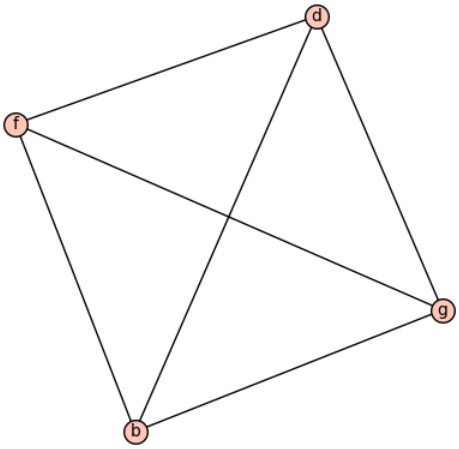
Let’s define and examine the following graph. Evaluate this cell multiple times and notice the vertex positions are not consistent.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

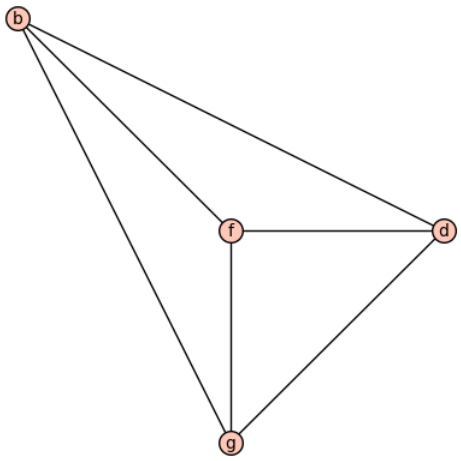
```
N = [
    ('g', 'b',),
    ('g', 'd',),
    ('g', 'f',),
    ('b', 'd',),
    ('b', 'f',),
    ('d', 'f',)
]

G = Graph(N)

G.show()
```



Layout options include: `"acyclic"`, `"circular"`, `"ranked"`, `"graphviz"`, `"planar"`, `"spring"`, or `"tree"`.

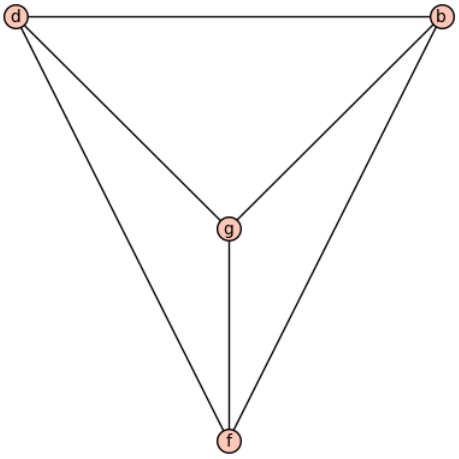
A planar graph can be drawn without any crossing edges. The default graph layout does not ensure the planar layout of a planar graph. Sage will return an error if you try to plot a non-planar graph with the planar layout.


```
G.plot(layout='planar').show()
```



Sage's `planar` algorithm sets the vertex positions. Alternatively, we can specify the positions in a dictionary. Let's position the G node in the center.


```
positions = {
    'g': (0, 0),
    'd': (-1, 1),
    'b': (1, 1),
    'f': (0, -1)
}

G.plot(pos=positions).show()
```



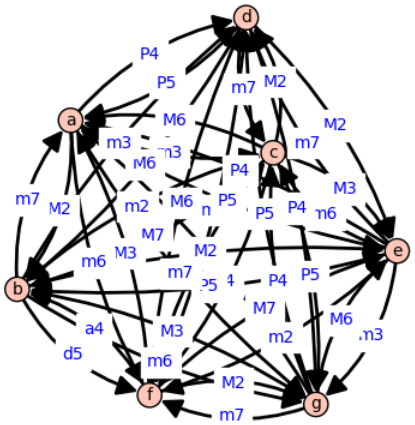
The following graph modeling the intervals in the C major scale is challenging to read. Let's think about how we can improve the presentation.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

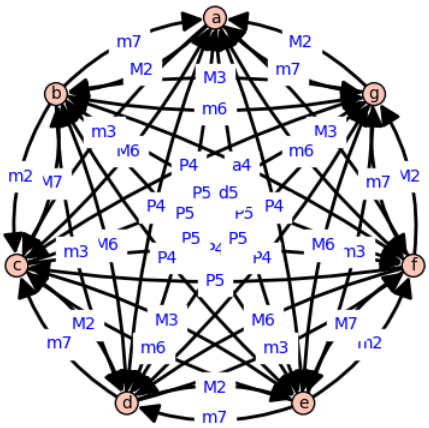
Login

```
I = [
    ("c", "d", "M2"), ("c", "e", "M3"), ("c", "f", "P4"), ("c", "g", "P5"), ("c", "a"
    ("d", "e", "M2"), ("d", "f", "m3"), ("d", "g", "P4"), ("d", "a", "P5"), ("d", "b"
    ("e", "f", "m2"), ("e", "g", "m3"), ("e", "a", "P4"), ("e", "b", "P5"), ("e", "c"
    ("f", "g", "M2"), ("f", "a", "M3"), ("f", "b", "a4"), ("f", "c", "P5"), ("f", "d"
    ("g", "a", "M2"), ("g", "b", "M3"), ("g", "c", "P4"), ("g", "d", "P5"), ("g", "e"
    ("a", "b", "M2"), ("a", "c", "m3"), ("a", "d", "P4"), ("a", "e", "P5"), ("a", "f"
    ("b", "c", "m2"), ("b", "d", "m3"), ("b", "e", "P4"), ("b", "f", "d5"), ("b", "g"
]

C = DiGraph(I, multiedges=True,)

C.plot(edge_labels=True).show()
```



In this case, the graph is not planar. The `circular` layout organizes the vertices for improved readability.

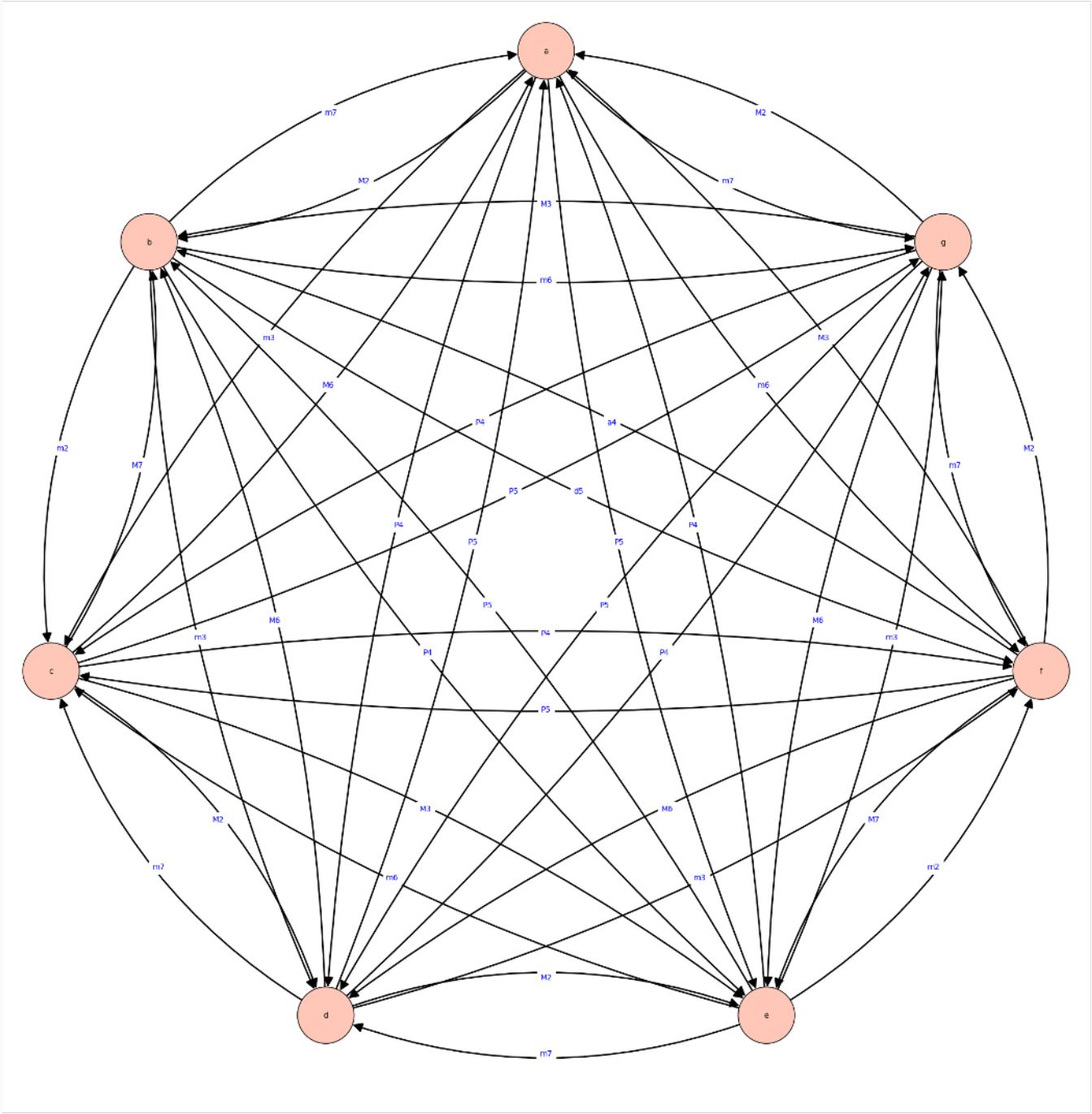

```
C.plot(edge_labels=True, layout='circular')
```



View in a New Tab

Increasing the `figsize` improves the definition of the arrows. For an even better view of the Graph, right-click the image and view it in a new tab.

```
C.plot(
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```



Edge Style

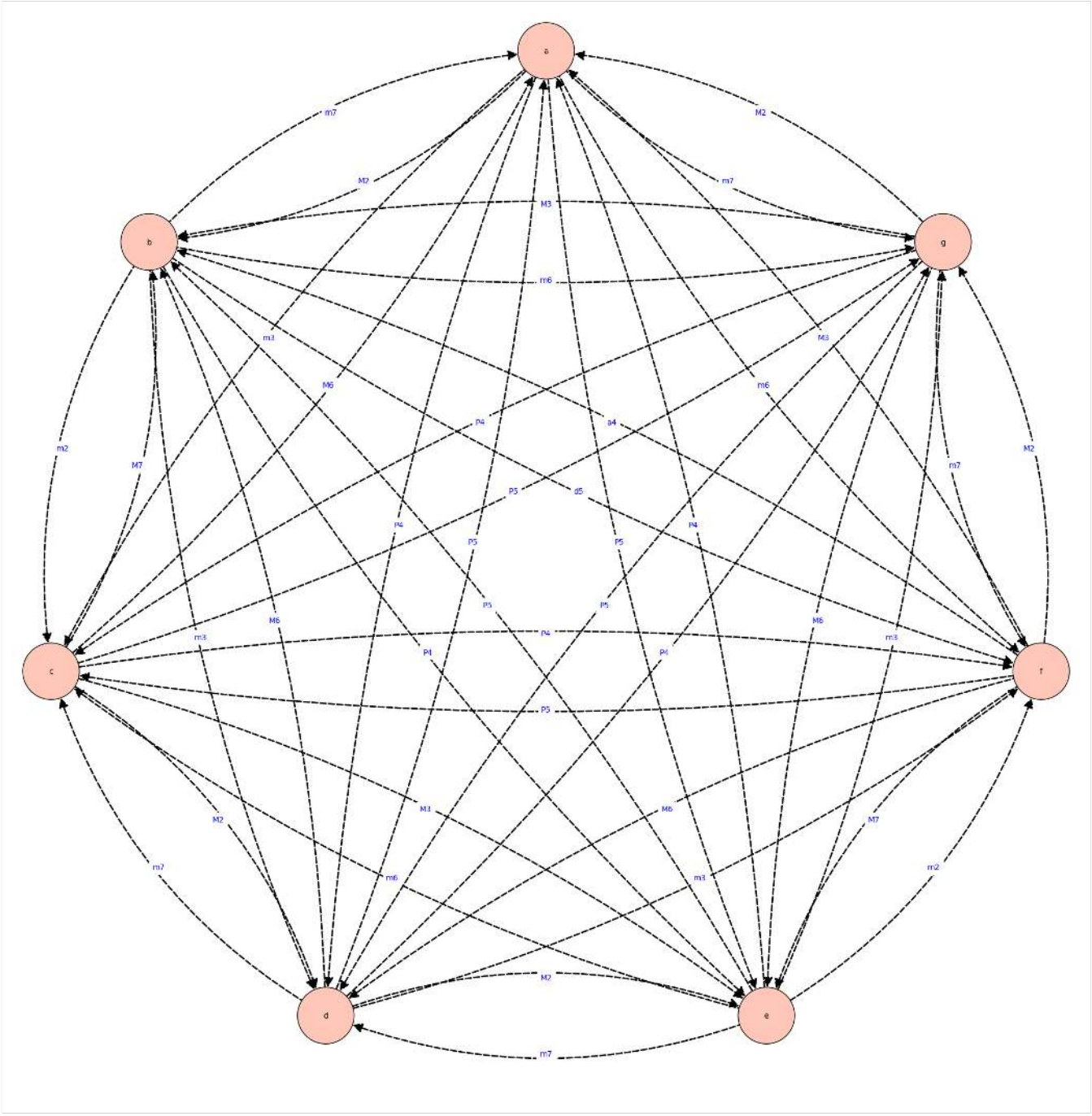
The options for `edge_style` include `“solid”` , `“dashed”` , `“dotted”` , or `"dashdot”` .

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
C.plot(
    edge_style='dashed',
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```



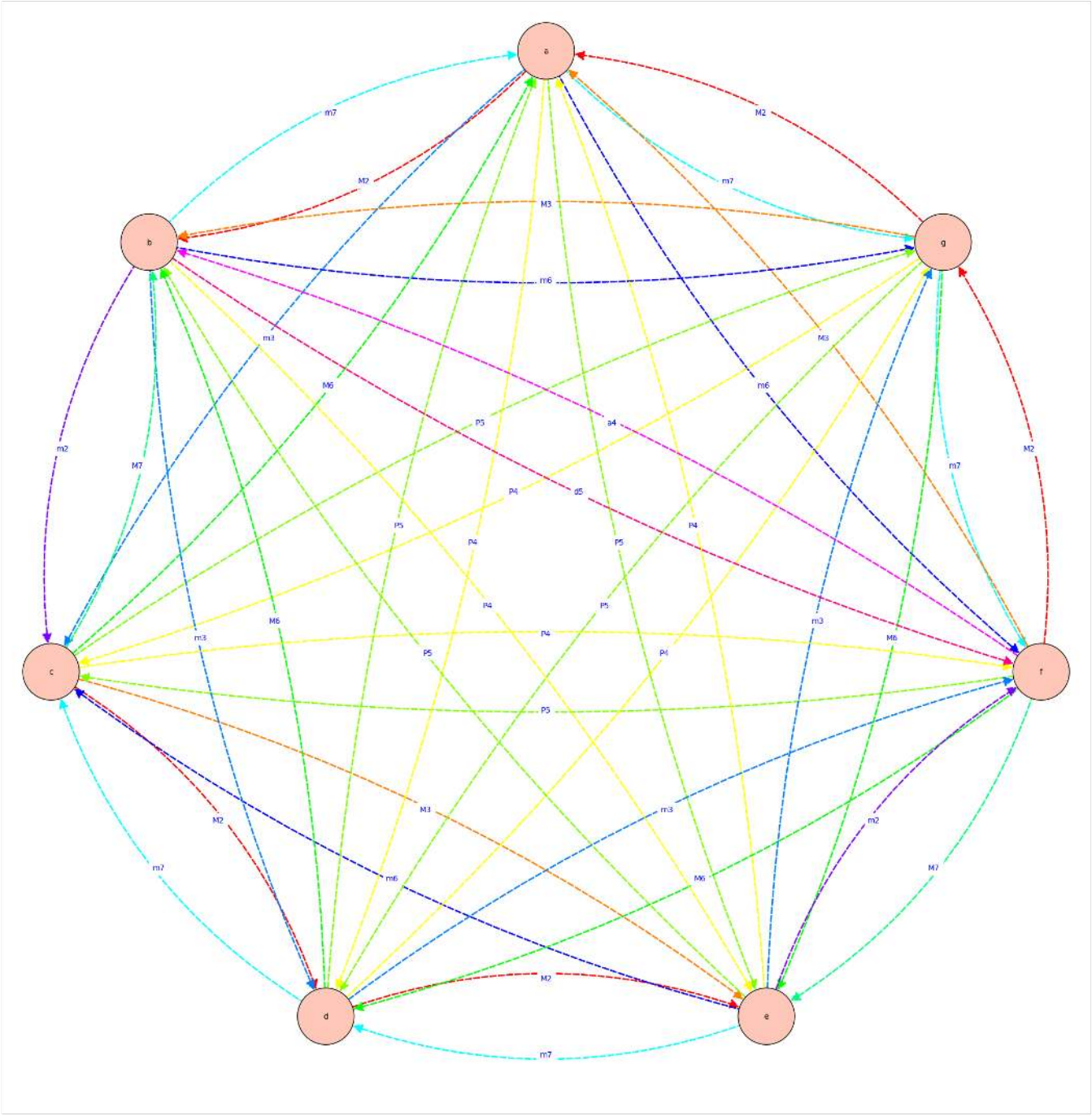
Improve the definition between the edges by using a different color for each edge. The `color_by_label` method automatically maps the colors to edges.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
C.plot(
    edge_style='dashed',
    color_by_label=True,
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```



3-Dimensional

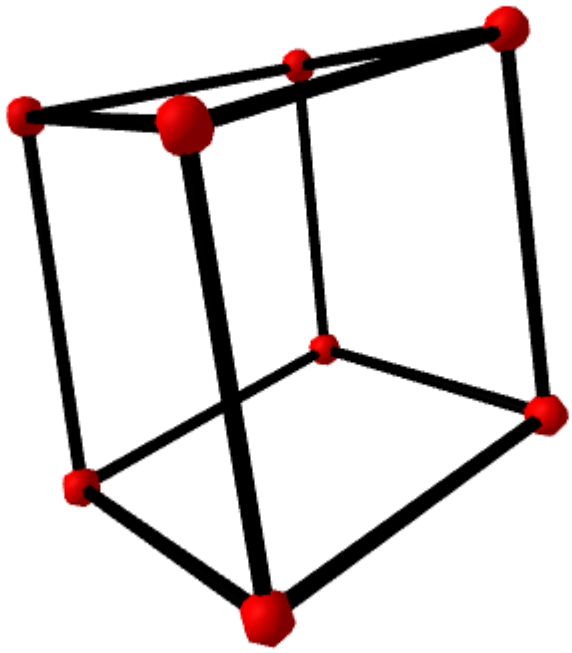
View a 3D representation of graph with `show3d()` . Click and drag the image to change the perspective. Zoom in on the image by pinching your computer's touchpad.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = graphs.CubeGraph(3)
G.show3d()
```

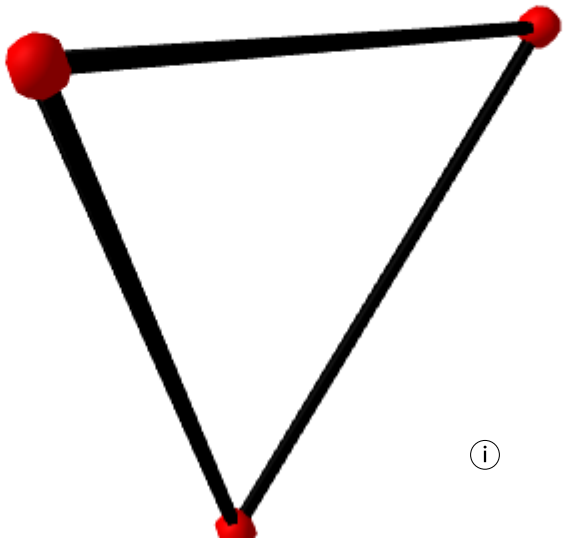


Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = graphs.TetrahedralGraph()
G.show3d()
```

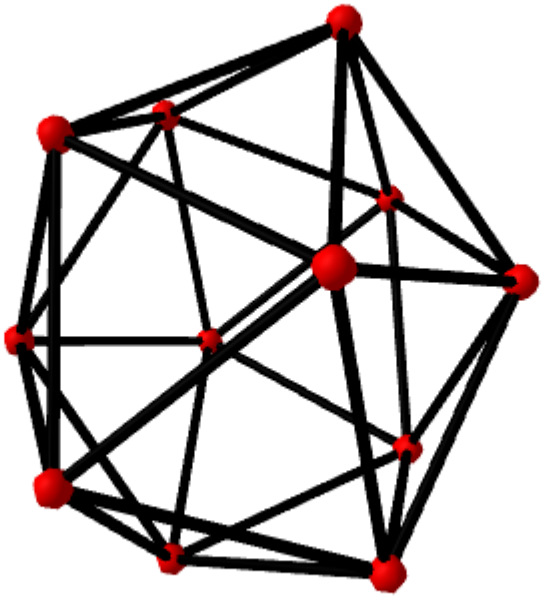


Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = graphs.IcosahedralGraph()
G.show3d()
```

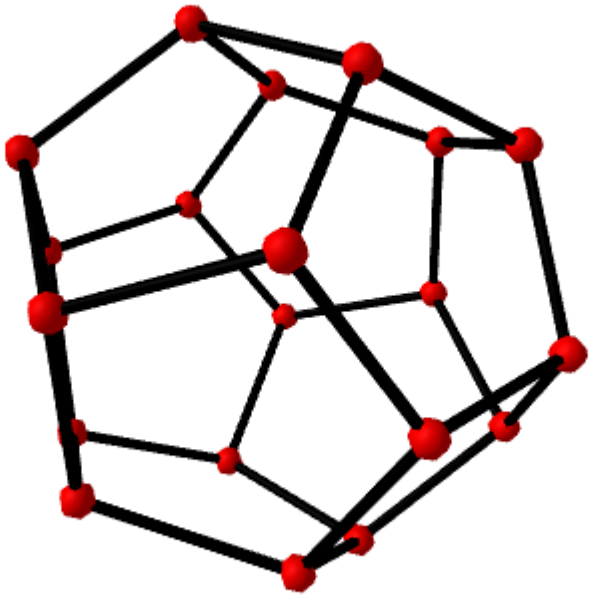


Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = graphs.DodecahedralGraph()
G.show3d()
```

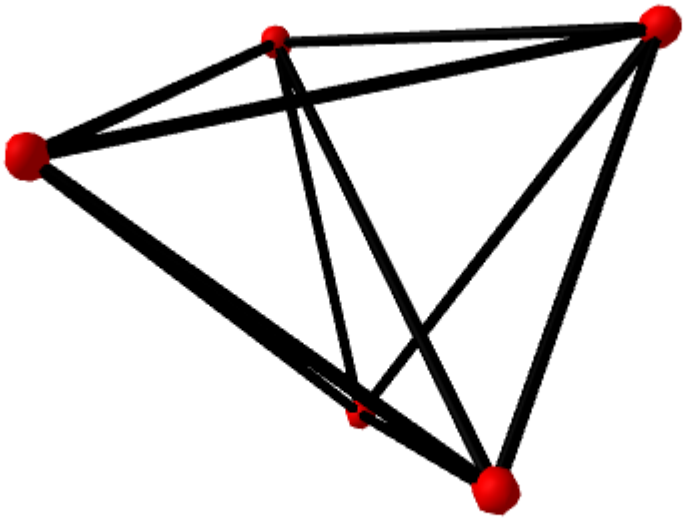


Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = graphs.CompleteGraph(5)
G.show3d()
```

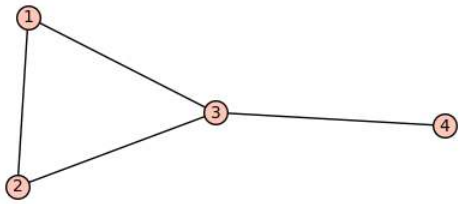


7.3: Paths

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:

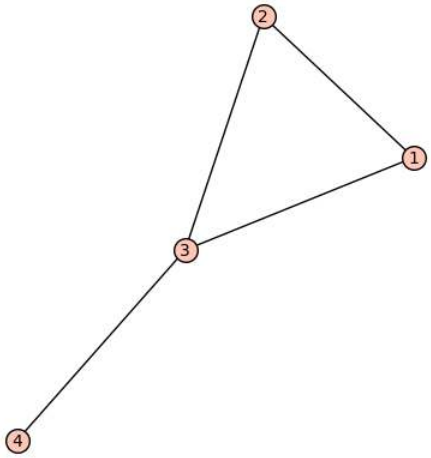

```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.show()
G.all_paths(1, 4)
```



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.show()
G.shortest_path(1, 4)
```



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

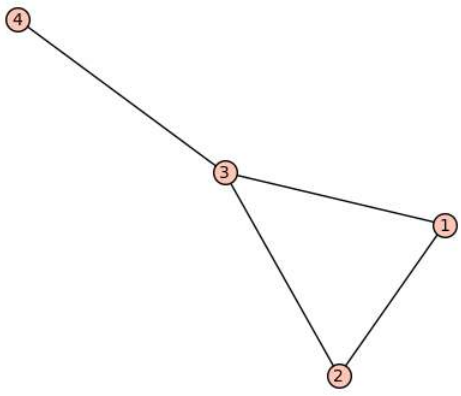
To determine if a graph is connected, we can use the `is_connected()` function:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

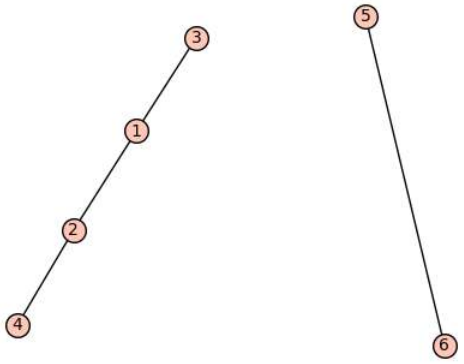
```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.show()
G.is_connected()
```



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.

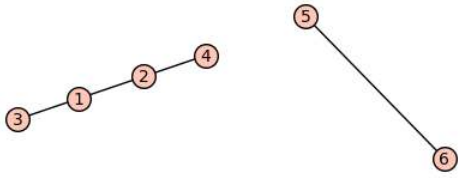
For example, the following graph is not connected:

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



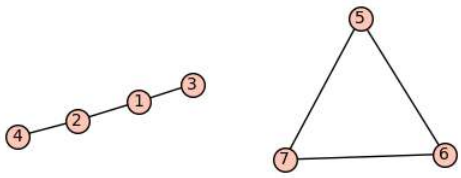
To identify all connected components of a graph, the `connected_components()` function can be utilized:


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



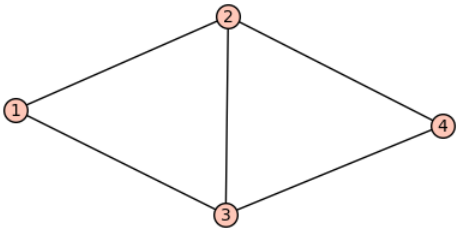
We can visualize the graph as a disjoint union of its connected components, by plotting it.


```
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.show()

# Calculates the diameter of the graph.
G.diameter()
```



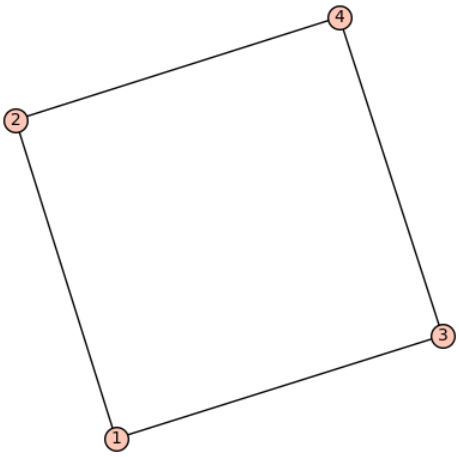
A graph is bipartite if its set of vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = Graph({1: [2, 3], 2: [4], 3: [4]})
G.show()
G.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 $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$.

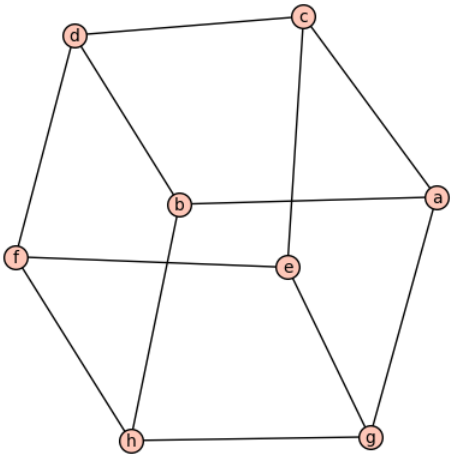
This means there is a bijection between the set of vertices such that every time two vertices determine an edge in the first graph, the image of these vertices by the bijection also determines an edge in the second graph, and vice versa. Essentially, the two graphs have the same structure, but the vertices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

Note

Graph isomorphism identifies structures relevant to chemistry, biology, machine learning, and neural networks.


```
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)
```

```
True
```

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:

$$G=(V,E) \text{ is connected if and only if} G'=(V',E') \$\text{is connected}$$

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

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

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's 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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 invariant_under_isomorphism(C, D)

NameError: name 'C' is not defined
```

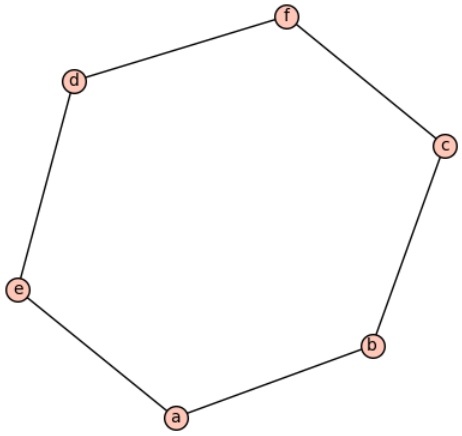
Let's construct a different pair of graphs A and B defined as follow

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```

```
Are both graphs connected? No
Do both graphs have same number of vertices and edges? Yes
Do both graphs have the same degree sequence? Yes

Therefore, the graphs are not isomorphic.
```

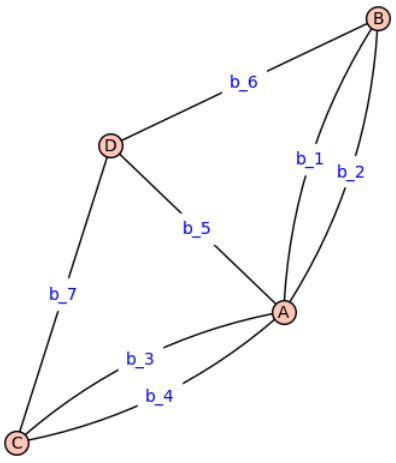
7.5: Euler and Hamilton

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)
```



Note

Eulerian circuits and paths have practical applications for reducing travel and costs in logistics, waste management, the airline industry, and postal service.

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G.is_eulerian()
```

```
False
```

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.

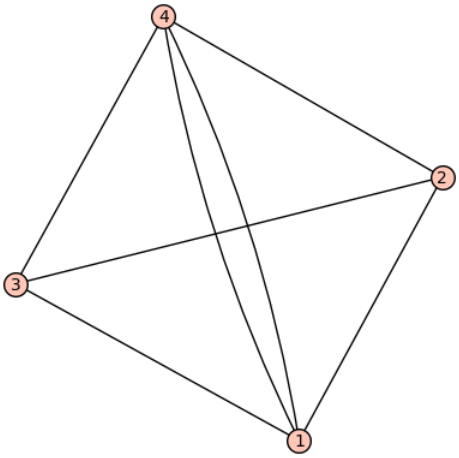
We can use `path=True` to determine if a graph contains an Euler path. Sage will return the beginning and the end of the path.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

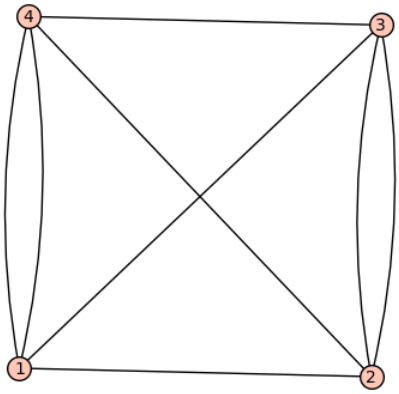
Login

```
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=
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)
```

```
([(0, 5), (5, 4), (4, 3), (3, 2), (2, 1), (1, 0)], [0, 5, 4, 3, 2, 1, 0])
```

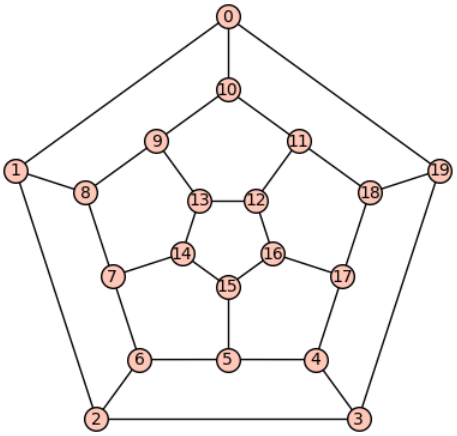
Hamilton

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.


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



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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

```
True
```

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.

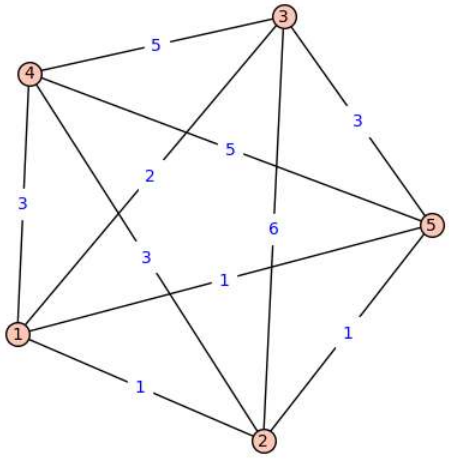
Let's explore the following graph:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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)
```



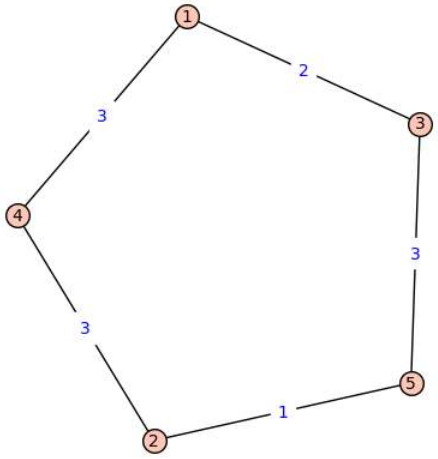
We can ask Sage if the graph contains a Hamiltonian cycle.


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

```
(True, [1, 2, 3, 4, 5])
```

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.

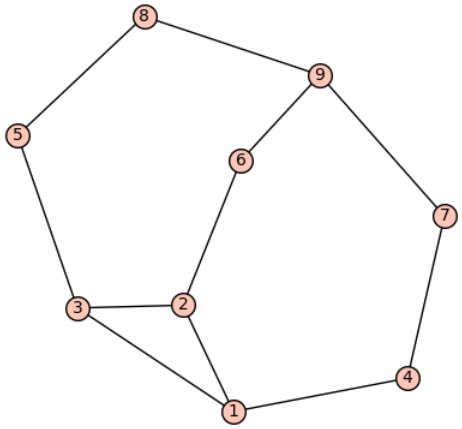

```
sumWeights = sum(h.edge_labels())
print(sumWeights)
```

```
12
```

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.

Login

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



7.6: Graphs in Action

Imagine you are a bike courier tasked with making deliveries to each City Colleges of Chicago (CCC) campus location. Per your contract, you get paid per delivery, not per hour. Therefore, finding the most efficient delivery route is in your best interest. We assume the bike delivery routes are the same distance in each direction.

Bike Courier Delivery Route Problem

Let's make a plan to solve our delivery route problem.

1. Find the distances in miles between each CCC location.
2. Make a graph of the CCC locations. Each location is a node. Each edge is a bike route. The weight of the edges represents the distance of the bike route between locations.
3. Use the traveling salesperson algorithm to calculate the optimal delivery route.

Locations

Table 7.6.1. CCC Addresses

Name	Address
Harold Washington College	30 E. Lake Street, Chicago, IL 60601
Harry Truman College	1145 West Wilson Ave, Chicago, IL 60640
Kennedy-King College	6301 South Halsted St, Chicago, IL 60621
Malcolm X College	1900 W. Jackson, Chicago, IL 60612
Olive-Harvey College	10001 South Woodlawn Ave, Chicago, IL 60628
Richard J. Daley College	7500 South Pulaski Rd, Chicago, IL 60652
Wilbur Wright College	4300 N. Narragansett Ave, Chicago, IL 60634

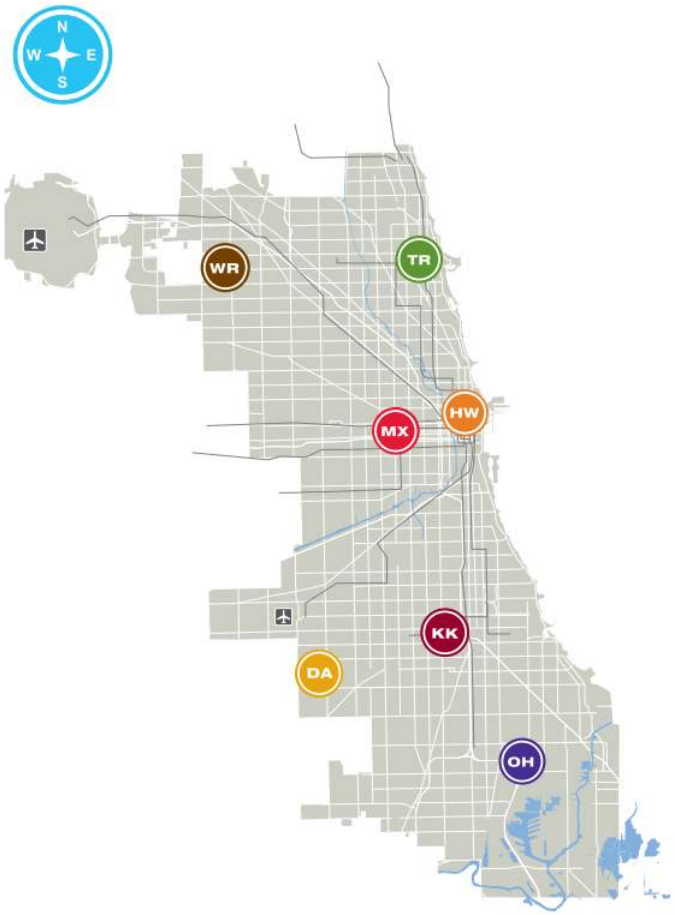

N
W
E
S
WR
TR
HW
MX
KK
DA
OH

Figure **7.6.2.** City Colleges of Chicago

Graph

We will represent each College as a node with the initials of the College name. The weight of the edge will represent the miles in between the locations. Since we are using bike routes, we are assuming each direction between two locations has the same distance. For example, express the route between Harold Washington College and Harry Truman College as `("HW", "HT", 6.5)`.


```
routes = [
    ("HW", "HT", 6.5),
    ("HW", "KK", 8.3),
    ("HW", "MX", 3.2),
    ("HW", "OH", 15.4),
    ("HW", "RD", 11.9),
    ("HW", "WW", 10.7),

    ("HT", "KK", 13.6),
    ("HT", "MX", 7.1),
    ("HT", "OH", 22.1),
    ("HT", "RD", 17.3),
    ("HT", "WW", 7.9),

    ("KK", "MX", 8.3),
    ("KK", "OH", 8.1),
    ("KK", "RD", 5.7),
    ("KK", "WW", 16.9),

    ("MX", "OH", 16.2),
    ("MX", "RD", 10.2),
    ("MX", "WW", 10.2),

    ("OH", "RD", 10.0),
    ("OH", "WW", 24.9),
```

```
    ("RD", "WW", 18.3)
]
routes
```

```
[('HW', 'HT', 6.50000000000000),
 ('HW', 'KK', 8.30000000000000),
 ('HW', 'MX', 3.20000000000000),
 ('HW', 'OH', 15.4000000000000),
 ('HW', 'RD', 11.9000000000000),
 ('HW', 'WW', 10.7000000000000),
 ('HT', 'KK', 13.6000000000000),
 ('HT', 'MX', 7.10000000000000),
 ('HT', 'OH', 22.1000000000000),
 ('HT', 'RD', 17.3000000000000),
 ('HT', 'WW', 7.90000000000000),
 ('KK', 'MX', 8.30000000000000),
 ('KK', 'OH', 8.10000000000000),
 ('KK', 'RD', 5.70000000000000),
 ('KK', 'WW', 16.9000000000000),
 ('MX', 'OH', 16.2000000000000),
 ('MX', 'RD', 10.2000000000000),
 ('MX', 'WW', 10.2000000000000),
 ('OH', 'RD', 10.0000000000000),
 ('OH', 'WW', 24.9000000000000),
 ('RD', 'WW', 18.3000000000000)]
```

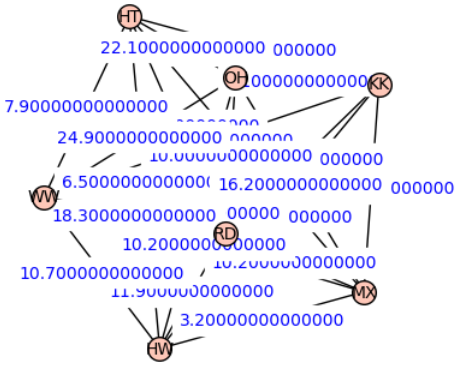
Create a `Graph` from the edge `list` :

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = Graph(routes)
G.show(edge_labels=True)
```



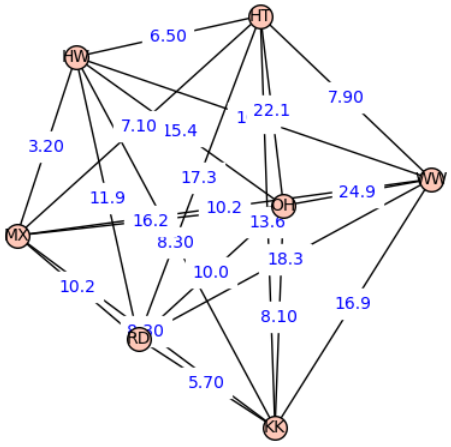
The trailing zeros of the floating point values are hard to read. Let's loop through the edge `list` and display the numbers with 3 points of precision.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

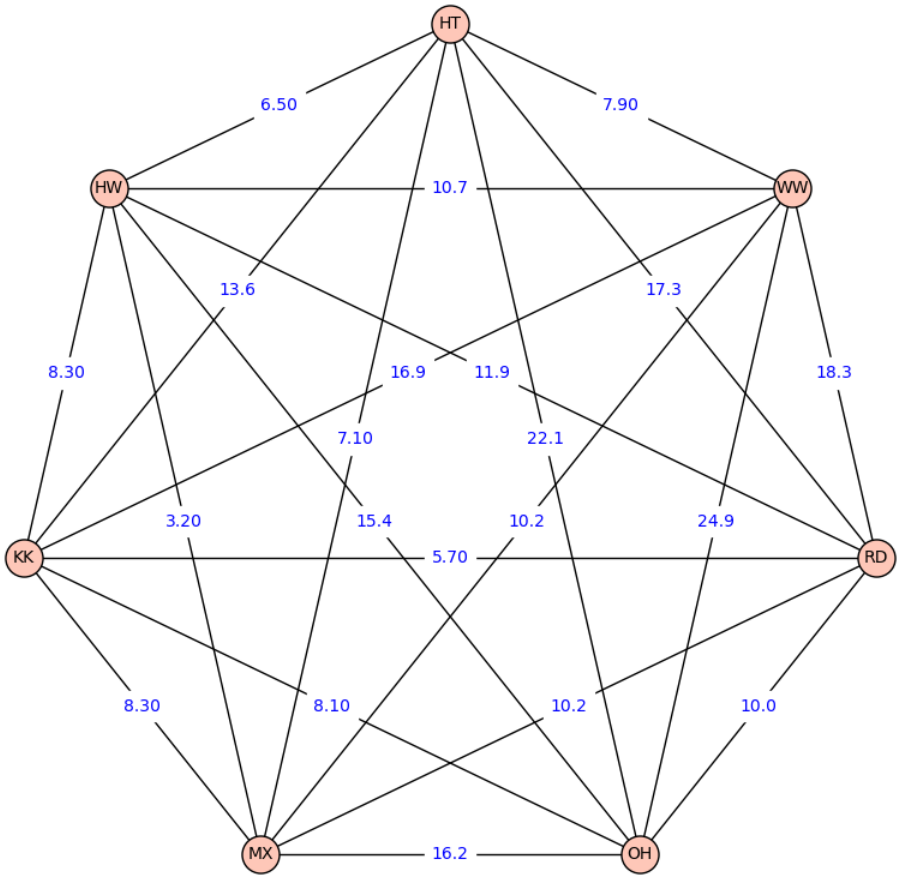
Login

```
for u, v, label in G.edge_iterator():
    G.set_edge_label(u, v, n(label, digits=3))

G.show(edge_labels=True)
```



Since this graph is not planar, improve the layout with the `"circular"` parameter. We can also improve the readability by increasing the `vertex_size` and `figsize` .

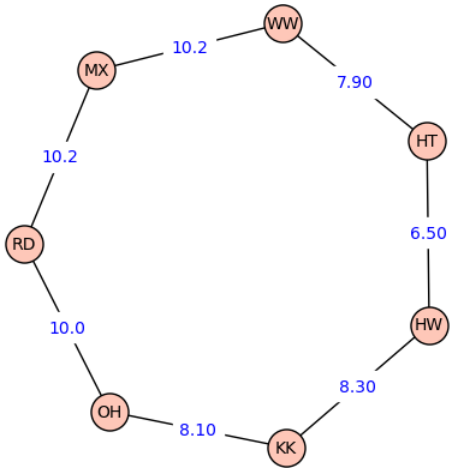
```
G.show(
    edge_labels=True,
    layout="circular",
    vertex_size=500,
    figsize=10,
)
```



Now that we have a clearer idea of the routes, let's find the most efficient delivery route using the traveling salesperson algorithm.

Login

```
optimal_route = G.traveling_salesman_problem(use_edge_labels=True, maximize=False)
optimal_route.show(
    edge_labels=True,
    vertex_size=500,
)
```



We can set the vertex positions to resemble their positions on the map. We can use the latitude and longitude values of the locations and then reverse them when we supply the values to the position `dictionary` .

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
positions = {
    'HW': (-87.62682604591349, 41.88609733324964),
    'HT': (-87.65901943241516, 41.9646769664519),
    'KK': (-87.6435785385309, 41.77847328856264),
    'MX': (-87.67453475017268, 41.87800548491064),
    'OH': (-87.5886722734757, 41.71006715754713),
    'RD': (-87.72315805813204, 41.75677704810169),
    'WW': (-87.78738482318016, 41.95836512405638),
}
```

CHAPTER OVERVIEW

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 $|E| = |v| - 1$.

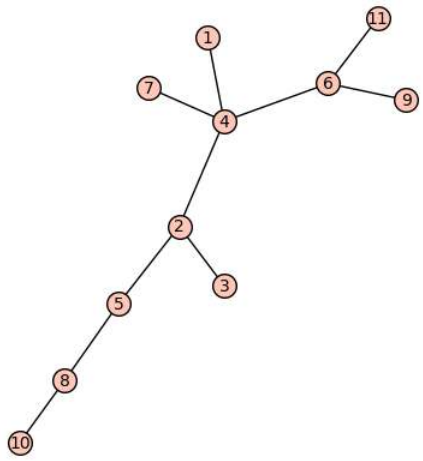
Let's explore the following graph:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
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()
```



Note

Trees are a common data structure used in file explorers, parsers, and decision making.

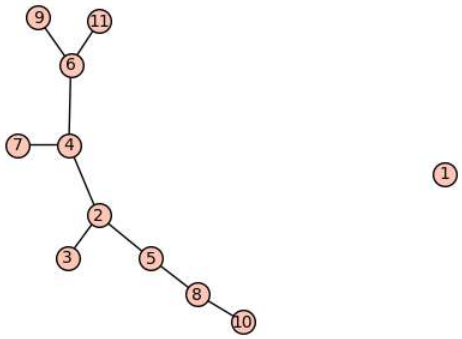
Let’s ask Sage if this graph is a tree.


```
G.is_tree()
```

```
True
```

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()
```



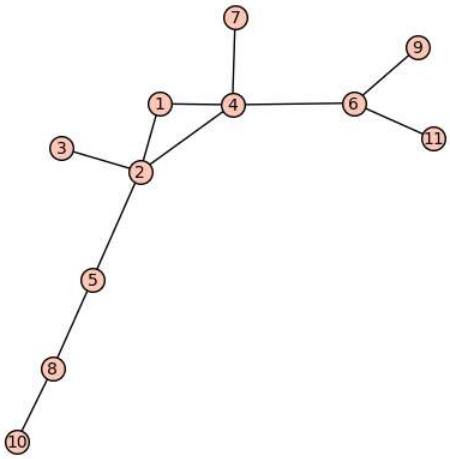
However, we can see that the graph is still a forest.


```
G_removed_edge.is_forest()
```

```
True
```

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, various algorithms can calculate a spanning tree, including depth-first search and breadth-first search.

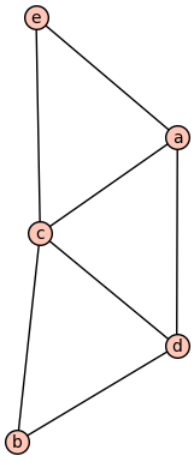
Breadth-first search algorithm

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 are visited.

The output of this algorithm is a spanning tree.

The `breadth_first_search()` function provides a flexible method for traversing both directed and undirected graphs. Let's consider the following graph:


```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4}, "c":{"e":2}, "d":{"a":5, "c":4}})
G.show()
print(list(G.breadth_first_search(start="a", report_distance=True)))
```



In the example above, the `start` parameter begins the traversal at vertex `a`. The `report_distance=True`, parameter reports pairs in the format `(vertex, distance)`. Distance is the length of the path from the start vertex. From the output

above, we see:

- The distance from vertex `a` to vertex `a` is `0`.
- The distance from vertex `a` to vertex `d` is `1`.
- The distance from vertex `a` to vertex `e` is `1`.
- The distance from vertex `a` to vertex `c` is `1`.
- The distance from vertex `a` to vertex `b` is `2`.

We can also set the parameter `edges=True` to return the edges of the BFS tree. Sage will raise an error if you use the `edges` and `report_distance` parameters simultaneously.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4}, "c":{"e":2}, "d":{"a":5, "c":4}})
s = list(G.breadth_first_search("a", edges=True))
print(s)
Graph(s)
```

```
[('a', 'd'), ('a', 'e'), ('a', 'c'), ('d', 'b')]
```

The above graph is a spanning tree, but not necessarily a minimum spanning tree. Let's check how many spanning trees exist.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G.spanning_trees_count()
```

```
21
```

Iterate over all the spanning trees of a graph with `spanning_trees()`.

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4}, "c":{"e":2}, "d":{"a":5, "c":4}})
spanning_trees = list(G.spanning_trees(labels=True))
for i, tree in enumerate(spanning_trees):
    print(f"Spanning Tree {i + 1}: {tree.edges()}")
    show(tree.plot())
```

```
Spanning Tree 1: [('a', 'c', 8), ('b', 'c', 6), ('c', 'd', 4), ('c', 'e', 2)]
```

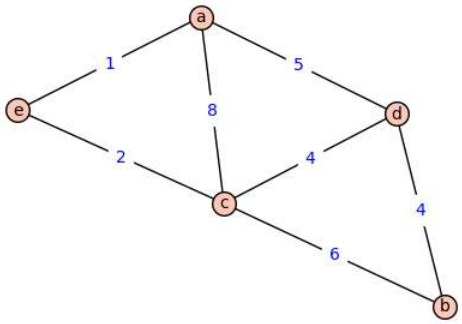
Given a weighted graph of all possible spanning trees we can calculate, we may be interested in the minimal one. A ***minimal spanning tree*** is a spanning tree whose sum of weights is minimal. Prim's Algorithm calculates a minimal spanning tree.

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. In R, select the cheapest vertex connected to the growing spanning tree L and add it to L
3. Follow the same procedure until all the vertices are in L

The output of this algorithm is a minimal spanning tree.


```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4}, "c":{"e":2}, "d":{"a":5, "c":4}})
G.show(edge_labels = True)
```



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.

Note

Minimal spanning trees influence the efficient design of networks and roads.

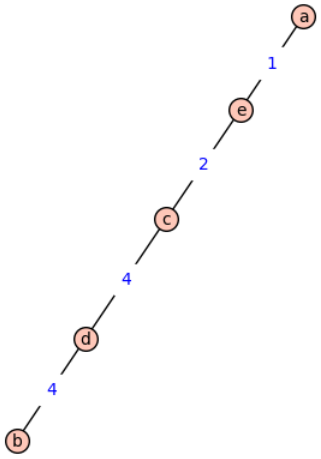

```
G.min_spanning_tree(by_weight=True)
```

```
[('b', 'd', 4), ('c', 'e', 2), ('a', 'e', 1), ('c', 'd', 4)]
```

From the output of `min_spanning_tree(by_weight=True)`, we see an edge `list` of the minimal spanning tree. Each element of the edge lis is a `tuple` where the first two values are vertices, and the third value is the edge weight or label.

Let's visualize the minimal spanning tree.

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



Let's define a function to view the minimal spanning tree in the context of the original graph. The function parameters include:

- `graph` : A SageMath Graph object.
- `mst_color` : Color for edges part of the MST (default: `'darkred'`).
- `non_mst_color` : Color for edges not part of the MST (default: `'lightblue'`).
- `figsize` : Dimensions for the graph image.


```
def visualize_mst(input_graph, mst_color='darkred', non_mst_color='lightblue', figsiz
    try:
        if not input_graph.is_connected():
            print("The graph must be connected")
            return

        mst_edges = input_graph.min_spanning_tree(by_weight=True)
        print("MST Edges:", mst_edges)
        Graph(mst_edges).show(edge_labels=True, figsize=figsize, edge_color=mst_color

        edge_colors = {mst_color: [], non_mst_color: []}
```

```
        mst_edge_set = set((v1, v2) for v1, v2, _ in mst_edges)

        for edge in input_graph.edges():
            v1, v2, _ = edge
            if (v1, v2) in mst_edge_set or (v2, v1) in mst_edge_set:
                edge_colors[mst_color].append((v1, v2))
            else:
                edge_colors[non_mst_color].append((v1, v2))

        print("MST overlaid on the original graph:")
        p = input_graph.plot(edge_labels=True, edge_colors=edge_colors, figsize=figsi
        show(p)

    except Exception as e:
        print("Error:", e)
```

```
```

Let's generate a random graph and view the minimal spanning tree.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
import random

vertices = 5
G = Graph([(i, j, random.randint(1, 20)) for i in range(vertices) for j in range(i+1,

visualize_mst(G)
```

```
MST Edges: [(0, 1, 3), (1, 2, 6), (1, 3, 4), (2, 4, 2)]
```

The following graph contains 9 vertices.

```
import random

vertices = 9
G = Graph([(i, j, random.randint(1, 20)) for i in range(vertices) for j in range(i+1,

visualize_mst(G, figsize=10)
```

```
MST Edges: [(1, 5, 4), (2, 7, 5), (0, 3, 4), (4, 5, 3), (0, 5, 4), (0, 6, 2), (1, 7,
```

The following graph contains 15 vertices.


```
import random

vertices = 15
G = Graph([(i, j, random.randint(1, 20)) for i in range(vertices) for j in range(i+1,

visualize_mst(G, figsize=10)
```

```
MST Edges: [(0, 1, 5), (1, 2, 1), (3, 6, 5), (4, 7, 3), (5, 11, 3), (2, 6, 3), (2, 7,
```

8.3: Trees in Action

Imagine your task is to create a railway between all the City Colleges of Chicago (CCC) campus locations. The contract requests that you use minimal track material to save construction costs. For simplicity's sake, assume each railway is a straight line between campuses.

Railway Problem

Let's make a plan to solve our railway construction optimization problem.

1. Find the latitude and longitude of each CCC campus location.
2. Use the Haversine formula to calculate the distances between the locations. The Haversine formula requires latitude and longitude for inputs and computes the shortest path between two points on a sphere.
3. Make a graph of the CCC campuses. Each location is a node. Each railway path is an edge. Each railway path is the shortest path between locations. The weight of the edges represents the distance between locations.
4. Find the minimum spanning tree (MST) of the CCC graph.

Location Distances

Table 8.3.1. City Colleges of Chicago Locations

Name	(Latitude, Longitude)
Harold Washington College	(41.88609733324964, -87.62682604591349)
Harry Truman College	(41.9646769664519, -87.65901943241516)
Kennedy-King College	(41.77847328856264, -87.6435785385309)
Malcolm X College	(41.87800548491064, -87.67453475017268)
Olive-Harvey College	(41.71006715754713, -87.5886722734757)
Richard J. Daley College	(41.75677704810169, -87.72315805813204)
Wilbur Wright College	(41.95836512405638, -87.78738482318016)

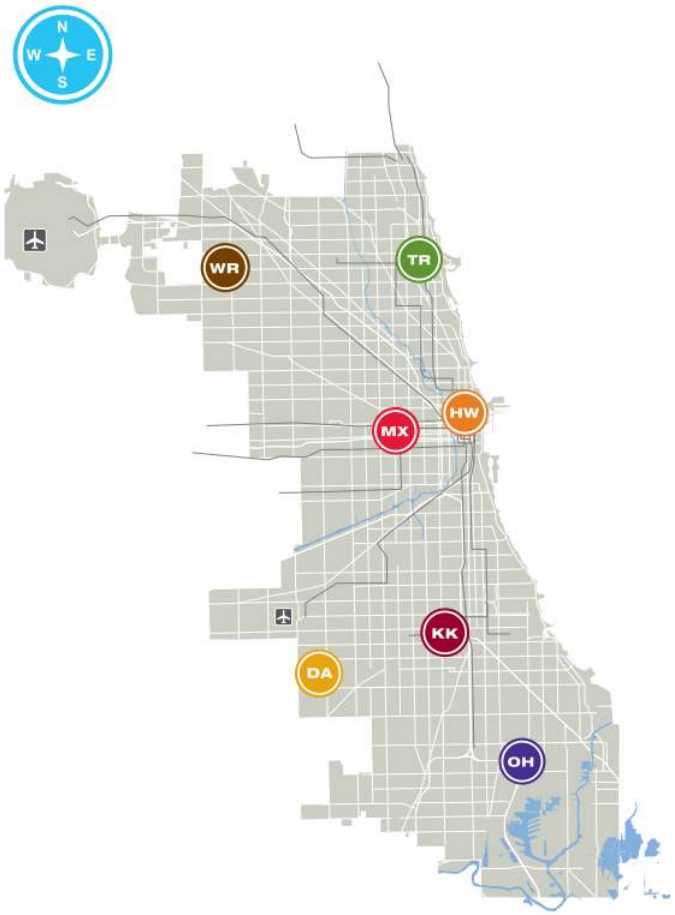

N
W
E
S
WR
TR
HW
MX
KK
DA
OH

Figure **8.3.2.** City Colleges of Chicago

Now, let's calculate the distances between campus locations. We will first create a `dictionary` to store the campus name, latitude, and longitude values.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
lat_long = {
    "HW": (41.88609733324964, -87.62682604591349),
    "HT": (41.9646769664519, -87.65901943241516),
    "KK": (41.77847328856264, -87.6435785385309),
    "MX": (41.87800548491064, -87.67453475017268),
    "OH": (41.71006715754713, -87.5886722734757),
    "RD": (41.75677704810169, -87.72315805813204),
    "WW": (41.95836512405638, -87.78738482318016)
}
lat_long
```

```
{'HW': (41.8860973332496, -87.6268260459135),
 'HT': (41.9646769664519, -87.6590194324152),
 'KK': (41.7784732885626, -87.6435785385309),
 'MX': (41.8780054849106, -87.6745347501727),
 'OH': (41.7100671575471, -87.5886722734757),
 'RD': (41.7567770481017, -87.7231580581320),
 'WW': (41.9583651240564, -87.7873848231802)}
```

Since the Earth is curved, we cannot use the Euclidean distance. We will use the Haversine formula instead. Note that the Haversine formula still produces an approximation because the Earth is not a perfect sphere. Here is a function to compute the Haversine formula.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
def haversine(lat1, lon1, lat2, lon2):
    '''Reference: https://cs.nyu.edu/~visual/home/proj/tiger/gisfaq.html'''
    import math

    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    dlat = lat2 - lat1
    dlon = lon2 - lon1

    a = math.sin(dlat / 2)**2 + \
        math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2

    c = 2 * math.asin(min(1.0, math.sqrt(a)))

    # Earth's approximate radius in kilometers
    R = 6367.0

    distance = R * c

    return distance

print("Ready to use `haversine()`")
```

```
Ready to use `haversine()`
```

Now we can make an edge `list` . We will represent each campus as a node with the initials of the college name. The weight of the edge will represent the Haversine value between the locations. For example, express the route between Harold Washington College and Harry Truman College as `("HW", "HT", Haversine)` .

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
distances = []
colleges = list(lat_long.items())
for i in range(len(colleges)):
    college1, (lat1, lon1) = colleges[i]
```

```
    for j in range(i + 1, len(colleges)):
        college2, (lat2, lon2) = colleges[j]
        dist = haversine(lat1, lon1, lat2, lon2)
        distances.append((college1, college2, dist))

print("\nDistances between colleges (in kilometers):")
for edge in distances:
    college1, college2, dist = edge
    print(f"{college1} - {college2}: {dist:.2f} km")
```

```
Distances between colleges (in kilometers):
HW - HT: 9.13 km
HW - KK: 12.04 km
HW - MX: 4.05 km
HW - OH: 19.82 km
HW - RD: 16.44 km
HW - WW: 15.52 km
HT - KK: 20.73 km
HT - MX: 9.72 km
HT - OH: 28.89 km
HT - RD: 23.70 km
HT - WW: 10.63 km
KK - MX: 11.35 km
KK - OH: 8.86 km
KK - RD: 7.02 km
KK - WW: 23.26 km
MX - OH: 19.97 km
MX - RD: 14.06 km
MX - WW: 12.92 km
OH - RD: 12.30 km
OH - WW: 32.13 km
RD - WW: 23.02 km
```

Graph

Swap (Latitude,Longitude) coordinates for plotting with (x,y) coordinates.

```
pos = {college: (lon, lat) for college, (lat, lon) in lat_long.items()}
pos
```

```
{'HW': (-87.6268260459135, 41.8860973332496),
 'HT': (-87.6590194324152, 41.9646769664519),
 'KK': (-87.6435785385309, 41.7784732885626),
 'MX': (-87.6745347501727, 41.8780054849106),
 'OH': (-87.5886722734757, 41.7100671575471),
 'RD': (-87.7231580581320, 41.7567770481017),
 'WW': (-87.7873848231802, 41.9583651240564)}
```

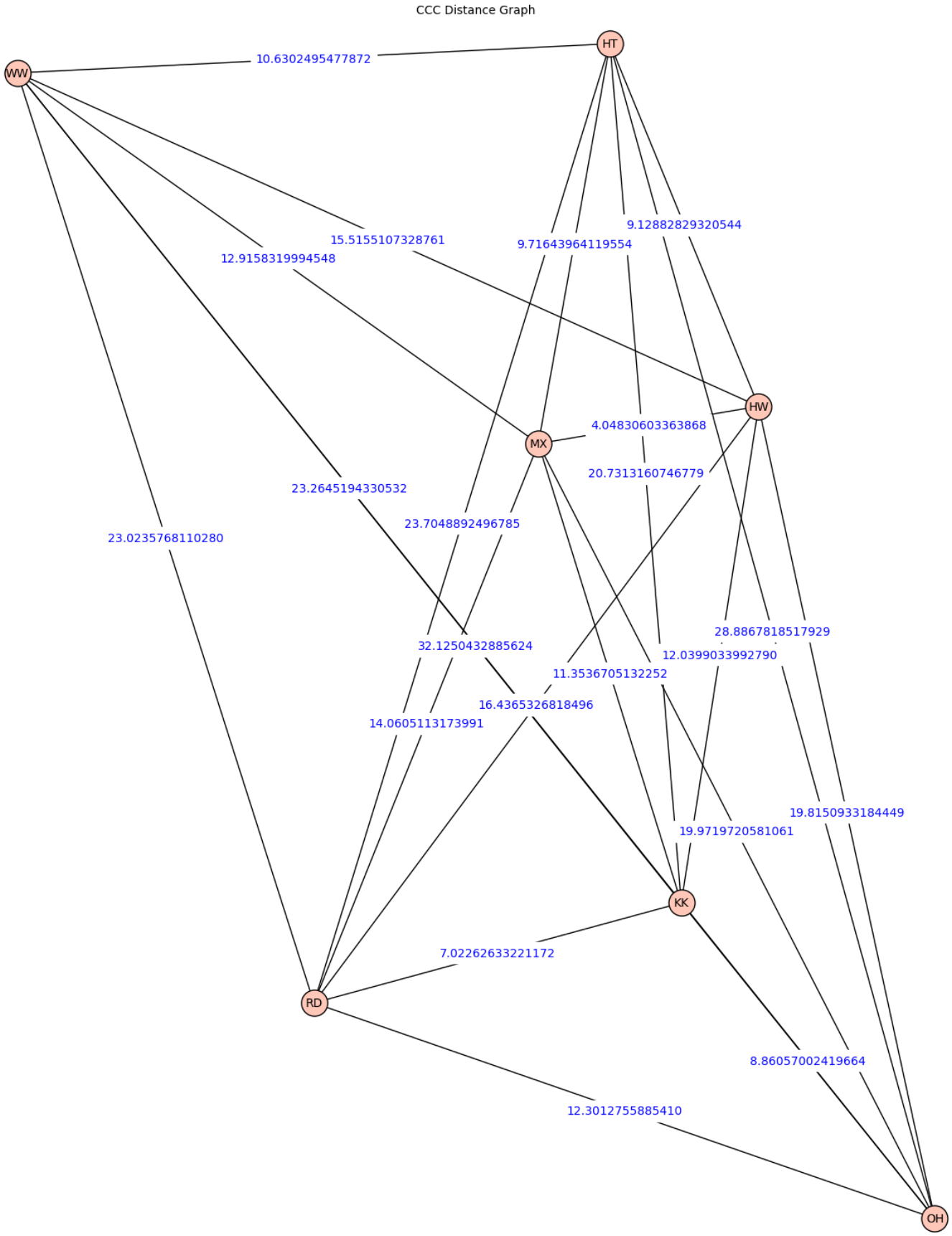
Create a `Graph` from the edge `list` :

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
G = Graph(distances)
G.show(
    pos=pos,  # Positions are (longitude, latitude)
    edge_labels=True,
    vertex_size=500,
    figsize=20,
    title="CCC Distance Graph"
)
```


CCC Distance Graph
HT
WW
10.6302495477872
9.12882829320544
15.5155107328761
9.71643964119554
12.9158319994548
HW
4.04830603363868
MX
20.7313160746779
23.2645194330532
23.7048892496785
23.0235768110280
28.8867818517929
32.1250432885624
12.0399033992790
11.3536705132252
16.4365326818496
14.0605113173991
19.8150933184449
19.9719720581061
KK
7.02262633221172
RD
8.86057002419664
12.3012755885410
OH

Railway

So far, we have encountered various concepts for connecting a graph's vertices, including the Hamilton path and the MST. Let's consider what technique is best suited for solving the problem of constructing a railway that optimizes material costs.

The previous chapter used the traveling salesperson algorithm to optimize a delivery route. Since we aim to optimize material costs, you might think of following a similar approach: apply the traveling salesperson algorithm, eliminate the greatest edge from the Hamilton circuit, and design the railway with the minimum Hamilton path. If we take a Hamilton circuit and eliminate one edge, we obtain a spanning tree. While the Hamilton path optimizes graph traversal by visiting each vertex exactly once in a single path, it does not guarantee that all vertices are connected with the minimal total weight.

In a Hamilton path, the requirement to visit each vertex in a single path can force the inclusion of high-weight edges. Alternatively, the MST is not restricted by the requirement of connecting vertices with a path. The MST can avoid high-weight edges by connecting vertices without regard to forming a path as long as the graph remains connected and acyclic. Although the minimum Hamilton path is one of many possible spanning trees, it is not an MST. Prim's Algorithm ensures the weight of the spanning tree is minimal because, at each iteration, it selects the smallest-weight edge.

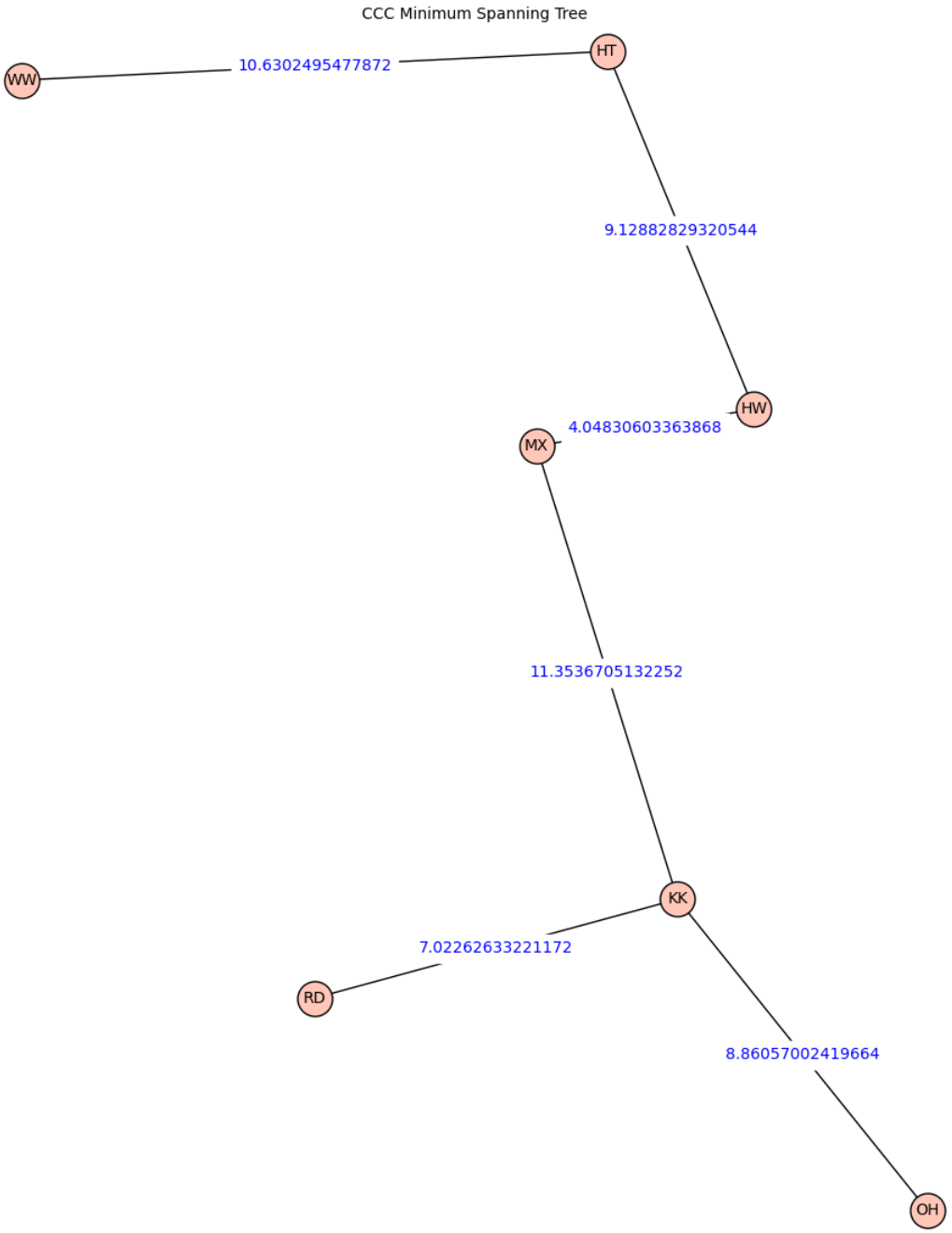
Let's find the MST edge `list` of the campus locations with the `min_spanning_tree(by_weight=True)` function.


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

```
[('HT', 'HW', 9.12882829320544),
 ('KK', 'MX', 11.3536705132252),
 ('HW', 'MX', 4.04830603363868),
 ('KK', 'OH', 8.86057002419664),
 ('KK', 'RD', 7.02262633221172),
 ('HT', 'WW', 10.6302495477872)]
```

Visualize the MST with the vertex positions mapped to the geographical coordinates of each campus location.

```
Graph(mst).show(
    pos=pos,
    edge_labels=True,
    vertex_size=500,
    figsize=15,
    title="CCC Minimum Spanning Tree"
)
```



Conclusion

In this exercise, we only optimized construction material costs. In a real-world scenario, we may want to create a railway that optimizes both travel time and material costs. In the case of the Chicago L train system, the railway resembles a tree when ignoring the downtown Loop. The L receives criticism for its lack of interconnectivity. For example, finding an efficient route connecting the end of the Blue Line with the end of the Red Line is challenging because a traveler may need to commute all the way downtown from one end of the railway to reach another end. As an interesting challenge, you can optimize both travel time and construction costs.

CHAPTER OVERVIEW

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

A ***lattice*** is 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.

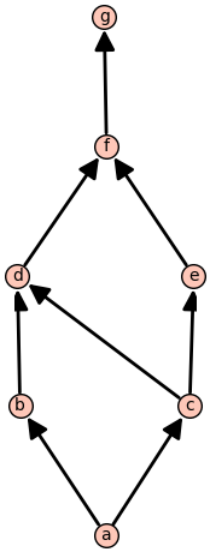
First, let's define the lists of elements and relations we will use for the following examples:


```
elements = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

relations = [
    ['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
    ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']
]
print("Elements: ", elements)
print("Relations: ", relations)
```

```
Elements:  ['a', 'b', 'c', 'd', 'e', 'f', 'g']
Relations:  [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'], ['c', 'e'], ['d', 'f'],
```

Create a poset from a tuple of elements and relations.

```
PO = Poset((elements, relations))
PO.show()
```



The function `is_lattice()` determines whether the poset is a lattice.


```
PO.is_lattice()
```

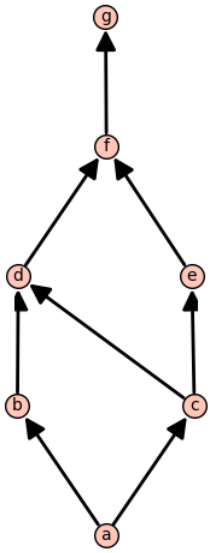
```
True
```

Note

Lattices have practical applications in computer science, such as static program analysis and distributed programming.

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 `LatticePoset()` function will raise an error if the poset is not a lattice.

```
LP = LatticePoset((elements, relations))
LP.show()
```



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.


```
PO.is_join_semilattice()
```

```
True
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
PO.join('b', 'f')
```

```
'f'
```

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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
PO.is_meet_semilattice()
```

```
True
```

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

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

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

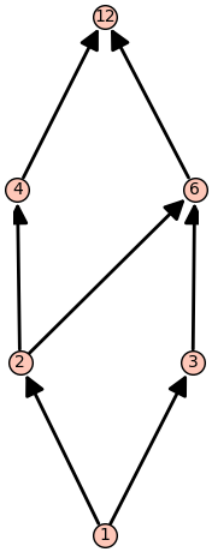
```
'a'
```

Divisor Lattice

The Sage `DivisorLattice()` function returns the divisor lattice of an integer.

The elements of the lattice are divisors of n and $x < y$ in the lattice if x divides y.


```
Posets.DivisorLattice(12).show()
```



9.2: Tables of Operations

This section examines the representation of meet (∧) and join (∨) operations within lattices using operation tables.

Meet Operation Table

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

To output 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.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
elements = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

relations = [
    ['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
    ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']
]

L = LatticePoset((elements, relations))
M = L.meet_matrix()
show(M)
```

$$/* <![CDATA[*/ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 2 & 2 & 2 & 2 \\ 0 & 1 & 2 & 3 & 2 & 3 & 3 \\ 0 & 0 & 2 & 2 & 4 & 4 & 4 \\ 0 & 1 & 2 & 3 & 4 & 5 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} /*]] > */ \tag{9.2.1}$$

From the output matrix, we can see that each entry is not the actual value of the meet of the elements a_i and a_j but just its position in the lattice. Let's show the values:

Show the output as a table:

```
linear_extension = L.linear_extension()

values_meet_matrix = [
    [
        linear_extension[M[i, j]]
        for j in range(len(elements))
    ]
    for i in range(len(elements))
]

values_meet_matrix
```

```
[['a', 'a', 'a', 'a', 'a', 'a', 'a'],
 ['a', 'b', 'a', 'b', 'a', 'b', 'b'],
 ['a', 'a', 'c', 'c', 'c', 'c', 'c'],
 ['a', 'b', 'c', 'd', 'c', 'd', 'd'],
 ['a', 'a', 'c', 'c', 'e', 'e', 'e'],
 ['a', 'b', 'c', 'd', 'e', 'f', 'f'],
 ['a', 'b', 'c', 'd', 'e', 'f', 'g']]
```

Join Operation Table

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


```
J = L.join_matrix()

show(J)
```

[Math Processing Error]
Output the elements of the poset:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
linear_extension = L.linear_extension()

values_join_matrix = [
    [
        linear_extension[J[i, j]]
        for j in range(len(elements))
    ]
    for i in range(len(elements))
]

values_join_matrix
```

```
[['a', 'b', 'c', 'd', 'e', 'f', 'g'],
 ['b', 'b', 'd', 'd', 'f', 'f', 'g'],
 ['c', 'd', 'c', 'd', 'e', 'f', 'g'],
 ['d', 'd', 'd', 'd', 'f', 'f', 'g'],
 ['e', 'f', 'e', 'f', 'e', 'f', 'g'],
 ['f', 'f', 'f', 'f', 'f', 'f', 'g'],
 ['g', 'g', 'g', 'g', 'g', 'g', 'g']]
```

Show the output as a table instead of a matrix.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
import pandas as pd

df = pd.DataFrame(
    values_join_matrix,
    index=elements,
    columns=elements
)

df
```

	a	b	c	d	e	f	g
a	a	b	c	d	e	f	g
b	b	b	d	d	f	f	g
c	c	d	c	d	e	f	g
d	d	d	d	d	f	f	g
e	e	f	e	f	e	f	g
f	f	f	f	f	f	f	g
g	g	g	g	g	g	g	g

CHAPTER OVERVIEW

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

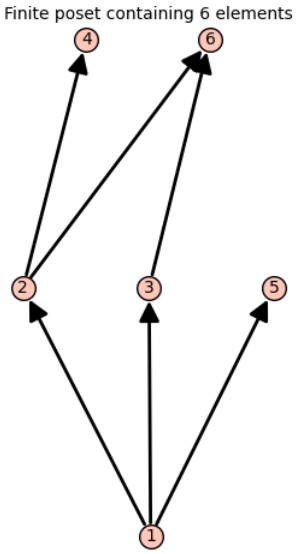
A Boolean algebra is a bounded lattice that is both complemented and distributive. Let's define the `is_boolean_algebra()` function to determine whether a given poset is a Boolean algebra. The function accepts a finite partially ordered set as input and returns a tuple containing a boolean value and a message explaining the result. Run the following cell to define the function and call it in other cells.


```
def is_boolean_algebra(P):
    try:
        L = LatticePoset(P)
    except ValueError as e:
        return False, str(e)
    if not L.is_bounded():
        return False, "The lattice is not bounded."
    if not L.is_distributive():
        return False, "The lattice is not distributive."
    if not L.is_complemented():
        return False, "The lattice is not complemented."
    return True, "The poset is a Boolean algebra."
```

```

```

Let's check if the following poset is a Boolean algebra.

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

P = Poset((S, attrcall("divides")))

show(P)
```

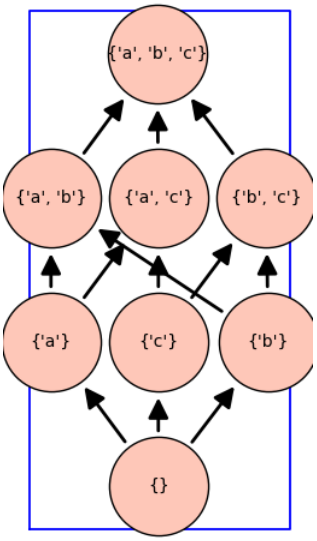



```
is_boolean_algebra(P)
```

```
(False, 'not a join-semilattice: no top element')
```

When we pass `P` to the `is_boolean_algebra()` function, `LatticePoset()` raises an error because `P` is not a lattice. The `ValueError` provides more information about the absence of a top element. Therefore, `P` is not a Boolean algebra.

```
T = Subsets(['a', 'b', 'c'])

Q = Poset((T, lambda x, y: x.issubset(y)))

Q.plot(vertex_size=3500, border=True)
```




```
is_boolean_algebra(Q)
```

```
(True, 'The poset is a Boolean algebra.')
```

Let’s examine the divisor lattice of 30:

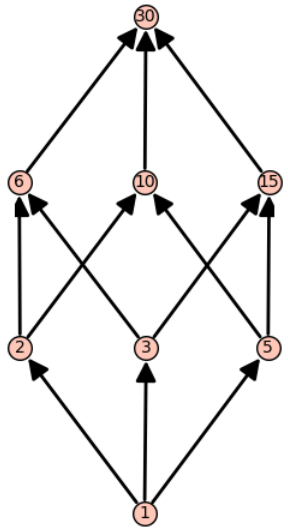
Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
dl30 = Posets.DivisorLattice(30)
show(dl30)
is_boolean_algebra(dl30)
```

Finite lattice containing 8 elements with distinguished linear extension



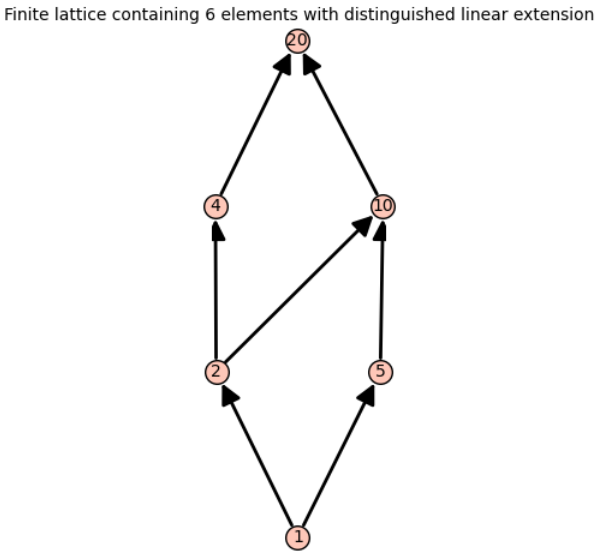
Now for the divisor lattice of 20:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
dl20 = Posets.DivisorLattice(20)
show(dl20)
is_boolean_algebra(dl20)
```



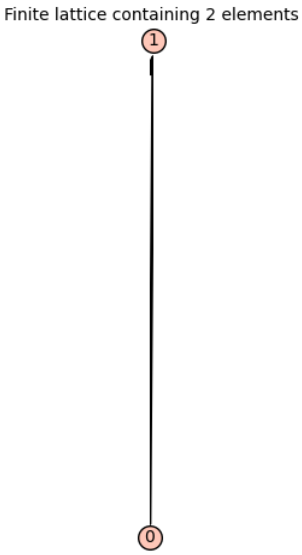
Here is a classic example in the field of computer science:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
B = posets.BooleanLattice(1)
show(B)
is_boolean_algebra(B)
```



10.2: Boolean functions

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

> **Note**
>
> Boolean algebra influences the design of digital circuits. For example, simplifying a digital circuit can minimize the number of gates used and reduce the manufacturing cost.

A ***minterm*** of the Boolean variables x1,x2,…,xn is the Boolean product y1·y2·…·yn where each yi=xi or .yi=xi—.

A sum of minterms is called a ***sum-of-products*** expansion. In this section, we will examine various methods for finding the sum-of-products expansion of a Boolean function.

To find the sum-of-products expansion using a truth table, we first convert the `truthtable()` into a form that is iterable with `get_table_list()`. For every row where the output value is `True`, we construct a minterm:

- Include the variable as is if its value is `True`
- Include the negation of the variable if its value is `False`
- The `zip` function pairs each variable with its corresponding value, allowing us to create minterms efficiently.
- We add each minterm to the `sop_expansion list` using the `&` operator.
- Finally, we join all minterms with the `|` operator to form the sum-of-products expansion.
- The function returns the sum-of-products expansion as a `sage.logic.boolformula.BooleanFormula` instance.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
def truth_table_sop(expression):
    # Check if the input is a string, and if so, convert it to a formula object
    if isinstance(expression, str):
        h = propcalc.formula(expression)
    elif isinstance(expression, sage.logic.boolformula.BooleanFormula):
        h = expression
    else:
        raise ValueError

    table_list = h.truthtable().get_table_list()
    sop_expansion = []

    for row in table_list[1:]:  # Skip the header row
        if row[-1]:  # If the output value is True
            minterm = []
```

```
            for var, value in zip(table_list[0], row[:-1]):  # Iterate over each vari
                if value:
                    minterm.append(var)  # Include variable as is if True
                else:
                    minterm.append(f'~{var}')  # Include the negated variable if Fals
            sop_expansion.append(' & '.join(minterm))  # Join variables in the minter

    sop_result = ' | '.join(f'({m})' for m in sop_expansion)  # Join minterms using t
    return propcalc.formula(sop_result)
```

```
```

For your convenience, our `truth_table_sop` function converts `String` input with `propcalc.formula`. Therefore, the input accepts `String` representations of Boolean expressions. Alternatively, you may pass an instance of `sage.logic.boolformula.BooleanFormula` directly to the function.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
truth_table_sop("x & (y | z)")
```

```
(x&~y&z)|(x&y&~z)|(x&y&z)
```

Let's verify that the sum-of-products expansion we found with the truth table is equivalent to the original expression.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
truth_table_sop("x & (y | z)") == propcalc.formula("x & (y | z)")
```

```
True
```

Our `sop_expansion` function mimics the manual process of finding the sum-of-products expansion of a Boolean function. This process does not guarantee the minimal form of the Boolean expression.

If we dig around in the Sage source code, we can find a commented-out `Simplify()` function that relied on the `Boolopt` package and the Quine-McCluskey algorithm. The Quine-McCluskey algorithm guarantees the minimal form of the Boolean expression, but the exponential complexity of the algorithm makes it impractical for large expressions. Moreover, in the Sage documentation, we see a placeholder function called `Simplify()` that returns a `NotImplementedError` message. The Sage community is waiting for someone to implement this function with the Espresso algorithm. While the Espresso algorithm does not guarantee the minimal form of the Boolean expression, it is more efficient than the Quine-McCluskey algorithm.

Sage integrates well with Python libraries like SymPy, which have built-in functions for Boolean simplification. The SymPy `SOPform` function takes the variables as the first argument and the minterms as the second argument. The function returns the sum-of-products expansion of the Boolean function in the smallest sum-of-products form. To use the SymPy `SOPform` function in Sage, first extract the variables and minterms of an expression.

We extract the variables from the first row of the truth table.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
expression = propcalc.formula("x & (y | z)")
table_list = expression.truthtable().get_table_list()
variables = table_list[0]
print(variables)
```

```
['x', 'y', 'z']
```

We make the variables compatible with the SymPy `SOPform` function by converting them to SymPy symbols.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
from sympy import symbols
sympy_variables = symbols(' '.join(variables))
print(sympy_variables)
```

```
(x, y, z)
```

We extract the minterms from the rows where the output is `True` .

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
minterms = [row[:-1] for row in table_list[1:] if row[-1]]
print(minterms)
```

```
[[True, False, True], [True, True, False], [True, True, True]]
```

Now that we have the variables and minterms, we can use the SymPy `SOPform` function to find the sum-of-products expansion of the Boolean function.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
from sympy.logic import SOPform
from sympy import symbols

def sympy_sop(expression):
    # Convert input expression to Sage formula object if necessary
    if isinstance(expression, str):
        formula_object = propcalc.formula(expression)
```

```
    elif isinstance(expression, sage.logic.boolformula.BooleanFormula):
        formula_object = expression
    else:
        raise ValueError("Invalid input: expression must be a string or a BooleanForm

    # Generate the truth table from the formula object
    truth_table = formula_object.truthtable()
    table_list = truth_table.get_table_list()

    # Extract variables and minterms from the truth table
    variables = table_list[0]
    minterms = [row[:-1] for row in table_list[1:] if row[-1]]

    # Convert Sage variables to SymPy symbols
    sympy_variables = symbols(' '.join(variables))

    # Use SymPy to compute the SOP form
    sop_result = SOPform(sympy_variables, minterms)

    return propcalc.formula(str(sop_result))
```

```

```

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
sympy_sop("x & (y | z)")
```

```
(x&y)|(x&z)
```

Let’s verify that the sum-of-products expansion we found with SymPy is equivalent to the original expression.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
sympy_sop("x & (y | z)") == propcalc.formula("x & (y | z)")
```

```
True
```

Now, we present a manual method for finding the sum of products by applying the Boolean identities. Let's find the sum-of-products expansion of the Boolean function

$$h(x, y) = x + \bar{y}$$

We can apply the Boolean identities and use Sage to verify our work. Currently, we have a sum of two terms but no products. We can apply the identity law to introduce the product terms. Now, we have the equivalent expression

Warning: Do not attempt to apply the identity law or null law within the `formula` function. If you try to directly apply the identity law within the formula function like so, `propcalc.formula("x & 1 | 1 ~y")`, Sage will raise an error because `propcalc.formula` interprets `1` as a variable. Variables cannot start with a number.

The `formula` function only supports variables and the following operators:

- `&` and
- `|` or
- `~` not
- `^` xor
- `->` ifthen
- `<->` ifandonlyif

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

[Math Processing Error]

Apply the complement law and verify that our transformed expression is equivalent to the original expression.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

[Math Processing Error]

Apply the distributive law and verify that our transformed expression is equivalent to the original expression.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

[Math Processing Error]

Apply the idempotent law and verify that our transformed expression is equivalent to the original expression.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

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

[Math Processing Error]

We started with the expression,

[Math Processing Error]

After applying the identity, complement, and distributive laws, we transformed the Boolean function into the sum-of-products expansion

[Math Processing Error]

CHAPTER OVERVIEW

11: Logic Gates

This chapter explains how to process binary inputs in Sage to produce specific outputs based on basic logic gates, such as ,AND, ,OR, and .NOT. Then, we show how these gates combine to form more complex circuits and integrate into everyday electronics, using built-in and customized Sage functions to simulate and analyze their behavior.

11.1: Logic Gates

Logic gates are the foundation of digital circuits. They process binary inputs to produce specific outputs. The basic logic gates are ,AND, ,OR, and .NOT. Derived gates include ,NAND, ,NOR, ,XOR, and .XNOR. Each gate has its own symbol and behavior defined by a truth table.

> **Note**
>
> Logic gates combine to form complex systems such as CPUs and memory circuits.

AND Gate

The `AND` gate produces a `1` only when both inputs are `1` .

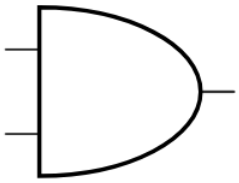


Figure 11.1.1. AND Gate


```
from sympy.logic.boolalg import And
from sympy.abc import A, B
And(A, B)
```

$/* <![CDATA[*/ A \wedge B /*]]> */$

Truth table for the `AND` gate:

```
# Generate truth table for AND gate
print("\nA | B | A AND B")
print("--|---|--------")
for A in [False, True]:
    for B in [False, True]:
        print(f"{int(A)} | {int(B)} | {int(bool(And(A, B)))}")
```

```
A | B | A AND B
--|---|--------
0 | 0 | 0
0 | 1 | 0
1 | 0 | 0
1 | 1 | 1
```

OR GATE

The `OR` gate produces a `1` if at least one input is `1` .

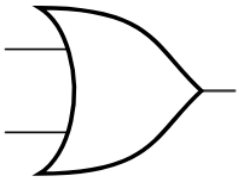


Figure 11.1.2. OR Gate


```
from sympy.logic.boolalg import Or
from sympy.abc import A, B
Or(A, B)
```

$/* <![CDATA[*/ A \vee B /*]]> */$

Truth table for the `OR` gate:

```
# Generate truth table for OR gate
print("\nA | B | A OR B")
print("--|---|--------")
for A in [False, True]:
    for B in [False, True]:
        print(f"{int(A)} | {int(B)} | {int(bool(Or(A, B)))}")
```

```
A | B | A OR B
--|---|--------
0 | 0 | 0
0 | 1 | 1
1 | 0 | 1
1 | 1 | 1
```

NOT Gate

The `NOT` gate inverts the input: `1` becomes `0`, and `0` becomes `1`.

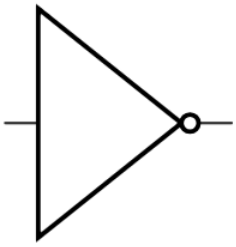


Figure 11.1.3. NOT Gate

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
from sympy.logic.boolalg import Not
from sympy.abc import A
Not(A)
```

$/* <![CDATA[*/ \neg A /*]]> */$

Truth table for the `NOT` gate:


```
# Generate truth table for NOT gate
print("\nA | NOT A")
print("--|-------")
for A in [False, True]:
    print(f"{int(A)} | {int(bool(Not(A)))}")
```

```
A | NOT A
--|-------
0 | 1
1 | 0
```

NAND Gate

`NAND` : Produces `0` only when both inputs are `1` .

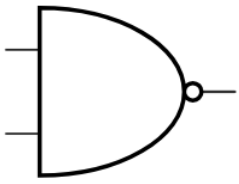


Figure 11.1.4. NAND Gate

NOR Gate

`NOR` : Produces `1` only when both inputs are `0` .

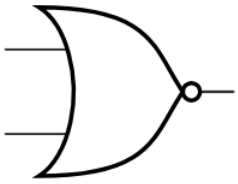


Figure 11.1.5. NOR Gate

XOR Gate

`XOR` : Produces `1` when inputs differ.

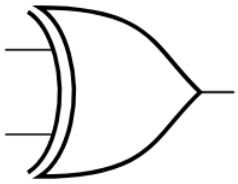


Figure 11.1.6. XOR Gate

XNOR Gate

`XNOR` : Produces `1` when inputs are the same.

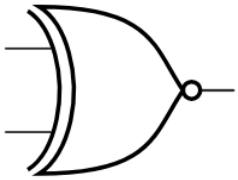


Figure 11.1.7. XNOR Gate


```
from sympy.logic.boolalg import And, Or, Not, Xor

def nand(A, B):
    return Not(And(A, B))

def nor(A, B):
    return Not(Or(A, B))
```

```
def xor(A, B):
    return Xor(A, B)

def xnor(A, B):
    return Not(Xor(A, B))

# User-defined inputs
A = 1 # Replace with 0 or 1 for input A
B = 0 # Replace with 0 or 1 for input B
gate = "xor" # Replace with "nand", "nor", "xor", or "xnor"


if gate == "nand":
    result = nand(A, B)
elif gate == "nor":
    result = nor(A, B)
elif gate == "xor":
    result = xor(A, B)
elif gate == "xnor":
    result = xnor(A, B)
else:
    result = "Invalid gate type! Please use 'nand', 'nor', 'xor', or 'xnor'."

result
```

$/* <![CDATA[*/ \text{True} /*]]> */$

11.2: Combinations of Logic Gates

Logic gates can be combined to create more complex circuits that perform specific tasks. By linking gates together, we can create circuits that process multiple inputs to produce a desired output. For example, combining an `AND` gate and a `NOT` gate results in a `NAND` gate, which inverts the output of the `AND` gate. More complex circuits, such as half-adders and multiplexers, are built by combining basic gates in strategic ways.

Let's look at a circuit. We evaluate this circuit by setting True for X, Y, and False for Z below using Sage.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
from sympy.logic.boolalg import And, Or, Not
from sympy.abc import X, Y, Z

# Define the logic circuit
F = Or(And(Not(X), Y, Z), And(X, Not(Y), Z), And(X, Y, Not(Z)), And(X, Y, Z))

# Evaluate the logic circuit with values for X, Y, and Z
circuit_output = F.subs({X: True, Y: True, Z: False})
circuit_output
```

$$/* <![CDATA[*/ \text{True} /*]]> */$$

Boolean algebra provides a way to simplify complex logic circuits. By using Boolean algebra rules, you can take a complicated circuit and reduce it to a simpler form without changing its functionality.

Here's a practical example. Consider the following Boolean expression, which combines several gates:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
# Original Boolean expression
from sympy import simplify
from sympy.logic.boolalg import And, Or, Not
from sympy.abc import x, y, z

# Define the expression
D = Or(And(Not(x), y, z), And(x, Not(y), z), And(x, y, Not(z)), And(x, y, z))
D
```

$$/* <![CDATA[*/ (x \wedge y \wedge z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge z \wedge \neg y) \vee (y \wedge z \wedge \neg x) /*]]> */$$

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
# Simplified Boolean expression
simplified_D = simplify(D)
simplified_D
```

$$/* <![CDATA[*/ (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) /*]]> */$$

Truth tables are a visual way to represent how inputs to a logic circuit map to its outputs. For each possible combination of inputs, the table shows the corresponding outputs, making it easier to analyze and understand the behavior of the circuit.

Let's create a truth table for the simplified circuit.

$$F = (x \text{ and } y) \text{ or } (x \text{ and } z) \text{ or } (y \text{ and } z)$$

Here, we will show the intermediary steps to find the final output of the function.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
from sympy.logic.boolalg import And, Or, truth_table
from sympy.abc import x, y, z

# Define the logic function
intermediate1 = And(x, y) # x AND y
intermediate2 = And(x, z) # x AND z
intermediate3 = And(y, z) # y AND z

# (x AND y) OR (x AND z) OR (y AND z)
final_output = Or(intermediate1, intermediate2, intermediate3)

# Variables and expressions
variables = [x, y, z]
expressions = [intermediate1, intermediate2, intermediate3, final_output]

# Header names and column widths
headers = ["x", "y", "z", "x AND y", "x AND z", "y AND z", "F"]
column_widths = [5, 5, 5, 10, 10, 10, 5] # Adjust widths as needed

# Print header row with adjusted spacing
header_row = " | ".join(h.ljust(w) for h, w in zip(headers, column_widths))
print(header_row)
print("-" * len(header_row))

# Generate and print the truth table rows
for row in truth_table(final_output, variables):
    inputs = row[0]
    outputs = [int(bool(expr.subs(dict(zip(variables, inputs))))) for expr in express
    table_row = " | ".join(str(int(bool(x))).ljust(w) for x, w in zip(list(inputs) + (
    print(table_row)
```

```
x     | y     | z     | x AND y    | x AND z    | y AND z    | F
-------------------------------------------------------------------
0     | 0     | 0     | 0          | 0          | 0          | 0
0     | 0     | 1     | 0          | 0          | 0          | 0
0     | 1     | 0     | 0          | 0          | 0          | 0
0     | 1     | 1     | 0          | 0          | 1          | 1
1     | 0     | 0     | 0          | 0          | 0          | 0
1     | 0     | 1     | 0          | 1          | 0          | 1
1     | 1     | 0     | 1          | 0          | 0          | 1
1     | 1     | 1     | 1          | 1          | 1          | 1
```

CHAPTER OVERVIEW

12: Finite State Machines

This chapter delves into a powerful abstract model, namely the *finite-state machines*. Beyond the theoretical framework, the content of this chapter demonstrates the use of Sage to define, model, build, visualize, and execute examples of state machines, showcasing their application in solving real-world problems.

12.1: Definitions and Components

The defining feature of any abstract machine is its memory structure, ranging from a *finite* set of states in the case of finite-state machines to more complex memory systems (e.g., *Turing machines* and *Petri nets*).

A ***Finite-State Machine (FSM)*** is a computational model that has a finite set of possible states S, a finite set of possible input symbols (the input alphabet) X, and a finite set of possible output symbols (the output alphabet) Z. The machine can exist in one of the states at any time, and based on the machine's input and its current state, it can transition to any other state and produce an output. The functions that take in the machine's current state and its input and map them to the machine's future state and its output are referred to as the *state transition* function and the *output* function, respectively. The default state of an FSM is referred to as the *initial state*.

Mealy State Machine

A Mealy finite-state machine is defined by the tuple (S, X, Z, w, t, s_0) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, a finite set that corresponds to the set of all memory configurations that the machine can have at any time.
- The state s_0 is called the *initial state*.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $Z = \{z_0, z_1, z_2, \ldots, z_k\}$ is the output alphabet.
- $w : S \times X \to Z$ is the output function, which specifies which output symbol $w(s, x) \in Z$ is written onto the output device when the machine is in state s and the input symbol x is read.
- $t : S \times X \to S$ is the next-state (or transition) function, which specifies which state $t(s, x) \in S$ the machine should move to when it is currently in state s and it reads the input symbol x.

Other Types of Finite State Machines

Moore Machine

In a ***Moore Machine***, the output depends *solely* on the current state. Unlike Mealy state machine, this machine must enter a new state for the output to change.

A Moore machine is also represented by the 6-tuple (S, X, Z, w, t, s_0) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, and s_0 is the initial state.
- The state s_0 is called the *initial state*.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $Z = \{z_0, z_1, z_2, \ldots, z_k\}$ is the output alphabet.
- $w : S \to Z$ is the output function, which specifies which output symbol $w(s) \in Z$ associated with the machine current state s.
- $t : S \times X \to S$ is the transition function, which specifies which next state $t(s, x) \in S$ the machine should move to when its current state is s and it has the input symbol x.

Finite-State Automaton

A *final state* (also known as the accepted state) is defined as a *special* predefined state that indicates whether an input sequence is valid or accepted by the finite-state machine. The set F of all final states is a subset of the states set S.

A ***Finite-State Automaton*** is a finite-state machine *with no output*, and it is represented by the 5-tuple (S, X, t, s_0, F) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, s_0 is the initial state, and F is the set of final states.
- The state s_0 is called the *initial state*.
- The subset $F \subset S$ is the set of all final states of the machine.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $t : S \times X \to S$ is the transition function, which specifies which next state $t(s, x) \in S$ the machine should move to when its current state is s and it has the input symbol x.

When the state machine processes a finite input sequence, it transitions through various states based on each input in the sequence and the current state of the machine. If, after processing the entire sequence, the machine lands in any of the *final states*, then the input sequence is considered valid (or recognized according to the machine's rules). Otherwise, the input sequence is rejected as invalid.

Deterministic Finite Automaton (DFA)

A ***Deterministic Finite Automaton (DFA)*** is a simplified automaton in which each state has exactly one transition for each input. DFAs are typically used for lexical analysis, language recognition, and pattern matching.

> **Note**
>
> A text parser or a string-matching application that recognizes a specific language or regular expressions are real-world examples of DFA use.

Nondeterministic Finite Automaton (NFA)

Unlike a DFA, an ***NFA*** allows multiple transitions for the same input or even transitions without consuming input (ϵ-transitions).

Turing Machine

A ***Turing Machine*** is an expansion of an FSM, which includes infinite tape memory representing both the input and output streams (shared stream). Unlike all other FSMs, a Turing machine can alter the input/output stream, and as such, it is capable of simulating any algorithm. Turing machines are the theoretical foundation for modern computation (any general-purpose computer executing any algorithm can be modeled as a Turing Machine).

Finite state machines are a foundational concept in computer science, often associated with tasks related to system designs (circuits and digital computers, algorithms, etc.). However, the vast and rich domain of applications of state machines extends far beyond simple simulations to the full control logic of complex industrial processes and workflows. These tasks can vary in complexity, ranging from a simple parity check to managing traffic patterns, a programming language compiler, or natural language recognition and processing.

State machines offer a structured way to model systems with discrete states and transitions. Different variants, such as the Mealy machine and Moore machine, have distinct characteristics and, as such, can adapt to various applications.

12.2: Finite State Machines in Sage

Although Sage includes a dedicated built-in rich module to handle various types of state machines, it may not always be sufficient to address certain use cases or implement specific custom behaviors of the machine. Additionally, the built-in module allows state machines to be defined and constructed in different ways, providing greater flexibility and making it more suitable from a programmer's perspective. However, it may not fully conform to the precise definition given earlier. This highlights that it is still possible to model, construct, display, and run relatively simple state machines by utilizing general-purpose tools, such as graphs and transition matrices, to represent and operate on state machines.

> **Note**
>
> While Sage provides basic tools to represent and simulate state machines, it may not natively support more complex state machine features such as parallel states or hierarchical transitions.

The Elevator State Machine

Let's design a basic controller to an elevator to show the process of defining states, creating a state transition graph, visualizing the state machine, and simulating its execution in Sage.

Consider a 3-level elevator (floors 1 through 3). The elevator has 3 buttons for users to select the destination floor (only one can be selected at a time). Depending on the current position and the selected floor, the elevator can go up, go down, or remain on the same floor.

Description of the Elevator FSM

This elevator system can be modeled and simulated using a finite-state machine with states $S = \{f_1, f_2, f_3\}$ representing each floor, the user inputs set $X = \{b_1, b_2, b_3\}$ (where b_i represents the button for i^{th} floor), and the outputs set $Z = \{U, D, N\}$ for 'going up', 'going down', or 'going nowhere'.

The components of this FSM are transcribed in the following table.

Table 12.2.1. The Elevator State Machine Definition

	next			output		
current	b_1	b_2	b_3	b_1	b_2	b_3
f_1	f_1	f_2	f_3	N	U	U
f_2	f_1	f_2	f_3	D	N	U
f_3	f_1	f_2	f_3	D	D	N

The following steps outline the approach to build and test the elevator controller system:

1. Define the elements of the Finite State Machine: States, Inputs, Transitions, and Outputs.
2. Construct the State Machine.
3. Run the machine using a sample input set.

Elements of the Elevator FSM

The first step is to define the states and transitions in the state machine, which can be represented using lists and dictionaries.

```
# Define state, input and output sets
states = ['f1', 'f2', 'f3']
inputs = ['b1', 'b2', 'b3']
outputs = ['U', 'D', 'N']

# Transitions as a dictionary {(current_state, input): next_state}
transitions = {
    ('f1', 'b1'): 'f1',
    ('f1', 'b2'): 'f2',
    ('f1', 'b3'): 'f3',

    ('f2', 'b1'): 'f1',
    ('f2', 'b2'): 'f2',
    ('f2', 'b3'): 'f3',

    ('f3', 'b1'): 'f1',
    ('f3', 'b2'): 'f2',
    ('f3', 'b3'): 'f3',
}

# The machine outputs control how the elevator would move
outputs = {
    ('f1', 'b1'): 'N',
    ('f1', 'b2'): 'U',
    ('f1', 'b3'): 'U',

    ('f2', 'b1'): 'D',
    ('f2', 'b2'): 'N',
    ('f2', 'b3'): 'U',

    ('f3', 'b1'): 'D',
    ('f3', 'b2'): 'D',
    ('f3', 'b3'): 'N',
}

# Display the machine configuration
print('States: ', states)
print('Transitions: ', transitions)
print('Outputs: ', outputs)
```

```
States:  ['f1', 'f2', 'f3']
Transitions:  {('f1', 'b1'): 'f1', ('f1', 'b2'): 'f2', ('f1', 'b3'): 'f3', ('f2', 'b1
```

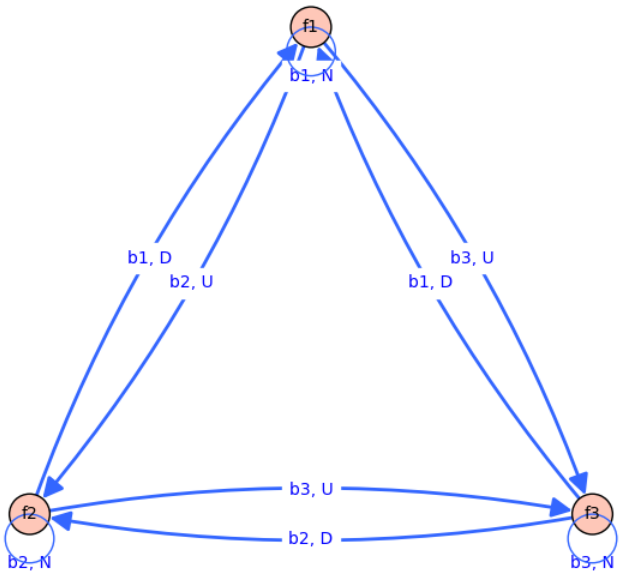
```
Outputs:  {('f1', 'b1'): 'N', ('f1', 'b2'): 'U', ('f1', 'b3'): 'U', ('f2', 'b1'): 'D'
```

Graph Model of the Elevator FSM

An FSM can be modeled as a graph where vertices represent the states, and the directed edge between vertices is the relationship between two states (the transition from one state to the other). The weight of a directed edge between two vertices represents the pair of input and output associated with the transition between the two states.

In Sage, the `DiGraph` class can be used to represent the states, transitions, and outputs of the state machine as a directed graph, leveraging the graph structure to visualize the state machine representation.


```
# 'DiGraph' is imported by default. If not, it can be imported as follow
# from sage.graphs.digraph import DiGraph

# Initialize a directed graph
elevator_fsm = DiGraph(loops=True)

# Add states as vertices
elevator_fsm.add_vertices(states)

# Add transitions and outputs as edges
for (_state, _input), next_state in transitions.items():
    _output = outputs[(_state, _input)]
    edge_label = f"{_input}, {_output}"
    elevator_fsm.add_edge(_state, next_state, label=edge_label)

# Display the graph (state machine)
elevator_fsm.show(
    figsize=[5.6, 5.6],
    layout='circular',
    vertex_size=250,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0,
)
```



The `show()` method renders a graphical representation of the state machine. Each vertex in the graph represents a state, and each directed edge represents a transition, labeled as (input, output).

Run the Elevator State Machine

Next, the state machine's behavior can be simulated by defining a function that processes a list of inputs and transitions through the states accordingly.


```
# Function to run the state machine
def run_state_machine(start_state, inputs):
    current_state = start_state
    for _input in inputs:
        print(f"Current state: {current_state}, Input: {_input}")

        if (current_state, _input) in transitions:
            current_output = outputs[(current_state, _input)]
            current_state = transitions[(current_state, _input)]
            print(
                f"Transitioned to: {current_state}\n"
                f"Output: {current_output}\n"
            )
```

```
        else:
            print(
                f"No transition/output available for input {_input} in state {current_
            )
            break

    print(f"Last state: {current_state}")

# Example of running the state machine
start_state = 'f2'
inputs = ['b1', 'b1', 'b3', 'b2']

run_state_machine(start_state, inputs)
```

```
Current state: f2, Input: b1
Transitioned to: f1
Output: D

Current state: f1, Input: b1
Transitioned to: f1
Output: N

Current state: f1, Input: b3
Transitioned to: f3
Output: U

Current state: f3, Input: b2
Transitioned to: f2
Output: D

Last state: f2
```

The `run_state_machine()` function simulates the state machine by processing a list of inputs starting from an initial state.

The Traffic Light State Machine

Let's design a simple traffic light controller to illustrate alternative methods for defining, visualizing, and executing finite state machines in Sage.

Consider a simplified traffic light system controlled by preset timers. This system operates through three phases that represent the flow of road traffic: Free-flowing, Slowing-down, and Halted. These phases correspond to the traffic light signals: green, yellow, and red, controlling the flow of traffic. The system uses three timer settings: 30 seconds, 20 seconds, and 5 seconds. When a timer expires, it triggers the transition to the next phase. Initially, the light is green, the traffic is flowing, and:

- When the 30-second timer expires, the traffic light changes from green to yellow, and traffic begins to slow down.
- When the 5-second timer expires, the traffic light changes from yellow to red, bringing traffic to a complete stop.
- When the 20-second timer expires, the traffic light changes from red to green, allowing traffic to start moving again.

Description of the Traffic Light FSM

In this traffic light system, the three phases representing the flow of road traffic: *Free-flowing* (F), *Slowing-down* (S), and *Halted* (H) are the states $S = \{F, S, H\}$ of the FSM. These phases correspond to the traffic light signals: green (G), yellow (Y), and red (

R), which are the outputs set $Z = \{G, Y, R\}$ of the system. The timers driving the transitions are the inputs set $X = \{t_{5s}, t_{20s}, t_{30s}\}$ of this traffic light system.

The following table summarize the elements of the traffic light FSM.

Table 12.2.2. The Traffic Light State Machine Definition

	next			output		
current	t_{5s}	t_{20s}	t_{30s}	t_{5s}	t_{20s}	t_{30s}
F	F	F	S	G	G	Y
S	H	S	S	R	Y	Y
Y	H	F	H	R	G	R

By applying the same steps and approach as in the previous section, the traffic light controller system will be built and tested, this time utilizing the Sage built-in module and functions.

Using `FiniteStateMachine' Module

Sage `FiniteStateMachine` built-in library provides a powerful tool to model, construct as well as simulate state machines of various systems. This module will be leveraged to showcase its capabilities on the given example, and demonstrating how it can be used to construct and display the FSM, manage its state transitions and outputs.

The command `FiniteStateMachine()` constructs an *empty* state machine (no states, no transitions).


```
from sage.combinat.finite_state_machine import FSMState

# FSM states, inputs and outputs
states = ['F', 'S', 'H']         # Free-flowing, Slowing-down, Halted
inputs = ['t30s', 't5s', 't20s'] # timer durations before state transitions
outputs = ['G', 'Y', 'R']        # traffic light: Green, Yellow, Red

# Create an empty state machine object
traffic_light_fsm = FiniteStateMachine()
traffic_light_fsm
```

```
Empty finite state machine
```

The function `FSMState()` defines a state for a given label. The `is_initial` flag can be set to true to set the current state as the *initial state* of the finite state machine. The method `add_state()` appends the created state to an existing state machine.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
# Define a new state then adding it
free_flowing = FSMState('F', is_initial=True)
traffic_light_fsm.add_state(free_flowing)

# Adding more states by their labels (saving state handlers, to use them in state tra
slowing_down = traffic_light_fsm.add_state('S')
halted = traffic_light_fsm.add_state('H')

# the FiniteStateMachine instance
traffic_light_fsm
```

```
Finite state machine with 3 states
```

To check whether or not a finite state machine has a state defined, `has_state()` method can be used by passing in the state label (case-sensitive).

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
traffic_light_fsm.has_state('F')
```

```
True
```

The function `states()` enumerates the list of all defined states of the state machine.

```
traffic_light_fsm.states()
```

```
['F', 'S', 'H']
```

The method `initial_states()` lists the defined initial state(s) of the state machine.


```
traffic_light_fsm.initial_states()
```

```
['F']
```

To define a new transition between two states (as well as the input triggering the transition, and the output associated with the state transition), the method `FSMTransition()` can be used. The method `add_transition()` attaches the defined transition to the state machine, and the function `transitions()` enumerates the list of all defined transitions of the state machine.

```
from sage.combinat.finite_state_machine import FSMTransition

# defining 3 transitions, and associating them the state machine
# After 30sec, transition from free-flowing to slowing-down, and set traffic light to
traffic_light_fsm.add_transition(
    FSMTransition(
        from_state=free_flowing,
        to_state=slowing_down,
        word_in='t30s',
        word_out='Y'
    )
)

# After 5sec, transition from slowing-down to halted and set traffic light to red
traffic_light_fsm.add_transition(FSMTransition(slowing_down, halted, 't5s', 'R'))

# After 30sec, transition from halted back to free-flowing, and set traffic light to
traffic_light_fsm.add_transition(FSMTransition(halted, free_flowing, 't20s', 'G'))

traffic_light_fsm.transitions()
```

```
[Transition from 'F' to 'S': 't30s'|'Y',
 Transition from 'S' to 'H': 't5s'|'R',
 Transition from 'H' to 'F': 't20s'|'G']
```

An alternative method for defining state transitions in an FSM is by using the `add_transitions_from_function()` method. This approach accepts a callable function that takes two states as arguments: the source state and the target state. The following code demonstrates how this can be implemented.


```
from sage.combinat.finite_state_machine import FSMTransition

# define state transitions, inputs and outputs
def transit_function(state1, state2):
```

```
    if state1=='F':
        if state2 =='S':
            return ('t30s', 'Y')

    elif state1=='S':
        if state2 =='H':
            return ('t5s', 'R')

    elif state1=='H':
        if state2 =='F':
            return ('t20s', 'G')

    # all other 'no-transition' combinations
    return None

traffic_light_fsm.add_transitions_from_function(transit_function)
traffic_light_fsm.transitions()
```

```
[Transition from 'F' to 'S': 't30s'|'Y',
 Transition from 'S' to 'H': 't5s'|'R',
 Transition from 'H' to 'F': 't20s'|'G']
```

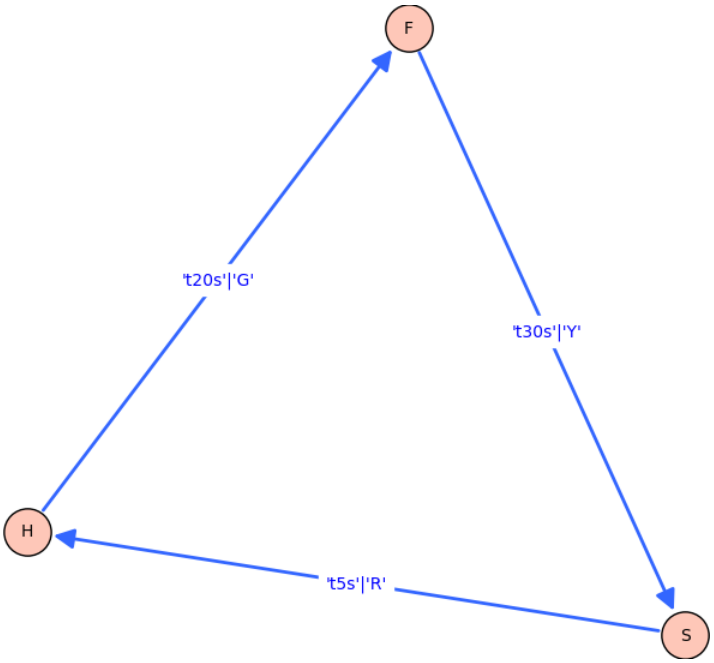
Once the states and transitions are defined, the state machine can be run using `process()` method, which then returns the intermediary outputs during the state machine run.


```
# pass in the initial state and the list of inputs
*_, outputs_history = traffic_light_fsm.process(
    initial_state=free_flowing,
    input_tape=['t30s', 't5s', 't20s'],
)

# print out the outputs of the state machine run
outputs_history
```

```
['Y', 'R', 'G']
```

The `graph()` command displays the graph representation of the state machine.


```
traffic_light_fsm.graph().show(
    figsize=[6, 6],
    vertex_size=800,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```



The `FiniteStateMachine` class also offers LaTeX representation of the state machine using the `latex_options()` method.

```
# define printout options
traffic_light_fsm.latex_options(
    format_state_label=lambda x: x.label(),
)

# display commands
print(latex(traffic_light_fsm))
```

```
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, initial] (v0) at (3.000000, 0.000000) {$F$};
\node[state] (v1) at (-1.500000, 2.598076) {$S$};
\node[state] (v2) at (-1.500000, -2.598076) {$H$};
\path[->] (v0) edge node[rotate=330.00, anchor=south] {$ $} (v1);
\path[->] (v2) edge node[rotate=30.00, anchor=south] {$ $} (v0);
\path[->] (v1) edge node[rotate=90.00, anchor=south] {$ $} (v2);
\end{tikzpicture}
```

Note that the LaTeX printout may not have all elements displayed. However, it can still be customized further. The following figure shows a rendering of the above LaTeX commands.

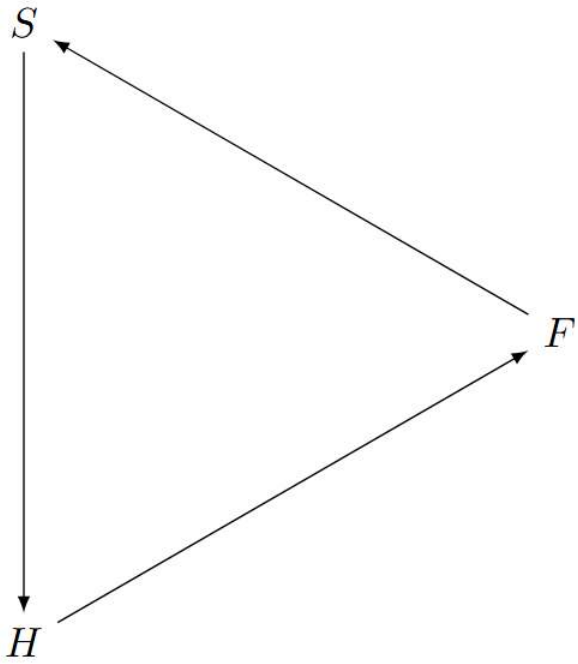


Figure **12.2.3.** FSM graph output.

Using `Transducer' Module

Sage `Transducer` is a specialization of the generic `FiniteStateMachine` class. The `Transducer` class creates a finite state machine that support optional final states, and whose transitions have input and output labels.

Let's see how to create another state machine using `Transducer` and for the same traffic light example.


```
# the module allows the instantiation of a state machine by passing
# the entire state machine definition to the constructor
state_machine_definition = {
    # from-state: [
    #   a list of tuples
    #   (to-state, input, output)
    # ]
    'F': [
```

```
        ('F', 't5s',  'G'),
        ('F', 't20s', 'G'),
        ('S', 't30s', 'Y'),
    ],
    'S': [
        ('H', 't5s',  'R'),
        ('S', 't20s', 'Y'),
        ('S', 't30s', 'Y'),
    ],
    'H': [
        ('H', 't5s',  'R'),
        ('F', 't20s', 'G'),
        ('H', 't30s', 'R'),
    ],
}

traffic_light_transducer = Transducer(
    state_machine_definition,
    initial_states=['F']
)
traffic_light_transducer
```

```
Transducer with 3 states
```

The member variable `input_alphabet` lists the set of the transducer inputs set.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
traffic_light_transducer.input_alphabet
```

```
['t5s', 't20s', 't30s']
```

The member variable `output_alphabet` lists the set of the transducer outputs set.

```
traffic_light_transducer.output_alphabet
```

```
['Y', 'G', 'R']
```

Since a `Transducer` is also a `FiniteStateMachine`, the method `has_state()` can still be used to check whether or not a given state exists in the defined transducer (by passing in the case-sensitive state label).


```
traffic_light_transducer.has_state('F')
```

```
True
```

The function `states()` enumerates the list of all defined states of the state machine.


```
traffic_light_transducer.states()
```

```
['F', 'S', 'H']
```

The method `initial_states()` lists the defined initial state(s) of the state machine.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
traffic_light_transducer.initial_states()
```

```
['F']
```

After defining the states and transitions, the transducer can be executed using the `process()` method from the parent `FiniteStateMachine` class. This method returns the intermediate outputs generated during the execution of the state machine.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
# fetching the initial state by its label
free_flowing = traffic_light_transducer.state('F')

# pass in the initial state and the list of inputs
*_, outputs_history = traffic_light_transducer.process(
    initial_state=free_flowing,
    input_tape=['t30s', 't5s', 't20s'],
)

outputs_history
```

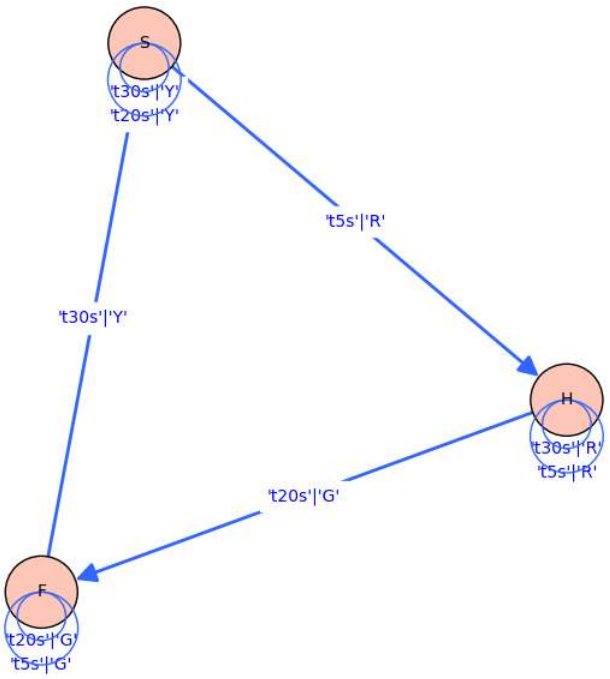
```
['Y', 'R', 'G']
```

The `graph()` command displays the graph representation of the transducer-based state machine.

Note

The `latex_options()` method of the base class `FiniteStateMachine` also is inherited and can also be used with `Transducer` state machine to output LaTeX representation.


```
traffic_light_transducer.graph().show(
    figsize=[6, 6],
    vertex_size=800,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```



The above are basic commands with a typical workflow of defining and running of simple finite state machines. The general structure of the state machine can be adapted to fit different use cases. The examples shown can be customized and fine-tuned to reflect more complex scenarios (more states, different input sequences, etc.)

12.3: State Machine in Action

Traffic light systems are crucial for regulating traffic. These systems use carefully coordinated signals to ensure safety for both vehicles and pedestrians. In the previous section, the traffic light system was modeled in an overly simplistic way. This section adds complexity to account for pedestrian presence, ensuring safe crossings while maintaining smooth traffic flow.

Traffic Light Controller: Problem Overview

Let's design a traffic light system for a two-way road with pedestrian crossings. This system coordinates the movement of vehicles and pedestrians using lights to indicate when vehicles can proceed, slow down, or stop, and when pedestrians can cross safely. Vehicle traffic lights include three signals: Red, Yellow, and Green. For simplicity, the pedestrian lights also use three signals: red, yellow, and green. Signal transitions are governed by timers, as described in the previous section, with each timer triggering a transition event after a predefined duration.

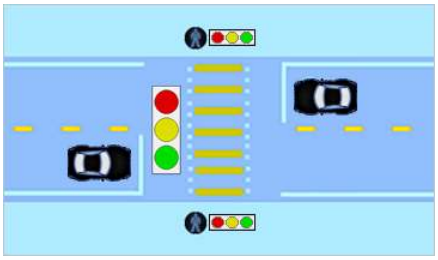


Figure **12.3.1.** Simple Traffic Light

The system must ensure safety and smooth traffic flow by coordinating appropriate traffic and pedestrian light configurations. Initially, vehicle traffic proceeds with a traffic green light, while pedestrian crossing is prohibited with a pedestrian red light.

Elements of the FSM Model

The goal here is to define a state machine model that can control this traffic light system, construct it, then put it under test. This system has different configurations of lights: Red (`R`), Yellow (`Y`), and Green (`G`) for traffic, and red (`r`), yellow (`y`), and green (`g`) for pedestrians. Note that not all possible combinations makes sense.

For inputs, the system leverage four independent timers with different preset durations and triggering different use cases as follows:

- 30sec timer :t30s: drives the traffic light transition from `G` to `Y` . The pedestrian light remains `r` and unchanged.
- 5sec timer :t5s: drives the traffic light transition from `Y` to `R` , and the pedestrian light transition from `r` to `g` .
- 20sec timer :t20s: drives the pedestrian light transition from `g` to `y` , while the traffic light remains `R` and unchanged.
- 10sec timer :t10s: drives the pedestrian light transition from `y` to `r` , and the traffic light transition from `R` back to `G` .

From the above timers and lights configurations, the following set of 4 distinct states emerges:

- State `Yr` : Where the traffic light is Yellow, pedestrian light is red.
- State `Rg` : Where the traffic light is Red, pedestrian light is green.
- State `Ry` : Where the traffic light is Red, pedestrian light is yellow.
- State `Gr` : Where the traffic light is Green, pedestrian light is red.

Finally, the system's outputs corresponding to each of the above are the light configurations and would be similar to the states:

- `(Y,r)` : Traffic light turning Yellow and the pedestrian light remains red.
- `(R,g)` : Traffic light turning Red and the pedestrian light turning green.
- `(R,y)` : Traffic light remains Red and the pedestrian light turning yellow.
- `(G,r)` : Traffic light turning Green and the pedestrian light turning red.

The following table summarize the elements of the new traffic light FSM.

Table 12.3.2. The Traffic Light State Machine Definition

		next			output			
current	t_{5s}	t_{10s}	t_{20s}	t_{30s}	t_{5s}	t_{10s}	t_{20s}	t_{30s}

Gr	–	–	–	Yr	–	–	–	(Y, r)
Yr	Rg	–	–	–	(R, g)	–	–	–
Rg	–	–	Ry	–	–	–	(R, y)	–
Ry	–	Gr	–	–	–	(G, r)	–	–

The symbol − indicates no state change, and no output change.

Construct the FSM

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
# FSM elements
states = ['Gr', 'Yr', 'Rg', 'Ry']
inputs = ['t5s', 't10s', 't20s', 't30s']
outputs = ['(G,r)', '(Y,r)', '(R,g)', '(R,y)']

# Traffic light state machine definition
significant_configs = [
    # from-state, to-state, input, output
    ('Gr', 'Yr', 't30s', '(Y,r)'),
    ('Yr', 'Rg', 't5s',  '(R,g)'),
    ('Rg', 'Ry', 't20s', '(R,y)'),
    ('Ry', 'Gr', 't10s', '(G,r)'),
]

machine_configs = {}
for config in significant_configs:
    (fr, to, evt, out) = config
    # Add the significant transition
    machine_configs[fr] = [
        (to, evt, out),
    ]
    # Add no-state change transitions
    machine_configs[fr].extend(
        [(fr, event, '_') for event in inputs if event != evt]
    )
```

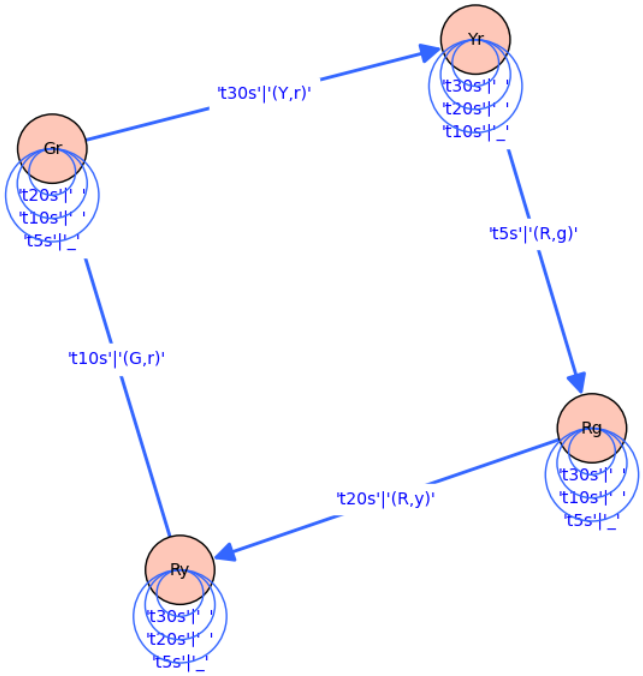
```
traffic_light_controller = FiniteStateMachine(
    machine_configs,
    initial_states=['Gr']
)

print('States:', traffic_light_controller.states())
print()
[print(_) for _ in traffic_light_controller.transitions()]
print()

print('-'*100)
print('FSM Config:', machine_configs)
traffic_light_controller
```

```
States: ['Gr', 'Yr', 'Rg', 'Ry']

Transition from 'Gr' to 'Yr': 't30s'|'(Y,r)'
Transition from 'Gr' to 'Gr': 't5s'|'_'
Transition from 'Gr' to 'Gr': 't10s'|'_'
Transition from 'Gr' to 'Gr': 't20s'|'_'
Transition from 'Yr' to 'Rg': 't5s'|'(R,g)'
Transition from 'Yr' to 'Yr': 't10s'|'_'
Transition from 'Yr' to 'Yr': 't20s'|'_'
Transition from 'Yr' to 'Yr': 't30s'|'_'
Transition from 'Rg' to 'Ry': 't20s'|'(R,y)'
Transition from 'Rg' to 'Rg': 't5s'|'_'
Transition from 'Rg' to 'Rg': 't10s'|'_'
Transition from 'Rg' to 'Rg': 't30s'|'_'
Transition from 'Ry' to 'Gr': 't10s'|'(G,r)'
Transition from 'Ry' to 'Ry': 't5s'|'_'
Transition from 'Ry' to 'Ry': 't20s'|'_'
Transition from 'Ry' to 'Ry': 't30s'|'_'

----------------------------------------------------------------------------------------------------
FSM Config: {'Gr': [('Yr', 't30s', '(Y,r)'), ('Gr', 't5s', '_'), ('Gr', 't10s', '_'),
```

Display the State Transition Graph

The FSM is visualized as before (a directed graph with nodes representing states and edges showing transitions).

```
traffic_light_controller.graph().plot(
    figsize=[6, 6],
    vertex_size=800,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```



Simulate a Full Cycle Run of the FSM

The simulation starts in the initial state (`Gr`) and transitions through all states.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
# pass in the initial state and the list of inputs
Gr = traffic_light_controller.state('Gr')
*_, outputs_history = traffic_light_controller.process(
    initial_state=Gr,
    input_tape=['t30s', 't5s', 't20s', 't10s', 't30s'],
)

# print out the outputs of the state machine run
print("FSM outputs:")
[print(_) for _ in outputs_history];
print()
```

```
FSM outputs:
(Y,r)
(R,g)
(R,y)
(G,r)
(Y,r)
```

It is worth noting that using Sage built-in modules could produce an error when handling transitions that were not defined in the FSM. For instance, in the previous example, if the timer durations _pattern_ for the input does not match the defined transitions, the output will be a `None` value. Similarly, an exception would be thrown if attempting to run the FSM starting at state that is not part of the FSM definition.

Index

D

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: Discrete Math with SageMath

Webpages: 66

All licenses found:

- Undeclared: 100% (66 pages)

By Page

Detailed Licensing

Overview

Title: Discrete Math with SageMath

Webpages: 66

All licenses found:

- Undeclared: 100% (66 pages)

By Page