

DISCRETE MATH WITH SAGEMATH



Hellen Colman
City Colleges of Chicago

Discrete Math with SageMath:
Learn math with open-source software

Authors: Zunaid Ahmed, Hellen Colman, Samuel Lubliner

Contributors: Allaoua Boughrira, Michael Kattner

Math Department
City Colleges of Chicago

For the best reading experience, we encourage you to use the book directly at

Discrete Math with SageMath: Learn math with open-source software

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 an 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 borne 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 were 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 adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

The LibreTexts name, logo, and book covers are trademarked by LibreTexts, Inc. (a 501(c)(3) not-for-profit organization) and protected against unauthorized use.

This text was compiled on 10/03/2026

TABLE OF CONTENTS

LibreTexts info

License

1: Getting Started

- 1.1: Intro to Sage
- 1.2: Data Types
- 1.3: Flow Control Structures
- 1.4: Defining Functions
- 1.5: Object-Oriented Programming
- 1.6: Display Values
- 1.7: Debugging
- 1.8: Documentation
- 1.9: Miscellaneous Features
- 1.10: Run Sage in the browser

2: Set Theory

- 2.1: Creating Sets
- 2.2: Cardinality
- 2.3: Operations on Sets

3: Combinatorics

- 3.1: Combinatorics

4: Logic

- 4.1: Logical Operators
- 4.2: Truth Tables
- 4.3: Analyzing Logical Equivalences

5: Relations

- 5.1: Introduction to Relations
- 5.2: Digraphs
- 5.3: Properties
- 5.4: Equivalence
- 5.5: Partial Order
- 5.6: Relations in Action

6: Functions

- 6.1: Functions
- 6.2: Recursion

7: Graph Theory

- 7.1: Basics
- 7.2: Plot Options
- 7.3: Paths

- 7.4: Isomorphism
- 7.5: Euler and Hamilton
- 7.6: Graphs in Action

8: Trees

- 8.1: Definitions and Theorems
- 8.2: Search Algorithms
- 8.3: Trees in Action

9: Lattices

- 9.1: Lattices
- 9.2: Tables of Operations

10: Boolean Algebra

- 10.1: Boolean Algebra
- 10.2: Boolean functions

11: Logic Gates

- 11.1: Logic Gates
- 11.2: Combinations of Logic Gates

12: Finite State Machines

- 12.1: Definitions and Components
- 12.2: Finite State Machines in Sage
- 12.3: State Machine in Action

Colophon

Index

Detailed Licensing

References

License

©2024–2025 Zunaid Ahmed, Hellen Colman, Samuel Lubliner

This work is licensed under the Creative Commons Attribution-International License (CC BY 4.0). To view a copy of this license, visit [CreativeCommons.org](https://creativecommons.org)



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

1.2: Data Types

1.3: Flow Control Structures

1.4: Defining Functions

1.5: Object-Oriented Programming

1.6: Display Values

1.7: Debugging

1.8: Documentation

1.9: Miscellaneous Features

1.10: Run Sage in the browser

1: Getting Started is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by LibreTexts.

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:

```
1 # Lines that start with a pound sign are comments
2 # and ignored by Sage
3 1+1
```

run

run all

restart

restart & run all

2

```
1 100 - 1
```

run run all restart restart & run all

99

1 3*4

run run all restart restart & run all

12

Sage supports two exponentiation operators:

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

run run all restart restart & run all

8

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

run run all restart restart & run all

8

Division in Sage:

```
1 5 / 3 # Returns a rational number
```

run run all restart restart & run all

5/3

```
1 5 / 3.0 # Returns a floating-point approximation
```

run run all restart restart & run all

```
1.666666666666667
```

Integer division and modulo:

```
1 5 // 3 # Returns the quotient
```

run run all restart restart & run all

```
1
```

```
1 5 % 3 # Returns the remainder
```

run

run all

restart

restart & run all

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.

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

run

run all

restart

restart & run all

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.

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

run

run all

restart

restart & run all

True

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

1.1: Intro to Sage is shared under a [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/) license and was authored, remixed, and/or curated by LibreTexts.

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.

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

run

run all

restart

restart & run all

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.

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

run

run all

restart

restart & run all

$$\frac{\sqrt{2}}{\log(3)} \quad (1.4.1)$$

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

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

runrun allrestartrestart & run all

$$\frac{2}{3} \quad (1.4.1)$$

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

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

runrun allrestartrestart & run all

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

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

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

runrun allrestartrestart & run all

```
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).

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

runrun allrestartrestart & run all

```
[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:

```
1 l[0]
```

run run all restart restart & run all

```
1
```

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

```
1 len(1)
```

run run all restart restart & run all

```
4
```

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

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

run

run all

restart

restart & run all

```
(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.

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

run

run all

restart

restart & run all

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

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

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

run run all restart restart & run all

```
<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`.

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

run run all restart restart & run all

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

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

```
1 d = {
2     "title": "Discrete Math with SageMath",
3     "institution": "City Colleges of Chicago",
4     "topics_covered": [
5         "Set Theory",
6         "Combinations and Permutations",
7         "Logic",
8         "Quantifiers",
9         "Relations",
10        "Functions",
11        "Recursion",
12        "Graphs",
13        "Trees",
14        "Lattices",
15        "Boolean Algebras",
16        "Finite State Machines"
17    ],
```

```
18     "format": ["Web", "PDF"]
19 }
20 type(d)
```

runrun allrestartrestart & run all

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

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

```
1 import pprint
2 pprint.pprint(d)
```

runrun allrestartrestart & run all

```
{'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']}
```

1.2: Data Types is shared under a [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/) license and was authored, remixed, and/or curated by LibreTexts.

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.

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

runrun allrestartrestart & run all

```
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:

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

run

run all

restart

restart & run all

```
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:

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

run run all restart restart & run all

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

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

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

run run all restart restart & run all

```
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.

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

run

run all

restart

restart & run all

```
[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:

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

run

run all

restart

restart & run all

```
[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:

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

runrun allrestartrestart & run all

```
0
1
2
3
4
```

1.3: Flow Control Structures is shared under a [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/) license and was authored, remixed, and/or curated by LibreTexts.

1.4: Defining Functions

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

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

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

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

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

runrun allrestartrestart & run all

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

```
1 pascal_row(5)
```

run

run all

restart

restart & run all

```
[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` :

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

run

run all

restart

restart & run all

```
-----
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:

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

runrun allrestartrestart & run all

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:

```
1 pascal_row(-5)
```

run run all restart restart & run all

```
-----
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.

```
1 def pascal_row(n):
2     """
3     Return row `n` of Pascal's triangle.
4
5     INPUT:
6     - ``n`` -- non-negative integer; the row number of Pascal's triangle to return.
7       The row index starts from 0, which corresponds to the top row.
8
9     OUTPUT: list; row `n` of Pascal's triangle as a list of integers.
10
11     EXAMPLES:
12     This example illustrates how to get various rows of Pascal's triangle (0-indexed) :
13
14         sage: pascal_row(0) # the top row
15         [1]
16         sage: pascal_row(4)
17         [1, 4, 6, 4, 1]
18
19     It is an error to provide a negative value for `n`:
20         sage: pascal_row(-1)
21         Traceback (most recent call last):
22         ...
23         ValueError: `n` must be a non-negative integer
24
25     .. NOTE::
```

run run all restart restart & run all

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

```
1 help(pascal_row)
```

run

run all

restart

restart & run all

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:

```
1 pascal_row??
```

run

run all

restart

restart & run all

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.

1.4: Defining Functions is shared under a [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/) license and was authored, remixed, and/or curated by LibreTexts.

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.

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

runrun allrestartrestart & run all

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

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

run run all restart restart & run all

```
<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.

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

run run all restart restart & run all

```
['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.

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

run

run all

restart

restart & run all

```
<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.

[1.5: Object-Oriented Programming](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by LibreTexts.

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.

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

run

run all

restart

restart & run all

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

Using `print()` removes the quotes.

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

run

run all

restart

restart & run all

```
Hello world!
```

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

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

run

run all

restart

restart & run all

$$\frac{\sqrt{2}}{\log(3)}$$

(1.2.1)

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

```
1 a = x^2
2 b = pi
3 show(a)
4 show(b)
```

run

run all

restart

restart & run all

$$x^2$$

(1.2.1)

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

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

run

run all

restart

restart & run all

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

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

```
1 %display latex
2 # Notice we don't need the show() function
3 sqrt(2) / log(3)
```

run

run all

restart

restart & run all

$$\sqrt{5}$$

$$\frac{\sqrt{2}}{\log(3)}$$

(1.2.1)

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

1 zz

run

run all

restart

restart & run all

Integer Ring

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

```
1 %display plain
2 sqrt(2) / log(3)
```

run

run all

restart

restart & run all

$\sqrt{2}/\log(3)$

1 zz

Integer Ring

1.6: Display Values is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by LibreTexts.

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.

```
1 # Run this cell and see what happens
2 1message = "Hello, World!"
3 print(1message)
```

run

run all

restart

restart & run all

```
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:

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

run run all restart restart & run all

Hello, World!

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

```
1 print(Hi)
```

run run all restart restart & run all

```
-----  
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.

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

run run all restart restart & run all

```
Hi
```

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

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

run run all restart restart & run all

```
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` :

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

run run all restart restart & run all

```
-----  
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.

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

run run all restart restart & run all

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

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

```
1 l = [1, 2, 3]
```

```
2 l.len()
```

run run all restart restart & run all

```
-----  
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.

```
1 len(l)
```

run run all restart restart & run all

```
3
```

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

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

runrun allrestartrestart & run all

```
-----
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()` .

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

runrun allrestartrestart & run all

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

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

```
1 factorial(-5)
```

runrun allrestartrestart & run all

```
-----
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.py:1177
    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.py:598
    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:1633
    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.py: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.

```
1 factorial(5)
```

run

run all

restart

restart & run all

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:

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

run run all restart restart & run all

```
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.

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

run run all restart restart & run all

```
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.

1.7: Debugging is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by LibreTexts.