Calcul Mathématique avec `Python/SageMath`

Cyrille Merleau Nono Saha

June 30, 2025

CIMPA TRAP 2025

University of Leipzig/ScaDS.AI
Max Planck Institute for Mathematics in the Sciences
Lancaster Leipzig University

Plan du cours

Team and resources

Team and resources

Instructors

- Nono Saha Cyrille Merleau
- The participants (You)

Course resources (s)

- GitHub
- Readings and link to the course tools
- No specific material: A reading list is given on the references page.

Course Timetable (s)

- Lectures: 01.07, 04.07, 08.07, and 10.07

Programming environment

Setting your environment up

- Conda with a specific python version

```
$conda --version
```

- Check your package manager

```
$pip --version or pip3 --version
```

- Create your conda environment

```
$conda create --name CIMPA sage python=3.11
$conda activate CIMPA
```

- Run sage on your

```
$sage
```

Introduction to Git

What is Git?

- Modern version control system
- Mature, actively maintained, open-source project (Linus Torvalds, 2005)
- Works well on a wide range of operating systems and IDEs

Main characteristics

- Distributed VCS *vs* CVC
- Performance: Algorithms, File content/names
- Security: Top priority and SHA1 hashing algorithm for commits, content, file-folder
- Flexibility: small and large projects, many OS, branches and tags

Essential git commands

Init and clone directories

- **git init**: creates an empty Git repository
- **git clone**: clone a repository into a new directory
- More importantly, learn to use the "manual"

Basic commands

- **git pull**: get recent updates from the remote to the local branch
- **git add** ¡file or folder¿: add file contents to the index
- **git commit** -m ¡some comments¿: record changes to the repository
- **git push** ¡origin¿ ¡branch¿: update remote refs along with associated objects

More commands

- **git merge**, **git fetch**, **git status**, **git log**
- More on Git Document

Introduction to Python

What is Python?

- A programming language that boasts ease of use
- Dynamic typing and garbage-collected language
- Batteries included (pypi.python.org)

Several advantages

- Code readability with the use of significant indentation via the off-side rule
- High level and for general purposes
- Structured, functional and OO-programming

Important!!!!!

- Network sockets, database handles, windows, and file descriptors are not included in the garbage-collection
- Need of other methods (e.g. destructors)

Variables: basic types

What is a variable?

- A value stored in computer memory
- It should have a name and store the corresponding value
- Use a combination of alphanumeric characters and the underscore character for names
- Convention recommends lowercase characters with words separated by an underscore for readability

Type system:

- **Boolean** (or **bool**): e.g. True, False
- **Float**: e.g. 1., 1.0, 2.4
- **Integer** (or **int**): e.g. 1, 34
- **Complex Number** (or **complex**): e.g. 1+1j , 1+0j, 4j, etc..

Variable: dynamic types

Therefore, most basic operations will coerce a variable to a consistent type suitable for the operation.

Boolean operations and comparisons

```
1 and True
## True
0 or 1
## 1
not 0
## True
not (0+0j)
## True
not (0+1j)
## False
```

```
5. > 1
## True
5. == 5
## True
1 > True
## False
(1+0j) == 1
## True
'abc' < "ABC"
## False
```

Variables: basic mathematical operations

Math operations

```
1 + 5
## 6
1 + 5.
## 6.
1 * 5.
## 5.0
True * 5
## 5
(1 + 0j) - (1 + 1j)
## -1j
```

```
5 / 1.
## 5.0
5 / 2
## 2.5
5 // 2
## 2
5 % 2
## 1
7 ** 2
## 49
```

Now, what if I do?

```
"abc" + 5  ## Error? What type of error? Why?
"abc" + str(5) ## Does that correct the prev error?
"abc" ** 2 # What about this?
"abc" * 3 # And this?
```

Variables: casting and assignment

Casting using type functions: e.g. float(), int(), etc..

```
float ("0.5")
## 0.5
float(True)
## 1.0
int(1.1)
## 1
int("2")
## 2
```

```
bool(0)
## False
bool("hello")
## ??
str(3.14159)
## "3.14159"
str(True)
## "True"
```

Now, what if I do?

```
int("2.1")
## Error? What type of error? Why?
```

Variable assignment

```
x = 100 ## Assign a value of 100 to a variable named x
a = b = 5 ## Assign a value of 5 to both variables a and b
```

Variables: special values

No missing values and non-finite floating point values are available. There is a None type similar to NULL in `R`, `Java`, `JavaScript`.

```
1 / 0
## Error in py_call_impl(callable, dots$args,
dots$keywords):
ZeroDivisionError: division by zero
## Detailed traceback:
## File "<string>", line 1, in <module>

1. / 0 # qu'en est il de ceci?

float("nan")
## nan
float("-inf")
## -inf, we can do 5 > float("inf")
```

Variables: string literals

Strings can be defined using a couple of different ways,

```
'allows embedded "double" quotes'
##' allows embedded "double" quotes'
```

```
"allows embedded 'single' quotes"
## "allows embedded 'single' quotes"
```

Strings can also be triple quoted, using single or double quotes, which allows the string to span multiple lines.

```
"""line one line two line three"""
## 'line one\nline two\nline three'
```

Several methods are possible:

```
x = "Hello wolrd! 1234"
x.find("!"), x.isalnum(), x.title(), x.swapcase(),
x.split()
```

Variables: sequence types

Lists in Python

Python lists are heterogeneous, ordered, mutable containers of objects.

```
x = [0,1,1,0]; x
## [0, 1, 1, 0], we can use subsetting with x[start:stop:step]

[0, True, "abc"]
## [0, True, 'abc'] mutate an element, x[-1] = 2

[0, [1,2], [3,[4]]]
## [0, [1, 2], [3, [4]]], can we assign?

x = [0,1,1,0]
type(x)
## <class 'list'> we can sort with x.sort()

y = [0, True, "abc"]
type(y)
## <class 'list'> is y.sort() still work?
```

Variables: sequence types

Unpacking lists in Python

Unpacking into multiple variables when doing "assignment",

```
x, y = [1,2]
x
## 1 similarly we can do x, y = [[0,1], [2, 3]]
y
## 2
x, y = [1, [2, 3]]
x
## 1 or something like (x1,y1), (x2,y2)=[[0,1], [2, 3]]
y
## [2, 3]
```

Extended unpacking:

```
x, *y = [1,2,3] ## what about this x, y = [1,2,3]?
y ## [2, 3] what about *x, y = [1,2,3]?
```

Variables: sequence types

Tuples in `Python`

`Python` tuples are heterogenous, ordered, immutable (or non-mutable) containers of values.

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

(1,True," abc")
## (1, True, 'abc')

(1,(2,3))
## (1, (2, 3))

x = (1,2,3)
x[2] = 5 ## What will happen here?
```

Variable: ranges as a sequence type

These are the last sequence types and are somewhat special - ranges are homogeneous, ordered, and immutable "containers" of integers.

Examples:

```
range(10) ## range(0, 10)

range(0,10) ## range(0, 10)

range(0,10,2) ## range(0, 10, 2)

range(10,0,-1) ## range(10, 0, -1)

list(range(10)) ## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. )
```

Remarque: What about this `list(range(10,0,-1))`?

Homework

Programming like a hipster!

- Write a program that computes a square root of any given integer. NB: making use of no Python library.
- Given a list, write a program that returns an ascendent sorted list.

An introduction to OOP

Basic syntax

These are the basic components of Python's object-oriented system.

```python
class Rectangle:
	"""An abstract representation of a rectangle"""
	# Attributes
	p1 = (0,0)
	p2 = (1,2)

	# Methods
	def area(self):
		return abs(self.p1[0] - self.p2[0])
		*abs(self.p1[1] - self.p2[1])

	# Setters
	def setP1(self, p1):
		self.p1 = p1
```

Interfaces and abstract classes

Two important points

- Interfaces are classes that contain methods without implementations
- Abstract classes are classes with at least one method without implementation

Example

```python
class AbstractRectangle(abc.ABC):
        def __init__(self, p1=(0,0), p2=(1,2)):
                self.p1 = p1
                self.p2 = p2

        @abc.abstractmethod
        def area(self):
                pass
```

Objects

What is an object?

- Objects are instances of a class
- Objects can also represent different states of a class
- Objects are coherent entities that store data and the code (or instructions) working on that data.

Example

```
x = Rectangle()
x.area()
```

```
y = Rectangle(p2= (5,4))
y.area()
```

Class attributes

We can examine all of a class's methods and attributes using `dir()`,

```
dir(Rectangle)
['__class__','__delattr__','__dict__','__dir__',
'__doc__','__eq__','__format__','__ge__',
'__getattribute__','__gt__','__hash__',
'__init__','__init_subclass__','__iter__','__le__',
'__lt__','__module__','__ne__','__new__',
'__reduce__','__reduce_ex__','__repr__',
'__setattr__','__sizeof__','__str__',
'__subclasshook__','__weakref__','area']
```

Where did p1 and p2 go?

```
dir(Rectangle())
['__class__','__delattr__','__dict__','__dir__'...
```

Instantiation (constructors)

When instantiating a class (e.g. `Rectangle()`) we invoke the `__init__()` method if it is present in the classes' definition.

```python
class Rectangle:
        """An abstract representation of a rectangle"""
        # Constructor
        def __init__(self, p1 = (0,0), p2 = (1,1)):
                self.p1 = p1
                self.p2 = p2

        # Methods
        def area(self):
                return ((self.p1[0] - self.p2[0])
                *(self.p1[1] - self.p2[1]))
```

Method chaining

We will see several objects (e.g., `exp(-x).diff().diff()`) that allow for method chaining to construct a pipeline of operations. We can achieve the same effect by having our class methods return 'self'.

```python
class Rectangle:
        """An abstract representation of a rectangle"""
        # Constructor
        def __init__(self, p1 = (0,0), p2 = (1,1)):
                self.p1 = p1
                self.p2 = p2
        # Methods
        def area(self):
                return ((self.p1[0] - self.p2[0])
                *(self.p1[1] - self.p2[1]))
```

Object string formating

All class objects have a default print method/string conversion method, but the default behaviour is not very useful,

```
print(Rectangle())
## <__main__.rect object at 0x290aa1a60>

str(Rectangle())
## '<__main__.rect object at 0x290aa1ca0>'
```

Both of the above are handled by the `__str__()` method, which is implicitly created for our class - we can override this,

```
def rect_str(self):
      return f"Rectangle[{self.p1}, {self.p2}]
      area={self.area()}"

Rectangle.__str__ = rect_str
```

Class representation

There is another special method responsible for printing the object (see `Rectangle()` above), called `__repr__()`, which is used to print the class representation. If possible, this is intended to be a valid Python expression that can recreate the object.

```python
def rect_repr(self):
        return f"Rectangle({self.p1}, {self.p2})"

rect.__repr__ = rect_repr
```

```python
Rectangle()
## Rectangle((0, 0), (1, 1))
```

```python
repr(Rectangle())
## Rectangle((0, 0), (1, 1))
```

OOP: Inheritance

Part of the object-oriented system is that classes can inherit from other classes, meaning they gain access to all of their parent's attributes and methods. It models a **Is a** relationship.

In an inheritance relationship:

- Classes inherited from another are derived classes, subclasses, or subtypes.
- Classes from which other classes are derived are called base classes or superclasses.
- A derived class is said to derive, inherit, or extend a base class.

```
class Square(Rectangle):
        pass
```

```
Square()
## Rectangle((0, 0), (1, 1))
```

OOP: Multiple inheritance

A class can be derived from more than one superclass in Python. This is called multiple inheritance.

```python
class Worm:
        def __init__ (self, name):
                self.name = name
        def eat(self):
                print(self.name +" swallows")

class Fly:
        def __init__ (self, name):
                self.name = name
        def eat(self):
                print(self.name +" is nibbling..")

class ButterFly(Worm, Fly):
        pass
```

OOP Inheritance: overriding methods

```
class Square(Rectangle):
        def __init__(self, p1=(0,0), l=1):
                assert isinstance(l, (float, int)), "numnber, please"
                p2 = (p1[0]+l, p1[1]+l)
                self.l = l
                super().__init__(p1, p2)

        def setP1(self, p1):
                self.p1 = p1
                self.p2 = (self.p1[0]+self.l, self.p1[1]+self.l)
                return self
        def setP2(self, p2):
                raise RuntimeError("Squares take l not p2")
```

OOP: Making an object iterable

When using an object with a for loop, Python looks for the `__iter__()` method, which is expected to return an iterator object (e.g. iter() of a list, tuple, etc...).

```python
class Rectangle:
""" An object representation of a rectangle"""
# Constructor
       def __init__(self, p1 = (0,0), p2 = (1,1)):
              self.p1 = p1; self.p2 = p2
       # Methods
       def area(self):
              return ((self.p1[0] - self.p2[0])
              *(self.p1[1] - self.p2[1]))
       def __iter__(self):
              return iter( [ self.p1, (self.p1[0],
                                   self.p2[1]), self.p2,(self.p2[0],
                                   self.p1[1])])
```

OOP: Composition in Python

Composition is an OOP concept that models a relationship. In composition, a class known as a composite contains an object of another class, referred to as a component. In other words, a composite class has a component of another class.

Example: We already used it in the previous Example, but how? and where?

Remarks

- Composition is more flexible than inheritance because it models a loosely coupled relationship
- Changes to a component class have minimal or no effects on the composite class
- Designs based on composition are more suitable for change

Example of composition in Python

```python
class Salary:
        def __init__(self, pay, bonus):
                self.pay = pay
                self.bonus = bonus
        def annual_salary(self):
                return (self.pay*12)+self.bonus

class EmployeeOne:
        def __init__(self, name, age, pay, bonus):
                self.name = name
                self.age = age
                self.obj_salary = Salary(pay, bonus) # comp
        def total_sal(self):
                return self.obj_salary.annual_salary()
```

OOP: Aggregation in `Python`

Aggregation is a concept in which an object of one class can own or access another independent object of another class.

- It represents **Has-A**'s relationship.
- It is a unidirectional association, i.e. a one-way relationship. For Example, a department can have students, but the opposite is not possible, and thus it is unidirectional.
- In Aggregation, both entries can survive individually, which means ending one entity will not affect another.

Example of aggregation in Python

```python
class Salary:
        def __init__(self, pay, bonus):
                self.pay = pay
                self.bonus = bonus

        def annual_salary(self):
                return (self.pay*12)+self.bonus

class EmployeeOne:
        def __init__(self, name, age, sal):
                self.name = name
                self.age = age
                self.agg_salary = sal # Aggregation

        def total_sal(self):
                return self.agg_salary.annual_salary()
```

OOP: more relationships...

- **Association**: which expresses a uses-a relationship. For Example, a student may be associated with a course. They will use the course. This relationship is common in database systems, where it is often represented by one-to-one, one-to-many, and many-to-many associations.
- **Delegation**: which models a **can-do** relationship, where an object hands a task over to another object, which takes care of executing the task.
- **Dependency injection**: a design pattern you can use to achieve loose coupling between a class and its components. With this technique, you can provide an object's dependencies from the outside rather than inheriting or implementing them in the object itself.

Polymorphism in Python

A set of classes implementing the same interface with specific behaviours for concrete classes is a great way to unlock Polymorphism.

Polymorphism is when you can use objects of different classes interchangeably because they share a standard interface.

Example

Python strings, lists, and tuples are all sequence data types. This means that they implement an interface that's common to all sequences.

We can use them in similar ways. For Example, you can:

- Loop them because they provide the `.__iter__()` method
- Items are accessed through the `__getitem__()` method
- Determine their number of items because they include the `.__len__()` method

Wrapping up the OOP in `Python`

1. Python classes and how to use them to make your code more reusable, modular, flexible, and maintainable
2. Classes are the building blocks of object-oriented programming in Python
3. With classes, you can solve complex problems by modelling real-world objects, their properties, and their behaviours
4. Classes provide an intuitive and human-friendly approach to complex programming problems, making your life more pleasant.
5. We can use special classes such as interfaces and abstract classes to unlock properties like Polymorphism in python
6. Classes can interact through associations, aggregations, composition, inheritance, dependency injection and delegation

Exercise: Object-Oriented Programming in Python

Tasks:

1. Define a Python class `Book` with attributes for `title`, `author`, and `year`.
2. Write an `__init__` method that initializes these attributes.
3. Add a method `description` that returns a string like `"Title by Author (Year)"`.
4. Create an instance of your class for the book "The Hobbit" by J.R.R. Tolkien, published in 1937, and print its description.

Optional: Extend the class with a method to check if the book is a classic (published before 1970).

Solution: Object-Oriented Programming in Python

```python
class Book:
        def __init__(self, title, author, year):
                self.title = title
                self.author = author
                self.year = year

        def description(self):
                return f"{self.title} by {self.author} ({self.year})"

        def is_classic(self):
                return self.year < 1970

# Create an instance
hobbit = Book("The Hobbit", "J.R.R. Tolkien", 1937)
print(hobbit.description()) # The Hobbit by J.R.R. Tolkien (1937)
print(hobbit.is_classic()) # True
```

Exercise: Object-Oriented Programming — Authors and Books

Define two Python classes representing books and authors.

Tasks:

1. Define a class `Author` with attributes: `firstname`, `lastname`, `affiliation`, and `address`.
2. Define a class `Book` with attributes: `title`, `year`, and `author` (where `author` is an object of class `Author`).
3. Write appropriate `__init__` methods for both classes.
4. Add a `description` method in `Book` that returns a string like: `"Title (Year) by Firstname Lastname, Affiliation"`.
5. Create an instance of `Author` for "J.R.R. Tolkien" (affiliated with "University of Oxford", address: "Oxford, UK") and a `Book` instance for "The Hobbit" (1937, by Tolkien). Print the book description.

Solution: Author and Book Classes in Python

```python
class Author:
def __init__(self, firstname, lastname, affiliation, address):
       self.firstname = firstname
       self.lastname = lastname
       self.affiliation = affiliation
       self.address = address
class Book:
       def __init__(self, title, year, author):
              self.title = title
              self.year = year
              self.author = author # Author object

def description(self):
       return f"{self.title} ({self.year}) by {self.author.firstname} {
           self.author.lastname}, {self.author.affiliation}"

# Create Author and Book instances
tolkien = Author("J.R.R.", "Tolkien", "University of Oxford", "Oxford,UK
    ")
hobbit = Book("The Hobbit", 1937, tolkien)
print(hobbit.description())
```

`Sympy` and symbolic mathematics

Sympy basics: importing a module/package in Python

What is Sympy?

SymPy is a Python library that enables symbolic mathematics, including calculus operations like differentiation and integration. It allows users to work with mathematical expressions, equations, and functions symbolically rather than just numerical computation

How to install Sympy?

```
pip install sympy
```

Example test

```python
import sympy as sym
x, y = sym.symbols('x y')
```

Sympy basics: differentiation, integration, and limits.

Defferentiation

Use `sympy.diff(function, variable)` to find the derivative of a function with respect to a variable:

```
f = x**2 + 2*x + 1
df_dx = sym.diff(f, x) # Derivative of f with respect to x
print(df_dx) # Output: 2*x + 2
```

Integration

```
integral_f = sym.integrate(f, x) #Indefinite integral of f with respect
     to x
print(integral_f) #Output: x**3/3 + x**2 + x
```

Limits

Use sympy.limit(function, variable, point) to evaluate the limit of a function as the variable approaches a specific point:

```
limit_f = sym.limit((sym.sin(x) / x), x, 0)
print(limit_f) # Output: 1
```

Application to Physics (1): Uniformly Accelerated Motion

The position of an object under constant acceleration:

$$x(t) = x_0 + v_0 t + \frac{1}{2} a t^2$$

Symbolic computation with sympy:

```python
from sympy import symbols, Eq, solve, N

# Define symbols
x, x0, v0, a, t = symbols('x x0 v0 a t')

# Kinematics equation
eq = Eq(x, x0 + v0*t + (1/2)*a*t**2)

# Given: x0 = 0, v0 = 5 (m/s), a = 2 (m/s^2), t = 3
subs = {x0: 0, v0: 5, a: 2, t: 3}
x_val = eq.subs(subs)

print("Position at t=3s:", N(x_val.rhs)) # Output: 21.0
```

Application to Physics (2): The Ideal Gas Law

The ideal gas equation relates pressure, volume, temperature, and the number of moles:

$$PV = nRT$$

Suppose we wish to solve for the pressure P.

Symbolic computation with `sympy`:

```python
P, V, n, R, T = symbols('P V n R T') # Define symbols

# Ideal gas law
eq = Eq(P*V, n*R*T)

# Solve for pressure
P_sol = solve(eq, P)[0]
print("P =", P_sol)

# Numerical example: V=10 L, n=2 mol, T=300 K, R=0.0821 L*atm/(mol*K)
values = {V: 10, n: 2, T: 300, R: 0.0821}
P_num = P_sol.subs(values)
print("Pressure:", P_num) # Output: 4.926
```

Solving ODE Systems via Laplace Transform with `sympy`

$$\dot{x} = 3x + 4y, \quad \dot{y} = -4x + 3y, \quad x(0) = 1,\, y(0) = 0$$

```python
from sympy import symbols, Function, laplace_transform,
    inverse_laplace_transform, dsolve
from sympy.abc import t, s

# Define functions
x = Function('x')
y = Function('y')

# System as equations
eq1 = x(t).diff(t) - 3*x(t) - 4*y(t)
eq2 = y(t).diff(t) + 4*x(t) - 3*y(t)

# Solve system with initial conditions
sol = dsolve([eq1, eq2], [x(t), y(t)], ics={x(0):1, y(0):0})

print(sol)
```

Result: $x(t) = e^{3t}\cos(4t), \quad y(t) = e^{3t}\sin(4t)$

Live demo...

Reading list

- C. Thomas Wu, (2010) "An Introduction to Object-Oriented Programming with Java", McGraw-Hill
- Priestley, (2003) "Practical Object-Oriented Design with UML", McGraw-Hill
- Peter Van Roy and Seif Haridi (2004) "Concepts, Techniques, and Models of Computer Programming", MIT Press.
- Haskell, R. & Hanna, A. (2008) "Introduction to Functional Programming in Python and C++", Cambridge University Press.
- Andrew Troelsen and Philip Japikse (2020) "Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming", Apress.
- Axel Rauschmayer (2019) "JavaScript for impatient programmers", Dr. Axel Rauschmayer.

List of important commands

- Create your environment with

```
conda create -f ‘yml file’ -n ‘env_name’ python=3.9
```

- Activate that env

```
condo activate <env_name>
```

- Install Jupyter

```
conda install -y jupyter
```

- Add the current env in the notebook kernel

```
python -m ipykernel install --user --name <env_name> --display-name
     ’kernel_name’
```

- Export your conda in yml file

```
conda env export | grep -v ‘^prefix:’ > env.yml
```

Group Theory with SageMath

Introduction to Group Theory with SageMath

Group Theory studies algebraic structures called *groups*.

A **group** is a set G with a binary operation $\cdot$ satisfying:

- **Closure:** $a, b \in G \implies a \cdot b \in G$
- **Associativity:** $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- **Identity:** $\exists e \in G$ such that $e \cdot a = a \cdot e = a$
- **Inverses:** $\forall a \in G, \exists b \in G$ such that $a \cdot b = b \cdot a = e$

Example in SageMath: Symmetric group S_3

```python
G = SymmetricGroup(3)
print("Elements of S_3:", list(G))
print("Is S_3 abelian?", G.is_abelian())
```

Output:

- Elements of S_3: `[(1,2,3), (1,3,2), (1,2), (2,3), (1,3), ()]`
- Is S_3 abelian? `False`

Examples of Groups in SageMath

1. Symmetric group S_3

```
G = SymmetricGroup(3)
print(list(G)) # Permutations of 3 elements
print(G.is_abelian()) # Is S_3 abelian? (False)
```

2. Integers under Addition $(\mathbb{Z}, +)$

```
Z = IntegerModRing(7)
print(list(Z)) # Elements: 0,1,...,6 (modulo 7)
print(Z.is_commutative()) # True, Z/7Z is abelian under addition
```

3. Cyclic group C_4 of order 4

```
C4 = CyclicPermutationGroup(4)
print(C4.order()) # 4
print(C4.is_cyclic()) # True
```

4. Quaternion group Q_8

```
Q8 = QuaternionGroup()
print(Q8.is_nonabelian()) # True
print(Q8.center()) # Center elements of Q_8
```

Subgroup Operations in SageMath

Let's work with the symmetric group S_4:

```
G = SymmetricGroup(4)
# 1. List all subgroups (up to isomorphism)
subs = G.subgroups()

# 2. Generate a subgroup from elements
a = G((1,2)) # permutation (1 2)
b = G((3,4)) # permutation (3 4)
H = G.subgroup([a, b])
print(f "Order of H: {H.order()}, Elements:, {list(H)}")

# 3. Normal closure (smallest normal subgroup containing an element)
N = G.normal_closure([a])

# 4. Intersection of two subgroups
H1 = G.subgroup([G((1,2))]); H2 = G.subgroup([G((1,2,3,4))])
intersection = H1.intersection(H2)
```

Exercise: Subgroup Operations in S_3

Let $G = S_3$ be the symmetric group of degree 3.

Tasks:

1. List all the subgroups of G.
2. Find a subgroup H of order 2 and list its elements.
3. Is H a normal subgroup of G? Justify your answer.
4. Find the intersection of H with a subgroup K of order 3 in G.

Hint: You can use SageMath commands like

```
G = SymmetricGroup(3)
G.subgroups()
```

Usefull tutorials