Crash Course:
SageMath for Post-Quantum Cryptography

Lorenz Panny

Academia Sinica

Post-Quantum Crypto Mini-School, Taipei, 12 July 2022

What?

- SageMath is a mathematical software package.

What?

- SageMath is a mathematical software package.
- It has interfaces to many other libraries and programs.

What?

- SageMath is a mathematical software package.
- It has interfaces to many other libraries and programs.
- All functionality is accessible from Python.

What?

- SageMath is a mathematical software package.
- It has interfaces to many other libraries and programs.
- All functionality is accessible from Python.

$\Longrightarrow$ No need to learn separate ad-hoc programming languages for each of PARI/GP, Singular, Magma, etc. — *Just learn Sage!*

What?

- SageMath is a mathematical software package.
- It has interfaces to many other libraries and programs.
- All functionality is accessible from Python.

$\Longrightarrow$ No need to learn separate ad-hoc programming languages for each of PARI/GP, Singular, Magma, etc. — *Just learn Sage!*

\+ Tons of functionality, easy to use.

What?

- SageMath is a mathematical software package.
- It has interfaces to many other libraries and programs.
- All functionality is accessible from Python.

$\Longrightarrow$ No need to learn separate ad-hoc programming languages for each of PARI/GP, Singular, Magma, etc. — *Just learn Sage!*

\+ Tons of functionality, easy to use.

\+ Free, open source, made by volunteers. (**You** can help!)

What?

- SageMath is a mathematical software package.
- It has interfaces to many other libraries and programs.
- All functionality is accessible from Python.

$\Longrightarrow$ No need to learn separate ad-hoc programming languages for each of PARI/GP, Singular, Magma, etc. — *Just learn Sage!*

\+ Tons of functionality, easy to use.

\+ Free, open source, made by volunteers. (**You** can help!)

\- Sometimes buggy, and sometimes painfully slow.

Note to Python veterans

There are *some* differences between Sage and Python.

Note to Python veterans

There are *some* differences between Sage and Python.

- `.sage` files are treated as Sage code.
- `.py` files are treated as Python code.

Example

```
user@machine:~$ sage
+--------------------------------------------------------------------+
| SageMath version 9.6, Release Date: 2022-05-15                     |
| Using Python 3.10.5. Type "help()" for help.                       |
+--------------------------------------------------------------------+
```

Example

```
user@machine:~$ sage
+--------------------------------------------------------------------+
| SageMath version 9.6, Release Date: 2022-05-15                     |
| Using Python 3.10.5. Type "help()" for help.                       |
+--------------------------------------------------------------------+
sage: a = Mod(42, 1337)          # 42 + 1337ℤ
```

Example

```
user@machine:~$ sage
+--------------------------------------------------------------------+
| SageMath version 9.6, Release Date: 2022-05-15                     |
| Using Python 3.10.5. Type "help()" for help.                       |
+--------------------------------------------------------------------+
sage: a = Mod(42, 1337)          # 42 + 1337ℤ
sage: a*a                        # remembers modulus
427
```

Example

```
user@machine:~$ sage
+--------------------------------------------------------------------+
| SageMath version 9.6, Release Date: 2022-05-15                     |
| Using Python 3.10.5. Type "help()" for help.                       |
+--------------------------------------------------------------------+
sage: a = Mod(42, 1337)          # 42 + 1337ℤ
sage: a*a                        # remembers modulus
427
sage: a^2                        # ^ means exponentiation
427
```

Example

```
user@machine:~$ sage
+--------------------------------------------------------------------+
| SageMath version 9.6, Release Date: 2022-05-15                     |
| Using Python 3.10.5. Type "help()" for help.                       |
+--------------------------------------------------------------------+
sage: a = Mod(42, 1337)          # 42 + 1337ℤ
sage: a*a                        # remembers modulus
427
sage: a^2                        # ^ means exponentiation
427
sage: a^1336                     # Fermat primality test
973                              # ≠1 ⟹ not prime
```

Example

```
user@machine:~$ sage
+--------------------------------------------------------------------+
| SageMath version 9.6, Release Date: 2022-05-15                     |
| Using Python 3.10.5. Type "help()" for help.                       |
+--------------------------------------------------------------------+
sage: a = Mod(42, 1337)          # 42 + 1337ℤ
sage: a*a                        # remembers modulus
427
sage: a^2                        # ^ means exponentiation
427
sage: a^1336                     # Fermat primality test
973                              # ≠1 ⟹ not prime
sage: is_prime(1337)             # Sage agrees
False
```

Example

```
user@machine:~$ sage
+--------------------------------------------------------------------+
| SageMath version 9.6, Release Date: 2022-05-15                     |
| Using Python 3.10.5. Type "help()" for help.                       |
+--------------------------------------------------------------------+
sage: a = Mod(42, 1337)          # 42 + 1337ℤ
sage: a*a                        # remembers modulus
427
sage: a^2                        # ^ means exponentiation
427
sage: a^1336                     # Fermat primality test
973                              # ≠1 ⟹ not prime
sage: is_prime(1337)             # Sage agrees
False
sage: factor(1337)               # ...and factors it
7 * 191
```

Parents (1)

`Mod(42,1337)` showcases a very important concept:

> In Sage, every *object* remembers its *parent* structure.

Parents (1)

`Mod(42,1337)` showcases a very important concept:

> In Sage, every *object* remembers its *parent* structure.

```
sage: a = Mod(42,1337)
sage: a
42
```

Parents (1)

`Mod(42,1337)` showcases a very important concept:

> In Sage, every *object* remembers its *parent* structure.

```
sage: a = Mod(42,1337)
sage: a
42
sage: a.parent()
Ring of integers modulo 1337
```

Parents (1)

`Mod(42,1337)` showcases a very important concept:

> In Sage, every *object* remembers its *parent* structure.

```
sage: a = Mod(42,1337)
sage: a
42
sage: a.parent()
Ring of integers modulo 1337
sage: parent(42)
Integer Ring
```

Parents (1)

`Mod(42,1337)` showcases a very important concept:

> In Sage, every *object* remembers its *parent* structure.

```
sage: a = Mod(42,1337)
sage: a
42
sage: a.parent()
Ring of integers modulo 1337
sage: parent(42)
Integer Ring
sage: parent(42/5)
Rational Field
```

Parents (2)

The parent determines the behavior of many operations:

Parents (2)

The parent determines the behavior of many operations:

```
sage: -(42)                    # ℤ
-42
sage: -Mod(42,1000)            # ℤ/1000
958
sage: -Mod(42,1337)            # ℤ/1337
1295
```

Parents (3)

Sage automatically coerces between parents in many cases:

Parents (3)

Sage automatically coerces between parents in many cases:

```
sage: Mod(42,1000) * 77
234
sage: 5042 == Mod(42,1000)
True
```

Parents (3)

Sage automatically coerces between parents in many cases:

```
sage: Mod(42,1000) * 77
234
sage: 5042 == Mod(42,1000)
True
```

This example does not work:

```
sage: Mod(42,1000) + 1/7
# ...
TypeError: unsupported operand parent(s) for +:
    'Ring of integers modulo 1000' and 'Rational Field'
```

Parents (4)

Parents are objects as well and can be constructed explicitly:

Parents (4)

Parents are objects as well and can be constructed explicitly:

```
sage: R = Zmod(1337)
sage: R
Ring of integers modulo 1337
```

Parents (4)

Parents are objects as well and can be constructed explicitly:

```
sage: R = Zmod(1337)
sage: R
Ring of integers modulo 1337
sage: type(R)
<class 'sage.rings.finite_rings.integer_mod_ring
                .IntegerModRing_generic_with_category'>
```

Parents (5)

We can explicitly convert objects to a particular parent:

Parents (5)

We can explicitly convert objects to a particular parent:

```
sage: R = Zmod(1337)
```

```
sage: a = 1379
sage: a.parent()
Integer Ring
sage: b = R(a)
sage: b
42
```

Parents (5)

We can explicitly convert objects to a particular parent:

```
sage: R = Zmod(1337)
```

```
sage: a = 1379
sage: a.parent()
Integer Ring
sage: b = R(a)
sage: b
42
```

```
sage: c = 42/5
sage: c.parent()
Rational Field
sage: R(c)
1078
```

Parents (summary)

1. In Sage, every *object* remembers its *parent* structure.
2. The parent determines the behavior of many operations.
3. Sage automatically coerces between parents in many cases.
4. Parents are normal objects, can be constructed explicitly.
5. We can explicitly convert objects to a particular parent.

Operators

- `+`, `-`, `*` all do the obvious thing.

Operators

- `+`, `-`, `*` all do the obvious thing.
- `%` is modulo.

Operators

- `+`, `-`, `*` all do the obvious thing.
- `%` is modulo.
- `/` is true division, `//` is approximate division:

```
sage: 5 / 3
5/3
sage: 5 // 3     # rounding down
1
```

Operators

- `+`, `-`, `*` all do the obvious thing.
- `%` is modulo.
- `/` is true division, `//` is approximate division:

```
sage: 5 / 3
5/3
sage: 5 // 3     # rounding down
1
```

- `^` is exponentiation (*not* bitwise XOR).

Operators

- `+`, `-`, `*` all do the obvious thing.
- `%` is modulo.
- `/` is true division, `//` is approximate division:

```
sage: 5 / 3
5/3
sage: 5 // 3     # rounding down
1
```

- `^` is exponentiation (*not* bitwise XOR).
- `~` is multiplicative inverse (*not* bitwise NOT):

```
sage: ~7
1/7
sage: ~Mod(65537,2678871240)
444196313
sage: 65537 * 444196313 % 2678871240
1
```

Control structures (1)

...are all inherited from Python:

- Conditionals:

```
if condition1:
    # do something
elif condition2:
    # do a different thing
else:
    # do yet another different thing
```

Control structures (1)

...are all inherited from Python:

- Conditionals:

```
if condition1:
    # do something
elif condition2:
    # do a different thing
else:
    # do yet another different thing
```

- Loops:

```
while condition:
    # do something
```

```
for thing in sequence:
    # process the thing
```

Control structures (2)

- Defining and calling a function:

```
def fun(x,y):
    # do some things with x and y, giving z
    return z

result = fun("Hello", 5)
```

Control structures (2)

- Defining and calling a function:

```
def fun(x,y):
    # do some things with x and y, giving z
    return z

result = fun("Hello", 5)
```

- *List comprehensions*:

```
sage: xs = [n*(n+1)//2 for n in range(10)]
sage: xs
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

Polynomials (1)

To construct $R := F[x]$, we can use the syntax:

```
sage: R.<x> = F[]
```

Polynomials (1)

To construct $R := F[x]$, we can use the syntax:

```
sage: R.<x> = F[]
```

This defines the *generator* `x` and its parent `R`. Example:

```
sage: R.<x> = Zmod(101)[]
sage: R
Univariate Polynomial Ring in x over Ring of integers modulo 101
sage: x
x
sage: x.parent() is R
True
```

Polynomials (2)

The resulting variable x can be used to create polynomials:

```
sage: R.<x> = Zmod(101)[]
sage: f = x^5 - 3*x^2 + x + 7          # <-------
sage: f.parent()
Univariate Polynomial Ring in x over Ring of integers modulo 101
sage: f
x^5 + 98*x^2 + x + 7
sage: f.degree()                       # degree
5
sage: f(42)                            # evaluation
69
sage: f(42).parent()
Ring of integers modulo 101
sage: f % (x^2-44)                     # and +, *, //, etc.
18*x + 77
```

Polynomials (3)

We can change the base ring of many Sage objects:

```
sage: R.<x> = Zmod(101)[]
sage: f = x^5 - 3*x^2 + x + 7
sage: g = f.change_ring(ZZ)
sage: g
x^5 + 98*x^2 + x + 7
sage: g.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: g(42)
130864153
sage: g(42) % 101
69
```

Indexing, iteration, etc.

Most Sage objects support conversions to lists, indexing, etc.:

```
sage: R.<x> = Zmod(101)[]
sage: f = R([7, 1, -3, 0, 0, 1])
sage: f
x^5 + 98*x^2 + x + 7
sage: list(f)
[7, 1, 98, 0, 0, 1]
sage: f[2]
98
sage: f[9999]
0
sage: for c in f: print(c)
7
1
98
0
0
1
```

Quotients (1)

- Quotients of parents are simply parents again.
- We've already seen an example:

```
sage: ZZ.quotient(13)
Ring of integers modulo 13
sage: ZZ.quotient(13) is Zmod(13)
True
```

Quotients (2)

- Example: $\mathbb{Z}[\mathrm{i}] \cong \mathbb{Z}[x]/(x^2+1)$.

```
sage: P.<x> = ZZ[]
sage: R.<i> = P.quotient(x^2+1)
sage: i^2
-1
sage: i.parent()
Univariate Quotient Polynomial Ring in i
          over Integer Ring with modulus x^2 + 1
```

Quotients (2)

- Example: $\mathbb{Z}[\mathrm{i}] \cong \mathbb{Z}[x]/(x^2+1)$.

```
sage: P.<x> = ZZ[]
sage: R.<i> = P.quotient(x^2+1)
sage: i^2
-1
sage: i.parent()
Univariate Quotient Polynomial Ring in i
          over Integer Ring with modulus x^2 + 1
```

- Projection and lifting:

```
sage: R(x^3+x^2+1)
-i
sage: (i+1).lift()
x + 1
sage: R(x^3+x^2+1).lift()
-x
```

Finite fields

...are constructed using GF:

```
sage: GF(7)
Finite Field of size 7
```

Finite fields

...are constructed using GF:

```
sage: GF(7)
Finite Field of size 7
```

```
sage: GF(49)
Finite Field in z2 of size 7^2
sage: GF(49).modulus()
x^2 + 6*x + 3
sage: F49.<t> = GF(49)
sage: t^2 + 6*t + 3
0
sage: a = F49.random_element()
sage: a
2*t + 5
sage: a in GF(7)
False
sage: a^8 in GF(7)
True
```

Linear algebra (1)

Matrices and vectors are well-supported:

```
sage: A = matrix([[1,2],[3,4]])
sage: A
[1 2]
[3 4]
sage: v = vector([5,6])
sage: v
(5, 6)
sage: A * v
(17, 39)
```

Vectors are truly one-dimensional; can become row or column:

```
sage: v * A
(23, 34)
sage: A.transpose() * v
(23, 34)
```

Linear algebra (2)

Solving linear systems over arbitrary fields is straightforward:

```
sage: A = matrix(GF(71), [[1,4,7], [9,1,3]])
sage: y = vector(GF(71), [5,5])
sage: x = A.solve_right(y)            # particular solution
sage: x
(41, 62, 0)
sage: A*x
(5, 5)
sage: ker = A.right_kernel()          # the kernel
sage: ker
Vector space of degree 3 and dimension 1
                              over Finite Field of size 71
Basis matrix:
[ 1 12 64]
sage: ker.basis()
[(1, 12, 64)]
sage: A * vector([1, 12, 64])
(0, 0)
```

Lattices

...are usually best represented as row matrices over $\mathbb{Z}$ in Sage.

Example:

```
sage: L = matrix(ZZ, [[21621621617,1], [99999999977,0]])
sage: L
[21621621617             1]
[99999999977             0]
sage: R = L.LLL()
sage: R
[          13           37]
[-2405721711   845253590]
sage: R[0,:]
[13 37]
```

(There also exists an `IntegralLattice` class, but I almost never use it.)

Multivariate polynomials (1)

Creating multivariate polynomial rings works exactly as before:

```
sage: R.<x,y,z> = QQ[]
sage: f = x^2 + y^2 - z^2
sage: f(3,4,5)
0
```

Multivariate polynomials (1)

Creating multivariate polynomial rings works exactly as before:

```
sage: R.<x,y,z> = QQ[]
sage: f = x^2 + y^2 - z^2
sage: f(3,4,5)
0
```

To access specific coefficients, we can index with monomials:

```
sage: g = x^2 + x*y + 4/5*z^2 - 4/3*z
sage: g[z^2]
4/5
sage: g[x*y]
1
sage: g[y]
0
```

Multivariate polynomials (2)

We can use many variables:

```
sage: R = PolynomialRing(GF(97), 'v', 100)
sage: R
Multivariate Polynomial Ring in v0, v1, v2, ..., v99
                              over Finite Field of size 97
sage: R.gens()
(v0, v1, v2, v3, v4, v5, ..., v96, v97, v98, v99)
sage: f = R.random_element()
sage: f
-5*v27*v52 + 39*v2*v55 + 39*v60*v73 - 47*v13*v94
sage: f.monomials()
[v27*v52, v2*v55, v60*v73, v13*v94]
```

Multivariate polynomials (3)

Powerful equation-solving tools are available:

```
sage: R.<x,y,z> = QQ[]
sage: I = R.ideal([x^2+y^2-z^2, z-y-1, y-x-1])
sage: I.groebner_basis()
[z^2 - 6*z + 5, x - z + 2, y - z + 1]
sage: I.elimination_ideal([y,z])
Ideal (x^2 - 2*x - 3) of Multivariate Polynomial Ring
                                    in x, y, z over Rational Field
sage: factor(x^2 - 2*x - 3)
(x - 3) * (x + 1)
sage: I.variety()
[{z: 5, y: 4, x: 3}, {z: 1, y: 0, x: -1}]
```

Elliptic curves & isogenies

Wait for Thursday! ☺

Cryptographic building blocks (1)

Crucially: Hashing. Many functions are available:

```
sage: import hashlib
sage: hashlib.algorithms_guaranteed
{'md5', 'sha1', 'sha224', 'sha256', 'sha384', 'sha512',
 'sha3_224', 'sha3_256', 'sha3_384', 'sha3_512',
  'shake_128', 'shake_256', 'blake2b', 'blake2s'}
```

Usage:

```
sage: hashlib.shake_128(b'Hi!').digest(16)
b't<\xea\xa5\xa7\x88\xd7\xed]\t\xc6\xf1z\xc4e\x96'
sage: list(hashlib.shake_128(b'Hi!').digest(16))
[116, 60, 234, 165, 167, 136, 215, 237, 93, 9, 198,
                                241, 122, 196, 101, 150]
```

Cryptographic building blocks (2)

Other helpful tools come from *PyCryptodome*:

```
sage: from Crypto.Cipher import AES
sage: zeroes = bytes([0] * 16)
sage: aes = AES.new(zeroes, AES.MODE_ECB)
sage: aes.encrypt(zeroes).hex()
66e94bd4ef8a2c3b884cfa59ca342b2e
```

Help, I'm lost!

- Vast documentation online including plenty of examples.
  (In some cases, the documenation is outdated, incomplete, unhelpful, or wrong.)

Help, I'm lost!

- Vast documentation online including plenty of examples.
  (In some cases, the documenation is outdated, incomplete, unhelpful, or wrong.)

- Use ? in a Sage shell to quickly access the documentation:

```
sage: R.<x> = Zmod(257)[]
sage: R.random_element?
Signature:
     R.random_element(degree=(-1, 2), *args, **kwds)
Docstring:
     Return a random polynomial of given degree
     or with given degree bounds.
#  ...
```

Help, I'm lost!

- Vast documentation online including plenty of examples.
  (In some cases, the documenation is outdated, incomplete, unhelpful, or wrong.)

- Use ? in a Sage shell to quickly access the documentation:

```
sage: R.<x> = Zmod(257)[]
sage: R.random_element?
Signature:
     R.random_element(degree=(-1, 2), *args, **kwds)
Docstring:
     Return a random polynomial of given degree
     or with given degree bounds.
#  ...
```

- Feel free to ask!

Some exercises

https://yx7.cc/docs/sage/sagepqc_taipei_exercises.pdf