oblivious

Release 7.0.0

Nth Party, Ltd.

Apr 26, 2023

CONTENTS

Python library that serves as an API for common cryptographic primitives used to implement OPRF, OT, and PSI protocols.

CHAPTER

ONE

PURPOSE

This library provides pure-Python implementations, Python wrappers for libsodium and mcl, and additional utility methods for cryptographic primitives that are often used to implement oblivious pseudorandom function (OPRF), oblivious transfer (OT), and private set intersection (PSI) protocols.

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install oblivious
```

It is possible to install the library together with packages that bundle dynamic/shared libraries, such as rbcl and/or mclbn256 (note that in some environments, the brackets and/or commas may need to be escaped):

```
python -m pip install oblivious[rbcl]
python -m pip install oblivious[mclbn256]
python -m pip install oblivious[rbcl,mclbn256]
```

The library can be imported in the usual ways:

```
import oblivious
from oblivious import ristretto
from oblivious import bn254
```

2.1 Examples

This library supports concise construction of elliptic curve points and scalars. The examples below use the `ristretto` module that provides data structures for working with the Ristretto group:

```
>>> from oblivious.ristretto import point, scalar
>>> p = point.hash('abc'.encode()) # Point derived from a hash of a string.
>>> s = scalar() # Random scalar.
>>> t = scalar.from_int(0) # Scalar corresponding to the zero residue.
```

Built-in Python operators are overloaded to support point operations (such as addition, subtraction, negation, and equality) and scalar operations (such as multiplication by a scalar and inversion of scalars):

```
>>> q = s * p
>>> p == (~s) * q
True
>>> p == ((~s) * s) * p
True
>>> p + q == q + p
True
>>> t * p == p - p
True
```

The point and scalar classes have common conversion methods that correspond to those supported by bytes objects (and in some cases, these classes are themselves derived from bytes):

```
>>> hex = '35c141f1c2c43543de9d188805a210abca3cd39a1e986304991ceded42b11709'
>>> s = scalar.from_hex(hex)
>>> s.hex()
'35c141f1c2c43543de9d188805a210abca3cd39a1e986304991ceded42b11709'
```

In addition, Base64 conversion methods are included to support concise encoding and decoding of point and scalar objects:

```
>>> s.to_base64()
'NcFB8cLENUPenRiIBaIQq8o805oemGMEmRzt7UKxFwk='
>>> s == scalar.from_base64('NcFB8cLENUPenRiIBaIQq8o805oemGMEmRzt7UKxFwk=')
True
```

For more information and background about the underlying mathematical structures and primitives found in the ristretto module, consult materials about Curve25519, the Ristretto group, and the related Ed25519 system.

2.2 Using Pure Python or a Shared/Dynamic Library

Each module within this library can export two variants of its primitives and data structures: one corresponding to pure-Python implementations and another corresponding to shared/dynamic library wrappers.

For example, the ristretto module exports two container classes/namespaces: python and sodium. These encapsulate pure-Python implementations and shared/dynamic library (*i.e.*, libsodium) wrappers, respectively, of all operations and classes available in the ristretto module. This makes it possible to explicitly choose whether an operation requires only Python or also requires the presence of a compiled copy of libsodium on the host system.

The example below uses pure-Python implementations of the scalar multiplication operation (relying on the ge25519 library):

```
>>> from oblivious.ristretto import python
>>> p = python.point.hash('abc'.encode())
>>> s = python.scalar.hash('123'.encode())
>>> (s * p).to_base64()
'SrC7vA9sSR5f4E27ALxk14MPotTYR6B33B4ZN+mQXFA='
```

To check whether an instance of the libsodium shared/dynamic library has been loaded successfully, the check below can be performed:

```
>>> from oblivious.ristretto import sodium
>>> sodium is not None # Was the dynamic/shared library loaded?
True
```

In the example below, the scalar multiplication operation invokes a binding for the crypto_scalarmult_ristretto255 function exported by libsodium:

```
>>> p = sodium.point.hash('abc'.encode())
>>> s = sodium.scalar.hash('123'.encode())
>>> (s * p).to_base64()
'SrC7vA9sSR5f4E27ALxk14MPotTYR6B33B4ZN+mQXFA='
```

The class methods exported by the `ristretto` module directly (*e.g.*, the method `__add__` within the class `point` that is imported via the statement `from oblivious.ristretto import point`) correspond either (A) to libsodium wrappers if an instance of libsodium is found and loaded or (B) to pure-Python implementations if all attempts to load a working instances of libsodium fail. The ordered list below summarizes what definitions are exported under various conditions and the ordered sequence of attempts to locate and load an instance of libsodium.

1. Under all conditions, the wrapper class `python` is defined and encapsulates a pure-Python variant of every low-level operation and class available in the `ristretto` module. **As a starting default**, all classes exported directly by the `ristretto` module correspond to the pure-Python implementations.
2. If a shared/dynamic library instance of libsodium is found on the system and successfully loaded during one of the attempts below, then the wrapper class `sodium` is defined:
    a. the built-in `ctypes.util.find_library` function is able to locate `'sodium'` or `'libsodium'` and it is loaded successfully;
    b. a file `libsodium.so` or `libsodium.dll` under the paths specified by the `PATH` and `LD_LIBRARY_PATH` environment variables is found and loaded successfully; or
    c. the optional rbcl package is installed and the compiled subset of libsodium included in that package is loaded successfully.
3. If `sodium` is **not** `None`, then the `sodium` class encapsulates libsodium wrappers for low-level operations and for every class exported by the `ristretto` module. Furthermore, **those classes exported directly by the library are redefined** to use the bindings available in the loaded instance of libsodium. The `python` class is still exported, as well, and all operations and class methods encapsulated within `python` remain as-is (*i.e.*, pure-Python implementations).

The classes within the `bn254` module (both those that are pure-Python implementations and those that are wrappers for functions in the mcl library) are organized in a similar manner. More information is available in the documentation for the `bn254` module.

CHAPTER

THREE

DEVELOPMENT

All installation and development dependencies are fully specified in `pyproject.toml`. The `project.optional-dependencies` object is used to specify optional requirements for various development tasks. This makes it possible to specify additional options (such as `docs`, `lint`, and so on) when performing installation using pip:

```
python -m pip install .[docs,lint]
```

3.1 Documentation

The documentation can be generated automatically from the source files using Sphinx:

```
python -m pip install .[docs]
cd docs
sphinx-apidoc -f -e -E --templatedir=_templates -o _source .. && make html
```

3.2 Testing and Conventions

All unit tests are executed and their coverage is measured when using pytest (see the `pyproject.toml` file for configuration details, and note that unit tests that require rbcl and/or mclbn256 are skipped if the corresponding optional package is not installed):

```
python -m pip install .[test]
python -m pytest
```

Concise unit tests are implemented with the help of fountains; new reference specifications for the tests in a given testing module can be generated by running that testing module directly:

```
python test/test_ristretto.py
python test/test_bn254.py
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/oblivious test/test_ristretto.py test/test_bn254.py
```

3.3 Contributions

In order to contribute to the source code, open an issue or submit a pull request on the GitHub page for this library.

3.4 Versioning

Beginning with version 0.1.0, the version number format for this library and the changes to the library associated with version number increments conform with Semantic Versioning 2.0.0.

3.5 Publishing

This library can be published as a package on PyPI by a package maintainer. First, install the dependencies required for packaging and publishing:

```
python -m pip install .[publish]
```

Ensure that the correct version number appears in `pyproject.toml`, and that any links in this README document to the Read the Docs documentation of this package (or its dependencies) have appropriate version numbers. Also ensure that the Read the Docs project for this library has an automation rule that activates and sets as the default all tagged versions. Create and push a tag for this version (replacing `?.?.?` with the version number):

```
git tag ?.?.?
git push origin ?.?.?
```

Remove any old build/distribution files. Then, package the source into a distribution archive:

```
rm -rf build dist src/*.egg-info
python -m build --sdist --wheel .
```

Finally, upload the package distribution archive to PyPI:

```
python -m twine upload dist/*
```

3.5.1 ristretto module

This module exports the classes *point* and *scalar* for representing points and scalars. It also exports the two wrapper classes/namespaces *python* and *sodium* that encapsulate pure-Python and shared/dynamic library variants of the above (respectively) and also include low-level operations that correspond more directly to the functions found in the underlying libraries.

- Under all conditions, the wrapper class *python* is defined and encapsulates a pure-Python variant of every class exported by this module as a whole. It also includes pure-Python variants of low-level operations that correspond to functions found in the underlying libraries.
- If a shared/dynamic library instance of the libsodium library is found on the system (and successfully loaded at the time this module is imported) or the optional rbcl package is installed, then the wrapper class *sodium* is defined. Otherwise, the exported variable `sodium` is assigned `None`.
- If a dynamic/shared library instance is loaded, all classes exported by this module correspond to the variants defined within *sodium*. Otherwise, they correspond to the variants defined within *python*.

For most users, the classes *point* and *scalar* should be sufficient. When using the low-level operations that correspond to a specific implementation (*e.g.*, *oblivious.ristretto.sodium.add*), users are responsible for ensuring that inputs have the type and/or representation appropriate for that operation.

class oblivious.ristretto.**point**(*bs: Optional[bytes] = None*)

Bases: *bytes*

Class for representing a point. Because this class is derived from *bytes*, it inherits methods such as `bytes.hex` and `bytes.fromhex`.

```
>>> len(point.random())
32
>>> p = point.hash('123'.encode())
>>> p.hex()
'047f39a6c6dd156531a25fa605f017d4bec13b0b6c42f0e9b641c8ee73359c5f'
>>> point.fromhex(p.hex()) == p
True
```

classmethod random() → *oblivious.ristretto.point*

Return random point object.

```
>>> len(point.random())
32
```

classmethod bytes(*bs:* bytes) → *oblivious.ristretto.point*

Return the point object obtained by transforming the supplied bytes-like object.

```
>>> p = point.bytes(hashlib.sha512('123'.encode()).digest())
>>> p.hex()
'047f39a6c6dd156531a25fa605f017d4bec13b0b6c42f0e9b641c8ee73359c5f'
```

classmethod hash(*bs:* bytes) → *oblivious.ristretto.point*

Return point object by hashing supplied bytes-like object.

```
>>> point.hash('123'.encode()).hex()
'047f39a6c6dd156531a25fa605f017d4bec13b0b6c42f0e9b641c8ee73359c5f'
```

classmethod base(*s:* oblivious.ristretto.scalar) → Optional[*oblivious.ristretto.point*]

Return base point multiplied by supplied scalar if the scalar is valid; otherwise, return `None`.

```
>>> point.base(scalar.hash('123'.encode())).hex()
'4c207a5377f3badf358914f20b505cd1e2a639672a9c240e5aff522e2446005'
```

Use of the scalar corresponding to the zero residue is permitted.

```
>>> p = point()
>>> point.base(scalar.from_int(0)) + p == p
True
```

classmethod from_bytes(*bs:* bytes) → *oblivious.ristretto.point*

Return the instance corresponding to the supplied bytes-like object.

```
>>> p = point.bytes(hashlib.sha512('123'.encode()).digest())
>>> p == point.from_bytes(p.to_bytes())
True
```

classmethod from_base64(*s: str*) → *oblivious.ristretto.point*
Construct an instance from its Base64 UTF-8 string representation.

```
>>> point.from_base64('hoVaKq3oIlxEndP2Nqv3Rdbmiu4iinZE6Iwo+kcKAik=').hex()
'86855a2aade8225c449dd3f636abf745d6e68aee228a7644e88c28fa470a0229'
```

canonical() → *oblivious.ristretto.point*
Normalize the representation of this instance into its canonical form.

```
>>> p = point.hash('123'.encode())
>>> p.canonical() == p
True
```

__mul__(*other: Any*) → NoReturn
A point cannot be a left-hand argument for a multiplication operation.

```
>>> point() * scalar()
Traceback (most recent call last):
  ...
TypeError: point must be on right-hand side of multiplication operator
```

__rmul__(*other: Any*) → NoReturn
This functionality is implemented exclusively in the method *scalar.__mul__*, as that method pre-empts this method when the second argument has the correct type (*i.e.*, it is a *scalar* instance). This method is included so that an exception can be raised if an incorrect argument is supplied.

```
>>> p = point.hash('123'.encode())
>>> 2 * p
Traceback (most recent call last):
  ...
TypeError: point can only be multiplied by a scalar
```

__add__(*other:* oblivious.ristretto.point) → Optional[*oblivious.ristretto.point*]
Return the sum of this instance and another point.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> (p + q).hex()
'7076739c9df665d416e68b9512f5513bf1d0181a2aacefdeb1b7244528a4dd77'
>>> p + (q - q) == p
True
```

__sub__(*other:* oblivious.ristretto.point) → Optional[*oblivious.ristretto.point*]
Return the result of subtracting another point from this instance.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> (p - q).hex()
'1a3199ca7debfe31a90171696d8bab91b99eb23a541b822a7061b09776e1046c'
>>> p - p == point.base(scalar.from_int(0))
True
```

__neg__() → *oblivious.ristretto.point*
Return the negation of this instance.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> ((p + q) + (-q)) == p
True
```

to_bytes() → *bytes*

Return the bytes-like object that represents this instance.

```
>>> p = point()
>>> p.to_bytes() == p
True
```

to_base64() → str

Return the Base64 UTF-8 string representation of this instance.

```
>>> p = point.from_base64('hoVaKq3oIlxEndP2Nqv3Rdbmiu4iinZE6Iwo+kcKAik=')
>>> p.to_base64()
'hoVaKq3oIlxEndP2Nqv3Rdbmiu4iinZE6Iwo+kcKAik='
```

class oblivious.ristretto.**scalar**(*bs: Optional[bytes] = None*)

Bases: *bytes*

Class for representing a scalar. Because this class is derived from *bytes*, it inherits methods such as bytes.hex and bytes.fromhex.

```
>>> len(scalar.random())
32
>>> s = scalar.hash('123'.encode())
>>> s.hex()
'a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27a03'
>>> scalar.fromhex(s.hex()) == s
True
```

classmethod random() → *oblivious.ristretto.scalar*

Return random non-zero scalar object.

```
>>> len(scalar.random())
32
```

classmethod bytes(*bs:* bytes) → Optional[*oblivious.ristretto.scalar*]

Return scalar object obtained by transforming supplied bytes-like object if it is possible to do; otherwise, return None.

```
>>> s = python.scl()
>>> t = scalar.bytes(s)
>>> s.hex() == t.hex()
True
```

classmethod hash(*bs:* bytes) → *oblivious.ristretto.scalar*

Return scalar object by hashing supplied bytes-like object.

```
>>> scalar.hash('123'.encode()).hex()
'a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27a03'
```

classmethod from_int(*i: int*) → *oblivious.ristretto.scalar*

Construct an instance from its integer (*i.e.*, residue) representation.

```
>>> p = point()
>>> zero = scalar.from_int(0)
>>> zero * p == p - p
True
>>> one = scalar.from_int(1)
>>> one * p == p
True
>>> two = scalar.from_int(2)
>>> two * p == p + p
True
```

Negative integers are supported (and automatically converted into their corresponding least nonnegative residues).

```
>>> q = point()
>>> p - p == scalar.from_int(0) * p
True
>>> q - p - p == q + (scalar.from_int(-2) * p)
True
```

classmethod from_bytes(*bs:* bytes) → Optional[*oblivious.ristretto.scalar*]

Return the instance corresponding to the supplied bytes-like object.

```
>>> s = python.scl()
>>> t = scalar.from_bytes(s)
>>> s.hex() == t.hex()
True
```

classmethod from_base64(*s: str*) → *oblivious.ristretto.scalar*

Construct an instance from its Base64 UTF-8 string representation.

```
>>> scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=').hex()
'312d0c9130f69153bec9f5d0386a95135eb45eebf130af5f1fed1c6ed15f2500'
```

__invert__() → *oblivious.ristretto.scalar*

Return the inverse of this instance (modulo `2**252 + 27742317777372353535851937790883648493`).

```
>>> s = scalar()
>>> p = point()
>>> ((~s) * (s * p)) == p
True
```

The scalar corresponding to the zero residue cannot be inverted.

```
>>> ~scalar.from_int(0)
Traceback (most recent call last):
  ...
ValueError: cannot invert scalar corresponding to zero
```

__mul__(*other: Union[*oblivious.ristretto.scalar, oblivious.ristretto.point*]*) →
Union[*oblivious.ristretto.scalar*, *oblivious.ristretto.point*]

Multiply the supplied scalar or point by this instance.

```
>>> p = point.hash('123'.encode())
>>> s = scalar.hash('456'.encode())
>>> (s * p).hex()
'f61b377aa86050aaa88c90f4a4a0f1e36b0000cf46f6a34232c2f1da7a799f16'
>>> p = point.from_base64('hoVaKq3oIlxEndP2Nqv3Rdbmiu4iinZE6Iwo+kcKAik=')
>>> s = scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=')
>>> (s * s).hex()
'd4aecf034f60edc5cb32cdd5a4be6d069959aa9fd133c51c9dcfd960ee865e0f'
>>> isinstance(s * s, scalar)
True
>>> (s * p).hex()
'2208082412921a67f42ea39974819 0d2b889228372509f2f2d9929813d074e1b'
>>> isinstance(s * p, point)
True
```

Multiplying any point or scalar by the scalar corresponding to the zero residue yields the point or scalar corresponding to zero.

```
>>> scalar.from_int(0) * point() == p - p
True
>>> scalar.from_int(0) * scalar() == scalar.from_int(0)
True
```

Any attempt to multiply a value or object of an incompatible type by this instance raises an exception.

```
>>> s * 2
Traceback (most recent call last):
  ...
TypeError: multiplication by a scalar is defined only for scalars and points
```

__rmul__(*other: Union[*oblivious.ristretto.scalar, oblivious.ristretto.point*]*)

A scalar cannot be on the right-hand side of a non-scalar.

```
>>> point() * scalar()
Traceback (most recent call last):
  ...
TypeError: point must be on right-hand side of multiplication operator
```

to_bytes() → *bytes*

Return the bytes-like object that represents this instance.

```
>>> s = scalar()
>>> s.to_bytes() == s
True
```

__int__() → int

Return the integer (*i.e.*, least nonnegative residue) representation of this instance.

```
>>> s = scalar()
>>> int(s * (~s))
1
```

to_int() → int

Return the integer (*i.e.*, least nonnegative residue) representation of this instance.

```
>>> s = scalar()
>>> (s * (~s)).to_int()
1
```

to_base64() → str
Return the Base64 UTF-8 string representation of this instance.

```
>>> s = scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=')
>>> s.to_base64()
'MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA='
```

class oblivious.ristretto.**python**
Bases: object

Wrapper class for pure-Python implementations of primitive operations.

This class encapsulates pure-Python variants of all low-level operations and of both classes exported by this module: *python.scl*, *python.rnd*, *python.inv*, *python.smu*, *python.pnt*, *python.bas*, *python.can*, *python.mul*, *python.add*, *python.sub*, *python.neg*, *python.point*, and *python.scalar*. For example, you can perform addition of points using the pure-Python point addition implementation.

```
>>> p = python.pnt()
>>> q = python.pnt()
>>> python.add(p, q) == python.add(q, p)
True
```

Pure-Python variants of the *python.point* and *python.scalar* classes always employ pure-Python implementations of operations when their methods are invoked.

```
>>> p = python.point()
>>> q = python.point()
>>> p + q == q + p
True
```

Nevertheless, all bytes-like objects, *point* objects, and *scalar* objects accepted and emitted by the various operations and class methods in *python* are compatible with those accepted and emitted by the operations and class methods in *sodium*.

static pnt(*h: Optional[bytes] = None*) → bytes
Return point from 64-byte vector (normally obtained via hashing).

```
>>> p = python.pnt(hashlib.sha512('123'.encode()).digest())
>>> p.hex()
'047f39a6c6dd156531a25fa605f017d4bec13b0b6c42f0e9b641c8ee73359c5f'
```

static bas(*s: bytes*) → bytes
Return base point multiplied by supplied scalar.

```
>>> python.bas(scalar.hash('123'.encode())).hex()
'4c207a5377f3badf358914f20b505cd1e2a6396720a9c240e5aff522e2446005'
```

static can(*p: bytes*) → bytes
Normalize the representation of a point into its canonical form.

```
>>> p = point.hash('123'.encode())
>>> python.can(p) == p
True
```

static mul(*s: bytes*, *p: bytes*) → bytes

Multiply the point by the supplied scalar and return the result.

```
>>> p = python.pnt(hashlib.sha512('123'.encode()).digest())
>>> s = python.scl(bytes.fromhex(
...     '35c141f1c2c43543de9d188805a210abca3cd39a1e986304991ceded42b11709'
... ))
>>> python.mul(s, p).hex()
'183a06e0fe6af5d7913afb40baefc4dd52ae718fee77a3a0af8777c89fe16210'
```

static add(*p: bytes*, *q: bytes*) → bytes

Return sum of the supplied points.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> python.add(p, q).hex()
'7076739c9df665d416e68b9512f5513bf1d0181a2aacefdeb1b7244528a4dd77'
```

static sub(*p: bytes*, *q: bytes*) → bytes

Return result of subtracting second point from first point.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> python.sub(p, q).hex()
'1a3199ca7debfe31a90171696d8bab91b99eb23a541b822a7061b09776e1046c'
```

static neg(*p: bytes*) → bytes

Return the additive inverse of a point.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> python.add(python.neg(p), python.add(p, q)) == q
True
```

static rnd() → bytes

Return random non-zero scalar.

```
>>> len(python.rnd())
32
```

classmethod scl(*s: Optional[bytes] = None*) → Optional[bytes]

Return supplied byte vector if it is a valid scalar; otherwise, return `None`. If no byte vector is supplied, return a random scalar.

```
>>> s = python.scl()
>>> t = python.scl(s)
>>> s == t
True
>>> python.scl(bytes([255] * 32)) is None
True
```

static inv(*s: bytes*) → bytes
Return the inverse of a scalar (modulo `2**252 + 27742317777372353535851937790883648493`).

```
>>> s = python.scl()
>>> p = python.pnt()
>>> python.mul(python.inv(s), python.mul(s, p)) == p
True
```

static smu(*s: bytes*, *t: bytes*) → bytes
Return scalar multiplied by another scalar.

```
>>> s = python.scl()
>>> t = python.scl()
>>> python.smu(s, t) == python.smu(t, s)
True
```

class point(*bs: Optional[bytes] = None*)
Bases: `bytes`

class scalar(*bs: Optional[bytes] = None*)
Bases: `bytes`

class oblivious.ristretto.**sodium**
Bases: `object`

Wrapper class for binary implementations of primitive operations.

When this module is imported, it makes a number of attempts to locate an instance of the shared/dynamic library file of the libsodium library on the host system. The sequence of attempts is listed below, in order.

1. It uses `ctypes.util.find_library` to look for `'sodium'` or `'libsodium'`.
2. It attempts to find a file `libsodium.so` or `libsodium.dll` in the paths specified by the `PATH` and `LD_LIBRARY_PATH` environment variables.
3. If the rbcl package is installed, it reverts to the compiled subset of libsodium included in that package.

If all of the above fail, then *sodium* is assigned the value `None` and all classes exported by this module default to their pure-Python variants (*i.e.*, those encapsulated within *python*). To confirm that a dynamic/shared library *has been found* when this module is imported, evaluate the expression `sodium is not None`.

If a shared/dynamic library file has been loaded successfully, this class encapsulates shared/dynamic library variants of both classes exported by this module and of all the underlying low-level operations: *sodium.scl*, *sodium.rnd*, *sodium.inv*, *sodium.smu*, *sodium.pnt*, *sodium.bas*, *sodium.can*, *sodium.mul*, *sodium.add*, *sodium.sub*, *sodium.neg*, *sodium.point*, and *sodium.scalar*. For example, you can perform addition of points using the point addition implementation found in the libsodium shared/dynamic library found on the host system.

```
>>> p = sodium.pnt()
>>> q = sodium.pnt()
>>> sodium.add(p, q) == sodium.add(q, p)
True
```

Methods found in the shared/dynamic library variants of the *point* and *scalar* classes are wrappers for the shared/dynamic library implementations of the underlying operations.

```
>>> p = sodium.point()
>>> q = sodium.point()
```

(continues on next page)

(continued from previous page)

```
>>> p + q == q + p
True
```

Nevertheless, all bytes-like objects, *point* objects, and *scalar* objects accepted and emitted by the various operations and class methods in *sodium* are compatible with those accepted and emitted by the operations and class methods in *python*.

static pnt(*h: Optional[bytes] = None*) → bytes

Construct a point from its 64-byte vector representation (normally obtained via hashing).

```
>>> p = sodium.pnt(hashlib.sha512('123'.encode()).digest())
>>> p.hex()
'047f39a6c6dd156531a25fa605f017d4bec13b0b6c42f0e9b641c8ee73359c5f'
```

static bas(*s: bytes*) → bytes

Return the base point multiplied by the supplied scalar.

```
>>> sodium.bas(scalar.hash('123'.encode())).hex()
'4c207a5377f3badf358914f20b505cd1e2a6396720a9c240e5aff522e2446005'
```

static can(*p: bytes*) → bytes

Normalize the representation of a point into its canonical form.

```
>>> p = point.hash('123'.encode())
>>> sodium.can(p) == p
True
```

static mul(*s: bytes*, *p: bytes*) → bytes

Multiply a point by a scalar and return the result.

```
>>> p = sodium.pnt(hashlib.sha512('123'.encode()).digest())
>>> s = sodium.scl(bytes.fromhex(
...     '35c141f1c2c43543de9d188805a210abca3cd39a1e986304991ceded42b11709'
... ))
>>> sodium.mul(s, p).hex()
'183a06e0fe6af5d7913afb40baefc4dd52ae718fee77a3a0af8777c89fe16210'
```

static add(*p: bytes*, *q: bytes*) → bytes

Return the sum of the supplied points.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> sodium.add(p, q).hex()
'7076739c9df665d416e68b9512f5513bf1d0181a2aacefdeb1b7244528a4dd77'
```

static sub(*p: bytes*, *q: bytes*) → bytes

Return the result of subtracting the right-hand point from the left-hand point.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> sodium.sub(p, q).hex()
'1a3199ca7debfe31a90171696d8bab91b99eb23a541b822a7061b09776e1046c'
```

static neg(*p: bytes*) → bytes

Return the additive inverse of a point.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> sodium.add(sodium.neg(p), sodium.add(p, q)) == q
True
```

static rnd() → bytes

Return random non-zero scalar.

```
>>> len(sodium.rnd())
32
```

classmethod scl(*s: Optional[bytes] = None*) → Optional[bytes]

Return supplied byte vector if it is a valid scalar; otherwise, return `None`. If no byte vector is supplied, return a random scalar.

```
>>> s = sodium.scl()
>>> t = sodium.scl(s)
>>> s == t
True
>>> sodium.scl(bytes([255] * 32)) is None
True
```

static inv(*s: bytes*) → bytes

Return the inverse of a scalar (modulo `2**252 + 27742317777372353535851937790883648493`).

```
>>> s = sodium.scl()
>>> p = sodium.pnt()
>>> sodium.mul(sodium.inv(s), sodium.mul(s, p)) == p
True
```

static smu(*s: bytes*, *t: bytes*) → bytes

Return the product of two scalars.

```
>>> s = sodium.scl()
>>> t = sodium.scl()
>>> sodium.smu(s, t) == sodium.smu(t, s)
True
```

class point(*bs: Optional[bytes] = None*)

Bases: `bytes`

class scalar(*bs: Optional[bytes] = None*)

Bases: `bytes`

3.5.2 bn254 module

This module exports the classes *point*, *scalar*, *point2*, and *scalar2* for representing points and scalars. It also exports the two wrapper classes/namespaces *python* and *mcl* that encapsulate pure-Python and shared/dynamic library variants of the above (respectively) and also include low-level operations that correspond more directly to the functions found in the underlying libraries.

- Under all conditions, the wrapper class *python* is defined and encapsulates a pure-Python variant of class exported by this module as a whole. It also includes pure-Python variants of low-level operations that correspond to functions found in the underlying libraries.

- If the mclbn256 package is installed (which includes a bundled copy of the mcl dynamic/shared libray), then the wrapper class *mcl* is defined. Otherwise, the exported variable `mcl` is assigned `None`.
- If the mclbn256 package is installed, all classes exported by this module correspond to the variants defined in *mcl*. Otherwise, they correspond to the variants defined in *python*.

For most users, the classes *point*, *scalar*, *point2*, and *scalar2* should be sufficient.

When using the classes within *python* and/or *mcl*, users should be aware that objects corresponding to one implementation (*e.g.*, instances of *oblivious.bn254.mcl.point*) are not compatible with instances corresponding to the other implementation (*e.g.*, the methods of the *oblivious.bn254.python.point* class). When using the primitive operations that correspond to a specific implementation (*e.g.*, *oblivious.bn254.mcl.add*), users are responsible for ensuring that inputs have the type and/or representation appropriate for that operation's internal implementation.

class oblivious.bn254.**point**(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: object

Wrapper class for a bytes-like object that corresponds to a point.

classmethod random() → *oblivious.bn254.point*

Return random point object.

```
>>> len(point.random())
96
```

classmethod bytes(*bs:* bytes) → *oblivious.bn254.point*

Return point object obtained by transforming supplied bytes-like object if it is possible to do so; otherwise, return `None`.

The bytes-like object need not be the binary representation of a point or its coordinate(s). For a strict deserialization from a bytes-like object, use *point.from_bytes*.

```
>>> p = point.bytes(hashlib.sha512('123'.encode()).digest())
>>> p.hex()[:64]
'6d68495eb4d539db4df70fd24d54fae37c9adf7dfd8dc705ccb8de8630e7cf22'
```

classmethod hash(*bs:* bytes) → *oblivious.bn254.point*

Return point object by hashing supplied bytes-like object.

```
>>> point.hash('123'.encode()).hex()[:64]
'825aa78af4c88d6de4abaebabf1a96f668956b92876cfb5d3a44829899cb480f'
```

classmethod base(*s:* oblivious.bn254.scalar) → *oblivious.bn254.point*

Return base point multiplied by supplied scalar if the scalar is valid.

```
>>> point.base(scalar.hash('123'.encode())).canonical().hex()[:64]
'2d66076815cda25556bab4a930244ac284412267e9345aceb98d71530308401a'
```

classmethod from_bytes(*bs:* bytes) → *oblivious.bn254.point*

Deserialize the supplied binary representation of an instance and return that instance.

```
>>> p = point.hash('123'.encode())
>>> bs = p.to_bytes()
>>> point.from_bytes(bs) == p
True
>>> type(bs) is bytes
True
```

classmethod fromhex(*s: str*) → *oblivious.bn254.point*
Construct an instance from its hexadecimal UTF-8 string representation.

```
>>> point.fromhex(
...     'b89ec91191915a72d4ec4434be7b438893975880b21720995c2b2458962c4e0a'
...     'd0efebb5c303e4d1f8461b44ec768c587eca8b0abc01d4cb0d878b076154940d'
...     '8effffffffffff158affffffffff39b9cdffffffff2ec6a2f5ffff7ff2a42b21'
... ).canonical().hex()[:64]
'b89ec91191915a72d4ec4434be7b438893975880b21720995c2b2458962c4e0a'
```

classmethod from_base64(*s: str*) → *oblivious.bn254.point*
Construct an instance from its Base64 UTF-8 string representation.

```
>>> point.from_base64(
...     'hQIYpQRHupyyfPFoEm8rfmKV6i6VUP7vmngQWpxS3AEJD29fKVMW39l2oDLB+Ece'
...     '5PqBuRzCyiRb8xYIelEII47///////8Viv//////ObnN/////y7GovX//3/ypCsh'
... ).canonical().hex()[:64]
'850218a50447ba9cb27cf168126f2b7e6295ea2e9550feef9a78105a9c52dc01'
```

canonical() → *oblivious.bn254.point*
Normalize the representation of this point into its canonical form and return the result. This takes the z-coordinate, which may not always be equal to 1, and multiplies all coordinates x, y, and z by z's multiplicative inverse. The resulting canonical representation is unique (*i.e.*, two points are equal if and only if their canonical forms are equal) and in the form (x/z, y/z, 1).

```
>>> a = point.hash('123'.encode())
>>> p = a + a + a + a
>>> p == p
True
>>> p.to_bytes() == p.to_bytes()
True
>>> p.to_bytes() == p.canonical().to_bytes() and p.__class__ != python.point
False
>>> p.canonical().to_bytes() == p.canonical().to_bytes()
True
>>> p.canonical().to_bytes() == p.canonical().canonical().to_bytes()
True
>>> point.from_bytes(p.to_bytes()) == p
True
>>> point.from_bytes(p.canonical().to_bytes()) == p
True
>>> point.from_bytes(p.to_bytes()) == point.from_bytes(p.canonical().to_bytes())
True
>>> type(p.canonical()) is point
True
```

__mul__(*other: Any*) → NoReturn
Use of this method is not permitted. A point cannot be a left-hand argument.

```
>>> point() * scalar()
Traceback (most recent call last):
  ...
TypeError: point must be on right-hand side of multiplication operator
```

__rmul__(*other: Any*) → NoReturn

This functionality is implemented exclusively in the method *scalar.__mul__*, as that method pre-empts this method when the second argument has the correct type (*i.e.*, it is a *scalar* instance). This method is included so that an exception can be raised if an incorrect argument is supplied.

```
>>> p = point.hash('123'.encode())
>>> 2 * p
Traceback (most recent call last):
  ...
TypeError: point can only be multiplied by a scalar
```

__add__(*other:* oblivious.bn254.point) → *oblivious.bn254.point*

Return sum of this point and another point.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> (p + q).canonical().hex()[:64]
'1ea48cab238fece46bd0c9fb562c859e318e17a8fb75517a4750d30ca79b911c'
```

__sub__(*other:* oblivious.bn254.point) → *oblivious.bn254.point*

Return the result of subtracting another point from this point.

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> (p - q).canonical().hex()[:64]
'a43a5ce1931b1300b62e5d7e1b0c691203bfd85fafd9585dc5e47a7e2acfea22'
```

__neg__() → *oblivious.bn254.point*

Return the negation (additive inverse) of this point

```
>>> p = point.hash('123'.encode())
>>> q = point.hash('456'.encode())
>>> (p + q).canonical().hex()[:64]
'1ea48cab238fece46bd0c9fb562c859e318e17a8fb75517a4750d30ca79b911c'
```

__matmul__(*other:* oblivious.bn254.point2) → *oblivious.bn254.scalar2*

Return the result of pairing another second-level point with this instance.

This method is only defined for the classes *oblivious.bn254.mcl.point* and *oblivious.bn254.mcl.point2* that are available when the mclbn256 **package is installed**. Otherwise, *oblivious.bn254.point* and *oblivious.bn254.point2* correspond to the pure-Python implementations of these classes (for which this method is not defined).

```
>>> p = point.hash('123'.encode())
>>> q = point2.base(scalar.from_int(456))
>>> z = (p @ q).hex()[700:]
>>> z == 'd01f7e038b05acc5519eeda026c4aa111eb12f3483f274c60e34e6ec7571435df707'
True
```

The pairing function is bilinear.

```
>>> p = point.random()
>>> q = point2.random()
>>> s = scalar.random()
>>> t = scalar.random()
```

(continues on next page)

(continued from previous page)

```
>>> -((~s) * (s * p)) - p == scalar.from_int(-2) * p
True
>>> (s * (t * p)) @ q == (s * p) @ (t * q)
True
```

Suppose there are two points: one multiplied by the scalar `s` and the other multiplied by the scalar `t`. Their equality can be determined by using a balancing point: `g**(~s * t)`. If the pairing of `t * x` with `g` is the same as the pairing with `s * y` and `g**(~s * t)`, then `x` equals `y`.

```
>>> x = y = p
>>> g = point2.base(scalar.from_int(1))
>>> b = point2.base(~s * t)
>>> (t * x) @ g == (s * y) @ b
True
```

__len__() → int

Return length (in bytes) of the binary representation of this instance.

```
>>> len(point())
96
```

__bytes__() → *bytes*

Serialize this instance and return its binary representation.

```
>>> p = point.hash('123'.encode())
>>> bs = bytes(p)
>>> point.from_bytes(bs) == p
True
>>> type(bs) is bytes
True
>>> len(bs)
96
```

to_bytes() → *bytes*

Serialize this instance and return its binary representation.

```
>>> p = point.hash('123'.encode())
>>> bs = p.to_bytes()
>>> point.from_bytes(bs) == p
True
>>> type(bs) is bytes
True
>>> len(bs)
96
```

hex() → str

Return a hexadecimal representation of this instance.

```
>>> p = point.hash('123'.encode())
>>> p.hex()[:64]
'825aa78af4c88d6de4abaebabf1a96f668956b92876cfb5d3a44829899cb480f'
```

to_base64() → str

Return the Base64 UTF-8 string representation of this instance.

```
>>> p = point.from_base64(
...     'hQIYpQRHupyyfPFoEm8rfmKV6i6VUP7vmngQWpxS3AEJD29fKVMW39l2oDLB+Ece'
...     '5PqBuRzCyiRb8xYIelEII47///////8Viv//////ObnN/////y7GovX//3/ypCsh'
... )
>>> p.to_base64()[-64:]
'5PqBuRzCyiRb8xYIelEII47///////8Viv//////ObnN/////y7GovX//3/ypCsh'
```

class oblivious.bn254.**scalar**(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: object

Class for representing a scalar.

classmethod random() → *oblivious.bn254.scalar*

Return random non-zero scalar object.

```
>>> len(scalar.random())
32
```

classmethod bytes(*bs:* bytes) → Optional[*oblivious.bn254.scalar*]

Return scalar object obtained by transforming supplied bytes-like object if it is possible to do so; otherwise, return None.

```
>>> s = scalar()
>>> t = scalar.bytes(bytes(s))
>>> s.hex() == t.hex()
True
```

classmethod hash(*bs:* bytes) → *oblivious.bn254.scalar*

Return scalar object by hashing supplied bytes-like object.

```
>>> scalar.hash('123'.encode()).hex()[:64]
'93d829354cb3592743174133104b5405ba6992b67bb219fbde3e394d70505913'
```

classmethod from_int(*i: int*) → *oblivious.bn254.scalar*

Construct an instance from its corresponding integer (*i.e.*, residue) representation.

The integer can be in the range from -16798108731015832284940804142231733909759579603404752749028378864165570215948 to 16798108731015832284940804142231733909759579603404752749028378864165570215948 (or a corresponding one in the range from -8399054365507916142470402071115866954879789801702376374514189432082785107974 to 8399054365507916142470402071115866954879789801702376374514189432082785107974). Any values outside of this range will not be reduced, may be truncated, or may raise a ValueError. Zero-valued scalars are technically allowed, but cannot be used for point-scalar multiplication.

```
>>> int(scalar.from_int(
...     -
↪16798108731015832284940804142231733909759579603404752749028378864165570215948
... ))
-1
>>> int(scalar.from_int(
...     -
↪8399054365507916142470402071115866954879789801702376374514189432082785107974
... ))
-8399054365507916142470402071115866954879789801702376374514189432082785107974
>>> int(scalar.from_int(
...     12345678
```

(continues on next page)

(continued from previous page)

```
... ))
12345678
```

classmethod from_bytes(*bs:* bytes) → *oblivious.bn254.scalar*

Deserialize the supplied binary representation of an instance and return that instance.

```
>>> s = scalar.hash('123'.encode())
>>> bs = s.to_bytes()
>>> scalar.from_bytes(bs) == s
True
>>> type(bs) is bytes
True
```

classmethod fromhex(*s: str*) → *oblivious.bn254.scalar*

Construct an instance from its hexadecimal UTF-8 string representation.

```
>>> scalar.fromhex(
...     '3ab45f5b1c9339f1d25b878cce1c053b5492b4dc1affe689cbe141769f655e1e'
... ).hex()[:64]
'3ab45f5b1c9339f1d25b878cce1c053b5492b4dc1affe689cbe141769f655e1e'
```

classmethod from_base64(*s: str*) → *oblivious.bn254.scalar*

Construct an instance from its Base64 UTF-8 string representation.

```
>>> scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=').
↪hex()[:64]
'312d0c9130f69153bec9f5d0386a95135eb45eebf130af5f1fed1c6ed15f2500'
```

__invert__() → *oblivious.bn254.scalar*

Return the inverse of a scalar.

```
>>> s = scalar()
>>> p = point()
>>> ((~s) * (s * p)) == p
True
```

__mul__(*other: Union[*oblivious.bn254.scalar, oblivious.bn254.point, oblivious.bn254.point2*]*) → Optional[Union[*oblivious.bn254.scalar*, *oblivious.bn254.point*, *oblivious.bn254.point2*]]

Multiply supplied scalar, point, or second-level point by this instance.

```
>>> s = scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=')
>>> (s * s).hex()[:64]
'0497a5b6a7992e7d77b59c07d4457e8bb3cf580603cfd19e05d1f31342141b00'
>>> isinstance(s * s, scalar)
True
>>> p = point.from_base64(
...     'hQIYpQRHupyyfPFoEm8rfmKV6i6VUP7vmngQWpxS3AEJD29fKVMW39l2oDLB+Ece'
...     '5PqBuRzCyiRb8xYIelEII47///////8Viv//////ObnN/////y7GovX//3/ypCsh'
... )
>>> (s * p).canonical().hex()[:64]
'eee31d1780ea41771357da19a81eaddf2e7fa560142067b433764cbf98be9002'
>>> isinstance(s * p, point)
True
```

If the second argument is a *point2* object, this method pre-empts *point2.__rmul__*.

```
>>> p = point2.hash('123'.encode())
>>> (s * p).canonical().hex()[:128] == (
...     '451f144e06deecbfe5a1527f2b5cc6f12bbde91c1fdf0d5326ad79ffc53bb106'
...     '6d800275af625de83d72d815335832027cc60c34f22e8c5f89f953740a409702'
... )
True
```

Any attempt to multiply a value or object of an incompatible type by this instance raises an exception.

```
>>> s * 2
Traceback (most recent call last):
  ...
TypeError: multiplication by a scalar is defined only for scalars and points
```

__rmul__(*other: Any*) → NoReturn

A scalar cannot be on the right-hand side of a non-scalar.

```
>>> point() * scalar()
Traceback (most recent call last):
  ...
TypeError: point must be on right-hand side of multiplication operator
```

__add__(*other:* oblivious.bn254.scalar) → *oblivious.bn254.scalar*

Add another scalar to this instance.

```
>>> s = scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=')
>>> (s + s).hex()[:64]
'625a182261ec23a77c93eba171d42a27bc68bdd6e3615ebf3eda39dca2bf4a00'
>>> isinstance(s + s, scalar)
True
```

```
>>> z = scalar2.from_base64(
...     'GNDgZXmP+k7MoKfMbiuNbTJp9+tBNSXlm3MTMrAuqAVLkMic6T5EUlV/U6rl+PEy'
...     'IYWN3i6mpQNz8YWPEShwIQmz2veiF6IKPYcmMvjO1kPTJkmyQaZ2Ab7IVb1D8HcQ'
...     'iN9yK8rMj9F08WrX7xsdXmItk7fP7GOFw1PBN5Ok4Bw3HvVM2DaojhsfHhmNVm07'
...     'vlqiJXNDBfFMrJr5yCE1HIazZRkBBdG35xAR0TYu1q2RM6LCxCiY6evD0WBKYI8Q'
...     'hGHTeyRjtPiPDMtCe17qhxTuiPjDPa9+5KP0XKi8qRGHlWq7TSfeG0nl4Fn5BPu0'
...     'dYBHFasloXMG+g4ZbzBSFTpg36BzynjeSe3qrJxUyrPQE4dQwjwLaN55JKadG6AC'
...     'okrFkmIqRcWcHe1xM3lCkqCWAr1Xo2YB01Q4hG/LNw85wPp6FbNNFKtvle5b9bJr'
...     '1d7x5+0HNQki1EXaB9k2Ii21uqnewVLCsrz8Rs3m/SLnAnGvihZOd+WAjOYCCVof'
... )
>>> (z + z).hex()[700:]
'4a1f476a7553bd83a5dd5179f98d9acddae4c505e25e95df6734c901198d83ad9019'
>>> isinstance(z + z, scalar2)
True
```

__sub__(*other:* oblivious.bn254.scalar) → *oblivious.bn254.scalar*

Subtract this instance from another scalar.

```
>>> s = scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=')
>>> (s - s).hex() == '00' * len(s)
True
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(s - s, scalar)
True
```

__neg__() → *oblivious.bn254.scalar*
Negate this instance.

```
>>> s = scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=')
>>> (-s).hex()
'dcd2f36ecf096e4d52360a2fc7150aeca94ba1148e1c855ae212e3d1b004fe24'
>>> isinstance(-s, scalar)
True
```

__len__() → int
Return length (in bytes) of the binary representation of this instance.

```
>>> len(scalar())
32
```

__bytes__() → *bytes*
Serialize this instance and return its binary representation.

```
>>> s = scalar.hash('123'.encode())
>>> bs = bytes(s)
>>> scalar.from_bytes(bs) == s
True
>>> type(bs) is bytes
True
```

to_bytes() → *bytes*
Serialize this instance and return its binary representation.

```
>>> s = scalar.hash('123'.encode())
>>> bs = s.to_bytes()
>>> scalar.from_bytes(bs) == s
True
>>> type(bs) is bytes
True
```

__int__() → *bytes*
Compute and return the numerical representation of this instance.

```
>>> s = scalar.from_int(123)
>>> n = int(s)
>>> scalar.from_int(n) == s
True
>>> type(n) is int
True
```

to_int() → *bytes*
Compute and return the numerical representation of this instance.

```
>>> s = scalar.from_int(123)
>>> n = s.to_int()
```

(continues on next page)

(continued from previous page)

```
>>> scalar.from_int(n) == s
True
>>> type(n) is int
True
```

hex() → str

Return a hexadecimal representation of this instance.

```
>>> s = scalar.hash('123'.encode())
>>> s.hex()
'93d829354cb3592743174133104b5405ba6992b67bb219fbde3e394d70505913'
```

to_base64() → str

Return the Base64 UTF-8 string representation of this instance.

```
>>> s = scalar.from_base64('MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA=')
>>> s.to_base64()
'MS0MkTD2kVO+yfXQOGqVE160XuvxMK9fH+0cbtFfJQA='
```

class oblivious.bn254.**point2**(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: object

Wrapper class for a bytes-like object that corresponds to a point.

classmethod random() → *oblivious.bn254.point2*

Return random instance.

```
>>> len(point2.random())
192
```

classmethod bytes(*bs: Union[*bytes, *bytearray]*) → *oblivious.bn254.point2*

Construct an instance that corresponds to the supplied binary representation.

```
>>> p = point2.bytes(hashlib.sha512('123'.encode()).digest())
>>> p.canonical().hex()[:128] == (
...     '4c595542640a69c4a70bda55c27ef96c133cd1f4a5f83b3371e571960c018e19'
...     'c54aaec2069f8f10a00f12bcbb3511cdb7356201f5277ec5e47da91405be2809'
... )
True
```

classmethod hash(*bs: Union[*bytes, *bytearray]*) → *oblivious.bn254.point2*

Construct an instance by hashing the supplied bytes-like object.

```
>>> point2.hash('123'.encode()).canonical().hex()[:128] == (
...     '30326199f303fce7a77cff6d2fb0b3de8cd409d1d562f3543f7d064cdc58d309'
...     '7e88038ad76e85e5df26e4a9486a657b0431c8e7e09b0a1abf90fc874c515207'
... )
True
```

classmethod base(*s:* oblivious.bn254.scalar) → *oblivious.bn254.point2*

Return base second-level point multiplied by the supplied scalar if the scalar is valid; otherwise, return None.

```
>>> point2.base(scalar.hash('123'.encode())).canonical().hex()[:128] == (
...     'e7000fb12d206112c73fe1054e9d77b35c77881eba6598b7e035171d90b13e0c'
...     '33c8ad2c92acb446fa958f3001b6c15aaf0f00092534a9d567541f9fadc64e09'
... )
True
```

classmethod from_bytes(*bs:* bytes) → *oblivious.bn254.point2*

Deserialize the supplied binary representation of an instance and return that instance.

```
>>> p = point2.hash('123'.encode())
>>> bs = p.to_bytes()
>>> point2.from_bytes(bs) == p
True
>>> type(bs) is bytes
True
```

classmethod fromhex(*s: str*) → *oblivious.bn254.point2*

Construct an instance from its hexadecimal UTF-8 string representation.

```
>>> p = point2.fromhex(
...     'ab4efa2bcdeb825a67b12a10132ae1addca840ed248f83ae7dd987370dd47a05'
...     '31c10b08ada0e24c0327d85b108e826a55bf3dc3286488327fac75e05e293b20'
...     '01cbf919b53884d02b85aab9b0091eeda114fa65ca5d75620da26c4d164aa509'
...     '2a2d55b6f311bfe52d24adf7b4b0b6ce12ed486a37c474d35a2b373be8a3f71c'
...     '8effffffffffff158affffffffff39b9cdffffffff2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
>>> p.canonical().hex()[:64]
'ab4efa2bcdeb825a67b12a10132ae1addca840ed248f83ae7dd987370dd47a05'
```

classmethod from_base64(*s: str*) → *oblivious.bn254.point2*

Construct an instance from its Base64 UTF-8 string representation.

```
>>> p = point2.from_base64(
...     'xRuTJv/OWkIPMxRoCQIqNYoSixnWfMxeYwSJnjdJwxlp9E9f6oKefvbfYlJeygmK'
...     'YDQniir3r/EYExFuClZ7H5X00GEqz7TcoqDl5EpwLDAvrTW3GNA2lOpHvc1F/eQc'
...     'obJoTn35OzzK7qd/87Y3pOKNQaNENKO19DMzw9Lt+hiO////////FYr//////zm5'
...     'zf////8uxqL1//9/8qQrIQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
... )
>>> p.to_base64()[-64:]
'zf////8uxqL1//9/8qQrIQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

canonical() → *oblivious.bn254.point2*

Normalize the representation of this point into its canonical form and return the result. This takes the z-coordinate, which may not always be equal to 1, and multiplies all coordinates x, y, and z by z's multiplicative inverse. The resulting canonical representation is unique (*i.e.*, two second-level points are equal if and only if their canonical forms are equal) and in the form (*x1/z1*, *y1/z1*, *x2/z2*, *y2/z2*, 1, 0).

```
>>> a = point2.hash('123'.encode())
>>> q = a + a + a + a
>>> q == q
True
>>> q.to_bytes() == q.to_bytes()
```

(continues on next page)

(continued from previous page)

```
True
>>> q.to_bytes() == q.canonical().to_bytes() and q.__class__ != python.point2
False
>>> q.canonical().to_bytes() == q.canonical().to_bytes()
True
>>> q.canonical().to_bytes() == q.canonical().canonical().to_bytes()
True
>>> point2.from_bytes(q.to_bytes()) == q
True
>>> point2.from_bytes(q.canonical().to_bytes()) == q
True
>>> point2.from_bytes(q.to_bytes()) == point2.from_bytes(bytes(q.canonical()))
True
>>> type(q.canonical()) is point2
True
```

__mul__(*other: Any*) → NoReturn

Use of this method is not permitted. A point cannot be a left-hand argument.

```
>>> point2() * scalar()
Traceback (most recent call last):
  ...
TypeError: second-level point must be on right-hand side of multiplication␣
↪operator
```

__rmul__(*other: Any*) → NoReturn

This functionality is implemented exclusively in the method *scalar.__mul__*, as that method pre-empts this method when the second argument has the correct type (*i.e.*, it is a *scalar* instance). This method is included so that an exception can be raised if an incorrect argument is supplied.

```
>>> p = point2.hash('123'.encode())
>>> 2 * p
Traceback (most recent call last):
  ...
TypeError: second-level point can only be multiplied by a scalar
```

__add__(*other:* oblivious.bn254.point2) → Optional[*oblivious.bn254.point2*]

Return sum of this instance and another second-level point.

```
>>> p = point2.hash('123'.encode())
>>> q = point2.hash('456'.encode())
>>> (p + q).canonical().hex()[:128] == (
...     'cb0fc423c1bac2ac2df47bf5f5548a42b0d0a0da325bc77243d15dc587a7b221'
...     '9808a1649991ddf770f0060333aab4d499580b123f109b5cb180f1f8a75a090e'
... )
True
```

__sub__(*other:* oblivious.bn254.point2) → Optional[*oblivious.bn254.point2*]

Return the result of subtracting another second-level point from this instance.

```
>>> p = point2.hash('123'.encode())
>>> q = point2.hash('456'.encode())
>>> (p - q).canonical().hex()[:128] == (
```

(continues on next page)

(continued from previous page)

```
...     'e97a70c4e3a5369ebbb1dcf0cc1135c8c8e04a4ec7cffdf875ac429d66846d0b'
...     '191b090909c40a723027b07ac44435a6ade3813d04b3632a17c92c5c98718902'
... )
True
```

__neg__() → *oblivious.bn254.point2*

Return the negation (additive inverse) of this instance.

```
>>> p = point2.hash('123'.encode())
>>> (-p).canonical().hex()[:128] == (
...     '30326199f303fce7a77cff6d2fb0b3de8cd409d1d562f3543f7d064cdc58d309'
...     '7e88038ad76e85e5df26e4a9486a657b0431c8e7e09b0a1abf90fc874c515207'
... )
True
```

__matmul__(*other:* oblivious.bn254.point) → *oblivious.bn254.scalar2*

Return the result of pairing another point with this instance.

Input-swapped alias of *point.__matmul__*.

__len__() → int

Return length (in bytes) of the binary representation of this instance.

```
>>> len(point2())
192
```

__bytes__() → *bytes*

Serialize this instance and return its binary representation.

```
>>> len(bytes(point2()))
192
```

to_bytes() → *bytes*

Serialize this instance and return its binary representation.

```
>>> p = point2.hash('123'.encode())
>>> bs = p.to_bytes()
>>> point2.from_bytes(bs) == p
True
>>> type(bs) is bytes
True
```

hex() → str

Return a hexadecimal representation of this instance.

```
>>> p = point2.hash('123'.encode())
>>> p.canonical().hex() == (
...     '30326199f303fce7a77cff6d2fb0b3de8cd409d1d562f3543f7d064cdc58d309'
...     '7e88038ad76e85e5df26e4a9486a657b0431c8e7e09b0a1abf90fc874c515207'
...     '2c6a88bb448065eb748df632b1d872e02f54b6f56fdb84a7b1cb388fe551fb08'
...     '04464efa186bd4b1371e53d6f31f0e2f50ff553b6264a43331b42c976a0c541f'
...     '8efffffffffff158affffffffff39b9cdfffffffff2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
```

(continues on next page)

(continued from previous page)

```
... )
True
```

to_base64() → str

Return the Base64 UTF-8 string representation of this instance.

```
>>> p = point2.from_base64(
...     'zn07zy59PMhe396h9AQ+FY3LqfzmaRmbVmfwKaQqTxStH2ZPqGwBjv99STlWrenq'
...     'Mkfc3PCxRgM1xVaJGN+WExXhuDn4V40nkdpxtU85VFgE4aj0CMUoD99bqTEqBSYD'
...     '50haF1C7mDxMRxmMXZinYDEMynRY69C1vTQ5IgcCdh+O////////FYr//////zm5'
...     'zf////8uxqL1//9/8qQrIQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
... )
>>> p.to_base64()[-64:]
'zf////8uxqL1//9/8qQrIQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

class oblivious.bn254.**scalar2**(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: object

Class for representing second-level scalars.

classmethod random() → *oblivious.bn254.scalar2*

Return random non-zero second-level scalar.

```
>>> isinstance(scalar2.random(), scalar2)
True
>>> len(scalar2.random())
384
```

classmethod bytes(*bs:* bytes) → Optional[*oblivious.bn254.scalar2*]

Return second-level scalar object obtained by transforming the supplied bytes-like object if it is possible to do so; otherwise, return None.

```
>>> s = scalar2()
>>> t = scalar2.bytes(bytes(s))
>>> s.hex() == t.hex()
True
```

classmethod hash(*bs: Union[*bytes, *bytearray]*) → *oblivious.bn254.scalar2*

Return an instance derived by hashing the supplied bytes-like object.

```
>>> s = python.scalar2.hash(bytes([123]))
>>> s.hex()[700:]
'e91ed56ea67d29047d588ffaf78f9ed317ff13e7f63e53276ff32988c49184e17b22'
```

```
>>> s0 = python.scalar2.hash(secrets.token_bytes(64))
>>> s1 = python.scalar2.hash(secrets.token_bytes(64))
>>> python.sse(python.smu2(python.smu2(s0, python.inv2(s0)), s1)) == python.
↪sse(s1)
True
```

classmethod from_bytes(*bs:* bytes) → *oblivious.bn254.scalar2*

Deserialize the supplied binary representation of an instance and return that instance.

```
>>> s = scalar2.hash('123'.encode())
>>> bs = s.to_bytes()
>>> scalar2.from_bytes(bs) == s
True
>>> type(bs) is bytes
True
```

classmethod fromhex(*s: str*) → *oblivious.bn254.scalar2*

Construct an instance from its hexadecimal UTF-8 string representation.

```
>>> s_hex = (
...     '18d0e065798ffa4ecca0a7cc6e2b8d6d3269f7eb413525e59b731332b02ea805'
...     '4b90c89ce93e4452557f53aae5f8f13221858dde2ea6a50373f1858f11287021'
...     '09b3daf7a217a20a3d872632f8ced643d32649b241a67601bec855bd43f07710'
...     '88df722bcacc8fd174f16ad7ef1b1d5e622d93b7cfec6385c353c13793a4e01c'
...     '371ef54cd836a88e1b1f1e198d566d3bbe5aa225734305f14cac9af9c821351c'
...     '86b365190105d1b7e71011d1362ed6ad9133a2c2c42898e9ebc3d1604a608f10'
...     '8461d37b2463b4f88f0ccb427b5eea8714ee88f8c33daf7ee4a3f45ca8bca911'
...     '87956abb4d27de1b49e5e059f904fbb475804715ab25a17306fa0e196f305215'
...     '3a60dfa073ca78de49edeaac9c54cab3d0138750c23c0b68de7924a69d1ba002'
...     'a24ac592622a45c59c1ded7133794292a09602bd57a36601d35438846fcb370f'
...     '39c0fa7a15b34d14ab6f95ee5bf5b26bd5def1e7ed07350922d445da07d93622'
...     '2db5baa9dec152c2b2bcfc46cde6fd22e70271af8a164e77e5808ce602095a1f'
... )
>>> s = scalar2.fromhex(s_hex)
>>> s.hex() == s_hex
True
```

classmethod from_base64(*s: str*) → *oblivious.bn254.scalar2*

Construct an instance from its Base64 UTF-8 string representation.

```
>>> b64s = (
...     'GNDgZXmP+k7MoKfMbiuNbTJp9+tBNSXlm3MTMrAuqAVLkMic6T5EUlV/U6rl+PEy'
...     'IYWN3i6mpQNz8YWPEShwIQmz2veiF6IKPYcmMvjO1kPTJkmyQaZ2Ab7IVb1D8HcQ'
...     'iN9yK8rMj9F08WrX7xsdXmItk7fP7GOFw1PBN5Ok4Bw3HvVM2DaojhsfHhmNVm07'
...     'vlqiJXNDBfFMrJr5yCE1HIazZRkBBdG35xAR0TYu1q2RM6LCxCiY6evD0WBKYI8Q'
...     'hGHTeyRjtPiPDMtCe17qhxTuiPjDPa9+5KP0XKi8qRGHlWq7TSfeG0nl4Fn5BPu0'
...     'dYBHFasloXMG+g4ZbzBSFTpg36BzynjeSe3qrJxUyrPQE4dQwjwLaN55JKadG6AC'
...     'okrFkmIqRcWcHe1xM3lCkqCWAr1Xo2YB01Q4hG/LNw85wPp6FbNNFKtvle5b9bJr'
...     '1d7x5+0HNQki1EXaB9k2Ii21uqnewVLCsrz8Rs3m/SLnAnGvihZOd+WAjOYCCVof'
... )
>>> s = scalar2.from_base64(b64s)
>>> s.to_base64() == b64s
True
```

__invert__() → *oblivious.bn254.scalar2*

Return the inverse of this instance.

```
>>> s = scalar2.hash('123'.encode())
>>> ~(~s) == s
True
>>> ~s == s
False
```

(continues on next page)

(continued from previous page)

```
>>> bytes(~s).hex()[700:] == (
...     'ff13804852ea3ad35e8316d90a6d5dde854517e74cfc27ba676f429eb4fd52cd9b0c'
... )
True
```

__mul__(*other:* oblivious.bn254.scalar2) → *oblivious.bn254.scalar2*

Multiply this instance by another second-level scalar.

```
>>> s = scalar2.from_base64(
...     'GNDgZXmP+k7MoKfMbiuNbTJp9+tBNSXlm3MTMrAuqAVLkMic6T5EUlV/U6rl+PEy'
...     'IYWN3i6mpQNz8YWPEShwIQmz2veiF6IKPYcmMvjO1kPTJkmyQaZ2Ab7IVb1D8HcQ'
...     'iN9yK8rMj9F08WrX7xsdXmItk7fP7GOFw1PBN5Ok4Bw3HvVM2DaojhsfHhmNVm07'
...     'vlqiJXNDBfFMrJr5yCE1HIazZRkBBdG35xAR0TYu1q2RM6LCxCiY6evD0WBKYI8Q'
...     'hGHTeyRjtPiPDMtCe17qhxTuiPjDPa9+5KP0XKi8qRGHlWq7TSfeG0nl4Fn5BPu0'
...     'dYBHFasloXMG+g4ZbzBSFTpg36BzynjeSe3qrJxUyrPQE4dQwjwLaN55JKadG6AC'
...     'okrFkmIqRcWcHe1xM3lCkqCWAr1Xo2YB01Q4hG/LNw85wPp6FbNNFKtvle5b9bJr'
...     '1d7x5+0HNQki1EXaB9k2Ii21uqnewVLCsrz8Rs3m/SLnAnGvihZOd+WAjOYCCVof'
... )
>>> bytes(s * s).hex()[700:]
'6f11685b89b03431dac6dc9d129c6a31cc5e3036f7f781d7460ab9f532a06845bd15'
>>> scalar2() * point()
Traceback (most recent call last):
  ...
TypeError: second-level scalar can only be multiplied by another second-level␣
↪scalar
```

__rmul__(*other: Any*) → NoReturn

A second-level scalar cannot be on the right-hand side of a non-scalar.

```
>>> 2 * scalar2()
Traceback (most recent call last):
  ...
TypeError: second-level scalar must be on left-hand side of multiplication␣
↪operator
```

__add__(*other:* oblivious.bn254.scalar2) → *oblivious.bn254.scalar2*

Add another second-level scalar to this instance.

```
>>> z = scalar2.from_base64(
...     'GNDgZXmP+k7MoKfMbiuNbTJp9+tBNSXlm3MTMrAuqAVLkMic6T5EUlV/U6rl+PEy'
...     'IYWN3i6mpQNz8YWPEShwIQmz2veiF6IKPYcmMvjO1kPTJkmyQaZ2Ab7IVb1D8HcQ'
...     'iN9yK8rMj9F08WrX7xsdXmItk7fP7GOFw1PBN5Ok4Bw3HvVM2DaojhsfHhmNVm07'
...     'vlqiJXNDBfFMrJr5yCE1HIazZRkBBdG35xAR0TYu1q2RM6LCxCiY6evD0WBKYI8Q'
...     'hGHTeyRjtPiPDMtCe17qhxTuiPjDPa9+5KP0XKi8qRGHlWq7TSfeG0nl4Fn5BPu0'
...     'dYBHFasloXMG+g4ZbzBSFTpg36BzynjeSe3qrJxUyrPQE4dQwjwLaN55JKadG6AC'
...     'okrFkmIqRcWcHe1xM3lCkqCWAr1Xo2YB01Q4hG/LNw85wPp6FbNNFKtvle5b9bJr'
...     '1d7x5+0HNQki1EXaB9k2Ii21uqnewVLCsrz8Rs3m/SLnAnGvihZOd+WAjOYCCVof'
... )
>>> (z + z).hex()[700:]
'4a1f476a7553bd83a5dd5179f98d9acddae4c505e25e95df6734c901198d83ad9019'
>>> isinstance(z + z, scalar2)
True
```

__len__() → int
Return length (in bytes) of the binary representation of this instance.

```
>>> len(scalar2.random())
384
```

__bytes__() → *bytes*
Serialize this instance and return its binary representation.

```
>>> s = scalar2.hash('123'.encode())
>>> bs = bytes(s)
>>> scalar2.from_bytes(bs) == s
True
>>> type(bs) is bytes
True
>>> len(bs)
384
```

to_bytes() → *bytes*
Serialize this instance and return its binary representation.

```
>>> s = scalar2.hash('123'.encode())
>>> bs = s.to_bytes()
>>> scalar2.from_bytes(bs) == s
True
>>> type(bs) is bytes
True
>>> len(bs)
384
```

hex() → str
Return a hexadecimal representation of this instance.

```
>>> s = scalar2.from_base64(
...     'GNDgZXmP+k7MoKfMbiuNbTJp9+tBNSXlm3MTMrAuqAVLkMic6T5EUlV/U6rl+PEy'
...     'IYWN3i6mpQNz8YWPEShwIQmz2veiF6IKPYcmMvjO1kPTJkmyQaZ2Ab7IVb1D8HcQ'
...     'iN9yK8rMj9F08WrX7xsdXmItk7fP7GOFw1PBN5Ok4Bw3HvVM2DaojhsfHhmNVm07'
...     'vlqiJXNDBfFMrJr5yCE1HIazZRkBBdG35xAR0TYu1q2RM6LCxCiY6evD0WBKYI8Q'
...     'hGHTeyRjtPiPDMtCe17qhxTuiPjDPa9+5KP0XKi8qRGHlWq7TSfeG0nl4Fn5BPu0'
...     'dYBHFasloXMG+g4ZbzBSFTpg36BzynjeSe3qrJxUyrPQE4dQwjwLaN55JKadG6AC'
...     'okrFkmIqRcWcHe1xM3lCkqCWAr1Xo2YB01Q4hG/LNw85wPp6FbNNFKtvle5b9bJr'
...     '1d7x5+0HNQki1EXaB9k2Ii21uqnewVLCsrz8Rs3m/SLnAnGvihZOd+WAjOYCCVof'
... )
>>> s.hex() == (
...     '18d0e065798ffa4ecca0a7cc6e2b8d6d3269f7eb413525e59b731332b02ea805'
...     '4b90c89ce93e4452557f53aae5f8f13221858dde2ea6a50373f1858f11287021'
...     '09b3daf7a217a20a3d872632f8ced643d32649b241a67601bec855bd43f07710'
...     '88df722bcacc8fd174f16ad7ef1b1d5e622d93b7cfec6385c353c13793a4e01c'
...     '371ef54cd836a88e1b1f1e198d566d3bbe5aa225734305f14cac9af9c821351c'
...     '86b365190105d1b7e71011d1362ed6ad9133a2c2c42898e9ebc3d1604a608f10'
...     '8461d37b2463b4f88f0ccb427b5eea8714ee88f8c33daf7ee4a3f45ca8bca911'
...     '87956abb4d27de1b49e5e059f904fbb475804715ab25a17306fa0e196f305215'
...     '3a60dfa073ca78de49edeaac9c54cab3d0138750c23c0b68de7924a69d1ba002'
...     'a24ac592622a45c59c1ded7133794292a09602bd57a36601d35438846fcb370f'
```

(continues on next page)

(continued from previous page)

```
...     '39c0fa7a15b34d14ab6f95ee5bf5b26bd5def1e7ed07350922d445da07d93622'
...     '2db5baa9dec152c2b2bcfc46cde6fd22e70271af8a164e77e5808ce602095a1f'
... )
True
```

to_base64() → str

Return the Base64 UTF-8 string representation of this instance.

```
>>> b64s = (
...     'GNDgZXmP+k7MoKfMbiuNbTJp9+tBNSXlm3MTMrAuqAVLkMic6T5EUlV/U6rl+PEy'
...     'IYWN3i6mpQNz8YWPEShwIQmz2veiF6IKPYcmMvjO1kPTJkmyQaZ2Ab7IVb1D8HcQ'
...     'iN9yK8rMj9F08WrX7xsdXmItk7fP7GOFw1PBN5Ok4Bw3HvVM2DaojhsfHhmNVm07'
...     'vlqiJXNDBfFMrJr5yCE1HIazZRkBBdG35xAR0TYu1q2RM6LCxCiY6evD0WBKYI8Q'
...     'hGHTeyRjtPiPDMtCe17qhxTuiPjDPa9+5KP0XKi8qRGHlWq7TSfeG0nl4Fn5BPu0'
...     'dYBHFasloXMG+g4ZbzBSFTpg36BzynjeSe3qrJxUyrPQE4dQwjwLaN55JKadG6AC'
...     'okrFkmIqRcWcHe1xM3lCkqCWAr1Xo2YB01Q4hG/LNw85wPp6FbNNFKtvle5b9bJr'
...     '1d7x5+0HNQki1EXaB9k2Ii21uqnewVLCsrz8Rs3m/SLnAnGvihZOd+WAjOYCCVof'
... )
>>> s = scalar2.from_base64(b64s)
>>> s.to_base64() == b64s
True
```

class oblivious.bn254.**python**

Bases: object

Wrapper class for pure-Python implementations of primitive operations.

This class encapsulates pure-Python variants of all classes exported by this module and of all the underlying low-level operations: *python.pnt*, *python.bas*, *python.can*, *python.ser*, *python.des*, *python.mul*, *python.add*, *python.sub*, *python.neg*, python.par, *python.rnd*, *python.scl*, *python.sse*, *python.sde*, *python.inv*, *python.smu*, *python.sad*, *python.ssu*, *python.sne*, *python.pnt2*, *python.bas2*, *python.can2*, *python.ser2*, *python.des2*, *python.mul2*, *python.add2*, *python.sub2*, *python.neg2*, *python.rnd2*, *python.scl2*, *python.sse2*, *python.sde2*, *python.inv2*, *python.smu2*, *python.sad2*, *python.point*, *python.scalar*, *python.point2*, and *python.scalar2*. For example, you can perform multiplication of scalars using the pure-Python scalar multiplication implementation.

```
>>> s = python.scl()
>>> t = python.scl()
>>> python.smu(s, t) == python.smu(t, s)
True
```

Pure-Python variants of the *python.point* and *python.scalar* classes always employ pure Python implementations of operations when their methods are invoked.

```
>>> p = python.scalar()
>>> q = python.scalar()
>>> p * q == q * p
True
```

static pnt(*h: Optional[bytes] = None*) → *oblivious.bn254.point*

Construct a point from its 64-byte vector representation (normally obtained via hashing).

```
>>> p = python.pnt(hashlib.sha512('123'.encode()).digest())
>>> p.hex()[:64]
```

(continues on next page)

(continued from previous page)

```
'6d68495eb4d539db4df70fd24d54fae37c9adf7dfd8dc705ccb8de8630e7cf22'
```

static bas(*s:* oblivious.bn254.scalar) → *oblivious.bn254.point*

Return the base point multiplied by the supplied scalar.

```
>>> bytes(python.bas(python.scalar.hash('123'.encode()))).hex()[:64]
'2d66076815cda25556bab4a930244ac284412267e9345aceb98d71530308401a'
```

static can(*p:* oblivious.bn254.point) → *oblivious.bn254.point*

Normalize the representation of a point into its canonical form and return the result.

```
>>> a = python.point.hash('123'.encode())
>>> p = python.add(a, a)
>>> p_can = python.can(python.add(a, a))
```

It may be the case that `ser(p_can) != ser(p)`, depending on the implementation. It is the responsibility of the user to ensure that only canonical forms are serialized if those serialized forms must be compared.

```
>>> mclbn256 = p.__class__ != python.point
>>> (python.ser(p_can) != python.ser(p)) or not mclbn256
True
```

Normalization is idempotent.

```
>>> python.can(p) == python.can(p_can)
True
```

static ser(*p:* oblivious.bn254.point) → bytes

Return the binary representation of a point.

```
>>> q = python.point2.hash('123'.encode())
>>> python.des(python.ser(q)) == q
True
```

static des(*bs: bytes*) → *oblivious.bn254.point*

Construct a point corresponding to the supplied binary representation.

```
>>> p = python.point.hash('123'.encode())
>>> python.ser_p = bytes.fromhex(
...     '825aa78af4c88d6de4abaebabf1a96f668956b92876cfb5d3a44829899cb480f'
...     'b03c992ec97868be765b98048118a96f42bdc466a963c243c223b95196304209'
...     '8effffffffffff158affffffffff39b9cdffffffff2ec6a2f5ffff7ff2a42b21'
... )
>>> python.des(python.ser_p) == p
True
>>> python.ser(python.des(python.ser_p)) == python.ser_p
True
```

static mul(*s:* oblivious.bn254.scalar, *p:* oblivious.bn254.point) → *oblivious.bn254.point*

Multiply a point by a scalar and return the result.

```
>>> p = python.pnt(hashlib.sha512('123'.encode()).digest())
>>> s = python.scl(bytes.fromhex(
```

(continues on next page)

(continued from previous page)

```
...     '35c141f1c2c43543de9d1888e5a210abca3cd39a1e986304991ceded42b11709'
... ))
>>> python.mul(s, p).hex()[:64]
'68b5dd61adaa83f1511efe7b4749481cc9f86e11bf82d82960b6c56373de0d24'
```

static add(*p:* oblivious.bn254.point, *q:* oblivious.bn254.point) → *oblivious.bn254.point*

Return the sum of the supplied points.

```
>>> p = python.point.hash('123'.encode())
>>> q = python.point.hash('456'.encode())
>>> python.point.to_bytes(python.add(p, q)).hex()[:64]
'1ea48cab238fece46bd0c9fb562c859e318e17a8fb75517a4750d30ca79b911c'
>>> python.add(python.sub(p, q), q) == p
True
```

static sub(*p:* oblivious.bn254.point, *q:* oblivious.bn254.point) → *oblivious.bn254.point*

Return the result of subtracting the right-hand point from the left-hand point.

```
>>> p = python.point.hash('123'.encode())
>>> q = python.point.hash('456'.encode())
>>> python.sub(p, q).hex()[:64]
'a43a5ce1931b1300b62e5d7e1b0c691203bfd85fafd9585dc5e47a7e2acfea22'
>>> python.sub(python.add(p, q), q) == p
True
```

static neg(*p:* oblivious.bn254.point) → *oblivious.bn254.point*

Return the additive inverse of a point.

```
>>> p = python.point.hash('123'.encode())
>>> python.point.to_bytes(python.neg(p)).hex()[:64]
'825aa78af4c88d6de4abaebabf1a96f668956b92876cfb5d3a44829899cb480f'
```

static rnd() → *oblivious.bn254.scalar*

Return random non-zero scalar.

```
>>> isinstance(python.rnd(), python.scalar)
True
```

classmethod scl(*s: Optional[Union[bytes, bytearray]] = None*) → Optional[*oblivious.bn254.scalar*]

Construct a scalar if the supplied bytes-like object represents a valid scalar; otherwise, return `None`. If no byte vector is supplied, return a random scalar.

```
>>> s = python.scl()
>>> t = python.scl(s)
>>> s == t
True
>>> python.scl(bytes([255] * 32)) is None
True
```

static sse(*s:* oblivious.bn254.scalar) → bytes

Return the binary representation of a scalar.

```
>>> s = python.scalar.hash('123'.encode())
>>> python.sde(python.sse(s)) == s
True
```

static sde(*bs:* *bytes*) → *oblivious.bn254.scalar*

Construct a scalar from its binary representation.

```
>>> s = python.scalar.hash('123'.encode())
>>> bs = bytes.fromhex(
...     '93d829354cb3592743174133104b5405ba6992b67bb219fbde3e394d70505913'
... )
>>> python.sde(bs) == s
True
>>> python.sse(python.sde(bs)) == bs
True
```

static inv(*s:* oblivious.bn254.scalar) → *oblivious.bn254.scalar*

Return the inverse of a scalar (modulo `r = 16798108731015832284940804142231733909759579603404752749028378` in the prime field *F**_r*).

```
>>> s = python.scl()
>>> p = python.pnt()
>>> python.mul(python.inv(s), python.mul(s, p)) == p
True
```

static smu(*s:* oblivious.bn254.scalar, *t:* oblivious.bn254.scalar) → *oblivious.bn254.scalar*

Return the product of two scalars.

```
>>> s = python.scl()
>>> t = python.scl()
>>> python.smu(s, t) == python.smu(t, s)
True
```

static sad(*s:* oblivious.bn254.scalar, *t:* oblivious.bn254.scalar) → *oblivious.bn254.scalar*

Return the sum of two scalars.

```
>>> s = python.scl()  # Could be `python.scl()`.
>>> t = python.scl()
>>> python.sad(s, t) == python.sad(t, s)
True
```

static ssu(*s:* oblivious.bn254.scalar, *t:* oblivious.bn254.scalar) → *oblivious.bn254.scalar*

Return the result of subtracting the right-hand scalar from the left-hand scalar.

```
>>> s = python.scl()
>>> t = python.scl()
>>> python.ssu(s, t) == python.sad(s, python.sne(t))
True
>>> python.ssu(s, t) == python.sne(python.ssu(t, s))
True
```

static sne(*s:* oblivious.bn254.scalar) → *oblivious.bn254.scalar*

Return the additive inverse of a scalar.

```
>>> s = python.scl()
>>> t = python.scl()
>>> python.sne(python.sne(s)) == s
True
```

static pnt2(*h: Optional[bytes] = None*) → *oblivious.bn254.point2*

Construct a second-level point if the supplied bytes-like object represents a valid second-level point; otherwise, return None. If no byte vector is supplied, return a random second-level point.

```
>>> p = python.pnt2(hashlib.sha512('123'.encode()).digest())
>>> python.point2.to_bytes(p).hex()[:128] == (
...     '4c595542640a69c4a70bda55c27ef96c133cd1f4a5f83b3371e571960c018e19'
...     'c54aaec2069f8f10a00f12bcbb3511cdb7356201f5277ec5e47da91405be2809'
... )
True
```

static bas2(*s:* oblivious.bn254.scalar) → *oblivious.bn254.point2*

Return the base second-level point multiplied by the supplied scalar.

```
>>> bytes(python.bas2(python.scalar.hash('123'.encode()))).hex()[:64]
'e7000fb12d206112c73fe1054e9d77b35c77881eba6598b7e035171d90b13e0c'
```

static can2(*p:* oblivious.bn254.point2) → *oblivious.bn254.point2*

Normalize the representation of a second-level point into its canonical form and return the result.

```
>>> p = python.bas2(scalar.from_int(1))
>>> python.ser(python.can2(p)).hex()[:64]
'669e6563afaa45af7cbc013d23f092bb3763d4dc41b97aef555bdf61de713f17'
```

static ser2(*p:* oblivious.bn254.point2) → bytes

Return the binary representation of a second-level point.

```
>>> p = python.point2.hash('123'.encode())
>>> python.des2(python.ser2(p)) == p
True
```

It is the responsibility of the user to ensure that only canonical representations of points are serialized.

static des2(*bs: bytes*) → *oblivious.bn254.point2*

Return the second-level point corresponding to the supplied binary representation thereof.

```
>>> p = python.point2.hash('123'.encode())
>>> ser_p = bytes.fromhex(
...     '30326199f303fce7a77cff6d2fb0b3de8cd409d1d562f3543f7d064cdc58d309'
...     '7e88038ad76e85e5df26e4a9486a657b0431c8e7e09b0a1abf90fc874c515207'
...     '2c6a88bb448065eb748df632b1d872e02f54b6f56fdb84a7b1cb388fe551fb08'
...     '04464efa186bd4b1371e53d6f31f0e2f50ff553b6264a43331b42c976a0c541f'
...     '8effffffffffff158affffffffff39b9cdffffffff2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
>>> python.des2(ser_p) == p
True
>>> python.ser(python.des2(ser_p)) == ser_p
True
```

static mul2(*s:* oblivious.bn254.scalar, *p:* oblivious.bn254.point2) → *oblivious.bn254.point2*
Multiply a second-level point by a scalar.

```
>>> p = python.point2.hash('123'.encode())
>>> s = python.scl(bytes.fromhex(
...     '35c141f1c2c43543de9d188805a210abca3cd39a1e986304991ceded42b11709'
... ))
>>> python.point2.to_bytes(python.mul2(s, p)).hex() == (
...     '5f6f2ace8566ca47354fbe244ae3e6a854c37011fb6d6ac56571c94169e4ab18'
...     '650bea4cfed5c9603e5949fe3d7509b17e20db4ff1f05129aad0d0a3bffb0008'
...     '3043c5a14b986882836b1c929952ea3881d04ca44d487d1ab2d4c0b171b87d14'
...     '5dca6dabb4f0ea7be5c95a861ed319d146b15d70542d3952af995a8bb35b8314'
...     '8effffffffffff158affffffffff39b9cdfffffffe2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static add2(*p:* oblivious.bn254.point2, *q:* oblivious.bn254.point2) → *oblivious.bn254.point2*
Return sum of the supplied second-level points.

```
>>> p = python.point2.hash('123'.encode())
>>> q = python.point2.hash('456'.encode())
>>> python.point2.to_bytes(python.add2(p, q)).hex() == (
...     'cb0fc423c1bac2ac2df47bf5f5548a42b0d0a0da325bc77243d15dc587a7b221'
...     '9808a1649991ddf770f0060333aab4d499580b123f109b5cb180f1f8a75a090e'
...     '83dd34d9ecdd6fd639230f7f0cf44b218fae4d879111de6c6c037e6ffdcdc823'
...     'f5a48318143873ca90ad512a2ea1854200eea5537cd0ac93691d5b94ff36b212'
...     '8effffffffffff158affffffffff39b9cdfffffffe2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static sub2(*p:* oblivious.bn254.point2, *q:* oblivious.bn254.point2) → *oblivious.bn254.point2*
Return the result of subtracting right-hand second-level point from the left-hand second-level point.

```
>>> p = python.point2.hash('123'.encode())
>>> q = python.point2.hash('456'.encode())
>>> python.point2.to_bytes(python.sub2(p, q)).hex() == (
...     'e97a70c4e3a5369ebbb1dcf0cc1135c8c8e04a4ec7cffdf875ac429d66846d0b'
...     '191b090909c40a723027b07ac44435a6ade3813d04b3632a17c92c5c98718902'
...     '407c58ed13cc0c0aaa43d0eafd44080080c8199401fe4f8ed7dd0eb5fba86817'
...     '141f74341ce3c4884f86a97f51f7c0b208fe52be336b7651252fa9881c93d203'
...     '8effffffffffff158affffffffff39b9cdfffffffe2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static neg2(*p:* oblivious.bn254.point2) → *oblivious.bn254.point2*
Return the negation of a second-level point.

```
>>> p = python.point2.hash('123'.encode())
>>> python.point2.to_bytes(python.neg2(p)).hex() == (
...     '30326199f303fce7a77cff6d2fb0b3de8cd409d1d562f3543f7d064cdc58d309'
...     '7e88038ad76e85e5df26e4a9486a657b0431c8e7e09b0a1abf90fc874c515207'
```

(continues on next page)

(continued from previous page)

```
...     'e7957744bb7f9abb9e7209cd4e27ae80d8ab490a1072af125034c7b09c12281c'
...     '0fbab105e7942bf5dbe1ac290ce01232b800aac41de98f86d04bd3a81758cf05'
...     '8efffffffffff158affffffffff39b9cdfffffff2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static rnd2() → *oblivious.bn254.scalar2*

Return random non-zero second-level scalar.

```
>>> isinstance(python.rnd2(), python.scalar2)
True
```

static scl2(*s: Optional[Union[bytes, bytearray]] = None*) → Optional[*oblivious.bn254.scalar2*]

Construct a second-level scalar if the supplied bytes-like object represents a valid second-level scalar; otherwise, return `None`. If no byte vector is supplied, return a random second-level scalar.

```
>>> bs = bytes.fromhex(
...     '18d0e065798ffa4ecca0a7cc6e2b8d6d3269f7eb413525e59b731332b02ea805'
...     '4b90c89ce93e4452557f53aae5f8f13221858dde2ea6a50373f1858f11287021'
...     '09b3daf7a217a20a3d872632f8ced643d32649b241a67601bec855bd43f07710'
...     '88df722bcacc8fd174f16ad7ef1b1d5e622d93b7cfec6385c353c13793a4e01c'
...     '371ef54cd836a88e1b1f1e198d566d3bbe5aa225734305f14cac9af9c821351c'
...     '86b365190105d1b7e71011d1362ed6ad9133a2c2c42898e9ebc3d1604a608f10'
...     '8461d37b2463b4f88f0ccb427b5eea8714ee88f8c33daf7ee4a3f45ca8bca911'
...     '87956abb4d27de1b49e5e059f904fbb475804715ab25a17306fa0e196f305215'
...     '3a60dfa073ca78de49edeaac9c54cab3d0138750c23c0b68de7924a69d1ba002'
...     'a24ac592622a45c59c1ded7133794292a09602bd57a36601d35438846fcb370f'
...     '39c0fa7a15b34d14ab6f95ee5bf5b26bd5def1e7ed07350922d445da07d93622'
...     '2db5baa9dec152c2b2bcfc46cde6fd22e70271af8a164e77e5808ce602095a1f'
... )
>>> python.scalar2.to_bytes(python.scl2(bs)).hex()[700:]
'36222db5baa9dec152c2b2bcfc46cde6fd22e70271af8a164e77e5808ce602095a1f'
```

static sse2(*s:* oblivious.bn254.scalar2) → bytes

Return the binary representation of a second-level scalar.

```
>>> s = python.scalar2.hash('123'.encode())
>>> python.sde2(python.sse2(s)) == s
True
```

static sde2(*bs: bytes*) → *oblivious.bn254.scalar2*

Construct a second-level scalar from its binary representation.

```
>>> s = python.scalar2.hash('123'.encode())
>>> bs = bytes.fromhex(
...     '36980a8359d40e106075488e80cf1479f2e6ba95d6a99a67832d21b7b94d8c1d'
...     '5eb4d655f23e1d5d499d51d1c552b5e7df6943091427cd080f582e120613a021'
...     '85898ef7d016e47a74a8df62316cc4ad975cb64bb63867ed9b5221f77bb9a121'
...     '7bd89cd213eee0c3fdf2e0e13ef9e30383ea5607c8d13fc10e04448a6c964a00'
...     '04a098a55beab0973222096631933360 8b2187ee2196eb5b4253bc2b1aea5303'
...     '654260dd687a2eb176a494258ff7ef753f93105a6f0e9f46c926afdbe31ff124'
...     '6bdd87c32537abcdb46ad542792edd74a229c9ba61abcd993f074237a91f5215'
```

(continues on next page)

(continued from previous page)

```
...     '8f6b07886895733edde15cb22129459162d89d3662826b74e4fcbe4e9e8c2420'
...     'bd53586a09f91ff8f67f92cba72c5b64a9c3965c01e93710200ab4e084955316'
...     'fb18950835b79fb4c2930efcc5fcaa9d82ee0faff036b80657daee233a445901'
...     '7df3e57cb535ed26162b3ee0f8961131a93fe3198dc5393d277ed8bac5532411'
...     '93b7ad15c52ca123fd26f592a2219b1bf118b3035893cc4abf614b422f978718'
... )
>>> python.sde2(bs) == s
True
>>> python.sse(python.sde2(bs)) == bs
True
```

static inv2(*s:* oblivious.bn254.scalar2) → *oblivious.bn254.scalar2*

Return the inverse of a second-level scalar.

```
>>> s = python.scl2()
>>> python.smu2(s, python.smu2(s, python.inv2(s))) == s
True
>>> python.smu2(python.smu2(s, s), python.inv2(s)) == s
True
```

static smu2(*s:* oblivious.bn254.scalar2, *t:* oblivious.bn254.scalar2) → *oblivious.bn254.scalar2*

Return second-level scalar multiplied by another scalar.

```
>>> s = python.scalar2.random()
>>> t = python.scalar2.random()
>>> python.smu2(s, t) == python.smu2(t, s)
True
```

static sad2(*s:* oblivious.bn254.scalar2, *t:* oblivious.bn254.scalar2) → *oblivious.bn254.scalar2*

Return scalar2 added to another scalar2.

```
>>> s = python.scl2()
>>> t = python.scl2()
>>> python.sad2(s, t) == python.sad2(t, s)
True
```

class point(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: bytes

class point2(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: bytes

class scalar(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: bytes

class scalar2(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: bytes

class oblivious.bn254.**mcl**

Bases: object

Wrapper class for binary implementations of primitive operations.

When this module is imported, it makes a number of attempts to locate an instance of the shared/dynamic library file of the mclbn256 library on the host system. The sequence of attempts is listed below, in order.

1. It uses `ctypes.util.find_library` to look for `'mcl'` or `'mclbn256'`.

2. It attempts to find a file `mclbn256.so` or `mclbn256.dll` in the paths specified by the `PATH` and `LD_LIBRARY_PATH` environment variables.

3. If the mclbn256 package is installed, it reverts to the compiled subset of mclbn256 included in that package.

If all of the above fail, then *mcl* is assigned the value `None` and all classes exported by this module default to their pure-Python variants (*i.e.*, those encapsulated within *python*). One way to confirm that a dynamic/shared library *has been found* when this module is imported is to evaluate the expression `mcl is not None`.

If a shared/dynamic library file has been loaded successfully, this class encapsulates shared/dynamic library variants of all classes exported by this module and of all the underlying low-level operations: *mcl.pnt*, *mcl.bas*, *mcl.can*, *mcl.ser*, *mcl.des*, *mcl.mul*, *mcl.add*, *mcl.sub*, *mcl.neg*, *mcl.par*, *mcl.rnd*, *mcl.scl*, *mcl.sse*, *mcl.sde*, *mcl.inv*, *mcl.smu*, *mcl.sad*, *mcl.ssu*, *mcl.sne*, *mcl.pnt2*, *mcl.bas2*, *mcl.can2*, *mcl.ser2*, *mcl.des2*, *mcl.mul2*, *mcl.add2*, *mcl.sub2*, *mcl.neg2*, *mcl.rnd2*, *mcl.scl2*, *mcl.sse2*, *mcl.sde2*, *mcl.inv2*, *mcl.smu2*, *mcl.sad2*, *mcl.point*, *mcl.scalar*, *mcl.point2*, and *mcl.scalar2*. For example, you can perform addition of points using the point addition implementation found in the shared/dynamic library bundled with the instance of the package mclbn256 that is found on the host system.

```
>>> p = mcl.pnt()
>>> q = mcl.pnt()
>>> mcl.add(p, q) == mcl.add(q, p)
True
```

Methods found in the shared/dynamic library variants of the *point*, *scalar*, *point2*, and *scalar2* classes are wrappers for the shared/dynamic library implementations of the underlying operations.

```
>>> p = mcl.point()
>>> q = mcl.point()
>>> p + q == q + p
True
```

static pnt(*h: Optional[Union[bytes, bytearray]] = None*) → mclbn256.mclbn256.G1

Construct a point if the supplied bytes-like object represents a valid point; otherwise, return `None`. If no byte vector is supplied, return a random point.

```
>>> p = mcl.pnt(hashlib.sha512('123'.encode()).digest())
>>> p.__class__ = point
>>> mcl.point.to_bytes(p).hex()[:64]
'6d68495eb4d539db4df70fd24d54fae37c9adf7dfd8dc705ccb8de8630e7cf22'
```

static bas(*s: mclbn256.mclbn256.Fr*) → mclbn256.mclbn256.G1

Return the base point multiplied by the supplied scalar.

```
>>> p = mcl.bas(mcl.scalar.hash('123'.encode())).normalize().normalize()
>>> p.__class__ = point
>>> mcl.point.to_bytes(p).hex()[:64]
'2d66076815cda25556bab4a930244ac284412267e9345aceb98d71530308401a'
```

static can(*p: mclbn256.mclbn256.G1*) → mclbn256.mclbn256.G1

Normalize the representation of a point into its canonical form and return the result.

```
>>> a = mcl.point.hash('123'.encode())
>>> p = mcl.add(a, a)
>>> p_can = mcl.can(mcl.add(a, a))
```

We may have `ser(p_can) != ser(p)` here, depending on the backend implementation. Either normalization matters, or MCl is not the backend.

```
>>> (mcl.ser(p_can) != mcl.ser(p)) or not mclbn256
True
```

Normalization is idempotent.

```
>>> mcl.can(p) == mcl.can(p_can)
True
```

static ser(*p: mclbn256.mclbn256.G1*) → bytes
Return the binary representation of a point.

```
>>> p = mcl.point.hash('123'.encode())
>>> mcl.des(mcl.ser(p)) == p
True
```

static des(*bs: bytes*) → mclbn256.mclbn256.G1
Construct a point corresponding to the supplied binary representation.

```
>>> p = mcl.point.hash('123'.encode())
>>> ser_p = bytes.fromhex(
...     '825aa78af4c88d6de4abaebabf1a96f668956b92876cfb5d3a44829899cb480f'
...     'b03c992ec97868be765b98048118a96f42bdc466a963c243c223b95196304209'
...     '8effffffffffff158affffffffff39b9cdffffffff2ec6a2f5ffff7ff2a42b21'
... )
>>> mcl.des(ser_p) == p
True
>>> mcl.ser(mcl.des(ser_p)) == ser_p
True
```

static mul(*s: mclbn256.mclbn256.Fr*, *p: mclbn256.mclbn256.G1*) → mclbn256.mclbn256.G1
Multiply a point by a scalar and return the result.

```
>>> p = mcl.pnt(hashlib.sha512('123'.encode()).digest())
>>> s = mcl.scl(bytes.fromhex(
...     '35c141f1c2c43543de9d1888d5a210abca3cd39a1e986304991ceded42b11709'
... ))
>>> q = mcl.mul(s, p).normalize().normalize()
>>> q.__class__ = point
>>> mcl.point.to_bytes(q).hex()[:64]
'68b5dd61adaa83f1511efe7b4749481cc9f86e11bf82d82960b6c56373de0d24'
```

static add(*p: mclbn256.mclbn256.G1*, *q: mclbn256.mclbn256.G1*) → mclbn256.mclbn256.G1
Return the sum of the supplied points.

```
>>> p = mcl.point.hash('123'.encode())
>>> q = mcl.point.hash('456'.encode())
>>> r = mcl.add(p, q).normalize().normalize()
>>> r.__class__ = point
>>> mcl.point.to_bytes(r).hex()[:64]
'1ea48cab238fece46bd0c9fb562c859e318e17a8fb75517a4750d30ca79b911c'
```

static sub(*p: mclbn256.mclbn256.G1*, *q: mclbn256.mclbn256.G1*) → mclbn256.mclbn256.G1
Return the result of subtracting the right-hand point from the left-hand point.

```
>>> p = mcl.point.hash('123'.encode())
>>> q = mcl.point.hash('456'.encode())
>>> r = mcl.sub(p, q).normalize().normalize()
>>> r.__class__ = point
>>> mcl.point.to_bytes(r).hex()[:64]
'a43a5ce1931b1300b62e5d7e1b0c691203bfd85fafd9585dc5e47a7e2acfea22'
```

static neg(*p: mclbn256.mclbn256.G1*) → mclbn256.mclbn256.G1

Return the additive inverse of a point.

```
>>> p = mcl.point.hash('123'.encode())
>>> q = mcl.neg(p)
>>> q.__class__ = point
>>> mcl.point.to_bytes(q).hex()[:64]
'825aa78af4c88d6de4abaebabf1a96f668956b92876cfb5d3a44829899cb480f'
```

static par(*p: Union[mclbn256.mclbn256.G1, mclbn256.mclbn256.G2]*, *q: Union[mclbn256.mclbn256.G1, mclbn256.mclbn256.G2]*) → mclbn256.mclbn256.GT

Compute the pairing function on two points.

```
>>> p = mcl.point.hash('123'.encode())
>>> q = mcl.point2.base(mcl.scalar.from_int(456))
>>> r = mcl.par(p, q)
>>> r.__class__ = mcl.scalar2
>>> mcl.scalar2.to_bytes(r).hex()[700:]
'd01f7e038b05acc5519eeda026c4aa111eb12f3483f274c60e34e6ec7571435df707'
```

The order of the two arguments is not important (as long as exactly one argument is an instance of *point* and the other is an instance of *point2*).

```
>>> r = mcl.par(q, p)
>>> r.__class__ = mcl.scalar2
>>> mcl.scalar2.to_bytes(r).hex()[700:]
'd01f7e038b05acc5519eeda026c4aa111eb12f3483f274c60e34e6ec7571435df707'
```

The pairing function is bilinear.

```
>>> p = mcl.point.random()
>>> s = mcl.scalar.random()
```

```
>>> t = mcl.scalar.random()
>>> q = mcl.point2.random()
>>> -((~s) * (s * p)) - p == mcl.scalar.from_int(-2) * p
True
>>> s * t * p @ q == s * p @ (t * q)
True
```

Suppose there are two points: one multiplied by the scalar `s` and the other multiplied by the scalar `t`. Their equality can be determined by using a balancing point: `g**(~s * t)`. If the pairing of `t * x` with `g` is the same as the pairing with `s * y` and `g**(~s * t)`, then `x` equals `y`.

```
>>> x = y = p
>>> g = mcl.point2.base(mcl.scalar.from_int(1))
```

(continues on next page)

(continued from previous page)

```
>>> b = mcl.point2.base(~s * t)
>>> (t * x) @ g == (s * y) @ b
True
```

This operation is defined only for a point and a second-level point. Any attempt to invoke the operation on values or objects of other types raises an exception.

```
>>> p @ (p + p)
Traceback (most recent call last):
  ...
TypeError: pairing is defined only for a point and a second-level point
>>> g @ b
Traceback (most recent call last):
  ...
TypeError: pairing is defined only for a point and a second-level point
```

Pairing is intended to be nonsingular.

```
>>> p @ q.clear()
Traceback (most recent call last):
  ...
TypeError: cannot meaningfully pair the infinity point
>>> p.clear() @ g
Traceback (most recent call last):
  ...
TypeError: cannot meaningfully pair the infinity point
```

static rnd() → mclbn256.mclbn256.Fr

Return random non-zero scalar.

```
>>> s = mcl.rnd()
>>> isinstance(s, Fr)
True
>>> s.__class__ = scalar
>>> len(mcl.scalar.to_bytes(s))
32
```

classmethod scl(*bs: Optional[Union[bytes, bytearray]] = None*) → Optional[mclbn256.mclbn256.Fr]

Construct a scalar if the supplied bytes-like object represents a valid scalar; otherwise, return `None`. If no byte vector is supplied, return a random scalar.

```
>>> s = mcl.scl()
>>> s.__class__ = scalar
>>> t = mcl.scl(mcl.scalar.to_bytes(s))
>>> s == t
True
>>> mcl.scl(bytes([255] * 32)) is None
True
```

static sse(*s: mclbn256.mclbn256.Fr*) → bytes

Return the binary representation of a scalar.

```
>>> s = mcl.scalar.hash('123'.encode())
>>> mcl.sde(mcl.sse(s)) == s
True
```

static sde(*bs: bytes*) → mclbn256.mclbn256.Fr

Return a scalar from its binary representation.

```
>>> s = mcl.scalar.hash('123'.encode())
>>> bs = bytes.fromhex(
...     '93d829354cb3592743174133104b5405ba6992b67bb219fbde3e394d70505913'
... )
>>> mcl.sde(bs) == s
True
>>> mcl.sse(mcl.sde(bs)) == bs
True
```

static inv(*s: mclbn256.mclbn256.Fr*) → mclbn256.mclbn256.Fr

Return inverse of a scalar (modulo `r = 16798108731015832284940804142231733909759579603404752749028378864`
in the prime field *F*_*r*).

```
>>> (s, p) = (mcl.scl(), mcl.pnt())
>>> mcl.mul(mcl.inv(s), mcl.mul(s, p)) == p
True
```

static smu(*s: mclbn256.mclbn256.Fr*, *t: mclbn256.mclbn256.Fr*) → mclbn256.mclbn256.Fr

Return scalar multiplied by another scalar.

```
>>> (s, t) = (mcl.scl(), mcl.scl())
>>> mcl.smu(s, t) == mcl.smu(t, s)
True
```

static sad(*s: mclbn256.mclbn256.Fr*, *t: mclbn256.mclbn256.Fr*) → mclbn256.mclbn256.Fr

Return scalar added to another scalar.

```
>>> (s, t) = (mcl.scl(), mcl.scl())
>>> mcl.sad(s, t) == mcl.sad(t, s)
True
```

static ssu(*s: mclbn256.mclbn256.Fr*, *t: mclbn256.mclbn256.Fr*) → mclbn256.mclbn256.Fr

Return the result of one scalar subtracted from another scalar.

```
>>> (s, t) = (mcl.scl(), mcl.scl())
>>> mcl.ssu(s, t) == mcl.sad(s, mcl.sne(t))
True
>>> mcl.ssu(s, t) == mcl.sne(mcl.ssu(t, s))
True
```

static sne(*s: mclbn256.mclbn256.Fr*) → mclbn256.mclbn256.Fr

Return the additive inverse of a scalar.

```
>>> (s, t) = (mcl.scl(), mcl.scl())
>>> mcl.sne(mcl.sne(s)) == s
True
```

static pnt2(*h: Optional[bytes] = None*) → mclbn256.mclbn256.G2
Construct a second-level point if the supplied bytes-like object represents a valid second-level point; otherwise, return None. If no byte vector is supplied, return a random second-level point.

```
>>> p = mcl.pnt2(hashlib.sha512('123'.encode()).digest())
>>> p.__class__ = point2
>>> mcl.point2.to_bytes(p.canonical().canonical()).hex()[:128] == (
...     '4c595542640a69c4a70bda55c27ef96c133cd1f4a5f83b3371e571960c018e19'
...     'c54aaec2069f8f10a00f12bcbb3511cdb7356201f5277ec5e47da91405be2809'
... )
True
```

static bas2(*s*) → mclbn256.mclbn256.G2
Return the base second-level point multiplied by the supplied scalar.

```
>>> r = mcl.bas2(mcl.scalar.hash('123'.encode())).normalize().normalize()
>>> r.__class__ = point2
>>> mcl.point2.to_bytes(r).hex()[:64]
'e7000fb12d206112c73fe1054e9d77b35c77881eba6598b7e035171d90b13e0c'
```

static can2(*p: mclbn256.mclbn256.G2*) → mclbn256.mclbn256.G2
Normalize the representation of a second-level point into its canonical form and return the result.

```
>>> p = mcl.bas2(scalar.from_int(1))
>>> mcl.ser(mcl.can2(p)).hex()[:64]
'669e6563afaa45af7cbc013d23f092bb3763d4dc41b97aef555bdf61de713f17'
```

static ser2(*p: mclbn256.mclbn256.G2*) → bytes
Return the binary representation of a second-level point.

```
>>> p = mcl.point2.hash('123'.encode())
>>> mcl.des2(mcl.ser2(p)) == p
True
```

It is the responsibility of the user to ensure that only canonical representations of points are serialized.

static des2(*bs: bytes*) → mclbn256.mclbn256.G2
Return the second-level point corresponding to the supplied binary representation thereof.

```
>>> p = mcl.point2.hash('123'.encode())
>>> mcl.ser_p = bytes.fromhex(
...     'b5b0a52e43ba71ae03317333da4ba9452dbdbbec353ade0c732348e0bea4ba1b'
...     '8860718e5ba784d55799ab292459a638f6399738a6de348742e6a789674f300d'
...     '7e59c60a595253ebf69bf0794b7a032e59b6b5037adba410d680b53ffac08517'
...     'cf5bc3be9d850ec64ea6939904cf66b66b6b4b82be03ee4f10661fedaf83841f'
...     'ba7e678442a658340a5b3c51eb5076d738cf88387ada6cbd1fe7f8d8a2268417'
...     'bc8aedbc99808b0450025d0c75b5f1ccb34bc69934cc620d9ea51038a1d98721'
... )
>>> mcl.des2(mcl.ser_p) == p
True
>>> mcl.ser(mcl.des2(mcl.ser_p)) == mcl.ser_p
True
```

static mul2(*s: mclbn256.mclbn256.Fr*, *p: mclbn256.mclbn256.G2*) → mclbn256.mclbn256.G2
Multiply a second-level point by a scalar.

```
>>> p = mcl.point2.hash('123'.encode())
>>> s = mcl.scl(bytes.fromhex(
...     '35c141f1c2c43543de9d188805a210abca3cd39a1e986304991ceded42b11709'
... ))
>>> r = mcl.mul2(s, p).normalize().normalize()
>>> r.__class__ = point2
>>> mcl.point2.to_bytes(r).hex() == (
...     '5f6f2ace8566ca47354fbe244ae3e6a854c37011fb6d6ac56571c94169e4ab18'
...     '650bea4cfed5c9603e5949fe3d7509b17e20db4ff1f05129aad0d0a3bffb0008'
...     '3043c5a14b986882836b1c929952ea3881d04ca44d487d1ab2d4c0b171b87d14'
...     '5dca6dabb4f0ea7be5c95a861ed319d146b15d70542d3952af995a8bb35b8314'
...     '8efffffffffff158affffffffff39b9cdfffffff2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static add2(*p: mclbn256.mclbn256.G2*, *q: mclbn256.mclbn256.G2*) → mclbn256.mclbn256.G2

Return sum of the supplied second-level points.

```
>>> p = mcl.point2.hash('123'.encode())
>>> q = mcl.point2.hash('456'.encode())
>>> r = mcl.add2(p, q).normalize().normalize()
>>> r.__class__ = point2
>>> mcl.point2.to_bytes(r).hex() == (
...     'cb0fc423c1bac2ac2df47bf5f5548a42b0d0a0da325bc77243d15dc587a7b221'
...     '9808a1649991ddf770f0060333aab4d499580b123f109b5cb180f1f8a75a090e'
...     '83dd34d9ecdd6fd639230f7f0cf44b218fae4d879111de6c6c037e6ffdcdc823'
...     'f5a48318143873ca90ad512a2ea1854200eea5537cd0ac93691d5b94ff36b212'
...     '8efffffffffff158affffffffff39b9cdfffffff2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static sub2(*p: mclbn256.mclbn256.G2*, *q: mclbn256.mclbn256.G2*) → mclbn256.mclbn256.G2

Return the result of subtracting the right-hand second-level point from the left-hand second-level point.

```
>>> p = mcl.point2.hash('123'.encode())
>>> q = mcl.point2.hash('456'.encode())
>>> r = mcl.sub2(p, q).normalize().normalize()
>>> r.__class__ = point2
>>> mcl.point2.to_bytes(r).hex() == (
...     'e97a70c4e3a5369ebbb1dcf0cc1135c8c8e04a4ec7cffdf875ac429d66846d0b'
...     '191b090909c40a723027b07ac44435a6ade3813d04b3632a17c92c5c98718902'
...     '407c58ed13cc0c0aaa43d0eafd44080080c8199401fe4f8ed7dd0eb5fba86817'
...     '141f74341ce3c4884f86a97f51f7c0b208fe52be336b7651252fa9881c93d203'
...     '8efffffffffff158affffffffff39b9cdfffffff2ec6a2f5ffff7ff2a42b21'
...     '0000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static neg2(*p: mclbn256.mclbn256.G2*) → mclbn256.mclbn256.G2

Return the negation of a second-level point.

```
>>> p = mcl.point2.hash('123'.encode())
>>> r = mcl.neg2(p).normalize().normalize()
>>> r.__class__ = point2
>>> mcl.point2.to_bytes(r).hex() == (
...     '30326199f303fce7a77cff6d2fb0b3de8cd409d1d562f3543f7d064cdc58d309'
...     '7e88038ad76e85e5df26e4a9486a657b0431c8e7e09b0a1abf90fc874c515207'
...     'e7957744bb7f9abb9e7209cd4e27ae80d8ab490a1072af125034c7b09c12281c'
...     '0fbab105e7942bf5dbe1ac290ce01232b800aac41de98f86d04bd3a81758cf05'
...     '8effffffffffff158affffffffff39b9cdfffffffff2ec6a2f5ffff7ff2a42b21'
...     '000000000000000000000000000000000000000000000000000000000000000'
... )
True
```

static rnd2() → mclbn256.mclbn256.GT

Return random non-zero second-level scalar.

```
>>> isinstance(mcl.rnd2(), GT)
True
```

static scl2(*s: Optional[Union[bytes, bytearray]] = None*) → Optional[mclbn256.mclbn256.GT]

Construct a second-level scalar if the supplied bytes-like object represents a valid second-level scalar; otherwise, return `None`. If no byte vector is supplied, return a random second-level scalar.

```
>>> bs = bytes.fromhex(
...     '18d0e065798ffa4ecca0a7cc6e2b8d6d3269f7eb413525e59b731332b02ea805'
...     '4b90c89ce93e4452557f53aae5f8f13221858dde2ea6a50373f1858f11287021'
...     '09b3daf7a217a20a3d872632f8ced643d32649b241a67601bec855bd43f07710'
...     '88df722bcacc8fd174f16ad7ef1b1d5e622d93b7cfec6385c353c13793a4e01c'
...     '371ef54cd836a88e1b1f1e198d566d3bbe5aa225734305f14cac9af9c821351c'
...     '86b365190105d1b7e71011d1362ed6ad9133a2c2c42898e9ebc3d1604a608f10'
...     '8461d37b2463b4f88f0ccb427b5eea8714ee88f8c33daf7ee4a3f45ca8bca911'
...     '87956abb4d27de1b49e5e059f904fbb475804715ab25a17306fa0e196f305215'
...     '3a60dfa073ca78de49edeaac9c54cab3d0138750c23c0b68de7924a69d1ba002'
...     'a24ac592622a45c59c1ded7133794292a09602bd57a36601d35438846fcb370f'
...     '39c0fa7a15b34d14ab6f95ee5bf5b26bd5def1e7ed07350922d445da07d93622'
...     '2db5baa9dec152c2b2bcfc46cde6fd22e70271af8a164e77e5808ce602095a1f'
... )
>>> s = mcl.scl2(bs)
>>> s.__class__ = mcl.scalar2
>>> mcl.scalar2.to_bytes(s).hex()[700:]
'36222db5baa9dec152c2b2bcfc46cde6fd22e70271af8a164e77e5808ce602095a1f'
```

static sse2(*s:* oblivious.bn254.scalar2) → bytes

Return the binary representation of a second-level scalar.

```
>>> s = scalar2.hash('123'.encode())
>>> mcl.sde2(mcl.sse2(s)) == s
True
```

static sde2(*bs: bytes*) → mclbn256.mclbn256.GT

Return the second-level scalar corresponding to the supplied binary representation thereof.

```
>>> s = mcl.scalar2.hash('123'.encode())
>>> bs = bytes.fromhex(
```

(continues on next page)

(continued from previous page)

```
...     '36980a8359d40e106075488e80cf1479f2e6ba95d6a99a67832d21b7b94d8c1d'
...     '5eb4d655f23e1d5d499d51d1c552b5e7df6943091427cd080f582e120613a021'
...     '85898ef7d016e47a74a8df62316cc4ad975cb64bb63867ed9b5221f77bb9a121'
...     '7bd89cd213eee0c3fdf2e0e13ef9e30383ea5607c8d13fc10e04448a6c964a00'
...     '04a098a55beab0973222096631933608b2187ee2196eb5b4253bc2b1aea5303'
...     '654260dd687a2eb176a494258ff7ef753f93105a6f0e9f46c926afdbe31ff124'
...     '6bdd87c32537abcdb46ad542792edd74a229c9ba61abcd993f074237a91f5215'
...     '8f6b07886895733edde15cb22129459162d89d3662826b74e4fcbe4e9e8c2420'
...     'bd53586a09f91ff8f67f92cba72c5b64a9c3965c01e93710200ab4e084955316'
...     'fb18950835b79fb4c2930efcc5fcaa9d82ee0faff036b80657daee233a445901'
...     '7df3e57cb535ed26162b3ee0f8961131a93fe3198dc5393d277ed8bac5532411'
...     '93b7ad15c52ca123fd26f592a2219b1bf118b3035893cc4abf614b422f978718'
... )
>>> mcl.sde2(bs) == s
True
>>> mcl.sse(mcl.sde2(bs)) == bs
True
```

static inv2(*s: mclbn256.mclbn256.GT*) → mclbn256.mclbn256.GT

Return the inverse of a second-level scalar.

```
>>> s = mcl.scl2()
>>> mcl.smu2(s, mcl.smu2(s, mcl.inv2(s))) == s
True
>>> mcl.smu2(mcl.smu2(s, s), mcl.inv2(s)) == s
True
```

static smu2(*s: mclbn256.mclbn256.GT*, *t: mclbn256.mclbn256.GT*) → mclbn256.mclbn256.GT

Return the product of two second-level scalars.

```
>>> p1 = mcl.point.hash('123'.encode())
>>> p2 = mcl.point.hash('456'.encode())
>>> q1 = mcl.point2.base(mcl.scalar.hash('123'.encode()))
>>> q2 = mcl.point2.base(mcl.scalar.hash('456'.encode()))
>>> s = p1 @ q1
>>> t = p2 @ q2
>>> mcl.smu2(s, t) == mcl.smu2(t, s)
True
```

static sad2(*s: mclbn256.mclbn256.GT*, *t: mclbn256.mclbn256.GT*) → mclbn256.mclbn256.GT

Return the sum of two second-level scalars.

```
>>> s = mcl.scl2()
>>> t = mcl.scl2()
>>> mcl.sad2(s, t) == mcl.sad2(t, s)
True
```

class point(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: `object`

class point2(*bs: Optional[Union[bytes, bytearray]] = None*)

Bases: `object`

class scalar(*bs: Optional[Union[bytes, bytearray]] = None*)
Bases: object

class scalar2(*bs: Optional[Union[bytes, bytearray]] = None*)
Bases: object

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

F

H

I

M

N

O

P

R

S

T