modulo

Release 2.1.0

Andrei Lapets

Jun 03, 2023

CONTENTS

Pure-Python library for working with modular arithmetic, congruence classes, and finite fields.

CHAPTER

ONE

PURPOSE

The library allows users to work with congruence classes (including finite field elements) as objects, with support for many common operations.

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install modulo
```

The library can be imported in the usual way:

```
from modulo import modulo
```

2.1 Examples

This library makes it possible to work with congruence classes (and sets of congruence classes such as finite fields) as objects. A congruence class is defined using a representative integer and a modulus:

```
>>> from modulo import modulo
>>> modulo(3, 7)
modulo(3, 7)
```

Built-in operators can be used to perform modular addition, modular subtraction, and modular negation of congruence classes:

```
>>> modulo(3, 7) + modulo(5, 7)
modulo(1, 7)
>>> modulo(1, 7) - modulo(4, 7)
modulo(4, 7)
>>> -modulo(5, 7)
modulo(2, 7)
```

Modular multiplication, division, inversion, and exponentiation are also supported (when they are defined):

```
>>> modulo(3, 7) * modulo(5, 7)
modulo(1, 7)
>>> modulo(1, 7) // modulo(3, 7)
modulo(5, 7)
>>> modulo(5, 7) ** 2
modulo(4, 7)
>>> modulo(5, 7) ** (-1)
modulo(3, 7)
```

Individual congruence classes can be compared with one another according to their least nonnegative residues (and, thus, can also be sorted):

```
>>> mod(2, 7) < mod(3, 7)
True
>>> list(sorted([mod(2, 3), mod(1, 3), mod(0, 3)]))
[modulo(0, 3), modulo(1, 3), modulo(2, 3)]
```

The membership operation is supported between integers and congruence classes:

```
>>> 3 in mod(3, 7)
True
>>> 10 in mod(3, 7)
True
>>> 4 in mod(3, 7)
False
```

A set of congruence classes such as a finite field can also be defined. The built-in length function `len` and the membership operator are supported:

```
>>> len(modulo(7))
7
>>> modulo(3, 7) in modulo(7)
True
```

The built-in `int` function can be used to retrieve the least nonnegative residue of a congruence class and the built-in `len` function can be used to retrieve the modulus of a congruence class or set of congruence classes (this is the recommended approach):

```
>>> c = modulo(3, 7)
>>> int(c)
3
>>> len(c)
7
```

Congruence classes and sets of congruence classes are also hashable (making it possible to use them as dictionary keys and as set members) and iterable:

```
>>> len({mod(0, 3), mod(1, 3), mod(2, 3)})
3
>>> list(mod(4))
[modulo(0, 4), modulo(1, 4), modulo(2, 4), modulo(3, 4)]
>>> from itertools import islice
>>> list(islice(mod(3, 7), 5))
[3, 10, 17, 24, 31]
```

The Chinese remainder theorem can be applied to construct the intersection of two congruence classes as a congruence class (when it is possible to do so):

```
>>> mod(23, 100) & mod(31, 49)
modulo(423, 4900)
>>> mod(2, 10) & mod(4, 20) is None
True
```

Some familiar forms of notation for referring to congruence classes (and sets thereof) are also supported:

```
>>> Z/(23*Z)
modulo(23)
>>> 23*Z
modulo(0, 23)
>>> 17 + 23*Z
modulo(17, 23)
```

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 --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):

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

Alternatively, all unit tests are included in the module itself and can be executed using doctest:

```
python src/modulo/modulo.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/modulo
```

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 0.2.0, 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 modulo module

Pure-Python library for working with modular arithmetic, congruence classes, and finite fields.

class modulo.modulo.**modulo**(**args: Union[int, Sequence[int]]*)

Bases: `object`

Class for representing both *individual congruence classes* (*e.g.*, finite field elements) and *sets of congruence classes* (*e.g.*, rings and finite fields such as **Z**/7**Z**). Common arithmetic and membership operations are supported for each, as appropriate.

When two integer arguments are supplied, the created instance represents the individual congruence class corresponding to the remainder of the first argument modulo the second argument. The instance `modulo(3, 7)` in the example below represents the congruence class 3 in the set **Z**/7**Z**. Note that **the synonym *mod* is made available to support more concise notation and is used throughout this documentation.**

```
>>> mod(3, 7)
modulo(3, 7)
```

Common modular arithmetic operations and the membership operator (via the special method *modulo.__contains__*) are supported for congruence class instances.

```
>>> mod(3, 7) + mod(2, 7)
modulo(5, 7)
>>> mod(0, 7) - mod(1, 7)
modulo(6, 7)
>>> mod(3, 7) * mod(2, 7)
modulo(6, 7)
>>> mod(3, 7) ** (-1)
modulo(5, 7)
>>> mod(3, 7) in mod(7)
True
>>> int(mod(3, 7))
3
>>> 3 in mod(3, 7)
True
>>> 10 in mod(3, 7)
True
>>> 4 in mod(3, 7)
False
```

Individual congruence classes can be compared with one another according to their least nonnegative residues (and, thus, can also be sorted).

```
>>> mod(2, 7) < mod(3, 7)
True
>>> list(sorted([mod(2, 3), mod(1, 3), mod(0, 3)]))
[modulo(0, 3), modulo(1, 3), modulo(2, 3)]
```

When one integer argument is supplied, the created instance represents the set containing the congruence classes modulo that integer. The instance `mod(7)` in the example below represents the set $\mathbf{Z}/7\mathbf{Z}$.

```
>>> len(mod(7))
7
```

Use of the membership operation is also supported for individual congruence classes that are themselves members of a set of congruence classes.

```
>>> mod(3, 7) in mod(7)
True
>>> mod(1, 2) in mod(7)
False
```

The built-in `int` function can be used to retrieve the least nonnegative residue of an instance (see *modulo.__int__*) and the built-in `len` function can be used to retrieve the modulus of an instance (see *modulo.__len__*).

```
>>> c = modulo(3, 7)
>>> int(c)
3
>>> len(c)
7
```

Congruence classes and sets of congruence classes are also hashable (making it possible to use them as dictionary

keys and as set members) and iterable.

```
>>> list(mod(4))
[modulo(0, 4), modulo(1, 4), modulo(2, 4), modulo(3, 4)]
>>> len({mod(0, 3), mod(1, 3), mod(2, 3)})
3
>>> from itertools import islice
>>> list(islice(mod(3, 7), 5))
[3, 10, 17, 24, 31]
```

The Chinese remainder theorem can be applied to construct the intersection of two congruence classes as a congruence class (when it is possible to do so).

```
>>> mod(23, 100) & mod(31, 49)
modulo(423, 4900)
>>> mod(2, 10) & mod(4, 20) is None
True
```

Special methods such as *modulo.__getitem__* and synonyms such as *Z* make it possible to use a number of different forms of notation for creating congruence classes and sets thereof.

```
>>> Z/(23*Z)
modulo(23)
>>> 23*Z
modulo(0, 23)
>>> 17 + 23*Z
modulo(17, 23)
>>> 17 % mod(23)
modulo(17, 23)
>>> cs = mod(23)
>>> cs[17]
modulo(17, 23)
```

Constructor invocations involving arguments that have incorrect types raise exceptions.

```
>>> mod()
Traceback (most recent call last):
  ...
TypeError: must provide either a modulus or an integer and a modulus
>>> mod(-2)
Traceback (most recent call last):
  ...
ValueError: modulus must be a positive integer
>>> mod(1.2, 7)
Traceback (most recent call last):
  ...
ValueError: residue must be an integer
```

__add__(*other: Union[*modulo.modulo.modulo*, int]*) → *modulo.modulo.modulo*

Perform modular addition.

```
>>> mod(1, 4) + mod(2, 4)
modulo(3, 4)
>>> mod(1, 4) + 2
modulo(3, 4)
```

Attempts to invoke the operator on arguments having incorrect types raise exceptions.

```
>>> mod(1, 3) + mod(2, 4)
Traceback (most recent call last):
  ...
ValueError: congruence classes do not have the same modulus
>>> mod(1, 3) + mod(4)
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
>>> mod(1, 3) + 'a'
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
>>> mod(4) + 2
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
```

__radd__(*other: Union[*modulo.modulo.modulo*, int]*) → *modulo.modulo.modulo*

If this instance is a congruence class, perform modular addition of congruence classes (even if the left-hand argument is an integer and/or representative of a congruence class).

```
>>> mod(1, 4) + mod(2, 4)
modulo(3, 4)
>>> 2 + mod(1, 4)
modulo(3, 4)
```

If this instance is a set of congruence classes, support use of familiar mathematical notation to construct congruence classes.

```
>>> 2 + mod(4)
modulo(2, 4)
>>> 2 + 4*Z
modulo(2, 4)
```

__sub__(*other: Union[*modulo.modulo.modulo*, int]*) → *modulo.modulo.modulo*

Perform modular subtraction.

```
>>> mod(1, 4) - mod(2, 4)
modulo(3, 4)
>>> mod(1, 4) - 3
modulo(2, 4)
```

Attempts to invoke the operator on arguments having incorrect types raise exceptions.

```
>>> mod(4) - 3
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
```

__rsub__(*other: Union[*modulo.modulo.modulo*, int]*) → *modulo.modulo.modulo*

Perform modular subtraction.

```
>>> 3 - mod(1, 4)
modulo(2, 4)
```

Attempts to invoke the operator on arguments having incorrect types raise exceptions.

```
>>> 3 - mod(4)
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
```

__pos__() → *modulo.modulo.modulo*
Identity function on congruence classes.

```
>>> +mod(4, 7)
modulo(4, 7)
```

Any attempt to invoke the operator on an argument having an incorrect type raises an exception.

```
>>> +mod(4)
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class
```

__neg__() → *modulo.modulo.modulo*
Return the additive inverse of a congruence class.

```
>>> -mod(4, 7)
modulo(3, 7)
```

Any attempt to invoke the operator on an argument having an incorrect type raises an exception.

```
>>> -mod(4)
Traceback (most recent call last):
  ...
TypeError: can only negate a congruence class
```

__mul__(*other: Union[*modulo.modulo.modulo, *int]*) → *modulo.modulo.modulo*
Perform modular multiplication.

```
>>> mod(1, 4) * mod(2, 4)
modulo(2, 4)
>>> mod(2, 7) * 3
modulo(6, 7)
```

Attempts to invoke the operator on arguments having incorrect types raise exceptions.

```
>>> mod(7) * 3
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
```

__rmul__(*other: Union[*modulo.modulo.modulo, *int]*) → *modulo.modulo.modulo*
Perform modular multiplication.

```
>>> 3 * mod(2, 7)
modulo(6, 7)
```

Attempts to invoke the operator on arguments having incorrect types raise exceptions.

```
>>> 3 * mod(7)
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
```

__floordiv__(*other: Union[*modulo.modulo.modulo*, int]*) → *modulo.modulo.modulo*

Perform modular division (*i.e.*, multiplication by the inverse).

```
>>> mod(4, 7) // mod(2, 7)
modulo(2, 7)
>>> mod(6, 17) // mod(3, 17)
modulo(2, 17)
```

Any attempt to divide by a congruence class that is not invertible – or to invoke the operator on arguments that have incorrect types – raises exceptions.

```
>>> mod(17) // mod(3, 17)
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or integer
>>> mod(4, 6) // mod(2, 6)
Traceback (most recent call last):
  ...
ValueError: congruence class has no inverse
```

__pow__(*other: Union[int,* modulo.modulo.modulo*], modulo: Optional[int] = None*) → *modulo.modulo.modulo*

Perform modular exponentiation (including inversion, if the supplied exponent is negative).

```
>>> mod(4, 7) ** 3
modulo(1, 7)
>>> mod(4, 7) ** (-1)
modulo(2, 7)
>>> mod(4, 7) ** (-2)
modulo(4, 7)
>>> pow(mod(4, 7), 3)
modulo(1, 7)
>>> pow(mod(4, 7), 3, 7)
modulo(1, 7)
```

The exponent can be an integer or an instance of *modulo*. The latter option can enable concise notation when exponent values are treated as elements within their own group (such as when leveraging Euler's theorem).

```
>>> pow(mod(4, 7), mod(2, 6)) * pow(mod(4, 7), mod(4, 6))
modulo(1, 7)
```

Attempts to invoke the operator on arguments that lack required properties (*e.g.*, congruence classes that are not invertible) – or that have incorrect types – raise an exception.

```
>>> pow(mod(7), 3)
Traceback (most recent call last):
  ...
TypeError: can only exponentiate a congruence class
>>> pow(mod(3, 7), 'a')
Traceback (most recent call last):
  ...
TypeError: exponent must be an integer or congruence class
>>> pow(mod(4, 7), 3, 8)
Traceback (most recent call last):
  ...
ValueError: modulus does not match congruence class modulus
>>> pow(mod(4, 6), -1, 6)
Traceback (most recent call last):
  ...
ValueError: congruence class has no inverse
```

__invert__() → *modulo.modulo.modulo*

Return the multiplicative inverse of a congruence class.

```
>>> ~mod(4, 7)
modulo(2, 7)
```

Any attempt to invoke the operator on an instance that lacks the required properties (*e.g.*, a congruence class that is not invertible) raises an exception.

```
>>> ~mod(4, 6)
Traceback (most recent call last):
  ...
ValueError: congruence class has no inverse
```

__mod__(*other: int*) → *modulo.modulo.modulo*

If this instance is a congruence class, return a congruence class with a modified modulus attribute.

```
>>> mod(3, 10) % 7
modulo(3, 7)
>>> mod(11, 23) % 2
modulo(1, 2)
```

This operation is only defined for congruence classes and the second argument must be an integer.

```
>>> mod(10) % 2
Traceback (most recent call last):
  ...
ValueError: modulus cannot be modified for a set of congruence classes
>>> mod(3, 10) % 1.23
Traceback (most recent call last):
  ...
TypeError: right-hand argument must be an integer
```

__rmod__(*other: int*) → *modulo.modulo.modulo*

If this instance is a set of congruence classes, construct a congruence class corresponding to the supplied integer.

```
>>> 7 % mod(11)
modulo(7, 11)
```

This operation is only defined for a set of congruence classes.

```
>>> 7 % mod(3, 11)
Traceback (most recent call last):
  ...
ValueError: expecting a set of congruence classes as the second argument
>>> 1.23 % mod(11)
Traceback (most recent call last):
  ...
TypeError: left-hand argument must be an integer
```

__truediv__(*other: int*) → *modulo.modulo.modulo*

Transform a congruence class into a related congruence class that is obtained by dividing both the residue and the modulus by the same positive integer.

```
>>> mod(2, 10) / 2
modulo(1, 5)
```

Only a congruence class can be transformed, and both the residue and modulus must be divisible by the supplied integer.

```
>>> mod(4) / 2
Traceback (most recent call last):
  ...
ValueError: can only transform a congruence class
>>> mod(3, 4) / 2
Traceback (most recent call last):
  ...
ValueError: residue and modulus must both be divisible by the supplied integer
>>> mod(3, 9) / 3.0
Traceback (most recent call last):
  ...
TypeError: second argument must be an integer
```

This method is made available primarily for use in applying the Chinese remainder theorem (*e.g.*, as is done in *modulo.__and__*) and similar processes.

__and__(*other:* modulo.modulo.modulo) → Optional[Union[*modulo.modulo.modulo*, set]]

Return the intersection of two congruence classes, represented as a congruence class. The result is constructed via an application of the Chinese remainder theorem).

```
>>> mod(2, 3) & mod(4, 5)
modulo(14, 15)
>>> mod(1, 10) & mod(1, 14)
modulo(1, 70)
>>> mod(2, 10) & mod(2, 14)
modulo(2, 70)
>>> mod(23, 100) & mod(31, 49)
modulo(423, 4900)
>>> mod(2, 10) & mod(4, 20) is None
True
```

The example below compares the outputs from this method (across a range of inputs) with the results of an exhaustive search. Note the use of congruence class instances as iterables (via *modulo.__iter__*).

```
>>> from itertools import islice
>>> all(
...     int(modulo(a, m) & modulo(b, n)) in (
...         set(islice(modulo(a, m), 20)) & set(islice(modulo(b, n), 20))
...     )
...     for m in range(1, 20) for a in range(m)
...     for n in range(1, 20) for b in range(n)
...     if (a % gcd(m, n) == b % gcd(m, n))
... )
True
>>> all(
...     (modulo(a, m) & modulo(b, n) is None) and (
...         set(islice(modulo(a, m), 20)) & set(islice(modulo(b, n), 20)) ==␣
↪set()
...     )
...     for m in range(1, 20) for a in range(m)
...     for n in range(1, 20) for b in range(n)
...     if (a % gcd(m, n) != b % gcd(m, n))
... )
True
```

Both arguments must be congruence classes.

```
>>> mod(2, 3) & mod(7)
Traceback (most recent call last):
  ...
ValueError: intersection operation is only defined for two congruence classes
>>> mod(2, 3) & 1.23
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class
```

__eq__(*other:* modulo.modulo.modulo) → bool

Return a boolean value indicating whether this instance represents the same congruence class or set of congruence classes as the other instance.

```
>>> mod(3, 7) == mod(3, 7)
True
>>> mod(2, 7) == mod(3, 7)
False
>>> mod(4) == mod(4)
True
>>> mod(5) == mod(7)
False
```

Both arguments must be congruence classes or sets thereof.

```
>>> modulo(2, 3) == 2
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or set of congruence classes
```

__ne__(*other:* modulo.modulo.modulo) → bool

Return a boolean value indicating whether this instance represents a different congruence class (or set of congruence classes) than the other instance.

```
>>> mod(2, 7) != mod(3, 7)
True
>>> mod(3, 7) != mod(3, 7)
False
>>> mod(5) != mod(7)
True
>>> mod(4) != mod(4)
False
```

Both arguments must be congruence classes or sets thereof.

```
>>> modulo(2, 3) != 2
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class or set of congruence classes
```

__lt__(*other:* modulo.modulo.modulo) → bool

Allow comparison and sorting of congruence classes (according to their least nonnegative residues).

```
>>> mod(2, 7) < mod(3, 7)
True
>>> mod(3, 7) < mod(3, 7)
False
>>> mod(5, 7) < mod(3, 7)
False
>>> mod(9, 7) < mod(3, 7)
True
>>> list(sorted([mod(2, 3), mod(1, 3), mod(0, 3)]))
[modulo(0, 3), modulo(1, 3), modulo(2, 3)]
```

Congruence classes with different moduli, sets of congruence classes, and other incompatible objects cannot be compared.

```
>>> mod(2, 3) < mod(1, 4)
Traceback (most recent call last):
  ...
ValueError: congruence classes do not have the same modulus
>>> mod(3) < mod(5)
Traceback (most recent call last):
  ...
ValueError: sets of congruence classes cannot be compared
>>> mod(3) < 5
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class
```

__le__(*other:* modulo.modulo.modulo) → bool

Allow comparison and sorting of congruence classes (according to their least nonnegative residues).

```
>>> mod(2, 7) <= mod(3, 7)
True
>>> mod(3, 7) <= mod(3, 7)
True
>>> mod(5, 7) <= mod(3, 7)
False
>>> mod(9, 7) <= mod(3, 7)
True
>>> list(sorted([mod(2, 3), mod(1, 3), mod(0, 3)]))
[modulo(0, 3), modulo(1, 3), modulo(2, 3)]
```

__gt__(*other:* modulo.modulo.modulo) → bool

Allow comparison and sorting of congruence classes (according to their least nonnegative residues).

```
>>> mod(3, 7) > mod(2, 7)
True
>>> mod(3, 7) > mod(3, 7)
False
>>> mod(1, 7) > mod(3, 7)
False
>>> mod(3, 7) > mod(9, 7)
True
>>> list(sorted([mod(2, 3), mod(1, 3), mod(0, 3)]))
[modulo(0, 3), modulo(1, 3), modulo(2, 3)]
```

__ge__(*other:* modulo.modulo.modulo) → bool

Allow comparison and sorting of congruence classes (according to their least nonnegative residues).

```
>>> mod(3, 7) >= mod(2, 7)
True
>>> mod(3, 7) >= mod(3, 7)
True
>>> mod(1, 7) >= mod(3, 7)
False
>>> mod(3, 7) >= mod(9, 7)
True
>>> list(sorted([mod(2, 3), mod(1, 3), mod(0, 3)]))
[modulo(0, 3), modulo(1, 3), modulo(2, 3)]
```

__contains__(*other: Union[*modulo.modulo.modulo*, int]*) → bool

Membership function for integers, congruence classes, and sets of congruence classes.

```
>>> 3 in mod(4, 7)
False
>>> 4 in mod(4, 7)
True
>>> 11 in mod(4, 7)
True
>>> mod(4, 7) in mod(7)
True
>>> mod(4, 5) in mod(7)
False
```

Attempts to perform invalid membership checks raise exceptions.

```
>>> 3 in mod(4)
Traceback (most recent call last):
  ...
TypeError: can only check if a congruence class is a member of a set of␣
↪congruence classes
>>> mod(4) in mod(4)
Traceback (most recent call last):
  ...
TypeError: can only check if a congruence class is a member
>>> 'a' in mod(7)
Traceback (most recent call last):
  ...
TypeError: can only check if a congruence class is a member of a set of␣
↪congruence classes
>>> 'a' in mod(4, 7)
Traceback (most recent call last):
  ...
TypeError: can only check if an integer is a member of a congruence class
```

issubset(*other:* modulo.modulo.modulo) → bool

Return a boolean value indicating whether a congruence class of integers is a subset of another congruence class of integers.

```
>>> mod(2, 8).issubset(mod(2, 4))
True
>>> mod(6, 8).issubset(mod(2, 4))
True
>>> mod(3, 4).issubset(mod(0, 2))
False
```

Only pairs of congruence classes can be compared using this method.

```
>>> mod(6).issubset(mod(2, 4))
Traceback (most recent call last):
  ...
ValueError: subset relationship is only defined between congruence classes
>>> mod(2, 8).issubset(mod(4))
Traceback (most recent call last):
  ...
ValueError: subset relationship is only defined between congruence classes
>>> mod(2, 8).issubset(4)
Traceback (most recent call last):
  ...
TypeError: expecting a congruence class
```

__iter__() → Union[Iterable[int], Iterable[*modulo*]]

Allow iteration over all nonnegative (infinitely many) members (if this instance is a congruence class), or all congruence classes (if this instance is a set of congruence classes).

```
>>> [i for (_, i) in zip(range(5), mod(3, 7))]
[3, 10, 17, 24, 31]
>>> list(mod(4))
[modulo(0, 4), modulo(1, 4), modulo(2, 4), modulo(3, 4)]
```

__getitem__(*index: int*) → Union[*modulo.modulo.modulo*, int]

Allow efficient retrieval of individual members of a congruence class or set of congruence classes.

```
>>> cs = modulo(7)
>>> cs[2]
modulo(2, 7)
>>> cs[-2]
modulo(5, 7)
>>> c = modulo(2, 7)
>>> c[0]
2
>>> c[-1]
-5
>>> c[2]
16
```

The supplied index must be an integer.

```
>>> c['a']
Traceback (most recent call last):
  ...
TypeError: index must be an integer
```

__len__() → int

Return the number of elements in a set of congruence classes (*e.g.*, a ring or finite field) or the modulus for a congruence class.

```
>>> len(mod(36))
36
>>> len(mod(2, 4))
4
```

Use of the built-in `len` function is the recommended approach for retrieving the `modulus` attribute of a *modulo* instance.

__int__() → int

Return the least nonnegative residue (*i.e.*, the canonical integer representative) of an instance that represents a congruence class.

```
>>> int(mod(2, 4))
2
```

A set of congruence classes (*e.g.*, a finite field) cannot be represented as a single integer.

```
>>> int(mod(4))
Traceback (most recent call last):
  ...
TypeError: can only convert a congruence class to an integer
```

Use of the built-in `int` function is the recommended approach for retrieving the `residue` attribute of a *modulo* instance.

__repr__() → str

Return the string representation of this congruence class or set of congruence classes.

```
>>> mod(2, 4)
modulo(2, 4)
>>> mod(7)
modulo(7)
```

__str__() → str
Return the string representation of this congruence class or set of congruence classes.

```
>>> mod(2, 4)
modulo(2, 4)
>>> mod(7)
modulo(7)
```

modulo.modulo.**mod**
Alias for *modulo*.

modulo.modulo.**Z**
Alias for *modulo*.

PYTHON MODULE INDEX

INDEX

Symbols

I

M

Z