gf Documentation

Release 0.2.4

Gerald Klix

Apr 21, 2021

CONTENTS

Contents:

CHAPTER

ONE

OVERVIEW

gf lets you write generic functions generic functions with multi-methods, that dispatch on all their arguments.

1.1 Simple Example

```
>>> from gf import generic, method
>>> add = generic()
>>> type(add)
<class 'function'>
```

Lets define *add* for two integers:

```
>>> @method()
... def add(n0: int, n1: int):
...     return n0 + n1
```

Lets test it:

```
>>> add(1, 2)
3
```

Calling *add* with instances of other types fails:

```
>>> add("Hello ", "World")
Traceback (most recent call last):
...
NotImplementedError: Generic '__main__.add' has no implementation for type(s): __
↪builtin__.str, __builtin__.str
```

Of course *add* can also by defined for two strings:

```
>>> @method()
... def add(s0: str, s1: str):
...     return s0 + s1
```

Now our hello world example works:

```
>>> add("Hello ", "World")
'Hello World'
```

add can also be defined for a string and an integer:

```
>>> @method()
... def add(s: str, n: int):
...     return s + str(n)
```

Thus we can add a string and a number:

```
>>> add("You ", 2)
'You 2'
```

1.2 Python's Special Methods

`gf.Object` implements (nearly) all of the special instance methods of a python object as a generic function. The package includes a rational number implementation that makes heavy use of this feature:

```
@method()
def __add__(a: object, b: Rational):
    """Add an object and a rational number.

    `a` is converted to a :class:`Rational` and then both are added."""
    return Rational(a) + b

@method(Rational, object)
def __add__(a: Rational, b: object):
    """Add a rational number and an object.

    `b` is converted to a :class:`Rational` and then both are added."""
    return a + Rational(b)
```

`gf.Object` also has a more Smalltalk means of overwriting `object.__str__()` and `object.__repr__()` using a file like object. Again the rational example is instructive about its usage.

```
@method()
def __out__(rational: Rational, writer: Writer):
    """Write a nice representation of the rational.

    Denominators that equal 1 are not printed."""
    writer("%d", rational.numerator)
    if rational.denominator != 1:
        writer(" / %d", rational.denominator)

@method()
def __spy__(rational: Rational, writer: Writer):
    """Write a debug representation of the rational."""
    writer("%s(", rational.__class__.__name__)
    if rational.numerator != 0:
            writer("%r", rational.numerator)
            if rational.denominator != 1:
                writer(", %r", rational.denominator)
    writer(")")
```

1.3 Installation

As usual *gf3* can be installed with *pip*, like this:

```
pip install gf3
```

1.4 Documentation

The whole documentation is available at in the following formats

HTML http://gf3.klix.ch (Also servers as *gf*'s homepage)

PDF http://gf3.klix.ch/gf3.pdf

1.5 Changes

A short sketch of the changes done in each release.

1.5.1 Release 0.2.4

The following was changed in Release 0.2.4:

- The `push()`-method accepts an identation string for identing writers.
- The methods `push()` and `pop()` now accept arbitrary arguments in the general case.
- Successfully tested the whole framework with Python 3.5.

1.5.2 Release 0.2.3

The following was changed in Release 0.2.3:

- Fixed the long description.
- Wrote some documentation about changing the implementation class of a generic function.

1.5.3 Release 0.2.2

The following was changed in Release 0.2.2:

- Write more documentation. Especially documented the *merge* and the *isgeneric* functions.
- Consistency between the long text and on PyPi and the documentation.

1.5.4 Release 0.2.1

Needed to bump the version information, because the homepage in the package-information was wrong[1] and a new upload was needed.

1.5.5 Release 0.2.0

The following was changed in Release 0.2.0:

- Ported the whole module to Python 3.6 and Python 3.7.
- Exclusively uses parameter annotations to specify the types to dispatch on.
- Added standard conforming default implementations for methods like `__add__()`. All these methods now raise a proper *TypeError* instead of raising a *NotImplementedError*.
- Added some means to write generic functions that dispatch types only. This is the generic function equivalent of a class-method.

[1] Silly me discovered about the shutdown of pythonhosted.org after version 0.2.0 was uploaded.

- Added some means to dispatch on single objects. This is the equivalent adding methods to class-instances[2].
- The package name for PyPi is now `gf3`.

1.5.6 Release 0.1.4

The following was fixed in Release 0.1.4:

- Fixed an issue with variadic methods. Sometimes definitions of variadic methods added after the method was already called where not added.
- Specified and implemented a precedence rule for overlapping variadic methods of generic functions.
- Improved generated documentation for variadic methods.
- Fixed the markup of some notes in the documentation.

1.5.7 Release 0.1.3

The following was changed in Release 0.1.3:

- Added variadic methods, e.g. multi-methods with a variable number of arguments.
- Improved the long description text a bit and fixed bug in its markup.
- Fixed invalid references in the long description.

1.5.8 Release 0.1.2

The following was changed in Release 0.1.2:

- Added a generic functions for `gf.Object.__call__()`.
- Added a *gf.go.FinalizingMixin*.
- *gf.generic()* now also accepts a type.
- Improved the exception information for ambiguous calls.
- Fixed some documentation glitches.

1.5.9 Release 0.1.1

This was the initial release.

[2] Of course this is not possible with standard python classes and their instances.

CHAPTER

TWO

APPLICATION PROGRAMMING INTERFACE

The generic function package provides means to define generic functions and multi-methods. Additionally classes are provided that enable the user to implement nearly all of Python's special methods as multi-methods.

2.1 Basic Usage

One can define generic functions are generics and multi-methods with Python`s special method support with juts two decorator functions and optionally one decorator method.

2.1.1 Defining Generic Functions

Generic functions must be defined with the *generic()*-function.

`@gf.`**`generic`**`(`*default_function*`)`

Create a generic function with a default implementation provided by *default_function*.

Parameters **`default_function`** (*callable_object*) – The generic's default implementation.

The generic's name and docstring are taken from the *default_function*.

Specialisations for different call signatures can be added with the *method()* and `<generic>.method()` decorators.

Note: *callable_object* can be a function or a type (a new style class).

For example the generic `foo()`'s default implemenations just answers the arguments passed as a tuple:

```
>>> from gf import generic
>>> @generic
... def foo(*arguments):
...     """Answers its arguments."""
...     return arguments
```

`foo()` can be called just like an ordinary function.

```
>>> foo(1, 2, 3)
(1, 2, 3)
```

`gf.`**`generic`**`()`

Create an unnamed generic function with no default function and no implementation.

Defining a generic function in this way has the same effect as defining a generic function with a default function that raises a *NotImplementedError*.

This form is the simplest way to define a generic:

```
>>> bar = generic()
```

If this generic function is called a *NotImplementedError* is raised:

```
>>> bar("Hello", "World")
Traceback (most recent call last):
...
NotImplementedError: Generic None has no implementation for type(s): builtins.
↪str, builtins.str
```

Note: The name is added later when the first multi-method is added with *method()*.

gf.**generic**(*name*)

Create a generic function with a name and no default implementation.

Parameters **name** (*str*) – The generic's name accessible with the *__name__* attribute.

If you define `bar()` in this way, a *NotImplementedError* raised will contain the generics name:

```
>>> bar = generic("bar")
>>> bar("Hello", "World")
Traceback (most recent call last):
...
NotImplementedError: Generic 'bar' has no implementation for type(s): builtins.
↪str, builtins.str
```

The docstring however is still *None*:

```
>>> print(bar.__doc__)
None
```

gf.**generic**(*name*, *doc*)

Create a generic function with a name and a docstring, but no default implementation.

Parameters

- **name** (*str*) – The generic's name accessible with the *.__name__* attribute.
- **doc** (*str*) – The generic's docstring accessible with the *.__doc__* attribute.

```
>>> bar = generic("bar", "A silly generic function for demonstration purposes")
```

The generic now also has a docstring:

```
>>> print(bar.__doc__)
A silly generic function for demonstration purposes
```

2.1.2 Adding Multi-Methods

Multi-methods can be added to a generic function with the *method()*-function

or the `<generic>.method()` method.

@gf.**method**()(*implementation_function*)

Add a multi-method for the types given in the *implementation_function*s type hints to the generic decorated.

Parameters **implementation_function** (*FunctionType*) – The function implementing the multi-method.

A multi-method specialising the `foo()` generic for two integers can be added as such:

```
>>> from gf import method
>>> @method()
... def foo(i0: int, i1: int):
...     return i0 + i1
```

This makes `foo()` answer 3 for the following call:

```
>>> foo(1, 2)
3
```

As you can see the types to dispatch on are defined using type hints as defined by PEP 484. This contrasts with the Python 2 version of GF, where types have to be passed as arguments to the decorator.

Caution: The generic function the multi-method is added to, must be defined in the multi-method's implementation function's global name-space.

If this is not the case use the `<generic>.method()` decorator.

`@gf.`**`variadic_method`**`()`*(implementation_function)*

Add a multi-method with a variable number of arguments to the generic decorated.

Parameters **`implementation_function`** (*`FunctionType`*) – The function implementing the multi-method.

This does essentially the same as *`method()`*, but accepts additional arguments to ones the specified by the parameter annotations. This is done by virtually adding an infinite set of of method defintions with the type `object` used for the additional arguments.

This decorator can be used to implement functions like this:

```
@variadic_method()
def varfun1(tc: TC, *arguments):
    return (tc,) + tuple(reversed(arguments))
```

This function can be called like this:

```
varfun1(to, "a", "b", "c")
```

and behaves in this case like being defined as:

```
@method()
def varfun1(tc: TC, o1: object, o2: object: o3: object, o4: object):
    return (tc,) + (o4, o3, o2, o1)
```

Overlaps of variadic method definitions with non-variadic method definitions are always resolved towards the non-variadic method with the explicitly specified types.

`<generic>.method(implementation_function)`

Directly define a multi-method.

Parameters **`implementation_function`** (*`FunctionType`*) – The function implementing the multi-method.

<*generic*> needs not to be available in the implementation function's name-space. Additionally *implementation_function* can have a different name than the generic. The later leads to defining an alias of the generic function.

For example a multi-method also available as bar can be defined as such:

```
>>> @foo.method()
... def foobar(a_string: str):
...     return "<%s>" % a_string
```

With this definition one can either call `foo()` with a string as follows:

```
>>> foo("Hi")
'<Hi>'
```

Or `foobar()`:

```
>>> foobar("Hi")
'<Hi>'
```

<generic>.variadic_method()(implementation_function)

Directly define a multi-method with a variable number of arguments.

Parameters **implementation_function** (*FunctionType*) – The function implementing the multi-method.

This decorator is the variadic variant of `<generic>.method()`.

2.2 Advanced Usage

The follows section describe advanced uses cases of *gf*.

2.2.1 Calling Other Multi-Methods of The Same Generic

gf implements two ways to calls other methods of the same generic.

New Style of `super` Calls

The new way of calling other methods with the same arity of the same generic (mis)uses the built-in `super` type.

`super` can be used to define `foo()` for integers:

```
>>> @method()
... def foo(an_integer: int):
...     return foo(super(str, str(an_integer)))
```

Calling `foo()` with an integer now works as expected:

```
>>> foo(42)
'<42>'
```

Old Style of `super` Calls

Note: This way of calling other methods of a generic function is deprecated.

As it is sometimes necessary with ordinary single dispatch methods to call methods defined in a base class, it it is sometimes necessary to reuse other implementations of a generic. For this purpose the generic has `super()`-method.

Deprecated since version 0.2.

<generic>.super(*types)(*arguments)

Parameters

- **types** (*type*) – An optionally empty list of built-in types or new-style classes.
- **arguments** – An optionally empty list of Python objects.

Directly retrieve and call the multi-method that implements the generic's functionally for *types*.

One can add a (silly) implementation for string objects to `foo()` like this:

```
>>> @method()
... def foo(an_integer: int):
...     return foo.super(str)(an_integer)
```

With this definition the generic's default implementation will be called for *float*-objects:

```
>>> foo(42.0)
(42.0,)
```

While calling `foo()` with an integer yields a formatted string:

```
>>> foo(42)
'<42>'
```

Caution: It is not checked whether the *arguments* passed are actually instances of *types*. This is consistent with Python's notion of duck typing.

2.2.2 Dispatch on Instances

Since version 0.2 *gf*'s default dispatch algorithm dispatches on single instances, too:

```
>>> silly = generic('silly')
>>> @method()
... def silly(bla: str):
...     return '<S|' + bla + '|S>'
```

Thus we can call `silly()` with a string:

```
>>> silly('Hello')
'<S|Hello|S>'
```

But we can also define `silly()` for two integers:

```
>>> @method()
... def silly(hitchhiker: 42, star_trek: 47):
...     return 'Bingo'
```

Now we can call `silly()` with exactly the right numbers:

```
>>> silly(42, 47)
'Bingo'
```

But calling silly with others fails like this:

```
>>> silly(21, 21)
Traceback (most recent call last):
...
NotImplementedError: Generic 'silly' has no implementation for type(s): builtins.
↪int, builtins.int
```

The old behavior can be achieved by using the dispatch type `Dispatch.ON_CLASS`.

```
>>> from gf import Dispatch
>>> even_sillier = generic(Dispatch.ON_CLASS)
>>> @method()
```

(continues on next page)

(continued from previous page)

```
... def even_sillier(bla: str):
...     return '<SS|' + bla + '|SS>'
```

Thus we can call `even_sillier()` with a string:

```
>>> even_sillier('Hello')
'<SS|Hello|SS>'
```

But we can't define `even_sillier()` for two integers:

```
>>> @method()
... def even_sillier(hitchhiker: 42, star_trek: 47):
...     return 'Bingo'
Traceback (most recent call last):
...
TypeError: Can't dispatch on instances in state: Dispatch.ON_CLASS
```

Note: The enumeration `Dispatch` also has the option `ON_OBJECT`, which is the new default for generic functions.

2.2.3 Class-Generics

Since release 0.2 *gf* has some means to define the equivalent of a class function with generic functions. This capability is defined on a per *generic()* basis and not – as one might expect – on a per *method()* basis.

The following example will explain this feature:

```
>>> cm = generic(
...     'cm',
...     """A class method (sort of)""",
...     Dispatch.ON_OBJECT_AND_CLASS_HIERARCHY)
```

One now can add methods to `cm()` that dispatch on the class passed:

```
>>> @method()
... def cm(integer: int):
...     return 'Integer'
>>> @method()
... def cm(string: str):
...     return 'String'
```

This way we can dispatch on classes like:

```
>>> cm(int)
'Integer'
>>> cm(str)
'String'
>>> cm(1)
'Integer'
>>> cm('Sepp')
'String'
```

With the same dispatch type it is also possible to dispatch on instances:

```
>>> @method()
... def cm(wow: 42):
...     return 'Jackpot'
>>> cm(42)
```

(continues on next page)

(continued from previous page)

```
'Jackpot'
>>> cm(4711)
'Integer'
```

The normal – and also the pre version 0.2 – behavior is to dispatch on the type of an instance only:

```
>>> im = generic(
...     'im',
...     """An instance method""",
...     Dispatch.ON_OBJECT)
```

One now can add methods to `im()` that dispatch on the class of an instance passed as argument:

```
>>> @method()
... def im(integer: int):
...     return 'Integer'
>>> @method()
... def im(string: str):
...     return 'String'
```

But we can't dispatch on class arguments:

```
>>> im(int)
Traceback (most recent call last):
...
NotImplementedError: Generic 'im' has no implementation for type(s): builtins.type
>>> im(str)
Traceback (most recent call last):
...
NotImplementedError: Generic 'im' has no implementation for type(s): builtins.type
>>> im(1)
'Integer'
>>> im('Sepp')
'String'
```

Since the type of class is `type` we can write one method to dispatch on all classes:

```
>>> @method()
... def im(cls: type):
...     return 'Class'
```

and get at least:

```
>>> im(int)
'Class'
>>> im(str)
'Class'
```

As mentioned above, it is also possible to dispatch on instances with the default dispatch type `Dispatch.ON_OBJECT`:

```
>>> @method()
... def im(wow: 42):
...     return 'Oh Baby'
>>> im(42)
'Oh Baby'
>>> im(4711)
'Integer'
```

2.2.4 Merging Generics

Two generic functions can merged into one generic function with the help of the `merge()` function like this[1]:

```
>>> g_one = generic()
>>> @method()
... def g_one(a: 1):
...     return 'one'
>>> g_two = generic()
>>> @method()
... def g_two(a: 2):
...     return 'two'
```

Both can be called with mixed results:

```
>>> g_one(1)
'one'
>>> g_two(2)
'two'
>>> g_one(2)
Traceback (most recent call last):
...
NotImplementedError: Generic None has no implementation for type(s): builtins.int
>>> g_two(1)
Traceback (most recent call last):
...
NotImplementedError: Generic None has no implementation for type(s): builtins.int
```

Both generics can be merged into one generic by using the generic function `merge()`:

```
>>> from gf import merge
>>> g_both = merge(g_one, g_two)
>>> g_both(1)
'one'
>>> g_both(2)
'two'
```

2.2.5 Testing For Being a Generic Function

If the need arises one can test any object for being a generic function with the help of the `isgeneric()` generic function

```
>>> from gf import isgeneric
>>> isgeneric(0)
False
>>> isgeneric(object)
False
>>> isgeneric(g_one)
True
>>> isgeneric(g_both)
True
>>> isgeneric(isgeneric)
True
>>> isgeneric(generic)
True
```

[1] This functionality was necessary for one of my own projects, but my be rather useless for ordinary Python projects.

2.2.6 The Implementation Class

Behind the function generated by *generic()* there is ordinary Python class. This Python class can be access with the `get_implementation()` generic function like this:

```
>>> from gf import get_implementation
>>> get_implementation(g_one)
<gf.base.GenericFunction object at ...>
```

Of course it is also possible to provide a different implementation class when creating a generic function[2]:

```
>>> from gf.base import GenericFunction
>>> class TracingGenericFunction(GenericFunction):
...
...     def __call__(self, *args):
...         print('Calling %r with %r' % (self.name, args))
...         return super().__call__(*args)
```

A generic function with the new implementation can be generated like this:

```
>>> from gf.base import _generic
>>> tg = _generic(
...     name='tg',
...     implementation_constructor=TracingGenericFunction)
>>> get_implementation(tg)
<TracingGenericFunction object at ...>
>>> @method()
... def tg(a: int, b: int):
...     return a ** b
```

It can be invoked in the usual way:

```
>>> tg(2, 4)
Calling 'tg' with (2, 4)
16
```

For easier use of those traced generics, on should define a convenience function like this:

```
>>> tracing_generic = generic('tracing_generic')
>>> @method()
... def tracing_generic(name:str):
...     return _generic(
...             name=name,
...             implementation_constructor=TracingGenericFunction)
```

With this generic function[3] one could define `tg()` much easier like this:

```
>>> tg = tracing_generic('tg')
>>> @method()
... def tg(a: int, b: int):
...     return a ** b
```

Calling the `tg()` works like mentioned above:

```
>>> tg(2, 6)
Calling 'tg' with (2, 6)
64
```

[2] `GenericFunction` and `_generic()` are not part of the API and must there fore be imported from `gf.base`.

[3] To be really useful `tracing_generic()` should be defined for more types to enable its users to pass documentation-strings, instances of `Dispatch` and the like.

2.3 The Generic `Object`-Library

The *gf*-package also provides an abstract base class called `gf.AbstractObject` and class called `gf.Object`.

Both classes map nearly all of Python's special methods to generic functions.

There is also a `Writer`-class and some convenience and helper generics like `as_string()`, `spy()`, `__out__()` and `__spy__()`.

The implementation of the aforementioned objects is contained in the *gf.go*, but the objects are also available for direct import from *gf*.

The following text is generated from the docstring in *gf.go*.

class gf.go.**AbstractObject**(**arguments*)

An abstract (mixin) class that maps all the python magic functions to generics.

__abs__()

Same as abs(a).

Calls the *__abs__()*-generic with its arguments.

__add__()

Same as a + b.

Calls the *__add__()*-generic with its arguments.

__and__()

Same as a & b.

Calls the *__and__()*-generic with its arguments.

__call__(**arguments*)

Call the *__call__()* generic function.

__concat__()

Same as a + b, for a and b sequences.

Calls the *__concat__()*-generic with its arguments.

__contains__()

Same as b in a (note reversed operands).

Calls the *__contains__()*-generic with its arguments.

__delitem__()

Same as del a[b].

Calls the *__delitem__()*-generic with its arguments.

__divmod__()

Return the tuple (x//y, x%y). Invariant: div*y + mod == x.

Calls the *__divmod__()*-generic with its arguments.

__eq__()

Same as a == b.

Calls the *__eq__()*-generic with its arguments.

__float__()

Convert the object to a *float* by calling the *__float__()* generic.

__floordiv__()

Same as a // b.

Calls the *__floordiv__()*-generic with its arguments.

__ge__()
Same as a >= b.

Calls the *__ge__()*-generic with its arguments.

__getitem__()
Same as a[b].

Calls the *__getitem__()*-generic with its arguments.

__gt__()
Same as a > b.

Calls the *__gt__()*-generic with its arguments.

__iadd__()
Same as a += b.

Calls the *__iadd__()*-generic with its arguments.

__iand__()
Same as a &= b.

Calls the *__iand__()*-generic with its arguments.

__iconcat__()
Same as a += b, for a and b sequences.

Calls the *__iconcat__()*-generic with its arguments.

__ifloordiv__()
Same as a //= b.

Calls the *__ifloordiv__()*-generic with its arguments.

__ilshift__()
Same as a <<= b.

Calls the *__ilshift__()*-generic with its arguments.

__imatmul__()
Same as a @= b.

Calls the *__imatmul__()*-generic with its arguments.

__imod__()
Same as a %= b.

Calls the *__imod__()*-generic with its arguments.

__imul__()
Same as a *= b.

Calls the *__imul__()*-generic with its arguments.

__index__()
Same as a.__index__()

Calls the *__index__()*-generic with its arguments.

__init__(**arguments*)
Call the *__init__()* generic function.

__int__()
Convert the object to an *int* by calling the *__int__()* generic.

__inv__()
Same as ~a.

Calls the *__inv__()*-generic with its arguments.

__invert__()
Same as ~a.

Calls the *__invert__()*-generic with its arguments.

__ior__()
Same as a |= b.

Calls the *__ior__()*-generic with its arguments.

__ipow__()
Same as a **= b.

Calls the *__ipow__()*-generic with its arguments.

__irshift__()
Same as a >>= b.

Calls the *__irshift__()*-generic with its arguments.

__isub__()
Same as a -= b.

Calls the *__isub__()*-generic with its arguments.

__iter__()
iter(iterable) -> iterator iter(callable, sentinel) -> iterator

Get an iterator from an object. In the first form, the argument must supply its own iterator, or be a sequence. In the second form, the callable is called until it returns the sentinel.

Calls the *__iter__()*-generic with its arguments.

__itruediv__()
Same as a /= b.

Calls the *__itruediv__()*-generic with its arguments.

__ixor__()
Same as a ^= b.

Calls the *__ixor__()*-generic with its arguments.

__le__()
Same as a <= b.

Calls the *__le__()*-generic with its arguments.

__lshift__()
Same as a << b.

Calls the *__lshift__()*-generic with its arguments.

__lt__()
Same as a < b.

Calls the *__lt__()*-generic with its arguments.

__matmul__()
Same as a @ b.

Calls the *__matmul__()*-generic with its arguments.

__mod__()
Same as a % b.

Calls the *__mod__()*-generic with its arguments.

__mul__()
Same as a * b.

Calls the *__mul__()*-generic with its arguments.

__ne__()
Same as a != b.

Calls the *__ne__()*-generic with its arguments.

__neg__()
Same as -a.

Calls the *__neg__()*-generic with its arguments.

__not__()
Same as not a.

Calls the *__not__()*-generic with its arguments.

__or__()
Same as a | b.

Calls the *__or__()*-generic with its arguments.

__pos__()
Same as +a.

Calls the *__pos__()*-generic with its arguments.

__pow__()
Same as a ** b.

Calls the *__pow__()*-generic with its arguments.

__radd__(*argument1*)
Same as a + b.

Calls the *__add__()*-generic with its arguments reversed.

__rand__(*argument1*)
Same as a & b.

Calls the *__and__()*-generic with its arguments reversed.

__rdivmod__(*argument1*)
Return the tuple (x//y, x%y). Invariant: div*y + mod == x.

Calls the *__divmod__()*-generic with its arguments reversed.

__repr__()
Answer the objects debug-string by calling *spy()*.

__rfloordiv__(*argument1*)
Same as a // b.

Calls the *__floordiv__()*-generic with its arguments reversed.

__rlshift__(*argument1*)
Same as a << b.

Calls the *__lshift__()*-generic with its arguments reversed.

__rmod__(*argument1*)
Same as a % b.

Calls the *__mod__()*-generic with its arguments reversed.

__rmul__(*argument1*)
Same as a * b.

Calls the *__mul__()*-generic with its arguments reversed.

__ror__(*argument1*)
Same as a | b.

Calls the *__or__()*-generic with its arguments reversed.

__rpow__(*argument1*)
Same as a ** b.

Calls the *__pow__()*-generic with its arguments reversed.

__rrshift__(*argument1*)
Same as a >> b.

Calls the *__rshift__()*-generic with its arguments reversed.

__rshift__()
Same as a >> b.

Calls the *__rshift__()*-generic with its arguments.

__rsub__(*argument1*)
Same as a - b.

Calls the *__sub__()*-generic with its arguments reversed.

__rtruediv__(*argument1*)
Same as a / b.

Calls the *__truediv__()*-generic with its arguments reversed.

__rxor__(*argument1*)
Same as a ^ b.

Calls the *__xor__()*-generic with its arguments reversed.

__setitem__()
Same as a[b] = c.

Calls the *__setitem__()*-generic with its arguments.

__str__()
Answer the objects print-string by calling *as_string()*.

__sub__()
Same as a - b.

Calls the *__sub__()*-generic with its arguments.

__truediv__()
Same as a / b.

Calls the *__truediv__()*-generic with its arguments.

__weakref__
list of weak references to the object (if defined)

__xor__()
Same as a ^ b.

Calls the *__xor__()*-generic with its arguments.

class gf.go.**CSep**
A conditional separator

class gf.go.**FinalizingMixin**
A mixin class with *__del__* implemented as a generic function.

Note: This functionality was separated into mixin class, because Python's cycle garbage collector does not collect classes with a *__del__()* method. Inherit from this class if you really need *__del__()*.

__del__()
Call the *__del__()* generic function.

__weakref__
list of weak references to the object (if defined)

class gf.go.**IndentingWriter**(**arguments*)
A writer that maintains a stack of indents.

class gf.go.**Object**(**arguments*)
Just like *AbstractObject*, but can be instantiated.

class gf.go.**Writer**(**arguments*)
A simple wrapper around a file like object.

Writer's purpose is to simplify the implementation of the generics *__out__()* and *__spy__()*.

Writer instances are initialised either with a file_like object or with no arguments. In the later case ab instance of *StringIO* is used.

Output is done by simply calling the writer with at least one string object. The first argument acts as a %-template for formating the other arguments.

The class is intended to be sub-classed for formatted output.

gf.go.**__abs__**(**arguments*)
Same as abs(a).

Called by the *AbstractObject.__abs__()* special method.

gf.go.**__add__**(**arguments*)
Same as a + b.

Called by the *AbstractObject.__add__()* special method. Also called by *AbstractObject.__radd__()* with arguments reversed.

Multi methods:

gf.go.**__add__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__and__**(**arguments*)
Same as a & b.

Called by the *AbstractObject.__and__()* special method. Also called by *AbstractObject.__rand__()* with arguments reversed.

Multi methods:

gf.go.**__and__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__call__**(**arguments*)
__call__() is called when instances of *AbstractObject* are called.

Multi methods:

gf.go.**__call__**(*writer:* Writer)
Write a newline on the file-like object.

gf.go.**__call__**(*writer:* Writer, *directive: AbstractDirective*, **directives*)
Only execute directives. *This is a variadic method.*

gf.go.**__call__**(*writer:* Writer, *text: str*, **arguments*)
Write text % arguments on the file-like object. *This is a variadic method.*

gf.go.**__concat__**(**arguments*)
Same as a + b, for a and b sequences.

Called by the *AbstractObject.__concat__()* special method.

gf.go.**__contains__**(**arguments*)
Same as b in a (note reversed operands).

Called by the *AbstractObject.__contains__()* special method.

gf.go.**__del__**(**arguments*)
__del__() is called when instances of *FinalizingMixin* are about to be destroyed.

gf.go.**__delitem__**(**arguments*)
Same as del a[b].

Called by the *AbstractObject.__delitem__()* special method.

gf.go.**__divmod__**(**arguments*)
Return the tuple (x//y, x%y). Invariant: div*y + mod == x.

Called by the *AbstractObject.__divmod__()* special method. Also called by *AbstractObject.__rdivmod__()* with arguments reversed.

Multi methods:

gf.go.**__divmod__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__eq__**(**arguments*)
Same as a == b.

Called by the *AbstractObject.__eq__()* special method.

Multi methods:

gf.go.**__eq__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__float__**(**arguments*)
Convert an *AbstractObject* to a *float*.

gf.go.**__floordiv__**(**arguments*)
Same as a // b.

Called by the *AbstractObject.__floordiv__()* special method. Also called by *AbstractObject.__rfloordiv__()* with arguments reversed.

Multi methods:

gf.go.**__floordiv__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__ge__**(**arguments*)
Same as a >= b.

Called by the *AbstractObject.__ge__()* special method.

Multi methods:

gf.go.**__ge__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__getitem__**(**arguments*)
Same as a[b].

Called by the *AbstractObject.__getitem__()* special method.

gf.go.**__gt__**(**arguments*)
Same as a > b.

Called by the *AbstractObject.__gt__()* special method.

Multi methods:

gf.go.**__gt__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__iadd__**(**arguments*)
Same as a += b.

Called by the *AbstractObject.__iadd__()* special method.

Multi methods:

gf.go.**__iadd__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__iand__**(**arguments*)
Same as a &= b.

Called by the *AbstractObject.__iand__()* special method.

Multi methods:

gf.go.**__iand__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__iconcat__**(**arguments*)
Same as a += b, for a and b sequences.

Called by the *AbstractObject.__iconcat__()* special method.

gf.go.**__ifloordiv__**(**arguments*)
Same as a //= b.

Called by the *AbstractObject.__ifloordiv__()* special method.

Multi methods:

gf.go.**__ifloordiv__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__ilshift__**(**arguments*)
Same as a <<= b.

Called by the *AbstractObject.__ilshift__()* special method.

Multi methods:

gf.go.**__ilshift__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__imatmul__**(**arguments*)
Same as a @= b.

Called by the *AbstractObject.__imatmul__()* special method.

Multi methods:

gf.go.**__imatmul__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__imod__**(**arguments*)
Same as a %= b.

Called by the *AbstractObject.__imod__()* special method.

Multi methods:

gf.go.**__imod__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__imul__**(**arguments*)
Same as a *= b.

Called by the *AbstractObject.__imul__()* special method.

Multi methods:

gf.go.__imul__(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.__index__(**arguments*)
Same as a.__index__()

Called by the *AbstractObject.__index__()* special method.

gf.go.__init__(**arguments*)
__init__() initializes instantiates instances of *AbstractObject* and it's subclasses.

It has a multi method for *Object*. This multi-method does not accept any additional parameters and has no effect. There is no method for *AbstractObject*, therefore this class can not be instantiated.

Multi methods:

gf.go.__init__(*writer:* Writer)
Initialize the *Write* with a *StringIO* object.

gf.go.__init__(*writer:* Writer, *file_like: object*)
Initialize the *Write* with a file like object.

param file_like A file-like object.

gf.go.__init__(*an_object:* Object)
Do nothing for *Object*.

gf.go.__int__(**arguments*)
Convert an *AbstractObject* to an *int*.

gf.go.__inv__(**arguments*)
Same as ~a.

Called by the *AbstractObject.__inv__()* special method.

gf.go.__invert__(**arguments*)
Same as ~a.

Called by the *AbstractObject.__invert__()* special method.

gf.go.__ior__(**arguments*)
Same as a |= b.

Called by the *AbstractObject.__ior__()* special method.

gf.go.__ipow__(**arguments*)
Same as a **= b.

Called by the *AbstractObject.__ipow__()* special method.

Multi methods:

gf.go.__ipow__(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.__irshift__(**arguments*)
Same as a >>= b.

Called by the *AbstractObject.__irshift__()* special method.

Multi methods:

gf.go.__irshift__(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.__isub__(**arguments*)
Same as a -= b.

Called by the *AbstractObject.__isub__()* special method.

Multi methods:

gf.go.**__isub__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__iter__**(**arguments*)
iter(iterable) -> iterator iter(callable, sentinel) -> iterator

Get an iterator from an object. In the first form, the argument must supply its own iterator, or be a sequence. In the second form, the callable is called until it returns the sentinel.

Called by the *AbstractObject.__iter__()* special method.

gf.go.**__itruediv__**(**arguments*)
Same as a /= b.

Called by the *AbstractObject.__itruediv__()* special method.

Multi methods:

gf.go.**__itruediv__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__ixor__**(**arguments*)
Same as a ^= b.

Called by the *AbstractObject.__ixor__()* special method.

Multi methods:

gf.go.**__ixor__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__le__**(**arguments*)
Same as a <= b.

Called by the *AbstractObject.__le__()* special method.

Multi methods:

gf.go.**__le__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__lshift__**(**arguments*)
Same as a << b.

Called by the *AbstractObject.__lshift__()* special method. Also called by *AbstractObject.__rlshift__()* with arguments reversed.

Multi methods:

gf.go.**__lshift__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__lt__**(**arguments*)
Same as a < b.

Called by the *AbstractObject.__lt__()* special method.

Multi methods:

gf.go.**__lt__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__matmul__**(**arguments*)
Same as a @ b.

Called by the *AbstractObject.__matmul__()* special method.

Multi methods:

gf.go.**__matmul__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__mod__**(**arguments*)
Same as a % b.

Called by the *AbstractObject.__mod__()* special method. Also called by *AbstractObject.__rmod__()* with arguments reversed.

Multi methods:

gf.go.**__mod__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__mul__**(**arguments*)
Same as a * b.

Called by the *AbstractObject.__mul__()* special method. Also called by *AbstractObject.__rmul__()* with arguments reversed.

Multi methods:

gf.go.**__mul__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__ne__**(**arguments*)
Same as a != b.

Called by the *AbstractObject.__ne__()* special method.

Multi methods:

gf.go.**__ne__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__neg__**(**arguments*)
Same as -a.

Called by the *AbstractObject.__neg__()* special method.

gf.go.**__not__**(**arguments*)
Same as not a.

Called by the *AbstractObject.__not__()* special method.

gf.go.**__or__**(**arguments*)
Same as a | b.

Called by the *AbstractObject.__or__()* special method. Also called by *AbstractObject.__ror__()* with arguments reversed.

gf.go.**__out__**(**arguments*)
Create a print string of an object using a *Writer*.

Multi methods:

gf.go.**__out__**(*self: object*, *write:* Writer)
Write a just `str()` of self.

gf.go.**__out__**(*self:* AbstractObject, *write:* Writer)
Write a just `str()` of self by directly calling `object.__str__()`.

gf.go.**__pos__**(**arguments*)
Same as +a.

Called by the *AbstractObject.__pos__()* special method.

gf.go.**__pow__**(**arguments*)
Same as a ** b.

Called by the *AbstractObject.__pow__()* special method. Also called by *AbstractObject.__rpow__()* with arguments reversed.

Multi methods:

gf.go.**__pow__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__rshift__**(**arguments*)
Same as a >> b.

Called by the *AbstractObject.__rshift__()* special method. Also called by *AbstractObject.__rrshift__()* with arguments reversed.

Multi methods:

gf.go.**__rshift__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__setitem__**(**arguments*)
Same as a[b] = c.

Called by the *AbstractObject.__setitem__()* special method.

gf.go.**__spy__**(**arguments*)
Create a print string of an object using a *Writer*.

Note: The function's name was taken from Prolog's *spy* debugging aid.

Multi methods:

gf.go.**__spy__**(*self: object*, *write:* Writer)
Write a just `repr()` of self.

gf.go.**__spy__**(*self:* AbstractObject, *write:* Writer)
Write a just `repr()` of self by directly calling `object.__repr__()`.

gf.go.**__sub__**(**arguments*)
Same as a - b.

Called by the *AbstractObject.__sub__()* special method. Also called by *AbstractObject.__rsub__()* with arguments reversed.

Multi methods:

gf.go.**__sub__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__truediv__**(**arguments*)
Same as a / b.

Called by the *AbstractObject.__truediv__()* special method. Also called by *AbstractObject.__rtruediv__()* with arguments reversed.

Multi methods:

gf.go.**__truediv__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**__xor__**(**arguments*)
Same as a ^ b.

Called by the *AbstractObject.__xor__()* special method. Also called by *AbstractObject.__rxor__()* with arguments reversed.

Multi methods:

gf.go.**__xor__**(*o0: object*, *o1: object*)
Defaults to not comparable.

gf.go.**as_indented_string**(**arguments*)
Answer an object's indented string.

This is done by creating a *IndentingWriter* instance and calling the *__out__()* generic with the object and the writer. The using `Writer.get_text()` to retrieve the text written.

`gf.go.`**`as_string`**`(*arguments)`
Answer an object's print string.

This is done by creating a *Writer* instance and calling the *__out__()* generic with the object and the writer. The using `Writer.get_text()` to retrieve the text written.

`gf.go.`**`get_text`**`(*arguments)`
Get the text written so far.

Multi methods:

`gf.go.`**`get_text`**`(`*writer:* Writer`)`
Get the text written so far.

> **Note** This method is only supported if the file-like object implements `StringIO`'s `getvalue()` method.

`gf.go.`**`pop`**`(*arguments)`
Restore the current indent from a stack of indents.

Multi methods:

`gf.go.`**`pop`**`(`*writer:* Writer, **ignored_arguments*`)`
Pop has, like *push*, no effect on an ordinary *Writer* instance. *This is a variadic method.*

`gf.go.`**`pop`**`(`*writer:* IndentingWriter, *steps: int*`)`
Pop *steps* levels of indentation.

`gf.go.`**`pop`**`(`*writer:* IndentingWriter`)`
Pop one level of indentation.

`gf.go.`**`push`**`(*arguments)`
Push the current indent on a stack of indents.

Multi methods:

`gf.go.`**`push`**`(`*writer:* Writer, **ignored_arguments*`)`
Push has no effect on an ordinary *Writer* instance. *This is a variadic method.*

`gf.go.`**`push`**`(`*writer:* IndentingWriter, *steps: int*`)`
Push the old indent on the stack and indent.

> Indentation is *steps* times the writer's indent width.

`gf.go.`**`push`**`(`*writer:* IndentingWriter, *indent: str*`)`
Indent by using *indent*.

`gf.go.`**`push`**`(`*writer:* IndentingWriter`)`
Push the old indent and indent one indent width.

`gf.go.`**`spy`**`(*arguments)`
Answer an object's debug string.

This is done by creating a *Writer* instance and calling the *__spy__()* generic with the object and the writer. The using `Writer.get_text()` to retrieve the text written.

Note: The function's name was taken from Prolog's *spy* debugging aid.

CHAPTER

THREE

A RATIONAL NUMBERS IMPLEMENTATION AS AN EXAMPLE FOR *GF*

The following text is taken from `gf.examples.rational`'s inline documenation.

rational an Implementation of Rational Numbers

The module provides rational arithmetic. Additionally the module servers as example for the generic function package.

Usually you only need its *Rational* class:

```
>>> from rational import Rational as R
```

Rational numbers can be constructed from integers:

```
>>> r2 = R(1, 2)
>>> r1 = R(1)
>>> r0 = R()
```

Construction from arbitrary objects is not possible: >>> R("Urmel") # doctest: +IGNORE_EXCEPTION_DETAIL Traceback (most recent call last): ... NotImplementedError: Generic 'gf.go.__init__' has no implementation for type(s): rational.Rational, __builtin__.str

Rationals also have a decent string representation:

```
>>> r0
Rational()
>>> print(r0)
0
>>> r1
Rational(1)
>>> print(r1)
1
>>> r2
Rational(1, 2)
>>> print(r2)
1 / 2
```

Ordinary arithmetic works as expected:

```
>>> print(R(1, 2) + R(1, 4))
3 / 4
>>> 1 + R(1, 2)
Rational(3, 2)
>>> print(R(2) / 1000)
1 / 500
>>> print(R(-5,-10))
1 / 2
>>> print(R(5, -10))
-1 / 2
```

(continues on next page)

(continued from previous page)

```
>>> print(-R(5, -10))
1 / 2
```

Comparison also works as expected:

```
>>> R(1, 2) == R(2, 4)
True
>>> R(4, 2) == 2
True
>>> 1 == R(1, 2)
False
>>> 3 == R(10, 5)
False
>>> R(1, 2) < R(3, 4)
True
>>> R(1, 2) < 1
True
>>> R(1, 2) < 1
True
>>> R(1, 2) > R(1, 4)
True
>>> 1 > R(1, 2)
True
>>> 2 > R(10, 7)
True
>>> R(10, 2) >= R(5)
True
>>> R() != R(1)
True
>>> R() != 0
False
>>> 1 != R(1)
False
```

The `decimal` module is supported as well:

```
>>> from decimal import Decimal as D
>>> R(D("0.375"))
Rational(3, 8)
>>> R(1, 2) + D("1.5")
Rational(2)
```

Even very long decimals do work:

```
>>> R(D("7.9864829273648218372937") * 4)
Rational(7986482927364821837293, 25000000000000000000)
```

Comparisons with `decimal.Decimal` instances are also supported:

```
>>> D("1.2") == R(24, 20)
True
>>> D("1.2") >= R(23, 20)
True
>>> R(23, 20) <= D("1.2")
True
```

Rationals can also converted to floats:

```
>>> float(R(1, 4))
0.25
```

class rational.**Rational**(**arguments*)
Rational is our rational numbers class.

rational.**__add__**(**arguments*)
Same as a + b.

Called by the AbstractObject.__add__() special method. Also called by AbstractObject.__radd__() with arguments reversed.

Multi methods:

gf.go.**__add__**(*o0: object*, *o1: object*)
Defaults to not comparable.

rational.**__add__**(*a:* Rational, *b:* Rational)
Add two rational numbers.

rational.**__add__**(*a: object*, *b:* Rational)
Add an object and a rational number.

a is converted to a Rational and then both are added.

rational.**__add__**(*a:* Rational, *b: object*)
Add a rational number and an object.

b is converted to a Rational and then both are added.

rational.**__eq__**(**arguments*)
Same as a == b.

Called by the AbstractObject.__eq__() special method.

Multi methods:

gf.go.**__eq__**(*o0: object*, *o1: object*)
Defaults to not comparable.

rational.**__eq__**(*a:* Rational, *b:* Rational)
Compare to rational numbers for equality.

rational.**__eq__**(*a:* Rational, *b: object*)
Compare a rational numbers and another object for equality.

rational.**__eq__**(*a:* Rational, *b: int*)
Compare a rational numbers and an integer for equality.

Note This is an optimisation for *int*.

rational.**__float__**(**arguments*)
Convert an AbstractObject to a *float*.

Multi methods:

rational.**__float__**(*rational:* Rational)
Convert a rational to a float.

rational.**__ge__**(**arguments*)
Same as a >= b.

Called by the AbstractObject.__ge__() special method.

Multi methods:

gf.go.**__ge__**(*o0: object*, *o1: object*)
Defaults to not comparable.

rational.**__ge__**(*a:* Rational, *b:* Rational)
Answer *True* if *a* is bigger or equal than *b*.

rational.**__ge__**(*a:* Rational, *b: object*)
Answer *True* if *a* is bigger or equal than *b*.

rational.**__gt__**(**arguments*)
Same as a > b.

Called by the `AbstractObject.__gt__()` special method.

Multi methods:

gf.go.**__gt__**(*o0: object*, *o1: object*)
Defaults to not comparable.

rational.**__gt__**(*a:* Rational, *b:* Rational)
Answer *True* if *a* is bigger than *b*.

rational.**__gt__**(*a:* Rational, *b: object*)
Answer *True* if *a* is bigger than *b*.

rational.**__init__**(**arguments*)
__init__() initializes instantiates instances of `AbstractObject` and it's subclasses.

It has a multi method for `Object`. This multi-method does not accept any additional parameters and has no effect. There is no method for `AbstractObject`, therefore this class can not be instantiated.

Multi methods:

gf.go.**__init__**(*writer: Writer*)
Initialize the *Write* with a *StringIO* object.

gf.go.**__init__**(*writer: Writer*, *file_like: object*)
Initialize the *Write* with a file like object.

param file_like A file-like object.

gf.go.**__init__**(*an_object: Object*)
Do nothing for *Object*.

rational.**__init__**(*rational:* Rational, *numerator: int*, *denominator: int*, *cancel: bool*)
Initialize the object with *numerator* and *denominator*.

param rational The rational number to be initialized.

param numerator The numerator.

param denominator The denominator.

param cancel A flag indicating, that *numerator`and `denominator* should be canceled.

rational.**__init__**(*rational:* Rational, *numerator: int*, *denominator: int*)

Initialize the object with *numerator* and *denominator*.

param rational The rational number to be initialized.

param numerator The numerator.

param denominator The denominator.

Call *__init__()* with all passed arguments and with the value of *CANCEL_EAGERLY* for the *cancel*-flag.

rational.**__init__**(*rational:* Rational, *numerator: int*)

Initialize the object with *numerator*.

param rational The rational number to be initialized.

param numerator The numerator.

Call *__init__()* with the *denominator* set to 1.

rational.**__init__**(*rational:* Rational)

Initialize the object to be 0.

param rational The rational number to be initialized.

Call *__init__()* with the *numerator* set to 0.

`rational.__init__`(*rational0:* Rational, *rational1:* Rational)

Initialize the object from another rational.

param rational0 The rational number to be initialized.

param rational1 The rational number the attributes are copied from.

`rational.__init__`(*rational0:* Rational, *rational1:* Rational, *rational2:* Rational)

Initialize the object from another rational.

param rational0 The rational number to be initialized.

param rational1 The rational acting as *numerator*.

param rational2 The rational acting as *denominator*.

Call *__init__()* with *rational0* as *numerator* and *rational1* / *rational2* as *denominator*.

`rational.__init__`(*rational:* Rational, *decimal: Decimal*)

Initialize the object from a `decimal.Decimal`.

param rational The rational number to be initialized.

param decimal The decimal number the rational is initialized from.

If the *decimal*'s exponent is negative compute a scaling denominator 10 ** -exponent and initialise *rational* with the decimal scaled by the denominator and the denominator.

In the other case the *decimal* is simply converted to an *int* and used as numerator.

`rational.__le__`(**arguments*)

Same as a <= b.

Called by the `AbstractObject.__le__()` special method.

Multi methods:

`gf.go.__le__`(*o0: object*, *o1: object*)

Defaults to not comparable.

`rational.__le__`(*a:* Rational, *b:* Rational)

Answer *True* if *a* is smaller than or equal *b*.

`rational.__le__`(*a:* Rational, *b: object*)

Answer *True* if *a* is smaller than or equal *b*.

`rational.__lt__`(**arguments*)

Same as a < b.

Called by the `AbstractObject.__lt__()` special method.

Multi methods:

`gf.go.__lt__`(*o0: object*, *o1: object*)

Defaults to not comparable.

`rational.__lt__`(*a:* Rational, *b:* Rational)

Answer *True* if *a* is smaller than *b*.

`rational.__lt__`(*a:* Rational, *b: object*)

Answer *True* if *a* is smaller than *b*.

`rational.__mul__`(**arguments*)

Same as a * b.

Called by the `AbstractObject.__mul__()` special method. Also called by `AbstractObject.__rmul__()` with arguments reversed.

Multi methods:

`gf.go.__mul__`(*o0: object*, *o1: object*)
Defaults to not comparable.

`rational.__mul__`(*a:* Rational, *b:* Rational)
Multiply two rational numbers.

`rational.__mul__`(*a: object*, *b:* Rational)
Multiply an object and a rational number.

a is converted to a *Rational* and then both are multiplied.

`rational.__mul__`(*a: object*, *b:* Rational)
Multiply a rational and an object.

b is converted to a *Rational* and then both are multiplied.

`rational.__ne__`(**arguments*)
Same as a != b.

Called by the `AbstractObject.__ne__()` special method.

Multi methods:

`gf.go.__ne__`(*o0: object*, *o1: object*)
Defaults to not comparable.

`rational.__ne__`(*a:* Rational, *b:* Rational)
Compare to rational numbers for inequality.

`rational.__ne__`(*a:* Rational, *b: object*)
Compare to rational numbers for inequality.

`rational.__neg__`(**arguments*)
Same as -a.

Called by the `AbstractObject.__neg__()` special method.

Multi methods:

`rational.__neg__`(*rational:* Rational)
Negate a rational number.

`rational.__out__`(**arguments*)
Create a print string of an object using a `Writer`.

Multi methods:

`gf.go.__out__`(*self: object*, *write: Writer*)
Write a just `str()` of self.

`gf.go.__out__`(*self: AbstractObject*, *write: Writer*)
Write a just `str()` of self by directly calling `object.__str__()`.

`rational.__out__`(*rational:* Rational, *writer: Writer*)
Write a nice representation of the rational.

Denominators that equal 1 are not printed.

`rational.__spy__`(**arguments*)
Create a print string of an object using a *Writer*.

Note: The function's name was taken from Prolog's *spy* debugging aid.

Multi methods:

`gf.go.__spy__`(*self: object*, *write: Writer*)
Write a just `repr()` of self.

gf.go.__**spy**__(*self: AbstractObject*, *write: Writer*)
Write a just repr() of self by directly calling object.__repr__().

rational.__**spy**__(*rational:* Rational, *writer: Writer*)
Write a debug representation of the rational.

rational.__**sub**__(**arguments*)
Same as a - b.

Called by the AbstractObject.__sub__() special method. Also called by AbstractObject.__rsub__() with arguments reversed.

Multi methods:

gf.go.__**sub**__(*o0: object*, *o1: object*)
Defaults to not comparable.

rational.__**sub**__(*a:* Rational, *b:* Rational)
Subtract two rational numbers.

rational.__**sub**__(*a: object*, *b:* Rational)
Subtract an object and a rational number.

a is converted to a *Rational* and then both are subtracted.

rational.__**sub**__(*a:* Rational, *b: object*)
Subtract a rational number and an object.

b is converted to a *Rational* and then both are subtracted.

rational.**gcd**(*a*, *b*)
gcd() computes GCD of to numbers.

CHAPTER

FOUR

ACKNOWLEDGEMENTS

Guido van Rossum created the core of this package. I just renamed some things and added some^H^H^H^H oodles of convenience stuff. Thank you Guido!

CHAPTER

FIVE

COPYRIGHT

CHAPTER

SIX

LICENSE

Since this package is derivative work of code that was published under the PSF License Agreement, it is licensed under the same license, too.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

r

INDEX

Symbols

A

C

F

G

I

M

O

P

R

S

V

W