
ruffini Documentation

Parri Gianluca

Jun 10, 2020

Contents

1	Tutorial	3
2	Reference	7
	Index	31

Ruffini is a simple python library to compute monomials and polynomials. It's open source on github at [gianlu-parri03/ruffini](https://github.com/gianlu-parri03/ruffini).

CHAPTER 1

Tutorial

```
>>> import ruffini
```

NB: This version is still WIP. The tutorial is based on the last release.

1.1 Part One: Monomials

Ok, so: we have imported ruffini because we wanted to use monomials and polynomials in python, right? Then, let's create a monomial!

First of all we're initializing variables:

```
>>> x = ruffini.Variable('x')
```

Done. Now we can finally create a monomial!

```
>>> 3*x # that's a monomial!
3x
```

thinks

```
>>> y = ruffini.Variable('y')
>>> -5*y
-5y
```

Uhm, yea... f-funny... uhm... what could we do next? We can do operations with them:

```
>>> (3*x) + (5*x) # Sum!
8x
>>>
>>> (7*y) - (-3*y) # Subtraction!
10y
>>>
```

(continues on next page)

(continued from previous page)

```
>>> (7*y) * (2*x) # Multiplication!
14xy
>>>
>>> (7*y) ** 2 # Power!
49y**2
>>>
>>> 5*x / 3*y # Division!
Traceback (most recent call last):
...
ValueError: variable's exponent must be positive
```

ouch, we can't divide by $3y$ if there are no y in the first term... let's try another time:

```
>>> (15*x*y) / (3*y) # Division! Again!
5x
```

It worked!

We can also calculate gcd (*greatest common divisor*) or lcm (*least common multiplier*), like this:

```
>>> gcd(15*x*y, 3*x, 3)
3
>>> lcm(15*x*y, 3*x, 2*y)
30xy
```

Hmmm, what's left... oh, found.

We can also evaluate a monomial:

```
>>> monomial = 3*x
```

ok, let's think. If you know that $x = 7$, what will $3x$ be equal to? You're right, 21!

```
>>> monomial.eval(x=7)
21
```

And yes, we can also set a variable's value to a monomial:

```
>>> monomial.eval(x=(3*y)) # 3(3y) = 9y
9y
```

Nice!

NB: in this tutorial, we created monomials by doing operations with variables. We can also initialize it directly with

```
>>> ruffini.Monomial(5, x=1, y=2)
5xy**2
```

or

```
>>> ruffini.Monomial(5, {'x': 1, 'y': 2})
5xy**2
```

It's just more verbose and less readable.

Ok, I think we're done with monomials: let's jump to polynomials!

1.2 Part Two: Polynomials

Welcome back! So... we were talking about monomials, right?

When we tried sum and subtraction between monomials, the variables were the same between terms, as you can see by doing

```
>>> (3*x).similar_to(5*x)
True
```

but, what happens when you sum two monomials that aren't similar? Let's find out!

```
>>> (3*x) + (5*y)
3x + 5y
```

wow! a polynomial!

```
>>> (3*x) + (5*y) - 9
3x + 5y - 9
```

another polynomial!

“C-Can I do operations between polynomials?” Well, you can do sums, subtractions and multiplications:

```
>>> ((3*x) + (5*y)) + ((-3*y) - 2) # Sum!
3x + 2y - 2
>>>
>>> (x + 3) - (2y + 1) # Subtraction!
x + 2 - 2y
>>>
>>> ((3*x) + (5*y)) * ((3*y) - 2) # Multiplication!
9xy - 6x + 15y**2 - 10y
```

but I've not finished factorization yet, so division and power aren't legal

```
>>> (x + 2) / 4
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for /: 'Polynomial' and 'int'
```

yeah, I know, it's a sad story.

We know polynomials are a sum of monomials, so they could be considered tuples. In fact, they are. Therefore, indexing is enabled:

```
>>> (2*x + 3)[0]
2x
```

but they're immutable objects

```
>>> (2*x + 3)[0] = 9
Traceback (most recent call last):
...
TypeError: 'Polynomial' object does not support item assignment
```

If we need the coefficient of a variable(s), we can use a new method, called `term_coefficient`:

```
>>> (5*x + 2*y - 3).term_coefficient(x)
5
>>> (5*x + 2*y - 3).term_coefficient(x=1)
5
>>> (5*x + 2*y - 3).term_coefficient({'x': 1})
5
```

Yes. There is, again, a more verbous and less readable way. This time is *really* verbous. But, if you want...

```
>>> Polynomial(2*x, 5*y, -9)
2x + 5y - 9
```

more verbous!

```
>>> Polynomial(Monomial(2, x=1), Monomial(5, y=1), -9)
2x + 5y - 9
```

convinced?

Eww, as I said before, factoring isn't finished yet, so our tutorial ends here. If you have any question or everything else, you can can contact me at gianluparri03@gmail.com

2.1 Variables

2.1.1 VariablesDicts

class ruffini.**VariablesDict** (*variables=None, **kwargs*)
Bases: object

A VariablesDict is a sort of dictionary with special features, created to manage the variables of a monomial.

In this case, we'll call keys variables and values exponents.

Its main features are:

- A VariablesDict is immutable
- The default exponent is 0; therefore, variables with exponent 0 aren't stored
- Variables aren't case sensitive
- Variables must be letters from the latin alphabet and one-character long
- Exponents must be positive integer

__init__ (*variables=None, **kwargs*)

Initialize the VariablesDict by giving it the pairs variable: exponent storing them in a dict (*variables*) or as keyword arguments:

```
>>> VariablesDict({'x': 5, 'y': 3})
{'x': 5, 'y': 3}
>>> VariablesDict(x=5, y=3)
{'x': 5, 'y': 3}
```

NB: Variables are sorted

```
>>> VariablesDict(k=2, b=3)
{'b': 3, 'k': 2}
```

As said above, variables aren't case sensitive, so they're always made lowercase

```
>>> VariablesDict(X=2)
{'x': 2}
```

It also converts the exponent in integer if it's a whole number

```
>>> VariablesDict(x=3.0)
{'x': 3}
```

It can raise an error if:

- *variable* is not a string

```
>>> VariablesDict({9: 9})
Traceback (most recent call last):
...
TypeError: variable's name must be a string
```

- *variable* is too long

```
>>> VariablesDict(xy=3)
Traceback (most recent call last):
...
ValueError: variable's name length must be one
```

- *variable* is not alphabetical

```
>>> VariablesDict(x2=9)
Traceback (most recent call last):
...
ValueError: variable's name must be alphabetical
```

- *exponent* is not an integer (or a whole number)

```
>>> VariablesDict(k=[])
Traceback (most recent call last):
...
TypeError: variable's exponent must be int
```

```
>>> VariablesDict(z=7.13)
Traceback (most recent call last):
...
TypeError: variable's exponent must be a whole number
```

- *exponent* is negative

```
>>> VariablesDict(f=-3)
Traceback (most recent call last):
...
ValueError: variable's exponent must be positive
```

It also checks if the dict is empty:

```
>>> VariablesDict(a=2, b=8, c=3).is_empty
False
>>> VariablesDict(x=0).is_empty
True
```

Raise `TypeError`, `ValueError`

`__getitem__` (*variable*)

Returns the exponent for the given variable

```
>>> VariablesDict(a=2) ['a']
2
```

It returns 0 if that variable does not exists

```
>>> VariablesDict(a=2) ['b']
0
```

It can return an error if:

- *variable* is not a string

```
>>> VariablesDict({9: 9})
Traceback (most recent call last):
...
TypeError: variable's name must be a string
```

- *variable* is too long

```
>>> VariablesDict(xy=3)
Traceback (most recent call last):
...
ValueError: variable's name length must be one
```

- *variable* is not alphabetical

```
>>> VariablesDict(x2=9)
Traceback (most recent call last):
...
ValueError: variable's name must be alphabetical
```

Return type `int`

`exponents` ()

Returns the exponents of the `VariablesDict` as a tuple

```
>>> VariablesDict(a=2, b=3).exponents()
(2, 3)
```

Return type `tuple`

`items` ()

Returns the pairs variable-exponent of the `VariablesDict` as a tuple of tuples

```
>>> VariablesDict(a=2, b=3).items()
(('a', 2), ('b', 3))
```

Return type tuple

__iter__()

Returns the iterator of the VariablesDict

```
>>> iter(VariablesDict(a=2, b=3))
{'a': 2, 'b': 3}
```

For more informations see `VariablesDict.__next__()`.

Return type *VariablesDict*

__next__()

Gets the next variable in the VariablesDict

```
>>> i = iter(VariablesDict(a=2, b=3))
>>> next(i)
'a'
>>> next(i)
'b'
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

Return type str

Raise StopIteration

__len__()

Returns the dict's len

```
>>> len(VariablesDict(a=2, b=3))
2
```

Return type int

__str__()

Returns the VariablesDict as a string, like a normal dict

```
>>> str(VariablesDict(x=2, y=3))
"{'x': 2, 'y': 3}"
```

Return type str

__repr__()

Returns the VariablesDict as a string

```
>>> repr(VariablesDict(y=5))
"{'y': 5}"
```

For more informations see `VariablesDict.__str__()`.

Return type `str`**__add__** (*other*)

Sums two `VariablesDict`, returning a `VariablesDict` whose exponents are the sum of the starting `VariablesDicts`' ones

```
>>> VariablesDict(x=5, y=3) + VariablesDict(y=5)
{'x': 5, 'y': 8}
```

It raises a `TypeError` if *other* is not a `VariablesDict`

```
>>> VariablesDict(x=1) + 3
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'VariablesDict' and 'int'
```

Return type `VariablesDict`**Raise** `TypeError`**__sub__** (*other*)

Return a `VariablesDict` whose values are the difference between the starting `VariablesDicts`' ones

```
>>> VariablesDict(x=5, y=3) - VariablesDict(x=1, y=2)
{'x': 4, 'y': 1}
```

If any exponent becomes negative, a `ValueError` will be raised instead:

```
>>> VariablesDict(c=2) - VariablesDict(c=3)
Traceback (most recent call last):
...
ValueError: variable's exponent must be positive
```

It raises a `TypeError` if *other* is not a `VariablesDict`

```
>>> VariablesDict(x=1) - 3
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for -: 'VariablesDict' and 'int'
```

Return type `VariablesDict`**Raise** `ValueError`, `TypeError`**__mul__** (*other*)

Returns a `VariablesDict` whose exponents are this one's, multiplied by a given (integer) number

```
>>> VariablesDict(a=2, b= 5) * 3
{'a': 6, 'b': 15}
```

If the number is negative, a `ValueError` is raised

```
>>> VariablesDict() * (-15)
Traceback (most recent call last):
...
ValueError: can't multiply a VariablesDict by a negative number
```

It raises a `TypeError` if *other* is not an integer

```
>>> VariablesDict(x=1) * '3'
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for *: 'VariablesDict' and 'str'
```

Return type *VariablesDict*

Raise TypeError, ValueError

`__truediv__` (*other*)

Returns a VariablesDict whose values are this one's divided by a given (integer) number

```
>>> VariablesDict(a=4, b=2) / 2
{'a': 2, 'b': 1}
```

If the VariablesDict is not divisible by the given number, it will raise a ValueError

```
>>> VariablesDict(x=7) / 3
Traceback (most recent call last):
...
ValueError: can't divide this VariablesDict by 3
```

To see if a VariablesDict is divisible by a number, you can use modulus operator (see more at *VariablesDict.__mod__()*):

It raises a TypeError if *other* is not an integer

```
>>> VariablesDict(x=1) / '3'
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for /: 'VariablesDict' and 'str'
```

Return type *VariablesDict*

Raises ValueError, TypeError

`__mod__` (*other*)

Checks if the VariablesDict can be divided by a number (True => can be divided by *other*).

```
>>> VariablesDict(a=2, b=4) % 2
True
>>> VariablesDict(a=2, b=4) % 3
False
```

It raises ValueError if *other* isn't a positive integer

```
>>> VariablesDict(k=2) % (-7)
Traceback (most recent call last):
...
ValueError: can't use modulus with VariablesDict and negative numbers
```

It raises a TypeError if *other* is not an integer

```
>>> VariablesDict(x=1) % '3'
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for %: 'VariablesDict' and 'str'
```

Return type bool

Raise TypeError, ValueError

__eq__ (*other*)

Checks if two VariablesDict are equivalent

```
>>> VariablesDict(a=2, b=4) == VariablesDict(b=4, a=2)
True
```

If *other* is not a VariablesDict it always returns False

```
>>> VariablesDict(a=2, b=4) == 3
False
```

Return type bool

__hash__ ()

Returns the hash of the VariablesDict by hashing the result of *VariablesDict.items()*.

Return type int

__bool__ ()

Returns the opposite of *is_empty*.

```
>>> VariablesDict().is_empty
True
>>> bool(VariablesDict())
False
```

Return type int

2.2 Monomials

2.2.1 Monomials

class ruffini.**Monomial** (*coefficient=1, variables={}, **kwargs*)

Bases: object

A Monomial is the product of a coefficient and some variables.

Monomials can be added, subtracted, multiplied and divided together (and with numbers). lcm and gcd between monomials (and numbers) is available, too.

You can also assign a value to the variables and calculate the value of that monomial with the value you assigned.

__init__ (*coefficient=1, variables={}, **kwargs*)

Creates a new Monomial. The default *coefficient* value is 1 (so it can be omitted); variables instead are empty for default

```
>>> Monomial(17, k=3)
17k**3
>>> Monomial()
1
```

Every *coefficient* will be transformed in an instance of `Fraction`

```
>>> type(Monomial(7, a=2).coefficient)
<class 'fractions.Fraction'>
```

Monomials can also be initialized by passing a dictionary (or anything similar) where are stored the variables:

```
>>> Monomial(2, {'x': 2, 'y': 1})
2x**2y
```

Variables will be stored in a *VariableDict*. For more infos, see `VariablesDict.__init__()`.

Once initialized the monomial, it calculates the monomial's total degree (which is the sum of the variables' degrees)

```
>>> Monomial(-2, a=2, b=1, c=3).degree
6
```

Raise `ValueError`, `TypeError`

similar_to (*other*)

Checks if two monomials are similar (if they have the same variables).

```
>>> m = Monomial(3, x=1, y=1)
>>> m.similar_to(Monomial(3.14))
False
>>> m.similar_to(Monomial(2, x=1, y=1))
True
```

If the second operand is not a monomial the result will always be *False*

```
>>> m.similar_to("")
False
```

When a monomial has no variables, if compared to a number the result will be *True*

```
>>> Monomial(6).similar_to(Fraction(1, 3))
True
```

Return type `bool`

has_root (*index*)

Checks if the monomial “has” the root: the monomial $4x^2$, for example, is a square, so we can say it has root 2, because $(4x^2)^{(1/2)}$ is a monomial $(2x)$.

```
>>> Monomial(4, x=2).has_root(2)
True
```

We can't say the same thing for $16a^4b$:

```
>>> Monomial(16, a=4, b=1).has_root(2)
False
```

Zero has all the roots

```
>>> Monomial(0).has_root(700)
True
```

Raises `TypeError`

Return type `bool`

root (*index*)

Calculates the root of given index of the monomial

```
>>> Monomial(4, x=2).root(2)
2x
>>> Monomial(-27, a=9).root(3)
-3a**3
>>> Monomial(0).root(700)
0
```

If a monomial hasn't a root, it raises a `ValueError`

```
>>> Monomial(5, b=2).root(3)
Traceback (most recent call last):
...
ValueError: this monomial hasn't root 3
```

To see if a monomial has a root, use `Monomial.has_root()`.

Raises `ValueError`

Return type `Monomial`

gcd (*other*)

Calculates the greatest common divisor of two monomials or numbers

```
>>> a = Monomial(5, x=1, y=1)
>>> b = Monomial(15, x=1)
>>> a.gcd(b)
5x
```

It works only with integer coefficient that aren't equal to zero

```
>>> a.gcd(3.14)
Traceback (most recent call last):
...
ValueError: Monomial coefficient must be a whole number
>>> a.gcd(Monomial(3.14))
Traceback (most recent call last):
...
ValueError: Monomial coefficient must be a whole number
>>> b.gcd(0)
Traceback (most recent call last):
...
ValueError: Coefficient can't be zero
```

The result is always positive

```
>>> c = Monomial(-30, x=1, y=1)
>>> b.gcd(c)
15x
```

If you want to calculate the gcd with more factors, you can use the shorthand `gcd()`.

Return type *Monomial*, Fraction

Raise TypeError, ValueError

lcm (*other*)

Calculates the least common multiple of two monomials or numbers

```
>>> a = Monomial(2, x=1, y=1)
>>> b = Monomial(-9, y=3)
>>> a.lcm(b)
18xy**3
```

If you want to know more, please check `Monomial.gcd()`.

If you want to calculate the lcm between more monomials, you can use the `lcm()` shorthand.

Return type *Monomial*, Fraction

Raise TypeError, ValueError

eval (*values*={}, ***kwargs*)

Evaluates the monomial, giving values for each variable

```
>>> m = Monomial(5, x=1, y=1)
>>> m.eval(x=2, y=3)
Fraction(30, 1)
```

NB: *if there are no variables left, it returns only the coefficient, as instance of :class:'Fraction'.*

You can also assign variables' values with a dictionary (or any subclass)

```
>>> m.eval({'x': 2, 'y': 3})
Fraction(30, 1)
```

If you omit some variables' values, those variables will remain as they were

```
>>> m.eval(x=2)
10y
```

You can declare some variables values which aren't in the monomial and the result won't change

```
>>> m.eval(b=7)
5xy
```

The value for a variable can be a Monomial as well

```
>>> m.eval(x=Monomial(3, y=1))
15y**2
```

Return type int, float, *Monomial*

Raise TypeError

__add__ (*other*)

Sums two monomials

```
>>> Monomial(5, x=1, y=3) + Monomial(-1.5, x=1, y=3)
7/2xy**3
```

You can also sum a monomial and a number, but the result will be an instance of *Polynomial*.

```
>>> Monomial(1, z=1) + 17
z + 17
```

Return type *Monomial, Polynomial*

Raise `TypeError`

__sub__ (*other*)

Subtracts two monomials

```
>>> Monomial(5, x=1) - Monomial(3, x=1)
2x
```

If the monomials are not similar or the second operand is a number, the result will be a polynomial

```
>>> Monomial(5, x=1, y=3) - Monomial(3, x=1)
5xy**3 - 3x
>>> Monomial(17, a=1, b=1) - 2.5
17ab - 5/2
```

Return type *Monomial, Polynomial*

Raise `TypeError`

__mul__ (*other*)

Multiplicates two monomials or a monomial and a number

```
>>> Monomial(5, x=1, y=2) * Monomial(2, a=1, b=1)
10abxy**2
>>> Monomial(3, c=2) * 5
15c**2
```

Return type *Polynomial, Monomial*

Raise `TypeError`

__truediv__ (*other*)

Divides two monomials or a monomial and a number

```
>>> Monomial(6, a=3) / Monomial(3, a=1)
2a**2
>>> Monomial(18, k=3) / 6
3k**3
```

If *other*'s variable's exponents are higher than this monomial's, it raises a *ValueError*

```
>>> Monomial(5) / Monomial(4, x=1)
Traceback (most recent call last):
...
ValueError: variable's exponent must be positive
```

Return type *Monomial*

Raise `ValueError, TypeError`

__pow__ (*exp*)

Raises a monomial to a given power

```
>>> Monomial(5, x=1) ** 2
25x**2
```

If the exponent is 0, the result will be 1

```
>>> Monomial(5, k=6) ** 0
1
```

It raises a *ValueError* if the exponent is negative

```
>>> Monomial(17, k=1) ** (-1)
Traceback (most recent call last):
...
ValueError: Exponent can't be negative
```

It raises a *TypeError* if *exp* isn't an instance of *int*.

```
>>> Monomial(3.14, a=3) ** 2.5
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for ** or pow(): 'Monomial' and 'float'
```

To calculate roots, use *Monomial.root()*.**Return type** *Monomial***Raise** *ValueError*, *TypeError***__radd__** (*other*)Reverse for *Monomial.__add__()*

```
>>> 18 + Monomial(3)
21
```

For more informations, see *Monomial.__add__()* docs.**Return type** *Monomial*, *Polynomial***Raise** *TypeError***__rsub__** (*other*)Reverse for *Monomial.__sub__()*

```
>>> 9 - Monomial(4)
5
```

For more informations, see *Monomial.__sub__()* docs.**Return type** *Polynomial*, *Monomial***Raise** *TypeError***__rmul__** (*other*)Reverse for *Monomial.__mul__()*

```
>>> 5 * Monomial(2, x=2)
10x**2
```

For more informations, see `Monomial.__mul__()` docs.

Return type *Monomial, Polynomial*

Raise `TypeError`

`__rtruediv__` (*other*)

Reverse for `Monomial.__truediv__()`

```
>>> 8 / Monomial(4)
2
```

For more informations, see `Monomial.__truediv__ docs()`.

Return type *Monomial*

Raise `ValueError, TypeError`

`__str__` ()

Returns the monomial as a string. Normally, it will return the coefficient and the variables without spaces or `*`.

The power is indicated with `**`.

Examples: `>>> str(Monomial(5, x=1, y=1))` '5xy' `>>> str(Monomial(a=2))` 'a**2' `>>> str(Monomial(-1, k=3))` '-k**3' `>>> str(Monomial(0, s=5))` '0' `>>> str(Monomial())` '1' `>>> str(Monomial(-1))` '-1'

Variables are displayed in alphabetical order

```
>>> str(Monomial(5, k=2, b=3))
'5b**3k**2'
```

Return type `str`

`__repr__` ()

Return the monomial as a string

```
>>> Monomial(5, x=5)
5x**5
```

For more informations, see `:func:Monomial.__str__()`.

Return type `str`

`__eq__` (*other*)

Checks if two monomials are equivalent

```
>>> Monomial(5, x=1) == Monomial(5, x=1)
True
```

If there are no variables, it can be compared also to a number

```
>>> Monomial(4) == 4
True
```

If the second operand isn't a monomial or a number, it will return *False*.

It works by comparing the hashes calculated with `Monomial.__hash__()`.

Return type `bool`

Raise `TypeError`

`__neg__()`

Returns the opposite of the monomial

```
>>> - Monomial(5, x=1, y=1)
-5xy
```

Return type *Monomial*

`__abs__()`

Returns the absolute value of the monomial

```
>>> abs(Monomial(-3, a=1, b=4))
3ab**4
```

Return type *Monomial*

`__hash__()`

Returns the hash for the Monomial

The hash for $8xy$, for example, is equivalent to the hash of $(8, ('x', 1), ('y', 1))$.

```
>>> hash(Monomial(8, x=1, y=1)) == hash((8, ('x', 1), ('y', 1)))
True
```

If the monomial has no variables, its hash will be equal to the coefficient's hash

```
>>> hash(Monomial(3)) == hash(3)
True
```

Return type `int`

2.2.2 gcd()

`ruffini.gcd(*args)`

Returns the gcd between two or more monomials. For more informations, see *Monomial.gcd()*

2.2.3 lcm()

`ruffini.lcm(*args)`

Returns the lcm between two or more monomials. For more informations, see *Monomial.lcm()*

2.3 Polynomials

2.3.1 Polynomials

class `ruffini.Polynomial` (*terms=()*, **args*)

Bases: `tuple`

A Polynomial object is the sum of two or more monomials and/or numbers.

You can sum, subtract and multiply instances of Polynomial.

You can assign a value to the variables and calculate the value of that polynomial with the value you assigned.

NB The Polynomial class is a subclass of tuple, so all the methods of tuple are automatically inherited from Polynomial; many of these methods are not in this docs.

static `__new__` (*cls, terms=(), *args*)

Create the polynomial by giving it a list of terms (a term can be a Monomial or a number); if two or more terms have the same variables, it will sum them together

```
>>> Polynomial(Monomial(2, x=2, y=2), Monomial(3, x=2, y=2))
5x**2y**2
```

Raise TypeError

`__init__` (*terms=(), *args*)

Initialize the polynomial, then calculate its degree (the highest degree between terms' ones)

```
>>> p = Polynomial(Monomial(a=1), Monomial(3))
>>> p
a + 3
>>> p.degree
1
```

term_coefficient (*variables=None, **kwargs*)

Return the coefficient of the term with the given variables

```
>>> x = Variable('x')
>>> y = Variable('y')
>>>
>>> p = 2*x*y - 6*x*y**3 + 8*x*y
>>> p
10xy - 6xy**3
>>>
>>> p.term_coefficient(x=1, y=1)
Fraction(10, 1)
```

If none is found, the result will be 0

```
>>> p.term_coefficient(k=1, b=2)
0
```

You can also give directly the variables as an argument

```
>>> p.term_coefficient(x*y)
Fraction(10, 1)
```

Return type int, float

factorize ()

With this method you can factorize the polynomial.

For more informations, see [factorize\(\)](#) docs.

Return type *FPolynomial*

zeros

Return a set of zeros for the polynomial

```
>>> x = Variable('x')
>>> p = 3*x**3 + 2*x**2 - 3*x - 2
>>> p.zeros
{Fraction(1, 1), Fraction(-2, 3), Fraction(-1, 1)}
```

It works only with polynomials with only a variable and a constant term

```
>>> Polynomial(3*x, Monomial(2, y=1)).zeros
Traceback (most recent call last):
...
ValueError: Can't calculate zeros for polynomials with more than a variable
```

```
>>> Polynomial(3*x, 5*x**2).zeros
Traceback (most recent call last):
...
ValueError: Can't calculate zeros for polynomials without a constant term
```

Return type set

Raises ValueError

eval (*values*={}, ***kwargs*)

Evaluates the polynomial, giving values for each variable

```
>>> p = Polynomial(Monomial(5, x=1), Monomial(3, y=1))
>>> p.eval(x=2, y=3)
19
```

For more informations, see [*Monomial.eval\(\)*](#).

Return type int, float, *Monomial*, *Polynomial*

__add__ (*other*)

Sum the polynomial with another polynomial, a monomial or a number, too.

```
>>> x = Variable('x')
>>> y = Variable('y')
>>> p = 3*x + 2*y
>>>
>>> p + (3*y + 2)
3x + 5y + 2
>>>
>>> p + 2*x
5x + 2y
>>>
>>> p + 1
3x + 2y + 1
```

Return type *Polynomial*

Raise TypeError

__sub__ (*other*)

Subtract the polynomial from another polynomial, a monomial or a number.

```

>>> x = Variable('x')
>>> y = Variable('y')
>>>
>>> p = 3*x + 2*y
>>>
>>> p - (3*y + 2)
3x - y - 2
>>>
>>> p - 2*x
x + 2y
>>>
>>> p - 1
3x + 2y - 1

```

Return type *Polynomial*

Raise `TypeError`

`__mul__` (*other*)

This method is used to multiply a polynomial by a polynomial, a monomial or a number:

```

>>> x = Variable('x')
>>> y = Variable('y')
>>>
>>> p = 3*x + 2*y
>>>
>>> p * (3*y + 2)
9xy + 6x + 6y**2 + 4y
>>>
>>> p * x
3x**2 + 2xy
>>>
>>> p * 4
12x + 8y

```

Return type *Polynomial*

Raise `TypeError`

`__radd__` (*other*)

This method is the reverse for *Polynomial.__add__* (). With this method, you can swap the two operands of the addition:

```

>>> 8 + Polynomial(Monomial(4, a=2))
4a**2 + 8

```

For more informations, see *Polynomial.__add__* () docs.

Return type *Polynomial*

Raise `TypeError`

`__rsub__` (*other*)

This method is the reverse for *Polynomial.__sub__* (). With this method, you can swap the two operands of the addition:

```

>>> 5 - Polynomial(Monomial(7, k=1))
-7k + 5

```

For more informations, see `Polynomial.__sub__ docs()`.

Return type *Polynomial*

Raise `TypeError`

`__rmul__` (*other*)

This method is the reverse for `Polynomial.__mul__()`. With this method, you can swap the two operands of the addition:

```
>>> 10 * Polynomial(Monomial(3.5, b=3))
35b**3
```

For more informations, see `Polynomial.__mul__()` docs.

Return type *Polynomial*, `NotImplemented`

Raise `TypeError`

`__str__` ()

Return the polynomial as a string. Powers are indicated with `**`.

```
>>> str(Polynomial(Monomial(4, a=4, b=1)))
'4a**4b'
>>> str(Polynomial(Monomial(a=2), Monomial(-2, c=2)))
'a**2 - 2c**2'
>>> str(Polynomial(Monomial(3, x=2), Monomial(6, y=3)))
'3x**2 + 6y**3'
```

To see how the single terms are printed, see the `Monomial.__str__()` docs.

Return type `str`

`__repr__` ()

Return the polynomial as a string.

```
>>> repr(Polynomial(Monomial(4, a=4, b=1)))
'4a**4b'
```

For more informations, see `Polynomial.__str__()`.

Return type `str`

`__eq__` (*other*)

Check if two polynomials are equivalent, comparing each term

```
>>> p0 = Polynomial(Monomial(4, a=4, b=1))
>>> p1 = Polynomial(Monomial(1, a=2), Monomial(-2, c=2))
>>> p2 = Polynomial(Monomial(-2, c=2), Monomial(1, a=2))
>>>
>>> p0 == p1
False
>>> p0 == p0
True
>>> p1 == p2
True
```

If a polynomial has a single term, it can also be compared to a monomial

```
>>> Polynomial(Monomial(3, f=2)) == Monomial(3, f=2)
True
```

Since a monomial with no variables can be compared to a number, if a polynomial has only a term - which is a monomial with no variables - it can be compared to a number

```
>>> Polynomial(Monomial(7)) == 7
True
```

In any other case, the result will be False.

```
>>> Polynomial() == {1, 2, 3}
False
```

Return type bool

__neg__()

Return the opposite of the polynomial, changing the sign of each term of the polynomial

```
>>> -Polynomial(Monomial(4, x=1), Monomial(2, y=2))
-4x - 2y**2
```

Return type *Polynomial*

2.4 Polynomials Factorization

2.4.1 FPolynomials

class ruffini.FPolynomial

Bases: tuple

A FPolynomial (factorized polynomial) object is a multiplication of two or more polynomials.

When you factor a polynomial, an FPolynomial instance is returned. On the other hand, you can multiply the polynomials of the fpolynomial and obtain the starting polynomial.

You can't perform any math operation with a factorized polynomial

NB The FPolynomial class is a subclass of tuple, so all the methods of tuple are automatically inherited from FPolynomial; many of these methods are not in this docs.

static **__new__**(cls, *factors)

Create the factorized polynomial giving it a list of factors (int, float, Fraction, Monomial or Polynomial).

```
>>> p = Polynomial(Monomial(2, x=2, y=2))
>>> FPolynomial(5, p)
5(2x**2y**2)
```

Every factor that is equal to 1 is not inserted in the fpolynomial

```
>>> len(FPolynomial(5, p, 1))
2
```

In the factors must be present at least a polynomial

```
>>> FPolynomial(5)
Traceback (most recent call last):
...
TypeError: There must be at least a polynomial
```

It converts all the factors to Polynomial and sort them by frequency.

Raise `TypeError`

eval()

Return the starting polynomial multiplying all the factors

```
>>> f = FPolynomial(5, Polynomial(Monomial(2, x=1), 3))
>>> f
5(2x + 3)
>>> f.eval()
10x + 15
```

The result will always be a Polynomial

```
>>> type(f.eval())
<class 'ruffini.polynomials.Polynomial'>
```

Return type *Polynomial*

__str__()

Return the factorized polynomial as a string.

```
>>> p1 = Polynomial(2, Monomial(3, x=1))
>>> p2 = Polynomial(Monomial(2, y=1), 17)
>>> print(FPolynomial(p1, p2))
(2 + 3x)(2y + 17)
```

If the first element is a monomial, an integer or a float, its brackets will be omitted

```
>>> print(FPolynomial(5, p1))
5(2 + 3x)
```

Otherwise, if a factor appears two times, the result will be like this

```
>>> print(FPolynomial(p2, p2))
(2y + 17)**2
```

```
>>> print(FPolynomial(p2, p2, 5))
5(2y + 17)**2
```

Return type `str`

__repr__()

Return the factorized polynomial as a string.

```
>>> p1 = Polynomial(2, Monomial(3, x=1))
>>> p2 = Polynomial(Monomial(2, y=1), 17)
>>> repr(FPolynomial(p1, p2))
'(2 + 3x)(2y + 17)'
```

For more informations, see :func:`FPolynomial.__str__()`.

Return type `str`

__eq__ (other)

Compare two factorized polynomials to see if they're equivalent.

```
>>> p = Polynomial(Monomial(5, x=2), 3)
>>> m = Monomial(2, y=1)
>>> FPolynomial(p, m) == FPolynomial(p, m)
True
```

If we swap factors, the result doesn't change

```
>>> FPolynomial(p, m) == FPolynomial(m, p)
True
```

If we compare two factorized polynomials with different factors, but which - once evaluated - they are equivalent, the result is true. For example:

```
>>> x = Variable("x")
>>> y = Variable("y")
>>>
>>> fp1 = FPolynomial(2*x - 3*y**2, 2*x - 3*y**2)
>>> fp2 = FPolynomial(3*y**2 - 2*x, 3*y**2 - 2*x)
>>>
>>> fp1.eval() == fp2.eval()
True
>>> fp1 == fp2
True
```

Return type bool

2.4.2 factorize()

`ruffini.factorize` (*polynomial*)

Factorize the given polynomial using some algorithms (sub functions), such as

`gcf()`, `group` [todo], `squares difference` [todo], `cubes sum` [todo], `cubes difference` [todo], `binomial square` [todo], `factorize_binomia_square()`, `trinomial square` [todo].

It works in recursive mode.

```
>>> factorize(Polynomial(Monomial(10, x=1), 15))
5(2x + 3)
```

If polynomial isn't a polynomial, it will raise a `TypeError`

```
>>> factorize('John')
Traceback (most recent call last):
...
TypeError: Can't factorize object of type 'str'
```

Return type *FPolynomial*

Raises `TypeError`

2.4.3 gcf()

`ruffini(gcf)` (*polynomial*)

Factorize a polynomial with the gcf (greates common facor) method. It works like this:

$$AX + AY + \dots = A(X + Y + \dots)$$

for example:

```
>>> gcf(Polynomial(Monomial(10, x=1), 15))
(5, 2x + 3)
```

If there isn't a gcf, it will return the starting polynomial

```
>>> gcf(Polynomial(Monomial(11, x=1), 15))
(11x + 15,)
```

The function returns a tuple.

Return type tuple

Raise TypeError

2.4.4 binomial_square()

`ruffini.binomial_square` (*polynomial*)

Check if the polynomial is a binomial square. It works like this:

$$A^2 + B^2 + 2AB = (A + B)^2$$

for example:

```
>>> p = Polynomial(Monomial(4, x=2), Monomial(9, y=4), Monomial(-12, x=1, y=2))
>>> p
4x**2 + 9y**4 - 12xy**2
>>> binomial_square(p)
(2x - 3y**2, 2x - 3y**2)
```

It can raise `ValueError` in three cases:

1. When polynomial's length isn't 3:

```
>>> binomial_square(Polynomial(1))
Traceback (most recent call last):
...
ValueError: Not a binomial square
```

2. When there aren't at least two squares:

```
>>> p = Polynomial(Monomial(4, x=2), Monomial(9, y=5), Monomial(3))
>>> binomial_square(p)
Traceback (most recent call last):
...
ValueError: Not a binomial square
```

3. When the third term isn't the product of the firsts:

```
>>> p = Polynomial(Monomial(4, x=2), Monomial(9, y=4), Monomial(3))
>>> binomial_square(p)
Traceback (most recent call last):
...
ValueError: Not a binomial square
```

Return type tuple**Raise** TypeError, ValueError

2.5 Equations

2.5.1 Equations

class ruffini.**Equation** (*first, second*)

Bases: object

You know what an equation is. Well, if you're reading this docs I hope you know. Anyway.

The sides of the equations are polynomials.

__init__ (*first, second*)

Initialize the Equation by giving the two sides.

```
>>> Equation(Monomial(2, x=1), 4)
2x - 4 = 0
```

As you may have noticed, it moves all the terms to the first side. It calculates the equation's variable and degree.

```
>>> equation = Equation(Monomial(2, x=1), 4)
>>> equation.first
2x - 4
>>> equation.second
0
>>> equation.degree
1
>>> equation.variable
'x'
```

NB: It raises a NotImplementedError if you try to create an equation with more than one variable

```
>>> Equation(Monomial(3, x=1), Monomial(2, y=1))
Traceback (most recent call last):
...
NotImplementedError: Too many variables
```

Raise NotImplementedError**__str__** ()

Return the equation formatted as a string

```
>>> print(Equation(Monomial(2, x=1), 4))
2x - 4 = 0
```

Return type str**__repr__** ()

Return the equation formatted as a string

```
>>> Equation(Monomial(2, x=1), 4)
2x - 4 = 0
```

For more informations, see `Equation.__str__()`.

Return type str

solve()

Solve the equation.

```
>>> Equation(Monomial(x=2), 4).solve()
(Fraction(2, 1), Fraction(-2, 1))
```

It works only with equations of degree 1 or 2; if it's higher, it raises a `NotImplementedError`.

```
>>> Equation(Monomial(x=3), 27).solve()
Traceback (most recent call last):
...
NotImplementedError: Can't solve equations with degree higher than 2
```

If the result is impossible or indeterminate it raises a `ValueError`

```
>>> Equation(Monomial(0, x=1), 0).solve()
Traceback (most recent call last):
...
ValueError: Equation impossible or indeterminate
```

Raise `ValueError`, `NotImplementedError`

Symbols

`__abs__()` (*ruffini.Monomial method*), 20
`__add__()` (*ruffini.Monomial method*), 16
`__add__()` (*ruffini.Polynomial method*), 22
`__add__()` (*ruffini.VariablesDict method*), 11
`__bool__()` (*ruffini.VariablesDict method*), 13
`__eq__()` (*ruffini.FPolynomial method*), 26
`__eq__()` (*ruffini.Monomial method*), 19
`__eq__()` (*ruffini.Polynomial method*), 24
`__eq__()` (*ruffini.VariablesDict method*), 13
`__getitem__()` (*ruffini.VariablesDict method*), 9
`__hash__()` (*ruffini.Monomial method*), 20
`__hash__()` (*ruffini.VariablesDict method*), 13
`__init__()` (*ruffini.Equation method*), 29
`__init__()` (*ruffini.Monomial method*), 13
`__init__()` (*ruffini.Polynomial method*), 21
`__init__()` (*ruffini.VariablesDict method*), 7
`__iter__()` (*ruffini.VariablesDict method*), 10
`__len__()` (*ruffini.VariablesDict method*), 10
`__mod__()` (*ruffini.VariablesDict method*), 12
`__mul__()` (*ruffini.Monomial method*), 17
`__mul__()` (*ruffini.Polynomial method*), 23
`__mul__()` (*ruffini.VariablesDict method*), 11
`__neg__()` (*ruffini.Monomial method*), 19
`__neg__()` (*ruffini.Polynomial method*), 25
`__new__()` (*ruffini.FPolynomial static method*), 25
`__new__()` (*ruffini.Polynomial static method*), 21
`__next__()` (*ruffini.VariablesDict method*), 10
`__pow__()` (*ruffini.Monomial method*), 17
`__radd__()` (*ruffini.Monomial method*), 18
`__radd__()` (*ruffini.Polynomial method*), 23
`__repr__()` (*ruffini.Equation method*), 29
`__repr__()` (*ruffini.FPolynomial method*), 26
`__repr__()` (*ruffini.Monomial method*), 19
`__repr__()` (*ruffini.Polynomial method*), 24
`__repr__()` (*ruffini.VariablesDict method*), 10
`__rmul__()` (*ruffini.Monomial method*), 18
`__rmul__()` (*ruffini.Polynomial method*), 24
`__rsub__()` (*ruffini.Monomial method*), 18

`__rsub__()` (*ruffini.Polynomial method*), 23
`__rtruediv__()` (*ruffini.Monomial method*), 19
`__str__()` (*ruffini.Equation method*), 29
`__str__()` (*ruffini.FPolynomial method*), 26
`__str__()` (*ruffini.Monomial method*), 19
`__str__()` (*ruffini.Polynomial method*), 24
`__str__()` (*ruffini.VariablesDict method*), 10
`__sub__()` (*ruffini.Monomial method*), 17
`__sub__()` (*ruffini.Polynomial method*), 22
`__sub__()` (*ruffini.VariablesDict method*), 11
`__truediv__()` (*ruffini.Monomial method*), 17
`__truediv__()` (*ruffini.VariablesDict method*), 12

B

`binomial_square()` (*in module ruffini*), 28

E

Equation (*class in ruffini*), 29
`eval()` (*ruffini.FPolynomial method*), 26
`eval()` (*ruffini.Monomial method*), 16
`eval()` (*ruffini.Polynomial method*), 22
`exponents()` (*ruffini.VariablesDict method*), 9

F

`factorize()` (*in module ruffini*), 27
`factorize()` (*ruffini.Polynomial method*), 21
FPolynomial (*class in ruffini*), 25

G

`gcd()` (*in module ruffini*), 20
`gcd()` (*ruffini.Monomial method*), 15
`gcf()` (*in module ruffini*), 27

H

`has_root()` (*ruffini.Monomial method*), 14

I

`items()` (*ruffini.VariablesDict method*), 9

L

`lcm()` (*in module ruffini*), [20](#)

`lcm()` (*ruffini.Monomial method*), [16](#)

M

`Monomial` (*class in ruffini*), [13](#)

P

`Polynomial` (*class in ruffini*), [20](#)

R

`root()` (*ruffini.Monomial method*), [15](#)

S

`similar_to()` (*ruffini.Monomial method*), [14](#)

`solve()` (*ruffini.Equation method*), [30](#)

T

`term_coefficient()` (*ruffini.Polynomial method*),
[21](#)

V

`VariablesDict` (*class in ruffini*), [7](#)

Z

`zeros` (*ruffini.Polynomial attribute*), [21](#)