ruffini Documentation

Parri Gianluca

Mar 24, 2020

Contents

Ruffini is a simple python library to compute monomials and polynomials. It's open source on github at gianluparri03/ruffini.

CHAPTER 1

Tutorial

```
>>> import ruffini
```

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
>>>
>>> (7*y) * (2*x) # Multiplication!
```

(continues on next page)

(continued from previous page)

```
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

CHAPTER 2

Reference

2.1 Variables

2.1.1 VariablesDicts

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

Bases: dict

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

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

The changes are:

- If a variable isn't in the dictionary, its exponent is 0
- Therefore, variables with exponent 0 won't be inserted
- Variables are made lowercase
- Variables must be letters from the latin alphabet and one-character long
- Exponents must be integer

NB VariablesDict is a sublass of dict, so all the methods of dict are inherited rom VariablesDict; many of these methods are not in this docs.

__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}
```

As said above, it asserts if variables are lowercase; if not they'll be transformed automatically:

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

```
>>> VariablesDict(c=9.0)
{'c': 9}
```

It can raise an error if:

- variable's name is too long (ValueError)

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

- variable's name is not alphabetical (ValueError)

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

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

```
>>> 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 (ValueError)

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

After that, it checks if the dictionary is empty:

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

Raise TypeError, ValueError

__setitem__(*key*, *value*)
Raises AttributeError: VariablesDict is immutable

Raise AttributeError

__delitem__(*key*)
Raises AttributeError: VariablesDict is immutable

Raise AttributeError

pop(*key*)
Raises AttributeError: VariablesDict is immutable

Raise AttributeError

clear()
Raises AttributeError: VariablesDict is immutable

Raise AttributeError

__getitem__(*key*)
Gets the exponent of a variable from the variable's name

```
>>> v = VariablesDict(a=2, b=3)
>>> v['a']
2
```

If a variable isn't in the dictionary, its value is 0

```
>>> v['k']
0
```

Return type int

__str__()
Returns the dict as a string (as a normal dict)

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

Variables are sorted alphabetically:

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

Return type str

__repr__()
Returns the dict 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}
>>> VariablesDict(x=18) + VariablesDict(y=4)
{'x': 18, 'y': 4}
>>> VariablesDict(a=36) + VariablesDict()
{'a': 36}
```

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}
>>> VariablesDict(x=18) - VariablesDict(x=18)
{}
```

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
```

Return type *VariablesDict*

Raise ValueError, TypeError

__mul__(*other*)

Returns a VariablesDict whose exponents are this one's, but 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
```

Return type *VariablesDict*

Raise TypeError, ValueError

__truediv__(*other*)

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

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

If the VariableDict 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__()*):

```
>>> VariablesDict(x=7) % 3
False
```

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
```

Return type bool

Raise TypeError

__hash__()

Returns the hash of the VariablesDict. It's equal to the tuple of its items.

```
>>> hash(VariablesDict(x=2)) == hash((('x', 2),))
True
```

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 togheter (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
```

If *coefficient* is an instance of float but it's a whole number (like 18.0), it will be transformed in int (in this case, 18)

```
>>> Monomial(7.0, a=2)
7a**2
```

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 the 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(6.28)
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/numbers different from zero

```
>>> a.gcd(3.14)
Traceback (most recent call last):
...
TypeError: Can't calculate gcd between Monomial and float
```

(continues on next page)

(continued from previous page)

```
>>> a.gcd(Monomial(3.14))
Traceback (most recent call last):
...
ValueError: Monomial coefficient must be int
>>> 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*, int

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 others informations like errors and limits, please check the documentation of Monomial().gcd().

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

Return type *Monomial*, int, float

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)
30
```

NB: *if there are no variables left, it returns only the coefficient, as instance of 'int' or 'float'*

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

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

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
```

Return type int, float, *Monomial*

Raise TypeError

__add__(*other*)
Sums two monomials.

```
>>> Monomial(5, x=1, y=3) + Monomial(-1.52, x=1, y=3)
3.48xy**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*)
Returns the subtraction between this monomial and *other*

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

If the monomials are not similar or the second operator 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 - 2.5
```

Return type *Monomial*, *Polynomial*

Raise TypeError

__mul__(*other*)
Multiplicates this monomial by *other*, which can be a monomial or a number

```
>>> Monomial(5, x=1, y=2) * Monomial(2, a=1, b=1)
10abxy**2
>>> Monomial(3, c=2) * 5
15c**2
>>> Monomial(k=3) * Monomial(k=3)
k**6
```

Return type *Polynomial*, *Monomial*

Raise TypeError

__truediv__(*other*)
Divide this monomial by another monomial or a number

```
>>> Monomial(6, a=3) / Monomial(3, a=1)
2a**2
>>> Monomial(18, k=3) / 6
3k**3
>>> Monomial(27, x=6) / Monomial(3, x=6)
9
```

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
>>> Monomial(4, c=6) ** 3
64c**18
```

If the exponent is 0, the result will be 1

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

It raises a *TypeError* if *exp* is an istance of *float*.

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

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
```

Return type *Monomial*

Raise ValueError, TypeError

__radd__(*other*)

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

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

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

Return type *Monomial*, *Polynomial*

Raise TypeError

__rsub__(*other*)
This method is the reverse for *Monomial.__sub__()*. With this method, you can swap the two operands of the subtraction:

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

For more informations, see *Monomial.__sub__()* docs.

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

Raise TypeError

__rmul__(*other*)
This method is the reverse for *Monomial.__mul__()*. With this method, you can swap the two operands of the multiplication:

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

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

Return type *Monomial*, *Polynomial*

Raise TypeError

__rtruediv__(*other*)
This method is the reverse for *Monomial.__truediv__()*. With this method, you can swap the two operands of the division:

```
>>> 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__()
Returns the monomial as a string

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

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

Return type str

__eq__(*other*)

Checks if two monomials are equivalent, comparing coefficients and variables

```
>>> 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 operator isn't a monomial or a number, it will return *False*.

Return type bool

Raise TypeError

__neg__()

Returns the opposite of the monomial inverting the coefficient

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

Return type *Monomial*

__abs__()

Returns the absolute value of the monomial, calculating the absolute value of the coefficient

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

Return type *Monomial*

__hash__()

Return 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*)
A shorthand to calculate the gcd between two or more monomials. For more informations, see *Monomial.gcd()*

2.2.3 lcm()

ruffini.**lcm**(**args*)
A shorthand to calculate 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 multiplicate 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 toghether

```
>>> 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)
10
```

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)
10
```

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
{-0.666666666666666, 1.0, -1.0}
```

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
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 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'
```

(continues on next page)

(continued from previous page)

```
>>> 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 istance 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, 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
```

(continues on next page)

(continued from previous page)

```
>>> fp1 == fp2
True
```

Return type bool

2.4.2 factorize()

ruffini.**factorize**(*polynomial*)

Factorize the given polynomial using some algorythms (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*, *Monomial*, int, float

Raises TypeError

2.4.3 gcf()

ruffini.**gcf**(*polynomial*)

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

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

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 will always return a FPolynomial.

Return type *FPolynomial*

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)**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 *FPolynomial*

Raise TypeError, ValueError

Index

Symbols

B

C

E

F

G

H

L

M

P

R

S

T

V

Z