Pattern Matcher Documentation

Release 0.5.0+3.g570b70e

Manuel Krebber

Apr 30, 2019

Contents

MatchPy is a library for pattern matching on symbolic expressions in Python.

Work in progress

CHAPTER 1

Installation

MatchPy is available via PyPI, and for Conda via conda-forge. It can be installed with `pip install matchpy` or `conda install -c conda-forge matchpy`.

CHAPTER 2

Overview

This package implements pattern matching in Python. Pattern matching is a powerful tool for symbolic computations, operating on symbolic expressions. Given a pattern and an expression (which is usually called *subject*), the goal of pattern matching is to find a substitution for all the variables in the pattern such that the pattern becomes the subject. As an example, consider the pattern $f(x)$, where f is a function and x is a variable, and the subject $f(a)$, where a is a constant symbol. Then the substitution that replaces x with a is a match. MatchPy supports associative and/or commutative function symbols, as well as sequence variables, similar to pattern matching in Mathematica.

A detailed example of how to use MatchPy can be found here.

MatchPy supports both one-to-one and many-to-one pattern matching. The latter makes use of similarities between patterns to efficiently find matches for multiple patterns at the same time.

A list of publications about MatchPy can be found *below*.

2.1 Expressions

Expressions are tree-like data structures, consisting of operations (functions, internal nodes) and symbols (constants, leaves):

```
>>> from matchpy import Operation, Symbol, Arity
>>> f = Operation.new('f', Arity.binary)
>>> a = Symbol('a')
>>> print(f(a, a))
f(a, a)
```

Patterns are expressions which may contain wildcards (variables):

```
>>> from matchpy import Pattern, Wildcard
>>> x = Wildcard.dot('x')
>>> print(Pattern(f(a, x)))
f(a, x_)
```

In the previous example, x is the name of the variable. However, it is also possible to use wildcards without names:

```
>>> w = Wildcard.dot()
>>> print(Pattern(f(w, w)))
f(_, _)
```

It is also possible to assign variable names to entire subexpressions:

```
>>> print(Pattern(f(w, a, variable_name='y')))
y: f(_, a)
```

2.2 Pattern Matching

Given a pattern and an expression (which is usually called subject), the idea of pattern matching is to find a substitution that maps wildcards to expressions such that the pattern becomes the subject. In MatchPy, a substitution is a dict that maps variable names to expressions.

```
>>> from matchpy import match
>>> y = Wildcard.dot('y')
>>> b = Symbol('b')
>>> subject = f(a, b)
>>> pattern = Pattern(f(x, y))
>>> substitution = next(match(subject, pattern))
>>> print(substitution)
{x  a, y  b}
```

Applying the substitution to the pattern results in the original expression.

```
>>> from matchpy import substitute
>>> print(substitute(pattern, substitution))
f(a, b)
```

2.3 Sequence Wildcards

Sequence wildcards are wildcards that can match a sequence of expressions instead of just a single expression:

```
>>> z = Wildcard.plus('z')
>>> pattern = Pattern(f(z))
>>> subject = f(a, b)
>>> substitution = next(match(subject, pattern))
>>> print(substitution)
{z  (a, b)}
```

2.4 Associativity and Commutativity

MatchPy natively supports associative and/or commutative operations. Nested associative operators are automatically flattened, the operands in commutative operations are sorted:

```
>>> g = Operation.new('g', Arity.polyadic, associative=True, commutative=True)
>>> print(g(a, g(b, a)))
g(a, a, b)
```

Associativity and commutativity is also considered for pattern matching:

```
>>> pattern = Pattern(g(b, x))
>>> subject = g(a, a, b)
>>> print(next(match(subject, pattern)))
{x  g(a, a)}
>>> h = Operation.new('h', Arity.polyadic)
>>> pattern = Pattern(h(b, x))
>>> subject = h(a, a, b)
>>> list(match(subject, pattern))
[]
```

2.5 Many-to-One Matching

When a fixed set of patterns is matched repeatedly against different subjects, matching can be sped up significantly by using many-to-one matching. The idea of many-to-one matching is to construct a so called discrimination net, a data structure similar to a decision tree or a finite automaton that exploits similarities between patterns. In MatchPy, there are two such data structures, implemented as classes: DiscriminationNet and ManyToOneMatcher. The DiscriminationNet class only supports syntactic pattern matching, that is, operations are neither associative nor commutative. Sequence variables are not supported either. The ManyToOneMatcher class supports associative and/or commutative matching with sequence variables. For syntactic pattern matching, the DiscriminationNet should be used, as it is usually faster.

```
>>> pattern1 = Pattern(f(a, x))
>>> pattern2 = Pattern(f(y, b))
>>> matcher = ManyToOneMatcher(pattern1, pattern2)
>>> subject = f(a, b)
>>> matches = matcher.match(subject)
>>> for matched_pattern, substitution in sorted(map(lambda m: (str(m[0]), str(m[1])),␣
↪matches)):
...     print('{} matched with {}'.format(matched_pattern, substitution))
f(a, x_) matched with {x  b}
f(y_, b) matched with {y  a}
```

CHAPTER 3

Roadmap

Besides the existing features, we plan on adding the following to MatchPy:

- Support for Mathematica's `Alternatives`: For example `f(a | b)` would match either `f(a)` or `f(b)`.
- Support for Mathematica's `Repeated`: For example `f(a..)` would match `f(a)`, `f(a, a)`, `f(a, a, a)`, etc.
- Support pattern sequences (`PatternSequence` in Mathematica). These are mainly useful in combination with `Alternatives` or `Repeated`, e.g. `f(a | (b, c))` would match either `f(a)` or `f(b, c)`. `f((a a)..)` would match any `f` with an even number of `a` arguments.
- All these additional pattern features need to be supported in the `ManyToOneMatcher` as well.
- Better integration with existing types such as `dict`.
- Code generation for both one-to-one and many-to-one matching. There is already an experimental implementation, but it still has some dependencies on MatchPy which can probably be removed.
- Improving the documentation with more examples.
- Better test coverage with more randomized tests.
- Implementation of the matching algorithms in a lower-level language, for example C, both for performance and to make MatchPy's functionality available in other languages.

CHAPTER 4

Contributing

If you have some issue or want to contribute, please feel free to open an issue or create a pull request. Help is always appreciated!

The Makefile has several tasks to help development:

- To install all needed packages, you can use `make init` .
- To run the tests you can use `make test`. The tests use pytest.
- To generate the documentation you can use `make docs` .
- To run the style checker (pylint) you can use `make check` .

If you have any questions or need help with setting things up, please open an issue and we will try the best to assist you.

CHAPTER 5

Publications

MatchPy: Pattern Matching in Python Manuel Krebber and Henrik Barthels Journal of Open Source Software, Volume 3(26), pp. 2, June 2018.

Efficient Pattern Matching in Python Manuel Krebber, Henrik Barthels and Paolo Bientinesi Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing, November 2017.

MatchPy: A Pattern Matching Library Manuel Krebber, Henrik Barthels and Paolo Bientinesi Proceedings of the 15th Python in Science Conference, July 2017.

Non-linear Associative-Commutative Many-to-One Pattern Matching with Sequence Variables Manuel Krebber Master Thesis, RWTH Aachen University, May 2017

If you want to cite MatchPy, please reference the JOSS paper:

```
@article{krebber2018,
    author    = {Manuel Krebber and Henrik Barthels},
    title     = {{M}atch{P}y: {P}attern {M}atching in {P}ython},
    journal   = {Journal of Open Source Software},
    year      = 2018,
    pages     = 2,
    month     = jun,
    volume    = {3},
    number    = {26},
    doi       = "10.21105/joss.00670",
    web       = "http://joss.theoj.org/papers/10.21105/joss.00670",
}
```

CHAPTER 6

Table of Contents

6.1 Linear Algebra Example

As an example, we will write the classes necessary to construct linear algebra equations. These equations consist of scalars, vectors, and matrices, as well as multiplication, addition, transposition, and inversion.

Lets start by importing everything we need:

```
>>> from matchpy import *
```

6.1.1 Symbols

First off, we create simple classes for our scalars and vectors:

```
>>> class Scalar(Symbol):
...     pass
>>> class Vector(Symbol):
...     pass
```

Now we can create vectors and scalars like this:

```
>>> a = Scalar('a')
>>> v = Vector('v')
```

For matrices, we want to be able to specify additional properties that a matrix has, for example it might be a diagonal or triangular matrix. We will just use a set of strings for the properties:

```
>>> class Matrix(Symbol):
...     def __init__(self, name, properties=[]):
...         super().__init__(name)
...         self.properties = frozenset(properties)
```

Now we can create matrices like this:

```
>>> M1 = Matrix('M1', ['diagonal', 'square'])
>>> M2 = Matrix('M2', ['symmetric', 'square'])
>>> M3 = Matrix('M3', ['triangular'])
```

6.1.2 Operations

We can quickly create a new operation using the *Operation.new* factory method:

```
>>> Times = Operation.new('*', Arity.variadic, 'Times', associative=True, one_
↪identity=True, infix=True)
```

We need to specify a name (`'*'`) and arity for the operation. In case that the name is not a valid python identifier, we also need to specify a class name (`'Times'`). The matrix multiplication is associative, but not commutative. In addition, we set *one_identity* to `True`, which means that a multiplication with a single operand can be replaced by that operand:

```
>>> Times(a)
Scalar('a')
```

The infix property is used when printing terms so that they look prettier:

```
>>> print(Times(a, v))
(a * v)
```

An alternative way of adding a new operation, is creating a subclass of *Operation* manually. This is especially useful, if you want to add custom methods or properties to your operations. For example, we can customize the string formatting of the transposition:

```
>>> class Transpose(Operation):
...     name = '^T'
...     arity = Arity.unary
...     def __str__(self):
...         return '({})^T'.format(self.operands[0])
```

Lets define the remaining operations:

```
>>> Plus = Operation.new('+', Arity.variadic, 'Plus', one_identity=True, infix=True,␣
↪commutative=True, associative=True)
>>> Inverse = Operation.new('I', Arity.unary, 'Inverse')
```

Finally, we can compose more complex terms:

```
>>> print(Plus(Times(v, Transpose(v)), Times(a, Inverse(M1))))
((a * I(M1)) + (v * (v)^T))
```

Note that the summands are automatically sorted, because *Plus* is commutative.

6.1.3 Wildcards and Variables

In patterns, we can use *wildcards* as a placehold that match anything:

```
>>> _ = Wildcard.dot()
>>> is_match(a, Pattern(_))
True
```

However, for our linear algebra patterns, we want to distinguish between different kinds of symbols. Hence, we can make use of *symbol wildcards*, e.g. to create a wildcard that only matches vectors:

```
>>> _v = Wildcard.symbol(Vector)
>>> is_match(a, Pattern(_v))
False
>>> is_match(v, Pattern(_v))
True
```

We can also assign a name to wildcards and in that case, we call them variables. These names are used to populate the match substitution in case there is a match:

```
>>> x_ = Wildcard.dot('x')
>>> next(match(Times(a, v), Pattern(Times(x_, _v))))
{'x': Scalar('a')}
```

6.1.4 Constraints

Patterns can be limited in what is matched by adding constraints. A constraint is essentially a callback, that gets the match substitution and can return either `True` or `False`. You can either use the *CustomConstraint* class with any (lambda) function, or create your own subclass of *Constraint*.

For example, if we want to only match diagonal matrices with a certain variable, we can create a constraint for that:

```
>>> C_ = Wildcard.symbol('M3', Matrix)
>>> C_is_diagonal_matrix = CustomConstraint(lambda M3: 'diagonal' in M3.properties)
>>> pattern = Pattern(C_, C_is_diagonal_matrix)
```

Then the variable *M3* will only match diagonal matrices:

```
>>> is_match(M1, pattern)
True
>>> is_match(M2, pattern)
False
```

6.1.5 Example: Simplifying multiplication with inverse matrix

Now, we can build patterns to find whatever subexpressions we are interested in. For example, we could remove all occurences of a matrix being multiplied with its inverse. For that we need sequence wildcards. Instead of matching a single term, they can match a sequence of terms. We can create sequence variables like this:

```
>>> ctx1 = Wildcard.plus('ctx1')
>>> ctx2 = Wildcard.star('ctx2')
```

`ctx1` is a plus variables and matches a sequence one or more terms. `ctx2` is a star variables and can match any sequence of terms, including the empty one. With these sequence variables, we can create the rules:

```
>>> x = Wildcard.dot('x')
>>> simplify_matrix_inverse_rules = [
...     ReplacementRule(
...         Pattern(Times(ctx1, x, Inverse(x), ctx2)),
...         lambda ctx1, ctx2, x: Times(*ctx1, *ctx2)
...     ),
...     ReplacementRule(
```

(continues on next page)

(continued from previous page)

```
...         Pattern(Times(ctx2, x, Inverse(x), ctx1)),
...         lambda ctx1, ctx2, x: Times(*ctx2, *ctx1)
...     )
... ]
```

We need two variations of the rule to make sure that we do not accidentially create an empty product. In the first rule, at least one operand must preceed the inverse pair. In the second one, at least one operand must come after it.

For the actual replacement, we can use the *replace_all* function:

```
>>> expr = Times(M1, Inverse(M1), M2)
>>> replace_all(expr, simplify_matrix_inverse_rules)
Matrix('M2')
```

For the case that there are no other factors in the product, we can add another rule that replaces it with the identity matrix:

```
>>> Identity = Matrix('I')
>>> simplify_matrix_inverse_rules.append(
...     ReplacementRule(
...         Pattern(Times(x, Inverse(x))),
...         lambda x: Identity
...     )
... )
```

Lets see this new rule in action:

```
>>> expr2 = Times(M1, Inverse(M1))
>>> replace_all(expr2, simplify_matrix_inverse_rules)
Matrix('I')
```

Because `Times` is associative, these rules even work for more complex expressions:

```
>>> expr3 = Times(M1, M1, M2, Inverse(Times(M1, M2)), M2)
>>> replace_all(expr3, simplify_matrix_inverse_rules)
Times(Matrix('M1'), Matrix('M2'))
```

Note that we can normalize a matrix product inside an inversion by moving it outside, i.e. using the equality $(AB)^{-1} = B^{-1}A^{-1}$:

```
>>> y = Wildcard.dot('y')
>>> simplify_matrix_inverse_rules.append(
...     ReplacementRule(
...         Pattern(Inverse(Times(x, y))),
...         lambda x, y: Times(Inverse(y), Inverse(x))
...     )
... )
```

This allows us to simplify an expression like this:

```
>>> expr4 = Times(M1, M2, Inverse(Times(M3, M1, M2)))
>>> replace_all(expr4, simplify_matrix_inverse_rules)
Inverse(Matrix('M3'))
```

Or this:

```
>>> expr5 = Times(M1, M2, Inverse(Times(M3, M2)))
>>> replace_all(expr5, simplify_matrix_inverse_rules)
Times(Matrix('M1'), Inverse(Matrix('M3')))
```

6.1.6 Example: Finding matches for a BLAS kernel

Lets assume we want to find all subexpressions of some expression which we can compute efficiently with the ?TRMM BLAS routine. These all have the form $\alpha op(A)B$ or $\alpha Bop(A)$ where $op(A)$ is either A or A^T and A is a triangular matrix. Here, we will ignore α and just assume it as 1.

First, we define the variables and constraints we need:

```
>>> A_ = Wildcard.symbol('A', Matrix)
>>> B_ = Wildcard.symbol('B', Matrix)
>>> before_ = Wildcard.star('before')
>>> after_ = Wildcard.star('after')
>>> A_is_triangular = CustomConstraint(lambda A: 'triangular' in A.properties)
```

Then we can construct the patterns, again using context variables to capture the remaining operands:

```
>>> trmm_patterns = [
...     Pattern(Times(before_, A_, B_, after_), A_is_triangular),
...     Pattern(Times(before_, Transpose(A_), B_, after_), A_is_triangular),
...     Pattern(Times(before_, B_, A_, after_), A_is_triangular),
...     Pattern(Times(before_, B_, Transpose(A_), after_), A_is_triangular),
... ]
```

Then, we can find all matching subexpressions using *one_to_one.match*:

```
>>> expr = Times(Transpose(M3), M1, M3, M2)
>>> for i, pattern in enumerate(trmm_patterns):
...     for substitution in match(expr, pattern):
...         print('Pattern {} matched with {} as A and {} as B'.format(i,␣
↪substitution['A'], substitution['B']))
Pattern 0 matched with M3 as A and M2 as B
Pattern 1 matched with M3 as A and M1 as B
Pattern 2 matched with M3 as A and M1 as B
```

6.2 matchpy package

6.2.1 Subpackages

matchpy.expressions package

Submodules

matchpy.expressions.constraints module

Contains several pattern constraint classes.

A pattern constraint is used to further filter which subjects a pattern matches.

The most common use would be the *CustomConstraint*, which wraps a lambda or function to act as a constraint:

```
>>> a_symbol_constraint = CustomConstraint(lambda x: x.name.startswith('a'))
>>> pattern = Pattern(x_, a_symbol_constraint)
>>> is_match(Symbol('a1'), pattern)
True
>>> is_match(Symbol('b1'), pattern)
False
```

There is also the *EqualVariablesConstraint* which will try to unify the substitutions of the variables and only match if it succeeds:

```
>>> equal_constraint = EqualVariablesConstraint('x', 'y')
>>> pattern = Pattern(f(x_, y_), equal_constraint)
>>> is_match(f(a, a), pattern)
True
>>> is_match(f(a, b), pattern)
False
```

You can also create a subclass of the *Constraint* class to create your own custom constraint type.

class Constraint

Bases: object

Base for pattern constraints.

A constraint is essentially a callback, that receives the match Substitution and returns a bool indicating whether the match is valid.

You have to override all the abstract methods if you wish to create your own subclass.

__call__(*match: matchpy.expressions.substitution.Substitution*) → bool

Return True, iff the constraint is fulfilled by the substitution.

Override this in your subclass to define the actual constraint behavior.

Parameters **match** – The (current) match substitution. Note that the matching is done from left to right, so not all variables may have a value yet. You need to override *variables* so that the constraint gets called once all the variables it depends on have a value assigned to them.

Returns True, iff the constraint is fulfilled by the substitution.

__eq__(*other*)

Constraints need to be equatable.

__hash__()

Constraints need to be hashable.

variables

The names of the variables the constraint depends upon.

Used by matchers to decide when a constraint can be evaluated (which is when all the dependency variables have been assigned a value). If the set is empty, the constraint will only be evaluated once the whole match is complete.

with_renamed_vars(*renaming: Dict[str, str]*) → matchpy.expressions.constraints.Constraint

Return a *copy* of the constraint with renamed variables. This is called when the variables in the expression are renamed and hence the ones in the constraint have to be renamed as well. A later invocation of *__call__()* will have the new variable names. You will have to implement this if your constraint needs to use the variables of the match substitution. Note that this can be called multiple times and you might have to account for that. Also, this should not modify the original constraint but rather return a copy. :param renaming: A dictionary mapping old names to new names.

Returns A copy of the constraint with renamed variables.

class EqualVariablesConstraint(**variables*)

Bases: *matchpy.expressions.constraints.Constraint*

A constraint that ensure multiple variables are equal.

The constraint tries to unify the substitutions for the variables and is fulfilled iff that succeeds.

__init__(**variables*) → None

Parameters ***variables** – The names of the variables to check for equality.

variables

The names of the variables the constraint depends upon.

Used by matchers to decide when a constraint can be evaluated (which is when all the dependency variables have been assigned a value). If the set is empty, the constraint will only be evaluated once the whole match is complete.

with_renamed_vars(*renaming*)

Return a *copy* of the constraint with renamed variables. This is called when the variables in the expression are renamed and hence the ones in the constraint have to be renamed as well. A later invocation of `__call__()` will have the new variable names. You will have to implement this if your constraint needs to use the variables of the match substitution. Note that this can be called multiple times and you might have to account for that. Also, this should not modify the original constraint but rather return a copy. :param renaming: A dictionary mapping old names to new names.

Returns A copy of the constraint with renamed variables.

class CustomConstraint(*constraint: Callable[..., bool]*)

Bases: *matchpy.expressions.constraints.Constraint*

Wrapper for lambdas of functions as constraints.

The parameter names have to be the same as the the variable names in the expression:

```
>>> constraint = CustomConstraint(lambda x, y: x.name < y.name)
>>> pattern = Pattern(f(x_, y_), constraint)
>>> is_match(f(a, b), pattern)
True
>>> is_match(f(b, a), pattern)
False
```

The ordering of the parameters is not important. You only need to have the parameters needed for the constraint, not all variables occurring in the pattern.

Note, that the matching happens from left left to right, so not all variables may have been assigned a value when constraint is called. For constraints over multiple variables you should attach the constraint to the last variable occurring in the pattern or a surrounding operation.

__init__(*constraint: Callable[..., bool]*) → None

Parameters **constraint** – The constraint callback.

Raises `ValueError` – If the callback has positional-only or variable parameters (*args and **kwargs).

variables

The names of the variables the constraint depends upon.

Used by matchers to decide when a constraint can be evaluated (which is when all the dependency variables have been assigned a value). If the set is empty, the constraint will only be evaluated once the whole match is complete.

with_renamed_vars(*renaming*)
Return a *copy* of the constraint with renamed variables. This is called when the variables in the expression are renamed and hence the ones in the constraint have to be renamed as well. A later invocation of `__call__()` will have the new variable names. You will have to implement this if your constraint needs to use the variables of the match substitution. Note that this can be called multiple times and you might have to account for that. Also, this should not modify the original constraint but rather return a copy. :param renaming: A dictionary mapping old names to new names.

Returns A copy of the constraint with renamed variables.

matchpy.expressions.expressions module

This module contains the expression classes.

Expressions can be used to model any kind of tree-like data structure. They consist of *operations* and *symbols*. In addition, *patterns* can be constructed, which may additionally, contain *wildcards* and variables.

You can define your own symbols and operations like this:

```
>>> f = Operation.new('f', Arity.variadic)
>>> a = Symbol('a')
>>> b = Symbol('b')
```

Then you can compose expressions out of these:

```
>>> print(f(a, b))
f(a, b)
```

For more information on how to create you own *operations* and *symbols* you can look at their documentation.

Normal expressions are immutable and hence hashable:

```
>>> expr = f(b, x_)
>>> print(expr)
f(b, x_)
>>> hash(expr) == hash(expr)
True
```

Hence, some of the expression's properties are cached and nor updated when you modify them:

```
>>> expr.is_constant
False
>>> expr.operands = [a]
>>> expr.is_constant
False
>>> print(expr)
f(a)
>>> f(a).is_constant
True
```

Therefore, you should modify an expression but rather create a new one:

```
>>> expr2 = type(expr)(*[a])
>>> expr2.is_constant
True
>>> print(expr2)
f(a)
```

class Expression(*variable_name*)

Bases: object

Base class for all expressions.

Do not subclass this class directly but rather *Symbol* or *Operation*. Creating a direct subclass of Expression might break several (matching) algorithms.

head

The head of the expression. For an operation, it is the type of the operation (i.e. a subclass of *Operation*). For wildcards, it is None. For symbols, it is the symbol itself.

Type Optional[Union[type, *Atom*]]

__getitem__(*position: Union[Tuple[int, ...], slice]*) → matchpy.expressions.expressions.Expression

Return the subexpression at the given position(s).

It is also possible to use a slice notation to extract a sequence of subexpressions:

```
>>> expr = f(a, b, a, c)
>>> expr[(1, ):(2, )]
[Symbol('b'), Symbol('a')]
```

Parameters **position** – The position as a tuple. See *preorder_iter()* for its format. Alternatively, a range of positions can be passed using the slice notation.

Returns The subexpression at the given position(s).

Raises IndexError – If the position is invalid, i.e. it refers to a non-existing subexpression.

__init__(*variable_name*)

Initialize self. See help(type(self)) for accurate signature.

collect_symbols(*symbols: multiset.Multiset*) → None

Recursively adds all symbols occuring in the expression to the given multiset.

This is used internally by *symbols*. Needs to be overwritten by inheriting expression classes that can contain symbols. This method can be used when gathering the *symbols* of multiple expressions, because only one multiset needs to be created and that is more efficient.

Parameters **symbols** – Multiset of symbols. All symbols contained in the expression are recursively added to this multiset.

collect_variables(*variables: multiset.Multiset*) → None

Recursively adds all variables occuring in the expression to the given multiset.

This is used internally by *variables*. Needs to be overwritten by inheriting container expression classes. This method can be used when gathering the *variables* of multiple expressions, because only one multiset needs to be created and that is more efficient.

Parameters **variables** – Multiset of variables. All variables contained in the expression are recursively added to this multiset.

is_constant

True, iff the expression does not contain any wildcards.

is_syntactic

True, iff the expression does not contain any associative or commutative operations or sequence wildcards.

preorder_iter(*predicate: Optional[Callable[Expression, bool]] = None*) → Iterator[Tuple[matchpy.expressions.expressions.Expression, Tuple[int, ...]]]

Iterates over all subexpressions that match the (optional) predicate.

Parameters **predicate** – A predicate to filter what expressions are yielded. It gets the expression and if it returns `True`, the expression is yielded.

Yields Every subexpression along with a position tuple. Each item in the tuple is the position of an operation operand:

- `()` is the position of the root element
- `(0, )` that of its first operand
- `(0, 1)` the position of the second operand of the root's first operand.
- etc.

A variable's expression always has the position `0` relative to the variable, i.e. if the root is a variable, then its expression has the position `(0, )`.

symbols
A multiset of the symbol names occurring in the expression.

variables
A multiset of the variables occurring in the expression.

with_renamed_vars(*renaming*) → matchpy.expressions.expressions.Expression
Return a copy of the expression with renamed variables.

class Arity
Bases: `matchpy.expressions.expressions._ArityBase`

Arity of an operator as (`int`, `bool`) tuple.

The first component is the minimum number of operands. If the second component is `True`, the operator has fixed width arity. In that case, the first component describes the fixed number of operands required. If it is `False`, the operator has variable width arity.

binary = Arity(min_count=2, fixed_size=True)

nullary = Arity(min_count=0, fixed_size=True)

polyadic = Arity(min_count=2, fixed_size=False)

ternary = Arity(min_count=3, fixed_size=True)

unary = Arity(min_count=1, fixed_size=True)

variadic = Arity(min_count=0, fixed_size=False)

class Atom(*variable_name*)
Bases: *matchpy.expressions.expressions.Expression*

Base for all atomic expressions.

class Symbol(*name: str*, *variable_name=None*)
Bases: *matchpy.expressions.expressions.Atom*

An atomic constant expression term.

It is uniquely identified by its name.

name
The symbol's name.

Type str

__init__(*name: str*, *variable_name=None*) → None

Parameters **name** – The name of the symbol that uniquely identifies it.

collect_symbols(*symbols*)
Recursively adds all symbols occuring in the expression to the given multiset.

This is used internally by `symbols`. Needs to be overwritten by inheriting expression classes that can contain symbols. This method can be used when gathering the `symbols` of multiple expressions, because only one multiset needs to be created and that is more efficient.

Parameters **symbols** – Multiset of symbols. All symbols contained in the expression are recursively added to this multiset.

with_renamed_vars(*renaming*) → matchpy.expressions.expressions.Symbol
Return a copy of the expression with renamed variables.

class Wildcard(*min_count: int*, *fixed_size: bool*, *variable_name=None*, *optional=None*)
Bases: *matchpy.expressions.expressions.Atom*

A wildcard that matches any expression.

The wildcard will match any number of expressions between *min_count* and *fixed_size*. Optionally, the wildcard can also be constrained to only match expressions satisfying a predicate.

min_count
The minimum number of expressions this wildcard will match.

Type int

fixed_size
If `True`, the wildcard matches exactly *min_count* expressions. If `False`, the wildcard is a sequence wildcard and can match *min_count* or more expressions.

Type bool

__init__(*min_count: int*, *fixed_size: bool*, *variable_name=None*, *optional=None*) → None

Parameters

- **min_count** – The minimum number of expressions this wildcard will match. Must be a non-negative number.
- **fixed_size** – If `True`, the wildcard matches exactly *min_count* expressions. If `False`, the wildcard is a sequence wildcard and can match *min_count* or more expressions.

Raises `ValueError` – if *min_count* is negative or when trying to create a fixed zero-length wildcard.

static dot(*name=None*) → matchpy.expressions.expressions.Wildcard
Create a *Wildcard* that matches a single argument.

Parameters **name** – An optional name for the wildcard.

Returns A dot wildcard.

head = None

static optional(*name*, *default*) → matchpy.expressions.expressions.Wildcard
Create a *Wildcard* that matches a single argument with a default value.

If the wildcard does not match, the substitution will contain the default value instead.

Parameters

- **name** – The name for the wildcard.
- **default** – The default value of the wildcard.

Returns A n optional wildcard.

static plus(*name=None*) → matchpy.expressions.expressions.Wildcard
Creates a *Wildcard* that matches at least one and up to any number of arguments

Parameters **name** – Optional variable name for the wildcard.

Returns A plus wildcard.

static star(*name=None*) → matchpy.expressions.expressions.Wildcard
Creates a *Wildcard* that matches any number of arguments.

Parameters **name** – Optional variable name for the wildcard.

Returns A star wildcard.

static symbol(*name: str = None, symbol_type: Type[matchpy.expressions.expressions.Symbol] = <class 'matchpy.expressions.expressions.Symbol'>*) → matchpy.expressions.expressions.SymbolWildcard
Create a *SymbolWildcard* that matches a single *Symbol* argument.

Parameters

- **name** – Optional variable name for the wildcard.
- **symbol_type** – An optional subclass of *Symbol* to further limit which kind of symbols are matched by the wildcard.

Returns A *SymbolWildcard* that matches the *symbol_type*.

with_renamed_vars(*renaming*) → matchpy.expressions.expressions.Wildcard
Return a copy of the expression with renamed variables.

class Operation(*operands: List[matchpy.expressions.expressions.Expression], variable_name=None*)
Bases: *matchpy.expressions.expressions.Expression*

Base class for all operations.

Do not instantiate this class directly, but create a subclass for every operation in your domain. You can use *new()* as a shortcut for doing so.

__getitem__(*key: Union[Tuple[int, ...], slice]*) → matchpy.expressions.expressions.Expression
Return the subexpression at the given position(s).

It is also possible to use a slice notation to extract a sequence of subexpressions:

```
>>> expr = f(a, b, a, c)
>>> expr[(1, ):(2, )]
[Symbol('b'), Symbol('a')]
```

Parameters **position** – The position as a tuple. See `preorder_iter()` for its format. Alternatively, a range of positions can be passed using the slice notation.

Returns The subexpression at the given position(s).

Raises `IndexError` – If the position is invalid, i.e. it refers to a non-existing subexpression.

__init__(*operands: List[matchpy.expressions.expressions.Expression], variable_name=None*) → None
Create an operation expression.

Parameters ***operands** – The operands for the operation expression.

Raises

- `ValueError` – if the operand count does not match the operation's arity.
- `ValueError` – if the operation contains conflicting variables, i.e. variables with the same name that match different things. A common example would be mixing sequence and fixed variables with the same name in one expression.

arity = Arity(min_count=0, fixed_size=False)
The arity of the operator.

Trying to construct an operation expression with a number of operands that does not fit its operation's arity will result in an error.

Type *Arity*

associative = False
True if the operation is associative, i.e. `f(a, f(b, c)) = f(f(a, b), c)`.

This attribute is used to flatten nested associative operations of the same type. Therefore, the *arity* of an associative operation has to have an unconstrained maximum number of operand.

Type bool

collect_symbols(*symbols*) → None
Recursively adds all symbols occuring in the expression to the given multiset.

This is used internally by `symbols`. Needs to be overwritten by inheriting expression classes that can contain symbols. This method can be used when gathering the `symbols` of multiple expressions, because only one multiset needs to be created and that is more efficient.

Parameters **symbols** – Multiset of symbols. All symbols contained in the expression are recursively added to this multiset.

collect_variables(*variables*) → None
Recursively adds all variables occuring in the expression to the given multiset.

This is used internally by `variables`. Needs to be overwritten by inheriting container expression classes. This method can be used when gathering the `variables` of multiple expressions, because only one multiset needs to be created and that is more efficient.

Parameters **variables** – Multiset of variables. All variables contained in the expression are recursively added to this multiset.

commutative = False
True if the operation is commutative, i.e. `f(a, b) = f(b, a)`.

Note that commutative operations will always be converted into canonical form with sorted operands.

Type bool

head
alias of *Operation*

infix = False
True if the name of the operation should be used as an infix operator by str().

Type bool

name = None
Name or symbol for the operator.

This needs to be overridden in the subclass.

Type str

static new(*name: str*, *arity: matchpy.expressions.expressions.Arity*, *class_name: str = None*, ***, *associative: bool = False*, *commutative: bool = False*, *one_identity: bool = False*, *infix: bool = False*) → Type[matchpy.expressions.expressions.Operation]

Utility method to create a new operation type.

Example:

```
>>> Times = Operation.new('*', Arity.polyadic, 'Times', associative=True,␣
↪commutative=True, one_identity=True)
>>> Times
Times['*', Arity(min_count=2, fixed_size=False), associative, commutative,␣
↪one_identity]
>>> str(Times(Symbol('a'), Symbol('b')))
'*(a, b)'
```

Parameters

- **name** – Name or symbol for the operator. Will be used as name for the new class if `class_name` is not specified.
- **arity** – The arity of the operator as explained in the documentation of *Operation*.
- **class_name** – Name for the new operation class to be used instead of name. This argument is required if *name* is not a valid python identifier.

Keyword Arguments

- **associative** – See *associative*.
- **commutative** – See *commutative*.
- **one_identity** – See *one_identity*.
- **infix** – See *infix*.

Raises `ValueError` – if the class name of the operation is not a valid class identifier.

one_identity = False

True if the operation with a single argument is equivalent to the identity function.

This property is used to simplify expressions, e.g. for `f` with `f.one_identity = True` the expression `f(a)` if simplified to `a`.

Type bool

with_renamed_vars(*renaming*) → matchpy.expressions.expressions.Operation

Return a copy of the expression with renamed variables.

class SymbolWildcard(*symbol_type: Type[matchpy.expressions.expressions.Symbol] = <class 'matchpy.expressions.expressions.Symbol'>*, *variable_name=None*)

Bases: *matchpy.expressions.expressions.Wildcard*

A special *Wildcard* that matches a *Symbol*.

symbol_type

A subclass of *Symbol* to constrain what the wildcard matches. If not specified, the wildcard will match any *Symbol*.

__init__(*symbol_type: Type[matchpy.expressions.expressions.Symbol] = <class 'matchpy.expressions.expressions.Symbol'>*, *variable_name=None*) → None

Parameters **symbol_type** – A subclass of *Symbol* to constrain what the wildcard matches. If not specified, the wildcard will match any *Symbol*.

Raises `TypeError` – if *symbol_type* is not a subclass of *Symbol*.

with_renamed_vars(*renaming*) → matchpy.expressions.expressions.SymbolWildcard
Return a copy of the expression with renamed variables.

class Pattern(*expression*, **constraints*)
Bases: `object`

A pattern is a term that can be matched against another subject term.

A pattern can contain variables and can optionally have constraints attached to it. Those constraints a predicates which limit what the pattern can match.

__init__(*expression*, **constraints*) → None

Parameters

- **expression** – The term that forms the pattern.
- ***constraints** – Optional constraints for the pattern.

global_constraints
The subset of the pattern constraints which are global.

A global constraint does not define dependency variables and can only be evaluated, once the match has been completed.

is_syntactic
True, iff the pattern is *syntactic*.

local_constraints
The subset of the pattern constraints which are local.

A local constraint has a defined non-empty set of dependency variables. These constraints can be evaluated once their dependency variables have a substitution.

make_dot_variable(*name*)
Create a new variable with the given name that matches a single term.

Parameters **name** – The name of the variable

Returns The new dot variable.

make_plus_variable(*name*)
Create a new variable with the given name that matches any number of terms.

Only matches sequences with at least one argument.

Parameters **name** – The name of the variable

Returns The new plus variable.

make_star_variable(*name*)
Create a new variable with the given name that matches any number of terms.

Can also match an empty argument sequence.

Parameters **name** – The name of the variable

Returns The new star variable.

make_symbol_variable(*name*, *symbol_type=<class 'matchpy.expressions.expressions.Symbol'>*)
Create a new variable with the given name that matches a single symbol.

Optionally, a symbol type can be specified to further limit what the variable can match.

Parameters

- **name** – The name of the variable
- **symbol_type** – The symbol type must be a subclass of *Symbol*. Defaults to *Symbol* itself.

Returns The new symbol variable.

class AssociativeOperation
Bases: object

class CommutativeOperation
Bases: object

class OneIdentityOperation
Bases: object

matchpy.expressions.functions module

is_constant(*expression*)
Check if the given expression is constant, i.e. it does not contain Wildcards.

is_syntactic(*expression*)
Check if the given expression is syntactic, i.e. it does not contain sequence wildcards or associative/commutative operations.

get_head(*expression*)
Returns the given expression's head.

match_head(*subject*, *pattern*)
Checks if the head of subject matches the pattern's head.

preorder_iter(*expression*)
Iterate over the expression in preorder.

preorder_iter_with_position(*expression*)
Iterate over the expression in preorder.

Also yields the position of each subexpression.

is_anonymous(*expression*)
Returns True iff the expression does not contain any variables.

contains_variables_from_set(*expression*, *variables*)
Returns True iff the expression contains any of the variables from the given set.

create_operation_expression(*old_operation*, *new_operands*, *variable_name=True*)

rename_variables(*expression: matchpy.expressions.expressions.Expression, renaming: Dict[str, str]*) → matchpy.expressions.expressions.Expression
Rename the variables in the expression according to the given dictionary.

Parameters

- **expression** – The expression in which the variables are renamed.
- **renaming** – The renaming dictionary. Maps old variable names to new ones. Variable names not occuring in the dictionary are left unchanged.

Returns The expression with renamed variables.

op_iter(*operation*)

op_len(*operation*)

get_variables(*expression*, *variables=None*)
Returns the set of variable names in the given expression.

matchpy.expressions.substitution module

Contains the *Substitution* class which is a specialized dictionary.

A substitution maps a variable to a replacement value. The variable is represented by its string name. The replacement can either be a plain expression, a sequence of expressions, or a Multiset of expressions:

```
>>> subst = Substitution({'x': a, 'y': (a, b), 'z': Multiset([a, b])})
>>> print(subst)
{x  a, y  (a, b), z  {a, b}}
```

In addition, the *Substitution* class has some helper methods to unify multiple substitutions and nicer string formatting.

class Substitution
Bases: dict

Special dict for substitutions with nicer formatting.

The key is a variable's name and the value the replacement for it.

extract_substitution(*subject: matchpy.expressions.expressions.Expression*, *pattern: matchpy.expressions.expressions.Expression*) → bool
Extract the variable substitution for the given pattern and subject.

This assumes that subject and pattern already match when being considered as linear. Also, they both must be *syntactic*, as sequence variables cannot be handled here. All that this method does is checking whether all the substitutions for the variables can be unified. So, in case it returns False, the substitution is invalid for the match.

..warning:

```
This method mutates the substitution and will even do so in case the␣
↪extraction fails.

Create a copy before using this method if you need to preserve the original␣
↪substitution.
```

Example

With an empty initial substitution and a linear pattern, the extraction will always succeed:

```
>>> subst = Substitution()
>>> subst.extract_substitution(f(a, b), f(x_, y_))
True
>>> print(subst)
{x  a, y  b}
```

Clashing values for existing variables will fail:

```
>>> subst.extract_substitution(b, x_)
False
```

For non-linear patterns, the extraction can also fail with an empty substitution:

```
>>> subst = Substitution()
>>> subst.extract_substitution(f(a, b), f(x_, x_))
False
>>> print(subst)
{x  a}
```

Note that the initial substitution got mutated even though the extraction failed!

Parameters

- **subject** – A *syntactic* subject that matches the pattern.
- **pattern** – A *syntactic* pattern that matches the subject.

Returns `True` iff the substitution could be extracted successfully.

rename(*renaming: Dict[str, str]*) → matchpy.expressions.substitution.Substitution

Return a copy of the substitution with renamed variables.

Example

Rename the variable *x* to *y*:

```
>>> subst = Substitution({'x': a})
>>> subst.rename({'x': 'y'})
{'y': Symbol('a')}
```

Parameters **renaming** – A dictionary mapping old variable names to new ones.

Returns A copy of the substitution where variable names have been replaced according to the given renaming dictionary. Names that are not contained in the dictionary are left unchanged.

try_add_variable(*variable_name: str, replacement: Union[Tuple[expressions.Expression, ...], multiset.Multiset, expressions.Expression]*) → None

Try to add the variable with its replacement to the substitution.

This considers an existing replacement and will only succeed if the new replacement can be merged with the old replacement. Merging can occur if either the two replacements are equivalent. Replacements can also be merged if the old replacement for the variable_name was unordered (i.e. a `Multiset`) and the new one is an equivalent ordered version of it:

```
>>> subst = Substitution({'x': Multiset(['a', 'b'])})
>>> subst.try_add_variable('x', ('a', 'b'))
>>> print(subst)
{x  (a, b)}
```

Parameters

- **variable** – The name of the variable to add.
- **replacement** – The replacement for the variable.

Raises `ValueError` – if the variable cannot be merged because it conflicts with the existing substitution for the variable_name.

union(**others*) → matchpy.expressions.substitution.Substitution

Try to merge the substitutions.

If a variable occurs in multiple substitutions, try to merge the replacements. See *union_with_variable()* to see how replacements are merged.

Does not modify any of the original substitutions.

Example:

```
>>> subst1 = Substitution({'x': Multiset(['a', 'b']), 'z': a})
>>> subst2 = Substitution({'x': ('a', 'b'), 'y': ('c', )})
>>> print(subst1.union(subst2))
{x  (a, b), y  (c), z  a}
```

Parameters **others** – The other substitutions to merge with this one.

Returns The new substitution with the other substitutions merged.

Raises ValueError – if a variable occurs in multiple substitutions but cannot be merged because the substitutions conflict.

union_with_variable(*variable: str, replacement: Union[Tuple[expressions.Expression, ...], multiset.Multiset, expressions.Expression]*) → matchpy.expressions.substitution.Substitution

Try to create a new substitution with the given variable added.

See *try_add_variable()* for a version of this method that modifies the substitution in place.

Parameters

- **variable_name** – The name of the variable to add.
- **replacement** – The substitution for the variable.

Returns The new substitution with the variable_name added or merged.

Raises ValueError – if the variable cannot be merged because it conflicts with the existing substitution for the variable.

matchpy.matching package

Submodules

matchpy.matching.bipartite module

Contains classes and functions related to bipartite graphs.

The *BipartiteGraph* class is used to represent a bipartite graph as a dictionary. In particular, *BipartiteGraph.find_matching()* can be used to find a maximum matching in such a graph.

The function *enum_maximum_matchings_iter* can be used to enumerate all maximum matchings of a *BipartiteGraph*.

class BipartiteGraph(**args*, ***kwargs*)

Bases: typing.MutableMapping

A bipartite graph representation.

This class is a specialized dictionary, where each edge is represented by a 2-tuple that is used as a key in the dictionary. The value can either be True or any value that you want to associate with the edge.

For example, the edge from 1 to 2 with a label 42 would be set like this:

```
>>> graph = BipartiteGraph()
>>> graph[1, 2] = 42
```

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

as_graph() → None
Returns a `graphviz.Graph` representation of this bipartite graph.

clear() → None. Remove all items from D.

edges()

edges_with_labels()
Returns a view on the edges with labels.

find_matching() → Dict[TLeft, TRight]
Finds a matching in the bipartite graph.

This is done using the Hopcroft-Karp algorithm with an implementation from the `hopcroftkarp` package.

Returns A dictionary where each edge of the matching is represented by a key-value pair with the key being from the left part of the graph and the value from te right part.

limited_to(*left: Set[TLeft], right: Set[TRight]*) → matchpy.matching.bipartite.BipartiteGraph[[TLeft, TRight], TEdgeValue]
Returns the induced subgraph where only the nodes from the given sets are included.

without_edge(*edge: Tuple[TLeft, TRight]*) → matchpy.matching.bipartite.BipartiteGraph[[TLeft, TRight], TEdgeValue]
Returns a copy of this bipartite graph with the given edge removed.

without_nodes(*edge: Tuple[TLeft, TRight]*) → matchpy.matching.bipartite.BipartiteGraph[[TLeft, TRight], TEdgeValue]
Returns a copy of this bipartite graph with the given edge and its adjacent nodes removed.

enum_maximum_matchings_iter(*graph: matchpy.matching.bipartite.BipartiteGraph[[TLeft, TRight], TEdgeValue]*) → Iterator[Dict[TLeft, TRight]]

matchpy.matching.many_to_one module

Contains the *ManyToOneMatcher* which can be used for fast many-to-one matching.

You can initialize the matcher with a list of the patterns that you wish to match:

```
>>> pattern1 = Pattern(f(a, x_))
>>> pattern2 = Pattern(f(y_, b))
>>> matcher = ManyToOneMatcher(pattern1, pattern2)
```

You can also add patterns later:

```
>>> pattern3 = Pattern(f(a, b))
>>> matcher.add(pattern3)
```

A pattern can be added with a label which is yielded instead of the pattern during matching:

```
>>> pattern4 = Pattern(f(x_, y_))
>>> matcher.add(pattern4, "some label")
```

Then you can match a subject against all the patterns at once:

```
>>> subject = f(a, b)
>>> matches = matcher.match(subject)
>>> for matched_pattern, substitution in sorted(map(lambda m: (str(m[0]), str(m[1])),␣
↪matches)):
...     print('{} matched with {}'.format(matched_pattern, substitution))
f(a, b) matched with {}
f(a, x_) matched with {x  b}
f(y_, b) matched with {y  a}
some label matched with {x  a, y  b}
```

Also contains the *ManyToOneReplacer* which can replace a set `ReplacementRule` at one using a *ManyToOneMatcher* for finding the matches.

class ManyToOneMatcher(**patterns*, *rename=True*)

Bases: `object`

__init__(**patterns*, *rename=True*) → None

Parameters ***patterns** – The patterns which the matcher should match.

classmethod _collect_variable_renaming(*expression: matchpy.expressions.expressions.Expression*, *position: List[int] = None*, *variables: Dict[str, str] = None*) → Dict[str, str]

Return renaming for the variables in the expression.

The variable names are generated according to the position of the variable in the expression. The goal is to rename variables in structurally identical patterns so that the automaton contains less redundant states.

_internal_add(*pattern: matchpy.expressions.expressions.Pattern*, *label*, *renaming*) → int

Add a new pattern to the matcher.

Equivalent patterns are not added again. However, patterns that are structurally equivalent, but have different constraints or different variable names are distinguished by the matcher.

Parameters **pattern** – The pattern to add.

Returns The internal id for the pattern. This is mainly used by the `CommutativeMatcher`.

add(*pattern: matchpy.expressions.expressions.Pattern*, *label=None*) → None

Add a new pattern to the matcher.

The optional label defaults to the pattern itself and is yielded during matching. The same pattern can be added with different labels which means that every match for the pattern will result in every associated label being yielded with that match individually.

Equivalent patterns with the same label are not added again. However, patterns that are structurally equivalent, but have different constraints or different variable names are distinguished by the matcher.

Parameters

- **pattern** – The pattern to add.
- **label** – An optional label for the pattern. Defaults to the pattern itself.

as_graph() → None

constraint_vars

constraints

finals

is_match(*subject: matchpy.expressions.expressions.Expression*) → bool
Check if the subject matches any of the matcher's patterns.

Parameters **subject** – The subject to match.

Returns True, if the subject is matched by any of the matcher's patterns. False, otherwise.

match(*subject: matchpy.expressions.expressions.Expression*) → Iterator[Tuple[matchpy.expressions.expressions.Expression, matchpy.expressions.substitution.Substitution]]
Match the subject against all the matcher's patterns.

Parameters **subject** – The subject to match.

Yields For every match, a tuple of the matching pattern and the match substitution.

pattern_vars

patterns

rename

root

states

class ManyToOneReplacer(**rules*)
Bases: object

Class that contains a set of replacement rules and can apply them efficiently to an expression.

__init__(**rules*)
A replacement rule consists of a *pattern*, that is matched against any subexpression of the expression. If a match is found, the *replacement* callback of the rule is called with the variables from the match substitution. Whatever the callback returns is used as a replacement for the matched subexpression. This can either be a single expression or a sequence of expressions, which is then integrated into the surrounding operation in place of the subexpression.

Note that the pattern can therefore not be a single sequence variable/wildcard, because only single expressions will be matched.

Parameters ***rules** – The replacement rules.

add(*rule: matchpy.functions.ReplacementRule*) → None
Add a new rule to the replacer.

Parameters **rule** – The rule to add.

replace(*expression: matchpy.expressions.expressions.Expression*, *max_count: int = inf*) → Union[matchpy.expressions.expressions.Expression, Sequence[matchpy.expressions.expressions.Expression]]
Replace all occurrences of the patterns according to the replacement rules.

Parameters

- **expression** – The expression to which the replacement rules are applied.
- **max_count** – If given, at most *max_count* applications of the rules are performed. Otherwise, the rules are applied until there is no more match. If the set of replacement rules is not confluent, the replacement might not terminate without a *max_count* set.

Returns The resulting expression after the application of the replacement rules. This can also be a sequence of expressions, if the root expression is replaced with a sequence of expressions by a rule.

replace_post_order(*expression: matchpy.expressions.expressions.Expression*) → Union[matchpy.expressions.expressions.Expression, Sequence[matchpy.expressions.expressions.Expression]]

Replace all occurrences of the patterns according to the replacement rules.

Replaces innermost expressions first.

Parameters

- **expression** – The expression to which the replacement rules are applied.
- **max_count** – If given, at most *max_count* applications of the rules are performed. Otherwise, the rules are applied until there is no more match. If the set of replacement rules is not confluent, the replacement might not terminate without a *max_count* set.

Returns The resulting expression after the application of the replacement rules. This can also be a sequence of expressions, if the root expression is replaced with a sequence of expressions by a rule.

matchpy.matching.one_to_one module

match(*subject: matchpy.expressions.expressions.Expression*, *pattern: matchpy.expressions.expressions.Pattern*) → Iterator[matchpy.expressions.substitution.Substitution]

Tries to match the given *pattern* to the given *subject*.

Yields each match in form of a substitution.

Parameters

- **subject** – An subject to match.
- **pattern** – The pattern to match.

Yields All possible match substitutions.

Raises `ValueError` – If the subject is not constant.

match_anywhere(*subject: matchpy.expressions.expressions.Expression*, *pattern: matchpy.expressions.expressions.Pattern*) → Iterator[Tuple[matchpy.expressions.substitution.Substitution, Tuple[int, ...]]]

Tries to match the given *pattern* to the any subexpression of the given *subject*.

Yields each match in form of a substitution and a position tuple. The position is a tuple of indices, e.g. the empty tuple refers to the *subject* itself, `(0, )` refers to the first child (operand) of the subject, `(0, 0)` to the first child of the first child etc.

Parameters

- **subject** – An subject to match.
- **pattern** – The pattern to match.

Yields All possible substitution and position pairs.

Raises `ValueError` – If the subject is not constant.

matchpy.matching.syntactic module

This module contains various many-to-one matchers for syntactic patterns:

- There *DiscriminationNet* class that is a many-to-one matcher for syntactic patterns.

- The *SequenceMatcher* can be used to match patterns with a common surrounding operation with some fixed syntactic patterns.
- The *FlatTerm* representation for an expression flattens the expression's tree structure and allows faster preoder traversal.

Furthermore, the module contains some utility functions for working with flatterms.

class FlatTerm(*expression: Union[matchpy.expressions.expressions.Expression, Sequence[Union[matchpy.expressions.expressions.Symbol, matchpy.expressions.expressions.Wildcard, Type[matchpy.expressions.expressions.Operation], Type[matchpy.expressions.expressions.Symbol], str]]]*)

Bases: typing.Sequence

A flattened representation of an *Expression*.

This is a subclass of list. This representation is similar to the prefix notation generated by *Expression.preorder_iter()*, but contains some additional elements.

Operation expressions are represented by the `type()` of the operation before the operands as well as `OPERATION_END` after the last operand of the operation:

```
>>> FlatTerm(f(a, b))
[f, a, b, )]
```

Variables are not included in the flatterm representation, only wildcards remain.

```
>>> FlatTerm(x_)
[_]
```

Consecutive wildcards are merged, as the *DiscriminationNet* cannot handle multiple consecutive sequence wildcards:

```
>>> FlatTerm(f(_, _))
[f, _[2], )]
>>> FlatTerm(f(_, __, __))
[f, _[3+], )]
```

Furthermore, every `SymbolWildcard` is replaced by its `symbol_type`:

```
>>> class SpecialSymbol(Symbol):
...     pass
>>> _s = Wildcard.symbol(SpecialSymbol)
>>> FlatTerm(_s)
[<class '__main__.SpecialSymbol'>]
```

Symbol wildcards are also not merged like other wildcards, because they can never be sequence wildcards:

```
>>> FlatTerm(f(_, _s))
[f, _, <class '__main__.SpecialSymbol'>, )]
```

__init__(*expression: Union[matchpy.expressions.expressions.Expression, Sequence[Union[matchpy.expressions.expressions.Symbol, matchpy.expressions.expressions.Wildcard, Type[matchpy.expressions.expressions.Operation], Type[matchpy.expressions.expressions.Symbol], str]]]*) → None

Initialize self. See help(type(self)) for accurate signature.

static _combined_wildcards_iter(*flatterm: Iterator[Union[matchpy.expressions.expressions.Symbol, matchpy.expressions.expressions.Wildcard, Type[matchpy.expressions.expressions.Operation], Type[matchpy.expressions.expressions.Symbol], str]]*) → Iterator[Union[matchpy.expressions.expressions.Symbol, matchpy.expressions.expressions.Wildcard, Type[matchpy.expressions.expressions.Operation], Type[matchpy.expressions.expressions.Symbol], str]]

Combine consecutive wildcards in a flatterm into a single one.

classmethod _flatterm_iter(*expression: matchpy.expressions.expressions.Expression*) → Iterator[Union[matchpy.expressions.expressions.Symbol, matchpy.expressions.expressions.Wildcard, Type[matchpy.expressions.expressions.Operation], Type[matchpy.expressions.expressions.Symbol], str]]

Generator that yields the atoms of the expressions in prefix notation with operation end markers.

classmethod empty() → matchpy.matching.syntactic.FlatTerm

An empty flatterm.

is_syntactic

True, iff the flatterm is *syntactic*.

classmethod merged(**flatterms*) → matchpy.matching.syntactic.FlatTerm

Concatenate the given flatterms to a single flatterm.

Parameters ***flatterms** – The flatterms which are concatenated.

Returns The concatenated flatterms.

is_operation(*term: Any*) → bool

Return True iff the given term is a subclass of *Operation*.

is_symbol_wildcard(*term: Any*) → bool

Return True iff the given term is a subclass of *Symbol*.

class DiscriminationNet(**patterns*)

Bases: typing.Generic

An automaton to distinguish which patterns match a given expression.

This is a DFA with an implicit fail state whenever a transition is not actually defined. For every pattern added, an automaton is created and then the product automaton with the existing one is used as the new automaton.

The matching assumes that patterns are linear, i.e. it will treat all variables as non-existent and only consider the wildcards.

__init__(**patterns*) → None

Parameters ***patterns** – Optional pattern to initially add to the discrimination net.

classmethod _generate_net(*flatterm: matchpy.matching.syntactic.FlatTerm, final_label: T*) → matchpy.matching.syntactic._State[T]

Generates a DFA matching the given pattern.

add(*pattern: Union[matchpy.expressions.expressions.Pattern, matchpy.matching.syntactic.FlatTerm], final_label: T = None*) → int

Add a pattern to the discrimination net.

Parameters

- **pattern** – The pattern which is added to the DiscriminationNet. If an expression is given, it will be converted to a *FlatTerm* for internal processing. You can also pass a *FlatTerm* directly.

- **final_label** – A label that is returned if the pattern matches when using `match()`. This will default to the pattern itself.

Returns The index of the newly added pattern. This is used internally to later to get the pattern and its final label once a match is found.

as_graph() → None
Renders the discrimination net as graphviz digraph.

is_match(*subject: Union[matchpy.expressions.expressions.Expression, matchpy.matching.syntactic.FlatTerm]*) → bool
Check if the given subject matches any pattern in the net.

Parameters **subject** – The subject that is matched. Must be constant.

Returns True, if any pattern matches the subject.

match(*subject: Union[matchpy.expressions.expressions.Expression, matchpy.matching.syntactic.FlatTerm]*) → Iterator[Tuple[T, matchpy.expressions.substitution.Substitution]]
Match the given subject against all patterns in the net.

Parameters **subject** – The subject that is matched. Must be constant.

Yields A tuple `(final label, substitution)`, where the first component is the final label associated with the pattern as given when using `add()` and the second one is the match substitution.

class SequenceMatcher(**patterns*)
Bases: `object`

A matcher that matches many *syntactic* patterns in a surrounding sequence.

It can match patterns that have the form $f(x^*, s_1, \ldots, s_n, y^*)$ where

- f is a non-commutative operation,
- x^*, y^* are star sequence wildcards or variables (they can be the same of different), and
- all the s_i are syntactic patterns.

After adding these patterns with `add()`, they can be matched simultaneously against a subject with `match()`. Note that all patterns matched by one sequence matcher must have the same outer operation f.

operation
The outer operation that all patterns have in common. Is set automatically when adding the first pattern and is check for all following patterns.

__init__(**patterns*) → None

Parameters ***patterns** – Initial patterns to add to the sequence matcher.

add(*pattern: matchpy.expressions.expressions.Pattern*) → int
Add a pattern that will be recognized by the matcher.

Parameters **pattern** – The pattern to add.

Returns An internal index for the pattern.

Raises

- `ValueError` – If the pattern does not have the correct form.
- `TypeError` – If the pattern is not a non-commutative operation.

as_graph() → None
Renders the underlying discrimination net as graphviz digraph.

classmethod can_match(*pattern: matchpy.expressions.expressions.Pattern*) → bool
Check if a pattern can be matched with a sequence matcher.

Parameters **pattern** – The pattern to check.

Returns True, iff the pattern can be matched with a sequence matcher.

match(*subject: matchpy.expressions.expressions.Expression*) → Iterator[Tuple[matchpy.expressions.expressions.Pattern, matchpy.expressions.substitution.Substitution]]
Match the given subject against all patterns in the sequence matcher.

Parameters **subject** – The subject that is matched. Must be constant.

Yields A tuple `(pattern, substitution)` for every matching pattern.

operation

6.2.2 Submodules

matchpy.functions module

This module contains various functions for working with expressions.

- With *substitute()* you can replace occurrences of variables with an expression or sequence of expressions.
- With *replace()* you can replace a subexpression at a specific position with a different expression or sequence of expressions.
- With *replace_many()* works the same as *replace()*, but you can replace multiple positions at once.
- With *replace_all()* you can apply a set of replacement rules repeatedly to an expression.
- With *is_match()* you can check whether a pattern matches a subject expression.

substitute(*expression: Union[matchpy.expressions.expressions.Expression, matchpy.expressions.expressions.Pattern], substitution: matchpy.expressions.substitution.Substitution*) → Union[matchpy.expressions.expressions.Expression, List[matchpy.expressions.expressions.Expression]]
Replaces variables in the given *expression* using the given *substitution*.

```
>>> print(substitute(f(x_), {'x': a}))
f(a)
```

If nothing was substituted, the original expression is returned:

```
>>> expression = f(x_)
>>> result = substitute(expression, {'y': a})
>>> print(result)
f(x_)
>>> expression is result
True
```

Note that this function returns a list of expressions iff the expression is a variable and its substitution is a list of expressions. In other cases were a substitution is a list of expressions, the expressions will be integrated as operands in the surrounding operation:

```
>>> print(substitute(f(x_, c), {'x': [a, b]}))
f(a, b, c)
```

If you substitute with a `Multiset` of values, they will be sorted:

```
>>> replacement = Multiset([b, a, b])
>>> print(substitute(f(x_, c), {'x': replacement}))
f(a, b, b, c)
```

Parameters

- **expression** – An expression in which variables are substituted.
- **substitution** – A substitution dictionary. The key is the name of the variable, the value either an expression or a list of expression to use as a replacement for the variable.

Returns The expression resulting from applying the substitution.

replace(*expression: matchpy.expressions.expressions.Expression, position: Sequence[int], replacement: Union[matchpy.expressions.expressions.Expression, List[matchpy.expressions.expressions.Expression]]*) → Union[matchpy.expressions.expressions.Expression, List[matchpy.expressions.expressions.Expression]]

Replaces the subexpression of `expression` at the given `position` with the given `replacement`.

The original `expression` itself is not modified, but a modified copy is returned. If the replacement is a list of expressions, it will be expanded into the list of operands of the respective operation:

```
>>> print(replace(f(a), (0, ), [b, c]))
f(b, c)
```

Parameters

- **expression** – An `Expression` where a (sub)expression is to be replaced.
- **position** – A tuple of indices, e.g. the empty tuple refers to the `expression` itself, `(0, )` refers to the first child (operand) of the `expression`, `(0, 0)` to the first child of the first child etc.
- **replacement** – Either an `Expression` or a list of `Expressions` to be inserted into the `expression` instead of the original expression at that `position`.

Returns The resulting expression from the replacement.

Raises `IndexError` – If the position is invalid or out of range.

replace_all(*expression: matchpy.expressions.expressions.Expression, rules: Iterable[matchpy.functions.ReplacementRule], max_count: int = inf*) → Union[matchpy.expressions.expressions.Expression, Sequence[matchpy.expressions.expressions.Expression]]

Replace all occurrences of the patterns according to the replacement rules.

A replacement rule consists of a *pattern*, that is matched against any subexpression of the expression. If a match is found, the *replacement* callback of the rule is called with the variables from the match substitution. Whatever the callback returns is used as a replacement for the matched subexpression. This can either be a single expression or a sequence of expressions, which is then integrated into the surrounding operation in place of the subexpression.

Note that the pattern can therefore not be a single sequence variable/wildcard, because only single expressions will be matched.

Parameters

- **expression** – The expression to which the replacement rules are applied.
- **rules** – A collection of replacement rules that are applied to the expression.

- **max_count** – If given, at most *max_count* applications of the rules are performed. Otherwise, the rules are applied until there is no more match. If the set of replacement rules is not confluent, the replacement might not terminate without a *max_count* set.

Returns The resulting expression after the application of the replacement rules. This can also be a sequence of expressions, if the root expression is replaced with a sequence of expressions by a rule.

replace_many(*expression: matchpy.expressions.expressions.Expression, replacements: Sequence[Tuple[Sequence[int], Union[matchpy.expressions.expressions.Expression, List[matchpy.expressions.expressions.Expression]]]]*) → Union[matchpy.expressions.expressions.Expression, List[matchpy.expressions.expressions.Expression]]

Replaces the subexpressions of *expression* at the given positions with the given replacements.

The original *expression* itself is not modified, but a modified copy is returned. If the replacement is a sequence of expressions, it will be expanded into the list of operands of the respective operation.

This function works the same as *replace*, but allows multiple positions to be replaced at the same time. However, compared to just replacing each position individually with *replace*, this does work when positions are modified due to replacing a position with a sequence:

```
>>> expr = f(a, b)
>>> expected_result = replace_many(expr, [((0, ), [c, c]), ((1, ), a)])
>>> print(expected_result)
f(c, c, a)
```

However, using *replace* for one position at a time gives the wrong result:

```
>>> step1 = replace(expr, (0, ), [c, c])
>>> print(step1)
f(c, c, b)
>>> step2 = replace(step1, (1, ), a)
>>> print(step2)
f(c, a, b)
```

Parameters

- **expression** – An `Expression` where a (sub)expression is to be replaced.
- **replacements** – A collection of tuples consisting of a position in the expression and a replacement for that position. With just a single replacement pair, this is equivalent to using *replace*:

```
>>> replace(a, (), b) == replace_many(a, [((), b)])
True
```

Returns The resulting expression from the replacements.

Raises

- `IndexError` – If a position is invalid or out of range or if you try to replace a subterm of a term you are
- already replacing.

is_match(*subject: matchpy.expressions.expressions.Expression, pattern: matchpy.expressions.expressions.Expression*) → bool

Check whether the given *subject* matches given *pattern*.

Parameters

- **subject** – The subject.
- **pattern** – The pattern.

Returns True iff the subject matches the pattern.

class ReplacementRule(*pattern*, *replacement*)

Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *pattern*, *replacement*)
Create new instance of ReplacementRule(pattern, replacement)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new ReplacementRule object from a sequence or iterable

_replace(***kwds*)
Return a new ReplacementRule object replacing specified fields with new values

pattern
Alias for field number 0

replacement
Alias for field number 1

replace_all_post_order(*expression: matchpy.expressions.expressions.Expression*, *rules: Iterable[matchpy.functions.ReplacementRule]*) → Union[matchpy.expressions.expressions.Expression, Sequence[matchpy.expressions.expressions.Expression]]

Replace all occurrences of the patterns according to the replacement rules.

A replacement rule consists of a *pattern*, that is matched against any subexpression of the expression. If a match is found, the *replacement* callback of the rule is called with the variables from the match substitution. Whatever the callback returns is used as a replacement for the matched subexpression. This can either be a single expression or a sequence of expressions, which is then integrated into the surrounding operation in place of the subexpression.

Note that the pattern can therefore not be a single sequence variable/wildcard, because only single expressions will be matched.

Parameters

- **expression** – The expression to which the replacement rules are applied.
- **rules** – A collection of replacement rules that are applied to the expression.
- **max_count** – If given, at most *max_count* applications of the rules are performed. Otherwise, the rules are applied until there is no more match. If the set of replacement rules is not confluent, the replacement might not terminate without a *max_count* set.

Returns The resulting expression after the application of the replacement rules. This can also be a sequence of expressions, if the root expression is replaced with a sequence of expressions by a rule.

matchpy.utils module

This module contains various utility functions.

fixed_integer_vector_iter(*max_vector: Tuple[int, ...]*, *vector_sum: int*) → Iterator[Tuple[int, ...]]

Return an iterator over the integer vectors which

- are componentwise less than or equal to *max_vector*, and
- are non-negative, and where
- the sum of their components is exactly *vector_sum*.

The iterator yields the vectors in lexicographical order.

Examples

List all vectors that are between `(0, 0)` and `(2, 2)` componentwise, where the sum of components is 2:

```
>>> vectors = list(fixed_integer_vector_iter([2, 2], 2))
>>> vectors
[(0, 2), (1, 1), (2, 0)]
>>> list(map(sum, vectors))
[2, 2, 2]
```

Parameters

- **max_vector** – Maximum vector for the iteration. Every yielded result will be less than or equal to this componentwise.
- **vector_sum** – Every iterated vector will have a component sum equal to this value.

Yields All non-negative vectors that have the given sum and are not larger than the given maximum.

Raises `ValueError` – If *vector_sum* is negative.

weak_composition_iter(*n: int*, *num_parts: int*) → Iterator[Tuple[int, ...]]

Yield all weak compositions of integer *n* into *num_parts* parts.

Each composition is yielded as a tuple. The generated partitions are order-dependant and not unique when ignoring the order of the components. The partitions are yielded in lexicographical order.

Example

```
>>> compositions = list(weak_composition_iter(5, 2))
>>> compositions
[(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)]
```

We can easily verify that all compositions are indeed valid:

```
>>> list(map(sum, compositions))
[5, 5, 5, 5, 5, 5]
```

The algorithm was adapted from an answer to this Stackoverflow question.

Parameters

- **n** – The integer to partition.

- **num_parts** – The number of parts for the combination.

Yields All non-negative tuples that have the given sum and size.

Raises `ValueError` – If *n* or *num_parts* are negative.

commutative_sequence_variable_partition_iter(*values: multiset.Multiset, variables: List[matchpy.utils.VariableWithCount]*) → Iterator[Dict[str, multiset.Multiset]]

Yield all possible variable substitutions for given values and variables.

Note: The results are not yielded in any particular order because the algorithm uses dictionaries. Dictionaries until Python 3.6 do not keep track of the insertion order.

Example

For a subject like `fc(a, a, a, b, b, c)` and a pattern like `f(x__, y___, y___)` one can define the following input parameters for the partitioning:

```
>>> x = VariableWithCount(name='x', count=1, minimum=1, default=None)
>>> y = VariableWithCount(name='y', count=2, minimum=0, default=None)
>>> values = Multiset('aaabbc')
```

Then the solutions are found (and sorted to get a unique output):

```
>>> substitutions = commutative_sequence_variable_partition_iter(values, [x, y])
>>> as_strings = list(str(Substitution(substitution)) for substitution in␣
↪substitutions)
>>> for substitution in sorted(as_strings):
...     print(substitution)
{x  {a, a, a, b, b, c}, y  {}}
{x  {a, a, a, c}, y  {b}}
{x  {a, b, b, c}, y  {a}}
{x  {a, c}, y  {a, b}}
```

Parameters

- **values** – The multiset of values which are partitioned and distributed among the variables.
- **variables** – A list of the variables to distribute the values among. Each variable has a name, a count of how many times it occurs and a minimum number of values it needs.

Yields Each possible substitutions that is a valid partitioning of the values among the variables.

get_short_lambda_source(*lambda_func: function*) → Optional[str]

Return the source of a (short) lambda function. If it's impossible to obtain, return `None`.

The source is returned without the `lambda` and signature parts:

```
>>> get_short_lambda_source(lambda x, y: x < y)
'x < y'
```

This should work well for most lambda definitions, however for multi-line or highly nested lambdas, the source extraction might not succeed.

Parameters **lambda_func** – The lambda function.

Returns The source of the lambda function without its signature.

solve_linear_diop(*total: int*, **coeffs*) → Iterator[Tuple[int, ...]]

Yield non-negative integer solutions of a linear Diophantine equation of the format $c_1x_1 + \cdots + c_nx_n = total$.

If there are at most two coefficients, *base_solution_linear()* is used to find the solutions. Otherwise, the solutions are found recursively, by reducing the number of variables in each recursion:

1. Compute $d := gcd(c_2, \ldots, c_n)$
2. Solve $c_1x + dy = total$
3. Recursively solve $c_2x_2 + \cdots + c_nx_n = y$ for each solution for y
4. Combine these solutions to form a solution for the whole equation

Parameters

- **total** – The constant of the equation.
- ***coeffs** – The coefficients c_i of the equation.

Yields The non-negative integer solutions of the equation as a tuple $(x_1, \ldots, x_n)$.

generator_chain(*initial_data: T*, **factories*) → Iterator[T]

Chain multiple generators together by passing results from one to the next.

This helper function allows to create a chain of generator where each generator is constructed by a factory that gets the data yielded by the previous generator. So each generator can generate new data dependant on the data yielded by the previous one. For each data item yielded by a generator, a new generator is constructed by the next factory.

Example

Lets say for every number from 0 to 4, we want to count up to that number. Then we can do something like this using list comprehensions:

```
>>> [i for n in range(1, 5) for i in range(1, n + 1)]
[1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
```

You can use this function to achieve the same thing:

```
>>> list(generator_chain(5, lambda n: iter(range(1, n)), lambda i: iter(range(1,␣
↪i + 1))))
[1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
```

The advantage is, that this is independent of the number of dependant generators you have. Also, this function does not use recursion so it is safe to use even with large generator counts.

Parameters

- **initial_data** – The initial data that is passed to the first generator factory.
- ***factories** – The generator factories. Each of them gets passed its predecessors data and has to return an iterable. The data from this iterable is passed to the next factory.

Yields Every data item yielded by the generators of the final factory.

class cached_property(*getter*, *slot=None*)

Bases: property

Property with caching.

An extension of the builtin `property`, that caches the value after the first access. This is useful in case the computation of the property value is expensive.

Use it just like a regular property decorator. Cached properties cannot have a setter.

Example

First, create a class with a cached property:

```
>>> class MyClass:
...     @cached_property
...     def my_property(self):
...         print('my_property called')
...         return 42
>>> instance = MyClass()
```

Then, access the property and get the computed value:

```
>>> instance.my_property
my_property called
42
```

Now the result is cached and the original method will not be called again:

```
>>> instance.my_property
42
```

__init__(*getter*, *slot=None*)

Use it as a decorator:

```
>>> class MyClass:
...     @cached_property
...     def my_property(self):
...         return 42
```

The *slot* argument specifies a class slot to use for caching the property. You should use the *slot_cached_property* decorator instead as that is more convenient.

Parameters

- **getter** – The getter method for the property.
- **slot** – Optional slot to use for the cached value. Only relevant in classes that use slots. Use *slot_cached_property* instead.

Returns The wrapped `property` with caching.

slot_cached_property(*slot*)

Property with caching for classes with slots.

This is a wrapper around *cached_property* to be used with classes that have slots. It provides an extension of the builtin `property`, that caches the value in a slot after the first access. You need to specify which slot to use for the cached value.

Example

First, create a class with a cached property and a slot to hold the cached value:

```
>>> class MyClass:
...     __slots__ = ('_my_cached_property', )
...
...     @slot_cached_property('_my_cached_property')
...     def my_property(self):
...         print('my_property called')
...         return 42
...
>>> instance = MyClass()
```

Then, access the property and get the computed value:

```
>>> instance.my_property
my_property called
42
```

Now the result is cached and the original method will not be called again:

```
>>> instance.my_property
42
```

Parameters **slot** – The name of the classes slot to use for the cached value.

Returns The wrapped *cached_property*.

extended_euclid(*a: int*, *b: int*) → Tuple[int, int, int]

Extended Euclidean algorithm that computes the Bézout coefficients as well as $gcd(a, b)$

Returns `x, y, d` where x and y are a solution to $ax + by = d$ and $d = gcd(a, b)$. x and y are a minimal pair of Bézout's coefficients.

See Extended Euclidean algorithm or Bézout's identity for more information.

Example

Compute the Bézout coefficients and GCD of 42 and 12:

```
>>> a, b = 42, 12
>>> x, y, d = extended_euclid(a, b)
>>> x, y, d
(1, -3, 6)
```

Verify the results:

```
>>> import math
>>> d == math.gcd(a, b)
True
>>> a * x + b * y == d
True
```

Parameters

- **a** – The first integer.
- **b** – The second integer.

Returns A tuple with the Bézout coefficients and the greatest common divider of the arguments.

base_solution_linear(*a: int*, *b: int*, *c: int*) → Iterator[Tuple[int, int]]

Yield solutions for a basic linear Diophantine equation of the form $ax + by = c$.

First, the equation is normalized by dividing a, b, c by their gcd. Then, the extended Euclidean algorithm (*extended_euclid()*) is used to find a base solution (x_0, y_0).

All non-negative solutions are generated by using that the general solution is $(x_0 + bt, y_0 - at)$. Because the base solution is one of the minimal pairs of Bézout's coefficients, for all non-negative solutions either $t \geq 0$ or $t \leq 0$ must hold. Also, all the non-negative solutions are consecutive with respect to t.

Hence, by adding or subtracting a resp. b from the base solution, all non-negative solutions can be efficiently generated.

Parameters

- **a** – The first coefficient of the equation.
- **b** – The second coefficient of the equation.
- **c** – The constant of the equation.

Yields Each non-negative integer solution of the equation as a tuple `(x, y)`.

Raises `ValueError` – If any of the coefficients is not a positive integer.

6.3 Glossary

syntactic An *Expression* is syntactic iff it contains neither associative nor commutative operations and also does not contain sequence *wildcards* (i.e. *wildcards* with *fixed_size* set to `False`).

constant An *Expression* is *constant* iff it does not contain any *Wildcard*.

CHAPTER 7

Indices and Tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W