pyfuncol

Release 1.3.1

Andrea Veneziano

May 03, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

pyfuncol is a Python functional collections library. It *extends* collections built-in types with useful methods to write functional Python code. It uses Forbidden Fruit under the hood.

pyfuncol provides:

- Standard "eager" methods, such as `map`, `flat_map`, `group_by`, etc.
- Parallel methods, such as `par_map`, `par_flat_map`, etc.
- Pure methods that leverage memoization to improve performance, such as `pure_map`, `pure_flat_map`, etc.
- Lazy methods that return iterators and never materialize results, such as `lazy_map`, `lazy_flat_map`, etc.

pyfuncol can also be *used without forbiddenfruit*.

CHAPTER

TWO

GET STARTED

pyfuncol is supported only on Python 3.6+ and CPython (+ forbiddenfruit) is required for extending builtins. Albeit untested, it might still work on older Python versions.

To install it, run:

```
pip install pyfuncol
```

CHAPTER

THREE

EXAMPLES

> **Note:** If you are not using forbiddenfruit, the functions will not extend the builtins. Please see here for usage without forbiddenfruit

To use the methods, you just need to import pyfuncol. Some examples:

```
import pyfuncol

[1, 2, 3, 4].map(lambda x: x * 2).filter(lambda x: x > 4)
# [6, 8]

[1, 2, 3, 4].fold_left(0, lambda acc, n: acc + n)
# 10

{1, 2, 3, 4}.map(lambda x: x * 2).filter_not(lambda x: x <= 4)
# {6, 8}

["abc", "def", "e"].group_by(lambda s: len(s))
# {3: ["abc", "def"], 1: ["e"]}

{"a": 1, "b": 2, "c": 3}.flat_map(lambda kv: {kv[0]: kv[1] ** 2})
# {"a": 1, "b": 4, "c": 9}
```

pyfuncol provides parallel operations (for now `par_map`, `par_flat_map`, `par_filter` and `par_filter_not`):

```
[1, 2, 3, 4].par_map(lambda x: x * 2).par_filter(lambda x: x > 4)
# [6, 8]

{1, 2, 3, 4}.par_map(lambda x: x * 2).par_filter_not(lambda x: x <= 4)
# {6, 8}

{"a": 1, "b": 2, "c": 3}.par_flat_map(lambda kv: {kv[0]: kv[1] ** 2})
# {"a": 1, "b": 4, "c": 9}
```

pyfuncol provides operations leveraging memoization to improve performance (for now `pure_map`, `pure_flat_map`, `pure_filter` and `pure_filter_not`). These versions work only for **pure** functions (i.e., all calls to the same args return the same value) on hashable inputs:

```
[1, 2, 3, 4].pure_map(lambda x: x * 2).pure_filter(lambda x: x > 4)
# [6, 8]

{1, 2, 3, 4}.pure_map(lambda x: x * 2).pure_filter_not(lambda x: x <= 4)
```

(continues on next page)

(continued from previous page)

```
# {6, 8}

{"a": 1, "b": 2, "c": 3}.pure_flat_map(lambda kv: {kv[0]: kv[1] ** 2})
# {"a": 1, "b": 4, "c": 9}
```

pyfuncol provides lazy operations that never materialize results:

```
list([1, 2, 3, 4].lazy_map(lambda x: x * 2).lazy_filter(lambda x: x > 4))
# [6, 8]

list({1, 2, 3, 4}.lazy_map(lambda x: x * 2).lazy_filter_not(lambda x: x <= 4))
# [6, 8]

list({"a": 1, "b": 2, "c": 3}.lazy_flat_map(lambda kv: {kv[0]: kv[1] ** 2}))
# [("a", 1), ("b", 4), ("c", 9)]

set([1, 2, 3, 4].lazy_map(lambda x: x * 2).lazy_filter(lambda x: x > 4))
# {6, 8}
```

We support all subclasses with default constructors (`OrderedDict`, for example).

3.1 Usage without forbiddenfruit

If you are using a Python intepreter other than CPython, forbiddenfruit will not work.

Fortunately, if forbiddenfruit does not work on your installation or if you do not want to use it, pyfuncol also supports direct function calls without extending builtins.

```
from pyfuncol import list as pfclist

pfclist.map([1, 2, 3], lambda x: x * 2)
# [2, 4, 6]
```

CHAPTER

FOUR

CONTRIBUTING

4.1 Setup

Fork the repo, then install all development requirements with:

```
pip install -r development.txt
```

When your changes are ready, submit a pull request!

4.2 Style

For formatting and code style, we use black. Docstrings should follow the Google Python Style Guide.

4.3 Tests

To run the tests, execute `pytest` at the root of the project.

To run the tests with coverage enabled, execute:

```
pytest --cov-config=.coveragerc --cov=pyfuncol --cov-report=xml
```

4.4 Docs

The docs are hosted on Read the Docs. Source files are in `docs/source/`.

To build them locally, run in `docs/`:

```
make html
```

The HTML files will be stored in `docs/build/`.

4.5 Project structure

```
── docs - Contains the docs source code
├── pyfuncol - Contains all the library source code
    ├── tests - Contains tests for all the modules
    ├── __init__.py - Contains the function calls that extend built-in types
    ├── dict.py - Contains extension functions for dictionaries
    ├── list.py - Contains extension functions for lists
    └── ...
└── ...
```

4.6 Release

To publish a new release on PyPI:

1. Update the version in `setup.py`
2. Update the version (`release` field) in `docs/source/conf.py`
3. Create a new release on GitHub. The newly created tag and the release title should match the version in `setup.py` and `docs/source/conf.py` with 'v' prepended. An example: for version `1.1.1`, the tag and release title should be `v1.1.1`.

The GitHub release creation will trigger the deploy workflow that builds and uploads the project to PyPI.

4.7 Code of Conduct

Our Code of Conduct is here. By contributing to pyfuncol, you implicitly accept it.

CHAPTER

FIVE

API REFERENCE

5.1 pyfuncol package

5.1.1 Submodules

5.1.2 pyfuncol.dict module

`pyfuncol.dict.contains`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*, *key: pyfuncol.dict.A*) → bool

Tests whether this dict contains a binding for a key.

Parameters **key** – The key to find.

Returns True if the dict contains a binding for the key, False otherwise.

`pyfuncol.dict.size`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*) → int

Computes the size of this dict.

Returns The size of the dict.

`pyfuncol.dict.filter`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*, *p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Dict[pyfuncol.dict.A, pyfuncol.dict.B]

Selects all elements of this dict which satisfy a predicate.

Parameters **p** – The predicate to satisfy.

Returns The filtered dict.

`pyfuncol.dict.filter_not`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*, *p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Dict[pyfuncol.dict.A, pyfuncol.dict.B]

Selects all elements of this dict which do not satisfy a predicate.

Parameters **p** – The predicate to not satisfy.

Returns The filtered dict.

`pyfuncol.dict.flat_map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*, *f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Dict[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Dict[pyfuncol.dict.C, pyfuncol.dict.D]

Builds a new dict by applying a function to all elements of this dict and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new dict.

`pyfuncol.dict.foreach`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], pyfuncol.dict.U]*) → None

Apply f to each element for its side effects.

Parameters **f** – The function to apply to all elements for its side effects.

`pyfuncol.dict.is_empty`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*) → bool

Tests whether the dict is empty.

Returns True if the dict is empty, False otherwise.

`pyfuncol.dict.map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Tuple[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Dict[pyfuncol.dict.C, pyfuncol.dict.D]

Builds a new dict by applying a function to all elements of this dict.

Parameters **f** – The function to apply to all elements.

Returns The new dict.

`pyfuncol.dict.to_list`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*) → List[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]]

Converts this dict to a list of (key, value) pairs.

Returns A list of pairs corresponding to the entries of the dict

`pyfuncol.dict.to_iterator`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B]*) → Iterator[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]]

Converts this dict to an iterator of (key, value) pairs.

Returns An iterator of pairs corresponding to the entries of the dict

`pyfuncol.dict.count`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → int

Counts the number of elements in the collection which satisfy a predicate.

Note: will not terminate for infinite-sized collections.

Parameters **p** – The predicate used to test elements.

Returns The number of elements satisfying the predicate p.

`pyfuncol.dict.fold_left`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], z: pyfuncol.dict.B, op: Callable[[pyfuncol.dict.B, Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], pyfuncol.dict.B]*) → pyfuncol.dict.B

Applies a binary operator to a start value and all elements of this collection, going left to right.

Note: will not terminate for infinite-sized collections.

Note: might return different results for different runs, unless the underlying collection type is ordered or the operator is associative and commutative.

Parameters

- **z** – The start value.
- **op** – The binary operator.

Returns op(…op(z, x_1), x_2, …, x_n) where x1, …, xn are the elements of this collection. Returns z if this collection is empty.

Return type The result of inserting op between consecutive elements of this collection, going left to right with the start value z on the left

`pyfuncol.dict.fold_right`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], z: pyfuncol.dict.B, op: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B], pyfuncol.dict.B], pyfuncol.dict.B]*) → pyfuncol.dict.B

Applies a binary operator to a start value and all elements of this collection, going right to left.

Note: will not terminate for infinite-sized collections.

Note: might return different results for different runs, unless the underlying collection type is ordered or the operator is associative and commutative.

Parameters

- **z** – The start value.
- **op** – The binary operator.

Returns op(x_1, op(x_2, … op(x_n, z)…)) where x1, …, xn are the elements of this collection. Returns z if this collection is empty.

Return type The result of inserting op between consecutive elements of this collection, going right to left with the start value z on the right

`pyfuncol.dict.forall`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → bool

Tests whether a predicate holds for all elements of this collection.

Note: may not terminate for infinite-sized collections.

Parameters **p** – The predicate used to test elements.

Returns True if this collection is empty or the given predicate p holds for all elements of this collection, otherwise False.

`pyfuncol.dict.find`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Optional[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]]

Finds the first element of the collection satisfying a predicate, if any.

Note: may not terminate for infinite-sized collections.

Note: might return different results for different runs, unless the underlying collection type is ordered.

Parameters **p** – The predicate used to test elements.

Returns An option value containing the first element in the collection that satisfies p, or None if none exists.

`pyfuncol.dict.par_filter`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Dict[pyfuncol.dict.A, pyfuncol.dict.B]

Selects in parallel all elements of this dict which satisfy a predicate.

Parameters **p** – The predicate to satisfy.

Returns The filtered dict.

`pyfuncol.dict.par_filter_not`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Dict[pyfuncol.dict.A, pyfuncol.dict.B]

Selects in parallel all elements of this dict which do not satisfy a predicate.

Parameters **p** – The predicate to not satisfy.

Returns The filtered dict.

`pyfuncol.dict.par_flat_map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Dict[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Dict[pyfuncol.dict.C, pyfuncol.dict.D]

Builds a new dict by applying a function in parallel to all elements of this dict and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new dict.

`pyfuncol.dict.par_map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Tuple[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Dict[pyfuncol.dict.C, pyfuncol.dict.D]

Builds a new dict by applying a function in parallel to all elements of this dict.

Parameters **f** – The function to apply to all elements.

Returns The new dict.

`pyfuncol.dict.pure_map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Tuple[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Dict[pyfuncol.dict.C, pyfuncol.dict.D]

Builds a new dict by applying a function to all elements of this dict using memoization to improve performance.

WARNING: f must be a PURE function i.e., calling f on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **f** – The function to apply to all elements.

Returns The new dict.

`pyfuncol.dict.pure_flat_map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Dict[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Dict[pyfuncol.dict.C, pyfuncol.dict.D]

Builds a new dict by applying a function to all elements of this dict and using the elements of the resulting collections using memoization to improve performance.

WARNING: f must be a PURE function i.e., calling f on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **f** – The function to apply to all elements.

Returns The new dict.

`pyfuncol.dict.pure_filter`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Dict[pyfuncol.dict.A, pyfuncol.dict.B]

Selects all elements of this dict which satisfy a predicate using memoization to improve performance.

WARNING: p must be a PURE function i.e., calling p on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **p** – The predicate to satisfy.

Returns The filtered dict.

`pyfuncol.dict.pure_filter_not`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Dict[pyfuncol.dict.A, pyfuncol.dict.B]

Selects all elements of this dict which do not satisfy a predicate using memoization to improve performance.

WARNING: p must be a PURE function i.e., calling p on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **p** – The predicate not to satisfy.

Returns The filtered dict.

`pyfuncol.dict.lazy_map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Tuple[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Iterator[Tuple[pyfuncol.dict.C, pyfuncol.dict.D]]

Builds a new list of tuples by applying a function to all elements of this dict, lazily.

Parameters **f** – The function to apply to all elements.

Returns The new lazy list of tuples, as an iterator.

`pyfuncol.dict.lazy_flat_map`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], f: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], Dict[pyfuncol.dict.C, pyfuncol.dict.D]]*) → Iterator[Tuple[pyfuncol.dict.C, pyfuncol.dict.D]]

Builds a new list of tuples by applying a function to all elements of this dict and using the elements of the resulting collections, lazily.

Parameters **f** – The function to apply to all elements.

Returns The new lazy list of tuples, as an iterator.

`pyfuncol.dict.lazy_filter`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Iterator[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]]

Selects all elements of this dict which satisfy a predicate, lazily.

Parameters **p** – The predicate to satisfy.

Returns The filtered lazy list of tuples, as an iterator.

`pyfuncol.dict.lazy_filter_not`(*self: Dict[pyfuncol.dict.A, pyfuncol.dict.B], p: Callable[[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]], bool]*) → Iterator[Tuple[pyfuncol.dict.A, pyfuncol.dict.B]]

Selects all elements of this dict which do not satisfy a predicate, lazily.

Parameters **p** – The predicate to not satisfy.

Returns The filtered lazy list of tuples, as an iterator.

5.1.3 pyfuncol.list module

`pyfuncol.list.map`(*self: List[pyfuncol.list.A], f: Callable[[pyfuncol.list.A], pyfuncol.list.B]*) → List[pyfuncol.list.B]

Builds a new list by applying a function to all elements of this list.

Parameters **f** – The function to apply to all elements.

Returns The new list.

`pyfuncol.list.filter`(*self: List[pyfuncol.list.A], p: Callable[[pyfuncol.list.A], bool]*) → List[pyfuncol.list.A]

Selects all elements of this list which satisfy a predicate.

Parameters **p** – The predicate to satisfy.

Returns The filtered list.

`pyfuncol.list.filter_not`(*self: List[pyfuncol.list.A], p: Callable[[pyfuncol.list.A], bool]*) → List[pyfuncol.list.A]

Selects all elements of this list which do not satisfy a predicate.

Parameters **p** – The predicate to not satisfy.

Returns The filtered list.

`pyfuncol.list.flat_map`(*self: List[pyfuncol.list.A], f: Callable[[pyfuncol.list.A], List[pyfuncol.list.B]]*) → List[pyfuncol.list.B]

Builds a new list by applying a function to all elements of this list and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new list.

`pyfuncol.list.flatten`(*self: List[pyfuncol.list.A]*) → List[pyfuncol.list.B]

Converts this list of lists into a list formed by the elements of these lists.

Returns The flattened list.

`pyfuncol.list.contains`(*self: List[pyfuncol.list.A], elem: pyfuncol.list.A*) → bool

Tests whether this list contains a given value as element.

Parameters **elem** – The element to look for.

Returns True if the list contains the element, False otherwise.

`pyfuncol.list.distinct`(*self: List[pyfuncol.list.A]*) → List[pyfuncol.list.A]

Selects all the elements of this list ignoring the duplicates.

Returns The list without duplicates.

`pyfuncol.list.foreach`(*self: List[pyfuncol.list.A], f: Callable[[pyfuncol.list.A], pyfuncol.list.U]*) → None

Apply f to each element for its side effects.

Parameters **f** – The function to apply to all elements for its side effects.

`pyfuncol.list.group_by`(*self: List[pyfuncol.list.A], f: Callable[[pyfuncol.list.A], pyfuncol.list.K]*) → Dict[pyfuncol.list.K, List[pyfuncol.list.A]]

Partitions this list into a dict of lists according to some discriminator function.

Parameters **f** – The grouping function.

Returns A dictionary where elements are grouped according to the grouping function.

`pyfuncol.list.is_empty`(*self: List[pyfuncol.list.A]*) → bool

Tests whether the list is empty.

Returns True if the list is empty, False otherwise.

`pyfuncol.list.size`(*self: List[pyfuncol.list.A]*) → int

Computes the size of this list.

Returns The size of the list.

`pyfuncol.list.find`(*self: List[pyfuncol.list.A], p: Callable[[pyfuncol.list.A], bool]*) → Optional[pyfuncol.list.A]

Finds the first element of the list satisfying a predicate, if any.

Parameters **p** – The predicate to satisfy.

Returns The first element satisfying the predicate, otherwise None.

`pyfuncol.list.index_of`(*self: List[pyfuncol.list.A]*, *elem: pyfuncol.list.A*) → int

Finds index of first occurrence of some value in this list. Returns -1 if none exists.

Parameters **elem** – The element whose index is to find.

Returns The index of the first occurrence of the element, or -1 if it does not exists.

`pyfuncol.list.fold_left`(*self: List[pyfuncol.list.A]*, *z: pyfuncol.list.B*, *op: Callable[[pyfuncol.list.B, pyfuncol.list.A], pyfuncol.list.B]*) → pyfuncol.list.B

Applies a binary operator to a start value and all elements of this sequence, going left to right.

Parameters

- **z** – The start value.
- **op** – The binary operation.

Returns op(…op(z, x_1), x_2, …, x_n) where x1, …, xn are the elements of this sequence. Returns z if this sequence is empty.

Return type The result of inserting op between consecutive elements of this sequence, going left to right with the start value z on the left

`pyfuncol.list.fold_right`(*self: List[pyfuncol.list.A]*, *z: pyfuncol.list.B*, *op: Callable[[pyfuncol.list.A, pyfuncol.list.B], pyfuncol.list.B]*) → pyfuncol.list.B

Applies a binary operator to all elements of this list and a start value, going right to left.

Parameters

- **z** – The start value.
- **op** – The binary operation.

Returns op(x_1, op(x_2, … op(x_n, z)…)) where x1, …, xn are the elements of this list. Returns z if this list is empty.

Return type The result of inserting op between consecutive elements of this list, going right to left with the start value z on the right

`pyfuncol.list.forall`(*self: List[pyfuncol.list.A]*, *p: Callable[[pyfuncol.list.A], bool]*) → bool

Tests whether a predicate holds for all elements of this list.

Parameters **p** – The predicate used to test elements.

Returns True if this list is empty or the given predicate p holds for all elements of this list, otherwise False.

`pyfuncol.list.head`(*self: List[pyfuncol.list.A]*) → pyfuncol.list.A

Selects the first element of this iterable collection.

Note: might return different results for different runs, unless the underlying collection type is ordered.

Raises **IndexError** – If the iterable collection is empty.

`pyfuncol.list.tail`(*self: List[pyfuncol.list.A]*) → List[pyfuncol.list.A]

The rest of the collection without its first element.

Raises **IndexError** – If the iterable collection is empty.

`pyfuncol.list.take`(*self: List[pyfuncol.list.A]*, *n: int*) → List[pyfuncol.list.A]

Selects the first n elements.

Parameters **n** – The number of elements to take from this list.

Returns A list consisting only of the first n elements of this list, or else the whole list, if it has less than n elements. If n is negative, returns an empty list.

`pyfuncol.list.length`(*self: List[pyfuncol.list.A]*) → int

Returns the length (number of elements) of the list. *size* is an alias for length.

Returns The length of the list

`pyfuncol.list.to_iterator`(*self: List[pyfuncol.list.A]*) → Iterator[pyfuncol.list.A]

Converts this list to an iterator.

Returns An iterator

`pyfuncol.list.par_map`(*self: List[pyfuncol.list.A]*, *f: Callable[[pyfuncol.list.A], pyfuncol.list.B]*) → List[pyfuncol.list.B]

Builds a new list by applying a function in parallel to all elements of this list.

Parameters **f** – The function to apply to all elements.

Returns The new list.

`pyfuncol.list.par_filter`(*self: List[pyfuncol.list.A]*, *p: Callable[[pyfuncol.list.A], bool]*) → List[pyfuncol.list.A]

Selects in parallel all elements of this list which satisfy a predicate.

Parameters **p** – The predicate to satisfy.

Returns The filtered list.

`pyfuncol.list.par_filter_not`(*self: List[pyfuncol.list.A]*, *p: Callable[[pyfuncol.list.A], bool]*) → List[pyfuncol.list.A]

Selects in parallel all elements of this list which do not satisfy a predicate.

Parameters **p** – The predicate to not satisfy.

Returns The filtered list.

`pyfuncol.list.par_flat_map`(*self: List[pyfuncol.list.A]*, *f: Callable[[pyfuncol.list.A], List[pyfuncol.list.B]]*) → List[pyfuncol.list.B]

Builds a new list by applying a function in parallel to all elements of this list and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new list.

`pyfuncol.list.pure_map`(*self: List[pyfuncol.list.A]*, *f: Callable[[pyfuncol.list.A], pyfuncol.list.B]*) → List[pyfuncol.list.B]

Builds a new list by applying a function to all elements of this list using memoization to improve performance.

WARNING: f must be a PURE function i.e., calling f on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **f** – The PURE function to apply to all elements.

Returns The new list.

`pyfuncol.list.pure_flat_map(self: List[pyfuncol.list.A], f: Callable[[pyfuncol.list.A], List[pyfuncol.list.B]]) → List[pyfuncol.list.B]`

Builds a new list by applying a function to all elements of this list and using the elements of the resulting collections using memoization to improve performance.

WARNING: f must be a PURE function i.e., calling f on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **f** – The function to apply to all elements.

Returns The new list.

`pyfuncol.list.pure_filter(self: List[pyfuncol.list.A], p: Callable[[pyfuncol.list.A], bool]) → List[pyfuncol.list.A]`

Selects all elements of this list which satisfy a predicate using memoization to improve performance.

WARNING: p must be a PURE function i.e., calling p on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **p** – The predicate to satisfy.

Returns The filtered list.

`pyfuncol.list.pure_filter_not(self: List[pyfuncol.list.A], p: Callable[[pyfuncol.list.A], bool]) → List[pyfuncol.list.A]`

Selects all elements of this list which do not satisfy a predicate using memoization to improve performance.

WARNING: p must be a PURE function i.e., calling p on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **p** – The predicate not to satisfy.

Returns The filtered list.

`pyfuncol.list.lazy_map(self: List[pyfuncol.list.A], f: Callable[[pyfuncol.list.A], pyfuncol.list.B]) → Iterator[pyfuncol.list.B]`

Builds a new list by applying a function to all elements of this list, lazily.

Parameters **f** – The function to apply to all elements.

Returns The new lazy list, as an iterator.

`pyfuncol.list.lazy_filter(self: List[pyfuncol.list.A], p: Callable[[pyfuncol.list.A], bool]) → Iterator[pyfuncol.list.A]`

Selects all elements of this list which satisfy a predicate, lazily.

Parameters **p** – The predicate to satisfy.

Returns The filtered lazy list, as an iterator.

`pyfuncol.list.lazy_filter_not(self: List[pyfuncol.list.A], p: Callable[[pyfuncol.list.A], bool]) → Iterator[pyfuncol.list.A]`

Selects all elements of this list which do not satisfy a predicate, lazily.

Parameters **p** – The predicate to not satisfy.

Returns The filtered lazy list, as an iterator.

`pyfuncol.list.lazy_flat_map(self: List[pyfuncol.list.A], f: Callable[[pyfuncol.list.A], List[pyfuncol.list.B]]) → Iterator[pyfuncol.list.B]`

Builds a new lazy list by applying a function to all elements of this list and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new lazy list, as an iterator.

`pyfuncol.list.lazy_flatten(self: List[pyfuncol.list.A]) → Iterator[pyfuncol.list.B]`

Converts this list of lists into a lazy list formed by the elements of these lists.

Returns The flattened lazy list, as an iterator.

`pyfuncol.list.lazy_distinct(self: List[pyfuncol.list.A]) → Iterator[pyfuncol.list.A]`

Selects all the elements of this list ignoring the duplicates, lazily.

Returns The lazy list without duplicates, as an iterator.

`pyfuncol.list.lazy_take(self: List[pyfuncol.list.A], n: int) → Iterator[pyfuncol.list.A]`

Selects the first n elements, lazily.

Parameters **n** – The number of elements to take from this list.

Returns A lazy list (as an iterator) consisting only of the first n elements of this list, or else the whole lazy list, if it has less than n elements. If n is negative, returns an empty lazy list.

5.1.4 pyfuncol.set module

`pyfuncol.set.map(self: Set[pyfuncol.set.A], f: Callable[[pyfuncol.set.A], pyfuncol.set.B]) → Set[pyfuncol.set.B]`

Builds a new set by applying a function to all elements of this set.

Parameters **f** – The function to apply to all elements.

Returns The new set.

`pyfuncol.set.filter(self: Set[pyfuncol.set.A], p: Callable[[pyfuncol.set.A], bool]) → Set[pyfuncol.set.A]`

Selects all elements of this set which satisfy a predicate.

Parameters **p** – The predicate to satisfy.

Returns The filtered set.

`pyfuncol.set.filter_not(self: Set[pyfuncol.set.A], p: Callable[[pyfuncol.set.A], bool]) → Set[pyfuncol.set.A]`

Selects all elements of this set which do not satisfy a predicate.

Parameters **p** – The predicate to not satisfy.

Returns The filtered set.

`pyfuncol.set.flat_map(self: Set[pyfuncol.set.A], f: Callable[[pyfuncol.set.A], Set[pyfuncol.set.B]]) → Set[pyfuncol.set.B]`

Builds a new set by applying a function to all elements of this set and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new set.

`pyfuncol.set.contains`(*self: Set[pyfuncol.set.A]*, *elem: pyfuncol.set.A*) → bool

Tests whether this set contains a given value as element.

Parameters **elem** – The element to look for.

Returns True if the set contains the element, False otherwise.

`pyfuncol.set.foreach`(*self: Set[pyfuncol.set.A]*, *f: Callable[[pyfuncol.set.A], pyfuncol.set.U]*) → None

Apply f to each element of the set for its side effects.

Parameters **f** – The function to apply to all elements for its side effects.

`pyfuncol.set.group_by`(*self: Set[pyfuncol.set.A]*, *f: Callable[[pyfuncol.set.A], pyfuncol.set.K]*) → Dict[pyfuncol.set.K, Set[pyfuncol.set.A]]

Partitions this set into a dict of sets according to some discriminator function.

Parameters **f** – The grouping function.

Returns A dictionary where elements are grouped according to the grouping function.

`pyfuncol.set.is_empty`(*self: Set[pyfuncol.set.A]*) → bool

Tests whether the set is empty.

Returns True if the set is empty, False otherwise.

`pyfuncol.set.size`(*self: Set[pyfuncol.set.A]*) → int

Computes the size of this set.

Returns The size of the set.

`pyfuncol.set.find`(*self: Set[pyfuncol.set.A]*, *p: Callable[[pyfuncol.set.A], bool]*) → Optional[pyfuncol.set.A]

Finds the first element of the set satisfying a predicate, if any.

Parameters **p** – The predicate to satisfy.

Returns The first element satisfying the predicate, otherwise None.

`pyfuncol.set.fold_left`(*self: Set[pyfuncol.set.A]*, *z: pyfuncol.set.B*, *op: Callable[[pyfuncol.set.B, pyfuncol.set.A], pyfuncol.set.B]*) → pyfuncol.set.B

Applies a binary operator to a start value and all elements of this set, going left to right.

Note: might return different results for different runs, unless the underlying collection type is ordered or the operator is associative and commutative.

Parameters

- **z** – The start value.
- **op** – The binary operation.

Returns op(…op(z, x_1), x_2, …, x_n) where x1, …, xn are the elements of this set. Returns z if this set is empty.

Return type The result of inserting op between consecutive elements of this set, going left to right with the start value z on the left

`pyfuncol.set.fold_right`(*self: Set[pyfuncol.set.A]*, *z: pyfuncol.set.B*, *op: Callable[[pyfuncol.set.A, pyfuncol.set.B], pyfuncol.set.B]*) → pyfuncol.set.B

Applies a binary operator to all elements of this set and a start value, going right to left.

Note: might return different results for different runs, unless the underlying collection type is ordered or the operator is associative and commutative.

Parameters

- **z** – The start value.
- **op** – The binary operation.

Returns op(x_1, op(x_2, … op(x_n, z)…)) where x1, …, xn are the elements of this set. Returns z if this set is empty.

Return type The result of inserting op between consecutive elements of this set, going right to left with the start value z on the right

`pyfuncol.set.forall`(*self: Set[pyfuncol.set.A]*, *p: Callable[[pyfuncol.set.A], bool]*) → bool

Tests whether a predicate holds for all elements of this set.

Parameters **p** – The predicate used to test elements.

Returns True if this set is empty or the given predicate p holds for all elements of this set, otherwise False.

`pyfuncol.set.length`(*self: Set[pyfuncol.set.A]*) → int

Returns the length (number of elements) of the set. *size* is an alias for length.

Returns The length of the set

`pyfuncol.set.to_iterator`(*self: Set[pyfuncol.set.A]*) → Iterator[pyfuncol.set.A]

Converts this set to an iterator.

Returns An iterator

`pyfuncol.set.par_map`(*self: Set[pyfuncol.set.A]*, *f: Callable[[pyfuncol.set.A], pyfuncol.set.B]*) → Set[pyfuncol.set.B]

Builds a new set by applying in parallel a function to all elements of this set.

Parameters **f** – The function to apply to all elements.

Returns The new set.

`pyfuncol.set.par_filter`(*self: Set[pyfuncol.set.A]*, *p: Callable[[pyfuncol.set.A], bool]*) → Set[pyfuncol.set.A]

Selects in parallel all elements of this set which satisfy a predicate.

Parameters **p** – The predicate to satisfy.

Returns The filtered set.

`pyfuncol.set.par_filter_not`(*self: Set[pyfuncol.set.A]*, *p: Callable[[pyfuncol.set.A], bool]*) → Set[pyfuncol.set.A]

Selects in parallel all elements of this set which do not satisfy a predicate.

Parameters **p** – The predicate to not satisfy.

Returns The filtered set.

`pyfuncol.set.par_flat_map`(*self: Set[pyfuncol.set.A]*, *f: Callable[[pyfuncol.set.A], Set[pyfuncol.set.B]]*) → Set[pyfuncol.set.B]

Builds a new set by applying in parallel a function to all elements of this set and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new set.

`pyfuncol.set.pure_map`(*self: Set[pyfuncol.set.A], f: Callable[[pyfuncol.set.A], pyfuncol.set.B]*) → Set[pyfuncol.set.B]

Builds a new set by applying a function to all elements of this set using memoization to improve performance.

WARNING: f must be a PURE function i.e., calling f on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **f** – The PURE function to apply to all elements.

Returns The new set.

`pyfuncol.set.pure_flat_map`(*self: Set[pyfuncol.set.A], f: Callable[[pyfuncol.set.A], Set[pyfuncol.set.B]]*) → Set[pyfuncol.set.B]

Builds a new set by applying a function to all elements of this set and using the elements of the resulting collections using memoization to improve performance.

WARNING: f must be a PURE function i.e., calling f on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **f** – The function to apply to all elements.

Returns The new set.

`pyfuncol.set.pure_filter`(*self: Set[pyfuncol.set.A], p: Callable[[pyfuncol.set.A], bool]*) → Set[pyfuncol.set.A]

Selects all elements of this set which satisfy a predicate using memoization to improve performance.

WARNING: p must be a PURE function i.e., calling p on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **p** – The predicate to satisfy.

Returns The filtered set.

`pyfuncol.set.pure_filter_not`(*self: Set[pyfuncol.set.A], p: Callable[[pyfuncol.set.A], bool]*) → Set[pyfuncol.set.A]

Selects all elements of this set which do not satisfy a predicate using memoization to improve performance.

WARNING: p must be a PURE function i.e., calling p on the same input must always lead to the same result!

Type A must be hashable using *hash()* function.

Parameters **p** – The predicate not to satisfy.

Returns The filtered set.

`pyfuncol.set.lazy_map`(*self: Set[pyfuncol.set.A], f: Callable[[pyfuncol.set.A], pyfuncol.set.B]*) → Iterator[pyfuncol.set.B]

Builds a new set by applying a function to all elements of this set, lazily.

Parameters **f** – The function to apply to all elements.

Returns The new lazy set, as an iterator.

`pyfuncol.set.lazy_filter`(*self: Set[pyfuncol.set.A], p: Callable[[pyfuncol.set.A], bool]*) → Iterator[pyfuncol.set.A]

Selects all elements of this set which satisfy a predicate, lazily.

Parameters **p** – The predicate to satisfy.

Returns The filtered lazy set, as an iterator.

`pyfuncol.set.lazy_filter_not`(*self: Set[pyfuncol.set.A]*, *p: Callable[[pyfuncol.set.A], bool]*) → Iterator[pyfuncol.set.A]

Selects all elements of this set which do not satisfy a predicate, lazily.

Parameters **p** – The predicate to not satisfy.

Returns The filtered lazy set, as an iterator.

`pyfuncol.set.lazy_flat_map`(*self: Set[pyfuncol.set.A]*, *f: Callable[[pyfuncol.set.A], Set[pyfuncol.set.B]]*) → Iterator[pyfuncol.set.B]

Builds a new lazy set by applying a function to all elements of this set and using the elements of the resulting collections.

Parameters **f** – The function to apply to all elements.

Returns The new lazy set, as an iterator.

5.1.5 Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

S

T