FLPy

Release 1.0.0

Jérome Eertmans

Nov 01, 2021

PYTHON MODULES:

CHAPTER

ONE

FUNCTIONAL, BUT LAZY PYTHON

With **FLPy**, improve the readibility of your programs by leveraging functional programming.

1.1 Installation

FLPy has no external dependencies, and can be installed using `pip`:

```
pip install flpy
```

1.2 Examples

Given an input sequence `x`, print all, but the first, squared values that are divisible by 3 and collect the result into a list.

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  # Input sequence

# Usual way

>>> squares = map(lambda v: v * v, x)
>>> squares_div_by_3 = filter(lambda v: v % 3 == 0, squares)
>>> y = list(squares_div_by_3)[1:]  # We skip one value
>>> for v in y:
...     print(v)
36
81
>>> y
[36, 81]

# FLPy way

>>> from flpy import It

>>> y = (
...     It(x)
...     .map('|v| v * v')  # You can also use lambda or any other callable
...     .filter('|v| v % 3 == 0')
...     .skip(1)
...     .collect()  # Collects the iterator into a list
...     .for_each('|v| print(v)')  # Yet it still returns the list to `y`
```

(continues on next page)

(continued from previous page)

```
... )
36
81
>>> y
ItA<[36, 81]>
```

1.3 TODOs

Non-exhaustive list of ideas.

1. Implement parallel iterators
2. Add kwargs support for built-in functions
3. Increase # of examples
4. Improve docs
5. Write contribution guidelines
6. Setup mypy
7. Add unitary tests
8. Add benchmarks vs. pure Python to measure overhead
9. Add support for async iterators
10. ...

1.3.1 Documentation

flpy

flpy package

Submodules

flpy.iterators module

`It`(*x:* Iterable) → Union[*flpy.iterators.Iterable*, *flpy.iterators.Iterator*]

Create an instance of Iterable or Iterator object depending on if the argument implements `next` or not.

Parameters **x** – an iterable

Returns an Iterable or a subclass

Example

```
>>> from flpy import It
>>> type(It([1, 2, 3]))
<class 'flpy.iterators.Iterable'>
>>> type(It(i for i in [1, 2, 3]))
<class 'flpy.iterators.Iterator'>
```

class Iterable(*x=[]*)

Bases: object

An Iterable can wrap Python object that implements *iter*, and re-implements many builtin features with a functionnal programming approach.

accumulate(*f*) → *flpy.iterators.Iterator*

Apply built-in functools.accumulate on current object.

Parameters **f** – a *parse_func* compatible function

Returns an Iterator

chain(**its*) → *flpy.iterators.Iterator*

Chain current object with any number of objects that implements *iter* using itertools.chain.

Parameters **its** – see itertools.chain arguments

Returns an Iterator

Example

```
>>> from flpy import It
>>> It([1, 2, 3]).chain([4, 5, 6]).collect()
ItA<[1, 2, 3, 4, 5, 6]>
```

collect(*collector: Callable = <class 'list'>*) → *flpy.iterators.Iterable*

Collect the current Iterable into a new Iterable using *collector* function. By default, it will transform the content into a list.

Parameters **f** – a *parse_func* compatible function

Returns an Iterable

classmethod empty()

every(*n: int*)

Return current iterator, but one 1 item every *n* is kept.

Parameters **n** – the spacing between two items

Returns an Iterator

filter(*f*) → *flpy.iterators.Iterator*

Apply built-in filter on current object.

Parameters **f** – a *parse_func* compatible function

Returns an Iterator

Examples

```
>>> from flpy import It
>>> It([1, 2, 3]).filter('|x| x > 1').collect()
ItA<[2, 3]>
```

filter_map(*f: Optional[Union[Callable, str]]*) → *flpy.iterators.Iterator*

Chain *Iterable.map* and *Iterable.filter* to only return non-None results.

Parameters **f** – a *parse_func* compatible function

Returns an Iterator

for_each(*f: Optional[Union[Callable, str]]*) → Iteratable

Apply a function on each element and return self.

Parameters **f** – a *parse_func* compatible function

Returns self

Example

```
>>> from flpy import It
>>> it = It([1, 2, 3]).for_each('|v| print(v)')
1
2
3
>>> it
ItA<[1, 2, 3]>
>>> it = It(i for i in [1, 2, 3]).for_each('|v| print(v)').collect()
1
2
3
>>> it  # Here generator was consumed to <it> is now empty
ItA<[]>
```

iter() → *flpy.iterators.Iterator*

Return an iterator version of current object.

Returns an Iterator

Example

```
>>> from flpy import It
>>> it = It([1, 2, 3])
>>> next(it.iter())
1
```

map(*f*) → *flpy.iterators.Iterator*

Apply built-in `map` on current object.

Parameters **f** – a *parse_func* compatible function

Returns an Iterator

Examples

```
>>> from flpy import It
>>> It([1, 2, 3]).map('|x| x * x').collect()
ItA<[1, 4, 9]>
```

max() → Any

Apply built-in `max` on current object.

Returns the maximum

Examples

```
>>> from flpy import It
>>> It([1, 2, 3]).max()
3
```

min() → Any

Apply built-in `min` on current object.

Returns the minimum

Examples

```
>>> from flpy import It
>>> It([1, 2, 3]).min()
1
```

min_max() → Tuple[Any, Any]
Apply both *min* and **ax** on current object.

Returns the minimum and the maximum

Examples

```
>>> from flpy import It
>>> It([1, 2, 3]).min_max()
(1, 3)
```

reduce(*f*, **args*, ***kwargs*) → Any
Apply built-in `functools.reduce` on current object.

Parameters **f** – a *parse_func* compatible function

Returns the reduction result

repeat(*times: Optional[int] = None*) → *flpy.iterators.Iterator*
Return current iterator, repeated *n* times. If *n* is None, the repetition is infinite.

Parameters **n** – the number of repetitions

Returns an Iterator

set_value(*x:* Iterable) → *flpy.iterators.Iterable*
Change the content of current Iterable to be *x*.

Param an Iterable

Returns self

Example

```
>>> from flpy import It
>>> x = It([1, 2, 3])
>>> x.set_value([4, 5, 6])
ItA<[4, 5, 6]>
```

skip(*n: int*) → *flpy.iterators.Iterator*
Return current iterator, but with first *n* items are skipped.

Parameters **n** – the number of items to skip

Returns an Iterator

slice(**args: Optional[int]*) → *flpy.iterators.Iterator*
Return a slice of current iterable using `functools.islice`.

Parameters **args** – see `functools.islice` arguments

Returns an Iterator

take(*n: int*) → *flpy.iterators.Iterator*
Return current iterator, but with max *n* items are kept.

Parameters **n** – the number of items to keep

Returns an Iterator

to(*iterable:* flpy.iterators.Iterable, *safe: bool = True*) → *flpy.iterators.Iterable*
Move current iterable value into argument object. By default (if *safe*), the content of self will be set to an empty value, to avoid two Iterable objects sharing the same content.

Parameters

- **iterable** – target iterable
- **safe** – if safe, set content of self to an empty Iterable

Returns self

unwrap() → *Iterable*
Return the iterable object hold by current Iterable instance.

Returns an iterable

x

zip(**args*) → *flpy.iterators.Iterator*
Apply built-in `zip` on current object.

Parameters **args** – see `zip` arguments

Returns an Iterator

zip_longest(**args*)
Apply built-in `itertools.zip_longest` on current object.

Parameters **args** – see `itertools.zip_longest` arguments

Returns an Iterator

class Iterator(*x: Iterator = <generator object empty_iterator>*)
Bases: *flpy.iterators.Iterable*

A subclass of Iterable where the wrapped argument also implements `next` method, thus providing additional possibilities.

classmethod empty()

x

empty_iterable() → *Iterable*
Return an empty iterable, i.e., an empty list.

Returns an iterable

Example

```
>>> from flpy.iterators import empty_iterable
>>> empty_iterable()
[]
```

empty_iterator() → *Iterator*
Return an empty iterator.

Returns an iterator

Example

```
>>> from flpy.iterators import empty_iterator, It
>>> It(empty_iterator()).collect()
ItA<[]>
```

parse_func(*func: Optional[Union[Callable, str]]*) → Optional[Callable]

Parse a function into a callable. Input argument can either be a callable or a string with Rust-like syntax.

Rust-like syntax assumes arguments are enclosed between | and are separated by commas: |arg1, arg2, . . . , argn|.

Example

```
>>> from flpy.iterators import parse_func
>>> f = parse_func(lambda x, y: x * y)
>>> f(4, 5)
20
>>> f = parse_func('|x, y| x * y')
>>> f(4, 5)
20
```

takes_function(*func: Callable*) → Callable

Wrapper around class method that takes a function or string as first argument that parses it using *parse_func*.

Parameters **func** – a function

Returns a function

Module contents

CHAPTER

TWO

CONTRIBUTOR GUIDE

Work in progress..

CHAPTER

THREE

INDEX ET TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX