flupy Documentation

Release 1.1.0

Oliver Rice

Oct 24, 2022

Contents

flupy is a lightweight library and CLI for implementing python data pipelines with a fluent interface.

Under the hood, flupy is built on generators. That means its pipelines evaluate lazily and use a constant amount of memory no matter how much data are being processed. This allows flupy to tackle Petabyte scale data manipulation as easily as it operates on a small list.

CHAPTER 1

API

```
import json
from flupy import flu

logs = open('logs.jl', 'r')

error_count = (
    flu(logs)
    .map(lambda x: json.loads(x))
    .filter(lambda x: x['level'] == 'ERROR')
    .count()
)

print(error_count)
# 14
```

CHAPTER 2

CLI

The flupy library, and python runtime, are also accessible from *flu* command line utility:

```
$ cat logs.txt | flu "_.filter(lambda x: x.startswith('ERROR'))"
```

For more information about the *flu* command see *command line*.

CHAPTER 3

Getting Started

Requirements

Python 3.6+

Installation

```
$ pip install flupy
```

CHAPTER 4

Example

Since 2008, what domains are our customers comming from?:

```
from flupy import flu

customers = [
    {'name': 'Jane', 'signup_year': 2018, 'email': 'jane@ibm.com'},
    {'name': 'Fred', 'signup_year': 2011, 'email': 'fred@google.com'},
    {'name': 'Lisa', 'signup_year': 2014, 'email': 'jane@ibm.com'},
    {'name': 'Jack', 'signup_year': 2007, 'email': 'jane@apple.com'},
]

pipeline = (
    flu(customers)
    .filter(lambda x: x['signup_year'] > 2008)
    .map_item('email')
    .map(lambda x: x.partition('@')[2])
    .group_by() # defaults to identity
    .map(lambda x: (x[0], x[1].count()))
    .collect()
)

print(pipeline)
# [('google.com', 1), ('ibm.com', 2)]
```

CHAPTER 5

Contents

5.1 Welcome to Flupy

flupy is a lightweight library and CLI for implementing python data pipelines with a fluent interface.

Under the hood, flupy is built on generators. That means its pipelines evaluate lazily and use a constant amount of memory no matter how much data are being processed. This allows flupy to tackle Petabyte scale data manipulation as easily as it operates on a small list.

5.1.1 API

```
import json
from flupy import flu

logs = open('logs.jl', 'r')

error_count = (
    flu(logs)
    .map(lambda x: json.loads(x))
    .filter(lambda x: x['level'] == 'ERROR')
    .count()
)

print(error_count)
# 14
```

5.1.2 CLI

The flupy library, and python runtime, are also accessible from *flu* command line utility:

```
$ cat logs.txt | flu "_.filter(lambda x: x.startswith('ERROR'))"
```

For more information about the *flu* command see *command line*.

5.1.3 Getting Started

Requirements

Python 3.6+

Installation

```
$ pip install flupy
```

5.1.4 Example

Since 2008, what domains are our customers comming from?:

```
from flupy import flu

customers = [
    {'name': 'Jane', 'signup_year': 2018, 'email': 'jane@ibm.com'},
    {'name': 'Fred', 'signup_year': 2011, 'email': 'fred@google.com'},
    {'name': 'Lisa', 'signup_year': 2014, 'email': 'jane@ibm.com'},
    {'name': 'Jack', 'signup_year': 2007, 'email': 'jane@apple.com'},
]

pipeline = (
    flu(customers)
    .filter(lambda x: x['signup_year'] > 2008)
    .map_item('email')
    .map(lambda x: x.partition('@')[2])
    .group_by() # defaults to identity
    .map(lambda x: (x[0], x[1].count()))
    .collect()
)

print(pipeline)
# [('google.com', 1), ('ibm.com', 2)]
```

5.2 API Reference

5.2.1 Container

class flupy.**flu**(*iterable: Iterable[T]*)

A fluent interface to lazy generator functions

```
>>> from flupy import flu
>>> (
        flu(range(100))
        .map(lambda x: x**2)
        .filter(lambda x: x % 3 == 0)
        .chunk(3)
        .take(2)
```

(continues on next page)

(continued from previous page)

```
        .to_list()
    )
[[0, 9, 36], [81, 144, 225]]
```

5.2.2 Grouping

`flu.`**`chunk`**(*n: int*) → flupy.fluent.Fluent[typing.List[~T]][List[T]]

Yield lists of elements from iterable in groups of *n*

if the iterable is not evenly divisiible by *n*, the final list will be shorter

```
>>> flu(range(10)).chunk(3).to_list()
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

`flu.`**`flatten`**(*depth: int = 1*, *base_type: Type[object] = None*, *iterate_strings: bool = False*) → flupy.fluent.Fluent[typing.Any][Any]

Recursively flatten nested iterables (e.g., a list of lists of tuples) into non-iterable type or an optional user-defined base_type

Strings are treated as non-iterable for convenience. set iterate_string=True to change that behavior.

```
>>> flu([[0, 1, 2], [3, 4, 5]]).flatten().to_list()
[0, 1, 2, 3, 4, 5]
```

```
>>> flu([[0, [1, 2]], [[3, 4], 5]]).flatten().to_list()
[0, [1, 2], [3, 4], 5]
```

```
>>> flu([[0, [1, 2]], [[3, 4], 5]]).flatten(depth=2).to_list()
[0, 1, 2, 3, 4, 5]
```

```
>>> flu([[0, [1, 2]], [[3, 4], 5]]).flatten(depth=2).to_list()
[0, 1, 2, 3, 4, 5]
```

```
>>> flu([1, (2, 2), 4, [5, (6, 6, 6)]]).flatten(base_type=tuple).to_list()
[1, (2, 2), 4, 5, (6, 6, 6)]
```

```
>>> flu([[2, 0], 'abc', 3, [4]]).flatten(iterate_strings=True).to_list()
[2, 0, 'a', 'b', 'c', 3, 4]
```

`flu.`**`denormalize`**(*iterate_strings: bool = False*) → flupy.fluent.Fluent[typing.Tuple[typing.Any, ...]][Tuple[Any, ...]]

Denormalize iterable components of each record

```
>>> flu([("abc", [1, 2, 3])]).denormalize().to_list()
[('abc', 1), ('abc', 2), ('abc', 3)]
```

```
>>> flu([("abc", [1, 2])]).denormalize(iterate_strings=True).to_list()
[('a', 1), ('a', 2), ('b', 1), ('b', 2), ('c', 1), ('c', 2)]
```

```
>>> flu([("abc", [])]).denormalize().to_list()
[]
```

flu.**group_by**(*key: Callable[[T], Union[T, _T1]] = <function identity>, sort: bool = True*) → flupy.fluent.Fluent[typing.Tuple[typing.Union[~T, ~_T1], flupy.fluent.Fluent[~T]]][Tuple[Union[T, _T1], flupy.fluent.Fluent[~T][T]]]

Yield consecutive keys and groups from the iterable

key is a function to compute a key value used in grouping and sorting for each element. *key* defaults to an identity function which returns the unchaged element

When the iterable is pre-sorted according to *key*, setting *sort* to False will prevent loading the dataset into memory and improve performance

```
>>> flu([2, 4, 2, 4]).group_by().to_list()
[2, <flu object>), (4, <flu object>)]
```

Or, if the iterable is pre-sorted

```
>>> flu([2, 2, 5, 5]).group_by(sort=False).to_list()
[(2, <flu object>), (5, <flu object>)]
```

Using a key function

```
>>> points = [
    {'x': 1, 'y': 0},
    {'x': 4, 'y': 3},
    {'x': 1, 'y': 5}
]
>>> key_func = lambda u: u['x']
>>> flu(points).group_by(key=key_func, sort=True).to_list()
[(1, <flu object>), (4, <flu object>)]
```

flu.**window**(*n: int*, *step: int = 1*, *fill_value: Any = None*) → flupy.fluent.Fluent[typing.Tuple[typing.Any, ...]][Tuple[Any, ...]]

Yield a sliding window of width *n* over the given iterable.

Each window will advance in increments of *step*:

If the length of the iterable does not evenly divide by the *step* the final output is padded with *fill_value*

```
>>> flu(range(5)).window(3).to_list()
[(0, 1, 2), (1, 2, 3), (2, 3, 4)]
```

```
>>> flu(range(5)).window(n=3, step=2).to_list()
[(0, 1, 2), (2, 3, 4)]
```

```
>>> flu(range(9)).window(n=4, step=3).to_list()
[(0, 1, 2, 3), (3, 4, 5, 6), (6, 7, 8, None)]
```

```
>>> flu(range(9)).window(n=4, step=3, fill_value=-1).to_list()
[(0, 1, 2, 3), (3, 4, 5, 6), (6, 7, 8, -1)]
```

5.2.3 Selecting

flu.**filter**(*func: Callable[[...], bool], *args, **kwargs*) → flupy.fluent.Fluent[~T][T]

Yield elements of iterable where *func* returns truthy

```
>>> flu(range(10)).filter(lambda x: x % 2 == 0).to_list()
[0, 2, 4, 6, 8]
```

flu.**take**(*n: Optional[int] = None*) → flupy.fluent.Fluent[~T][T]
Yield first *n* items of the iterable

```
>>> flu(range(10)).take(2).to_list()
[0, 1]
```

flu.**take_while**(*predicate: Callable[[T], bool]*) → flupy.fluent.Fluent[~T][T]
Yield elements from the chainable so long as the predicate is true

```
>>> flu(range(10)).take_while(lambda x: x < 3).to_list()
[0, 1, 2]
```

flu.**drop_while**(*predicate: Callable[[T], bool]*) → flupy.fluent.Fluent[~T][T]
Drop elements from the chainable as long as the predicate is true; afterwards, return every element

```
>>> flu(range(10)).drop_while(lambda x: x < 3).to_list()
[3, 4, 5, 6, 7, 8, 9]
```

flu.**unique**(*key: Callable[[T], Hashable] = <function identity>*) → flupy.fluent.Fluent[~T][T]
Yield elements that are unique by a *key*.

```
>>> flu([2, 3, 2, 3]).unique().to_list()
[2, 3]
```

```
>>> flu([2, -3, -2, 3]).unique(key=abs).to_list()
[2, -3]
```

5.2.4 Transforming

flu.**enumerate**(*start: int = 0*) → flupy.fluent.Fluent[typing.Tuple[int, ~T]][Tuple[int, T]]
Yields tuples from the chainable where the first element is a count from initial value *start*.

```
>>> flu(range(5)).zip_longest(range(3, 0, -1)).to_list()
[(0, 3), (1, 2), (2, 1), (3, None), (4, None)]
```

flu.**join_left**(*other: Iterable[_T1], key: Callable[[T], Hashable] = <function identity>, other_key: Callable[[_T1], Hashable] = <function identity>*) → flupy.fluent.Fluent[typing.Tuple[~T, typing.Union[~_T1, NoneType]]][Tuple[T, Optional[_T1]]]
Join the iterable with another iterable using equality between *key* applied to self and *other_key* applied to *other* to identify matching entries

When no matching entry is found in *other*, entries in the iterable are paired with None

Note: join_left loads *other* into memory

```
>>> flu(range(6)).join_left(range(0, 6, 2)).to_list()
[(0, 0), (1, None), (2, 2), (3, None), (4, 4), (5, None)]
```

flu.**join_inner**(*other: Iterable[_T1], key: Callable[[T], Hashable] = <function identity>, other_key: Callable[[_T1], Hashable] = <function identity>*) → flupy.fluent.Fluent[typing.Tuple[~T, ~_T1]][Tuple[T, _T1]]

Join the iterable with another iterable using equality between *key* applied to self and *other_key* applied to *other* to identify matching entries

When no matching entry is found in *other*, entries in the iterable are filtered from the results

Note: join_inner loads *other* into memory

```
>>> flu(range(6)).join_inner(range(0, 6, 2)).to_list()
[(0, 0), (2, 2), (4, 4)]
```

flu.**map**(*func: Callable[[T], _T1], *args, **kwargs*) → flupy.fluent.Fluent[~_T1][_T1]

Apply *func* to each element of iterable

```
>>> flu(range(5)).map(lambda x: x*x).to_list()
[0, 1, 4, 9, 16]
```

flu.**map_attr**(*attr: str*) → flupy.fluent.Fluent[typing.Any][Any]

Extracts the attribute *attr* from each element of the iterable

```
>>> from collections import namedtuple
>>> MyTup = namedtuple('MyTup', ['value', 'backup_val'])
>>> flu([MyTup(1, 5), MyTup(2, 4)]).map_attr('value').to_list()
[1, 2]
```

flu.**map_item**(*item: Hashable*) → flupy.fluent.Fluent[flupy.fluent.SupportsGetItem[~T]][flupy.fluent.SupportsGetItem[~T][T]]

Extracts *item* from every element of the iterable

```
>>> flu([(2, 4), (2, 5)]).map_item(1).to_list()
[4, 5]
```

```
>>> flu([{'mykey': 8}, {'mykey': 5}]).map_item('mykey').to_list()
[8, 5]
```

flu.**zip**(**iterable*) → Union[flupy.fluent.Fluent[typing.Tuple[~T, ...]][Tuple[T, ...]], flupy.fluent.Fluent[typing.Tuple[~T, ~_T1]][Tuple[T, _T1]], flupy.fluent.Fluent[typing.Tuple[~T, ~_T1, ~_T2]][Tuple[T, _T1, _T2]], flupy.fluent.Fluent[typing.Tuple[~T, ~_T1, ~_T2, ~_T3]][Tuple[T, _T1, _T2, _T3]]]

Yields tuples containing the i-th element from the i-th argument in the chainable, and the iterable

```
>>> flu(range(5)).zip(range(3, 0, -1)).to_list()
[(0, 3), (1, 2), (2, 1)]
```

flu.**zip_longest**(**iterable, fill_value: Any = None*) → flupy.fluent.Fluent[typing.Tuple[~T, ...]][Tuple[T, ...]]

Yields tuples containing the i-th element from the i-th argument in the chainable, and the iterable Iteration continues until the longest iterable is exhaused. If iterables are uneven in length, missing values are filled in with fill value

```
>>> flu(range(5)).zip_longest(range(3, 0, -1)).to_list()
[(0, 3), (1, 2), (2, 1), (3, None), (4, None)]
```

```
>>> flu(range(5)).zip_longest(range(3, 0, -1), fill_value='a').to_list()
[(0, 3), (1, 2), (2, 1), (3, 'a'), (4, 'a')]
```

5.2.5 Side Effects

flu.**rate_limit**(*per_second: Union[int, float] = 100*) → flupy.fluent.Fluent[~T][T]
Restrict consumption of iterable to n item *per_second*

```
>>> import time
>>> start_time = time.time()
>>> _ = flu(range(3)).rate_limit(3).to_list()
>>> print('Runtime', int(time.time() - start_time))
1.00126 # approximately 1 second for 3 items
```

flu.**side_effect**(*func: Callable[[T], Any], before: Optional[Callable[[], Any]] = None, after: Optional[Callable[[], Any]] = None*) → flupy.fluent.Fluent[~T][T]
Invoke *func* for each item in the iterable before yielding the item. *func* takes a single argument and the output is discarded *before* and *after* are optional functions that take no parameters and are executed once before iteration begins and after iteration ends respectively. Each will be called exactly once.

```
>>> flu(range(2)).side_effect(lambda x: print(f'Collected {x}')).to_list()
Collected 0
Collected 1
[0, 1]
```

5.2.6 Summarizing

flu.**count**() → int
Count of elements in the iterable

```
>>> flu(['a','b','c']).count()
3
```

flu.**sum**() → Union[T, int]
Sum of elements in the iterable

```
>>> flu([1,2,3]).sum()
6
```

flu.**min**() → SupportsLessThanT
Smallest element in the interable

```
>>> flu([1, 3, 0, 2]).min()
0
```

flu.**max**() → SupportsLessThanT
Largest element in the interable

```
>>> flu([0, 3, 2, 1]).max()
3
```

flu.**reduce**(*func: Callable[[T, T], T]*) → T
Apply a function of two arguments cumulatively to the items of the iterable, from left to right, so as to reduce the sequence to a single value

```
>>> flu(range(5)).reduce(lambda x, y: x + y)
10
```

flu.**fold_left**(*func: Callable[[S, T], S], initial: S*) → S
Apply a function of two arguments cumulatively to the items of the iterable, from left to right, starting with *initial*, so as to fold the sequence to a single value

```
>>> flu(range(5)).fold_left(lambda x, y: x + str(y), "")
'01234'
```

flu.**first**(*default: Any = <flupy.fluent.Empty object>*) → T
Return the first item of the iterable. Raise IndexError if empty or default if provided.

```
>>> flu([0, 1, 2, 3]).first()
0
>>> flu([]).first(default="some_default")
'some_default'
```

flu.**last**(*default: Any = <flupy.fluent.Empty object>*) → T
Return the last item of the iterble. Raise IndexError if empty or default if provided.

```
>>> flu([0, 1, 2, 3]).last()
3
>>> flu([]).last(default='some_default')
'some_default'
```

flu.**head**(*n: int = 10, container_type: Callable[[Iterable[T]], Collection[T]] = <class 'list'>*) → Collection[T]
Returns up to the first *n* elements from the iterable.

```
>>> flu(range(20)).head()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> flu(range(15)).head(n=2)
[0, 1]
```

```
>>> flu([]).head()
[]
```

flu.**tail**(*n: int = 10, container_type: Callable[[Iterable[T]], Collection[T]] = <class 'list'>*) → Collection[T]
Return up to the last *n* elements from the iterable

```
>>> flu(range(20)).tail()
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
>>> flu(range(15)).tail(n=2)
[13, 14]
```

flu.**to_list**() → List[T]
Collect items from iterable into a list

```
>>> flu(range(4)).to_list()
[0, 1, 2, 3]
```

flu.**collect**(*n: int = None, container_type: Callable[[Iterable[T]], Collection[T]] = <class 'list'>*) → Collection[T]
Collect items from iterable into a container

```
>>> flu(range(4)).collect()
[0, 1, 2, 3]
```

```
>>> flu(range(4)).collect(container_type=set)
{0, 1, 2, 3}
```

```
>>> flu(range(4)).collect(n=2)
[0, 1]
```

5.2.7 Non-Constant Memory

flu.**group_by**(*key: Callable[[T], Union[T, _T1]] = <function identity>, sort: bool = True*) → flupy.fluent.Fluent[typing.Tuple[typing.Union[~T, ~_T1], flupy.fluent.Fluent[~T]]][Tuple[Union[T, _T1], flupy.fluent.Fluent[~T][T]]]

Yield consecutive keys and groups from the iterable

key is a function to compute a key value used in grouping and sorting for each element. *key* defaults to an identity function which returns the unchaged element

When the iterable is pre-sorted according to *key*, setting *sort* to False will prevent loading the dataset into memory and improve performance

```
>>> flu([2, 4, 2, 4]).group_by().to_list()
[2, <flu object>), (4, <flu object>)]
```

Or, if the iterable is pre-sorted

```
>>> flu([2, 2, 5, 5]).group_by(sort=False).to_list()
[(2, <flu object>), (5, <flu object>)]
```

Using a key function

```
>>> points = [
    {'x': 1, 'y': 0},
    {'x': 4, 'y': 3},
    {'x': 1, 'y': 5}
]
>>> key_func = lambda u: u['x']
>>> flu(points).group_by(key=key_func, sort=True).to_list()
[(1, <flu object>), (4, <flu object>)]
```

flu.**join_left**(*other: Iterable[_T1], key: Callable[[T], Hashable] = <function identity>, other_key: Callable[[_T1], Hashable] = <function identity>*) → flupy.fluent.Fluent[typing.Tuple[~T, typing.Union[~_T1, NoneType]]][Tuple[T, Optional[_T1]]]

Join the iterable with another iterable using equality between *key* applied to self and *other_key* applied to *other* to identify matching entries

When no matching entry is found in *other*, entries in the iterable are paired with None

Note: join_left loads *other* into memory

```
>>> flu(range(6)).join_left(range(0, 6, 2)).to_list()
[(0, 0), (1, None), (2, 2), (3, None), (4, 4), (5, None)]
```

flu.**join_inner**(*other: Iterable[_T1], key: Callable[[T], Hashable] = <function identity>, other_key: Callable[[_T1], Hashable] = <function identity>*) → flupy.fluent.Fluent[typing.Tuple[~T, ~_T1]][Tuple[T, _T1]]

Join the iterable with another iterable using equality between *key* applied to self and *other_key* applied to *other* to identify matching entries

When no matching entry is found in *other*, entries in the iterable are filtered from the results

Note: join_inner loads *other* into memory

```
>>> flu(range(6)).join_inner(range(0, 6, 2)).to_list()
[(0, 0), (2, 2), (4, 4)]
```

flu.**shuffle**() → flupy.fluent.Fluent[~T][T]

Randomize the order of elements in the interable

Note: shuffle loads the entire iterable into memory

```
>>> flu([3,6,1]).shuffle().to_list()
[6, 1, 3]
```

flu.**sort**(*key: Optional[Callable[[Any], Any]] = None, reverse: bool = False*) → flupy.fluent.Fluent[~SupportsLessThanT][SupportsLessThanT]

Sort iterable by *key* function if provided or identity otherwise

Note: sorting loads the entire iterable into memory

```
>>> flu([3,6,1]).sort().to_list()
[1, 3, 6]
```

```
>>> flu([3,6,1]).sort(reverse=True).to_list()
[6, 3, 1]
```

```
>>> flu([3,-6,1]).sort(key=abs).to_list()
[1, 3, -6]
```

flu.**tee**(*n: int = 2*) → flupy.fluent.Fluent[flupy.fluent.Fluent[~T]][flupy.fluent.Fluent[~T][T]]

Return n independent iterators from a single iterable

once tee() has made a split, the original iterable should not be used anywhere else; otherwise, the iterable could get advanced without the tee objects being informed

```
>>> copy1, copy2 = flu(range(5)).tee()
>>> copy1.sum()
10
>>> copy2.to_list()
[0, 1, 2, 3, 4]
```

flu.**unique**(*key: Callable[[T], Hashable] = <function identity>*) → flupy.fluent.Fluent[~T][T]

Yield elements that are unique by a *key*.

```
>>> flu([2, 3, 2, 3]).unique().to_list()
[2, 3]
```

```
>>> flu([2, -3, -2, 3]).unique(key=abs).to_list()
[2, -3]
```

5.3 Command Line

The flupy CLI is a platform agnostic application that give full access to the flupy API and python from your shell.

5.3.1 Usage

```
$ flu -h

usage: flu [-h] [-v] [-f FILE] [-i [IMPORT [IMPORT ...]]] command

flupy: a fluent interface for python

positional arguments:
  command               command to execute against input

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -f FILE, --file FILE  path to input file
  -i [IMPORT [IMPORT ...]], --import [IMPORT [IMPORT ...]]
                                                modules to import
                                                Syntax: <module>:<object>:<alias>
```

5.3.2 Basic Examples

When input data are provided to the *flu* command, an instance of the flu object is preprepared with that input and stored in the the variable _.

Note: for more information on writing flupy commands, see API Reference

Piping from another command (stdin)

Example: Show lines of a log file that are errors:

```
$ cat logs.txt | flu '_.filter(lambda x: x.starswith("ERROR"))'
```

Reading from a file

Example: Show lines of a log file that are errors:

```
$ flu -f logs.txt '_.filter(lambda x: x.starswith("ERROR"))'
```

No Input data

flupy does not require input data if it can be generated from within python e.g. with *range(10)*. When no input data are provided, iterable at the beginning of the flupy command must be wraped into a flu instance.

Example: Even integers less than 10:

```
$ flu 'flu(range(10)).filter(lambda x: x%2==0)'
```

5.3.3 Import System

Passing *-i* or *–import* to the cli allows you to import standard and third party libraries installed in the same environment.

Import syntax

-i <module>:<object>:<alias>

Note: for multiple imports pass *-i* multiple times

Import Examples

import os:

```
$ flu 'flu(os.environ)' -i os
```

from os import environ:

```
$ flu 'flu(environ)' -i os:environ
```

from os import environ as env:

```
$ flu 'flu(env)' -i os:environ:env
```

import os as opsys:

```
$ flu 'flu(opsys.environ)' -i os::opsys
```

5.4 License

flupy is under the MIT License. See the LICENSE file.

5.4.1 Conditions for Contributors

By contributing to this software project, you are agreeing to the following terms and conditions for your contributions: First, you agree your contributions are submitted under the MIT license. Second, you represent you are authorized to make the contributions and grant the license. If your employer has rights to intellectual property that includes your contributions, you represent that you have received permission to make contributions and grant the required license on behalf of that employer.

5.5 Design Influances

- more-itertools
- pyspark

- pydash
- sqlalchemy
- scala

5.6 Version History

5.6.1 1.0.0

- **New Capabilities:**
  - Everything

5.6.2 1.1.0

- Remove support for calling instance methods on uninitialized flu class passing an interable as the *self* argument
- Remove *flupy.Fluent* from top level *flupy* public API
- Remove *flupy.with_iter* from API

5.6.3 1.1.2

- Change *Fluent* class name to *flu* and remove class alias to improve docs readability
- Add type hints for *flu.sum*

Python Module Index

f

Index