fluent Documentation

Release 2.1

Martin Häcker

Mar 19, 2021

FOR MORE INFORMATION SEE:

FLUENTPY - THE FLUENT PYTHON LIBRARY

Fluentpy provides fluent interfaces to existing APIs such as the standard library, allowing you to use them in an object oriented and fluent style.

Fluentpy is inspired by JavaScript's `jQuery` and `underscore` / `lodash` and takes some inspiration from the collections API in Ruby and SmallTalk.

Please note: **This library is based on an wrapper**, that returns another wrapper for any operation you perform on a wrapped value. See the section **Caveats** below for details.

See Fowler, Wikipedia for definitions of fluent interfaces.

1.1 Motivation: Why use `fluentpy`?

Many of the most useful standard library methods such as `map`, `zip`, `filter` and `join` are either free functions or available on the wrong type or module. This prevents fluent method chaining.

Let's consider this example:

```
>>> list(map(str.upper, sorted("ba,dc".split(","), reverse=True)))
['DC', 'BA']
```

To understand this code, I have to start in the middle at `"ba,dc".split(",")`, then backtrack to `sorted(...,reverse=True)`, then to `list(map(str.upper, ...))`. All the while making sure that the parentheses all match up.

Wouldn't it be nice if we could think and write code in the same order? Something like how you would write this in other languages?

```
>>> _("ba,dc").split(",").sorted(reverse=True).map(str.upper)._
['DC', 'BA']
```

"Why no, but python has list comprehensions for that", you might say? Let's see:

```
>>> [each.upper() for each in sorted("ba,dc".split(","), reverse=True)]
['DC', 'BA']
```

This is clearly better: To read it, I have to skip back and forth less. It still leaves room for improvement though. Also, adding filtering to list comprehensions doesn't help:

```
>>> [each.upper() for each in sorted("ba,dc".split(","), reverse=True) if each.
→upper().startswith('D')]
['DC']
```

The backtracking problem persists. Additionally, if the filtering has to be done on the processed version (on `each.upper().startswith()`), then the operation has to be applied twice - which sucks because you write it twice and compute it twice.

The solution? Nest them!

```
>>> [each for each in
        (inner.upper() for inner in sorted("ba,dc".split(","), reverse=True))
        if each.startswith('D')]
['DC']
```

Which gets us back to all the initial problems with nested statements and manually having to check closing parentheses.

Compare it to this:

```
>>> processed = []
>>> parts = "ba,dc".split(",")
>>> for item in sorted(parts, reverse=True):
>>>     uppercases = item.upper()
>>>     if uppercased.startswith('D')
>>>         processed.append(uppercased)
```

With basic Python, this is as close as it gets for code to read in execution order. So that is usually what I end up doing.

But it has a huge drawback: It's not an expression - it's a bunch of statements. That makes it hard to combine and abstract over it with higher order methods or generators. To write it you are forced to invent names for intermediate variables that serve no documentation purpose, but force you to remember them while reading.

Plus (drumroll): parsing this still requires some backtracking and especially build up of mental state to read.

Oh well.

So let's return to this:

```
>>> (
    _("ba,dc")
    .split(",")
    .sorted(reverse=True)
    .map(str.upper)
    .filter(_.each.startswith('D')._)
    ._
)
('DC',)
```

Sure you are not used to this at first, but consider the advantages. The intermediate variable names are abstracted away - the data flows through the methods completely naturally. No jumping back and forth to parse this at all. It just reads and writes exactly in the order it is computed. As a bonus, there's no parentheses stack to keep track of. And it is shorter too!

So what is the essence of all of this?

Python is an object oriented language - but it doesn't really use what object orientation has taught us about how we can work with collections and higher order methods in the languages that came before it (I think of SmallTalk here, but more recently also Ruby). Why can't I make those beautiful fluent call chains that SmallTalk could do 30 years ago in Python?

Well, now I can and you can too.

1.2 Features

1.2.1 Importing the library

It is recommended to rename the library on import:

```
>>> import fluentpy as _
```

or

```
>>> import fluentpy as _f
```

I prefer _ for small projects and _f for larger projects where gettext is used.

1.2.2 Super simple fluent chains

_ is actually the function wrap in the fluentpy module, which is a factory function that returns a subclass of
Wrapper(). This is the basic and main object of this library.

This does two things: First it ensures that every attribute access, item access or method call off of the wrapped object
will also return a wrapped object. This means, once you wrap something, unless you unwrap it explicitly via ._ or
.unwrap or .to(a_type) it stays wrapped - pretty much no matter what you do with it. The second thing this
does is that it returns a subclass of Wrapper that has a specialized set of methods, depending on the type of what is
wrapped. I envision this to expand in the future, but right now the most useful wrappers are: IterableWrapper,
where we add all the Python collection functions (map, filter, zip, reduce, . . .), as well as a good batch of methods from
itertools and a few extras for good measure. CallableWrapper, where we add .curry() and .compose()
and TextWrapper, where most of the regex methods are added.

Some examples:

```
# View documentation on a symbol without having to wrap the whole line it in
↪parantheses
>>> _([]).append.help()
Help on built-in function append:

append(object, /) method of builtins.list instance
    Append object to the end of the list.

# Introspect objects without awkward wrapping stuff in parantheses
>>> _(_).dir()
fluentpy.wrap(['CallableWrapper', 'EachWrapper', 'IterableWrapper', 'MappingWrapper',
↪'ModuleWrapper', 'SetWrapper', 'TextWrapper', 'Wrapper',
'_', '_0', '_1', '_2', '_3', '_4', '_5', '_6', '_7', '_8', '_9',
...
, '_args', 'each', 'lib', 'module', 'wrap'])
>>> _(_).IterableWrapper.dir()
fluentpy.wrap(['_',
...,
'accumulate', 'all', 'any', 'call', 'combinations', 'combinations_with_replacement',
↪'delattr',
'dir', 'dropwhile', 'each', 'enumerate', 'filter', 'filterfalse', 'flatten', 'freeze',
↪ 'get',
'getattr', 'groupby', 'grouped', 'hasattr', 'help', 'iaccumulate', 'icombinations', '
icombinations_with_replacement', 'icycle', 'idropwhile', 'ieach', 'ienumerate',
↪'ifilter',
```

(continues on next page)

```
'ifilterfalse', 'iflatten', 'igroupby', 'igrouped', 'imap', 'ipermutations', 'iproduct
→', 'ireshape',
'ireversed', 'isinstance', 'islice', 'isorted', 'issubclass', 'istar_map', 'istarmap',
→ 'itee',
'iter', 'izip', 'join', 'len', 'map', 'max', 'min', 'permutations', 'pprint',
→'previous', 'print',
'product', 'proxy', 'reduce', 'repr', 'reshape', 'reversed', 'self', 'setattr', 'slice
→', 'sorted',
'star_call', 'star_map', 'starmap', 'str', 'sum', 'to', 'type', 'unwrap', 'vars', 'zip
→'])

# Did I mention that I hate wrapping everything in parantheses?
>>> _([1,2,3]).len()
3
>>> _([1,2,3]).print()
[1,2,3]

# map over iterables and easily curry functions to adapt their signatures
>>> _(range(3)).map(_(dict).curry(id=_, delay=0)._)._
({'id': 0, 'delay': 0}, {'id': 1, 'delay': 0}, {'id': 2, 'delay': 0})
>>> _(range(10)).map(_.each * 3).filter(_.each < 10)._
(0, 3, 6, 9)
>>> _([3,2,1]).sorted().filter(_.each<=2)._
[1,2]

# Directly work with regex methods on strings
>>> _("foo,  bar,     baz").split(r",\s*")._
['foo', 'bar', 'baz']
>>> _("foo,  bar,     baz").findall(r'\w{3}')._
['foo', 'bar', 'baz']

# Embedd your own functions into call chains
>>> seen = set()
>>> def havent_seen(number):
...     if number in seen:
...         return False
...     seen.add(number)
...     return True
>>> (
...     _([1,3,1,3,4,5,4])
...     .dropwhile(havent_seen)
...     .print()
... )
(1, 3, 4, 5, 4)
```

And much more. Explore the method documentation for what you can do.

1.2.3 Imports as expressions

Import statements are (ahem) statements in Python. This is fine, but can be really annoying at times.

The `_.lib` object, which is a wrapper around the Python import machinery, allows to import anything that is accessible by import to be imported as an expression for inline use.

So instead of

```python
>>> import sys
>>> input = sys.stdin.read()
```

You can do

```python
>>> lines = _.lib.sys.stdin.readlines()._
```

As a bonus, everything imported via lib is already pre-wrapped, so you can chain off of it immediately.

1.2.4 Generating lambdas from expressions

`lambda` is great - it's often exactly what the doctor ordered. But it can also be annoying if you have to write it down every time you just want to get an attribute or call a method on every object in a collection. For Example:

```python
>>> _([{'fnord':'foo'}, {'fnord':'bar'}]).map(lambda each: each['fnord'])._
('foo', 'bar')

>>> class Foo(object):
>>>     attr = 'attrvalue'
>>>     def method(self, arg): return 'method+'+arg
>>> _([Foo(), Foo()]).map(lambda each: each.attr)._
('attrvalue', 'attrvalue')

>>> _([Foo(), Foo()]).map(lambda each: each.method('arg'))._
('method+arg', 'method+arg')
```

Sure it works, but wouldn't it be nice if we could save a variable and do this a bit shorter?

Python does have `attrgetter`, `itemgetter` and `methodcaller` - they are just a bit inconvenient to use:

```python
>>> from operator import itemgetter, attrgetter, methodcaller
>>> __([{'fnord':'foo'}, {'fnord':'bar'}]).map(itemgetter('fnord'))._
('foo', 'bar')
>>> _([Foo(), Foo()]).map(attrgetter('attr'))._
('attrvalue', 'attrvalue')

>>> _([Foo(), Foo()]).map(methodcaller('method', 'arg'))._
('method+arg', 'method+arg')

_([Foo(), Foo()]).map(methodcaller('method', 'arg')).map(str.upper)._
('METHOD+ARG', 'METHOD+ARG')
```

To ease this, `_.each` is provided. `each` exposes a bit of syntactic sugar for these (and the other operators). Basically, everything you do to `_.each` it will record and later 'play back' when you generate a callable from it by either unwrapping it, or applying an operator like `` `+ - * / <` ``, which automatically call unwrap.

```python
>>>  _([1,2,3]).map(_.each + 3)._
(4, 5, 6)
```

(continues on next page)

```
>>> _([1,2,3]).filter(_.each < 3)._
(1, 2)

>>> _([1,2,3]).map(- _.each)._
(-1, -2, -3)

>>> _([dict(fnord='foo'), dict(fnord='bar')]).map(_.each['fnord']._)._
('foo', 'bar')

>>> _([Foo(), Foo()]).map(_.each.attr._)._
('attrvalue', 'attrvalue')

>>> _([Foo(), Foo()]).map(_.each.method('arg')._)._
('method+arg', 'method+arg')

>>> _([Foo(), Foo()]).map(_.each.method('arg').upper()._)._
('METHOD+ARG', 'METHOD+ARG')
# Note that there is no second map needed to call `.upper()` here!
```

The rule is that you have to unwrap `._` the each object to generate a callable that you can then hand off to `.map()`, `.filter()` or wherever you would like to use it.

1.2.5 Chaining off of methods that return None

A major nuisance for using fluent interfaces are methods that return None. Sadly, many methods in Python return None, if they mostly exhibit a side effect on the object. Consider for example `list.sort()`. But also all methods that don't have a `return` statement return None. While this is way better than e.g. Ruby where that will just return the value of the last expression - which means objects constantly leak internals - it is very annoying if you want to chain off of one of these method calls.

Fear not though, Fluentpy has you covered. :)

Fluent wrapped objects will have a `self` property, that allows you to continue chaining off of the previous 'self' object.

```
>>> _([3,2,1]).sort().self.reverse().self.call(print)
```

Even though both `sort()` and `reverse()` return None.

Of course, if you unwrap at any point with `.unwrap` or `._` you will get the true return value of None.

1.2.6 Easy Shell Filtering with Python

It could often be super easy to achieve something on the shell, with a bit of Python. But, the backtracking (while writing) as well as the tendency of Python commands to span many lines (imports, function definitions, . . .), makes this often just impractical enough that you won't do it.

That's why `fluentpy` is an executable module, so that you can use it on the shell like this:

```
$ echo 'HELLO, WORLD!' \
    | python3 -m fluentpy "lib.sys.stdin.readlines().map(str.lower).map(print)"
hello, world!
```

In this mode, the variables `lib`, `_` and `each` are injected into the namespace of of the `python` commands given as the first positional argument.

Consider this shell text filter, that I used to extract data from my beloved but sadly pretty legacy del.icio.us account. The format looks like this:

```
$ tail -n 200 delicious.html|head
<DT><A HREF="http://intensedebate.com/" ADD_DATE="1234043688" PRIVATE="0" TAGS="web2.
→0,threaded,comments,plugin">IntenseDebate comments enhance and encourage␣
→conversation on your blog or website</A>
<DD>Comments on static websites
<DT><A HREF="http://code.google.com/intl/de/apis/socialgraph/" ADD_DATE="1234003285"␣
→PRIVATE="0" TAGS="api,foaf,xfn,social,web">Social Graph API - Google Code</A>
<DD>API to try to find metadata about who is a friend of who.
<DT><A HREF="http://twit.tv/floss39" ADD_DATE="1233788863" PRIVATE="0" TAGS="podcast,
→sun,opensource,philosophy,floss">The TWiT Netcast Network with Leo Laporte</A>
<DD>Podcast about how SUN sees the society evolve from a hub and spoke to a mesh␣
→society and how SUN thinks it can provide value and profit from that.
<DT><A HREF="http://www.xmind.net/" ADD_DATE="1233643908" PRIVATE="0" TAGS=
→"mindmapping,web2.0,opensource">XMind - Social Brainstorming and Mind Mapping</A>
<DT><A HREF="http://fun.drno.de/pics/What.jpg" ADD_DATE="1233505198" PRIVATE="0" TAGS=
→"funny,filetype:jpg,media:image">What.jpg 480×640 pixels</A>
<DT><A HREF="http://fun.drno.de/pics/english/What_happens_to_your_body_if_you_stop_
→smoking_right_now.gif" ADD_DATE="1233504659" PRIVATE="0" TAGS="smoking,stop,funny,
→informative,filetype:gif,media:image">What_happens_to_your_body_if_you_stop_smoking_
→right_now.gif 800×591 pixels</A>
<DT><A HREF="http://www.normanfinkelstein.com/article.php?pg=11&ar=2510" ADD_DATE=
→"1233482064" PRIVATE="0" TAGS="propaganda,israel,nazi">Norman G. Finkelstein</A>

$ cat delicious.html | grep hosting \                                              ␣
→                               :(
   | python3  -c 'import sys,re; \
       print("\n".join(re.findall(r"HREF=\"([^\"]+)\"", sys.stdin.read())))'
https://uberspace.de/
https://gitlab.com/gitlab-org/gitlab-ce
https://www.docker.io/
```

Sure it works, but with all the backtracking problems I talked about already. Using `fluentpy` this could be much nicer to write and read:

```
 $ cat delicious.html | grep hosting \
     | python3 -m fluentpy 'lib.sys.stdin.read().findall(r"HREF=\"([^\"]+)\"").
→map(print)'
https://uberspace.de/
https://gitlab.com/gitlab-org/gitlab-ce
https://www.docker.io/
```

1.3 Caveats and lessons learned

1.3.1 Start and end Fluentpy expressions on each line

If you do not end each fluent statement with a `._`, `.unwrap` or `.to(a_type)` operation to get a normal Python object back, the wrapper will spread in your runtime image like a virus, 'infecting' more and more objects causing strange side effects. So remember: Always religiously unwrap your objects at the end of a fluent statement, when using `fluentpy` in bigger projects.

```
>>> _('foo').uppercase().match('(foo)').group(0)._
```

It is usually a bad idea to commit wrapped objects to variables. Just unwrap instead. This is especially sensible, since fluent chains have references to all intermediate values, so unwrapping chains give the garbage collector the permission to release all those objects.

Forgetting to unwrap an expression generated from `_.each` may be a bit surprising, as every call on them just causes more expression generation instead of triggering their effect.

That being said, `str()` and `repr()` output of fluent wrapped objects is clearly marked, so this is easy to debug.

Also, not having to unwrap may be perfect for short scripts and especially 'one-off' shell commands. However: Use Fluentpys power wisely!

1.3.2 Split expression chains into multiple lines

Longer fluent call chains are best written on multiple lines. This helps readability and eases commenting on lines (as your code can become very terse this way).

For short chains one line might be fine.

```
_(open('day1-input.txt')).read().replace('\n','').call(eval)._
```

For longer chains multiple lines are much cleaner.

```python
day1_input = (
    _(open('day1-input.txt'))
    .readlines()
    .imap(eval)
    ._
)

seen = set()
def havent_seen(number):
    if number in seen:
        return False
    seen.add(number)
    return True

(
    _(day1_input)
    .icycle()
    .iaccumulate()
    .idropwhile(havent_seen)
    .get(0)
    .print()
)
```

1.3.3 Consider the performance implications of Fluentpy

This library works by creating another instance of its wrapper object for every attribute access, item get or method call you make on an object. Also those objects retain a history chain to all previous wrappers in the chain (to cope with functions that return `None`).

This means that in tight inner loops, where even allocating one more object would harshly impact the performance of your code, you probably don't want to use `fluentpy`.

Also (again) this means that you don't want to commit fluent objects to long lived variables, as that could be the source of a major memory leak.

And for everywhere else: go to town! Coding Python in a fluent way can be so much fun!

1.4 Famous Last Words

This library tries to do a little of what libraries like `underscore` or `lodash` or `jQuery` do for Javascript. Just provide the missing glue to make the standard library nicer and easier to use. Have fun!

I envision this library to be especially useful in short Python scripts and shell one liners or shell filters, where Python was previously just that little bit too hard to use and prevented you from doing so.

I also really like its use in notebooks or in a python shell to smoothly explore some library, code or concept.

FLUENTPY

To use this module just import it with a short custom name. I recommend:

```
>>> import fluentpy as _ # for scripts / projects that don't use gettext
>>> import fluentpy as _f # for everything else
```

If you want / need this to be less magical, you can import the main wrapper normally

```
>>> from fluentpy import wrap # or `_`, if you're not using gettext
```

Then to use the module, wrap any value and start chaining off of it. To get started lets try to inrospect *fluentpy* using its own fluent interface:

```
$ python3 -m fluentpy '_(_).dir().print()'
$ python3 -m fluentpy '_(_).help()'
```

This is incidentally the second way to use this module, as a helper that makes it easier to write short shell filters quickly in python.:

```
$ echo "foo\nbar\nbaz" \
    | python3 -m fluentpy "lib.sys.stdin.readlines().map(each.call.upper()).map(print)
↪"
```

Try to rewrite that in classical python (as a one line shell filter) and see which version spells out what happens in which order more clearly.

For further documentation and development see this documentation or the source at https://github.com/dwt/fluent

- *Functions*
- *Classes*
- *Variables*

2.1 Functions

- `wrap()`: Factory method, wraps anything and returns the appropriate `Wrapper` subclass.

fluentpy.**wrap**(*wrapped*, *, *previous=None*)
>   Factory method, wraps anything and returns the appropriate `Wrapper` subclass.
>
>   This is the main entry point into the fluent wonderland. Wrap something and everything you call off of that will stay wrapped in the appropriate wrappers.
>
>   It is usually imported in one of the following ways:

```
>>> import fluentpy as _
>>> import fluentpy as _f
>>> from fluentpy import wrap
```

>   `wrap` is the original name of the function, though I rarely recommend to use it by this name.

2.2 Classes

- `Wrapper`: Universal wrapper.

- `ModuleWrapper`: The `Wrapper` for modules transforms attribute accesses into pre-wrapped imports of sub-modules.

- `CallableWrapper`: The `Wrapper` for callables adds higher order methods.

- `IterableWrapper`: The `Wrapper` for iterables adds iterator methods to any iterable.

- `MappingWrapper`: The `Wrapper` for mappings allows indexing into mappings via attribute access.

- `SetWrapper`: The `Wrapper` for sets is mostly like `IterableWrapper`.

- `TextWrapper`: The `Wrapper` for str adds regex convenience methods.

- `EachWrapper`: The `Wrapper` for expressions (see documentation for `each`).

class fluentpy.**Wrapper**(*wrapped*, *, *previous*)
>   Universal wrapper.
>
>   This class ensures that all function calls and attribute accesses (apart from such special CPython runtime accesses like `object.__getattribute__`, which cannot be intercepted) will be wrapped with the wrapper again. This ensures that the fluent interface will persist and everything that is returned is itself able to be chained from again.
>
>   All returned objects will be wrapped by this class or one of its sub classes, which add functionality depending on the type of the wrapped object. I.e. iterables will gain the collection interface, mappings will gain the mapping interface, strings will gain the string interface, etc.
>
>   If you want to access the actual wrapped object, you will have to unwrap it explicitly using `.unwrap` or `._`
>
>   Please note: Since most of the methods on these objects are actual standard library methods that are simply wrapped to rebind their (usually first) parameter to the object they where called on. So for example: `repr(something)` becomes `_(something).repr()`. This means that the (unchanged) documentation (often) still shows the original signature and refers to the original arguments. A little bit of common sense might therefore be required.

Inheritance

Wrapper

property unwrap
 Returns the underlying wrapped value of this wrapper instance.

 All other functions return wrappers themselves unless explicitly stated.

 Alias: _

property previous
 Returns the previous wrapper in the chain of wrappers.

 This allows you to walk the chain of wrappers that where created in your expression. Mainly used internally but might be useful for introspection.

property self
 Returns the previous wrapped object. This is especially usefull for APIs that return None.

 For example `_([1,3,2]).sort().self.print()` will print the sorted list, even though `sort()` did return `None`.

 This is simpler than using .previous as there are often multiple wrappers involved where you might expect only one. E.g. `_([2,1]).sort().self._ == [1,2]` but `_([2,1]).sort().previous._` will return the function `list.sort()` as the attrget and call are two steps of the call chain.

 This eases chaining using APIs that where not designed with chaining in mind. (Inspired by SmallTalk's default behaviour)

property proxy
 Allow access to shadowed attributes.

 Breakout that allows access to attributes of the wrapped object that are shadowed by methods on the various wrapper classes. Wrapped of course.

 ```
 >>> class UnfortunateNames(object):
 >>>     def previous(self, *args):
 >>>         return args
 ```

 This raises TypeError, because Wrapper.previous() shadows UnfortunateNames.previous():

 ```
 >>> _(UnfortunateNames()).previous('foo'))
 ```

 This works as expected:

 ```
 >>> _(UnfortunateNames()).proxy.previous('foo')._) == ('foo',)
 ```

call (*function*, **args*, ***kwargs*)
 Call function with self as its first argument.

```
>>> _('foo').call(list)._ == list('foo')
>>> _('fnord').call(textwrap.indent, prefix='  ')._ == textwrap.indent('fnord
↪', prefix='  ')
```

Call is mostly usefull to insert normal functions that express some algorithm into the call chain. For example like this:

```
>>> seen = set()
>>> def havent_seen(number):
...     if number in seen:
...         return False
...     seen.add(number)
...     return True
>>> (
...     _([1,3,1,3,4,5,4])
...     .dropwhile(havent_seen)
...     .print()
... )
```

Less obvious, it can also be used as a decorator, however the result can be confusing, so maybe not as recomended:

```
>>> numbers = _(range(5))
>>> @numbers.call
... def items(numbers):
...     for it in numbers:
...         yield it
...         yield it
>>> items.call(list).print()
```

Note the difference from `.__call__()`. This applies `function(self, ...)` instead of `self(...)`.

to (*function*, **args*, ***kwargs*)
    Like .call() but returns an unwrapped result.

    This makes `.to()` really convenient to terminate a call chain by converting to a type that perhaps itself con.

setattr (*name*, *value*, */*)
    Sets the named attribute on the given object to the specified value.

    setattr(x, 'y', v) is equivalent to ``x.y = v''

getattr (*object*, *name*[, *default*]) → value
    Get a named attribute from an object; getattr(x, 'y') is equivalent to x.y. When a default argument is given, it is returned when the attribute doesn't exist; without it, an exception is raised in that case.

hasattr (*name*, */*)
    Return whether the object has an attribute with the given name.

    This is done by calling getattr(obj, name) and catching AttributeError.

delattr (*name*, */*)
    Deletes the named attribute from the given object.

    delattr(x, 'y') is equivalent to ``del x.y''

isinstance (*class_or_tuple*, */*)
    Return whether an object is an instance of a class or of a subclass thereof.

A tuple, as in isinstance(x, (A, B, ...)), may be given as the target to check against. This is equivalent to isinstance(x, A) or isinstance(x, B) or ... etc.

issubclass(*class_or_tuple*, */*)

> Return whether 'cls' is a derived from another class or is the same class.
>
> A tuple, as in issubclass(x, (A, B, ...)), may be given as the target to check against. This is equivalent to issubclass(x, A) or issubclass(x, B) or ... etc.

dir([*object*]) → list of strings

> If called without an argument, return the names in the current scope. Else, return an alphabetized list of names comprising (some of) the attributes of the given object, and of attributes reachable from it. If the object supplies a method named __dir__, it will be used; otherwise the default dir() logic is used and returns:
>
>> for a module object: the module's attributes. for a class object: its attributes, and recursively the attributes
>>
>>> of its bases.
>>
>> **for any other object: its attributes, its class's attributes, and** recursively the attributes of its
>>> class's base classes.

vars([*object*]) → dictionary

> Without arguments, equivalent to locals(). With an argument, equivalent to object.__dict__.

print(*value*, *...*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)

> Prints the values to a stream, or to sys.stdout by default. Optional keyword arguments: file: a file-like object (stream); defaults to the current sys.stdout. sep: string inserted between values, default a space. end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream.

pprint(*stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)

> Pretty-print a Python object to a stream [default is sys.stdout].

help

> Define the built-in 'help'. This is a wrapper around pydoc.help (with a twist).

type = <function type>

str = <function str>

repr()

> Return the canonical string representation of the object.
>
> For many object types, including most builtins, eval(repr(obj)) == obj.

class fluentpy.**ModuleWrapper**(*wrapped*, ***, *previous*)

> The *Wrapper* for modules transforms attribute accesses into pre-wrapped imports of sub-modules.

Inheritance

```
┌─────────┐      ┌───────────────┐
│ Wrapper │─────▶│ ModuleWrapper │
└─────────┘      └───────────────┘
```

reload()
>   Reload the module and return it.
>
>   The module must have been successfully imported before.

class fluentpy.**CallableWrapper**(*wrapped*, ***, *previous*)
>   The *Wrapper* for callables adds higher order methods.

Inheritance

```
Wrapper ─────▶ CallableWrapper
```
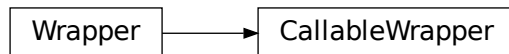
curry(**default_args*, ***default_kwargs*)
>   Like functools.partial, but with a twist.
>
>   If you use `wrap` or `_` as a positional argument, upon the actual call, arguments will be left-filled for those placeholders.
>
>    ```
>    >>> _(operator.add).curry(_, 'foo')('bar')._ == 'barfoo'
>    ```
>
>   If you use wrap._$NUMBER (with $NUMBER < 10) you can take full control over the ordering of the arguments.
>
>    ```
>    >>> _(a_function).curry(_._0, _._0, _.7)
>    ```
>
>   This will repeat the first argument twice, then take the 8th and ignore all in between.
>
>   You can also mix numbered with generic placeholders, but since it can be hard to read, I would not advise it.
>
>   There is also `_._args` which is the placeholder for the `*args` variable argument list specifier. (Note that it is only supported in the last position of the positional argument list.)
>
>    ```
>    >>> _(lambda x, y=3: x).curry(_._args)(1, 2)._ == (1, 2)
>    >>> _(lambda x, y=3: x).curry(x=_._args)(1, 2)._ == (1, 2)
>    ```

compose(*outer*)
>   Compose two functions.
>
>    ```
>    >>>  inner_function.compose(outer_function) \
>    ...     == lambda *args, **kwargs: outer_function(inner_function(*args,
>    ↪**kwargs))
>    ```

class fluentpy.**IterableWrapper**(*wrapped*, ***, *previous*)
>   The *Wrapper* for iterables adds iterator methods to any iterable.

Most iterators in Python 3 return an iterator by default, which is very interesting if you want to build efficient processing pipelines, but not so hot for quick and dirty scripts where you have to wrap the result in a list() or tuple() all the time to actually get at the results (e.g. to print them) or to actually trigger the computation pipeline.

Thus all iterators on this class are by default immediate, i.e. they don't return the iterator but instead consume it immediately and return a tuple. Of course if needed, there is also an i{map,zip,enumerate,...} version for your enjoyment that returns the iterator.

Iterating over wrapped iterators yields unwrapped elements by design. This is neccessary to make `fluentpy` interoperable with the standard library. This means you will have to rewrap occasionally in handwritten iterator methods or when iterating over a wrapped iterator

Where methods return infinite iterators, the non i-prefixed method name is skipped. See `icycle` for an an example.

Inheritance

```
┌─────────┐      ┌──────────────────┐
│ Wrapper │ ───▶ │ IterableWrapper  │
└─────────┘      └──────────────────┘
```
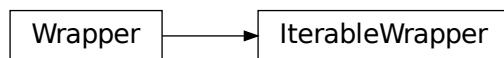
iter(*iterable*) → iterator
   iter(callable, sentinel) -> iterator

   Get an iterator from an object. In the first form, the argument must supply its own iterator, or be a sequence. In the second form, the callable is called until it returns the sentinel.

star_call(*function*, **args*, ***kwargs*)
   Calls `function(*self)`, but allows to prepend args and add kwargs.

get(*target_index*, *default=<object object>*)
   Like \`dict.get()\` but for IterableWrappers and able to deal with generators

join(*with_what=''*)
   Like str.join, but the other way around. Bohoo!

   Also calls str on all elements of the collection before handing it off to str.join as a convenience.

freeze = <function tuple>

len()
   Just like len(), but also works for iterators.

   Beware, that it has to consume the iterator to compute its length

max(*iterable*, ***[, *default=obj*, *key=func*]) → value
   max(arg1, arg2, *args, *[, key=func]) -> value

   With a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if the provided iterable is empty. With two or more arguments, return the largest argument.

min(*iterable*, ***[, *default=obj*, *key=func*]) → value
   min(arg1, arg2, *args, *[, key=func]) -> value

   With a single iterable argument, return its smallest item. The default keyword-only argument specifies an object to return if the provided iterable is empty. With two or more arguments, return the smallest argument.

sum(*start=0, /*)
> Return the sum of a 'start' value (default: 0) plus an iterable of numbers

> When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

any()
> Return True if bool(x) is True for any x in the iterable.

> If the iterable is empty, return False.

all()
> Return True if bool(x) is True for all values x in the iterable.

> If the iterable is empty, return True.

reduce(*function, sequence*[, *initial*]) → value
> Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

ieach(*a_function*)
> call `a_function` on each elment in self purely for the side effect, then yield the input element.

each(*a_function*)
> call `a_function` on each elment in self purely for the side effect, then yield the input element.

imap = <function map>

map = <function map>

istar_map = <function starmap>

istarmap = <function starmap>

star_map = <function starmap>

starmap = <function starmap>

ifilter = <function filter>

filter = <function filter>

ienumerate = <function enumerate>

enumerate = <function enumerate>

ireversed = <function reversed>

reversed = <function reversed>

isorted(**, key=None, reverse=False*)
> Return a new list containing all items from the iterable in ascending order.

> A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

sorted(**, key=None, reverse=False*)
> Return a new list containing all items from the iterable in ascending order.

> A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

igrouped(*group_length*)
> Cut self into tupels of length group_length s -> (s0,s1,s2,...sn-1), (sn,sn+1,sn+2,...s2n-1), (s2n,s2n+1,s2n+2,...s3n-1), ...

grouped(*group_length*)
> Cut self into tupels of length group_length s -> (s0,s1,s2,...sn-1), (sn,sn+1,sn+2,...s2n-1), (s2n,s2n+1,s2n+2,...s3n-1), ...

izip = <function zip>

zip = <function zip>

iflatten(*level=inf, stop_at_types=(<class 'str'>, <class 'bytes'>)*)
> Modeled after rubys array.flatten @see http://ruby-doc.org/core-1.9.3/Array.html#method-i-flatten

> Calling flatten on string likes would lead to infinity recursion, thus @arg stop_at_types. If you want to flatten those, use a combination of @arg level and @arg stop_at_types.

flatten(*level=inf, stop_at_types=(<class 'str'>, <class 'bytes'>)*)
> Modeled after rubys array.flatten @see http://ruby-doc.org/core-1.9.3/Array.html#method-i-flatten

> Calling flatten on string likes would lead to infinity recursion, thus @arg stop_at_types. If you want to flatten those, use a combination of @arg level and @arg stop_at_types.

ireshape(**spec*)
> Creates structure from linearity.

> Allows you to turn `(1,2,3,4)` into `((1,2),(3,4))`. Very much inspired by numpy.reshape. @see https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html

> @argument spec integer of tuple of integers that give the spec for the dimensions of the returned structure. The last dimension is inferred as needed. For example:

```
>>> _([1,2,3,4]).reshape(2)._ == ((1,2),(3,4))
```

> Please note that

```
>>> _([1,2,3,4]).reshape(2,2)._ == (((1,2),(3,4)),)
```

> The extra tuple around this is due to the specification being, two tuples of two elements which is possible exactly once with the given iterable.

> This iterator will *not* ensure that the shape you give it will generate fully 'rectangular'. This means that the last element in the generated sequnce the number of elements can be different! This tradeoff is made, so it works with infinite sequences.

reshape(**spec*)
> Creates structure from linearity.

> Allows you to turn `(1,2,3,4)` into `((1,2),(3,4))`. Very much inspired by numpy.reshape. @see https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html

> @argument spec integer of tuple of integers that give the spec for the dimensions of the returned structure. The last dimension is inferred as needed. For example:

```
>>> _([1,2,3,4]).reshape(2)._ == ((1,2),(3,4))
```

> Please note that

```
>>> _([1,2,3,4]).reshape(2,2)._ == (((1,2),(3,4)),)
```

The extra tuple around this is due to the specification being, two tuples of two elements which is possible exactly once with the given iterable.

This iterator will *not* ensure that the shape you give it will generate fully 'rectangular'. This means that the last element in the generated sequnce the number of elements can be different! This tradeoff is made, so it works with infinite sequences.

igroupby = <function groupby>

groupby(**args*, ***kwargs*)
> See igroupby for most of the docs.

> Correctly consuming an itertools.groupby is surprisingly hard, thus this non tuple returning version that does it correctly.

itee()
> tee(iterable, n=2) –> tuple of n independent iterators.

islice = <function islice>

slice = <function islice>

icycle = <function cycle>

iaccumulate = <function accumulate>

accumulate = <function accumulate>

idropwhile = <function dropwhile>

dropwhile = <function dropwhile>

ifilterfalse = <function filterfalse>

filterfalse = <function filterfalse>

ipermutations = <function permutations>

permutations = <function permutations>

icombinations = <function combinations>

combinations = <function combinations>

icombinations_with_replacement = <function combinations_with_replacement>

combinations_with_replacement = <function combinations_with_replacement>

iproduct = <function product>

product = <function product>

class fluentpy.**MappingWrapper**(*wrapped*, ***, *previous*)
> The `Wrapper` for mappings allows indexing into mappings via attribute access.

> Indexing into dicts like objects. As JavaScript can.

```
>>> _({ 'foo': 'bar: }).foo == 'bar
```

Inheritance
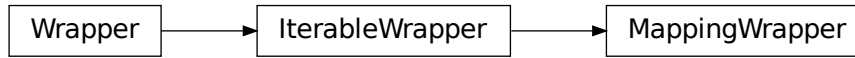
star_call (*function*, **args*, ***kwargs*)
  Calls function(**self), but allows to add args and set defaults for kwargs.

class fluentpy.**SetWrapper** (*wrapped*, ***, *previous*)
  The *Wrapper* for sets is mostly like *IterableWrapper*.

  Mostly like IterableWrapper

Inheritance
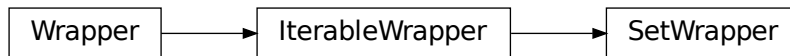
freeze = <function frozenset>

class fluentpy.**TextWrapper** (*wrapped*, ***, *previous*)
  The *Wrapper* for str adds regex convenience methods.

  Supports most of the regex methods as if they where native str methods

Inheritance
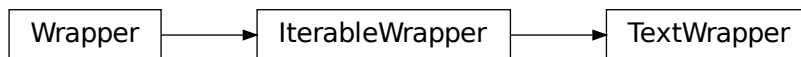
search (*string*, *flags=0*)
  Scan through string looking for a match to the pattern, returning a match object, or None if no match was found.

match (*string*, *flags=0*)
  Try to apply the pattern at the start of the string, returning a match object, or None if no match was found.

fullmatch (*string*, *flags=0*)
Try to apply the pattern at the start of the string, returning a match object, or None if no match was found.

split (*string*, *maxsplit=0*, *flags=0*)
Split the source string by the occurrences of the pattern, returning a list containing the resulting substrings. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.

findall (*string*, *flags=0*)
Return a list of all non-overlapping matches in the string.

If one or more capturing groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group.

Empty matches are included in the result.

finditer (*string*, *flags=0*)
Return an iterator over all non-overlapping matches in the string. For each match, the iterator returns a match object.

Empty matches are included in the result.

sub (*repl*, *string*, *count=0*, *flags=0*)
Return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl. repl can be either a string or a callable; if a string, backslash escapes in it are processed. If it is a callable, it's passed the match object and must return a replacement string to be used.

subn (*repl*, *string*, *count=0*, *flags=0*)
Return a 2-tuple containing (new_string, number). new_string is the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in the source string by the replacement repl. number is the number of substitutions that were made. repl can be either a string or a callable; if a string, backslash escapes in it are processed. If it is a callable, it's passed the match object and must return a replacement string to be used.

class fluentpy.**EachWrapper** (*operation*, *name*)
The *Wrapper* for expressions (see documentation for *each*).

Inheritance

```
EachWrapper
```

in_ (*haystack*)
Implements a method version of the `in` operator.

So `_.each.in_('bar')` is roughly equivalent to `lambda each:  each in 'bar'`

not_in (*haystack*)
Implements a method version of the `not in` operator.

So `_.each.not_in('bar')` is roughly equivalent to `lambda each:  each not in 'bar'`

2.3 Variables

- *lib*
- *each*
- *_0*
- *_1*
- *_2*
- *_3*
- *_4*
- *_5*
- *_6*
- *_7*
- *_8*
- *_9*
- *_args*

fluentpy.**lib**

    Imports as expressions. Already pre-wrapped.

    All attribute accesses to instances of this class are converted to an import statement, but as an expression that returns the wrapped imported object.

    Example:

```
>>> lib.sys.stdin.read().map(print)
```

    Is equivalent to

```
>>> import sys
>>> wrap(sys).stdin.read().map(print)
```

    But of course without creating the intermediate symbol 'stdin' in the current namespace.

    All objects returned from lib are pre-wrapped, so you can chain off of them immediately.

```
fluentpy.wrap('virtual root module')
```

fluentpy.**each**

    Create functions from expressions.

    Use each.foo._ to create attrgetters, each['foo']._ to create itemgetters, each.foo()._ to create methodcallers or each == 'foo' (with pretty much any operator) to create callable operators.

    Many operations can be chained, to extract a deeper object by iterating a container. E.g.:

```
>>> each.foo.bar('baz')['quoox']._
```

    creates a callable that will first get the attribute foo, then call the method bar('baz') on the result and finally applies gets the item 'quoox' from the result. For this reason, all callables need to be unwrapped before they are used, as the actual application would just continue to build up the chain on the original callable otherwise.

Apply an operator like each < 3 to generate a callable that applies that operator. Different to all other cases, applying any binary operator auto terminates the expression generator, so no unwrapping is neccessary.

Note: The generated functions never wrap their arguments or return values.

```
fluentpy.wrap(each)
```

fluentpy.**_0**
> Placeholder for *CallableWrapper.curry()* to access argument at index 0.

```
fluentpy.wrap(0)
```

fluentpy.**_1**
> Placeholder for *CallableWrapper.curry()* to access argument at index 1.

```
fluentpy.wrap(1)
```

fluentpy.**_2**
> Placeholder for *CallableWrapper.curry()* to access argument at index 2.

```
fluentpy.wrap(2)
```

fluentpy.**_3**
> Placeholder for *CallableWrapper.curry()* to access argument at index 3.

```
fluentpy.wrap(3)
```

fluentpy.**_4**
> Placeholder for *CallableWrapper.curry()* to access argument at index 4.

```
fluentpy.wrap(4)
```

fluentpy.**_5**
> Placeholder for *CallableWrapper.curry()* to access argument at index 5.

```
fluentpy.wrap(5)
```

fluentpy.**_6**
> Placeholder for *CallableWrapper.curry()* to access argument at index 6.

```
fluentpy.wrap(6)
```

fluentpy.**_7**
> Placeholder for *CallableWrapper.curry()* to access argument at index 7.

```
fluentpy.wrap(7)
```

fluentpy.**_8**
> Placeholder for *CallableWrapper.curry()* to access argument at index 8.

```
fluentpy.wrap(8)
```

fluentpy.**_9**
> Placeholder for *CallableWrapper.curry()* to access argument at index 9.

```
fluentpy.wrap(9)
```

`fluentpy.`**`_args`**

> The *Wrapper* for str adds regex convenience methods.
>
> Supports most of the regex methods as if they where native str methods

```
fluentpy.wrap('*')
```

CHANGELOG

3.1 Development version

3.2 Version 2.1

- `Wrapper.self` will now always go back in the chain to the base of the last call, instead of onyl when the last callable returned `None`. This should fix the possible behaviour change when methods sometimes return None and sometimes a usefull value.

- Fixed inconsistencies on how `CallableWrapper.curry()` deals with too many arguments. In Python this leads to a `TypeError` - and now it does here too.

- Fixed bug that `_._args` behaved differently than documented. The documentation stated that `_(lambda x, y=3: x).curry(_._args)(1, 2)._ == (1, 2)` but it did instead return `tuple(1)`

- Add `__rmul__` and friends support on `_._each` to allow expressions like `4 % _._each` to work.

3.3 Version 2.0

3.3.1 Breaking changes

- `IterableWrapper.iter()` now returns unwrapped instances. This was changed for uniformity, as all other ways to iterate through a wrapped `IterableWrapper` returned unwrapped instances.

- Removed `Wrapper.tee()` and `Itertools.tee()` as `_(something).tee(a_function)` is easily replicable by `_(something).call(a_function).previous` and `IterableWrapper.tee()` prevented me from providing `itertools.tee`.

- `Wrapper.type()` returns a wrapped type for consistency.

- `Wrapper.{tuplify,listify,dictify,setify}` have been removed, use `Wrapper.call(a_type)` for a wrapped or `Wrapper.to(a_type)` for an unwrapped result instead.

- `_.each` now supports auto chaining of operations. That means you can type `_.each.foo['bar'].baz('quoox')._` to generate a function that applies all of these operations in order. This also means that all functions generated from `_.each` need to be unwrapped (`._`) before usage!

- `_.each.call` is removed, as `_.each.method('arg')` now works as expected, so `_.each.call` is not neccessary any more.

3.3.2 Notable Changes

- `CallableWrapper.curry()` now supports converting positional arguments to keyword arguments.

- `IterableWrapper.each()` to apply a function to every element just for the side effect while chaining off of the original value.

- `EachWrapper.in_(haystack)` and `EachWrapper.not_in(haystack)` support to mimik `lambda each: each in haystack`.

- All the top level classes have been renamed to have a common `-Wrapper` suffix for consistency and debuggability.

- Added new method `.to(a_type, *args, **kwargs)` that calls `a_type(self, **args, **kwars)` but returns an unwrapped result, to more smothely terminate call chains in common scenarios.

FOUR

PROJECT MATTERS

- Project Homepage: https://github.com/dwt/fluent/
- Bugs: https://github.com/dwt/fluent/issues
- Documentation: https://fluentpy.readthedocs.io/en/latest/
- Build Server: TODO

WHAT PROBLEM DOES FLUENT SOLVE

The fluent library is a syntactic sugar library for Python. It allows you to write more things as expressions, which traditionally require statements in Python. The goal is to allow writing beautiful fluent code with the standard library or your classes, as defined at https://en.wikipedia.org/wiki/Fluent_interface.

SIX

QUICK START

Fluent is a powerful library, that allows you to use existing libraries through a fluent interface. This is especially useful since most of the Python standard library was written in a way that makes it hard to be used in this style.

This makes *fluentpy* really usefull to write small Python shell filters, to do something that Python is good at, for example finding stuff with regexes:

```
$ python3 -m fluentpy "lib.sys.stdin.read().findall(r'(foo|bar)*').print()"
```

Or whatever other function from the standard lib or any library you would like to use. The Idea here is that while this is perfectly possible without fluent, it is just that little bit easier, to make it actually become fun and practical.

In this context you have basically three extra symbols `wrap` or `_`, `lib` and `each`

`wrap` is the factory for the object specific wrapper types. Every wrapped object has the fluent behaviour, i.e. every accessed property is also wrapped, while also gaining some type dependent special methods like regex methods on str like `.findall()` `.map()`, `.join()`, etc. on list, etc.

`lib` is a wrapper that allows to use any symbol that is anywhere in the standard library (or accessible via an import) by attribute access. For Example:

```
>>> import sys
>>> sys.stdin.read()
```

becomes

```
>>> lib.sys.stdin.read()
```

`each` you probably best think as a convenience lambda generator. It is meant to be a little bit more compact to write down operations you want to execute on every element in a collection.

```
>>> print(map(lambda x: x * 2, range(1,10)))
```

becomes

```
>>> wrap(range(1, 10)).map(each * 2).print()
```

Here `each * 2` is the same as `lambda x:  x * 2`. `each['foo']` becomes `lambda each: each['foo']`, `each.bar` becomes `lambda each:  each.bar`. `each.call.foo('bar')` becomes `lambda each:  each.foo('bar')` (Sorry about the `.call.` there, but I haven't found a way to get rid of it, pull requests welcome).

I suggest you use `.dir()` and `.help()` on the objects of this library to quickly get to know what they do.

SEVEN

USAGE IN SHORT SCRIPTS OR BIGGER PROJECTS

Just import fluent under the name you would like to use it. For short scripts I prefer _ but for projects where gettext is used, I prefer _f.

```
>>> import fluent as _
>>> _(range(10)).map(_.each * 3)
```

each and lib are available as symbols on _, or you can import them directly from fluent

```
>>> from fluent import wrap as _, lib, each
>>> _(range(10)).map(each * 3)
```

EIGHT

FURTHER INFORMATION

Read up on the *Narrative Documentation*, browse the *API Documentation* or take a look at some Example Code.

And most important of all: Have phun!

PYTHON MODULE INDEX

f
fluentpy, 11